Misliti na Javi

Prevod četvrtog izdanja

Bruce Eckel
MindView, Inc., predsednik

Preveo
Miljenko Šućur

PRENTICE
HALL

Misliti na Javi, prevod četvrtog izdanja

Glavni urednik	Olga Milanko
Redaktori	Stela Spasić
	Snežana Bisenić
Tehnički urednik	Sanja Tasić
Lektor i korektor	Vesna Đukić
Realizacija korica	Nataša Pavlov
Prelom teksta i obrada slika	Sanja Tasić
	Milica Dečanski
	Nataša Pavlov
Izdavač	Mikro knjiga, Beograd
Direktor	Dragan Tanaskoski
Štampa	Publikum, Beograd

Ako imate pitanja ili komentare, ili ako želite da dobijete besplatan katalog, pišite nam ili se javite:

Mikro knjiga
P. fah 20-87
11030 Beograd
tel: 011/3540-544
pisma@mikroknjiga.rs

Autorizovan prevod sa engleskog jezika knjige Thinking in Java, Fourth Edition.

Copyright © **2007 Mikro knjiga**. Sva prava zadržana. Nije dozvoljeno da ijedan deo ove knjige bude reprodukovan ili emitovan na bilo koji način, elektronski ili mehanički, uključujući fotokopiranje, snimanje ili bilo koji drugi sistem za beleženje, bez prethodne pismene dozvole izdavača.

Translation copyright © **2007 by Mikro knjiga, Thinking in Java, Fourth Edition by Bruce Eckel**. **Copyright** © 2006, All Rights Reserved. Published by arrangement with the original publisher, Pearson Education, Inc., publishing as **Prentice Hall, Inc**, a Company.

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004.438JAVA
004.42:004.738.5

EKEJL Брус
Misliti na Javi / Bruce Eckel ; prevod 4. izd. Miljenko Sučur. – Beograd : Mikro knjiga, 2007 (Beograd : Publikum) – XIII, 1201 str. : graf. prikazi ; 21 cm

Prevod dela: Thinking in Java. – Tiraž 500. – Bibliografija: 1170-1171. – Registar.

ISBN 978-86-7555-308-3

a) Програмски језик "Java" – Интернет – Програмирање
COBISS.SR – ID 141670412

MNJ4/289/290069M19O8P101S161K76 5.4

Kratak sadržaj

Predgovor	1
Uvod	8
1: Upoznavanje sa objektima	15
2: Sve je objekat	42
3: Operatori	65
4: Kontrolisanje izvršavanja	99
5: Inicijalizacija i čišćenje	115
6: Kontrola pristupa	159
7: Ponovno korišćenje klasa	180
8: Polimorfizam	211
9: Interfejsi	239
10: Unutrašnje klase	266
11: Čuvanje objekata	302
12: Obrada grešaka pomoću izuzetaka	345
13: Znakovni nizovi	392
14: Podaci o tipu	432
15: Generički tipovi	484
16: Nizovi	593
17: Detaljno razmatranje kontejnera	628
18: Javin ulazno-izlazni sistem	719
19: Nabrojani tipovi	805
20: Anotacije	845
21: Paralelno izvršavanje	885
22: Grafička korisnička okruženja	1045
A: Dodaci	1166
B: Resursi	1170
Spisak termina korišćenih u knjizi	1175
Indeks	1177

Sadržaj

Predgovor 1

Java SE5 i SE6 2
Java SE6 2
Četvrto izdanje 3
Izmene 3
Napomena o dizajnu korica 4
Zahvalnice 5

Uvod 8

Preduslovi 8
Učenje Jave 9
Ciljevi 9
Poučavanje na osnovu ove knjige 10
Dokumentacija na Webu 11
Vežbe 11
Javini temelji 11
Izvorni kôd 12
Način pisanja korišćen u knjizi 13
Greške 14

1: Upoznavanje sa objektima 15

Razvoj apstrakcije 15
Objekat ima interfejs 17
Objekat pruža usluge 19
Skrivena realizacija 20
Ponovno korišćenje realizacije 21
Nasleđivanje 22
Relacije IS-A i IS-LIKE-A 25
Virtuelizacija objekata preko polimorfizma 26
Hijerarhija s jedinstvenim korenom 29
Kontejneri 29
Parametrizovani (generički) tipovi 30
Obrada izuzetaka: postupanje s greškama 33
Paralelni rad 34
Java i Internet 34
Šta je Web? 34
Programiranje s klijentske strane 36
Programiranje sa serverske strane 40
Sažetak 41

2: Sve je objekat 42

Rad sa objektima preko referenci 42
Morate da napravite sve objekte 43
Gde se nalazi skladište 43
Specijalan slučaj: prosti tipovi 44
Nizovi u Javi 46
Nikada ne morate da uništite objekat 46
Oblast važenja 46
Oblast važenja objekata 47
Pravljenje novih tipova podataka: **klasa** 48
Polja i metode 48
Metode, argumenti i povratne vrednosti 50
Lista argumenata 51
Pravljenje programa na Javi 52
Vidljivost imena 52
Korišćenje drugih komponenata 52
Rezervisana reč **static** 53
Vaš prvi program na Javi 54
Prevođenje i izvršavanje 56
Komentari i ugrađena dokumentacija 57
Dokumentacioni komentari 58
Sintaksa 58
Ugrađeni HTML 59
Primeri oznaka 60
Primer dokumentacije 62
Stil programiranja 63
Sažetak 63
Vežbe 63

3: Operatori 65
Jednostavnije naredbe za ispisivanje .65
Korišćenje operatora u Javi .66
Prioriteti .66
Dodela vrednosti .67
Pojava pseudonima pri pozivanju metode 68
Matematički operatori .69
Unarni operatori minus i plus 71
Automatsko uvećanje i umanjenje .72
Operatori poređenja .73
Ispitivanje jednakosti objekata 73
Logički operatori .75
Nepotpuno izračunavanje 76
Literali .77
Eksponencijalni zapis 78
Operatori nad bitovima .79
Operatori pomeranja .80
Ternarni operator uslovljavanja .84
Operatori + i += za znakovne nizove .85
Česte greške prilikom korišćenja operatora .86
Eksplicitna konverzija tipova .87
Odsecanje i zaokruživanje 88
Unapređenje tipova 89
Java nema operator za određivanje veličine .89
Sažet pregled operatora .89
Sažetak .98

4: Kontrolisanje izvršavanja 99
Logičke vrednosti .99
Naredba **if-else** .99
Petlje .100
Petlja **do-while** 101
Petlja **for** 101
Operator zarez 103
Foreach sintaksa .103
Rezervisana reč **return** .106
Rezervisane reči **break** i **continue** .107
Čuveni goto .108
Naredba **switch** .112
Sažetak .114

5: Inicijalizacija i čišćenje 115
Garantovana inicijalizacija pomoću konstruktora .115
Preklapanje metoda .117
Razlikovanje preklopljenih metoda .119
Preklapanje i prosti tipovi 120
Preklapanje povratnih vrednosti 123
Podrazumevani konstruktori .124
Rezervisana reč **this** .125
Pozivanje konstruktorâ iz konstruktorâ 127
Značenje rezervisane reči **static** 129
Čišćenje: finalizacija i sakupljanje smeća .129
Čemu služi metoda **finalize()**? 130
Morate da čistite sami 131
Stanje okončanja 131
Kako radi sakupljač smeća? 133
Inicijalizacija članica .136
Zadavanje inicijalizacije 137
Inicijalizacija konstruktorima .139
Redosled inicijalizacije 139
Inicijalizacija statičnih elemenata 140
Eksplicitna inicijalizacija statičnih elemenata 143
Inicijalizacija nestatičnih instanci 144
Inicijalizacija nizova .145
Liste promenljivih parametara 150
Nabrojani tipovi .155
Sažetak .158

6: Kontrola pristupa 159
Paket: bibliotečka jedinica .160
Organizacija koda 161
Pravljenje jedinstvenih imena paketa 162
Lična biblioteka sa alatkama 166
Korišćenje uvoženja za promenu ponašanja programa 168
Upozorenje pri radu s paketima 168
Specifikatori pristupa u Javi .168
Paketni pristup 169
public: interfejs za pristup 169
private: to ne sme da se dira! 171
protected: „pristup nasleđivanjem" 172

Interfejs i realizacija 174
Pristup klasama 175
Sažetak.....178

7: Ponovno korišćenje klasa 180

Sintaksa kompozicije 180
Sintaksa nasleđivanja 183
Inicijalizacija osnovne klase 185
Delegiranje 188
Kombinovanje kompozicije i nasleđivanja 189
Garantovanje pravilnog čišćenja ... 191
Sakrivanje imena 194
Izbor između kompozicije i nasleđivanja 196
Rezervisana reč **protected** 197
Svođenje naviše.....198
Zašto „svođenje naviše"?..... 199
Ponovo o biranju između kompozicije i nasleđivanja 200
Rezervisana reč **final** 200
Finalni podaci 200
Finalne metode 204
Finalne klase..... 206
Pažljivo sa **final** 207
Inicijalizacija i učitavanje klasa.....208
Inicijalizacija i nasleđivanje 208
Sažetak.....210

8: Polimorfizam 211

Ponovo o svođenju naviše 211
Zanemarivanje tipa objekta 213
Začkoljica 214
Vezivanje poziva metoda..... 214
Dobijanje pravilnog ponašanja 215
Proširivost 218
Greška: „redefinisanje" privatnih metoda..... 221
Greška: polja i statične metode..... 222
Konstruktori i polimorfizam...224
Redosled poziva konstruktora 224
Nasleđivanje i čišćenje..... 226
Ponašanje polimorfnih metoda unutar konstruktora 230
Kovarijantni povratni tipovi ...232
Projektovanje pomoću nasleđivanja 233
Poređenje zamene i proširivanja ... 235
Svođenje naviše i podaci o tipu prilikom izvršavanja 236
Sažetak.....238

9: Interfejsi 239

Apstraktne klase i metode 239
Interfejsi 242
Potpuno razdvajanje 246
Višestruko nasleđivanje u Javi..251
Proširivanje interfejsa nasleđivanjem.....253
Sukobljavanje imena prilikom kombinovanja interfejsa..... 255
Prilagođavanje interfejsu 256
Polja u interfejsima 258
Inicijalizacija polja u interfejsima .. 259
Ugnežđivanje interfejsa 260
Interfejsi i Proizvođači.....262
Sažetak.....265

10: Unutrašnje klase 266

Pravljenje unutrašnjih klasa ...266
Veza ka spoljnoj klasi 268
Upotreba sintaksi **.this** i **.new** ..270
Unutrašnje klase i svođenje naviše.....271
Unutrašnje klase u metodama i oblastima važenja.....273
Anonimne unutrašnje klase ...275
Ponovo o proizvodnim metodama . 279
Ugnežđene klase 282
Klase unutar interfejsa 283
Pristupanje spoljašnjosti iz višestruko ugnežđenih klasa..... 284
Zašto unutrašnje klase? 285
Zaključci i povratni pozivi 287
Unutrašnje klase i kosturi upravljanja..... 290
Nasleđivanje unutrašnjih klasa.....296
Da li unutrašnja klasa može da se redefiniše? 297

Lokalne unutrašnje klase 299
Identifikatori unutrašnjih klasa.300
Sažetak.....301

11: Čuvanje objekata 302

Generičke klase i kontejneri za bezbedan rad s tipovima....303
Osnovni pojmovi 306
Dodavanje grupa elemenata ...307
Ispisivanje sadržaja kontejnera..309
Liste.....311
Iteratori 315
ListIterator 318
Ulančana lista 319
Pravljenje **steka** od ulančane liste.....320
Funkcionalnost skupa 322
Funkcionalnost mape.....326
Pravljenje reda za čekanje (od ulančane liste) 329
Prioritetni red čekanja (**PriorityQueue**) 330
Poređenje kolekcija i **Iteratora** .332
Foreach i iteratori.....336
Adapterska metoda 338
Sažetak.....341

12: Obrada grešaka pomoću izuzetaka 345

Koncepti 345
Osnovni izuzeci 346
Argumenti izuzetka..... 347
Hvatanje izuzetka.....348
Blok **try**..... 348
Blokovi za obradu izuzetaka..... 348
Pravljenje sopstvenih izuzetaka..349
Izuzeci i zapisivanje..... 352
Specifikacija izuzetaka 355
Hvatanje bilo kog izuzetka 356
Položaj nastanka izuzetka na steku izvršavanja 358
Ponovno generisanje izuzetka 359
Ulančavanje izuzetaka..... 362
Standardni izuzeci u Javi 365
Specijalan slučaj klase **RuntimeException** 365
Čišćenje odredbom **finally** 367
Čemu služi odredba **finally**?..... 368
Upotreba bloka **finally** tokom povratka iz metode pomoću rezervisane reči **return** 371
Mana: izgubljeni izuzetak..... 372
Ograničenja izuzetaka 374
Konstruktori.....377
Pronalaženje sličnih izuzetaka. .382
Drugi pristupi.....383
Istorija 384
Perspektive..... 386
Prosleđivanje izuzetaka na konzolu..387
Pretvaranje proverenih izuzetaka u neproverene 388
Uputstva za izuzetke.....390
Sažetak.....391

13: Znakovni nizovi 392

Nepromenljivi znakovni nizovi .392
Poređenje preklapanja operatora + i **StringBuildera** ..393
Nenamerna rekurzija 397
Operacije sa znakovnim nizovima 398
Formatiranje izlaza.....400
Metoda **printf()**..... 400
System.out.format()..... 400
Klasa **Formatter** 401
Specifikatori formata 402
Konverzije klase **Formatter** 403
String.format()..... 406
Regularni izrazi.....408
Osnove 408
Pravljenje regularnih izraza 411
Kvantifikatori..... 413
CharSequence..... 414
Klase **Pattern** i **Matcher** 414
Metoda **split()** 422
Operacije zamene 422
Metoda **reset()**..... 424
Regularni izrazi i Javin ulazno-izlazni sistem..... 425
Leksičko analiziranje ulaza 427
Graničnici klase **Scanner** 429
Leksička analiza pomoću regularnih izraza 430
Klasa **StringTokenizer**.....431
Sažetak.....431

14: Podaci o tipu 432

Potreba za prepoznavanjem tipa tokom izvršavanja 432
Objekat tipa Class.....434
Literali klase 439
Generičke reference klasa 441
Nova sintaksa za konverziju tipova . 444
Provera pre konverzije tipa 445
Korišćenje literala klase..... 451
Dinamički instanceof 453
Rekurzivno brojanje 454
Registrovane proizvodne metode.....456
Poređenje **instanceof** sa ekvivalencijama klase.....459
Refleksija: informacije o klasi u vreme izvršavanja 461
Čitanje metoda klase..... 462
Dinamički posrednici.....465
Null objekti.....469
Lažni objekti i vezivne funkcije 476
Interfejsi i podaci o tipu.....476
Sažetak.....482

15: Generički tipovi 484

Poređenje sa C++-om 485
Jednostavni generički tipovi ...485
Biblioteka n-torki 487
Klasa steka 490
NasumicnaLista 491
Generički interfejsi.....492
Generičke metode.....496
Korišćenje zaključivanja o tipu argumenta 497
Argumenti promenljive dužine i generičke metode..... 499
Generička metoda za upotrebu s **Generator**ima 500
Generator opšte namene 501
Pojednostavljenje upotrebe n-torki . 502
Uslužna metoda za **Set** 504
Anonimne unutrašnje klase....508
Pravljenje složenih modela 509
Tajanstveno brisanje.....512
Pristup u C++-u 513
Migracijska kompatibilnost 516
Problem s brisanjem 517
Šta se zbiva na granicama..... 518
Kompenzacija za brisanje 522
Pravljenje instanci tipova 523
Nizovi generičkih tipova..... 526
Granice 531
Džokerski argumenti 535
Koliko je prevodilac pametan?..... 537
Kontravarijansa..... 539
Neograničeni džokerski argumenti..... 541
Konverzija hvatanjem 547
Nedostaci.....549
Prosti tipovi ne mogu biti parametri tipa 549
Realizacija parametrizovanih interfejsa..... 551
Eksplicitna konverzija tipova i upozorenja..... 551
Preklapanje 553
Osnovna klasa otima interfejs 554
Samoograničeni tipovi.....555
Generički kôd koji se neobično ponavlja 555
Samoograničenje..... 557
Kovarijansa argumenata..... 559
Dinamička bezbednost tipova. .562
Izuzeci 563
Miksini 565
Miksini u jeziku C++ 566
Mešanje pomoću interfejsa..... 567
Korišćenje obrasca Decorator 568
Miksini s dinamičkim posrednicima..... 570
Latentni tipovi 572
Kompenzacija za nepostojanje latentnih tipova 576
Refleksija 576
Primena metode na sekvencu 577
Kada slučajno nemate odgovarajući interfejs 580
Simuliranje latentnih tipova pomoću adaptera 582
Upotreba funkcijskih objekata kao strategija 585
Sažetak: da li je eksplicitna konverzija tipova zaista tako loša?.....590
Pročitajte i ovo..... 592

16: Nizovi 593

Šta to nizove čini posebnim. . . .593
Nizovi su prvorazredni objekti. .595
Vraćanje niza vrednosti 597
Višedimenzionalni nizovi 599
Nizovi i generički tipovi. 603
Pravljenje podataka
za testiranje 605
Arrays.fill() 605
Generatori podataka 606
Pravljenje nizova od **Generatora** . . . 612
Metode klase **Arrays**. 616
Kopiranje niza 617
Poređenje nizova 618
Poređenje elemenata niza 619
Uređivanje niza 623
Pretraživanje uređenog niza 624
Sažetak. 626

17: Detaljno razmatranje kontejnera 628

Potpuna taksonomija
kontejnera 628
Popunjavanje kontejnera 629
Generatorsko rešenje. 630
Generatori **mapa** 632
Korišćenje apstraktnih klasa. 635
Funkcije interfejsa **Collection** . .643
Opcione operacije 646
Nepodržane operacije 647
Funkcije **liste**. 649
Skupovi i redosled skladištenja. .652
SortedSet 656
Redovi za čekanje 657
Redovi za čekanje s prioritetom 658
Dvostrani redovi za čekanje 660
Iscrpnije o **Mapama** 661
Performanse. 663
SortedMap 666
LinkedHashMap 667
Transformisanje ključeva
i ključevi za heširanje 668
Način rada metode **hashCode()** . . . 671
Transformisanje ključeva zbog
brzine 674
Redefinisanje metode **hashCode()** . . 678
Izbor realizacije. 683
Struktura za ispitivanje
performansi 684
Performanse različitih **Lista** 688
Opasnosti od mikroporedenja
performansi 694
Izbor skupa
(realizacije interfejsa **Set**) 696
Izbor mapa
(realizacija interfejsa **Map**). 698
Uslužne metode 701
Uređivanje i pretraživanje lista
(realizacija interfejsa **List**) 705
Kako **kolekciju** ili **mapu** učiniti
nepromenljivom 706
Sinhronizovanje **kolekcije** ili **mape** . .708
Čuvanje referenci 709
Kontejner **WeakHashMap** 712
Kontejneri Jave 1.0/1.1. 713
Vector i **Enumeration** 713
Klasa **Hashtable** 714
Klasa **Stack**. 714
Klasa **BitSet** 716
Sažetak. 718

18: Javin ulazno-izlazni sistem 719

Klasa **File**. 719
Listanje direktorijuma 719
Uslužne metode za direktorijume . . 723
Provera postojanja i pravljenje
direktorijuma. 728
Ulaz i izlaz 730
Vrste ulaznih tokova 730
Vrste izlaznih tokova
(**OutputStream**) 732
Dodavanje atributa i korisnih
interfejsa 732
Filtriranje ulaznog toka 733
Filtriranje izlaznog toka 734
Klase za čitanje i upisivanje. . . .734
Izvori i ponori podataka. 735
Menjanje ponašanja toka 736
Klase koje nisu promenjene 736
Poseban slučaj: klasa
RandomAccessFile 737
Tipične primene U/I tokova . . .737
Baferisana ulazna datoteka. 738
Čitanje iz memorije. 739

Formatiran ulaz iz memorije 739
Osnove pisanja u datoteku 740
Čuvanje i rekonstruisanje podataka.. 742
Čitanje i upisivanje datoteka s nasumičnim pristupom 744
Cevovodi..... 745
Uslužne klase za čitanje i pisanje 745
Čitanje binarnih datoteka..... 748
Standardni U/I tokovi 749
Čitanje standardnog ulaznog toka .. 749
Omotavanje toka **System.out** u **PrintWriter**..... 750
Preusmeravanje standardnog ulaza/izlaza..... 751
Upravljanje procesima.....752
Nove U/I klase 753
Konverzija podataka 757
Pribavljanje prostih tipova 759
Baferi prikaza..... 761
Rad s podacima pomoću bafera.... 765
Detaljno o baferima..... 766
Datoteke preslikane u memoriju ... 769
Zaključavanje datoteka 772
Komprimovanje 775
Jednostavno komprimovanje u formatu GZIP..... 776
Komprimovanje većeg broja datoteka u formatu Zip..... 777
Java arhive (JAR)..... 779
Serijalizovanje objekata 780
Pronalaženje klase..... 781
Upravljanje serijalizovanjem 785
Korišćenje trajnosti..... 793
XML.....799
Preferences 802
Sažetak.....804

19: Nabrojani tipovi 805

Osnovne mogućnosti nabrojanih tipova.....805
Uvoz statičnih članova u nabrojani tip..... 806
Dodavanje metoda nabrojanom tipu 807
Redefinisanje **enum** metoda..... 808
Nabrojani tipovi u naredbama **switch** 809
Misterija metode **values()** 810
Realizuje, ne nasleđuje.....813
Nasumičan izbor 814
Upotreba interfejsa za organizovanje.....815
Skup **EnumSet** umesto indikatora 819
Korišćenje mape **EnumMap** ...821
Metode koje se menjaju u zavisnosti od konstante 823
Stvaranje lanca odgovornosti pomoću nabrojanih tipova..... 826
Mašine stanja s nabrojanim tipovima..... 830
Višekratno otkrivanje tipa.....836
Otkrivanje tipa pomoću nabrojanih tipova 838
Korišćenje metoda koje se menjaju u zavisnosti od konstante nabrojanog tipa..... 840
Otkrivanje tipa pomoću mapa **EnumMap** 842
Korišćenje 2-D niza..... 843
Sažetak.....844

20: Anotacije 845

Osnovna sintaksa 846
Definisanje anotacije..... 846
Metaanotacije 848
Pisanje procesora anotacija 848
Elementi anotacija..... 849
Ograničenja podrazumevanih vrednosti 850
Generisanje spoljnih datoteka 850
Anotacije ne podržavaju nasleđivanje..... 854
Realizacija procesora..... 854
apt za obradu anotacija 857
Upotreba obrasca Visitor sa alatkom **apt**.....861
Jedinično testiranje pomoću anotacija 864
Testiranje generičkih tipova alatkom **@Unit** 873
„Svite" nisu potrebne 874
Realizacija interfejsa **@Unit** 875
Uklanjanje koda za testiranje..... 881
Sažetak.....883

21: Paralelno izvršavanje 885

Svi aspekti paralelnog izvršavanja....886
Brže izvršavanje....886
Poboljšan dizajn koda....889
Osnove višenitnog programiranja....890
Definisanje zadataka....890
Klasa **Thread**....891
Upotreba izvršilaca (interfejsa **Executor**)....893
Dobijanje povratnih vrednosti od zadataka....896
Spavanje....897
Prioritet....899
Prepuštanje....901
Servisne niti....901
Varijante programiranja....906
Terminologija....911
Kako se pridružiti postojećoj niti...912
Korisničko okruženje koje brzo reaguje....913
Grupe niti....914
Hvatanje izuzetaka....915
Deljenje resursa....918
Nepravilno pristupanje resursima...918
Razrešavanje takmičenja za deljene resurse....921
Atomske operacije i trenutna vidljivost....926
Atomske klase....932
Kritični odelici....933
Sinhronizacija s drugim objektima....939
Lokalno skladište niti....940
Gašenje zadataka....942
Ukrasna bašta....942
Gašenje blokiranih zadataka....945
Prekidanje izvršavanja....946
Provera postoji li prekid....954
Međusobna saradnja zadataka....956
wait() i **notifyAll()**....957
notify() u odnosu na **notifyAll()**...962
Proizvođači i potrošači....965
Proizvođači, potrošači i redovi za čekanje....971
Cevi za ulazno/izlazne operacije između zadataka....976
Uzajamna blokada....978
Nove komponente biblioteke...983
CountDownLatch – brava sa odbrojavanjem....983
CyclicBarrier....986
DelayQueue....988
PriorityBlockingQueue....991
Kontroler staklenika napravljen pomoću **ScheduledExecutora**....994
Semafor....997
Exchanger....1001
Simulacija....1003
Simulacija šalterskog službenika...1003
Simulacija restorana....1008
Raspodela posla....1013
Optimizacija performansi....1018
Poređenje tehnologija uzajamno isključivih brava (mutexa)....1018
Kontejneri bez zaključavanja....1027
Optimističko zaključavanje....1034
ReadWriteLock....1036
Aktivni objekti....1039
Sažetak....1042
Literatura za dalje usavršavanje...1044

22: Grafička korisnička okruženja 1045

Apleti....1047
Osnove Swinga....1047
Alatka za prikazivanje....1050
Pravljenje dugmeta....1051
Hvatanje događaja....1052
Višeredna polja za tekst....1054
Raspoređivanje elemenata....1056
Raspoređivač **BorderLayout**....1056
Raspoređivač **FlowLayout**....1057
Raspoređivač **GridLayout**....1058
Raspoređivač **GridBagLayout**....1058
Apsolutno pozicioniranje....1059
Raspoređivač **BoxLayout**....1059
Koji je pristup najbolji?....1059
Model događaja grafičke biblioteke Swing....1059
Tipovi događaja i prijemnika....1060
Praćenje više događaja....1066
Primeri Swing komponenata...1068
Dugmad....1069
Ikonice....1071

Priručni saveti ... 1073
Jednoredna polja za tekst ... 1073
Ivice ... 1075
Mali program za uređivanje teksta ... 1076
Polja za potvrdu ... 1077
Radio-dugmad ... 1078
Kombinovane liste (padajuće liste) ... 1080
Grafičke liste ... 1081
Okno s jezičcima ... 1083
Okviri za poruke ... 1083
Meniji ... 1085
Iskačući meniji ... 1091
Crtanje ... 1092
Okviri za dijalog ... 1095
Dijalozi za rad s datotekama ... 1099
HTML u komponentama biblioteke Swing ... 1101
Klizači i linije napredovanja ... 1102
Biranje izgleda i ponašanja ... 1103
Stabla, tabele i clipboard ... 1105
JNLP i Java Web Start ... 1105
Paralelno izvršavanje i Swing ... 1110
Dugotrajni zadaci ... 1111
Vizuelno višenitno programiranje ... 1118
Vizuelno programiranje i zrna Jave ... 1120
Šta je zrno? ... 1121
Ispitivanje zrna klasom **Introspector** ... 1123
Naprednije zrno ... 1128
Zrna Jave i sinhronizacija ... 1131
Pakovanje zrna ... 1135
Složenija podrška za zrna ... 1137
Više o zrnima ... 1137
Alternative za Swing ... 1137
Pravljenje Flash Web klijenata pomoću Flexa ... 1138
Zdravo, Flex ... 1139
Prevođenje MXML-a ... 1140
MXML i ActionScript ... 1141
Kontejneri i kontrole ... 1141
Efekti i stilovi ... 1143
Događaji ... 1144
Povezivanje s Javom ... 1144
Modeli podataka i povezivanje podataka ... 1147
Izgradnja i primena ... 1148
Izrada SWT aplikacija ... 1150
Instaliranje SWT-a ... 1150
Zdravo, SWT ... 1150
Izbegavanje redundantnosti ... 1153
Meniji ... 1155
Okna s karticama, dugmad i događaji ... 1156
Grafika ... 1160
Paralelno izvršavanje u SWT-u ... 1162
Poređenje SWT-a i Swinga ... 1164
Sažetak ... 1164
Resursi ... 1165

A: Dodaci 1166

Dodaci koji se mogu preuzeti sa Interneta ... 1166
Misliti na jeziku C: osnova za Javu ... 1166
Seminar Thinking in Java ... 1166
CD sa seminarom Hands-On Java ... 1167
Seminar Thinking in Objects ... 1167
Thinking in Enterprise Java ... 1167
Thinking in Patterns (with Java) ... 1168
Seminar Thinking in Patterns ... 1168
Konsultacije i revizije dizajna ... 1169

B: Resursi 1170

Softver ... 1170
Programi za uređivanje teksta i alatke za pravljenje aplikacija ... 1170
Knjige ... 1171
Analiza i projektovanje ... 1171
Jezik Python ... 1173
Spisak mojih knjiga ... 1174

Spisak termina korišćenih u knjizi 1175

Indeks 1177

Predgovor

Na početku sam prišao Javi kao „još jednom programskom jeziku", što ona umnogome i jeste.

ALI, KAKO JE VREME ODMICALO I ŠTO SAM JE VIŠE PROUČAVAO, POČEO SAM DA UVIĐAM DA je osnovna namena Jave drugačija od namene svih drugih jezika koje sam sretao.

Programiranje je snalaženje u složenosti: složenost problema koji želite da rešite dodaje se na složenost računara na kome se problem rešava. Zbog ove složenosti većina naših programskih projekata propadne. Pa ipak, za gotovo nijedan programski jezik za koji znam nije odlučeno da bi njegov *glavni* cilj trebalo da bude savladavanje složenosti razvoja i održavanja programa.[1] Naravno, mnoge odluke pri stvaranju programskih jezika donete su sa složenošću na umu, ali u nekom trenutku uvek bi se našle neke druge stvari za koje se smatralo da su neophodne u mešavini. Te druge stvari neizbežno prouzrokuju da programer koji koristi dati jezik na kraju udari glavom o zid. Na primer, C++ je morao da bude kompatibilan sa starijim jezikom C (da bi se omogućio lak prelazak C programerima), a pri tom i efikasan. I jedno i drugo su veoma korisni ciljevi i umnogome zaslužni za uspeh jezika C++, ali takođe unose dodatnu složenost zbog koje neki projekti ne bivaju završeni (naravno, možete kriviti programere i upravu, ali, ako jezik može da pomogne tako što će otkrivati vaše greške, zašto to ne bi i radio). Evo još jednog primera. Visual Basic (VB) je bio vezan za BASIC, koji baš nije bio proširiv jezik, tako da su sva proširenja nagomilana u VB-u proizvela zaista zastrašujuću sintaksu koja se veoma teško održava. Perl je vertikalno kompatibilan sa Awkom, Sedom, Grepom i drugim Unixovim alatima koje je trebalo da zameni, a posledica je bila da je često optuživan kako proizvodi „kôd samo za pisanje" (tj. posle nekoliko meseci ne možete ga više čitati). S druge strane, pri stvaranju jezika kao što su C++, VB, Perl i Smalltalk, deo napora je usmeren na rešavanje problema složenosti, pa oni veoma uspešno izlaze na kraj sa određenom vrstom problema.

Dok sam učio Javu, čini mi se da je na mene najviše uticalo to što je Sun među svojim ciljevima pri projektovanju imao i princip smanjenja složenosti *za programera*. Kao da su rekli: „Važno nam je da ubrzamo i pojednostavimo pravljenje robusnog koda". U ranim danima, ovaj princip je rezultovao kodom koji se nije brzo izvršavao (tako se to s vremenom popravilo), ali je zaista zadivljujuće skratio vreme potrebno za pisanje programa – za izradu programa na Javi potrebno je upola manje vremena nego za pravljenje odgovarajućeg C++ programa. Ovaj rezultat sam po sebi može da uštedi mnogo vremena i novca, ali Java se tu ne zaustavlja. Ona nastavlja da pakuje sve važne i složene zadatke, kao što su višenitni rad i mrežno programiranje, u svojstva jezika ili biblioteke koji pojednostavljuju te zadatke. Najzad, latila se nekih zaista veoma složenih problema kakvi su međuplatformsko programiranje, dinamička promena koda, čak i pitanja zaštite – od kojih se svaki na vašoj lestvici složenosti može svrstati od „prepreka" do „potpuno nemoguće". Stoga, uprkos problemima s performansama koje smo susreli, Java silno obećava: može od nas da napravi znatno produktivnije programere.

---

[1] Mislim da je programski jezik Python najbliži tom cilju. Pogledajte *www.Python.org*

Java na sve načine povećava opseg komunikacije *među ljudima*: stvaranjem programa, timskim radom na stvaranju programa, izgradnjom korisničkog okruženja kao veze između programa i korisnika, izvršavanjem programa na različitim tipovima računara i lakšim pisanjem programa kojima se pristupa preko Interneta.

Mislim da ogromna količina informacija koje putuju Internetom nije najvažniji pokazatelj značaja komunikacione revolucije; prava revolucija je u tome što ćemo moći mnogo lakše da razgovaramo – pojedinačno, u grupama, i sa celom planetom. Priča se da bi naredna revolucija iznedrila svojevrstan globalni um proizašao iz iskustva više ljudi i njihove međusobne povezanosti. Java može ali i ne mora, biti alat koji će podstaći tu revoluciju – i zbog same takve mogućnosti, osećam kako je značajno da i drugi nauče ovaj jezik.

Java SE5 i SE6

Ovo izdanje knjige mnogo duguje poboljšanjima Jave koja je Sun prvo nazvao JDK 1.5, a kasnije to izmenio u JDK5 ili J2SE5, te najzad ispustio zastarelo „2" i proglasio verziju Java SE5. Dobar deo izmena u jeziku Java SE5 trebalo je da poboljšaju iskustvo programera. Kao što ćete videti, projektanti Jave nisu u tome potpuno uspeli, ali su, uopšte uzev, napravili veliki korak u pravom smeru.

Jedan od važnih ciljeva ovog izdanja bio je potpuno obuhvatanje poboljšanja Jave SE5/6 – da se ona objašnjavaju i upotrebljavaju u celoj knjizi. To znači da je ovo izdanje, donekle hrabro, „samo za Javu SE5/6" i da se dobar deo koda u knjizi neće prevesti pomoću ranijih verzija Jave; ukoliko pokušate, sistem će se pobuniti i zaustaviti. Mislim ipak da su prednosti ovog pristupa vredne tog rizika.

Ako ste primorani da se služite ranijim verzijama Jave, na *www.MindView.net* možete besplatno preuzeti prethodna izdanja ove knjige. Zbog mnogih razloga, odlučio sam da u besplatnom elektronskom obliku ne ponudim tekuće izdanje knjige, nego samo prethodna.

Java SE6

Ova knjiga je bila ogroman projekat koji je zahtevao mnogo vremena, i pre nego što je izašla, pojavila se beta verzija Jave SE6 (pod radnim imenom *mustang*). Iako je Java SE6 donela nekoliko manjih izmena koje su poboljšale neke primere u knjizi, uglavnom nije uticala na sadržaj ove knjige; glavna obeležja Jave SE6 su povećanje brzine izvršavanja i novi elementi biblioteka, koja nisu tema ove knjige.

Programi navedeni u knjizi uspešno su testirani na kandidatskoj verziji Jave SE6, pa ne očekujem da će ona doneti izmene koje bi uticale na sadržaj ove knjige. Ukoliko u vreme objavljivanja zvanične verzije Jave SE6 bude važnih izmena, njih ćemo uvažiti u izvornom kodu knjige koji se može preuzeti na adresi *www.MindView.net*.

Na koricama knjige piše da je ona „za Javu SE5/6", što znači „napisana za Javu SE5 i veoma važne izmene koje je ta verzija unela u jezik, ali jednako primenljiva i na Javu SE6".

Četvrto izdanje

Zadovoljstvo pisanja novog izdanja knjige donosi činjenica se stvari mogu „uraditi kako treba", u skladu sa onim što sam saznao od izlaska poslednjeg izdanja. Često su ti uvidi tipa „Kada ne dobiješ ono što si hteo, dobiješ pouku", a za mene je to prilika da popravim ono što je pogrešno ili prosto zamorno. Jednako tako, pisanje novog izdanja dovodi do fascinantnih novih zamisli, a stid zbog pogrešaka mnogo je manji od uživanja u otkriću i mogućnosti da se zamisli izraze bolje nego pre.

Tu je i izazov da knjigu napišem tako da i vlasnici prethodnih izdanja požele da je kupe. To me tera da poboljšavam, prerađujem i premeštam sve što mogu da bi knjiga postala novo i vredno iskustvo za posvećene čitaoce.

Izmene

U ovom izdanju nema CD-a. Osnovni deo tog CD-a, multimedijski seminar *Thinking in C* (koji je za MindView napravio Chuck Allison), sada se može preuzeti u obliku Flash prezentacije. Polaznici tog seminara koji nedovoljno poznaju sintaksu C-a obučavaju se da ovladaju materijalom iz ove knjige. Iako dva poglavlja ove knjige daju pristojan uvod u sintaksu, ona možda neće biti dovoljna osobama bez adekvatnog predznanja, a prezentacija *Thinking in C* upravo njima pomaže da dostignu potreban nivo.

Poglavlje *Paralelno izvršavanje* (ranije „Višenitno izvršavanje") potpuno je prerađeno da bi odrazilo velike izmene u odgovarajućim bibliotekama Jave SE5, ali još uvek pruža temelj za osnovne koncepte paralelnog izvršavanja. Bez tog jezgra, teško ćete razumeti složenije oblasti paralelnog izvršavanja. Mnogo sam meseci radio na tome, uronjen u taj drugi „paralelan" svet, i na kraju je ispalo da to poglavlje daje ne samo osnovu, nego i naprednije uvide.

Značajnim novim mogućnostima Jave SE5 posvećena su nova poglavlja, a ostale su utkane u izmene postojećeg materijala. Pošto ja neprestano proučavam projektne obrasce, povećan je i njihov broj u knjizi.

U knjizi je bilo mnogo premeštanja, u dobroj meri nadahnutih predavačkom praksom i saznanjem da bi se moje poimanje onoga što treba da stane u jedno poglavlje moglo razborito kritikovati. Bio sam sklon nepromišljenom verovanju da tema mora biti „dovoljno velika" kako bi zaslužila sopstveno poglavlje. Ali, naročito tokom predavanja projektnih obrazaca, uvideo sam da polaznici seminara najbolje shvataju kada uvodim jedan po jedan obrazac i odmah uradim odgovarajuću vežbu, čak i ako zbog toga tek nakratko govorim. (Otkrio sam da je taj tempo prijatniji i za mene kao predavača.) Zato sam u ovoj verziji knjige pokušao da podelim poglavlja na teme i da ne brinem koliko su duga. Čini mi se da je tako bolje.

Uvideo sam i kolika je važnost testiranja programa. Ukoliko nemate ugrađen sistem za testiranje, u kojem se testiranje obavlja svaki put kada generišete program, ne možete znati koliko je vas program pouzdan. Da bih izbegao tu nesigurnost, napravio sam sistem za testiranje koji prikazuje i vrednuje rezultate svakog programa. (Sistem je napisan na Pythonu; naći ćete ga u kodu ove knjige koji se može preuzeti na lokaciji *www.MindView.net*.) Testiranje uopšte razmotreno je u dodatku objavljenom na adresi *http://MindView.net/Books/BetterJava*; tu su objašnjene osnovne veštine kojima bi, po mom mišljenju, svi programeri trebalo da vladaju.

Pored toga, pročešljao sam sve primere u knjizi i zapitao se: „Zašto sam to uradio baš ovako?" U većini slučajeva nešto sam izmenio i poboljšao, da bi primeri bili dosledniji i da bih pokazao ono što smatram najboljom praksom u pisanju Java koda (koliko se to može u tekstu uvodnog nivoa). Dizajn i realizacija mnogih postojećih primera, značajno su izmenjeni. Uklonjeni su primeri koji su izgubili smisao i dodati novi.[2]

Čitaoci su imali zaista divne komentare na prva tri izdanja ove knjige što mi je bilo veoma drago. Tu i tamo, poneko bi imao i primedbu i iz nekog razloga, periodično se javljala ista primedba „knjiga je preobimna". Po meni, ako čitaocu smeta samo obim dela, to i nije tako strašno (prisetimo se primedbe austrijskog cara o Mocartovom delu: „Previše nota!", iako se ja ni na koji način ne poredim s Mocartom). Mogu samo da pretpostavim kako takvu primedbu upućuje onaj ko će se tek upoznati sa širinom Jave i ko nije video ostale knjige o ovom jeziku. Ipak, pokušao sam da u ovom izdanju skratim delove koji su postali nepotrebni ili barem ne previše značajni. Po pravilu, pokušao sam da uklonim sve što više nije potrebno, ubacim izmene i poboljšam sve što sam mogao. Mislim da sam to mogao slobodno da uradim, jer je originalni materijal i dalje na Internetu (na adresi *www.MindView.net*), u obliku prva tri izdanja koje se mogu besplatno preuzeti, i dodataka za ovu knjigu.

Iskreno se izvinjavam svima koji još uvek ne mogu da podnesu veličinu ove knjige. Verovali ili ne, naporno sam radio da bih je smanjio.

Napomena o dizajnu korica

Korice knjige *Misliti na Javi* inspirisala je američka varijanta pokreta umetnost i zanatstvo (American Art & Crafts Movement) koji se javio početkom 20. veka i dostigao zenit između 1900. i 1920. Nastao je u Engleskoj kao reakcija na mašinsku proizvodnju, industrijsku revoluciju i veoma izražen dekorativni stil viktorijanskog doba. Pokret se zalagao za umeren dizajn i prirodne forme koje je promovisao umetnički pravac art nuvo (fr. *l'art nouveau*). Isticana je važnost umetnika kao pojedinca, a nije odbacivano korišćenje modernih alata. Postoje velike sličnosti s današnjom situacijom: prelazak u novi vek, napredak od korena računarske revolucije do nečega plemenitijeg i značajnijeg za pojedinca, isticanje veštine pisanja softvera umesto čiste proizvodnje koda.

Javu vidim kao pokušaj uzdizanja programera iznad nivoa tehničara operativnog sistema, u pravcu „softverskog majstora".

I autor i dizajner korica (koji su prijatelji od detinjstva) nalaze inspiraciju u ovom pokretu i obojica su okruženi nameštajem, lampama i drugim predmetima koji ili potiču iz tog perioda, ili su njime inspirisani.

Drugi detalj na ovim koricama je kutija za prikupljanje prirodnjačkih eksponata – u ovom slučaju insekata. Ti insekti su objekti stavljeni u kutije kao objekte. Same kutije su stavljene u objekat „korica", što ilustruje osnovni koncept kombinovanja u objektno orijentisanom programiranju. Programeru se lako nameće asocijacija na „bubice"; bubice su

[2] Većina primera je prevedena, pa je moguće da su se potkrale i greške. Ukoliko uočite grešku, molimo vas da pošaljete poruku na *redakcija@mikroknjiga.co.yu*. Originalne verzije svih primera naći ćete na *www.mindviewinc.com/TIJ4/CodeInstructions.html*.

ovde uhvaćene i, po svoj prilici, ubijene u tegli za uzorke i smeštene u male izložbene kutije: sve ovo nagoveštava Javinu sposobnost da pronalazi, prikazuje i obuzdava bubice (što je zbilja jedan od njenih najjačih aduta).

Za ovo izdanje naslikao sam akvarel prikazan na koricama, ispod stilizovane kutije.

Zahvalnice

Prvo, hvala saradnicima koji su radili sa mnom na seminaru, u konsultacijama i pri ostvarivanju nastavnih projekata. To su: Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer i Jamie King. Cenim vaše strpljenje, dok i dalje pokušavam da napravim najbolji model za međusobnu saradnju nezavisnih ljudi kao što smo mi.

Odnedavno, bez sumnje zaslugom Interneta, povezao sam se sa iznenađujuće velikim brojem ljudi koji mi pomažu u poslu, uglavnom radeći u svojim kućnim kancelarijama. U prošlo vreme, morao bih da iznajmim prilično veliki kancelarijski prostor gde bih smestio sve te ljude, ali zahvaljujući Mreži, ekspres pošti i telefonu, koristim njihovu pomoć bez tih dodatnih troškova. Tokom mojih pokušaja da naučim da se „lepo igram sa ostalima" svi vi ste bili veoma susretljivi, i nadam se da ću i dalje bolje raditi svoj posao uz tuđu pomoć. Paula Steuer je neprocenjiva zato što je preuzela moju ofrlje poslovnu praksu i postavila je na zdrave temelje (hvala Paula, što mi ne daš mira kada me mrzi da nešto uradim). Jonathan Wilcox, Esq. pročešljao je strukturu mog preduzeća, prevrnuo svaki kamen ispod kojeg se možda kriju škorpioni i naterao nas da prođemo kroz postupak pravno valjanog organizovanja. Hvala ti na staranju i upornosti. Sharlynn Cobaugh je napravila od sebe stručnjaka za obradu zvuka i postala je bitan deo tima za pravljenje multimedijskih kurseva, kao i za rešavanje drugih problema. Hvala na upornosti koju pokazuješ suočena s nepredvidivim računarskim problemima. Ljudi iz praške firme Amaio izvukli su me iz neprilika u nekoliko projekata. Daniel Will-Harris me je prvi upoznao s radom preko Interneta, a i glavni je dizajner svih grafičkih rešenja.

Tokom mnogih godina, Gerald Weinberg je putem svojih konferencija i radionica postao moj nezvanični trener i mentor, na čemu mu zahvaljujem.

Ervin Varga mi je izuzetno mnogo pomogao svojim tehničkim ispravkama 4. izdanja – iako su drugi pomagali u pojedinim poglavljima i primerima, Ervin je bio osnovni tehnički recenzent cele knjige, a preradio je i vodič s rešenjima za 4. izdanje. Ervin je pronašao greške i uneo poboljšanja u knjigu koja su nemerljiv doprinos ovom tekstu. Njegova temeljnost i pažnja koju obraća na pojedinosti začudujuće su, i on je najbolji recenzent kog sam imao. Hvala ti, Ervine.

Moj blog na lokaciji *www.Artima.com* Bila Vennersa bio je izvor pomoći kada sam tražio tuđa mišljenja. Hvala čitaocima koji su mi slali komentare na osnovu kojih sam razjasnio pojmove – to su James Watson, Howard Lovatt, Michael Barker i drugi, posebno oni koji su pomogli s generičkim tipovima.

Hvala Marku Welshu na njegovoj stalnoj pomoći.

Evan Cofsky mi stalno pruža veliku podršku svojim poznavanjem (napamet!) svih zamršenih pojedinosti o podešavanju i održavanju Web servera na Linuxu. On se stara da MindViewov server bude podešen i bezbedan.

Posebno hvala mom novom prijatelju, kafi, koja mi je ulila gotovo neograničen entuzijazam za ovaj projekat. Kafić Camp4 u gradu Crested Butte u Koloradu bio je stalno sastajalište ljudi koji su dolazili na MindViewove seminare, a tokom seminarskih odmora pružao nam je najbolju opskrbu hranom i pićem koju sam ikada dobio. Hvala mom drugaru Alu Smithu što ga je stvorio i od njega napravio tako zanimljivo i zabavno mesto. I svim barmenima kafića koji tako veselo toče piće.

Hvala ljudima u Prentice Hallu na tolerantnosti i ogromnom strpljenju kojim mi obezbeđuju sve što tražim.

Određene alatke su se pokazale nezamenljivim tokom procesa razvoja programa i ja osetim veliku zahvalnost prema njihovim stvaraocima svaki put kada ih upotrebim. Cygwin (*www.cygwin.com*) rešio mi je bezbroj problema koje Windows neće/ne može, svakog dana sam mu sve više privržen (da sam ga samo imao pre 15 godina kada je moj mozak još bio „zalemljen" za GNU Emacse). IBM-ov Eclipse (*www.eclipse.org*) zaista je divan doprinos zajednici projektanata softvera, pa očekujem da će i ubuduće davati velike rezultate, pošto se i dalje razvija. (Od kad je to IBM postao kul? Mora da mi je nešto promaklo.) JetBrains IntelliJ Idea nastavlja da krči nove kreativne puteve među razvojnim alatkama.

Na ovoj knjizi počeo sam da koristim Sparxsystemsov Enterprise Architect i on je brzo postao moja omiljena UML alatka. U raznim prilikama dobro je poslužio formater koda Jalopy Marca Hunsickera (*www.triemax.com*), a Marco je bio veoma susretljiv i konfigurisao ga prema mojim specifičnim potrebama. Katkada je bio koristan i JEdit Slava Pestova (*www.jedit.org*) sa svojim softverskim dodacima, i to je sasvim dobar editor za početnike na seminarima.

I naravno, ako to nisam već dovoljno puta rekao na svim mogućim mestima, za rešavanje problema stalno koristim Python (*www.Python.org*). On je delo mog drugara Guida Van Rossuma i grupe uvrnutih genija s kojima sam trčao nekoliko divnih dana (Time Petersu, uramio sam miša kojeg si pozajmio i zvanično ga nazvao „TimBotMiš"). Vi, momci, treba da nađete neko zdravije mesto za ručak. (Takođe, hvala celoj Python zajednici, oni su fenomenalna gomila ljudi.)

Mnogi su mi slali ispravke i svima sam dužnik, ali posebnu zahvalnost zaslužuju (za prvo izdanje): Kevin Raulerson (koji je pronašao obilje sjajnih bubica), Bob Resendes (zaista neverovatan), John Pinto, Joe Dante, Joe Sharp (sva trojica su bili čudesni), David Combs (mnoge gramatičke ispravke i razjašnjenja), dr Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristić, Neil Galarneau, David B. Malkovsky, Steve Wilkinson i mnogi drugi. Prof. dr Marc Meurrens uložio je veliki napor da u Evropi objavi i učini dostupnom elektronsku verziju prvog izdanja knjige.

Hvala svima koji su mi pomogli da ponovo napišem primere uz korišćenje biblioteke Swing (za 2. izdanje) i za svu drugu pomoć. To su Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis i svi koji su mi dali podršku.

U 4. izdanju, Chris Grindstaff je mnogo pomogao tokom razvoja odeljka o SWT-u, a Sean Neville je za mene napisao prvu verziju odeljka o Flexu.

Kad god pomislim da sam najzad sve naučio o programiranju za paraleleno izvršavanje, otvore se neka nova vrata i pokaže se da ima još planina koje moram preći. Hvala Brianu Goetzu što mi je pomogao da savladam sve prepreke u novoj verziji poglavlja *Paralelno izvršavanje* i što je otkrio (nadam se!) sve greške.

Nije iznenađenje što mi je poznavanje Delphija pomoglo da razumem Javu, budući da ta dva jezika imaju puno zajedničkih koncepata i dizajnerskih rešenja. Moji prijatelji koji se bave Delphijem pomogli su mi da proniknem u unutrašnjost tog sjajnog programskog okruženja. To su Marco Cantu (još jedan Italijan: da li dobro poznavanje latinskog doprinosi sklonosti ka programskim jezicima?), Neil Rubenking (koji se bavio jogom, vegetarijanstvom i zenom dok nije otkrio računare) i, naravno, Zack Urlocker (prvi direktor projekta Delphi), stari prijatelj s kojim sam proputovao svet. Svi mi smo dužnici briljantnom Andersu Hejlsbergu, koji se i dalje muči sa C#-om (a taj jezik je bio glavni uzor za Javu SE5, u šta ćete se uveriti u knjizi).

Moj pronicljivi prijatelj Richard Hale Shaw pružio mi je vrlo korisnu podršku (i Kim takođe). Richard i ja smo proveli više meseci zajedno, držeći seminare, pokušavajući da osmislimo savršen metod učenja za polaznike.

Dizajn knjige, dizajn korica i sliku na koricama uradio je moj prijatelj Daniel Will-Harris, priznati autor i dizajner (*www.Will-Harris.com*), koji se igrao sa samolepljivim slovima u osnovnoj školi dok je čekao da računari i stono izdavaštvo budu otkriveni, i koji se žalio na moje gunđanje zbog algebarskih problema. Ipak, knjigu sam složio sâm tako da sve greške u slaganju idu na moj račun. Pisao sam u Microsoftovom Wordu XP, a fotoslog je pripremljen u Adobeovom Acrobatu. Desilo se da sam bio u inostranstvu oba puta kada sam pravio finalnu verziju knjige – prvo izdanje sam slao iz Kejptauna u Južnoj Africi, a drugo iz Praga (što je moj doprinos eri elektronike). Treće i četvrto sam poslao iz Crested Butta u Koloradu. Font korišćen na koricama je *ITC Rennie Mackintosh*.

Posebno zahvaljujem svojim učiteljima i svim svojim studentima (koji su istovremeno i moji učitelji).

Dok sam radio na ovom izdanju, u krilu mi je često ležala mačka Molly i tako davala svoju toplu, krznenu podršku.

Listu prijatelja koji su me podržavali čine između ostalih i: Patty Gast (izvanredna maserka), Andrew Birnstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates iz časopisa *Midnight Engineering*, Larry Constantine i Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris i Laura Strand, porodica Almquist, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, porodice Robbins, porodice Moelter (i McMillan), Michael Wilk, Dave Stoner, porodica Cranston, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin i Sonda Donovan, Joe Lordi, Dave i Brenda Bartlett, Patti Gast, Blake, Annette i Jade, porodica Rentschler, porodica Sudek, Dick, Patty i Lee Eckel, Lynn i Todd i njihove porodice. I, naravno, moji mama i tata.

Uvod

„Čoveku je dao jezik, a jezik je stvorio misao, koja je mera Univerzuma" – Oslobođeni Prometej, *Šeli*

Ljudska bića... sasvim su pod vlašću određenog jezika koji je postao sredstvo izražavanja u njihovom društvu. U zabludi je ko misli da u prilagođavanju pojedinca stvarnosti jezik ne igra značajnu ulogu i da je jezik samo slučajno sredstvo rešavanja specifičnih problema opštenja i mišljenja. Činjenica je da je „stvarni svet" u velikoj meri nesvesno izgrađen na osnovu jezičkih navika grupe.
Status lingvistike kao nauke, *Edvard Sapir* 1929.

KAO I BILO KOJI LJUDSKI JEZIK, JAVA OMOGUĆUJE ISKAZIVANJE POJMOVA. S NARASTANJEM I usložnjavanjem problema, ovaj način izražavanja biće vam znatno lakši i fleksibilniji od bilo kojeg drugog, ukoliko bude uspešan.

Ne možete gledati na Javu samo kao na gomilu mogućnosti, jer neke mogućnosti nemaju smisla same za sebe. Celinu svih delova možete da koristite samo ako razmišljate o *projektovanju*, a ne o pisanju koda. Da biste razumeli Javu na ovaj način, morate razumeti i probleme koji se pri tome javljaju, kao i pri programiranju uopšte. Ova knjiga razmatra probleme pri programiranju, objašnjava zbog čega oni predstavljaju probleme i pokazuje postupke kojima ih Java rešava. Zato se skup mogućnosti koje objašnjavam u svakom poglavlju zasniva na načinu rešavanja pojedine vrste problema pomoću Jave. Na taj način, trudim se da vas dovedem, korak po korak, do tačke kada Java postaje vaš maternji jezik.

Od početka do kraja, moj stav počiva na tome da vi želite sebi da predočite model koji će vas dovesti do dubokog razumevanja jezika; ako naidete na zagonetku, moći ćete da je ubacite u svoj model i pronađete odgovor.

Preduslovi

U ovoj knjizi podrazumeva se da koliko-toliko poznajete programiranje: shvatate da je program skup naredaba, poznajete princip potprograma/funkcije/makroa, kontrolnih struktura kao što je „if", konstrukcija za petlje kao što je „while" itd. Sve ovo ste mogli naučiti na raznim mestima, recimo pri programiranju na nekom makro-jeziku ili pri radu sa alatom kao što je Perl. Bez obzira na to koliko ste u dosadašnjem programiranju ovladali osnovnim idejama, moći ćete da radite po ovoj knjizi. Naravno, knjiga će biti *lakša* C programerima, i još više C++ programerima, ali ne otpisujte sebe ako nemate iskustva u ovim jezicima – samo budite spremni da naporno radite. Takođe, uz pomoć multimedijalnog seminara *Thinking in C* koji možete preuzeti s lokacije *www.MindView.net*, ovladaćete u potpunosti osnovama potrebnim za učenje Jave. Kroz knjigu ću postepeno uvoditi koncepte objektno orijentisanog programiranja (OOP) i osnovnih Javinih kontrolnih mehanizama.

Česta pozivanja na osobine jezika C i C++ nisu komentari poznavaoca, već su pomoć programerima da uporede Javu s tim jezicima iz kojih je potekla. Trudiću se da ta poređenja budu jednostavna i objasniću sve što mislim da ne zna neko ko ne koristi C/C++.

Učenje Jave

Otprilike u isto vreme kada se pojavila moja prva knjiga, *Using C++* (Osborne/McGraw-Hill, 1989), počeo sam da predajem taj jezik. Predavanje programskih jezika postala je moja profesija: gledao sam glave koje klimaju, bele poglede i zbunjene izraze u publici širom sveta još od 1987. Kada sam počeo privatno da obučavam manje grupe ljudi, tokom vežbi sam otkrio da mnoga pitanja zbunjuju čak i one koji su se smešili i klimali glavom. Više godina predsedavao sam na odseku za C++ na Konferenciji za razvoj softvera (kasnije i na odseku za Javu), i primetio sam kako smo i ja i drugi govornici skloni da prosečnoj publici serviramo previše informacija za kratko vreme. Zbog različitih nivoa publike i načina na koji sam predstavljao materijal, na kraju bih izgubio deo slušalaca. Možda previše tražim, ali pošto osećam otpor prema tradicionalnom načinu predavanja (a verujem da većina slušalaca takav otpor oseća zbog dosade), pokušao sam da učinim da moja predavanja nikome ne budu dosadna.

Jedno vreme sam pravio dosta raznih prezentacija u relativno kratkom periodu. Tako sam došao u situaciju da učim „metodom pokušaja i pogrešaka" (što je dobar način i za projektovanje programa). Na kraju sam iskoristio sve što sam naučio kao predavač – i napravio kurs koji bih rado držao duže vreme. Sada ovaj kurs držim na javnim i privatnim seminarima o Javi; to je moj glavni uvodni seminar, koji pruža osnovu za naprednije seminare. O tim seminarima možete saznati više na adresi *www.MindView.net*. (Uvodni seminar je dostupan i u obliku CD-a *Hands on Java*. Podaci o njemu su na istoj Web lokaciji.)

Povratne informacije koje dobijam na svakom seminaru pomažu mi da menjam i preradujem materijal sve dok ne zaključim da je podesan kao sredstvo poučavanja. Ovu knjigu ne čine samo beleške sa seminara: u nju sam pokušao da spakujem što više informacija i spojio sam ih tako da vas vode od jednog do drugog predmeta. Zamislio sam da ova knjiga služi usamljenom čitaocu koji se bori s novim programskim jezikom.

Ciljevi

Kao i moja prethodna knjiga, *Misliti na jeziku C++*, i ova knjiga je pisana u skladu s načinom na koji ljudi uče jezik Java. Poglavlje u knjizi zamišljao sam kao jednu dobru lekciju na seminaru. Povratne informacije od slušalaca na seminarima pomogle su mi da otkrijem koji su delovi teži i zahtevaju dodatna objašnjenja. U oblastima gde sam bio preambiciozan i uvrstio previše novih pojmova odjednom, spoznao sam – kroz proces prikaza materijala – da više novih pojmova zahteva više objašnjavanja, a to lako zbunjuje slušaoca.

Cilj svakog poglavlja je da savladate jedan pojam, ili manje grupe povezanih pojmova, bez obzira na sve ostalo što se tu pominje. Na taj način možete da prihvatite svaki delić teorije, u zavisnosti od vašeg dotadašnjeg znanja, pre nego što produžite dalje.

Ciljevi koje sam hteo da ostvarim u ovoj knjizi jesu:

1. Da predstavim materijal u jednostavnim koracima, tako da lako savladate svaki koncept pre nego što nastavite. Da pažljivo odaberem redosled pojmova koje objašnjavam, tako da ne naiđete na nešto s čim se ranije niste susreli. Naravno, to nije uvek moguće; u takvim situacijama, dat je kratak uvodni opis.
2. Da koristim što jednostavnije i kraće primere. To me ponekad sprečava da se uhvatim u koštac sa „stvarnim" problemima, ali sam primetio da su početnici obično srećniji kada mogu da shvate svaki detalj primera, i nisu impresionirani obimom problema koji se rešavaju. Takođe postoji ozbiljno ograničenje količine koda koji može da se razume u učionici. Oni kojima se to ne dopada, neka ga prihvate kao pedagoški ispravniji pristup.
3. Da pružim ono što smatram bitnim za razumevanje jezika, a ne sve što znam. Verujem u hijerarhiju informacija, kao i da postoje činjenice koje 95% programera nikada neće imati potrebe da zna, a koje samo zbunjuju i povećavaju složenost jezika. Da uzmem primer iz C-a: ako zapamtite tabelu prioriteta operatora (ja nikada nisam), možete da pišete efikasan, ali nečitljiv kôd. Ali ako malo promislite o takvom stilu pisanja, shvatićete da zbunjuje onoga ko čita/održava kôd. Stoga zaboravite na prioritete i koristite zagrade svuda gde stvari nisu potpuno jasne.
4. Da svaki odeljak bude dovoljno usredsređen, tako da vreme izlaganja i pauza između izlaganja i vežbe budu kratki. To slušaoca drži budnim i aktivnim tokom seminara, a čitaocu pruža osećanje da brže ovladava materijom.
5. Da vam dam čvrst temelj, kako biste mogli dobro da razumete teoriju i da sami nastavite da proučavate Javu.

Poučavanje na osnovu ove knjige

Prvo izdanje ove knjige nastalo je na osnovu jednonedeljnog seminara. Dok je Java bila u povojima, to je bilo dovoljno vremena za savladavanje ovog jezika. Kako je Java rasla i obuhvatala sve više mogućnosti i biblioteka, ja sam tvrdoglavo nastojao da sve to ispredajem za sedmicu dana. U jednom trenutku, neki naručilac je od mene zatražio da predajem „samo osnove", i radeći to otkrio sam da je sabijanje svega u jednu sedmicu postalo mučno i za mene i za polaznike. Java više nije bila „jednostavan" jezik koji se može ispredavati za nedelju dana.

To iskustvo i spoznaja potakli su me da reorganizujem knjigu, koja je sada napisana tako da može da se ispredaje za dve sedmice na seminaru ili dva semestra na fakultetu. Uvodni deo se završava s poglavljem *Obrada grešaka pomoću izuzetaka*, čemu biste mogli da dodate uvod u JDBC, servlete i serverske Java stranice. To sačinjava osnovni kurs i jezgro CD-a *Hands-On Java*. Ostatak knjige obuhvata kurs srednjeg nivoa i nalazi se na CD-u *Intermediate Thinking in Java*. Oba CD-a možete kupiti na *www.MindView.net*.

Ukoliko vam trebaju informacije o dodatnom materijalu za predavače, obratite se Prentice-Hallu na lokaciji *www.prenhallprofessional.com*.

Dokumentacija na Webu

Jezik Java i njegove biblioteke iz Sun Microsystemsa (besplatno se preuzimaju s Weba) imaju dokumentaciju u elektronskom obliku, kojoj možete pristupati koristeći čitač Weba; gotovo svaka druga nezavisna distribucija Jave ima takav ili sličan sistem dokumentacije. U većini knjiga o Javi kopira se taj sistem. Stoga takvu dokumentaciju ili već imate, ili je možete preuzeti s Weba, i ukoliko to zaista nije neophodno, u ovoj knjizi je nećemo ponavljati. Obično mnogo brže možete da nađete opis neke klase pomoću čitača Weba, nego da ga tražite po knjizi (a dokumentacija na Webu je verovatno ažurnija). Jednostavno ću vas uputiti na „dokumentaciju JDK-a". Ova knjiga će obezbediti dodatne opise klasa samo kada je neophodno dopuniti dokumentaciju da biste shvatili određeni primer.

Vežbe

Primetio sam da jednostavne vežbe pomažu studentima da shvate seminarsko gradivo, pa dajem nekoliko vežbi na kraju svakog poglavlja.

Veći broj vežbi može relativno lako i u razumnom roku da se uradi u učionici, pod nadzorom instruktora koji proverava da li su svi studenti usvojili izloženu materiju. Neke vežbe su malo izazovnije, ali nijedna nije nerešiva.

Rešenja izabranih vežbi nalaze se u elektronskom dokumentu pod imenom *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Javini temelji

Sa ovom knjigom dobijate i besplatan multimedijalni seminar koji možete preuzeti sa *www.MindView.com*. To je seminar *Thinking in C*, koji vas uvodi u sintaksu, operatore i funkcije jezika C na osnovu kojih je Java napravljena. U prethodnim izdanjima ove knjige na engleskom, to se nalazilo na CD-u *Foundations for Java*, ali sada se može besplatno preuzeti.

Prvobitno sam od Chucka Allisona naručio da napravi *Thinking in C* kao nezavisan proizvod, ali sam odlučio da ga priložim uz drugo izdanje knjige *Thinking in C++* i drugo i treće izdanje knjige *Thinking in Java*, pošto su mi na seminar stalno dolazili ljudi bez dovoljnog poznavanja osnovne sintakse C-a. Izgleda da takva osoba misli: „Ja sam pametan programer i neću da učim C, nego C++ ili Javu, pa ću C preskočiti i odmah preći na C++/Javu". Po dolasku na seminar, ljudi polako shvataju da sam imao veoma dobar razlog što sam poznavanje C-a proglasio za preduslov za pohađanje.

Tehnologije se menjaju, i bilo je bolje preraditi *Thinking in C* u Flash prezentaciju koja se može preuzeti sa Interneta, nego prilagati ga na CD-u. Pošto je taj seminar na Internetu, svi mogu dobro da se pripreme *pre* nego što dođu na moj seminar.

Seminar *Thinking in C* i ovoj knjizi pribavlja veću publiku. Iako njena poglavlja *Operatori* i *Kontrolisanje izvršavanja* obuhvataju osnove Jave koje potiču iz C-a, mrežni seminar postepenije uvodi čitaoca, a za njega je potrebno još manje programerskog predznanja nego za knjigu.

Izvorni kôd

Kompletan izvorni kôd iz ove knjige dostupan je kao slobodan softver zaštićen autorskim pravom, u jednom paketu, na adresi *www.MindView.net*. Ovo je zvanična Web lokacija za distribuciju koda i elektronskog izdanja knjige, pa možete biti sigurni da ćete tu uvek naći najnoviju verziju. Kôd možete deliti i koristiti u učionicama i u druge obrazovne namene.

Osnovni cilj zadržavanja autorskog prava jeste da se poreklo koda ispravno navede i da se spreči objavljivanje koda u štampanim medijima bez dozvole. (Ukoliko je poreklo koda ispravno navedeno, primeri iz knjige mogu se navoditi u većini medija.)

U svakoj datoteci sa izvornim kodom naći ćete sledeću napomenu o autorskim pravima (obično na engleskom):

```
//:! AutorskaPrava.txt
Ovaj računarski izvorni kôd zaštićen je autorskim pravima (c)2006 MindView,
Inc.
Sva prava zadržana.

Dozvoljava se besplatna upotreba, kopiranje, prepravljanje i distribuiranje
ovog računarskog izvornog koda (Izvorni kôd) i njegove dokumentacije, u dole
navedene svrhe i bez pismenog odobrenja, ukoliko gornju poruku o autorskim
pravima, ovaj i sledećih pet numerisanih pasusa ne uklonite ni iz jedne
njegove kopije.

1. Dozvoljava se prevođenje Izvornog koda. Preveden Izvorni kôd smete da
uključite u privatne i komercijalne softverske programe, ali samo u izvršnom
formatu.

2. Neizmenjen Izvorni kôd smete da upotrebljavate za potrebe nastave i u
materijalima za prezentaciju, ukoliko navedete da je potekao iz knjige
„Misliti na Javi".

3. Dozvolu za korišćenje Izvornog koda u štampanim medijima možete pribaviti
ako se obratite na adresu:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941 Wayne@MindView.net

4. MindView, Inc. je zaštitio autorska prava Izvornog koda i dokumentacije.
Izvorni kôd je dostupan u obliku u kojem je dat i uz njega se ne dobija
nikakva eksplicitna ili implicitna garancija, uključujući garanciju o pro-
daji, pogodnosti za određenu upotrebu ili nekršenju nečijih prava. MindView,
Inc. ne jamči da će bilo koji program koji sadrži Izvorni kôd raditi bez
prekida ili grešaka. MindView, Inc. ne tvrdi da je Izvorni kôd ili bilo koji
softver koji ga obuhvata podesan za bilo koju namenu. Celokupan rizik, u vezi
s kvalitetom i performansama ovog softvera, snosi sam korisnik Izvornog koda.
Korisnik Izvornog koda razume da je Izvorni kôd napravljen radi istraživanja i
nastave, pa mu se savetuje da se ni u koju svrhu niti iz bilo kojeg razloga ne
oslanja isključivo na Izvorni kôd, niti na bilo koji program koji ga sadrži.
Ako se ispostavi da je Izvorni kôd neispravan, sam korisnik snosi sve tro-
škove servisiranja, opravke ili ispravki.
```

```
5. NI U JEDNOM SLUČAJU NI MINDVIEW, INC. NITI NJEGOV IZDAVAČ NEĆE BITI
ODGOVORNI BILO KOME, BEZ OBZIRA NA BILO KAKVU PRAVNU TEORIJU, ZA DIREKTNU,
INDIREKTNU, POSEBNU, POSLEDIČNU ILI SLUČAJNU ŠTETU, ZA BILO KAKAV PREKID
POSLOVANJA, GUBITAK PROFITA ILI POSLOVNIH PODATAKA, ILI BILO KAKAV DRUGI
NOVČANI GUBITAK ODNOSNO TELESNU OZLEDU KOJE SU IZAZVANE UPOTREBOM OVOG IZVOR-
NOG KODA I NJEGOVE DOKUMENTACIJE ILI SU POSLEDICA NEMOGUĆNOSTI UPOTREBE BILO
KOJEG REZULTUJUĆEG PROGRAMA, ČAK I KADA BI MINDVIEW, INC. ILI NJEGOV IZDAVAČ
BILI UPOZORENI NA MOGUĆNOST TAKVIH ŠTETA. MINDVIEW, INC. POSEBNO NE DAJE
NIKAKVU GARANCIJU, UKLJUČUJUĆI I IMPLICITNU GARANCIJU O PRODAJI I POGODNOSTI
ZA ODREĐENU UPOTREBU ALI NE OGRANIČAVAJUĆI SE NA TO. IZVORNI KÔD JE DOSTUPAN U
OBLIKU U KOJEM JE DAT I UZ NJEGA SE NE DOBIJA NIKAKVA USLUGA OD MINDVIEW, INC,
KOJI NEMA NITI PREUZIMA BILO KAKVE OBAVEZE ZA PRUŽANJE USLUGA, PODRŠKE,
AŽURIRANJA, POBOLJŠANJA ILI MODIFIKACIJA.
Molim, imajte u vidu da MindView, Inc. održava Web lokaciju http://www.Mind-
View.net (i njene zvanične duplikate). Jedino se odatle mogu preuzeti elek-
tronske kopije Izvornog koda, koje se pod prethodno navedenim uslovima mogu
preuzeti besplatno.

Ako mislite da ste u Izvornom kodu pronašli grešku, molim vas da ispravku
pošaljete preko sistema povratnih informacija koji možete pronaći na lokaciji
www.MindView.com.
///:~
```

Kôd možete koristiti u svojim nastavnim projektima (uključujući i vaše materijale za prezentaciju), dokle god zadržavate poruku o autorskim pravima koja se pojavljuje u svakoj izvornoj datoteci.

Način pisanja korišćen u knjizi

U ovoj knjizi identifikatori (metode, promenljive i imena klasa) ispisani su **polucrno**. Većina rezervisanih reči takođe je napisana polucrno, osim onih koje se toliko koriste da bi njihovo naglašavanje postalo zamorno.

Za pisanje primera u knjizi koristim određen stil. On odgovara stilu koji se koristi u kompaniji Sun za praktično sve programe koje možete naći na njihovoj Web lokaciji (pogledajte *http://java.sun.com/docs/codeconv/index.html*) i koji podržava većina razvojnih okruženja za Javu. Ako ste čitali moje druge radove, primetili ste takođe da se stil pisanja programa koji koristi Sun podudara s mojim stilom. Ovo mi je drago, iako ja s time (koliko znam) nisam imao nikakve veze. Pitanje stila može biti predmet višečasovnih rasprava, pa ću samo reći da kroz moje primere ne želim da namećem pravilan stil, nego imam lične razloge za korišćenje takvog stila. Pošto je Java programski jezik slobodne forme, možete da koristite bilo koji stil koji vam odgovara. Da biste formatiranje doterali kako vam odgovara i rešili pitanje stila, možete se poslužiti alatkom kao što je *Jalopy (www.triemax.com)* – dok sam pisao knjigu, koristio sam je.

Programi u ovoj knjizi ubačeni su u tekst direktno iz datoteka koje su prevođene i ispitivane na jednom automatskom sistemu. Stoga bi izvorni kôd štampan u knjizi trebalo da radi bez grešaka pri prevođenju.

Ova knjiga se zasniva na Javi SE5/6, i na njoj je i testirana. Ako hoćete da naučite nešto o ranijim varijantama jezika, a to nije obuhvaćeno ovim izdanjem, prvo, drugo i treće izdanje knjige (na engleskom) možete besplatno preuzeti s lokacije *www.MindView.net*.

Greške

Bez obzira na to koliko trikova pisac koristi da otkrije greške, neke se uvek provuku, a nov čitalac ih često odmah primeti. Ako otkrijete bilo šta što smatrate greškom, molim vas da iskoristite hipervezu za ovu knjigu na lokaciji *www.MindView.net*, prijavite grešku i pošaljete predlog za ispravku. Ceniću vašu pomoć.

Upoznavanje sa objektima

„Prirodu parcelišemo i razvrstavamo u koncepte kojima pripisujemo značenja, uglavnom zato što se držimo dogovora koji važi u našoj jezičkoj zajednici i kodifikovan je u obrascima našeg jezika ... uopšte ne možemo govoriti ukoliko se ne pridržavamo organizacije i klasifikacije podataka koje taj dogovor propisuje." Benjamin Lee Whorf (1897–1941)

RAČUNARSKA REVOLUCIJA JE NASTALA U MAŠINI. NAŠI PROGRAMSKI JEZICI STOGA TEŽE DA izgledaju kao ta mašina.

Ali računari nisu toliko mašine koliko pojačala uma („točkovi za um", kako Steve Jobs voli da kaže) i drugi način izražavanja. Zbog toga ti alati sve manje liče na mašine, a sve više na delove našeg uma, i na druge oblike izražavanja, kakvi su pisanje, slikanje, vajanje, animiranje i snimanje filmova. Objektno orijentisano programiranje (OOP) deo je ovog pomeranja ka korišćenju računara kao sredstva izražavanja.

Ovo poglavlje vas uvodi u osnovne koncepte OOP-a, uključivši i pregled metoda razvoja, uz pretpostavku da imate iskustva s programiranjem, iako to ne mora biti na jeziku C. Ako mislite da treba da se još pripremate za programiranje pre nego što se uhvatite u koštac sa ovom knjigom, proučite multimedijski seminar *Thinking in C*, koji možete preuzeti na adresi *www.MindView.net*.

Ovo poglavlje je i podloga i dodatni materijal. Mnogi se ne osećaju prijatno u svetu objektno orijentisanog programiranja ako pre toga ne razumeju celinu. Stoga je ovde dat bogat pregled koncepata OOP-a. Drugi ne mogu da shvate glavne principe dok prvo ne upoznaju barem neke mehanizme. Ako pripadate toj grupi i željni ste da otkrijete specifičnosti jezika, slobodno preskočite ovo poglavlje – u ovom trenutku to vas neće sprečiti da pišete programe ili naučite jezik. Međutim, poželežćete da se vratite na ovaj deo knjige da biste dopunili svoje znanje i shvatili zašto su objekti važni i kako da ih koristite pri pisanju programa.

Razvoj apstrakcije

Svi programski jezici obezbeđuju apstrakciju. Moglo bi se raspravljati o tome da li je složenost problema koje ste u stanju da rešite direktno povezana s vrstom i kvalitetom apstrakcije. Pod „vrstom" mislim na to „šta apstrahujete". Mašinski jezik je mala apstrakcija mašine na kojoj se programi izvršavaju. Mnogi takozvani „proceduralni" jezici koji su sledili mašinski (kao FORTRAN, BASIC i C) bili su apstrakcija mašinskog jezika. Ovi jezici su veliki napredak u odnosu na mašinski jezik, ali njihova primarna apstrakcija ipak zahteva od vas da razmišljate iz ugla strukture računara umesto iz ugla problema koji rešavate. Programer mora da uspostavi vezu između modela mašine (u „prostoru rešenja" koji predstavlja mesto gde realizujete rešenje problema, na primer, u računaru) i modela problema koji se rešava (u „prostoru problema" koji predstavlja mesto gde problem postoji, recimo poslovanje). Napor koji iziskuje ovo preslikavanje i činjenica da je ono nebitno za programski jezik proizvode programe koji se teško pišu i čije je održavanje skupo, a kao sporedni efekat nastaje celokupna industrija „programskih metoda".

Alternativa modelovanju mašine je modelovanje problema koji pokušavate da rešite. Rani jezici kao LISP i APL odražavali su pojedine predstave o svetu („Svi problemi se na kraju svode na liste" ili „Svi problemi su algoritamske prirode"). Prolog prebacuje sve probleme u korake odlučivanja. Stvoreni su jezici zasnovani na ograničenom programiranju i programiranju isključivo manipulacijom grafičkim simbolima (što se pokazalo kao previše restriktivno). Svaki ovaj pristup može biti dobro rešenje za određenu klasu problema kojoj su namenjeni, ali kada istupite iz tog domena, oni postaju nezgrapni.

Objektno orijentisani pristup ide korak dalje, obezbeđujući alate pomoću kojih programer predstavlja elemente u prostoru problema. Ovo predstavljanje u principu ne ograničava programera na jednu vrstu problema. Elemente u prostoru problema i njihovo predstavljanje u prostoru rešenja nazivamo „objekti". (Trebaće vam i drugi objekti koji nemaju svoj par u prostoru problema.) Ideja je da se programu dozvoli da se prilagodi nerazumljivom jeziku problema tako što će se dodati novi tipovi objekata, te kada čitate kôd koji opisuje rešenje, u isto vreme čitate i reči koje izražavaju problem. Ovo je mnogo fleksibilnija i snažnija apstrakcija od prethodne.[1] Stoga OOP dozvoljava da opišete problem iz ugla problema, umesto iz ugla računara na kome će se to rešenje izvršavati. Još uvek postoji povratna veza ka računaru: svaki objekat izgleda posve kao mali računar – on ima unutrašnje stanje i operacije koje možete zahtevati da izvrši. Međutim, ovo i nije tako loša analogija sa objektima u stvarnom svetu – svi imaju svoje karakteristike i osobeno se ponašaju.

Alan Kay je naveo pet osnovnih obeležja Smalltalka, prvog uspešnog objektno orijentisanog jezika, i jednog od jezika na kome je Java zasnovana. Ta obeležja predstavljaju čist i neiskvaren pristup objektno orijentisanom programiranju.

1. **Sve je objekat.** Posmatrajte objekat kao poboljšanu promenljivu; on čuva podatke, ali možete i da mu „postavite zahteve" koje ispunjava vršeći operacije nad tim podacima. Teoretski, možete uzeti bilo koju idejnu komponentu problema koji rešavate (pse, zgrade, usluge itd.) i predstaviti je kao objekat u svom programu.
2. **Program je skup objekata koji jedni drugima porukama saopštavaju šta da rade.** Da biste uputili zahtev objektu, vi „šaljete poruku" tom objektu. Konkretnije, možete zamisliti da je poruka zahtev da se pozove metoda koja pripada određenom objektu.
3. **Svaki objekat ima svoj memorijski prostor koji se sastoji od drugih objekata.** Drugačije rečeno, vi stvarate novu vrstu objekta praveći paket koji sadrži neke postojeće objekte. Stoga možete da usložnjavate program koji će biti skriven iza jednostavnih objekata.
4. **Svaki objekat ima tip.** Stručno rečeno, svaki objekat je *instanca (primerak)* neke *klase*, pri čemu su „klasa" i „tip" sinonimi. Najvažnija odlika klase glasi: „Koje poruke joj možete poslati?"

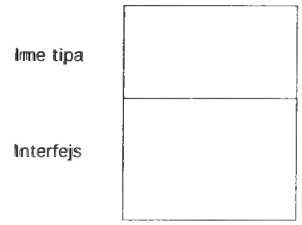
[1] Neki autori programskih jezika su smatrali da objektno orijentisano programiranje nije dovoljno da omogući lako rešavanje svih programskih problema, pa su podržavaliu kombinovanje različitih pristupa kroz *multistandardne* programske jezike. Pogledajte *Multiparadigm Programming in Leda*, Timothy Budd (Addison-Wesley 1995).

Pošto klasa opisuje skup objekata koji imaju identične karakteristike (elementi s podacima) i jednako se ponašaju (funkcionalnost), klasa je zaista tip podatka, jer i broj u formatu pokretnog zareza, na primer, takođe ima skup karakteristika i ponašanja. Razlika je u tome što programer definiše klasu koja odgovara problemu, umesto da bude prinuđen da koristi postojeći tip koji predstavlja memorijsku jedinicu unutar računara. Programski jezik proširujete dodavanjem novih tipova podatka, specifičnih za vaše potrebe. Programski sistem prihvata nove klase, o njima vodi računa i proverava tipove, kao što radi i sa ugrađenim tipovima.

Objektno orijentisan pristup ne ograničava se samo na pravljenje simulacija. Bez obzira na to da li se slažete sa stavom da je svaki program simulacija sistema koji projektujete, upotrebom tehnika, OOP-a veliki skup problema lako se može da svesti na jednostavna rešenja.

Kada se klasa ustanovi, možete da napravite koliko god želite objekata te klase, a zatim da radite s tim objektima kao da su elementi u problema koji pokušavate da rešite. Zaista, jedan od izazova objektno orijentisanog programiranja jeste ostvarivanje jednoznačnog preslikavanja između elemenata u prostoru problema i objekata u prostoru rešenja.

Kako da naterate objekat da uradi koristan posao? Mora postojati način da postavite zahtev objektu da nešto uradi, na primer, da završi transakciju, iscrta nešto na ekranu ili uključi prekidač. A svaki objekat može da zadovolji samo izvesne zahteve. Zahtevi koje možete da postavite objektu definisani su preko njegovog *interfejsa*, koji je određen tipom. Prost primer bi mogla biti sijalica:




```
Sijalica sj = new Sijalica();
sj.ukljuci();
```

Interfejs određuje koje zahteve možete da postavite određenom objektu. Naravno, ne gde mora postojati kod koji će zadovoljiti taj zahtev. Taj kod, zajedno sa skrivenim podacima, čini *implementaciju* (realizaciju ili primenu interfejsa). S tačke gledišta proceduralnog programiranja to i nije tako komplikovano. Tip uz svaki mogući zahtev ima pridruženu metodu i kada napravite određeni zahtev, poziva se odgovarajuća metoda. Za ovaj proces obično se kaže da smo „poslali poruku" (postavili zahtev) objektu, a objekat određuje šta će s porukom da uradi (izvrši kod).

U našem primeru, ime tipa/klase je **Sijalica**, ime posebnog objekta tipa **Sijalica** je **sj**, a zahtevi koje možemo postaviti objektu **Sijalica** su uključi se, isključi se, pojačaj svetlo ili priguši svetlo. Objekat tipa **Sijalica** pravite kada definišete *njereniu* (sj) na dati objekat i

pozovete **new**, čime zahtevate novi objekat tog tipa. Da biste poslali poruku objektu, navedite ime objekta i povežite ga s porukom – zahtevom, razdvajajući ih tačkom. S tačke gledišta korisnika unapred definisane klase, to je skoro sve što vam treba da biste programirali sa objektima.

Gornja slika je napravljena u skladu s formatom koji koristi *unifikovani jezik za modelovanje* (engl. *Unified Modeling Language, UML*). Svaka klasa je predstavljena pravougaonikom, ime tipa je u gornjem delu, *podaci članovi* koje treba opisati nalaze se u srednjem delu, a u donjem delu pravougaonika su *metode* (funkcije koje pripadaju tom objektu, koje primaju poruke koje šaljete tom objektu). Često se u UML dijagramima prikazuju samo ime klase i javne metode, a srednji deo ne – tako je i na prethodnoj slici. Ako vas zanima samo ime klase, nema potrebe da prikazujete ni donji deo.

Objekat pruža usluge

Dok pokušavate da razvijete ili shvatite strukturu nekog programa, bilo bi dobro da objekte smatrate „davaocima usluga". I vaš program će pružati usluge korisniku, i to pomoću usluga koje pružaju drugi objekti. Vaš cilj je da napravite (ili još bolje, pronađete u bibliotekama koda) skup objekata koji pružaju idealne usluge koje rešavaju vaš problem.

To biste mogli postići ako se zapitate: „Da ih mogu čarobnim štapićem izvući iz šešira, koji objekti bi odmah rešili moj problem?" Na primer, pretpostavimo da pišete program za knjigovodstvo. Mogli biste zamisliti određene objekte koji sadrže unapred definisane slike ekrana za unos podataka, drugi skup objekata koji obavljaju knjigovodstvene proračune, i objekat koji štampa čekove i račune na svim mogućim vrstama štampača. Neki od tih objekata možda već postoje? Kako bi izgledali oni koji ne postoje? Koje usluge bi *ti* objekti davali i koji bi im objekti bili potrebni za izvršavanje zadataka? Ako tako budete radili, doći ćete do tačke kada možete reći: „Ovaj objekat izgleda dovoljno jednostavno da se može napisati" ili „Mora da postoji ovakav objekat". To je racionalan način razlaganja problema na skup objekata.

Kada se objekat smatra davaocem usluga, stiče se još jedna prednost: time se povećava uskladenost objekta. *Velika uskladenost* je jedan od temeljnih kvaliteta projektovanja softvera: to znači da su razni aspekti softverske komponente (kao što je objekat, iako isto važi i za metodu ili biblioteku objekata) međusobno „dobro uklopljeni". Prilikom projektovanja objekata, programeri često trpaju previše funkcionalnosti u jedan objekat. Kada se radi o pomenutom modulu za štampanje čekova, možete odlučiti da vam treba objekat koji zna sve o formatiranju i štampanju. Verovatno će vas iskustvo naučiti da je to previše za jedan objekat i da vam treba tri ili više objekata. Jedan objekat bi mogao biti katalog svih mogućih izgleda čekova, kojem se mogu slati upiti radi podataka o tome kako odštampati određeni ček. Jedan objekat ili skup objekata može biti opšti (generički) interfejs za štampanje, koji zna sve o raznim vrstama štampača (ali ništa o knjigovodstvu – taj je kandidat za kupovinu, umesto da ga sami pišete). A treći objekat može koristiti usluge prethodna dva da bi obavio posao. Tako bi svaki objekat imao uskladen skup usluga koje pruža. U dobrom objektno orijentisanom dizajnu, svaki objekat dobro radi jedan posao, ali ne pokušava da radi više poslova. Ne samo da se tako mogu pronaći objekti koje treba kupiti (recimo, objekat interfejsa štampača), nego se proizvode i novi objekti koji se mogu ponovo upotrebljavati na drugim mestima (katalog izgleda čekova).

Jako ćete pojednostaviti sebi život ako objekte budete smatrali davaocima usluga. To će koristiti ne samo vama tokom procesa projektovanja, nego i drugima koji budu pokušavali da razumeju vaš kôd ili upotrebe neki od vaših objekata. Ako budu mogli da utvrde vrednost objekta na osnovu usluge koje on pruža, biće im mnogo lakše da ga uklope u svoje projekte.

Skrivena realizacija

Vrlo je korisno podeliti polje delatnosti na *autore klasa* (one koji prave nove tipove podataka) i *programere klijente*[4] (korisnike klasa koji te tipove upotrebljavaju u svojim aplikacijama). Cilj programera klijenta je da sakupi klase u „kutiju sa alatom", koju će koristiti za brzi razvoj aplikacija. Cilj autora klasa je da napravi klasu koja otkriva samo ono što je neophodno programeru klijentu, a sve ostalo drži sakriveno. Zašto? Programer klijent ne može da koristi sakrivene delove, što znači da autor klase može da promeni skriveni deo kad god hoće, ne razmišljajući da li će se to odraziti na ostale. Skriveni deo obično predstavlja osetljivu unutrašnjost objekta koju lako može da ošteti neobazriv ili neobavešten programer klijent, pa sakrivanje realizacije smanjuje mogućnost da se pojave greške u programima.

U svakom odnosu je važno da sve uključene strane poštuju granice. Kada pravite biblioteku, vi stvarate odnos sa klijentom koji je takođe programer, ali koji sastavlja aplikaciju koristeći vašu biblioteku, da bi, možda, napravio veću biblioteku.

Ako bi svi članovi klase bili dostupni svakome, onda bi programer klijent mogao da uradi bilo šta s klasom i ne bi postojao način da se nametnu pravila. Iako biste vi voleli da programer klijent ne radi direktno s nekim članicama vaše klase, bez kontrole pristupa ne bi postojao način da to sprečite. Sve bi bilo izloženo javnosti.

Stoga je prvi razlog za uvođenje kontrole pristupa onemogućavanje programera klijenta da pristupa delovima koje ne sme da dira, a koji su neophodni za interni rad s tipom podataka. Ti delovi nisu deo interfejsa koji je potreban korisnicima za rešavanje njihovih problema. Ovo je u isto vreme usluga programerima klijentima, jer lako mogu da razluče šta je za njih važno, a o čemu ne treba da misle.

Kontrolu pristupa treba uvesti i da bi se dozvolilo projektantu biblioteke da promeni način na koji klasa interno radi, a da ne mora da brine kako će se to odraziti na programere klijente. Na primer, možete realizovati neku klasu na jednostavan način, a kasnije otkriti da je treba ponovo napisati kako bi radila brže. Ako su interfejs i realizacija jasno razdvojeni i zaštićeni, ovo možete lako da izvedete.

Java koristi tri rezervisane reči da postavi granice unutar klase: **public**, **private** i **protected**. Ovi *specifikatori pristupa* određuju ko može da koristi definicije koje sledе iza njih. **public** (javni) znači da je naredni element dostupan svakome. Rezervisana reč **private** (privatni) znači da tom elementu ne može da pristupi niko sem autora klase, unutar metoda tog tipa. **private** predstavlja zid između autora i programera klijenta. Onaj ko pokuša da pristupi članici označenoj kao **private**, izazvaće grešku prilikom prevođenja.

[4] Zahvaljujem svom prijatelju Scottu Meyersu na ovom terminu.

Rezervisana reč **protected** (zaštićeni) deluje kao **private**, s tom razlikom što klase naslednice imaju pristup **zaštićenim** članicama, ali ne i **privatnim** članicama. O nasleđivanju ćemo govoriti uskoro.

Java ima i „podrazumevani" pristup koji se koristi ako ne zadate nijedan od navedenih specifikatora. Ovo se obično naziva *paketski pristup*, jer klase mogu da pristupe članicama drugih klasa u istom paketu, ali izvan tog paketa te iste članice vide se kao **privatne**.

Ponovno korišćenje realizacije

Kada se klasa napravi i testira, ona bi (u idealnom slučaju) trebalo da predstavlja korisnu jedinicu koda. Ispostavlja se da ponovno korišćenje ni izbliza nije tako lako ostvarljivo kako se mnogi nadaju; potrebno je iskustvo i pronicljivost da bi se napisao višekratno upotrebljiv objekat. Ali kada ga napišete, on moli da bude ponovo iskorišćen. Ponovno korišćenje koda je jedna od najvećih prednosti objektno orijentisanih programskih jezika.

Najjednostavniji način da ponovo iskoristite klasu jeste da direktno koristite objekat te klase; međutim, objekat te klase možete da stavite i u novu klasu. Ovo se naziva „pravljenje objekta člana" (engl. *member object*). Vaša nova klasa može biti sastavljena od ma koliko drugih objekata, ma kojeg tipa i u bilo kojoj kombinaciji koja vam je potrebna da biste postigli željenu funkcionalnost svoje nove klase. Novu klasu sastavljate od postojećih klasa, što se naziva *kompozicija* (ako se dešava dinamički, onda obično *agregacija*). Kompozicija se često poredi s relacijom „ima", na primer „auto ima motor".

(Na ovom UML dijagramu kompoziciju označava popunjen romb, koji kazuje da postoji jedan auto. Kad označavam spajanje, obično ću koristiti jednostavniji oblik: samo liniju, bez romba.)[5]

Kompozicija je vrlo fleksibilna. Objekti članovi vaše nove klase obično su privatni, što ih čini nedostupnim programerima klijentima koji koriste klasu. Ovo omogućava da izmenite te članove ne remeteći postojeći klijentski kôd. Možete menjati objekte članove i u vreme izvršavanja da biste dinamički menjali ponašanje svog programa. Nasleđivanje koje malo dalje opisujemo, nema ovu fleksibilnost pošto prevodilac mora da ugradi ograničenja za prevođenje u klase stvorene nasleđivanjem.

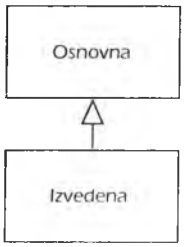
Pošto je nasleđivanje veoma bitno, u objektno orijentisanom programiranju često se veoma naglašava, i nov programer može pomisliti da nasleđivanje treba koristiti svuda. Kao rezultat može se dobiti veoma nezgrapan i veoma složen program. Umesto toga, prvo treba videti da li pri pravljenju novih klasa može da se iskoristi kompozicija, pošto je ona jednostavnija i fleksibilnija. Ako primenite ovaj pristup, program će biti čistiji. Kada steknete nešto iskustva, biće vam očigledno kada treba da koristite nasleđivanje.

[5] Ovo je u principu dovoljno detaljno za većinu dijagrama i ne treba da precizirate da li koristite agregaciju ili kompoziciju

Nasleđivanje

Sam po sebi, koncept objekta je veoma koristan. On omogućava da podatke i funkcionalnost grupišete po *konceptu*, tako da možete predstaviti odgovarajuću ideju u prostoru problema, umesto da budete prinuđeni da koristite izraze računara na kome radite. Kada se primeni rezervisana reč **class**, te ideje su izražene kao osnovne jedinice u ovom programskom jeziku.

Bila bi šteta da se namučite i napravite neku klasu a da onda budete primorani da pravite potpuno novu klasu koja ima sličnu funkcionalnost. Bilo bi jednostavnije da uzmemo postojeću klasu, da je kloniramo, a zatim dopunjujemo i menjamo kloniranu klasu. Ovo se zapravo postiže *nasleđivanjem* (engl. *inheritance*), osim kada se originalna klasa (koja se naziva *osnovna* klasa ili *natklasa* ili *klasa roditelj*) izmeni, a te izmene se odraze i na „klonu" (koji se naziva *izvedena* ili *nasleđena* klasa ili *potklasa* ili klasa *naslednik*; engl. *derived class*).



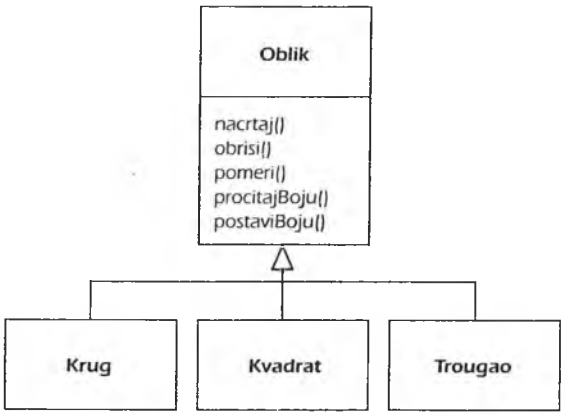
(Strelica u ovom UML dijagramu polazi od izvedene klase ka osnovnoj klasi. Kao što ćete videti, može da postoji više izvedenih klasa.)

Tip ne označava samo ograničenja skupa objekata; on ima odnose i s drugim tipovima. Dva tipa mogu imati neke zajedničke karakteristike i jednako se ponašati, ali pri tom jedan tip može imati još neke dodatne karakteristike i može obrađivati više poruka (ili ih drugačije obrađivati). Kod nasleđivanja, ova sličnost između tipova se izražava preko koncepta osnovnih i izvedenih tipova. Osnovni tip sadrži sve karakteristike i ponašanja koja su zajednička za tipove izvedene iz njega. Praveći osnovni tip, izražavate suštinu svojih ideja o nekim objektima u sistemu. Iz osnovnog tipa izvodite ostale tipove i pokazujete različite načine realizacije ove suštine.

Na primer, mašina za recikliranje raspoređuje komadiće otpada. Osnovni tip je „otpad", a svaki pojedini otpadak ima težinu, vrednost itd. i može biti isečen, istopljen ili rastavljen. Iz osnovnog tipa izvodimo posebne vrste otpada s dodatnim karakteristikama (boca ima boju), ili ponašanjima (aluminijumska konzerva može da se zdrobi, čelične konzerve privlači magnet). Ponašanja mogu biti različita (vrednost papira zavisi od njegove vrste i stanja). Koristeći nasleđivanje možete izgraditi hijerarhiju tipova. Ta hijerarhija opisuje problem koji pokušavate da rešite preko tipova koji se u problemu pojavljuju.

Drugi primer je klasičan „oblik" koji može da se koristi u CAD sistemu (engl. *Computer-Aided Design*) ili u nekoj kompjuterskoj igri. Osnovni tip je „oblik" a svaki oblik ima veličinu, boju, poziciju itd. Svaki oblik može da se iscrta, obriše, pomera, oboji itd. Odavde izvodimo (nasleđujemo) pojedine vrste oblika – krug, kvadrat, trougao i druge – od

kojih svaki može imati dodatne karakteristike i ponašanje. Neki oblici, na primer, mogu da se okreću. Neka ponašanja mogu biti drugačija, recimo kada želite da izračunate površinu oblika. Hijerarhija tipova objedinjuje sličnosti i razlike između oblika.

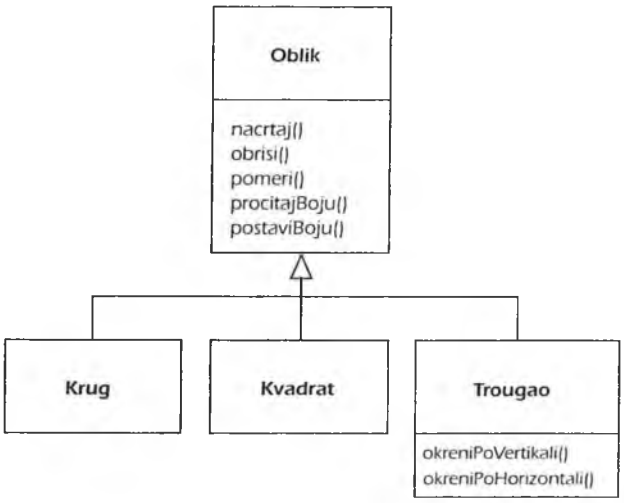


Predstavljanje rešenja i problema u istom obliku veoma je korisno jer vam ne treba veliki broj međumodela da biste od opisa problema došli do opisa rešenja. Kod objekata, hijerarhija tipova je osnovni model tako da sa opisa sistema u realnom svetu direktno prelazite na opis sistema kodom. Jedna od teškoća sa objektno orijentisanim programiranjem jeste ta da je previše jednostavno stići od početka od kraja. Um treniran da traži složena rešenja često spočetka zbunjuje ova jednostavnost.

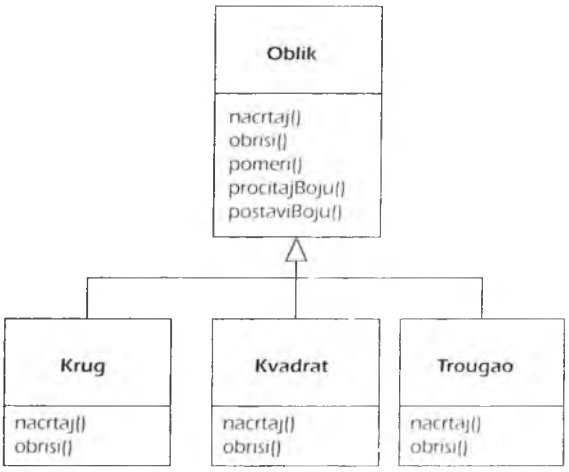
Kada iskoristite nasleđivanje iz postojećeg tipa, stvarate novi tip. Taj novi tip ne samo da sadrži sve članice postojećeg tipa (iako su **privatne** članice sakrivene i nedostupne), već, što je mnogo važnije, kopira interfejs osnovne klase. Znači, sve poruke koje možete da pošaljete objektima osnovne klase, takođe možete poslati i objektima izvedene klase. Pošto se tip klase određuje na osnovu poruka koje joj možemo poslati, to znači da je izvedena klasa *istog tipa kao* i *osnovna klasa*. U prethodnom primeru „krug je oblik". Ekvivalencija tipova dobijena nasleđivanjem je osnovni korak na putu ka razumevanju smisla objektno orijentisanog programiranja.

Pošto i osnovna klasa i izvedena klasa imaju isti osnovni interfejs, mora da postoji realizacija koja ide uz taj interfejs. Znači, mora postojati kôd koji se izvršava kada objekat primi određenu poruku. Ako nasledite klasu i ništa više ne uradite, metode interfejsa osnovne klase će preći i u izvedenu klasu. To znači da objekti izvedene klase imaju i isti tip i isto ponašanje, što i nije previše korisno.

Postoje dva načina da napravite razliku između svoje nove izvedene klase i osnovne klase. Prvi je jasan: izvedenoj klasi dodate potpuno nove metode. Te nove metode nisu deo interfejsa osnovne klase, što znači da osnovna klasa nije radila sve ono što ste hteli, pa ste joj zato dodali još metoda. Ovaj jednostavan način korišćenja nasleđivanja ponekad je savršeno rešenje problema. Međutim, pažljivo proučite da li su i vašoj osnovnoj klasi potrebne te dodatne metode. Ovaj proces otkrivanja i iteracije programa redovna je pojava u objektno orijentisanom programiranju.



Iako nasleđivanje ponekad može da nagovesti (posebno u Javi, gde rezervisana reč koja označava nasleđivanje glasi **extends** – proširuje) kako ćete dodavati nove metode interfejsu, to nije uvek istina. Drugi i važniji način da napravite razliku u novoj klasi jeste da *promenite* ponašanje postojeće metode osnovne klase. To se naziva *redefinisanje* metode (engl. *overriding*).

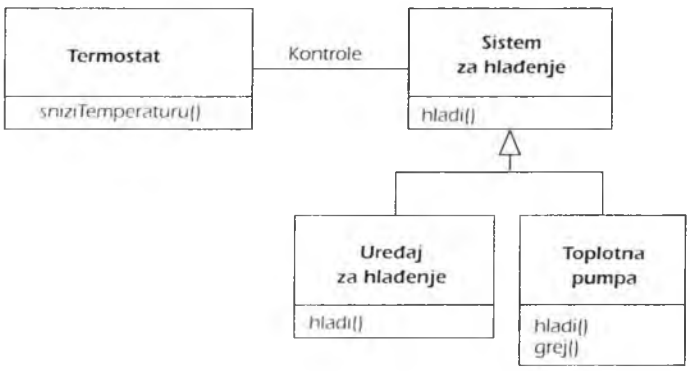


Da biste redefinisali metodu, napravite novu definiciju te metode u izvedenoj klasi. Vi kažete: „Koristim ovde istu metodu interfejsa, ali želim da ona u mom novom tipu radi nešto drugo."

Relacije JE i JE-KAO

Može se postaviti jedno pitanje o nasleđivanju: „Da li nasleđivanje treba da redefiniše *samo* metode osnovne klase (i da ne dodaje nove metode koje ne postoje u osnovnoj klasi)?" To bi značilo da je izvedena klasa *potpuno istog* tipa kao osnovna klasa, jer ima istovetan interfejs. Kao rezultat, objekat izvedene klase može se svesti na tip objekta osnovne klase. Ovo je tzv. *čista supstitucija*, ili *princip supstitucije*. Unekoliko, ovo je idealan način primene nasleđivanja. U ovom slučaju, mi se često pozivamo na relaciju između osnovne i izvedene klase kao na relaciju *je*, jer možemo reći „krug *je* oblik". Pokušaj da utvrdimo da li možemo da uspostavimo relaciju *je* između klasa, a da to ima smisla, predstavlja svojevrstan test nasleđivanja.

Ponekad morate da dodate nove elemente u interfejs izvedenog tipa i time proširite postojeći interfejs. Novi tip i dalje može biti sveden na osnovni tip, ali supstitucija nije savršena, jer nove metode nisu dostupne iz osnovnog tipa. Ovo se može opisati kao relacija *je-kao* (moj termin). Novi tip ima interfejs starog tipa ali sadrži i druge metode, tako da se ne može reći kako je potpuno isti. Uzmimo za primer uređaj za klimatizaciju. Pretpostavimo da su po kući razvedene kontrole za hlađenje, odnosno, kuća ima interfejs koji omogućava da kontrolišete hlađenje. Zamislite da se uređaj za klimatizaciju pokvari i da ga zamenite toplotnom pumpom koja može i da greje i da hladi. Toplotna pumpa *je kao* uređaj za hlađenje, ali ona može i više. Pošto je kontrolni sistem vaše kuće projektovan samo da kontroliše hlađenje, on je ograničen na komunikaciju s delom za hlađenje novog objekta. Interfejs novog objekta je proširen, a postojeći sistem ne poznaje ništa drugo osim originalnog interfejsa.



Čim proučite ovaj plan, postaće jasno da osnovna klasa „sistem za hlađenje" nije dovoljno opšta i da treba da se preimenuje u „sistem za kontrolu temperature", kako bi mogla da sadrži i grejanje – nakon čega bi princip supstitucije važio. Ovaj dijagram prikazuje šta u stvarnom svetu može da se desi tokom projektovanja.

Kada upoznate princip supstitucije, lako možete pomisliti da je taj pristup (čista supstitucija) jedini način da se nešto uradi, i zaista *jeste* dobro da program napravite na taj način. Ali shvatićete da u interfejs izvedene klase ponekad morate da dodate nove metode. Nakon što ispitate problem, trebalo bi da bude prilično očigledno o kojem se od ta dva slučaja radi.

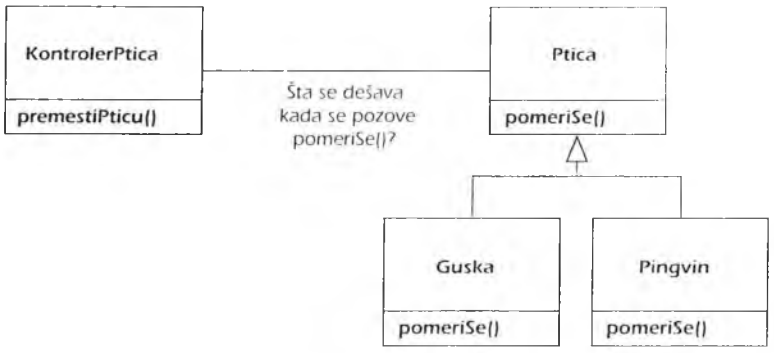
Virtuelizacija objekata preko polimorfizma

Ukoliko imate posla s hijerarhijom tipova, često objekat ne tretirate kao specifičan tip, već kao osnovni tip. To vam omogućava da pišete kôd koji ne zavisi od specifičnog tipa. U prethodnom primeru sa oblicima, metode manipulišu generičkim tipovima (oblicima) bez obzira na to da li se radi o krugovima, kvadratima, trouglovima ili čak oblicima koji još nisu definisani. Svi oblici mogu biti iscrtani, obrisani i pomerani, tako da te metode samo pošalju poruku objektu tipa oblik i ne brinu kako će objekat izaći na kraj s tom porukom.

Takav kôd se ne menja ni nakon dodavanja novih tipova, a dodavanjem novih tipova najčešće se proširuju objektno orijentisani programi kada treba da se savlada neka nova situacija. Na primer, možete da izvedete novi podtip oblika, pod imenom petougao, bez menjanja metoda koje rade samo s generičkim oblicima. Mogućnost da se program lako proširi izvođenjem novih podtipova jeste jedan od osnovnih načina da se kapsuliraju promene. Time se znatno unapređuje program i istovremeno smanjuju troškovi održavanja softvera.

Problem nastaje pri pokušaju da se objekti izvedenog tipa tretiraju kao generički osnovni tipovi (krugovi kao oblici, bicikli kao vozila, kormorani kao ptice itd.). Ako metoda naredi generičkom obliku da se iscrta ili generičkom vozilu da smota volan ili generičkoj ptici da se pomeri, prevodilac prilikom prevođenja ne može tačno da zna koji deo koda će biti izvršen. To i jeste poenta – kada je poruka poslata, programer ne *želi* da zna koji deo koda će biti izvršen. Metoda za crtanje može podjednako da se primeni na krug ili na kvadrat ili na trougao, a objekat će izvršiti odgovarajući kôd u zavisnosti od svog specifičnog tipa.

Ako ne morate znati koji će deo koda biti izvršen, dodajte nov podtip; kôd koji će taj novi tip izvršavati može biti drugačiji a da pri tom ne morate ništa da menjate u metodi koja ga poziva. Znači, prevodilac ne zna tačno koji će deo koda biti izvršen. Šta onda radi? Na primer, u sledećem dijagramu objekat **KontrolerPtica** radi samo s generičkim objektima tipa **Ptica** i ne zna kog su oni tipa. Ovo je pogodno iz perspektive **KontroleraPtica** jer ne treba pisati poseban kôd koji bi odredio s kojim se tačno tipom **Ptice** radi, ili kakvo je ponašanje te određene **Ptice**. Kako se događa da se, kada se pozove metoda **pomeriSe()** uz ignorisanje specifičnog tipa **Ptice**, primeni pravilno ponašanje (**Guska** hoda, leti ili pliva, a **Pingvin** hoda ili pliva)?



Odgovor leži u osnovnoj začkoljici objektno orijentisanog programiranja: prevodilac ne može da pozove funkciju na tradicionalan način. Prevodilac koji nije objektno orijentisan izaziva *rano vezivanje* (engl. *early binding*). Ovaj termin možda niste ranije čuli, jer o pozivanju funkcija niste razmišljali ni na koji drugi način. To znači da prevodilac generiše poziv funkcije određenog imena, a izvršni sistem kasnije razreši ovaj poziv ubacivanjem apsolutne adrese koda koji treba da se izvrši. U OOP-u, program ne može da odredi adresu koda sve do trenutka izvršavanja, tako da je neophodna neka druga šema kada se poruka šalje generičkom objektu.

Da bi rešili ovaj problem, objektno orijentisani jezici koriste koncept *kasnog vezivanja* (engl. *late binding*). Kada pošaljete poruku objektu, kôd koji se poziva ne biva određen sve do trenutka izvršavanja. Prevodilac ipak proverava da li telo specifične metode postoji te koji su tipovi argumenata i povratne vrednosti, ali ne zna koji će kôd izvršiti.

Da bi razrešila kasno povezivanje, umesto apsolutnog poziva Java koristi specijalan delić koda koji adresu tela metode izračunava pomoću informacija ugrađenih u konkretan objekat. (Ovaj proces je detaljno opisan u poglavlju o polimorfizmu.) Tako svaki objekat može da se ponaša različito, u skladu sa sadržajem svakog pojedinog specijalnog delića koda. Kada pošaljete poruku objektu, on sam odlučuje šta će s njom da uradi.

U nekim jezicima morate izričito navesti da određenu metodu treba kasno povezati (u jeziku C++ to se radi pomoću rezervisane reči **virtual**). U tim jezicima, metode se podrazumevano *ne* povezuju dinamički. U Javi se dinamičko povezivanje podrazumeva, tako da ne morate dodavati posebne rezervisane reči da biste dobili polimorfizam.

Posmatrajmo primer oblika. Dijagram porodice klasa (koje su zasnovane na istom uniformnom interfejsu) dat je ranije u ovom poglavlju. Da bismo prikazali polimorfizam, napisaćemo naredbe koje ignorišu specifične detalje tipova i obraćaju se samo osnovnoj klasi. Te naredbe *nisu povezane* sa informacijama specifičnim za pojedine tipove, pa se stoga jednostavnije pišu i lakše razumevaju. Ako pomoću nasleđivanja dodamo nov tip – **Sestougaonik**, na primer – te naredbe radiće jednako dobro s novim tipom **Oblika** kao što su radile sa već postojećim tipovima. Znači, program je *proširiv*.

Ako napišete metodu u Javi (što ćete uskoro naučiti):

```
void radiNesto(Oblik oblik) {
oblik.obrisiSe();
// ...
oblik.iscrtajSe();
}
```

ta metoda se obraća bilo kom **Obliku**, pa ne zavisi od specifičnog tipa objekta koji se iscrtava i briše. Ako u nekom drugom delu programa pozovemo funkciju **radiNesto()**:

```
Krug krug = new Krug();
Trougao trougao = new Trougao();
Linija linija = new Linija();
radiNesto(krug);
radiNesto(trougao);
radiNesto(linija);
```

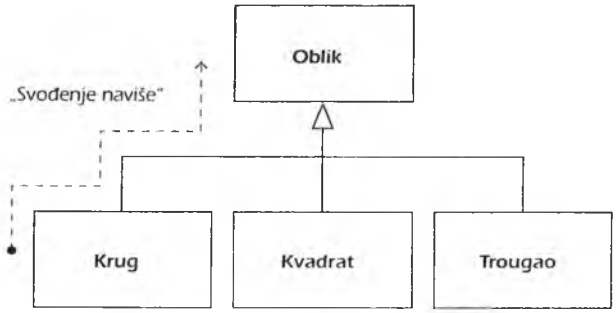
pozivi metode **radiNesto()** automatski rade ispravno, bez obzira na tačan tip objekta.

Ovo je prilično zadivljujući trik. Posmatrajte red:

```
radiNesto(krug);
```

Ovde je **Krug** prosleden metodi koja očekuje **Oblik**. Pošto **Krug** *jeste* **Oblik**, funkcija **radiNesto**() može tako da ga tretira. Odnosno, bilo koju poruku koju metoda **radiNesto**() može da pošalje **Obliku**, **Krug** može da prihvati. Zato je gornji poziv potpuno siguran i logičan.

Proces pri kome izvedeni tip tretiramo kao osnovni tip, nazivamo *svođenje naviše* (engl. *upcasting*). Ime *cast* je iskorišćeno zato što označava ubacivanje u kalup (engl. *casting into a mold* – ubacivanje u kalup), a *up* potiče od načina na koji se obično organizuje dijagram nasleđivanja, sa osnovnim tipom na vrhu i izvedenim klasama koje se lepezasto šire naniže. Prema tome, konverzija u osnovni tip je penjanje uz dijagram nasleđivanja: svođenje naviše ili *upcasting*.



U objektno orijentisanom programu uvek postoji svođenje naviše, jer pri tom ne morate znati tačan tip s kojim radite. Pogledajte funkciju **radiNesto**():

```
oblik.obrisiSe();
// ...
oblik.iscrtajSe();
```

Obratite pažnju na to da se nigde ne govori: „Ako si **Krug**, radi ovo, ako si **Kvadrat** radi ono itd." Tako napisan kôd koji proverava sve moguće tipove koje **Oblik** može da pokrije prljav je i morate da ga menjate svaki put kada dodate novu vrstu **Oblika**. U našem slučaju, kažemo: „Ti si oblik, ja znam da možeš da se iscrtaš i obrišeš, to i uradi, i sam vodi računa o detaljima".

Važno je da se naredbe u funkciji **radiNesto**() izvršavaju na pravi način. Sam poziv metode **iscrtajSe**() za **Krug** prouzrokuje izvršavanje drugačijeg koda nego kada pozivamo metodu **iscrtajSe**() za **Kvadrat** ili **Liniju**; ali kada poruku **iscrtajSe**() pošaljemo nepoznatom **Obliku**, dobija se ispravno ponašanje, zasnovano na stvarnom tipu **Oblika**. Ovo je dobra osobina, kao što je ranije pomenuto, jer kada prevodilac prevodi metodu **radiNesto**(), on ne zna tip s kojim radi. Stoga bismo obično očekivali da on pozove verziju metoda **obrisiSe**() i **iscrtajSe**() za osnovnu klasu **Oblik**, a ne za određeni **Krug**, **Kvadrat**

ili **Liniju**. Zbog polimorfizma se sve ipak ispravno odvija. Prevodilac i sistem za izvršavanje vode računa o svim detaljima. Zasad je dovoljno da znate da polimorfizam funkcioniše i kako da pišete programe koristeći taj pristup. Kada pošaljete poruku objektu, objekat će uraditi šta treba, čak i kada treba svoditi naviše.

Hijerarhija s jedinstvenim korenom

Jedno od pitanja u OOP-u koje je postalo veoma značajno od uvođenja C++-a glasi: da li sve klase treba da budu izvedene iz jedinstvene osnovne klase. U Javi (i gotovo svim ostalim OOP jezicima) odgovor je potvrdan. Ova osnovna klasa se naziva **Object**. Ispostavlja se da postoje brojne prednosti hijerarhije s jedinstvenim korenom.

Svi objekti u hijerarhiji s jedinstvenim korenom imaju zajednički interfejs i, prema tome, svi su istog osnovnog tipa. Postoji i druga mogućnost (koju nudi C++), da svi objekti ne budu istog osnovnog tipa. Po vertikalnoj kompatibilnosti ovo više odgovara modelu jezika C i može se smatrati manje restriktivnim; ali kada želite da se bavite objektno orijentisanim programiranjem u celini, morate napraviti svoju hijerarhiju da biste obezbedili istu fleksibilnost koja je već ugrađena u druge OOP jezike. Takođe, u svakoj novoj biblioteci klasa koju nabavite, biće korišćen neki drugi nekompatibilni interfejs. Treba uložiti napor (i verovatno višestruko nasleđivanje) da biste ugradili taj novi interfejs u svoj program. Da li je dodatna „fleksibilnost" C++-a vredna toga? Ako vam treba – zato što ste puno uložili u C – prilično je vredna. Ukoliko počinjete od nule, drugi jezici, poput Jave, često mogu biti mnogo produktivniji.

Svi objekti u hijerarhiji s jedinstvenim korenom (kakvu obezbeđuje Java) zasigurno imaju određenu zajedničku funkcionalnost – znate da možete da izvršite određene osnovne operacije nad svakim objektom u svom sistemu. Hijerarhija s jedinstvenim korenom, uz stvaranje dinamičkih objekata, veoma pojednostavljuje prosleđivanja argumenata.

Hijerarhija sa jedinstvenim korenom olakšava realizaciju *sakupljača smeća*, što je jedna od temeljnih prednosti Jave nad jezikom C++. Pošto je pri izvršavanju u svim objektima garantovano postojanje informacija o tipu, nikada nećete naići na objekat čiji tip ne možete da utvrdite. To je naročito važno kod operacija na sistemskom nivou, kao što je obrada izuzetaka, i obezbeđuje veću fleksibilnost pri programiranju.

Kontejneri

Po pravilu, ne možete znati koliko objekata će vam biti potrebno za rešavanje nekog zadatka, niti koliko dugo oni treba da postoje u memoriji. Ne znate ni kako da uskladištite te objekte. Ako pre izvršavanja programa ne znate ni broj objekata ni njihov vek trajanja, kako da odredite količinu memorijskog prostora za njihovo skladištenje?

Rešenje većine problema u objektno orijentisanom programiranju ponekad deluje neozbiljno: stvoriti još jedan tip objekta. Novi tip objekta koji rešava pomenuti problem čuva reference na druge objekte. Naravno, to isto možete da uradite i pomoću *nizova*, koji postoje u većini jezika. Ali taj novi objekat, obično pod imenom *kontejner* (naziva se i *kolekcija*, ali pošto Javine biblioteke koriste taj termin u drugom smislu, ova knjiga će se

držati imena „kontejner"), širiće se po potrebi da bi primio sve što u njega stavite. Stoga ne morate da znate koliko ćete objekata čuvati u kontejneru. Samo napravite kontejnerski objekat i prepustite njemu da se stara o detaljima.

Srećom, dobar OOP jezik sadrži skup kontejnera kao deo paketa. U jeziku C++ on je deo standardne C++ biblioteke i često se naziva standardna biblioteka šablona (engl. *Standard Template Library, STL*). Smalltalk ima prilično dobar skup kontejnera. I Java ima mnogo kontejnera u svojoj standardnoj biblioteci. U nekim bibliotekama jedan ili dva generička kontejnera dovoljni su za sve potrebe, dok u drugim bibliotekama (na primer u Javi) postoje različiti tipovi kontejnera za različite potrebe: više različitih klasa spiskova (tipova **Lista**) za čuvanje sekvenci, tipova **Map**a (koje neki nazivaju *asocijativni nizovi*) za pridruživanje jednih objekata drugima, tipova skupova (**Set**) za čuvanje po jednog primerka raznih tipova objekata i druge komponente među kojima su redovi čekanja, stabla, stekovi itd.

Sa stanovišta programa, za vas je važan kontejner s kojim možete da radite i rešite problem. Ako kontejner jednog tipa zadovoljava vaše potrebe, nema razloga da uvodite druge. Izbor treba da vam bude ponuđen iz dva razloga. Prvo, kontejneri obezbeđuju različite tipove interfejsa i načina rada. Stek ima interfejs i način rada različit od reda čekanja, koji se razlikuje od skupa ili liste. Za rešenje vašeg problema jedan od njih je obično bolji od ostalih. Drugo, različiti kontejneri obavljaju iste operacije s različitom efikasnošću. Na primer, postoje dve osnovne vrste lista: **ArrayList** i **LinkedList**. I jedna i druga su jednostavne sekvence koje mogu imati identične interfejse i, spolja gledano, načine rada. No, postoji bitna razlika u trajanju i zahtevima određenih operacija nad njima. Primera radi, nasumično pristupanje elementima kontejnera **ArrayList** je operacija s konstantnim vremenom izvršavanja; bez obzira na to koji element izaberemo, potrebno je isto vreme. Međutim, pomeranje kroz listu **LinkedList** do nasumično izabranog elementa je komplikovanije i treba više vremena da se pristupi elementima što su oni dalje od početka liste. S druge strane, ako želite da ubacite element usred sekvence, to je mnogo jednostavnije i brže u listi **LinkedList**, nego u **ArrayList**. Ove i druge operacije nisu jednako efikasne, što zavisi od strukture koja se nalazi u osnovi pojedine sekvence. U fazi pisanja programa, možete krenuti od kontejnera **LinkedList** a kada budete popravljali performanse, prebacite se na kontejner **ArrayList**. Zahvaljujući apstrakciji preko interfejsa **List**, prebacivanje iz jedne strukture u drugu imaće minimalan uticaj na vaš kod.

Parametrizovani (generički) tipovi

Pre Jave SE5, kontejneri su mogli da sadrže samo jedini univerzalni tip elemenata koji postoji u Javi: **Object**. Hijerarhija s jedinstvenim korenom znači da je sve tipa **Object**, tako da kontejner koji može da čuva **Object**, može da čuva bilo šta.[6] Ovo pojednostavljuje ponovno korišćenje kontejnera.

Da biste koristili takav kontejner, dodajte u njega reference na objekte, a kasnije ih tražite nazad. Međutim, pošto kontejner čuva samo tip **Object**, kada dodate referencu na svoj

[6] Ne može da čuva proste tipove, ali *automatsko pakovanje* (engl. *autoboxing*) koje je donela Java SE5 gotovo da potpuno uklanja to ograničenje. O tome će još biti reči u ovoj knjizi.

objekat u kontejner, ona se svodi naviše na **Object** i gubi identitet. Kada je izvadite iz kontejnera, dobijate referencu na **Object**, a ne referencu na tip koji ste stavili unutra. Kako onda da tu referencu vratite u nešto što ima konkretan tip objekta koji ste stavili u kontejner?

Ovde se ponovo koristi eksplicitna konverzija (pretvaranje) tipova, ali ovog puta se ne penjete uz hijerarhiju nasleđivanja do opštijeg tipa, već se spuštate niz hijerarhiju do određenijeg tipa. Ovaj način eksplicitne konverzije naziva se *svođenje naniže* (engl. *downcasting*). Pri svođenju naviše, na primer, znate da je **Krug** tipa **Oblik**, pa možete slobodno da izvršite svođenje naviše; međutim, neki **Object** nije obavezno **Krug** ili **Oblik**, pa svođenje naniže nije baš sigurno, osim ako znamo s čim imamo posla.

Ovo i nije previše opasno: ako izvršite pogrešno svođenje naniže, javlja se greška pri izvršavanju – *izuzetak* (engl. *exception*), o kome uskoro govorimo. Kada iz kontejnera uzimate reference, morate obezbediti neki način da zapamtite na šta se te reference odnose kako biste mogli da izvršite pravilno svođenje naniže.

Svođenje naniže i provere pri izvršavanju zahtevaju dodatno vreme za izvršavanje programa i dodatni napor za programera. Zar ne bi bilo logično da nekako napravimo kontejner tako da on zna koje tipove čuva, čime se isključuje potreba za svođenjem naniže kao i moguće greške. Ovo se rešava pomoću *parametrizovanih tipova*, a to su klase koje prevodilac automatski može da prilagodi tako da rade sa određenim tipom. Na primer, parametrizovani kontejner prevodilac bi mogao da prilagodi tako da prihvata i vraća samo klasu **Oblik**.

Jedna od velikih promena koje je donela Java SE5 jesu parametrizovani tipovi, koje u Javi nazivamo *generički tipovi*. Prepoznaćete ih po uglastim zagradama unutar kojih se navode; recimo, ovako se može napraviti **ArrayList** koji sadrži **Oblik**:

```
ArrayList<Oblik> oblici = new ArrayList<Oblik> ( );
```

Da bi se generički tipovi dobro iskoristili, izmenjene su i mnoge standardne komponente bibilioteke. Kao što ćete videti, generički tipovi su uticali na dobar deo koda u ovoj knjizi.

Pravljenje objekata i njihov životni vek

Jedan od najvažnijih činilaca jeste način na koji se objekti stvaraju i uništavaju. Da bi mogao da postoji, svakom objektu su neophodni neki resursi, prvenstveno memorija. Kada objekat više nije potreban, on mora da se počisti kako bi se resursi oslobodili i učinili dostupnim za ponovno korišćenje. U jednostavnim programskim situacijama pitanje kako očistiti objekat ne čini se previše komplikovanim: napravite objekat, koristite ga koliko vam je potreban, a onda neka se uništi. Međutim, moguce su i složenije situacije.

Pretpostavimo, na primer, da pišete program za upravljanje vazdušnim saobraćajem na nekom aerodromu. (Isti model bi mogao da se primeni i na rukovanje sanducima u skladištu, ili na sistem iznajmljivanja video kaseta, ili na smeštaj za kućne ljubimce.) Na prvi pogled, ovo izgleda jednostavno: napravite kontejner za čuvanje aviona, a zatim napravite novi avion i stavite ga u kontejner svaki put kada avion ude u zonu kontrole vazdušnog saobraćaja. Da biste očistili sistem kada avion napusti zonu, počistite iz memorije odgovarajući objekat avion.

Možda imate i neki drugi sistem za zapisivanje podataka o avionima o kojima ne treba neposredno voditi računa. Možda je to evidencija o planovima leta malih aviona koji napuštaju aerodrom. Znači, trebao bi vam drugi kontejner za male avione i kad god napravite objekat aviona, ako je to mali avion, stavili biste ga takođe i u ovaj drugi kontejner. Zatim bi neki pozadinski proces obavljao operacije nad objektima iz tog kontejnera kada računar ne radi ništa drugo.

Sada je problem nešto teži: kako uopšte možete znati kada da uništite objekte? Kada završite sa objektom, nekom drugom delu sistema možda je još uvek potreban. Isti problem se može pojaviti i postati vrlo složen u mnogim drugim situacijama i u programskim sistemima (kao što je C++) u kojima morate izričito da obrišete objekat kada s njim završite.

Gde se nalaze podaci iz objekta i kako se kontroliše trajanje objekta? C++ zauzima stav da je najvažnija efikasnost, pa programeru prepušta izbor. Da bi se postigla maksimalna brzina izvršavanja, čuvanje i trajanje mogu biti određeni prilikom pisanja programa, tako što se objekti stavljaju na stek (oni se ponekad nazivaju i *automatske* ili *vidljive* – engl. *scoped* – promenljive) ili u statičku oblast za čuvanje. Ta mesta imaju visok prioritet i prostor se u njima brzo zauzima i oslobađa, pa je kontrola nad njima veoma značajna u nekim situacijama. Međutim, time žrtvujete fleksibilnost jer morate da znate tačan broj, trajanje i tip objekata prilikom pisanja programa. Ako pokušavate da rešavate opštiji problem, na primer projektovanje pomoću računara (CAD), upravljanje skladištem ili kontrolu vazdušnog saobraćaja, ova metoda je previše restriktivna.

Drugi pristup je dinamičko stvaranje objekata u dinamičkoj oblasti memorije (engl. *heap*). Pri ovom pristupu, sve dok ne počne izvršavanje, ne znate koliko vam objekata treba, koliko će trajati, niti kog su tipa. Sve se to određuje kada program već radi (ovakav način pravljenja objekata zove se dinamički). Ako vam zatreba novi objekat, napravite ga u dinamičkoj memoriji, u trenutku kada vam je zatrebao. Pošto se oblašću za čuvanje upravlja dinamički, prilikom izvršavanja, zauzimanje memorijskog prostora traje znatno duže od odvajanja prostora na steku. Odvajanje prostora na steku često se postiže samo asemblerskom naredbom da se pokazivač steka pomeri naniže i da se vrati nazad. Vreme za koje se odvoji prostor u dinamičkoj memoriji, zavisi od dizajna mehanizma za skladištenje.

Dinamički pristup se zasniva na najčešće opravdanoj pretpostavci da su objekti složeni, tako da dodatni režijski troškovi za nalaženje prostora i njegovo oslobađanje neće bitno uticati na stvaranje objekata. Veća fleksibilnost dinamičkog pristupa je neophodna za rešavanje opštijih programskih problema.

Java isključivo koristi drugi pristup.[7] Svaki put kada želite da stvorite objekat, koristite rezervisanu reč **new** da biste napravili dinamički primerak tog objekta.

Nameće se još i pitanje trajanja objekata. U jezicima koji dozvoljavaju da se objekti stvaraju na steku, prevodilac određuje koliko dugo objekti traju i može automatski da ih uništi. Međutim, kada objekat stvorite dinamički, prevodilac nema informacije o njegovom životnom veku. U jeziku kao što je C++ morate programski da odredite kada ćete da uništite objekat, što može dovesti do „curenja" memorije ako se to ne uradi ispravno (to je čest problem u programima pisanim na C++-u). Java obezbeđuje tzv. *sakupljač smeća* (engl. *garbage collector*) koji automatski otkriva koji se objekat više ne upotrebljava i uništava ga.

[7] Prosti tipovi, o kojima ćete kasnije saznati više, predstavljaju specijalan slučaj.

Sakupljanje smeća je veoma korisno, jer smanjuje broj stavki na koje morate da mislite i količinu koda koji morate da napišete. Još je bitnije što sakupljanje smeća obezbeđuje mnogo viši nivo zaštite od podmuklih problema sa „curenjem" memorije (koje je mnoge projekte pisane na jeziku C++ oborilo na kolena).

U Javi, sakupljač smeća vodi računa o problemu oslobađanja memorije (iako u to ne spadaju drugi aspekti čišćenja objekta). Sakupljač smeća „zna" kada se objekat više ne koristi i automatski oslobađa memoriju koju je zauzimao taj objekat. To, uz činjenicu da su svi objekti izvedeni iz jedne osnovne klase **Object**, kao i da postoji samo jedan način za pravljenje objekata – dinamički – čini proces programiranja u Javi mnogo jednostavnijim od programiranja na jeziku C++. Mnogo je manje odluka koje treba da donesete i prepreka koje morate da prevaziđete.

Obrada izuzetaka: postupanje s greškama

Još od prvobitnih programskih jezika, obrada grešaka je bila jedna od najtežih oblasti. Pošto je teško napraviti dobru šemu za obradu grešaka, u mnogim jezicima zanemarena je ta oblast i to pitanje, a problem se prepušta projektantima biblioteka. Projektanti biblioteka su iznašli polovične mere primerene mnogim situacijama, ali koje se lako mogu zaobići, obično ignorisanjem. Osnovni problem sa svim šemama za obradu grešaka je to što se oslanjaju na obazrivost programera, i na ugovorene konvencije koje nisu nametnute jezikom. Ako programer nije obazriv – što se često dešava kad žuri – te šeme lako može ispustiti iz vida.

Obrada izuzetaka ugrađuje obradu grešaka direktno u programski jezik i ponekad čak i u operativni sistem. Izuzetak je objekat „bačen" (engl. *thrown*) s mesta greške, koji odgovarajući upravljač izuzecima koji obrađuje određeni tip grešaka, može da „uhvati" (engl. *catch*). Kao da je obrada izuzetaka drugi, paralelni put izvršavanja, kojim može da se krene kada stvari pođu naopako. Zato što koristi poseban put za izvršavanje, obrada izuzetaka ne mora da se meša s vašim kodom koji se normalno izvršava. Pošto ne morate stalno da proveravate da li je bilo grešaka, pisanje koda će najčešće biti jednostavnije. Sem toga, bačen izuzetak se razlikuje od vrednosti greške koja je vraćena iz metode i od indikatora stanja koji ukazuje na grešku, po tome što oni mogu da se ignorišu. Izuzetak se ne može ignorisati i garantuje se da će biti obrađen u nekom trenutku. Konačno, izuzeci obezbeđuju da se pouzdano izvučete iz loše situacije. Umesto da samo izađete iz programa, često ste u prilici da sredite stvari i nastavite sa izvršavanjem, čime dobijate mnogo robusnije programe.

Javina obrada izuzetaka se izdvaja od drugih programskih jezika jer je ugrađena od početka, pa ste prinuđeni da je koristite. To je jedini prihvatljivi način prijavljivanja grešaka. Ako ne napišete kod tako da pravilno obrađuje izuzetke, javiće se greška pri prevođenju. Ova garantovana doslednost katkada znatno pojednostavljuje obradu grešaka.

Treba napomenuti da obrada izuzetaka nije samo objektno orijentisana mogućnost, iako su u objektno orijentisanim jezicima izuzeci obično predstavljeni objektima. Obrada izuzetaka je postojala i pre objektno orijentisanih jezika.

Paralelni rad

Osnovni koncept u računarskom programiranju jeste obrada više od jednog zadatka istovremeno. Mnogi programski poslovi zahtevaju da program može da zaustavi rad, pozabavi se nekim drugim problemom a zatim se vrati glavnom procesu. Problemu se pristupalo na više načina. Na početku, programeri koji su poznavali računar do najsitnijih pojedinosti, pisali su prekidne servisne rutine, a suspenzija glavnog procesa je inicirana preko hardverskog prekida. Iako je sve to dobro radilo, bilo je teško i neprenosivo, jer je prebacivanje programa na novi tip računara bilo sporo i skupo.

Ponekad su za obradu zadataka koji se moraju odmah izvršiti prekidi rada neophodni, ali postoji velika klasa poslova u kojoj problem pokušavamo da izdelimo na više delova koji se izvršavaju zasebno i paralelno, da bi ceo program brže reagovao. Delovi koji se zasebno izvršavaju unutar programa nazivaju se niti (engl. *thread*), a ceo koncept *paralelni rad* (engl. *concurrency*). Tipičan primer paralelnog rada je korisničko okruženje. Zbog podele programa na niti, korisnik može da pritisne dugme i da dobije brz odziv, umesto da bude prinuđen da čeka dok program ne završi trenutni zadatak.

Nitima se obično raspodeljuje vreme jednog (i jedinog) procesora. Međutim, ako operativni sistem podržava višeprocesorski rad, svaka nit može biti dodeljena različitim procesorima, i tada one zaista mogu da rade paralelno. Jedna od korisnih osobina paralelnog rada na nivou jezika jeste sledeća: programer ne mora da brine postoji li jedan ili više procesora. Program je logički podeljen na niti i ako računar ima više od jednog procesora, program će se brže izvršavati, bez ikakvih posebnih prilagođavanja.

Nakon svega ovoga, paralelni rad zvuči prilično jednostavno. Postoji ipak jedna začkoljica: deljeni resursi. Ako se simultano izvršava više niti koje očekuju da pristupe istom resursu, pojaviće se problem. Na primer, dva procesa ne mogu istovremeno da šalju podatke istom štampaču. Da bi se rešio problem, resursi koji mogu biti deljeni, kao što je štampač, moraju biti zaključani dok se koriste. Znači, nit zaključa resurs, završi zadatak, a zatim otključa resurs, posle čega neko drugi može da ga koristi.

Paralelni rad je ugrađen u Javu, a Java SE5 ga dodatno podržava svojom bibliotekom.

Java i Internet

Ako je Java zapravo samo još jedan računarski programski jezik, možete se zapitati zašto je toliko važna i zašto se predstavlja kao revolucionarni korak u računarskom programiranju. Odgovor nije odmah očigledan, ukoliko se posmatra iz tradicionalne programerske perspektive. Iako je Java veoma korisna za rešavanje uobičajenih zasebnih programskih problema, još je važnije to što pomoću nje možete rešiti i programske probleme koji se tiču Weba.

Šta je Web?

Isprva Web može da deluje pomalo tajanstveno, uz celu priču o „krstarenju", „prisustvu" i „ličnim prezentacijama". Korisno je malo se udaljiti i sagledati šta Web zaista jeste, ali za to morate da razumete sisteme klijent/server, još jedan aspekt računarske obrade koji je pun zbunjujućih tema.

Klijent/server obrada

Osnovna ideja sistema klijent/server jeste postojanje centralnog skladišta informacija – neke vrste podataka, obično u bazi podataka – koje hoćete da šaljete, po zahtevu, grupi ljudi ili računara. U konceptu klijent/server ključno je to što je skladište informacija centralizovano tako da informacije mogu da se menjaju i da se te promene prenose svim korisnicima informacija. Skladište informacija, softver koji šalje informacije i računar (računari) gde se softver i informacije nalaze, nazivaju se *server*. Softver koji se nalazi na korisničkom računaru, sarađuje sa serverom, preuzima informacije, obrađuje ih i prikazuje na udaljenom računaru, naziva se *klijent*.

Osnovni koncept klijent/server obrade, znači, nije previše složen. Problemi se javljaju jer imate jedan server koji pokušava da opsluži više klijenata u isto vreme. Uglavnom se koristi sistem za upravljanje bazama podataka, tako da programer „uravnotežava" raspored podataka u tabelama da bi postigao optimalno korišćenje. Sistemi često dozvoljavaju klijentima i da dodaju informacije na server. To znači da morate obezbediti da novi podaci jednog klijenta ne „pregaze" nove podatke drugog klijenta, ili da se ti podaci ne izgube u procesu dodavanja u bazu. (Ovo se zove obrada transakcija – engl. *transaction processing*). Kada klijentski softver treba izmeniti, on mora da se prevede, očisti od grešaka i instalira na klijentskim računarima, što je u stvarnosti znatno komplikovanije i skuplje nego što mislite. Posebno je problematično podržati više tipova računara i operativnih sistema. Konačno, tu je i veoma bitno pitanje performansi: stotine klijenata mogu da postavljaju zahteve vašem serveru u bilo kom trenutku, pa je i najmanje zakašnjenje problematično. Da bi se kašnjenje svelo na najmanju meru, programeri naporno rade da rasterete procesne zadatke, prebacujući deo obrade na klijentski računar, a ponekad i na druge računare na serverskoj strani, koristeći takozvane *posrednike* (engl. *middleware*). (Posrednici se takođe koriste da bi se olakšalo održavanje.)

Realizacija tako jednostavnog postupka – slanja informacija – toliko je slojevita i složena da ceo problem deluje beznadežno zagonetno. I pored toga je veoma bitna: klijent/server obrada nosi otprilike polovinu svih programskih aktivnosti. Ona je zadužena za sve, počev od uzimanja narudžbina, transakcija s kreditnim karticama, pa do slanja raznih vrsta podataka – berzanskih, naučnih, državnih; šta god vam padne na pamet. U prošlosti smo se suočavali sa individualnim rešenjima individualnih problema; svaki put smo smišljali novo rešenje. Bilo je teško osmisliti ih, teško su se upotrebljavala i korisnik je morao da uči nov interfejs za svako od tih rešenja. Ceo klijent/server problem trebalo bi sveobuhvatno da se reši.

Web kao džinovski server

Web je u suštini jedan džinovski sistem klijent/server, i nešto više od toga, jer svi serveri i svi klijenti postoje zajedno i u isto vreme u samo jednoj mreži. To ne morate da znate jer u datom trenutku brinete samo o povezivanju i interakciji s jednim serverom (iako možda pri tom lutate Mrežom tražeći odgovarajući server).

U početku je to bio jednostavan jednosmerni proces. Vi ste ispostavljali zahtev serveru i on vam je prosleđivao datoteku koju je softver za čitanje (klijent) prevodio i formatirao na vašem lokalnom računaru. Ali, ubrzo su ljudi poželeli da rade više od čistog prosleđivanja

stranica sa servera. Želeli su punu klijent/server kompatibilnost kako bi klijent bio u mogućnosti da vraća podatke mogućnost na server, na primer, da pretražuje baze na serveru, dodaje nove informacije na server, ili da ostavi narudžbinu (što je sve zahtevalo posebne bezbednosne mere). Bili smo svedoci tih promena u razvoju Weba.

Čitač Weba je označio veliki korak napred – uveo je mogućnost da se delići informacija neizmenjeni prikazuju na bilo kom tipu računara. Ti čitači su ipak bili prilično primitivni i ubrzo su se zaglibili u zahtevima koji su im ispostavljani. Oni nisu bili previše interaktivni; zagušivali su server i sam Internet jer ste, svaki put kada je trebalo uraditi nešto što zahteva programiranje, morali da vraćate informacije serveru na obradu. Moglo je da prođe više sekundi ili minuta dok ne otkrijete da ste nešto pogrešno otkucali. Pošto je čitač bio predviđen samo za pregled, nije mogao da izvrši ni najjednostavnije zadatke obrade. (S druge strane, bio je potpuno bezbedan, jer na vašem računaru nije mogao da izvrši nijedan program, dakle ni da mu potencijalno donese grešku ili virus.)

Primenjeno je više pristupa rešenju ovog problema. Za početak, poboljšani su grafički standardi da bi omogućili bolju animaciju i video prikaz u čitačima. Drugi deo problema mogao je biti rešen samo ugrađivanjem u čitač mogućnosti za pokretanje programa na klijentskoj strani. To se naziva *programiranje s klijentske strane* (engl. *client-side programming*).

Programiranje s klijentske strane

Početni dizajn Weba (server–čitač) omogućavao je interaktivne sadržaje, ali je celokupnu interaktivnost obezbeđivao server. Server je pravio statičke stranice za klijentski čitač koji ih je tumačio i prikazivao. Elementarni *jezik za označavanje hiperteksta* (HTML) ima jednostavne mehanizme za prikupljanje podataka: polja za tekst, polja za potvrdu, radio-dugmad, liste i padajuće liste, kao i dugme koje može biti programirano samo da obriše podatke u obrascu ili da pošalje (engl. *submit*) podatke iz obrasca nazad serveru. Ovo slanje se obavlja preko interfejsa CGI (engl. *Common Gateway Interface*), koji postoji na svim Web serverima. Tekst koji se nalazi u poslatom paketu kazuje serveru šta s tim paketom da uradi. Najčešća akcija je da se pokrene program koji se nalazi na serveru, u direktorijumu koji se obično zove „cgi-bin". (Ako budete gledali adresno polje na vrhu svog čitača kada pritisnete neko dugme na Web stranici, ponekad ćete videti „cgi-bin" negde unutar sveg onog zamešateljstva.) Ti programi mogu biti napisani na skoro svim jezicima. Često se pisalo na Perlu, jer je on napravljen za rad s tekstom, a uz to se interpretira, pa može biti instaliran na bilo koji server, bez obzira na procesor ili operativni sistem. Danas se sve više koristi Python (*www.Python.org*), zato što je jači i jednostavniji.

Mnoge današnje moćne Web stranice izgrađene su isključivo na CGI programima, s kojima možete da uradite gotovo sve. Ipak, održavanje Web stranica izgrađenih na CGI programima može brzo da postane suviše složeno, a postoji i problem s vremenom odziva. Odziv CGI programa zavisi od toga koliko podataka se šalje, kao i od opterećenja servera i Interneta. (Uz to, pokretanje CGI programa ume da bude sporo.) Prvi projektanti Weba nisu predvideli koliko brzo će njegova propusnost da postane premala za nove vrste aplikacija. Na primer, skoro je nemoguće dosledno ostvariti bilo koju vrstu dinamičkog crtanja grafika jer za svaku verziju grafika mora da se napravi GIF slika a zatim prenese sa servera do klijenta. (GIF je akronim od *Graphic Interchange Format*, format za razmenu

grafika.) Pored toga, bez sumnje ste stekli neposredno iskustvo s proverom ispravnosti podataka na ulaznom obrascu. Pritisnete dugme za slanje na stranici, server pokrene CGI program koji otkrije grešku, formatira HTML stranicu obaveštavajući vas o grešci, a zatim vam vrati tu stranicu. Onda vi morate da se vratite na prethodnu stranicu i pokušate ponovo. Ovo ne samo da je sporo već je i zamorno.

Rešenje je programiranje s klijentske strane. Većina stonih računara na kojima rade čitači Weba sposobni su da urade ogroman posao, a s prvobitnim statičkim HTML pristupom samo su stajali i besposleno čekali da server isporuči sledeću stranicu. Programiranje s klijentske strane znači da čitač Weba radi sav posao koji može da obavi, korisnik mnogo brže dobija rezultat a osećaj zajedništva pri radu s Web stranicom je potpuniji.

Rasprave o programiranju s klijentske strane malo se razlikuju od rasprava o programiranju uopšte. Parametri su gotovo isti, ali je platforma drugačija: čitač Weba je sličan ograničenom operativnom sistemu. Na kraju, opet morate da programirate, i zato programiranje s klijentske strane povlači vrtoglav niz problema i rešenja. U daljem tekstu dajemo pregled pitanja i pristupa pri programiranju klijentske strane.

Dodaci

Razvoj dodataka (engl. *plug-ins*) predstavlja jedan od najdužih koraka napred u programiranju s klijentske strane. Programer dodaje novu funkcionalnost čitaču tako što korisnik u potrebnom trenutku preuzme deo koda koji se ugradi na odgovarajuće mesto u čitaču. Taj kôd govori čitaču: „Od sada možeš da obavljaš i ovu novu aktivnost". (Dodatak treba da preuzmete samo jednom.) Preko dodataka su čitačima pridodata brza i moćna proširenja, ali pisanje dodataka nije jednostavan zadatak i nije nešto što biste želeli da radite u okviru izgradnje određene stranice. Vrednost dodataka za programiranje s klijentske strane jeste to što oni omogućavaju stručnjaku da pravi nova proširenja i da ih dodaje čitaču bez odobrenja proizvođača čitača. Stoga, dodaci predstavljaju „zadnja vrata" koja omogućavaju stvaranje novih jezika za programiranje s klijentske strane (iako nisu svi jezici realizovani kao dodaci).

Skript-jezici

Dodaci su prouzrokovali razvoj skript-jezika za čitače. Pomoću skript-jezika ugrađujete izvorni kôd programa za klijentsku stranu direktno u HTML stranicu, a dodatak koji tumači taj jezik automatski se aktivira pri prikazivanju HTML stranice. Skript-jezici se obično prilično lako shvataju i pošto se pišu u delu HTML stranice, učitavaju se veoma brzo, jednim pristupom serveru na kome se nalazi ta stranica. Nedostatak je to što je vaš kôd izložen i svako može da ga vidi (i ukrade). U principu, ipak ne pravite neke sofisticirane stvari pomoću skript-jezika, pa ovaj nedostatak i nije neki problem.

Postoji jedan skript-jezik koji većina čitača Weba podržava i *bez* ikakvog dodatka – to je JavaScript (koji ima tek prolaznu sličnost s Javom, i moraćete ga učiti zasebno). Tako je nazvan samo da bi prigrabio deo Javinog marketinškog uzleta. Nažalost, većina čitača Weba je svoju podršku za JavaScript realizovala na svoj jedinstveni način, drugačije od ostalih čitača, pa čak i od ostalih svojih verzija. Nešto je popravila standardizacija JavaScripta u obliku *ECMAScripta*, ali je raznim čitačima trebalo mnogo vremena da dodu do

tog nivoa (tome je doprineo i Microsoft, gurajući svoj VBScript koji pomalo liči na JavaScript). Da bi program mogao da se izvršava na svim čitačima, po pravilu ga morate pisati koristeći najmanji zajednički imenilac svih postojećih verzija JavaScripta. Programska obrada grešaka, kao i otkrivanje i otklanjanje grešaka u toku pisanja programa, mogu se opisati samo kao mučenje. Dokaz tih poteškoća je činjenica da je na JavaScriptu tek nedavno napisan zaista složen program (to je Googleov GMail), što je zahtevalo veliki trud i stručnost.

Ovim se naglašava da su skript-jezici koji se koriste unutar čitača Weba predviđeni da reše određene vrste problema, prvenstveno da se naprave bogatija i interaktivnija grafička korisnička okruženja. Skript-jezik može da reši i do 80 procenata problema koje srećemo pri programiranju s klijentske strane. Vaši problemi verovatno spadaju u tih 80 procenata, a kako skript-jezici omogućuju lakši i brži rad, trebalo bi da razmislite o njima pre nego što se upustite u mnogo zapetljanija rešenja, kao što je programiranje na Javi.

Java

Ako skript-jezici mogu da razreše 80 procenata problema pri klijentskom programiranju, šta je s preostalih 20 procenata „zaista teških stvari"? Java je popularno rešenje za tih 20%. Pre svega, to je moćan programski jezik, siguran, međuplatformski i internacionalan. Java se neprekidno širi: dodaju se nove mogućnosti jezika i biblioteke koje mogu elegantno da reše probleme teške i u tradicionalnim programskim jezicima, kao što su istovremeni rad, pristup bazama, mrežno programiranje i distribuirana obrada. Java dopušta programiranje s klijentske strane preko *apleta* i pomoću *Java Web Starta*.

Aplet je mali program koji može da se izvršava samo unutar čitača Weba. Aplet se automatski preuzima kao deo Web stranice (kao što se, na primer, automatski preuzima grafika). Kada se aplet aktivira, on izvršava program. Deo njegove pogodnosti je to što omogućava da automatski distribuirate klijentski softver sa servera tek u trenutku kada je korisniku klijentski softver potreban, a ne ranije. Korisnik bespogovorno dobija najnoviju verziju klijentskog softvera i to bez složene ponovne instalacije. Zbog načina na koji je Java projektovana, programer treba da napravi samo jedan jedini program, a taj program automatski radi na svim računarima koji imaju čitače s podrškom za Javu. (Ovo, bez ikakve brige, obuhvata ogromnu većinu računara.) Pošto je Java potpun programski jezik, možete uraditi sav posao koji je moguć na klijentskoj strani, pre i posle postavljanja zahteva serveru. Na primer, nema potrebe da šaljete obrazac sa zahtevom preko Interneta kako biste otkrili da li ste pogrešili pri upisivanju datuma ili nekog drugog parametra; vaš računar može brzo da iscrtava grafike na osnovu podataka, umesto da čeka da ih server iscrta i vrati sliku. Pored trenutnog poboljšanja brzine i odziva, smanjuju se ukupni mrežni saobraćaj i opterećenje servera i sprečava usporavanje celog Interneta.

Druge mogućnosti

Pošteno rečeno, Java apleti nisu opravdali velika početna očekivanja. Kada se Java tek pojavila, najviše se govorilo upravo o apletima, jer je trebalo da oni najzad omoguće ozbiljno programiranje na klijentskoj strani, povećaju brzinu odziva i smanje protok podataka potreban Internet aplikacijama. Predvidale su se ogromne mogućnosti.

Na Webu se zaista mogu naći i veoma pametni apleti, ali do sveopšteg prelaska na aplete nije došlo. Verovatno je najveći problem bila veličina Javinog izvršnog okruženja (Java Runtime Environment, JRE) od 10 MB, koje je trebalo preuzeti s Weba i instalirati, što je uplašilo prosečnog korisnika. Sudbinu im je možda zapečatila činjenica da je Microsoft odlučio da JRE ne isporučuje kao deo svog Internet Explorera. U svakom slučaju, Java apleti se nisu proširili posvuda.

Bez obzira na to, u nekim situacijama Java aplete i *Java Web Start* aplikacije još uvek vredi imati. Ukoliko upravljate računarima korisnika, recimo unutar preduzeća, bilo bi pametno da distribuciju i ažuriranje klijentskih aplikacija obavljate pomoću ovih tehnologija, jer ćete time uštedeti znatnu količinu vremena, truda i novca, naročito ako morate često da ažurirate njihov softver.

U poglavlju *Grafička korisnička okruženja* upoznaćemo *Flex*, novu Adobeovu tehnologiju koja obećava – u njome se prave Flash apleti. Pošto više od 98 procenata svih čitača Weba ima Flash Player (na Windowsu, Linuxu i Macu), možemo smatrati da je on usvojen standard. Flash Player se instalira i ažurira lako i brzo. Jezik ActionScript je napravljen na osnovu ECMAScripta i stoga je prilično poznat, ali Flex omogućava programiranje bez brige o specifičnostima raznih čitača – zato je daleko privlačniji od JavaScripta. Pri programiranju na klijentskoj strani, ovu mogućnost vredi razmotriti.

.NET i C#

Neko vreme je glavni konkurent Java apleta bio Microsoftov ActiveX, iako samo na Windows računarima. Otad je Microsoft napravio prvog pravog konkurenta celoj Javi u liku platforme .NET i programskog jezika C#. Platforma .NET je približno isto što i Javina virtuelna mašina (*Java Virtual Machine*, JVM) plus Java biblioteke. JVM je softverska platforma na kojoj se izvršavaju Java programi, a C# ima velike sličnosti s Javom. Bez sumnje, radi se o dosad najboljem Microsoftovom proizvodu na području programskih jezika i programskih okruženja. Naravno, Microsoft je imao veliku prednost jer je mogao da uči na tuđim greškama, ali je bogami naučio. Prvi put od svog nastanka, Java je dobila pravu konkurenciju. Zato su projektanti Jave u Sunu dobro pogledali C#, dobro razmislili o tome zašto bi programeri hteli da pređu na C#, i odgovorili: načinili su fundamentalna poboljšanja Jave – Javu SE5.

Trenutno je najveća slabost .NET-a važno pitanje da li će Microsoft dozvoliti njegovo *potpuno* prenošenje na druge platforme. Microsoft tvrdi da se to može napraviti bez problema, i već postoji delimična realizacija .NET-a koja radi na Linuxu (projekat Mono, *www.gomono.com*), ali dok se ne napravi potpuna realizacija iz koje Microsoft nije ništa izbacio, .NET je rizično smatrati rešenjem za sve platforme.

Internet i intranet

Web je najopštije rešenje problema klijent/server, pa je logično da istu tehnologiju upotrebite i da biste rešili podgrupu tog problema, posebno klasični problem klijent/server *unutar* preduzeća. Pri tradicionalnom pristupu klijent/server, postoji problem zbog raznih tipova računara kao i problem zbog teškog instaliranja novog klijentskog softvera. Oba problema dobro su rešena pomoću čitača Weba i programiranja s klijentske strane.

Kada se tehnologija Weba koristi za informacionu mrežu ograničenu na određeno preduzeće, to nazivamo intranet. Intranet obezbeđuje mnogo veću sigurnost nego Internet jer pristup serverima u preduzeću možete fizički da kontrolišete. Izgleda da korisnici, kada jednom shvate osnovni princip rada čitača, mnogo lakše izlaze na kraj s različitim izgledima stranica i apleta, pa brže uče nove sisteme.

Problem s bezbednošću svrstava nas u jednu od grupa koje se obrazuju, čini se, po automatizmu, u svetu klijent/server programiranja. Ako se vaš program izvršava na Internetu, ne znate na kojoj platformi će on raditi i želite da budete sasvim sigurni da ne širite neispravan kôd. Treba vam nešto nezavisno od platforme i bezbedno, kao što su skript-jezik ili Java.

Ako radite na intranetu, pred vas se postavljaju drugačija ograničenja. Nije neuobičajeno da svi računari budu platforme Intel/Windows. Na intranetu ste sami odgovorni za kvalitet svog programa i možete da otklanjate greške kako se koja otkriva. Pored toga, možda već imate dosta nasleđenog koda koji ste koristili za tradicionalniji klijent/server pristup, pri čemu morate fizički da instalirate klijentske programe svaki put kada radite na poboljšanju. Vreme protraćeno na instaliranje poboljšanja pruža razlog više da pređemo na čitače gde su poboljšanja nevidljiva i automatska. (Ovaj problem rešava i Java Web Start.) Ako imate posla s takvim intranetom, najrazboritiji pristup je da krenete najkraćim putem koji vam omogućava da koristite postojeću bazu koda, umesto da ponovo pišete programe na novom jeziku.

Kada se suočite sa ovim zbunjujućim nizom rešenja za programiranje s klijentske strane, najbolji plan napada je analiza isplativosti. Razmotrite ograničenja koja vaš problem nameće i najkraći put do rešenja. Pošto je programiranje s klijentske strane ipak programiranje, uvek je dobro odabrati pristup koji omogućava najbrži razvoj u određenoj situaciji. Ovo je hrabar stav koji vas priprema za neizbežne susrete s problemima u razvoju programa.

Programiranje sa serverske strane

Do sada je pitanje programiranja sa serverske strane bilo zanemareno, a baš tu je po mišljenju mnogih Java imala najveće uspehe. Šta se dešava kada pošaljete zahtev serveru? Najčešće zahtevi glase: „Pošalji mi ovu datoteku". Vaš čitač zatim tumači tu datoteku na odgovarajući način: kao HTML stranicu, sliku, Java aplet, skript itd.

U složenije zahteve serveru obično spadaju i transakcije s bazama podataka. Obično se zahteva kompleksna pretraga baze podataka, koju server zatim formatira u HTML stranicu i šalje kao rezultat. (Naravno, ako je klijent inteligentniji, zato što sadrži kôd na Javi ili skript-jeziku, mogu se slati neobrađeni podaci, a potom formatirati na strani klijenta, što bi bilo brže i manje bi opterećivalo server.) Možda biste poželeli da registrujete svoje ime u bazi podataka kada pristupate nekoj grupi ili ostavljate narudžbinu, što će izazvati promene u toj bazi. Pisanje programa koji obrađuju takve zahteve na serveru naziva se programiranje sa serverske strane (engl. *server-side programming*). Programiranje sa serverske strane obavlja se na Perlu, Pythonu, C++-u ili nekom drugom jeziku za pisanje CGI programa, ali pojavili su se i napredniji sistemi. Među njima su i Web serveri zasnovani na Javi, koji omogućavaju pisanje takozvanih *servleta*. Kompanije koje razvijaju Web

stranice prelaze na Javu zbog tehnologije servleta i iz njih izvedenih JSP strana, najviše zato što se time isključuju problemi pri radu sa čitačima različitih mogućnosti. Programiranje sa serverske strane razmotreno je u knjizi *Thinking in Enterprise Java*, koju možete naći na adresi *www.MindView.net*.

Uprkos tome što se o Javi priča samo u vezi sa Internetom, ona je programski jezik opšte namene pomoću kogae možete rešiti bilo koji tip problema, kao i pomoću drugih jezika. Tu Javina snaga nije samo u prenosivosti, već i u lakoći programiranja, robusnosti, velikoj standardnoj biblioteci i brojnim bibliotekama drugih proizvođača koje se stalno razvijaju.

Sažetak

Znate kako izgleda proceduralni program: definicije podataka i pozivi funkcija. Da biste pronašli svrhu takvog programa, morate malo da se pomučite i pregledate pozive funkcija i koncepte niskog nivoa kako biste stvorili model u glavi. Zbog toga nam treba posredno predstavljanje kada projektujemo proceduralni program: sami za sebe, ti programi mogu da zbunjuju jer su načini izražavanja usmereni više ka računaru nego ka problemu koji rešavamo.

Pošto OOP dodaje mnoge nove ideje onima koje nalazite u proceduralnim jezicima, možda mislite da će Java program biti mnogo komplikovaniji nego ekvivalentni C program. Bićete prijatno iznenađeni: dobro napisan Java program je uglavnom daleko jednostavniji i mnogo razumljiviji od ekvivalentnog C programa. Imaćete definicije objekata koji predstavljaju ideje u vašem prostoru problema (umesto pitanja računarskog prikaza) i poruke poslate tim objektima koje predstavljaju aktivnosti u tom prostoru. Dobra strana u objektno orijentisanom programiranju jeste ta što je uz dobro projektovan program lako razumeti kôd pri čitanju. Obično ima i znatno manje koda, jer ćete mnoge svoje probleme rešiti ponovnim korišćenjem postojećeg koda iz biblioteka.

OOP i Java ne moraju da budu za svakoga. Važno je da procenite svoje potrebe i odlučite da li će ih Java optimalno zadovoljiti ili bi bilo bolje da radite s nekim drugim programskim sistemom (uključujući onaj koji trenutno koristite). Ako znate da će vaše potrebe biti usko specijalizovane u bliskoj budućnosti i ako imate specifična ograničenja koja Java možda ne može da zadovolji, onda ispitajte ostale raspoložive mogućnosti. Konkretno, preporučujem da razmotrite Python; posetite *www.Python.org*. Ako ipak izaberete Javu, barem ćete znati koje ste mogućnosti imali i zašto ste baš nju izabrali.

Sve je objekat

„Da govorimo drugačiji jezik, percipirali bismo nešto drugačiji svet."
Ludwig Witgenstein (1889–1951)

Iako je zasnovana na C++-u, Java je čistiji objektno orijentisani jezik.

JAVA I C++ SU HIBRIDNI JEZICI ALI SU PROJEKTANTI JAVE SMATRALI DA HIBRIDIZACIJA NIJE toliko bitna kao u C++-u. U hibridnom jeziku može se koristiti više programskih stilova; C++ je hibridan da bi obezbedio povratnu kompatibilnost s jezikom C. Pošto je C++ nadskup jezika C, on sadrži i mnoge nepoželjne nasleđene osobine koje previše komplikuju neke aspekte C++-a.

Jezik Java je napravljen za one koji žele da se bave samo objektno orijentisanim programiranjem. Pre nego što se uopšte upustite u programiranje, morate početi da razmišljate objektno-orijentisano (osim ako već ne razmišljate tako). Ovaj početni napor će vam se isplatiti jer ćete biti sposobni da programirate na jeziku koji se uči i koristi jednostavnije od mnogih drugih OOP jezika. U ovom poglavlju razmatraćemo osnovne delove Java programa i naučićemo da je u Javi (gotovo) sve objekat.

Rad sa objektima preko referenci

Svaki programski jezik ima svoje načine za rad sa elementima u memoriji. Programer često ne sme da gubi iz vida način rada koji se primenjuje. Da li sa elementom radite direktno ili preko posrednog predstavljanja (pokazivač u C-u ili C++-u), koje se opisuje posebnom sintaksom?

Sve ovo je pojednostavljeno u Javi. Sve tretirate kao objekat, istom doslednom sintaksom koju koristite svuda. Iako vi sve *tretirate* kao objekat, identifikator s kojim radite je, u stvari, „referenca" na objekat.[1] Zamislite televizor (objekat) s daljinskim upravljačem (referenca). Sve dok držite tu referencu, imate vezu s televizorom, ali kada neko kaže da promenite kanal ili da utišate zvuk, vi radite s referencom koja menja objekat. Ako želite da se šetate po sobi, a da i dalje podešavate televizor, ponećete daljinski/referencu, a ne televizor.

---

[1] Ovde može doći do nesuglasica. Ima ljudi koji kažu: „Očito, to je pokazivač", ali takav iskaz već sadrži pretpostavke o realizaciji. Po svojoj sintaksi, Java reference su mnogo srodnije C++ referencama nego pokazivačima. U prvom izdanju ove knjige odabrao sam nov termin „identifikator (engl. *handle*)" jer između referenci u C++-u i referenci u Javi postoje bitne razlike. Izlazio sam iz C++-a i nisam želeo da zbunjujem programere u C++-u, za koje sam smatrao da će biti najbrojniji korisnici Jave. Pre nego što sam pripremio drugo izdanje, video sam da je češće korišćen termin referenca, pošto svako ko prelazi sa C++-a mora da savlada mnoge druge pojmove pored terminologije referenci, još jedan pojam neće previše smetati. Međutim, neki se ne slažu čak ni sa terminom referenca. U jednoj knjizi tvrdi se kako je „potpuno pogrešno reći da Java podržava prosleđivanje preko referenci" jer su identifikatori objekata u Javi (po tom autoru) *u stvari* „objekti referenci". A pošto se sve (kako dalje tvrdi) *u stvari* prosleđuje po vrednosti, vi ne obavljate prosleđivanje preko referenci već „prosleđujete objekte referenci po vrednosti". Može se diskutovati o preciznosti tako zamršenih objašnjenja, ali smatram da moj pristup pojednostavljuje shvatanje koncepta, bez ikakvih gubitaka (pa, teoretičari jezika mogu da tvrde da vas lažem, ali ja ću reći da obezbeđujem odgovarajuću apstrakciju).

Daljinski upravljač može da postoji sam za sebe, bez televizora. Odnosno, to što imate referencu, ne znači da obavezno imate i objekat povezan s njom. Ako želite da čuvate reč ili rečenicu, treba da napravite referencu na objekat klase **String**:

```
String s;
```

Ali, ovim ste napravili *samo* referencu, ne i sam objekat. Ako pokušate u ovom trenutku da pošaljete poruku preko **s**, dobićete grešku pošto **s** u stvari nije povezano ni sa čim (nema televizora). Zato je sigurnije da uvek inicijalizujete referencu kada je pravite:

```
String s = "asdf";
```

Ovde se koristi posebna osobina Jave: znakovni niz se može inicijalizovati tekstom pod navodnicima. Obično, za objekte morate da koristite opštiji tip inicijalizacije.

Morate da napravite sve objekte

Kada napravite referencu, želite i da je povežete s novim objektom. Po pravilu, to radite pomoću rezervisane reči **new** kojom se kaže: „Napravi mi ovakav nov objekat". Znači, u gornjem primeru možete napisati:

```
String s = new String ("asdf");
```

Ovo ne samo da znači: „Napravi mi nov objekat klase **String**", već daje i informaciju o tome *kako* da se taj objekat napravi, tako što se navodi početni znakovni niz.

Pored tipa **String** Java ima obilje drugih unapred definisanih tipova. Mnogo je važnije da možete napraviti svoje tipove. U stvari, to je osnovna aktivnost u programiranju na Javi i o tome ćete učiti u ovoj knjizi.

Gde se nalazi skladište

Korisno je predstaviti neke aspekte izvršavanja programa, a posebno kako je organizovana memorija. Postoji pet različitih mesta za skladištenje podataka:

1. **Registri:** Tu se najbrže skladište podaci jer su registri odvojeni od ostalih skladišta: nalaze se unutar procesora. Međutim, broj registara je veoma ograničen, pa se registri automatski dodeljuju u skladu s potrebama. Vi nemate direktnu kontrolu, niti u vašem programu vidite ijedan dokaz da registri uopšte postoje. (S druge strane, C i C++ omogućavaju da prevodiocu sugerišete dodelu registara.)
2. **Stek:** Nalazi se u radnoj memoriji, ali ga procesor direktno kontroliše preko *pokazivača steka* (engl. *stack pointer*). Pokazivač steka se pomera naniže da bi zauzeo novu memoriju, a pomera se naviše da bi tu memoriju oslobodio. Ovo je veoma brz i efikasan način za dodeljivanje skladišta, od koga su brži samo registri. Java sistem mora da zna, dok pravi program, tačan životni vek svih stavki koji se čuvaju na steku. Ovo ograničenje omeđava fleksibilnost vašeg programa, pa dok se neka Javina skladišta nalaze na steku – posebno reference na objekte – sami objekti se ne čuvaju na steku.

3. **Dinamička memorija:** To je oblast memorije opšte namene (takođe u radnoj memoriji) gde obitavaju svi objekti. Dobra strana dinamičke memorije jeste to što, za razliku od steka, prevodilac ne mora da zna koliko dugo dodeljeno skladište mora da ostane u dinamičkoj memoriji. Stoga imamo veliku fleksibilnost pri korišćenju dinamičke memorije. Kad god vam zatreba neki objekat, napišite kôd za njegovo pravljenje koristeći **new**, i skladište će se napraviti u dinamičkoj memoriji. Naravno, morate platiti ovu fleksibilnost: skladište u dinamičkoj memoriji duže se pravi nego skladište na steku (kad biste uopšte *mogli* da napravite objekte na steku u Javi, kao što možete u C++-u).
4. **Konstantno skladište:** Konstantne vrednosti često se umeću direktno u kôd programa što je bezbedno jer se one nikad ne mogu promeniti. Ponekad se konstante izdvojeno grupišu kako bi opciono mogle da se stave u ROM memoriju (memorija samo za čitanje; engl. *read-only memory*), što se radi u računarima ugrađenim u druge uređaje.[2]
5. **Skladišta van radne memorije:** Ako su podaci potpuno izvan programa, oni mogu da postoje i kada se program ne izvršava, izvan kontrole programa. Dva osnovna primera za ovo su objekti *u tokovima podataka* (engl. *streamed objects*), što su objekti pretvoreni u tok bajtova, obično da bi se poslali na drugu mašinu, i *trajni objekti* (engl. *persistent objects*), što su objekti stavljeni na disk da bi sačuvali svoje stanje čak i kada se program završi. U ovim skladištima teško je pretvoriti objekte u nešto što može da postoji na drugom medijumu, a što može da postane regularni memorijski objekat kada je potrebno. Java obezbeđuje podršku za laku trajnost (engl. *lightweight persistence*), a mehanizmi kao što su JDBC i Hibernate obezbeđuju sofisticiraniju podršku za skladištenje i preuzimanje informacija o objektima u bazama podataka.

Specijalan slučaj: prosti tipovi

Jedna grupa tipova koji se često koriste u programiranju ima poseban tretman; možete ih smatrati osnovnim tipovima. Taj poseban tretman je neophodan iz sledećeg razloga: ako napravite objekat operatorom **new** – posebno malu, jednostavnu promenljivu – to neće biti baš efikasno jer **new** postavlja objekte u dinamičku memoriju. Za ove tipove Java se vraća na pristup koji imaju C i C++: umesto da napravi promenljivu korišćenjem operatora **new**, pravi automatsku promenljivu koja *nije referenca*. Promenljiva direktno čuva vrednost i rad s njom je mnogo efikasniji jer se čuva na steku.

Java unapred određuje veličinu svakog prostog tipa. Ove veličine se ne menjaju od jednog do drugog računara, kao što je slučaj u većini jezika. Nepromenljivost veličina je jedan od razloga zbog kog su programi na Javi prenosiviji nego oni na većini drugih jezika.

---

[2] Primer za to je skladište znakovnih nizova. U posebnom delu memorije s nepromenljivom (statičkom) adresom, automatski se skladište svi literalni znakovni nizovi i konstantni izrazi čiji je rezultat znakovni niz.

Prost tip	Veličina u bitovima	Minimum	Maksimum	Omotački tip
boolean	–	–	–	Boolean
char	16	Unicode 0	Unicode $2^{16}-1$	Character
byte	8	-128	+127	Byte
short	16	-2^{15}	$+2^{15}-1$	Short
int	32	-2^{31}	$+2^{31}-1$	Integer
long	64	-2^{63}	$+2^{63}-1$	Long
float	32	IEEE754	IEEE754	Float
double	64	IEEE754	IEEE754	Double
void	–	–	–	Void

Svi numerički tipovi su označeni (sadrže i pozitivne i negativne vrednosti), stoga ne tražite neoznačene tipove.

Veličina tipa **boolean** nije izričito definisana; zadato je samo da može da ima vrednosti **true** ili **false**.

Omotačke klase za proste tipove podataka omogućavaju da u dinamičkoj memoriji napravite objekat koji predstavlja određen prost tip. Na primer:

```
char c = 'x';
Character ch = new Character(c);
```

Možete takođe da koristite:

```
Character ch = new Character('x');
```

Java SE5 *automatskim pakovanjem* pretvara (omotava) proste tipove u objekte:

```
Character ch = 'x';
```

i *automatskim raspakivanjem* ponovo ih pretvara u proste tipove:

```
char c = ch;
```

Razlozi za omotavanje (pretvaranje) prostih tipova u objekte biće prikazani dalje u knjizi.

Brojevi visoke preciznosti

Java sadrži dve klase za obavljanje aritmetičkih operacija visoke preciznosti: **BigInteger** i **BigDecimal**. Iako one otprilike potpadaju pod istu kategoriju kao i omotačke klase, nijedna nema analogni prost tip.

Obe klase imaju metode koje obezbeđuju operacije analogne onima koje obavljate nad prostim tipovima. Odnosno, s klasama **BigInteger** i **BigDecimal** možete uraditi sve što i sa tipovima **int** ili **float**, samo morate koristiti metode umesto operatora. A pošto je zadatak komplikovaniji, operacije će biti sporije. Brzinu zamenjujete tačnošću.

BigInteger podržava celobrojne vrednosti proizvoljne veličine. To znači da možete tačno predstaviti celobrojne vrednosti bilo koje veličine, a da pri operacijama ne izgubite informacije.

BigDecimal služi za brojeve s pokretnim zarezom proizvoljne veličine; njih možete, na primer, koristiti za precizna novčana izračunavanja.

U dokumentaciji JDK potražite pojedinosti o konstruktorima i metodama ove dve klase.

Nizovi u Javi

Praktično svi programski jezici podržavaju neku vrstu nizova. Korišćenje nizova u C-u i C++-u rizično je jer su ti nizovi samo blokovi memorije. Ako program pristupi nizu izvan njegovog memorijskog bloka ili koristi memoriju pre inicijalizacije (što su česte greške u programiranju), doći će do nepredvidljivih situacija.

Jedan od osnovnih ciljeva Jave je sigurnost; u njoj se uopšte ne pojavljuju mnogi problemi koji muče programere u C-u i C++-u. U Javi je niz garantovano inicijalizovan i ne može mu se pristupiti van njegovog opsega. Opseg se proverava po cenu male količine dodatnog utroška memorije u svakom nizu, kao i uz proveru indeksa pri izvršavanju, ali se pretpostavlja da su sigurnost i povećana produktivnost vredni toga (a Java katkada može da optimizuje te operacije).

Kada pravite niz objekata, vi, u stvari, pravite niz referenci, i svaka od tih referenci automatski se inicijalizuje na posebnu vrednost. Za tu vrednost koristi se rezervisana reč **null**. Kada Java naiđe na **null**, ona prepoznaje da dotična referenca ne pokazuje na objekat. Pre nego što je upotrebite, svakoj referenci morate da dodelite objekat, i ako pokušate da koristite referencu koja još uvek ima vrednost **null**, problem će biti prijavljen pri izvršavanju. Na taj način su tipične greške s nizovima predupređene u Javi.

Možete da napravite i niz prostih tipova. Ponovo, prevodilac garantuje ispravnu inicijalizaciju jer memoriju za taj niz puni nulama.

Nizovi će biti detaljnije objašnjeni u narednim poglavljima.

Nikada ne morate da uništite objekat

Ostvarenje koncepta životnog veka promenljive u većini programskih jezika vrlo je naporno. Koliko dugo traje promenljiva? Ako treba da je uništite, kada bi to trebalo da uradite? Konfuzija zbog životnog veka promenljive može da dovede do mnogih grešaka, a u ovom odeljku pokazujemo da Java znatno pojednostavljuje ovo pitanje, obavljajući umesto vas sve čišćenje.

Oblast važenja

Većina proceduralnih jezika počiva na *oblasti važenja* (engl. *scope*). Ona određuje vidljivost i životni vek imena definisanih unutar te oblasti. U C-u, C++-u i Javi oblast važenja je određena vitičastim zagradama { }. Na primer:

```
{
  int x = 12;
  // dostupna je samo promenljiva x
  {
    int q = 96;
    // dostupne su i x i q

  // dostupna je samo x
  // q je "izvan oblasti važenja"
}
```

Promenljiva definisana unutar oblasti važenja dostupna je samo do kraja te oblasti.

Sav tekst nakon '//' pa do kraja reda predstavlja komentar.

Uvlačenje pomaže da se program pisan u Javi lakše čita. Pošto je Java jezik slobodne forme, dodatni razmaci, tabulatori i novi redovi ne utiču na rezultujući program.

U Javi *ne možete* da uradite sledeće, iako je to dozvoljeno u C-u i C++-u:

```
{
  int x = 12;
  {
    int x = 96; // nepropisno
  }
}
```

Prevodilac će objaviti da je promenljiva **x** već definisana. Stoga mogućnost C-a i C++-a da sakrije promenljive u većoj oblasti nije dozvoljena, jer su projektanti Jave mislili da to vodi do programa koji zbunjuju.

Oblast važenja objekata

Objekti u Javi nemaju isti životni vek kao prosti tipovi. Kada napravite objekat u Javi koristeći operator **new**, on postoji i nakon kraja oblasti važenja u kojoj je napravljen.

Stoga, ako koristite:

```
{
  String s = new String ("neki znakovni niz");
} // kraj oblasti važenja
```

referenca **s** će nestati na kraju oblasti važenja. Međutim, objekat klase **String** na koji je **s** ukazivala još uvek zauzima memoriju. U ovom deliću koda nema načina da pristupite objektu nakon oblasti važenja, jer je jedina referenca na njega nedostupna izvan oblasti važenja. Dalje u knjizi videćete kako reference na objekte mogu da se prosleđuju i umnožavaju celim tokom programa.

Pošto se objekti napravljeni pomoću operatora **new** zadržavaju dokle god su vam potrebni, ispostavlja se da mnoštvo problema u programiranju u C++-u ne postoji u Javi. U C++-u se morate pobrinuti ne samo za to da objekti postoje dokle god su vam potrebni, nego ih morate i uništiti kada s njima završite.

Zbog toga se postavlja važno pitanje. Ako Java ostavlja objekte da leže unaokolo, šta ih sprečava da prepune memoriju i zaustave program? Upravo bi se ta vrsta problema pojavila u C++-u. Na ovom mestu deluje malo magije. Java ima sakupljač smeća (engl. *garbage collector*); on vodi računa o svim objektima koji su napravljeni operatorom **new** i otkriva one na koje više nema referenci. Zatim oslobađa memoriju koju su zauzimali ti objekti, pa ta memorija može da se koristi za nove objekte. To znači da nikada ne morate sami da se brinete o oslobađanju memorije. Samo napravite objekte, a kada vam više nisu potrebni, oni će sami otići. Ovo eliminiše jednu klasu programskih problema, takozvano curenje memorije (engl. *memory leak*), pri kome programer zaboravi da oslobodi memoriju.

Pravljenje novih tipova podataka: **klasa**

Ako je sve objekat, šta određuje kako određeni objekti izgledaju i kako se ponašaju? Drugim rečima, šta uspostavlja *tip* objekta? Možete očekivati da postoji rezervisana reč „tip", i to bi svakako imalo smisla. Hronološki posmatrano, većina objektno orijentisanih jezika koristila je rezervisanu reč **class** da naznači: „Upravo ću ti reći kako novi tip objekta izgleda". Tu rezervisanu reč (koja je toliko česta da neće biti pisana polucrno dalje kroz knjigu) prati ime novog tipa. Na primer:

```
class NekoImeTipa { /* telo klase ide ovde */ }
```

Ovo uvodi nov tip, iako se telo klase sastoji samo od komentara (zvezdica i kosa crta i ono što je između tih znakova, o čemu će biti reči dalje u ovom poglavlju), pa malo šta možete s njom da uradite. Međutim, možete da napravite objekat ovog tipa koristeći **new**:

```
NekoImeTipa a = new NekoImeTipa();
```

Ali klasi ne možete reći da mnogo toga uradi (tj. ne možete joj poslati neke važne poruke) sve dok ne definišete neke njene metode.

Polja i metode

Kada definišete klasu (a sve što radite u Javi jeste definisanje klasa, pravljenje objekata tih klasa i slanje poruka tim objektima), u nju možete staviti dve vrste elemenata: *polja* (koja se ponekad nazivaju i *podaci članovi*) i *metode* (koje se katkada nazivaju *funkcije članice*). Polje je neki od prostih tipova ili objekat bilo kog tipa, s kojim možete da se povežete putem njegove reference. Reference na objekte morate da incijalizujete (koristeći **new**, kao što ste videli ranije), da biste ih povezali sa stvarnim objektima.

Svaki objekat ima zasebno skladište za svoja polja; obično se polja ne dele između objekata iste klase. Evo primera klase s nekim poljima:

```
class SamoPodaci {
  int i;
  double d;
  boolean b;
}
```

Ova klasa ne *radi* ništa, sem što čuva neke podatke. Ali ovako možete da napravite objekat:

```
SamoPodaci podaci = new SamoPodaci();
```

Poljima možete dodeliti vrednosti, ali prvo morate da znate kako da se obratite članu nekog objekta. To se postiže tako što se navede ime reference na taj objekat, zatim sledi tačka pa ime člana unutar objekta:

```
referencaNaObjekat.polje
```

Na primer:

```
podaci.i = 47;
podaci.d = 1.1;
podaci.b = false;
```

Vaš objekat može da sadrži i druge objekte koji sadrže podatke. Samo nastavite da „nadovezujete tačke".

Na primer:

```
mojAvion.leviRezervoar.kapacitet = 100;
```

Klasa **SamoPodaci** ne može skoro ništa da uradi osim da čuva podatke, jer nema metode. Da biste shvatili metode, prvo morate da naučite šta su *argumenti* i *povratne vrednosti* (ubrzo će biti opisani).

Podrazumevane vrednosti za podatke prostog tipa

Ako podatak prostog tipa ne inicijalizujete, on će dobiti podrazumevanu vrednost:

Prost tip	Podrazumevana vrednost
boolean	false
char	"\u0000" (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Dobro obratite pažnju na to da Java jamči dodelu podrazumevanih vrednosti samo kada se promenljiva koristi kao *članica klase*. To obezbeđuje da promenljive članice prostog tipa uvek budu inicijalizovane (to C++ ne radi), čime se smanjuje izvor grešaka. Ipak, ova početna vrednost ne mora da bude ispravna niti odgovarajuća za program koji pišete. Najbolje je da uvek izričito inicijalizujete svoje promenljive.

Ova garancija se ne odnosi na *lokalne* promenljive – one koje nisu polja klase. Stoga, ako unutar definicije metode imate:

```
int x;
```

x će imati proizvoljnu vrednost (kao i u C-u i C++-u) i neće automatski biti inicijalizovan na nulu. Pre nego što upotrebite **x**, sami mu morate dodeliti odgovarajuću vrednost. Ako ste zaboravili, Java je uvela poboljšanje u odnosu na C++: dobićete grešku pri prevođenju koja vam govori da promenljiva možda nije bila inicijalizovana. (Mnogi C++ prevodioci će vas upozoriti na neinicijalizovane promenljive, ali u Javi se one smatraju greškama.)

Metode, argumenti i povratne vrednosti

U mnogim jezicima (recimo, C-u i C++-u), termin *funkcija* opisuje imenovani potprogram. U Javi se češće koristi termin *metoda*, kao „način da se nešto uradi". Ako želite, možete nastaviti da razmišljate o „funkcijama", ali će se u ovoj knjizi nadalje koristiti u Javi uobičajen termin „metoda".

Metode u Javi određuju poruke koje objekat može da primi. Osnovni delovi metode su ime, argumenti, povratni tip i telo. Osnovni oblik ovako izgleda:

```
PovratniTip imeMetode( /* lista argumenata */ ) {
  /* telo metode */
}
```

Povratni tip opisuje tip vrednosti koja se vraća kao rezultat metode koju ste pozvali. Lista argumenata zadaje tipove i imena informacija koje želite da prosledite metodi. Ime metode i lista argumenata (koji zajedno čine njen *potpis*) jedinstveno identifikuju metodu.

Metode u Javi mogu biti napravljene samo kao deo klase. Metoda može da se poziva samo za neki objekat,[1] i taj objekat mora da bude sposoban da izvrši poziv. Ako pokušate da pozovete pogrešnu metodu objekta, dobićete poruku o grešci prilikom prevođenja. Metodu objekta pozivate tako što navedete ime objekta iza koga je tačka, zatim ime metode i njenu listu argumenata, recimo ovako:

```
imeObjekta.imeMetode(arg1, arg2, arg3);
```

Na primer, pretpostavimo da imate metodu **f()** koja nema argumente i vraća vrednost tipa **int**. Zatim, ako imate objekat pod imenom **a** koji ima metodu **f()**, možete napisati sledeće:

```
int x = a.f();
```

Tip povratne vrednosti mora da bude kompatibilan s tipom promenljive **x**.

Ovo pozivanje metode često se naziva i *slanje poruke objektu*. U prethodnom primeru, poruka je **f()** i objekat je **a**. Objektno orijentisano programiranje često se ukratko predstavlja kao jednostavno „slanje poruka objektima".

---

[1] **static** metode, o kojima ćete uskoro učiti, mogu biti pozvane *za klasu*, bez objekta.

Lista argumenata

Lista argumenata metode određuje koje informacije prosleđujete metodi. Kao što pretpostavljate, te informacije – kao i sve drugo u Javi – imaju formu objekata. Stoga u listi argumenata morate da navedete tipove objekata koji se prosleđuju i imena koja će se koristiti za svaki od njih. Kao i u drugim situacijama u Javi gde se čini da radite sa objektima, vi, u stvari, prosleđujete reference.[4] Tip reference mora da bude ispravan. Ako argument treba da bude objekat klase **String**, onda morate da prosledite objekat tipa **String** ili će prevodilac prijaviti grešku.

Razmotrimo metodu čiji argument je objekat tipa **String**. Sledi definicija metode koja mora biti postavljena unutar definicije klase da bi bila prevedena:

```
int skladiste(String s) {
  return s.length() * 2;
}
```

Ova metoda pokazuje koliko bajtova je potrebno za čuvanje informacije u određenom objektu tipa **String.** (Svaki **znak** u objektu tipa **String** dugačak je 16 bitova, odnosno dva bajta, zbog podržavanja standarda Unicode.) Argument je objekat tipa **String** pod imenom **s**. Kada se **s** prosledi u metodu, možete ga tretirati kao i svaki drugi objekat. (Možete mu slati poruke.) Ovde se poziva metoda **length()**, kao jedna od metoda klase **String**; ona vraća broj znakova u znakovnom nizu.

Obratite pažnju na upotrebu rezervisane reči **return** koja obavlja dva zadatka. Prvo, ona znači „napusti metodu, završio sam". Drugo, ako metoda proizvodi vrednost, ta vrednost se stavlja odmah iza rezervisane reči **return**. U ovom slučaju, povratna vrednost se proizvodi izračunavanjem izraza **s.length()** * 2.

Možete da vratite podatak bilo kog tipa, ali ako ne želite ništa da vratite, onda naznačite da metoda vraća **void**. Evo primera:

```
boolean indikator() { return true; }
double osnovaPrirodnogLog() { return 2.718; }
void nista() { return; }
void nista2() { }
```

Kada je povratni tip **void**, rezervisana reč **return** se koristi samo da bi se napustila metoda i stoga je nepotrebna kada se dođe do kraja metode. Iz metode se možete vratiti iz bilo koje tačke, ali ako ste naveli povratni tip koji nije **void**, prevodilac će vas, bez obzira na mesto sa kog vraćate, naterati (porukama o greškama) da vratite odgovarajući tip vrednosti.

U ovom trenutku možda izgleda da je program samo gomila objekata sa metodama čiji su argumenti drugi objekti i koji šalju poruke tim drugim objektima. To se uglavnom i dešava, ali u narednom poglavlju naučićete kako da uradite posao niskog nivoa, donošenjem odluka unutar metode. Ako u ovom poglavlju savladate slanje poruka – biće dovoljno.

---

[4] Uz uobičajeno izuzimanje ranije pomenutih prostih tipova podataka **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** i **double**. Po pravilu, vi prosleđujete objekte, što znači da prosleđujete reference na objekte.

Pravljenje programa na Javi

Postoji još nekoliko oblasti koje morate savladati pre nego što sastavite svoj prvi program u Javi.

Vidljivost imena

Kontrola imena je problem u svim programskim jezicima. Ako koristite ime u jednom delu programa, a drugi programer koristi isto ime u drugom delu, kako da razlikujete jedno ime od drugog i sprečite da se „sudare"? Ovo je posebno ozbiljan problem u C-u jer program često sadrži more imena nepodesnih za rad. C++ klase (na kojima su zasnovane Java klase) funkcije ugnežđuju unutar klasa, pa zato i ne mogu da se sudare sa imenima funkcija ugnežđenim unutar drugih klasa. Međutim, u C++-u se i dalje koriste globalni podaci i globalne funkcije pa je sudaranje još uvek moguće. Da bi se rešio taj problem, u C++ je pomoću dodatnih rezervisanih reči uveden *imenski prostor* (engl. *namespace*).

Zahvaljujući novom pristupu, Java je sve to mogla da izbegne. Za pravljenje jednoznačnih imena za biblioteke, projektanti Jave žele da koristite ime vašeg Internet domena u obrnutom poretku, pošto su ta imena sigurno jedinstvena. Pošto je ime mog domena **MindView.net**, moja uslužna biblioteka za otklanjanje nedostataka dobila bi ime **net.mindview.utility.foibles**. Iza obrnutog imena domena, ostale tačke treba da predstavljaju poddirektorijume.

U Javi 1.0 i Javi 1.1, oznake domena **com**, **edu**, **org**, **net** itd. po konvenciji su pisane velikim slovima, pa bi se biblioteka zvala: **NET.mindview.utility.foibles**. Tokom razvoja Jave 2 otkriveno je da to izaziva probleme, pa se sada celo ime paketa piše malim slovima.

Ovaj mehanizam znači da sve vaše datoteke automatski žive u svojim imenskim prostorima i da svaka klasa unutar datoteke garantovano ima jedinstven identifikator – o tome se jezik brine umesto vas.

Korišćenje drugih komponenata

Kad god želite da koristite unapred definisane klase u svom programu, prevodilac mora znati da ih pronađe. Naravno, klase mogu već da postoje u istoj datoteci iz koje se i pozivaju. U tom slučaju, samo koristite klasu – čak i ako se ona definiše tek kasnije u datoteci. Java eliminiše problem s takozvanim isturenim referencama.

Šta ako klasa postoji u nekoj drugoj datoteci? Pomislićete kako bi prevodilac trebalo da bude dovoljno pametan i da krene i nađe je, ali tu postoji problem. Zamislite da hoćete da koristite klasu određenog imena, ali za nju postoji više od jedne definicije (pod pretpostavkom da su različite). Ili još gore, zamislite da pišete program i u svoju biblioteku dodajete novu klasu, čije se ime sukobljava sa imenom neke postojeće klase.

Da biste rešili ovaj problem, morate da eliminišete sve potencijalne dvosmislenosti. To se postiže tako što pomoću rezervisane reči **import**, Java prevodiocu saopštite koje klase želite da koristite. **import** govori prevodiocu da uveze *paket* (engl. *package*) koji predstavlja biblioteku klasa. (U drugim jezicima, biblioteka pored klasa može da sadrži funkcije i podatke, ali setite se da u Javi sav kôd mora biti pisan unutar klasa.)

Najčešće ćete koristiti komponente iz standardnih biblioteka koje se isporučuju zajedno s prevodiocem. Uz njih, ne morate da brinete o dugačkim, invertovanim imenima domena; samo napišete, na primer:

```
import java.util.ArrayList;
```

kako biste saopštili prevodiocu da želite da koristite Javinu klasu **ArrayList**. Međutim, paket **util** sadrži više klasa i možete poželeti da koristite nekoliko njih a da ih pri tom izričito ne deklarišete. To se lako postiže oznakom * za slobodan izbor:

```
import java.util.*;
```

Skupove klasa češće ćete uvoziti na ovaj način nego pojedinačno.

Rezervisana reč **static**

Kada napravite klasu, obično opisujete kako objekti te klase izgledaju i kako će se ponašati. Objekat zapravo ne dobijate sve dok ga ne napravite operatorom **new** – u toj tački se objektu dodeljuje prostor u memoriji (skladište) i tek tada metode postaju dostupne.

Postoje dve situacije u kojima ovaj pristup nije dovoljan. Jedna je ako želite da imate isto skladište za određeno polje, bez obzira na to koliko objekata te klase napravite, ili čak želite da skladište postoji iako nije napravljen nijedan objekat. Druga je ako vam treba metoda koji nije pridružena nijednom određenom objektu te klase. Odnosno, treba vam metoda koju možete da pozovete iako nijedan objekat nije napravljen.

Oba ova efekta možete postići pomoću rezervisane reči **static**. Kada nešto označite kao **static**, to znači da određeno polje ili metoda nisu povezani ni s jednim objektom te klase. Stoga možete da pozovete metodu **static** ili da pristupite polju **static**, iako nikada niste napravili objekat te klase. U slučaju uobičajenih, nestatičnih polja i metoda, morate da napravite objekat i da ga koristite za pristup polju ili metodi.[3]

U nekim objektno orijentisanim jezicima koriste se termini *podaci klase* i *metode klase*, što znači da podaci i metode postoje samo za klasu u celini, a ne i za određene objekte klase. Ponekad se u literaturi o Javi takođe koriste ti termini.

Da biste polje ili metodu proglasili za statične, stavite rezervisanu reč **static** pre definicije. Na primer, sledeći oblik proizvodi i inicijalizuje statično polje:

```
class StaticTest {
  static int i = 47;
}
```

Čak i ako posle ovoga napravite dva objekta klase **StaticTest**, i dalje će postojati samo jedan primerak skladišta za promenljivu **StaticTest.i**. Oba objekta će deliti istu promenljivu **i**.

Naravno, pošto **static** metode ne zahtevaju da ijedan objekat bude napravljen pre nego što će se koristiti, one ne mogu *direktno* da pristupe nestatičnim članicama ili metodama. Pošto nestatični članovi i metode moraju biti povezani sa odgovarajućim objektom, iz statičnih metoda možete pristupati nestatičnim članovima samo preko nekog imenovanog objekta.

Razmotrimo sledeće:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

U ovom trenutku i **st1.i** i **st2.i** imaju istu vrednost, 47, pošto se odnose na isti deo memorije.

Postoje dva načina za pristup statičnoj promenljivoj. Kao što je pokazano u prethodnom primeru, možete joj pristupati preko objekta, na primer **st2.i**. Možete joj pristupati i direktno preko imena klase, što ne možete da uradite sa nestatičnom članicom.

```
StaticTest.i++;
```

Operator ++ uvećava promenljivu za 1. U ovom trenutku i **st1.i** i **st2.i** imaće i vrednost 48.

Navođenje imena klase je poželjniji način za pristupanje **statičnim** članovima. Ne samo da se time naglašava njihova statična priroda, nego se u nekim slučajevima prevodiocu daje veća mogućnost za optimizovanje.

Slična logika se primenjuje i na statične metode. Statičnoj metodi možete da pristupate bilo preko objekta kao i svakoj metodi, ili pomoću posebne dodatne sintakse **ImeKlase.metoda()**. Statična metoda se definiše na sličan način:

```
class MozeSeUvecati {
  static void uvecaj() { StaticTest.i++; }
}
```

Obratite pažnju na to da u klasi **MozeSeUvecati** metoda **uvecaj()** pomoću operatora ++ uvećava statičan podatak **i**. Metodu **uvecaj()** možete pozvati na uobičajen način, preko objekta:

```
MozeSeUvecati sp = new MozeSeUvecati();
sp.uvecaj();
```

Pošto je **uvecaj()** statična metoda, možete je pozvati direktno preko njene klase:

```
MozeSeUvecati.uvecaj();
```

Iako rezervisana reč **static**, primenjena na polje, definitivno menja način na koji se podatak stvara (po jedan za svaku klasu u odnosu na po jedan nestatičan za svaki objekat), kada se primeni na metodu, promene nisu tako velike. Metode se označavaju rezervisanom rečju **static** kako bi se mogle pozvati iako nije napravljen objekat. Ovo je veoma važno, kao što ćemo videti, za definisanje metode **main()** koja je početna tačka za pokretanje svih aplikacija.

Vaš prvi program na Javi

Konačno, evo prvog potpunog programa. On počinje ispisivanjem znakovnog niza, i datuma, koristeći klasu **Date** iz standardne Java biblioteke.

```
// Zdravo.java
import java.util.*;

public class Zdravo {
  public static void main(String[] args) {
    System.out.println("Zdravo, danas je: ");
    System.out.println(new Date());
  }
}
```

Na početku svake datoteke s programom, morate da navedete sve potrebne naredbe **import** da biste uvezli sve potrebne dodatne klase. Obratite pažnju na to da sam rekao „dodatne" jer postoji jedna biblioteka klasa koja se automatski uvozi u svaki Java program: **java.lang**. Pokrenite svoj Web čitač i pogledajte dokumentaciju o Javinom razvojnom okruženju (JDK). (Ako dokumentaciju o JDK-u niste već preuzeli s lokacije *http://java.sun.com*, uradite to sada.[6] Imajte u vidu da se ova dokumentacija ne isporučuje zajedno s JDK-om, pa je morate zasebno preuzeti.) U listi paketa videćete razne biblioteke klasa koje se isporučuju u paketu s Javinim razvojnim okruženjem. Izaberite paket **java.lang** i dobićete listu svih klasa koje su deo te biblioteke. Pošto je paket **java.lang** implicitno uključen u svaku datoteku s Java kodom, te klase su automatski dostupne. Klasa **Date** nije deo paketa **java.lang**, što znači da je morate uvesti kako biste je koristili. Ako ne znate u kojoj se biblioteci određena klasa nalazi, ili ako želite da vidite sve klase, odaberite „Tree" u dokumentaciji o Javi. Sada ćete videti sve klase Javine standardne biblioteke. Zatim možete iskoristiti funkciju čitača „find" da pronađete klasu **Date**. Kada to uradite, videćete da se ona na listi nalazi kao **java.util.Date**, što znači da je u paketu **util** i da morate da zadate **import java.util.*** kako biste koristili klasu **Date**.

Ako se vratite na početak, izaberete **java.lang** i zatim **System**, videćete da klasa **System** ima nekoliko polja, a ako odaberete **out**, otkrićete da je to statičan objekat klase **PrintStream**. Pošto je on statičan, ne morate ništa da pravite pomoću **new**. Objekat **out** je uvek tu i možete ga jednostavno koristiti. Ono što možete da radite sa ovim objektom **out**, određeno je njegovim tipom: **PrintStream**. Praktičnosti radi, **PrintStream** je u opisu prikazan kao hiperveza, stoga ako ga pritisnete, videćete listu svih metoda koje možete pozvati za tu klasu. Njih ima relativno mnogo i biće objašnjene dalje u knjizi. Zasad nas zanima samo **println**(), što znači „ispiši ono što ti dajem na konzoli i pređi u novi red". Dakle, kad god u bilo kojem programu nešto želite da ispišete na konzoli, možete napisati:

```
System.out.println("Znakovni niz za izlaz")
```

Ime klase i ime datoteke su isti. Kada pravite nezavisan program kao što je ovaj, jedna od klasa mora da ima isto ime kao datoteka. (Prevodilac se buni ako tako ne uradite.) Ta klasa mora da sadrži metodu pod imenom **main**() sa sledećim potpisom i povratnim tipom:

```
public static void main (String[] args) {
```

[6] Prevodilac za Javu i Sunova dokumentacija stalno se menjaju i najbolje ih je preuzimati neposredno od Suna. Ako je sami preuzmete, dobićete najnoviju verziju.

Rezervisana reč **public** označava da je metoda javno dostupna (što je detaljno opisano u poglavlju *Kontrola pristupa*). Argument metode **main()** jeste niz objekata klase **String**. Argumente **args** ne koristimo u ovom programu, ali prevodilac insistira da se tu nalaze jer se u njih smeštaju argumenti prosleđeni s komandne linije.

Red sa ispisom datuma prilično je zanimljiv:

```
System.out.println(new Date());
```

Argument je objekat klase **Date** koji se pravi samo da bi poslao svoju vrednost (koja se automatski pretvara u objekat tipa **String**) metodi **println()**. Čim se ovaj izraz završi, taj objekat više nije potreban i sakupljač smeća može da prođe i pokupi ga u bilo kom trenutku. Mi ne moramo da vodimo računa o njegovom čišćenju.

Kada proučite dokumentaciju o JDK-u sa lokacije *http://java.sun.com*, videćete da **System** ima mnoge druge metode koje omogućavaju izvođenje zanimljivih efekata. (Jedna od najvećih prednosti Jave jesu njene brojne standardne biblioteke.) Na primer:

```
//: object/ShowProperties.java

public class ShowProperties {
   public static void main(String[] args)  {
      System.getProperties() .list(System.out);
      System.out.println(System.getProperty("user.name"));
      System.out.println(
         System.getProperty("java.library.path"));
   }
}  ///:~
```

Prvi red u metodi **main()** prikazuje sva svojstva (engl. *properties*) sistema na kojem izvršavate program, dakle daje podatke o okruženju. Metoda **list()** šalje te rezultate svom argumentu **System.out**. Videćete u nastavku knjige da ih možete poslati i na druga mesta, recimo u neku datoteku. Možete zatražiti i određeno svojstvo – u ovom slučaju, korisničko ime (engl. *user name*) i putanju (engl. *path*) do Javine biblioteke (engl. *library*). (Malo kasnije objasnićemo neobične komentare na početku i na kraju.)

Prevođenje i izvršavanje

Da biste preveli i pokrenuli ovaj program, kao i sve druge u ovoj knjizi, prvo morate da imate razvojno okruženje za Javu. Postoji više nezavisnih razvojnih okruženja, ali pretpostavićemo da koristite besplatno razvojno okruženje JDK, kompanije Sun Microsystems. Ako koristite neki drugi razvojni sistem,[1] moraćete da pogledate dokumentaciju za taj sistem da biste saznali kako da prevodite i pokrećete programe.

Povežite se na Internet i pogledajte Web lokaciju *http://java.sun.com*. Tamo ćete naći informacije i veze koje će vas voditi kroz proces preuzimanja i instaliranja JDK-a za vašu platformu.

[1] Obično je to IBM-ov prevodilac jikes, koji je znatno brži od Sunovog javac (iako razlika nije velika kada pomoću *Anta* pravite grupe datoteka). Postoje i programi otvorenog koda za pravljenje Java prevodilaca, izvršnih okruženja i biblioteka.

Kada instalirate JDK i podesite putanje na svom računaru tako da programi **javac** i **java** budu dostupni, preuzmite i raspakujte izvorni kôd za ovu knjigu (možete ga naći na adresi *www.MindView.net*). Tako ćete napraviti poddirektorijum za svako poglavlje iz ove knjige. Idite u poddirektorijum **object** i upišite:

```
javac HelloDate.java
```

Ova komanda ne bi trebalo da proizvede nikakav odziv. Ako dobijete bilo kakvu poruku o grešci, znači da niste propisno instalirali JDK i treba da ispitate taj problem.

S druge strane, ako dobijete nazad samo komandnu liniju, možete da otkucate:

```
java HelloDate
```

i kao rezultat dobićete poruku i datum.

Navedeni proces možete da koristite za prevođenje i pokretanje svakog programa iz ove knjige. Međutim, videćete da izvorni kôd ove knjige u svakom poglavlju takođe ima i datoteku pod imenom **bild.xml**, koja sadrži „Ant" komande za automatsko prevođenje datoteka iz tog poglavlja. Automatski prevedene datoteke i Ant (kao i to odakle da ga preuzmete) detaljnije su opisani u dodatku koji ćete naći na *http://MindView.net/Books/BetterJava*, ali nakon što instalirate Ant (sa *http://jakarta.apache.org/ant*), dovoljno je da otkucate '**ant**' nakon komandnog odzivnika i programi iz svakog poglavlja biće prevedeni i pokrenuti. Ukoliko još niste instalirali Ant, komande **javac** i **java** otkucajte ručno.

Komentari i ugrađena dokumentacija

Postoje dva tipa komentara u Javi. Prvi je tradicionalni stil pisanja komentara jezika C, koji je nasledio i C++. Ti komentari počinju sa /* i nastavljaju se, moguće i u više redova, sve do */. Obratite pažnju na to da mnogi programeri svaki red komentara nastavljaju sa *. Često ćete videti:

```
/* Ovo je komentar
 * koji se nastavlja
 * u više redova
 */
```

Dobro zapamtite da se sve što se nalazi unutar /* i */ ignoriše, stoga nema razlike ako napišete:

```
/* Ovo je komentar koji
se nastavlja u više redova */
```

Druga vrsta komentara potiče iz C++-a. To je komentar u jednom redu, koji počinje sa // i nastavlja se do kraja reda. Ovaj način komentara je praktičan i često se koristi jer je jednostavan. Ne morate da lovite po tastaturi i da tražite / a zatim * (umesto toga, samo dvaput pritisnite isti taster) i ne morate da zatvorite komentar. Zato ćete često susretati:

```
// ovo je komentar u jednom redu
```

Dokumentacioni komentari

Verovatno da je najveći problem pri dokumentovanju koda bilo održavanje dokumentacije. Ako su kôd i dokumentacija razdvojeni, postaje nezgodno menjati dokumentaciju svaki put kada promenite kôd. Rešenje izgleda jednostavno: povežite kôd i dokumentaciju. Najlakše ćete to uraditi kad sve stavite u istu datoteku. Da biste zaokružili postupak, potrebna je posebna sintaksa za dokumentacione komentare, kao i alat kojim ćete te komentare izdvojiti i prebaciti ih u koristan oblik. To je uradila Java.

Alat za izdvajanje komentara naziva se *Javadoc* i deo je instalacije JDK-a. On koristi neke od tehnologija Java prevodioca da potraži posebne oznake komentara u programu. Izdvaja označene informacije i izvlači ime klase ili metode koja ide uz taj komentar. Na ovaj način možete sa minimalnom količinom rada generisati pristojnu dokumentaciju programa.

Rezultat ovog postupka je HTML datoteka koju možete da pregledate pomoću čitača Weba. Tako Javadoc omogućava da napravite i održavate samo jednu izvornu datoteku i automatski generišete korisnu dokumentaciju. Zahvaljujući programu *Javadoc* imamo jednostavan standard za pravljenje dokumentacije, pa možemo da očekujemo ili čak zahtevamo dokumentaciju iz svih Java biblioteka.

Uza sve, možete napisati sopstvene Javadoc identifikatore (engl. *handlers*), *doclete*, ukoliko komentare koje je izdvojio Javadoc hoćete posebno da obradite (recimo, da ih ispišete u drugačijem formatu). O docletima čitajte u dodatku na *http://MindView.net/Books/BetterJava*.

Sledi uvod u Javadoc i pregled njegovih obeležja. Potpun opis naći ćete u dokumentaciji o JDK-u. Kada je raspakujete, potražite poddirektorijum „tooldocs" ili pritisnite istoimenu hipervezu.

Sintaksa

Sve komande programa *Javadoc* nalaze se isključivo unutar komentara /**. Komentar se završava sa */ kao i obično. Postoje dva osnovna načina za korišćenje ovog sistema: ugrađeni HTML ili korišćenje dokumentacionih oznaka (engl. *doc tags*). *Samostojeće dokumentacione oznake* su komande koje počinju sa @ i koje se nalaze na početku reda komentara. (Zanemaruje se vodeća zvezdica.) *Nesamostalne dokumentacione oznake* mogu biti napisane bilo gde unutar Javadoc komentara i takođe počinju sa @, ali se nalaze između vitičastih zagrada.

Postoje tri tipa dokumentacionih komentara koji odgovaraju elementu kome komentar prethodi: klasi, polju ili metodi. Komentar klase stoji odmah ispred definicije klase; komentar polja se pojavljuje odmah ispred definicije polja, a komentar metode se javlja neposredno ispred definicije metode. Ovo je jednostavan primer:

```
//: object/Documentation1.java
/** Komentar klase */
public class Documentation1.java {
  /** Komentar polja */
  public int i;
  /** Komentar metode */
  public void f() {}
} ///: ~
```

Obratite pažnju na to da će Javadoc obraditi dokumentacione komentare samo za **javne** i **zaštićene** članice. Komentari za **privatne** članice i članice kojima se pristupa u paketu (videti poglavlje *Kontrola pristupa*) ignorišu se i nećete videti nikakav izlaz. (Međutim, možete koristiti indikator **-private** da biste uključili i **privatne** članice.) Ovo ima smisla, pošto su samo **javne** i **zaštićene** članice dostupne izvan datoteke, što je i perspektiva programera klijenata.

Rezultat obrade prethodnog primera je HTML datoteka koja ima isti standardni format kao i celokupna ostala dokumentacija, stoga će korisnicima delovati poznato i moći će lako da se kreću kroz klase. Prekucajte taj primer, propustite ga kroz *Javadoc* i pregledajte rezultujuću HTML datoteku, da biste sami videli opisane rezultate.

Ugrađeni HTML

Javadoc prepisuje HTML oznake iz komentara u HTML dokument koji generiše. To omogućava potpuno korišćenje HTML-a; međutim, osnovni cilj je da se omogući formatiranje koda, kao što je:

```
//: object/Documentation2.java
/**
* <pre>
*System.out.println(new Date());
* </pre>
*/
///: ~
public class Documentation2 {}
```

Takođe, HTML možete koristiti kao i u svakom drugom Web dokumentu, da formatirate običan tekst u svojim opisima:

```
//: object/Documentation3.java
/**
* Možete ubaciti <em>čak</em> i listu:
* <ol>
* <li> Prva tačka
* <li> Druga tačka
* <li> Treća tačka
* </ol>
*/
public class Documentation3 {}
```

Obratite pažnju na to da unutar dokumentacionog komentara, Javadoc odbacuje zvezdice na početku reda, kao i vodeće praznine. Javadoc sve ponovo formatira tako da se prilagodi standardnom izgledu dokumentacije. Ne koristite naslove kao što su <**h1**> ili oznake kao <**hr**>, jer Javadoc ubacuje svoje naslove i vaši će ih ometati.

Svi tipovi dokumentacionih komentara – komentar klase, polja i metode – podržavaju ugrađeni HTML.

Primeri oznaka

Evo primera nekih Javadoc oznaka koje se mogu stavljati u dokumentaciju koda. Pre nego što pomoću Javadoca pokušate da uradite bilo šta ozbiljno, trebalo bi da pročitate njemu posvećen odeljak u dokumentaciji JDK-a i tamo vidite sve moguće načine upotrebe Javadoca.

@see:

Ova oznaka služi za upućivanje na dokumentaciju u drugim klasama. Javadoc će generisati HTML sa oznakama **@see** kao hipervezama povezanim s drugom dokumentacijom. Oblici su:

```
@see imeklase
@see potpuno-opisano-imeklase
@see potpuno-opisano-imeklase#ime-metode
```

Svaki od njih dodaje generisanoj dokumentaciji hipervezu „Pogledaj takođe" (engl. *See Also*). Javadoc ne proverava ispravnost hiperveza koje joj predajete.

{@link *paket.klasa#član labela*}

Veoma slično oznaci **@see**, sem što se može koristiti direktno u istom nivou i što kao tekst hiperveze ispisuje *labelu*, a ne „Pogledaj takođe".

{@docRoot}

Daje relativnu putanju do korenskog direktorijuma dokumentacije. Koristi se za izričito hiperpovezivanje sa stranicama u stablu dokumentacije.

{@inheritDoc}

Tekući dokumentacioni komentar nasleđuje dokumentaciju najbliže osnovne klase ove klase.

@version

Njen oblik je:

```
@version informacija-o-verziji
```

gde je **informacija-o-verziji** bilo koja bitna informacija koju treba uključiti. Kada se indikator **–version** navede na komandnoj liniji programa *Javadoc*, informacija o verziji će biti prosleđena u generisanu HTML dokumentaciju.

@author

Njen oblik je:

```
@author informacija-o-autoru
```

gde je **informacija-o-autoru** verovatno vaše ime ali može da bude i vaša elektronska adresa ili bilo koja druga prigodna informacija. Kada se indikator **–author** navede u komandnoj liniji programa *Javadoc*, informacija o autoru biće prosleđena u generisanu HTML dokumentaciju.

Možete imati više autorskih oznaka za listu autora, ali one moraju da budu poređane uzastopno. Sve informacije o autorima biće spojene u jedan pasus u generisanom HTML-u.

@since

Ova oznaka omogućava da naznačite verziju klase koja je počela da koristi određenu mogućnost. Videćete da se pojavljuje u Java HTML dokumentaciji da naznači koja je verzija JDK-a korišćena.

@param

Koristi se za dokumentovanje metoda u obliku:

```
@param ime-parametra opis
```

gde je **ime-parametra** identifikator u listi parametara metode, a **opis** tekst koji se može protegnuti na nekoliko redova. Smatra se da je opis završen kada se naiđe na novu dokumentacionu oznaku. Možete imati koliko hoćete ovih oznaka, najčešće po jednu za svaki parametar.

@return

Koristi se za dokumentovanje metoda, a piše se u obliku:

```
@return opis
```

gde **opis** daje značenje povratne vrednosti. On može da se protegne na nekoliko redova.

@throws

O izuzecima govorimo u poglavlju *Obrada grešaka pomoću izuzetaka*. Ukratko, to su objekti koji mogu biti „bačeni" iz metode u slučaju greške. Iako samo jedan objekat izuzetka može da se pojavi kada pozovete metodu, određena metoda može da proizvede više različitih tipova izuzetaka a svi moraju biti unapred naznačeni. Stoga oznaka za izuzetak ima sledeći oblik:

```
@throws potpuno-opisano-imeklase opis
```

gde **potpuno-opisano-imeklase** nedvosmisleno daje ime klase izuzetka koja je negde definisana, a **opis** (koji može da se protegne na nekoliko redova) pokazuje zbog čega određeni tip izuzetka može da se pojavi pri pozivu metode.

@deprecated

Ovo se koristi da naznači mogućnosti koje su zastarele. Oznaka **@deprecated** pokazuje da više ne treba da koristite tu mogućnost, pošto će uskoro verovatno biti uklonjena. Prevodilac će vas upozoriti ako koristite metodu označenu sa **@deprecated**. U Javi SE5, **@deprecated** je zamenjena *anotacijom* **@Deprecated** (anotacijama je posvećeno poglavlje *Anotacije*).

Primer dokumentacije

Ponovo dajemo prvi program na Javi, ovog puta s dodatnim dokumentacionim komentarima:

```
//: object:Zdravo.java
import java.util.*;

/** Prvi primer programa iz knjige Misliti na Javi.
 * Ispisuje znakovni niz i današnji datum.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
*/
public class Zdravo {
 /** Ulazna tačka u klasu i aplikaciju
  * @param args niz argumenata tipa string
  * @throws exceptions ne baca izuzetke
 */
 public static void main(String[] args) {
  System.out.println("Zdravo, danas je: ");
  System.out.println(new Date());
 }
} /* Ispis: (55% match)
Zdravo, danas je:
Tue Feb 06 14:39:36 MDT 2007
* ///:~
```

U prvom redu datoteke koristi se moja tehnika: stavlja se **//:** kao posebne oznake za red s komentarom koji sadrži ime izvorne datoteke. Taj red sadrži informaciju o putanji do datoteke (**object** označava ovo poglavlje), iza koje sledi ime datoteke. Poslednji red takođe se završava komentarom (`///:~`) koji označava kraj listinga izvornog koda i omogućava da se taj kod automatski ažurira u tekstu ove knjige (nakon što ga prevodilac proveri) i da se izvrši.

Oznaka **/* Ispis:** naznačuje početak izlaza koji će ovaj program generisati. U ovom obliku, **(55% match)** naznačuje sistemu za testiranje da se izlaz prilično razlikuje od jednog do drugog izvršavanja i da treba da očekuje korelaciju od samo 55 procenata sa ovde prikazanim izlazom. Od onih primera u ovoj knjizi koji imaju neki izlaz, većina će sadržati ovaj oblik izlaza, pa ćete moći da izvršite svaki program i da proverite da li mu je izlaz ispravan.

Stil programiranja

Prema stilu opisanom u knjizi *Code Conventions for the Java Programming Language*[8] prvo slovo imena klase treba da bude veliko. Ako se ime klase sastoji od nekoliko reči, one se pišu zajedno (tj. ne koristite donje crte da razdvojite imena), i prvo slovo svake ugrađene reči je veliko, na primer:

```
class SveBojeDuge { // ...
```

Tako napisano ime klase liči na grbe kamile. Za skoro sve ostalo: metode, polja (promenljive članice) i imena referenci na objekte, usvojeni stil je isti kao i za klase, osim što se prvo slovo identifikatora piše malo. Na primer:

```
class SveBojeDuge {
  int ceoBrojKojiPredstavljaBoje;
  void promeniNijansuBoje (int novaNijansa) {
    // ...
  }
  // ...
}
```

Mislite na to kako korisnik takođe mora da kuca sva ova dugačka imena, stoga imajte milosti.

U Java kodu iz Sunovih biblioteka koristi se isti stil pisanja otvorenih i zatvorenih vitičastih zagrada poput stila upotrebljenog u ovoj knjizi.

Sažetak

Cilj ovog poglavlja je da naučite tek toliko Jave koliko je dovoljno da biste napisali jednostavan program. Stekli ste i uvid u jezik i njegove osnovne pojmove. Do sada su ipak svi primeri bili u znaku „uradi ovo, zatim uradi ono, a potom nešto treće". U naredna dva poglavlja upoznaćete osnovne operatore koji se upotrebljavaju u programiranju na Javi i naučićete da upravljate tokom programa.

Vežbe

Ubuduće će vežbe biti raspodeljene po celom tekstu poglavlja, ali u ovom su sve vežbe stavljene na kraj, pošto ste tek naučili da pišete najjednostavnije programe.

Iza rednog broja vežbe nalazi se broj u zagradama, koji u opsegu od 1 do 10 pokazuje njenu težinu.

Rešenja odabranih vežbi nalaze se u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na adresi *www.MindView.net*.

---

[8] *http://java.sun.com/docs/codeconv/index.html*. Nisam mogao da sledim sve smernice iz ove preporuke jer bi mi to uzelo previše prostora u knjizi i na prezentacijama, ali uverićete se da ovde upotrebljeni stil odgovara standardu Jave koliko god je to moguće.

Vežba 1: (2) Napišite klasu koja će sadržati neinicijalizovana polja tipa **int** i **char**, i ispišite njihove vrednosti kako biste se uverili da ih Java podrazumevano inicijalizuje.

Vežba 2: (1) Nakon primera **Zdravo.java** u ovom poglavlju, napravite program „zdravo, svete" koji prikazuje datu naredbu na ekranu. Potrebna vam je samo jedna metoda u klasi (main koja se izvršava kada se program pokrene). Setite se da treba da ostane **statična** i da uključite listu argumenata, iako je nećete koristiti. Prevedite program komandom **javac** i pokrenite ga komandom **java**. Ako koristite razvojno okruženje koje je drugačije od JDK-a, naučite kako da prevedete i izvršite programe u tom okruženju.

Vežba 3: (1) Pronađite delove koda koji obuhvataju klasu **NekoImeTipa** i pretvorite ih u program koji možete da prevedete i izvršite.

Vežba 4: (1) Pretvorite deliće koda koji obuhvataju klasu **SamoPodaci** u program koji možete da prevedete i izvršite.

Vežba 5: (1) Izmenite prethodno vežbanje tako da vrednosti podataka u klasi **SamoPodaci** budu dodeljene i ispisane iz metode **main()**.

Vežba 6: (2) Napišite program koji obuhvata i poziva metodu **skladiste()**, definisanu kao deo koda u ovom poglavlju.

Vežba 7: (1) Pretvorite **MozeSeUvecati** delove koda u program koji radi.

Vežba 8: (3) Napišite program koji pokazuje da postoji samo jedan primerak određenog **statičnog** polja u klasi, bez obzira na to koliko objekata te klase napravite.

Vežba 9: (2) Napišite program koji pokazuje da automatsko pakovanje funkcioniše za sve proste tipove i njihove omotače.

Vežba 10: (2) Napišite program koji ispisuje tri argumenta preuzeta s komandne linije. Da biste to uradili, pozabavite se indeksima u nizu objekta klase **String** s komadne linije.

Vežba 11: (1) Pretvorite primer **SveBojeDuge** u program koji možete da prevedete i izvršite.

Vežba 12: (2) Pronađite kod za drugu verziju programa **Zdravo.java**, koji je jednostavan primer dokumentacionih komentara. Propustite tu datoteku kroz **Javadoc** i pogledajte rezultate pomoću čitača Weba.

Vežba 13: (1) Propustite kroz **Javadoc** datoteke Dokumentacija1.java, Dokumentacija2.java i Dokumentacija3.java, i proverite rezultate pomoću čitača Weba.

Vežba 14: (1) Dodajte HTML listu dokumentaciji u prethodnom vežbanju.

Vežba 15: (1) Uzmite program iz vežbe 2 i dodajte mu dokumentacione komentare. Izdvojite te dokumentacione komentare u HTML datoteku koristeći **Javadoc** i pregledajte ih pomoću čitača Weba.

Vežba 16: (1) U poglavlju *Inicijalizacija i čišćenje* pronađite primer **Preklapanje.java** i dodajte mu dokumentacione komentare. Izdvojite te dokumentacione komentare u HTML datoteku koristeći **Javadoc** i pregledajte ih pomoću čitača Weba.

Operatori

Na najnižem nivou, u Javi se s podacima radi pomoću operatora.

KAKO JE JAVA IZVEDENA IZ C++-A, VEĆINA OVIH OPERATORA BIĆE POZNATA C I C++ programerima. U Javi su dodata neka poboljšanja i pojednostavljenja.

Ukoliko poznajete sintaksu C-a ili C++-a, možete preleteti kroz ovo i sledeće poglavlje i zadržati se samo na mestima gde se Java razlikuje od tih jezika. Međutim, ako malo zapnete u ovom poglavlju, prodite kroz multimedijalni seminar na CD-u: *Thinking in C* koji možete besplatno preuzeti s Web lokacije *www.mindview.net*. On sadrži audio lekcije, slajdove, vežbe i rešenja iz kojih se mogu steći osnove za učenje Jave.

Jednostavnije naredbe za ispisivanje

U prethodnom poglavlju, upoznali ste Javinu naredbu za ispisivanje:

```
System.out.println("Prilično mnogo kucanja");
```

Primetićete da tu ima ne samo prilično mnogo kucanja (i suvišnog napora za tetive), nego je i prilično nerazumljivo kada se pročita. Većina jezika starijih i novijih od Jave ima mnogo jednostavniji pristup tako često upotrebljavanoj naredbi.

U poglavlju *Kontrola pristupa*, govori se o pojmu *static import* iz Jave SE5; uz to je navedena i mala biblioteka koja pojednostavljuje pisanje naredbe za ispisivanje na ekranu/papiru. Međutim, ne morate znati te pojedinosti da biste mogli da je upotrebljavate. Koristeći je, preradićemo program iz prethodnog poglavlja:

```
//: operatori/ZdravoDatum.java
import java.util.*;
import static net.mindview.util.Print.*;

public class ZdravoDatum {
  public static void main(String[] args) {
    print("Zdravo, danas je: ");
    print(new Date());
  }
} /* Ispis: (55% match)
Zdravo, danas je:
Wed Oct 05 14:39:05 MDT 2005
*///:~
```

Rezultati su mnogo jasniji. Obratite pažnju na umetnutu rezervisanu reč **static** u drugoj naredbi **import**.

Da biste tu biblioteku mogli da koristite, morate preuzeti paket koda ove knjige s lokacije *www.MindView.net* ili s nekog od njenih preslikanih servera. Raširite stablo koda i dodajte njegov korenski direktorijum sistemskoj promenljivoj CLASSPATH svog računara. Jednom ćemo stići i do potpunog opisa putanje klasa (engl. *classpath*), ali neće škoditi

ako odmah počnete da se privikavate na borbu s njom. Nažalost, s Javom ćete morati počešće da se borite.

Iako upotreba biblioteke **net.mindview.util.Print** lepo pojednostavljuje većinu programa, ne može se opravdati njeno korišćenje baš posvuda. Ukoliko u programu ima malo naredaba za ispisivanje, preskačem uvoz (engl. *import*) te biblioteke i pišem dugački **System.out.println()**.

Vežba 1: (1) Napišite program u kojem upotrebljavate i skraćeni i normalni oblik naredaba za ispisivanje.

Korišćenje operatora u Javi

Operator kombinuje jedan ili više argumenata i daje novu vrednost. Argumenti se zadaju u obliku drugačijem od uobičajenih poziva metoda, ali efekat ostaje isti. Verovatno vam je dobro poznat opšti koncept operatora. Sabiranje i unarni plus (+), oduzimanje i unarni minus (-), množenje (*), deljenje (/) i dodela vrednosti (=) manje-više su isti u svakom programskom jeziku.

Svi operatori daju neku vrednost u zavisnosti od svojih operanada. Pored toga, operator može da promeni i vrednost operanda. To se naziva *sporedan efekat* (engl. *side effect*). Operatori koji menjaju svoje operande najčešće se upotrebljavaju baš radi generisanja tog sporednog efekta, ali treba imati na umu da i ti operatori proizvode vrednost, kao i operatori bez sporednog efekta.

Skoro svi operatori rade samo s prostim tipovima. Izuzeci su =, == i !=, koji rade sa svim objektima (i izazivaju nedoumice u radu sa objektima). Pored toga, klasa **String** podržava i operatore + i +=.

Prioriteti

Prioriteti operatora definišu kako će izraz biti izračunat kada se u njemu javlja više operatora. Java ima specifična pravila koja određuju poredak izračunavanja. Najlakše se pamti pravilo da se množenje i deljenje vrše pre sabiranja i oduzimanja. Programeri često zaborave ostala pravila o prioritetima, pa stoga nije na odmet da koristite zagrade za izričito navođenje poretka izračunavanja. Na primer, pogledajte naredbe (1) i (2):

```
//: operatori/Prioriteti.java

public class Prioriteti {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;          // (1)
    int b = x + (y - 2)/(2 + z);      // (2)
    System.out.println("a = " + a + " b = " + b);
  }
} /* Ispis:
a = 5 b = 1
*///:~
```

Te naredbe izgledaju gotovo jednako, ali iz izlaza vidite da zbog upotrebe zagrada u (2) imaju sasvim različita značenja.

Obratite pažnju na upotrebu operatora + u naredbi **System.out.println()**. U tom kontekstu, + znači „nadovezivanje znakovnih nizova" i, ako treba, „konverzija znakovnih nizova". Kada prevodilac naiđe na **String** + ne-**String**, pokušaće da konvertuje ne-**String** u **String**. Kao što vidite iz izlaza, **a** i **b** je uspešno konvertovao iz tipa **int** u **String**.

Dodela vrednosti

Vrednost se dodeljuje operatorom =. On znači „uzmi vrednost s desne strane koja se često naziva *dvrednost* (engl. *rvalue*), i kopiraj je na levu stranu, koja se često naziva *lvrednost* (engl. *lvalue*)". *Dvrednost* je bilo koja konstanta, promenljiva ili izraz koji može da se izračuna, ali *lvrednost* mora da bude posebna imenovana promenljiva. (Odnosno, mora postojati fizički prostor za smeštanje te vrednosti.) Na primer, možete dodeliti konstantnu vrednost promenljivoj

```
a = 4;
```

ali konstantnoj vrednosti ne možete ništa da dodelite – ona ne može biti *lvrednost*. (Ne možete napisati **4 = a;**.)

Dodela vrednosti prostog tipa je prilično očigledna. Pošto prosti tipovi čuvaju stvarnu vrednost, a ne referencu na objekat, kada dodelite vrednost promenljivoj prostog tipa, vi kopirate sadržaj s jednog mesta na drugo. Na primer, ako napišete **a = b** za proste tipove, onda se sadržaj promenljive **b** kopira u sadržaj promenljive **a**. Ako nakon toga nastavite da menjate **a**, ta izmena, naravno, neće uticati na **b**. Kao programer, u većini situacija ovo i očekujete.

Međutim, kada dodeljujete vrednosti objektima, stvari se menjaju. Dok radite sa objektom, vi radite s referencom, a kada dodeljujete „jedan objekat drugom", zapravo kopirate referencu s jednog mesta na drugo. To znači da će, kada napišete **c = d** za objekte, i **c** i **d** pokazivati na objekat na koji je prvobitno pokazivao **d**. Evo primera koji će pokazati to ponašanje:

```
//: operatori/Dodela.java
// Dodela vrednosti objektima može ponekad da prevari.
import static net.mindview.util.Print.*;

class Rezervoar {
  int nivo;
}

public class Dodela {
  public static void main(String[] args) {
    Rezervoar r1 = new Rezervoar();
    Rezervoar r2 = new Rezervoar();
    r1.nivo = 9;
    r2.nivo = 47;
```

```
    print("1: r1.nivo: " + r1.nivo +
          ", r2.nivo: " + r2.nivo);
    r1 = r2;
    print("2: r1.nivo: " + r1.nivo +
          ", r2.nivo: " + r2.nivo);
    r1.nivo = 27;
    print("3: r1.nivo: " + r1.nivo +
          ", r2.nivo: " + r2.nivo);
  }
} /* Ispis:
1: r1.nivo: 9, r2.nivo: 47
2: r1.nivo: 47, r2.nivo: 47
3: r1.nivo: 27, r2.nivo: 27
* ///:-
```

Klasa **Rezervoar** je jednostavna i njena dva primerka (**r1** i **r2**) napravljena su u metodi **main()**. Polju **nivo** u svakom primerku klase **Rezervoar** dodeljene su različite vrednosti, zatim je vrednost **r2** dodeljena primerku **r1** i potom je **r1** promenjen. U mnogim programskim jezicima očekivali biste da su **r1** i **r2** nezavisni sve vreme, ali, pošto ste dodeljivali reference, kada se izmeni objekat **r1**, menja se i objekat **r2**, zato što i **r1** i **r2** sadrže istu referencu koja pokazuje na isti objekat. (Prvobitna referenca koja se nalazila u **r1** i koja je pokazivala na objekat koji je čuvao vrednost 9, izmenjena je prilikom dodele i efektivno izgubljena; njen objekat će počistiti sakupljač smeća).

Ovaj fenomen često se naziva *pojava pseudonima* (engl. *aliasing*), i on je u osnovi način na koji Java radi sa objektima. Ali šta ako nećete da se u ovom slučaju pojavi pseudonim? Možete da precizirate dodelu i da napišete:

```
r1.nivo = r2.nivo;
```

Ovako će se zadržati dva odvojena objekta, umesto da jedan bude odbačen i da se obe reference povežu sa istim objektom. Uskoro ćete shvatiti da je rad s poljima unutar objekata prljav posao i da je u suprotnosti s principima dobrog objektno orijentisanog projektovanja. Pitanje pseudonima nije jednostavno, pa imajte na umu da dodela kod objekata može da dovede do iznenađenja.

Vežba 2: (1) Napišite klasu koja sadrži broj tipa **float** i pomoću nje pokažite pojavu pseudonima.

Pojava pseudonima pri pozivanju metode

Pseudonim će se takođe pojaviti kada objekte prosleđujete metodama:

```
//: operatori/ProslediObjekat.java
// Prosleđivanje objekata metodama može vas iznenaditi.
import static net.mindview.util.Print.*;

class Slovo {
  char c;
}
```

```
public class ProsledjiObjekat {
  static void f(Slovo y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Slovo x = new Slovo();
    x.c = 'a';
    print("1: x.c: " + x.c);
    f(x);
    print("2: x.c: " + x.c);
  }
} /* Ispis:
1: x.c: a
2: x.c: z
* ///:~
```

U mnogim programskim jezicima, metoda **f()** bi napravila kopiju svog argumenta **Slovo y** unutar oblasti važenja metode. Ali, pošto je prosleđena referenca, red

```
y.c = 'z';
```

u stvari menja objekat izvan **f()**.

Pitanje pseudonima i njegovo rešavanje je složeno; razmotreno je u jednom od mrežnih dodataka ove knjige. Zasad samo dobro pazite da izbegnete zamke.

Vežba 3: (1) Napišite klasu koja sadrži broj tipa **float** i pomoću nje pokažite pojavu pseudonima prilikom pozivanja metoda.

Matematički operatori

Osnovni matematički operatori su isti kao i u većini programskih jezika: sabiranje (+), oduzimanje (-), deljenje (/), množenje (*) i modulo (%), koji daje ostatak pri celobrojnom deljenju. Pri celobrojnom deljenju odseca se realni deo, a ne zaokružuje rezultat.

Java takođe koristi skraćeni C/C++ zapis za istovremeno obavljanje operacije i dodele. To se naznačava na sledeći način: iza operatora sledi znak jednakosti, i dosledno se može primeniti na sve operatore u jeziku (kada to ima smisla). Na primer: da biste dodali 4 promenljivoj **x** i rezultat dodelili **x**, pišite: **x += 4**.

Ovaj primer pokazuje korišćenje matematičkih operatora:

```
//: operatori/MatOperatori.java
// Prikaz matematičkih operatora
import java.util.*;
import static net.mindview.util.Print.*;

public class MatOperatori {
  public static void main(String[] args) {
    // Napravimo inicijalizovan generator slučajnih brojeva
    Random slucajan = new Random(47);
    int i, j, k;
```

```
    // Izaberimo broj od 1 do 100:
    j = slucajan.nextInt(100) + 1;
    print("j : " + j);
    k = slucajan.nextInt(100) + 1;
    print("k : " + k);
    i = j + k;
    print("j + k : " + i);
    i = j - k;
    print("j - k : " + i);
    i = k / j;
    print("k / j : " + i);
    i = k * j;
    print("k * j : " + i);
    i = k % j;
    print("k % j : " + i);
    j %= k;
    print("j %= k : " + j);
    // Test za brojeve s pokretnim zarezom
    float u,v,w;  // Primenljivo i na double
    v = slucajan.nextFloat();
    print("v : " + v);
    w = slucajan.nextFloat();
    print("w : " + w);
    u = v + w;
    print("v + w : " + u);
    u = v - w;
    print("v - w : " + u);
    u = v * w;
    print("v * w : " + u);
    u = v / w;
    print("v / w : " + u);
    // Sledeće radi i za char,
    // byte, short, int, long i double:
    u += v;
    print("u += v : " + u);
    u -= v;
    print("u -= v : " + u);
    u *= v;
    print("u *= v : " + u);
    u /= v;
    print("u /= v : " + u);
  }
} /* Ispis:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
```

```
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

Za generisanje brojeva, program prvo pravi objekat klase **Random**. Ukoliko pri njegovom pravljenju ne prosledite nijedan argument, Java uzima tekuće vreme kao vrednost za inicijalizaciju generatora pseudoslučajnih brojeva, pa prilikom svakog izvršavanja programa daje drugačiji izlaz. Međutim, u ovoj knjizi važno je da izlaz prikazan na kraju primera bude što ujednačeniji, da bi se mogao proveriti pomoću spoljnih sredstava. Ukoliko prilikom pravljenja **Random** objekta zadate isto *seme* (broj za inicijalizaciju generatora pseudoslučajnih brojeva), isti će pseudoslučajni brojevi biti generisani svaki put, pa se izlaz programa može proveriti[1]. U slučaju da želite drugačiji izlaz svaki put kada pokrenete program, slobodno izbacite seme.

Program generiše razne tipove slučajnih brojeva pomoću objekta klase **Random**, pozivajući metode: **nextInt**(), **nextLong**(), **nextFloat**() ili **nextDouble**(). Argument metode **nextInt**() zadaje gornju granicu generisanih brojeva. Donja granica je nula, a pošto je nećemo zbog nemogućnosti deljenja s nulom, rezultatu smo dodali 1.

Vežba 4: (2) Napišite program koji izračunava brzinu iz konstantnog puta i konstantnog vremena.

Unarni operatori minus i plus

Unarni minus (-) i unarni plus (+) isti su operatori kao i binarni minus i plus. Prevodilac prepoznaje šta ste želeli da upotrebite na osnovu načina kako pišete izraz. Na primer, značenje izraza

```
x = -a;
```

očigledno je. Prevodilac može da raspozna:

```
x = a * -b;
```

ali bi čitalac mogao da se zbuni, stoga je katkada jasnije ako se napiše:

```
x = a * (-b);
```

Unarni minus menja znak vrednosti. Unarni plus je suprotan unarnom minusu, iako nema nikakav efekat sem što tipove **byte**, **short** i **char** pretvara u **int**.

[1] Broj 47 se smatrao „čarobnim brojem" na koledžu koji sam pohađao, pa mi se to urezalo u sećanje.

Automatsko uvećanje i umanjenje

Java ima puno skraćenica, kao i C. Skraćenice doprinose lakšem pisanju koda i lakšem, ili težem, čitanju koda.

Dve zgodne skraćenice su operatori uvećanja i umanjenja (na engleskom se često zovu *auto-increment* i *auto-decrement*). Operator umanjenja je -- i označava „smanji za jedinicu". Operator uvećanja je ++ i označava „povećaj za jedinicu". Ako je **a** tipa **int**, na primer, izraz ++**a** ekvivalentan je izrazu (**a** = **a** + **1**). Ne samo da operatori za uvećanje i umanjenje menjaju vrednost promenljive, nego je i vraćaju kao rezultat.

Postoje dve verzije oba operatora, koje se često nazivaju *prefiksne* i *sufiksne* verzije. *Prefiksno uvećanje* je oblik kada se operator ++ pojavljuje pre promenljive, a *sufiksno uvećanje* kada se operator ++ pojavljuje nakon promenljive. Slično tome, *prefiksno umanjenje* je kada se operator -- pojavljuje pre promenljive, a *sufiksno umanjenje* kada se operator -- pojavljuje posle promenljive. Pri prefiksnom uvećanju i prefiksnom umanjenju (tj. ++**a** ili --**a**), najpre se izvodi operacija, a potom se vraća nova vrednost. Pri sufiksnom uvećanju i sufiksnom umanjenju (tj. **a**++ ili **a**--), prvo se vraća stara vrednost, a zatim se izvršava operacija. Kao primer:

```
//: operatori/AutomatskiOperatori.java
// Prikazuje operatore ++ i --.
import static net.mindview.util.Print.*;

public class AutomatskiOperatori {
  public static void main(String[] args) {
    int i = 1;
    print("i : " + i);
    print("++i : " + ++i); // Prefiksno uvećanje
    print("i++ : " + i++); // Sufiksno uvećanje
    print("i : " + i);
    print("--i : " + --i); // Prefiksno umanjenje
    print("i-- : " + i--); // Sufiksno umanjenje
    print("i : " + i);
  }
} /* Ispis:
i : 1
++i : 2
i++: 2
i : 3
--i : 2
i-- : 2
i : 1
* ///:~
```

Vidite da se vrednost za prefiksni oblik vraća nakon izvršavanja operacije, ali za sufiksni oblik dobijate vrednost promenljive pre nego što se operacija izvrši. Ovo su jedini operatori (osim onih koji upliću i dodelu) koji imaju sporedne efekte (odnosno, koji ne koriste samo vrednost operanada, već ih i menjaju).

Jedno od objašnjenja za poreklo imena C++, koje znači „jedan korak dalje od C-a", leži u operatoru uvećanja. U vreme nastanka Jave, Bill Joy (jedan od autora Jave) rekao je da je „Java = C++--" (C plus plus minus minus), nagoveštavajući kako je Java jezik C++ iz kojeg su izbačeni nepotrebni teški delovi, pa je zbog toga mnogo jednostavnija. Što budete više napredovali kroz ovu knjigu, videćete da su mnogi delovi jednostavniji, ali ima drugih stvari zbog kojih Java ipak nije mnogo jednostavnija od C++-a.

Operatori poređenja

Rezultat operatora poređenja je logička vrednost (**boolean**). Oni ocenjuju relaciju između vrednosti dva operanda. Izraz sa operatorima poređenja daje vrednost **true** ako je relacija tačna i **false** ako je relacija netačna. Operatori poređenja su manje od (<), veće od (>), manje ili jednako (<=), veće ili jednako (>=), jednako (==) i različito (!=). Jednakost i različitost se mogu primeniti na sve proste tipove podataka, ali se ostala poređenja ne mogu primeniti na tip **boolean**. Objekti tipa **boolean** mogu imati samo vrednosti **true** ili **false**, pa ne bi imalo smisla tražiti koji od njih je veći, a koji manji.

Ispitivanje jednakosti objekata

Operatori poređenja == i != mogu se primeniti i na sve objekte, ali njihovo značenje često zbunjuje početnike u Javi. Evo primera:

```
//: operatori/Jednakost.java

public class Jednakost {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
} /* Ispis:
false
true
*///:~
```

Naredba **System.out.println(n1 == n2)** ispisuje rezultat logičkog poređenja koje se unutar nje nalazi. Njegov rezultat bi sigurno trebalo da bude true, a onog sledećeg false, pošto su vrednosti oba objekta klase **Integer** iste. Ali iako je *sadržina* objekata ista, reference su različite, a operatori == i != porede reference objekata, a ne njihovu sadržinu. Stoga je rezultat prvog poređenja false, a drugog true. Naravno, ovo isprva iznenađuje.

Šta ako želite da poredite ekvivalenciju stvarnih sadržaja objekata? U tom slučaju morate da koristite posebnu metodu **equals()** koja postoji u svim objektima (ta metoda se ne koristi za proste tipove, za koje ionako dobro funkcionišu == i !=). Evo kako se ona koristi:

```
//: operatori/ MetodaEquals.java

public class MetodaEquals {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} /* Ispis:
true
*///:~
```

Rezultat će sada biti kao što i očekujete. Ah, ali to ipak nije baš tako jednostavno. Ako sami napravite klasu, na primer:

```
//: operatori/ MetodaEquals2.java
// equals() Podrazumevano ne poredi sadržinu.

class Vrednost {
  int i;
}
public class MetodaEquals2 {
  public static void main(String[] args) {
    Vrednost v1 = new Vrednost();
    Vrednost v2 = new Vrednost();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} /* Ispis:
false
*///:~
```

ponovo ćete biti zbunjeni: rezultat je **false**, zato što je podrazumevano ponašanje metode **equals**() da poredi reference. Stoga nećete dobiti željeno ponašanje osim ako ne *redefinišete* (engl. *override*) metodu **equals**() u vašoj novoj klasi. Nažalost, o redefinisanju nećete učiti sve do poglavlja *Ponovno korišćenje klasa*, a o pravilnom načinu definisanja metode **equals**() do poglavlja *Detaljno razmatranje kontejnera*, ali u međuvremenu samo pazite na način na koji se metoda **equals**() ponaša, jer će vas to možda sačuvati od problema.

Većina klasa iz Javine biblioteke realizuje metodu **equals**() tako da ona poredi sadržaj objekata umesto njihovih referenci.

Vežba 5: (2) Napravite klasu **Pas** koja sadrži dve promenljive tipa **String**: **ime** i **kaže**. U metodi **main**() napravite dva objekta tipa **Pas** sa imenima Fleki (koji kaže: „Raf!“) i Buva (koji kaže: „Vaf!“). Zatim prikažite njihova imena i šta kažu.

Vežba 6: (3) Posle vežbe 5, napišite novu referencu objekta tipa **Pas** i dodelite je Flekijevom objektu. Zatim obavite poređenje koristeći == i **equals**() za sve reference.

Logički operatori

Logički operatori konjunkcija (&&), disjunkcija (||) i negacija (!) daju vrednosti **true** ili **false** tipa **boolean**. U ovom primeru koriste se operatori poređenja i logički operatori:

```
//: operatori/Logicki.java
// Relacioni operatori (poređenja) i logički operatori.
import java.util.*;
import static net.mindview.util.Print.*;

public class Logicki {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    int i = slucajan.nextInt(100);
    int j = slucajan.nextInt(100);
    print("i = " + i);
    print("j = " + j);
    print("i > j je " + (i > j));
    print("i < j je " + (i < j));
    print("i >= j je " + (i >= j));
    print("i <= j je " + (i <= j));
    print("i == j je " + (i == j));
    print("i != j je " + (i != j));

    // Tip int u Javi ne može da se
    // koristi kao logički tip
    //! print("i && j je " + (i && j));
    //! print("i || j je " + (i || j));
    //! print("!i je " + !i);
        print("(i < 10) && (j < 10) je "
           + ((i < 10) && (j < 10)) );
        print("(i < 10) || (j < 10) je "
           + ((i < 10) || (j < 10)) );
  }
} /* Ispis:
i = 58
j = 55
i > j je true
i < j je false
i >= j je true
i <= j je false
i == j je false
i != j je true
(i < 10) && (j < 10) je false
(i < 10) || (j < 10) je false
*///:~
```

Operatore konjunkcije, disjunkcije i negacije možete primeniti samo na vrednosti tipa **boolean**. Za logičke uslove ne možete da koristite ostale (nelogičke) tipove, kao u jezicima C i C++. Ovakve neuspele pokušaje možete videti u komentarima sa oznakom //!

(takvo označavanje komentara omogućava njihovo automatsko uklanjanje, što olakšava testiranje). Naredni izrazi daju vrednosti tipa **boolean** tako što koriste poređenje, a zatim na te rezultate primene logičke operacije.

Obratite pažnju na to da se vrednost tipa **boolean** automatski konvertuje u odgovarajući tekstualni oblik ako se koristi tamo gde se očekuje objekat klase **String**.

Definiciju za **int** u prethodnom programu možete zameniti bilo kojim prostim tipom podataka osim **boolean**. Pazite na to da je poređenje brojeva u formatu pokretnog zareza veoma precizno. Broj koji se i po poslednjoj decimali razlikuje od drugog broja i dalje je „različit". Broj koji je na najmanjoj decimali iznad nule i dalje je „različit od nule".

Vežba 7: (3) Napišite program koji simulira bacanje novčića („pismo – glava").

Nepotpuno izračunavanje

Kada radite s logičkim operatorima, naići ćete na fenomen pod nazivom „nepotpuno izračunavanje". To znači da će izraz biti izračunavan *samo* do trenutka kada tačnost ili netačnost celog izraza može nedvosmisleno da se odredi. Zato preostali delovi logičkog izraza uopšte neće biti izračunati. Evo primera koji pokazuje nepotpuno izračunavanje:

```
//: operatori/NepotpunoIzracunavanje.java
// Prikazuje pojavu nepotpunog izračunavanja pri radu s logičkim operatorima.
import static net.mindview.util.Print.*;

public class NepotpunoIzracunavanje {
  static boolean test1(int vrednost) {
    print("test1(" + vrednost + ")");
    print("rezultat: " + (vrednost < 1));
    return vrednost < 1;
  }
  static boolean test2(int vrednost) {
    print("test2(" + vrednost + ")");
    print("rezultat: " + (vrednost < 2));
    return vrednost < 2;
  }
  static boolean test3(int vrednost) {
    print("test3(" + vrednost + ")");
    print("rezultat: " + (vrednost < 3));
    return vrednost < 3;
  }
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
      print("izraz ima logičku vrednost" + b);
  }
} /* Ispis:
test1(0)
rezultat: true
test2(2)
rezultat: false
izraz ima logičku vrednost false
*///:~
```

Svaki test vrši poređenje sa argumentom i vraća vrednosti **true** ili **false.** Takođe ispisuje i poruku da je bio pozvan. Testovi se pozivaju preko sledećeg izraza:

```
test1(0) && test2(2) && test3(2)
```

Prirodno je da pomislite kako će sva tri testa biti izvršena, ali izlaz pokazuje drugačije: Prvi test daje rezultat **true** pa se izračunavanje izraza nastavlja. Međutim, drugi test daje rezultat **false**. Pošto to znači da će i rezultat celog izraza sigurno biti **false**, zašto nastavljati izračunavanje izraza? To može da potraje. Baš zbog toga se koristi nepotpuno izračunavanje: ako ne treba do kraja izračunavati sve delove logičkog izraza, program će možda raditi brže.

Literali

Kada u programu doslovno navodite vrednosti, prevodilac obično tačno zna kojim tipom da ih predstavi. Ponekad se može javiti nedoumica. U tom slučaju morate da navodite prevodioca dodatnim informacijama u obliku znakova pridruženih vrednosti literala. Sledeći primer pokazuje te znakove:

```
//: operatori/Vrednosti.java
import static net.mindview.util.Print.*;

public class Vrednosti {
  public static void main(String[] args) {
    int i1 = 0x2f; // Heksadecimalno zadavanje celog broja
                   // (malim slovima)
    print("i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Heksadecimalno zadavanje celog broja
                   // (velikim slovima)
    print("i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Oktalno zadavanje celog broja (vodeća nula)
    print("i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // najveća vrednost za tip char, heksadecimalno
    print("c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // najveća vrednost za tip byte, heksadecimalno
    print("b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // najveća vrednost za tip short, heksadecimalno
    print("s: " + Integer.toBinaryString(s));
    long n1 = 200L; // sufiks za long
    long n2 = 200l; // sufiks za long (može da zbuni jer liči na jedinicu)
    long n3 = 200;
    float f1 = 1;
    float f2 = 1F; // sufiks za float
    float f3 = 1f; // sufiks za float
    double d1 = 1d; // sufiks za double
    double d2 = 1D; // sufiks za double
    // (Heksadecimalno i oktalno mogu se zadavati i brojevi tipa long)
} /* Ispis:
i1: 101111
i2: 101111
i3: 1111111
```

```
c: 1111111111111111
b: 1111111
s: 111111111111111
*///:~
```

Slovo iza literala zadaje njegov tip. Veliko ili malo **L** označava tip **long** (međutim, malo **l** zbunjuje zato što liči na jedinicu). Veliko ili malo **F** označava tip **float**. Veliko ili malo **D** označava tip **double**.

Sve celobrojne tipove podataka moguće je zadavati u heksadecimalnom obliku (sa osnovom 16), naznačavanjem vodećim **0x** ili **0X** nakon čega slede simboli 0–9 i a–f, bilo malim ili velikim slovima. Ako pokušate da inicijalizujete promenljivu vrednošću većom od one koju može da čuva (bez obzira na numerički oblik vrednosti), prevodilac će prijaviti grešku. U gornjem primeru obratite pažnju na maksimalne moguće heksadecimalne vrednosti za tipove **char**, **byte** i **short**. Ako ih prekoračite, prevodilac će ih automatski pretvoriti u **int** i javiti kako treba izvršiti sužavanje *eksplicitnom konverzijom* za datu dodelu. (Eksplicitne konverzije su definisane u nastavku poglavlja.) Tada ćete znati da ste prekoračili dozvoljenu vrednost.

Oktalni oblik (sa osnovom 8) naznačava se vodećom nulom u zapisu broja, koju slede cifre 0–7.

U C-u, C++-u i Javi ne postoji prikaz binarnih brojeva kao literala. Međutim, pri radu s heksadecimalnim i oktalnim zapisom rezultate je podesno prikazati u binarnom obliku. To se lako postiže metodama **static toBinaryString**() iz klasa **Integer** i **Long**. Imajte u vidu da se manji tipovi automatski konvertuju u **int** kada se proslede u **Integer.toBinaryString**().

Vežba 8: (2) Pokažite da heksadecimalni i oktalni zapis rade s brojevima tipa **long**. Za prikazivanje rezultata upotrebite **Long.toBinaryString**().

Eksponencijalni zapis

Za eksponencijalni zapis realnog broja koristi se notacija koju sam oduvek smatrao obeshrabrujućom.

```
//: operatori/Eksponenti.java
// "e" znači "10 na stepen".

public class Eksponenti {
  public static void main(String[] args) {
    // Veliko i malo 'e' su jednaki:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' je opciono
    double expDouble2 = 47e47; // Automatska konverzija u double
    System.out.println(expDouble);
  }
} /* Ispis:
1.39E-43
4.7E48
*///:~
```

U nauci i inženjerstvu 'e' označava osnovu prirodnog logaritma, koja otprilike iznosi 2,718. (Preciznija vrednost tipa **double** je dostupna u Javi kao **Math.E**.) Ona se koristi u eksponencijalnim izrazima kao što su $1,39 \times e^{-43}$, što znači $1,39 \times 2,718^{-43}$. Tvorci FORTRAN-a odlučili su da će **e** značiti „deset na stepen", što je čudno, jer se FORTRAN upotrebljava u nauci i inženjerstvu, i svako je mogao pomisliti da će tvorci obratiti pažnju na takvu dvosmislenost.[2] Bilo kako bilo, ovaj običaj je nastavljen u C-u, C++-u i sada u Javi. Stoga, ako ste se navikli na **e** kao na osnovu prirodnog logaritma, obratite pažnju kada u Javi vidite izraz kao što je **1,39 e-43f**; on označava $1,39 \times 10^{-43}$.

Prateći znak ne morate da koristite kada prevodilac može sam da odredi odgovarajući tip. Kada napišete:

```
long n3 = 200;
```

nema dvosmislenosti, pa je **L** iza broja 200 izlišno. Međutim, kada napišete:

```
float f4 = 1e-43f;   // 10 na -43. stepen
```

prevodilac obično tumači eksponencijalne brojeve kao brojeve dvostruke preciznosti (**double**), pa bi vam bez pratećeg slova **f** prijavio grešku, tražeći eksplicitnu konverziju iz **double** u **float**.

Vežba 9: (1) Prikažite najveći i najmanji broj koji se može zapisati pomoću eksponencijalnog zapisa tipova **float** i **double**.

Operatori nad bitovima

Operatori nad bitovima omogućavaju da radite s pojedinim bitovima od kojih se sastoji neki prosti tip podataka. Operatori nad bitovima primenjuju logičku (Bulovu) algebru nad odgovarajućim bitovima dva argumenta da bi izračunali rezultat.

Operatori nad bitovima vode poreklo od niskog nivoa jezika C; kada se direktno radi s hardverom i treba direktno postavljati bitove hardverskih registara. Java je trebalo da bude ugradena u aparate za prikazivanje Interneta na ekranu TV-a, pa je ova orijentacija ka niskom nivou i imala smisla. Međutim, operatore nad bitovima verovatno nećete previše koristiti.

Operator konjunkcije nad bitovima (&) daje jedinicu kao vrednost izlaznog bita ako su oba ulazna bita jednaka jedinici; inače, daje nulu. Operator disjunkcije nad bitovima (|) daje jedinicu kao vrednost izlaznog bita ako je bilo koji od ulaznih bitova jednak jedinici, a

[2] John Kirkham je napisao: „Računarima sam počeo da se bavim 1962. koristeći FORTRAN II na mašini IBM 1620. U to vreme, tokom šezdesetih i početkom sedamdesetih, FORTRAN je u potpunosti pisan velikim slovima. To potiče verovatno od toga što su stari uređaji za unos koristili 5-bitni Baudotov kod koji nije podržavao mala slova. Slovo E je u eksponencijalnoj notaciji uvek pisano kao veliko i nikada nije bilo mešano sa osnovom prirodnog logaritma, e, koja se uvek piše malim slovom. E je označavalo eksponent, obično 10, za korišćeni brojni sistem. U to vreme i oktalni format je bio široko rasprostranjen među programerima. Da sam naišao na oktalni broj u eksponencijalnoj notaciji, smatrao bih da je osnova 8. Sećam se da sam prvi put video eksponencijalni zapis napisan s malim e u kasnim sedamdesetim i smatrao sam ga zbunjujućim. Problem se pojavio kada su mala slova prodrla u FORTRAN, ne na samom početku. Mi smo, u stvari, imali funkcije koje smo koristili ako nam je zaista bila potrebna osnova prirodnog logaritma, ali su sve bile pisane velikim slovima."

daje nulu samo ako su oba ulazna bita jednaka nuli. Isključiva disjunkcija nad bitovima ili XOR (^) daje jedinicu kao vrednost izlaznog bita, ako je jedan ili drugi ulazni bit jednak jedinici, ali ne i oba. Negacija nad bitovima (~, koja se takođe naziva i *operator prvog komplementa*) jeste unarni operator; ima samo jedan argument. (Svi ostali operatori nad bitovima su binarni operatori – imaju dva operanda.) Negacija nad bitovima daje negaciju ulaznog bita – jedinicu ako je ulazni bit jednak nuli i nulu ako je ulazni bit jednak jedinici.

Za operatore nad bitovima i logičke operatore koriste se isti simboli, pa će vam biti lakše da se setite značenja ako se dosetite sledećeg: pošto su bitovi „mali", za operatore nad bitovima koristi se samo po jedan znak.

Operatori nad bitovima mogu da se kombinuju sa znakom = i time ujedine operaciju sa dodeljivanjem: dozvoljeni su &=, |= i ^=. (Pošto je ~ unarni operator, on ne može da se kombinuje sa znakom =.)

Tip **boolean** se tretira kao jednobitna vrednost, pa je situacija nešto drugačija. Nad njim možete da primenite konjunkciju, disjunkciju i isključivu disjunkciju nad bitovima, ali ne i negaciju nad bitovima (verovatno da bi se izbeglo mešanje s logičkom negacijom). Za tip **boolean** operatori nad bitovima imaju isti efekat kao i logički operatori, osim što se ne javlja nepotpuno izračunavanje. Operator isključive disjunkcije nad bitovima nema ekvivalentan logički operator. Zato ovaj operator predstavlja jedini način da na dve vrednosti tipa **boolean** primenite operaciju XOR. Na promenljive tipa **boolean** ne možete primenjivati operatore pomeranja koje ćemo upravo opisati.

Vežba 10: (3) Napišite program s dve binarne konstante koje imaju naizmenične jedinice i nule, s tim što je prvoj na najmanje značajnom mestu nula, a drugoj jedinica. (Uputstvo: to će vam biti najlakše s heksadecimalnim konstantama). Zadajte ta dva broja kao argumente svih operatora nad bitovima na sve moguće načine, a rezultate prikažite metodom **Integer.toBinaryString()**.

Operatori pomeranja

Operatori pomeranja (engl. *shift*) takođe rade s bitovima. Oni mogu da se koriste isključivo s prostim, celobrojnim tipovima. Operator pomeranja ulevo (<<) kao rezultat daje operand s leve strane operatora, pomeren ulevo za broj bitova naveden nakon operatora (nizi bitovi popunjavaju se nulama). Operator označenog pomeranja udesno (>>) kao rezultat daje operand s leve strane operatora, pomeren udesno za broj bitova naveden nakon operatora. Označeno pomeranje udesno >> koristi *produžavanje uz očuvanje znaka* (engl. *sign extension*): ako je vrednost pozitivna, viši bitovi se popunjavaju nulom; ako je vrednost negativna, viši bitovi se popunjavaju jedinicom. U Javu je takođe dodato i neoznačeno pomeranje udesno >>> koje koristi *produžavanje uz dodavanje nula* (engl. *zero extension*): bez obzira na znak, viši bitovi se popunjavaju nulom. Ovaj operator ne postoji u C-u ili C++-u.

Ako pomerate vrednosti tipa **char**, **byte** ili **short**, one će biti proširene na **int** pre nego što se izvrši pomeranje i rezultat će biti tipa **int**. Koristi se samo pet nižih bitova vrednosti s desne strane operatora. Ovo vas sprečava da izvršite pomeranje za više mesta nego što ima bitova unutar promenljive tipa **int**. Ako operišete nad vrednošću tipa **long**, dobićete

rezultat tipa **long**. Biće korišćeno samo šest nižih bitova vrednosti s desne strane operatora, pa ne možete da izvršite pomeranje za više mesta nego što ima bitova unutar promenljive tipa **long**.

Pomeranje može da se kombinuje sa znakom jednakosti (<<= ili >>= ili >>>=). Lvrednost se zamenjuje vrednošću lvrednost pomerenom za dvrednost mesta. Kada se neoznačeno pomeranje udesno kombinuje sa dodelom, postoji problem. Ako ovu operaciju primenite na vrednosti tipa **byte** ili **short**, nećete dobiti pravilne rezultate. Umesto toga, one se proširuju na **int**, pomeraju udesno, ali se skraćuju prilikom dodele, pa u tim slučajevima kao rezultat dobijate **–1**. To pokazuje naredni primer:

```
//: operatori/NeoznacenoPomeranjeUdesno.java
// Test za neoznačeno pomeranje udesno.
import static net.mindview.util.Print.*;

public class NeoznacenoPomeranjeUdesno {
  public static void main(String[] args) {
    int i = -1;
    print(Integer.toBinaryString(i));
    i >>>= 10;
    print(Integer.toBinaryString(i));
    long l = -1;
    print(Long.toBinaryString(l));
    l >>>= 10;
    print(Long.toBinaryString(l));
    short s = -1;
    print(Integer.toBinaryString(s));
    s >>>= 10;
    print(Integer.toBinaryString(s));
    byte b = -1;
    print(Integer.toBinaryString(b));
    b >>>= 10;
    print(Integer.toBinaryString(b));
    b = -1;
    print(Integer.toBinaryString(b));
    print(Integer.toBinaryString(b>>>10));
  }
} /* Ispis:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*///:~
```

U poslednjem pomeranju, rezultujuća vrednost nije dodeljena promenljivoj **b**, već je direktno ispisana, pa se dobija ispravna vrednost.

Evo primera koji pokazuje korišćenje svih operatora koji se odnose na bitove:

```
//: operatori/RadSaBitovima.java
// Korišćenje operatora nad bitovima.
import java.util.*;
import static net.mindview.util.Print.*;

public class RadSaBitovima {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    int i = slucajan.nextInt();
    int j = slucajan.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
    int maxpos = 2147483647;
    printBinaryInt("najveći", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("najmanji", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = slucajan.nextLong();
    long m = slucajan.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("najveći", ll);
    long lln = -9223372036854775808L;
    printBinaryLong("najmanji", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
```

```
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binarno:\n    " +
        Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binarno:\n    " +
        Long.toBinaryString(l));
  }
} /* Ispis:
-1, int: -1, binarno:
   11111111111111111111111111111111
+1, int: 1, binarno:
1
največi, int: 2147483647, binarno:
   01111111111111111111111111111111
najmanji, int: -2147483648, binarno:
   10000000000000000000000000000000
i, int: -1172028779, binarno:
   10111010001001000100001010010101
~i, int: 1172028778, binarno:
   01000101110110111011110101101010
-i, int: 1172028779, binarno:
   01000101110110111011110101101011
j, int: 1717241110, binarno:
   01100110010110110000010100010110
i & j, int: 570425364, binarno:
   00010000000000000000000010100
i | j, int: -25213033, binarno:
   11111110011111111010001110010111
i ^ j, int: -595638397, binarno:
   11011100011111111010001111000011
i << 5, int: 1149784736, binarno:
   01000100100010000101001010100000
i >> 5, int: -36625900, binarno:
   11111101110100010010001000010100
(~i) >> 5, int: 36625899, binarno:
   00010111011011011101111101011
i >>> 5, int: 97591828, binarno:
   00101110001001001000010100
(~i) >>> 5, int: 36625899, binarno:
   00010111011011011101111101011
...
*///:~
```

Dve metode na kraju, **printBinaryInt()** i **printBinaryLong()**, ispisuju vrednost tipa **int** ili **long** u binarnom obliku zajedno s tekstualnim opisom. Osim što pokazuje dejstvo svih operatora nad bitovima na vrednosti tipa **int** i **long**, ovaj primer pokazuje i minimalnu i maksimalnu vrednost kao i vrednosti +1 i –1 za tipove **int** i **long**, pa možete videti kako one izgledaju. Obratite pažnju na to da najviši bit predstavlja znak: 0 označava pozitivan broj, a 1 negativan. Prikazan je rezultat programa za deo koji se tiče tipa **int**.

Ovakav binarni zapis celih brojeva naziva se *drugi komplement*.

Vežba 11: (3) Krenite od broja s jednom binarnom jedinicom na najznačajnijem mestu. (Uputstvo: upotrebite heksadecimalnu konstantu). Pomerajte tu jedinicu operatorom označenog pomeranja udesno po svim mogućim binarnim položajima i sve ih prikažite metodom **Integer.toBinaryString()**.

Vežba 12: (3) Krenite od broja sa svim binarnim jedinicama. Pomerite ih za jedno mesto ulevo, a zatim ih operatorom neoznačenog pomeranja pomerajte udesno po svim mogućim binarnim položajima i sve ih prikažite metodom **Integer.toBinaryString()**.

Vežba 13: (1) Napišite metodu koja prikazuje **char** vrednosti u binarnom obliku. Rezultate rada metode prikažite na nekoliko različitih znakova.

Ternarni operator uslovljavanja

Ternarni operator *uslovljavanja* neobičan je jer ima tri operanda. Ovo je pravi operator, jer kao rezultat daje vrednost, za razliku od uobičajene naredbe **if-else** koju ćete videti u sledećem odeljku ovog poglavlja. Operator se koristi u obliku:

```
logički-izraz ? vrednost0 : vrednost1
```

Ako je vrednost *logičkog-izraza* **true**, izračunava se *vrednost0* i taj rezultat postaje vrednost operacije. Ako je vrednost *logičkog-izraza* **false**, izračunava se *vrednost1* i njen rezultat postaje vrednost cele operacije.

Naravno, možete da koristite i uobičajenu konstrukciju **if-else** (opisanu kasnije), ali je ternarni operator mnogo sažetiji. Iako je jedna od najistaknutijih osobina jezika C (iz koga ovaj operator potiče) mogućnost pisanja sažetih izraza i iako je ternarni operator delimično uveden zbog efikasnosti, treba biti oprezan pri njegovom svakodnevnom korišćenju jer lako može da proizvede nečitljiv kôd.

Operator uslovljavanja se razlikuje od konstrukcije **if-else** po tome što izračunava vrednost. Evo primera u kojem ćete videti tu razliku:

```
//: operatori/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
   static int ternarni(int i) {
      return i  < 10 ? i * 100 : i * 10;
   }
   static int standardniIfElse (int i) {
     if (i < 10)
```

```
        return i * 100;
      else
        return i * 10;
    }
    public static void main(String[] args)   {
       print(ternarni(9));
       print(ternarni(10));
       print(standardniIfElse(9));
       print(standardniIfElse(10));
    }
} /* Ispis:
900
100
900
100
*///:~
```

Vidite da je kôd metode **ternarni()** sažetiji od onoga što biste morali pisati da ternarnog operatora nema, kao u metodi **standardniIfElse()**. Međutim, **standardniIfElse()** se lakše čita i brže piše. Stoga dobro razmislite pre nego što upotrebite ternarni operator – po pravilu, to je umesno kada promenljivoj dodeljujete jednu od dve vrednosti.

Operatori + i += za znakovne nizove

Postoje operatori koji u Javi imaju specijalnu namenu: operatori + i += mogu se koristiti za nadovezivanje znakovnih nizova, kao što ste već videli. Čini se da je to razumljiv način korišćenja tih operatora, iako se ne uklapa u tradicionalni način njihovog korišćenja.

Ova mogućnost se činila kao dobra ideja projektantima C++-a, pa su u jezik dodali *preklapanje operatora* (engl. *operator overloading*) koje je omogućilo dodavanje značenja skoro svakom operatoru. Nažalost, preklapanje operatora je, u kombinaciji s nekim drugim ograničenjima C++-a, ispalo prilično komplikovano za programere koji su tu mogućnost hteli da ugrade u svoje klase. Iako bi se preklapanje operatora mnogo jednostavnije realizovalo na Javi nego na C++-u (što je pokazano u jeziku C# koji *ima* jednostavno preklapanje operatora), ova mogućnost je i dalje ocenjena kao previše složena, pa programeri na Javi ne mogu da preklope operatore, kao što mogu programeri na C#-u i C++-u.

Pri korišćenju operatora na znakovnim nizovima, javlja se zanimljiv efekat. Ako je prvi operand izraza tipa **String**, tada svi operandi koji slede takođe moraju biti tog tipa (setite se da će prevodilac automatski pretvoriti niz znakova pod navodnicima u **String**):

```
//: operatori/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperatori {
  public static void main(String[] args) {
int x = 0, y = 1, z = 2;
String s = "x, y, z ";
print(s + x + y + z);
    print(x + " " + s); // Pretvara x u String
```

```
    s += "(sabrano) = "; // Operator nadovezivanja
    print(s + (x + y + z));
    print("" + x); // Skraćeni zapis umesto Integer.toString()
  }
} /* Ispis:
x, y, z 012
0 x, y, z
x, y, z (sabrano) = 3
0
*///:~
```

Obratite pažnju na to da je prva naredba print ispisala **012**, a ne **3**, što bismo dobili da su ti celi brojevi sabrani. Ovo se desilo zato što će Java prevodilac pretvoriti **x**, **y** i **z** u znakovne ekvivalente i nadovezati ih, umesto da ih prvo sabere. Druga naredba print pretvorila je vodeću promenljivu u **String**, dakle ta konverzija ne zavisi od redosleda argumenata. Najzad, vidite kako je operator += upotrebljen da se promenljivoj **s** doda znakovni niz i kako je zagradom određen redosled izračunavanja izraza, da bi celi brojevi u njoj bili sabrani pre prikazivanja.

Imajte u vidu poslednji primer u metodi **main()**: prazan znakovni niz iza kojeg sledi + i neki prost tip. Java će automatski pretvoriti taj tip u **String**, a da nismo morali da pozivamo glomazniju izričitu metodu, u ovom slučaju, **Integer.toString()**.

Česte greške prilikom korišćenja operatora

Načinićete jednu od čestih grešaka ako pri radu sa operatorima izostavite zagrade a niste sasvim sigurni kako će teći izračunavanje izraza. Ovo važi i u Javi.

Veoma česta greška u C-u i C++-u je:

```
while (x = y) {
// ....
}
```

Programer je očigledno želeo da ispita jednakost (==), a ne da vrši dodelu. U C-u i C++-u rezultat ove dodele bi uvek bio **true**, ako je **y** različito od nule, i verovatno biste dobili beskrajnu petlju. Rezultat ovog izraza u Javi nije tipa **boolean**, ali prevodilac očekuje tip **boolean** i neće izvršiti konverziju iz tipa **int**, nego će pri prevođenju prijaviti grešku. Tako će problem biti otkriven pre nego što pokušate da pokrenete program. Stoga se ovaj tip zamke u Javi ne javlja. (Grešku pri prevođenju jedino nećete dobiti kada su **x** i **y** tipa **boolean**; tada je **x = y** ispravan izraz što bi u prethodnom primeru verovatno predstavljalo grešku u kucanju.)

Sličan problem u C-u i C++-u jeste korišćenje konjunkcije i disjunkcije nad bitovima umesto logičkih varijanata. Konjunkcije i disjunkcije nad bitovima se pišu s jednim simbolom (& ili |), dok se logička konjunkcija i disjunkcija pišu s dva (**&&** i **||**). Isto kao i sa = i ==, lako je greškom upisati samo jedan znak umesto dva. Javin prevodilac to ponovo sprečava, jer neće dozvoliti da iz nehata iskoristite pogrešan tip u izrazu.

Eksplicitna konverzija tipova

Kada je potrebno, Java automatski pretvara jedan tip podataka u drugi. Na primer, ako dodelite celobrojnu vrednost promenljivoj u formatu pokretnog zareza, prevodilac će automatski konvertovati tip **int** u tip **float**. Programer može i da zahteva eksplicitnu konverziju (engl. *casting*) ili da je sprovede u slučajevima u kojima se normalno ne bi dogodila.

Da biste izvršili eksplicitnu konverziju, navedite željeni tip unutar zagrada, s leve strane vrednosti koju treba konvertovati. Na primer:

```
//: operatori/izriciteKonverzije.java

public class izriciteKonverzije {
  public static void main(String[] args) {
     int i = 200;
     long lng = (long)i;
      lng = i; // "Proširujuća konverzija", pa eksplicitna konverzija
               // zapravo nije potrebna
     long lng2 = (long)200;
     lng2 = 200;
     // "Sužavajuća konverzija":
     i = (int) lng2; // Ovde je eksplicitna konverzija neophodna
  }
} ///:~
```

Kao što vidite, eksplicitnu konverziju je moguće izvršiti na numeričkoj vrednosti, kao i na promenljivoj. Imajte u vidu da je dozvoljena i izlišna eksplicitna konverzija. Na primer, prevodilac, kad god treba, automatski prevodi vrednost tipa **int** u tip **long**; ipak, suvišna konverzija se dozvoljava, jer time možete da je naglasite ili da kôd učinite jasnijim. U drugim situacijama, eksplicitna konverzija može biti neophodna da bi kôd uopšte mogao biti preveden.

U C-u i C++-u, automatska konverzija može izazvati malo glavobolje. U Javi je automatska konverzija bezbedna osim kada vršite takozvanu *sužavajuću konverziju* (tj. kada prelazite sa tipa podataka koji može da čuva više informacija na onaj koji čuva manje), pri čemu rizikujete da izgubite informacije. U tom slučaju prevodilac vas prisiljava da vršite eksplicitnu konverziju, obaveštavajući vas na taj način da „to može biti opasno – a ako ipak to želite, moraćete izričito da tražite konverziju". *Proširujuću konverziju* ne treba izričito zahtevati jer novi tip može da čuva znatno više informacija nego stari, pa se informacije nikad ne gube.

Java dozvoljava da izvršite eksplicitnu konverziju iz bilo kog prostog tipa u bilo koji drugi prost tip, izuzev tipa **boolean** koji uopšte ne možete konvertovati. Klase takođe ne dozvoljavaju eksplicitnu konverziju. Da biste konvertovali jednu klasu u drugu, za to mora da postoji posebna metoda. (Kasnije u knjizi videćete i da se može vršiti eksplicitna konverzija objekata u okviru *porodice* tipova; **Hrast** može biti eksplicitno konvertovan u **Drvo** i obrnuto, ali ne i u strani tip kao što je **Stena**.)

Odsecanje i zaokruživanje

Kada obavljate sužavajuću konverziju, morate paziti na odsecanje i zaokruživanje. Primera radi, ako realan broj tipa **float** izričito pretvorite u ceo broj tipa **int**, šta uradi Java? Na primer, broj 29.7 pretvarate u **int** – hoćete li dobiti 30 ili 29? Odgovor na to pitanje naveden je u sledećem primeru:

```
//: operatori/IzricitaKonverzijaBrojeva.java
// Šta se dešava kada broj tipa float
// ili double izričito pretvorite u ceo broj?
import static net.mindview.util.Print.*;

public class IzricitaKonverzijaBrojeva {
  public static void main(String[] args) {
    double iznad = 0.7, ispod = 0.4;
    float fiznad = 0.7f, fispod = 0.4f;
    print("(int)iznad: " + (int)iznad);
    print("(int)ispod: " + (int)ispod);
    print("(int)fiznad: " + (int)fiznad);
    print("(int)fispod: " + (int)fispod);
  }
} /* Ispis:
(int)iznad: 0
(int)ispod: 0
(int)fiznad: 0
(int)fispod: 0
*///:~
```

Dakle, odgovor je da se prilikom eksplicitne konverzije tipa **float** ili **double** u ceo broj (tip **int**), decimale uvek odsecaju. Ukoliko hoćete da rezultat bude zaokružen, upotrebite metodu **round**() iz biblioteke **java.lang.Math**:

```
//: operatori/ZaokruzivanjeBrojeva.java
// Zaokruzivanje brojeva tipa float i double.
import static net.mindview.util.Print.*;

public class ZaokruzivanjeBrojeva {
  public static void main(String[] args) {
    double iznad = 0.7, ispod = 0.4;
    float fiznad = 0.7f, fispod = 0.4f;
    print("Math.round(iznad): " + Math.round(iznad));
    print("Math.round(ispod): " + Math.round(ispod));
    print("Math.round(fiznad): " + Math.round(fiznad));
    print("Math.round(fispod): " + Math.round(fispod));
  }
} /* Ispis:
Math.round(iznad): 1
Math.round(ispod): 0
Math.round(fiznad): 1
Math.round(fispod): 0
*///:~
```

Pošto **round**() pripada biblioteci **java.lang**, nije potrebna posebna naredba **import** za njeno uvoženje i korišćenje.

Unapređenje tipova

Ako vršite bilo koju matematičku operaciju i operaciju nad bitovima s prostim tipovima podataka, koji su manji od **int** (tj., **char**, **byte** ili **short**), videćete da će te vrednosti biti unapređene u tip **int** pre nego što se operacije izvrše, a rezultat će takođe biti tipa **int**. Stoga, ako tu vrednost želite da ponovo dodelite manjem tipu, morate da koristite eksplicitnu konverziju. (A pošto vršite dodelu manjem tipu, mogu se izgubiti informacije.) U principu, najveći tip podataka u izrazu određuje veličinu rezultata tog izraza; ako pomnožite **float** i **double**, rezultat će biti tipa **double**; ako saberete **int** i **long**, rezultat će biti tipa **long**.

Java nema operator za određivanje veličine

U C-u i C++-u operator **sizeof**() zadovoljava specifičnu potrebu: on daje broj bajtova koje podaci zauzimaju. Prenosivost je najbitniji razlog za korišćenje operatora **sizeof**() u C-u i C++-u. Različiti tipovi podataka mogu imati različite veličine na različitim računarima. Stoga programer mora da ima informaciju o veličini tipova koje koristi, pre nego što izvrši neku operaciju za koju je bitna veličina promenljivih. Na primer, jedan računar može da čuva celobrojne veličine sa 32 bita, dok drugi to može da radi sa 16 bitova. Programi na prvom računaru mogli bi da čuvaju veće vrednosti u celobrojnim promenljivama. Kao što možete da zamislite, prenosivost zadaje velike glavobolje programerima na C-u i C++-u.

Javi ne treba operator za određivanje veličine jer su svi tipovi podataka uvek iste veličine, na svim računarima. Na ovom nivou ne razmišljajte o prenosivosti – ona je ugrađena u jezik.

Sažet pregled operatora

Sledeći primer pokazuje koji prosti tipovi mogu da koriste određene operatore. U osnovi, to je isti primer koji se stalno ponavlja, samo se upotrebljavaju drugi prosti tipovi. Ova datoteka se može prevesti bez grešaka, jer su redovi koji mogu da izazovu grešku pretvoreni u komentar pomoću //!.

```
//: operatori/SviOperatori.java
// Prikazuje sve operatore nad svim prostim tipovima podataka
// da bi pokazao koje su operacije dozvoljene.

public class SviOperatori {
  // Da bi se prihvatili rezultati logičkog testa:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
```

```
    // Aritmetički operatori:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Poređenje i logički:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Operatori nad bitovima:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Složena dodela:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Eksplicitna konverzija:
    //! char c = (char)x;
    //! byte b = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
}
void charTest(char x, char y) {
```

```
    // Aritmetički operatori:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Poređenje i logički:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori nad bitovima:
    x= (char)-y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Složena dodela:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Eksplicitna konverzija:
    //! boolean bl = (boolean)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void byteTest(byte x, byte y) {
```

```
    // Aritmetički operatori:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Poređenje i logički:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori nad bitovima:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Složena dodela:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Eksplicitna konverzija:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void shortTest(short x, short y) {
```

```
    // Aritmetički operatori:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Poređenje i logički:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori nad bitovima:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    x = (short)(x >>> 1);
    // Složena dodela:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Eksplicitna konverzija:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void intTest(int x, int y) {
```

```
    // Aritmetički operatori:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Poređenje i logički:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori nad bitovima:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Složena dodela:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Eksplicitna konverzija:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void longTest(long x, long y) {
```

```
  // Aritmetički operatori:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Poređenje i logički:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operatori nad bitovima:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  x = x >>> 1;
  // Složena dodela:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Eksplicitna konverzija:
  //! boolean b = (boolean)x;
  char c = (char)x;
  byte b = (byte)x;
  short s = (short)x;
  int i = (int)x;
  float f = (float)x;
  double d = (double)x;
}
void floatTest(float x, float y) {
```

```
  // Aritmetički operatori:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Poređenje i logički:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operatori nad bitovima:
  //! x = ~y;
  //! x = x & y;
  //! x = x | y;
  //! x = x ^ y;
  //! x = x << 1;
  //! x = x >> 1;
  //! x = x >>> 1;
  // Složena dodela:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  //! x <<= 1;
  //! x >>= 1;
  //! x >>>= 1;
  //! x &= y;
  //! x ^= y;
  //! x |= y;
  // Eksplicitna konverzija:
  //! boolean bl = (boolean)x;
  char c = (char)x;
  byte b = (byte)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
  double d = (double)x;
}
void doubleTest(double x, double y) {
```

```
    // Aritmetički operatori:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Poređenje i logički:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori nad bitovima:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Složena dodela:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Eksplicitna konverzija:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
} ///:~
```

Obratite pažnju na to da je tip **boolean** prilično ograničen. Možete mu dodeliti vrednosti **true** i **false** i ispitivati vrednost, ali ne možete da vršite sabiranje ili bilo koju drugu operaciju.

Kod tipova **char, byte** i **short** možete videti efekte unapredenja tipa (proširivanja) koje se javlja pri upotrebi aritmetičkih operatora. Svaka aritmetička operacija primenjena na ove tipove daje rezultat tipa **int**, koji mora eksplicitnom konverzijom da bude vraćen u prvobitni tip (što je sužavajuća konverzija pri kojoj se mogu izgubiti informacije). Eksplicitna konverzija nije potrebna kada koristite vrednosti tipa **int**, pošto je sve već tipa **int**. Ipak, nemojte se opustiti i misliti kako je sve sigurno. Ako pomnožite dve vrednosti tipa **int** koje su dovoljno velike, doći će do prekoračenja. To pokazuje naredni primer:

```
//: operatori/Prekoracenje.java
// Iznenađenje! Java dozvoljava prekoračenje.

public class Prekoracenje {
  public static void main(String[] args) {
    int veliki = Integer.MAX_VALUE;
    System.out.println("veliki = " + veliki);
    int veci = veliki * 4;
    System.out.println("jos veci = " + veci);
  }
} /* Ispis:
veliki = 2147483647
jos veci = -4
* ///:~
```

Ne dobijate niti poruku o grešci niti upozorenje od prevodioca, niti se javlja izuzetak pri izvršavanju. Java je dobra, ali nije *toliko* dobra.

Složena dodeljivanja *ne* zahtevaju eksplicitnu konverziju za tipove **char, byte** ili **short**, iako vrše proširenja koja imaju isti rezultat kao i direktne aritmetičke operacije. S druge strane, izostavljanjem eksplicitne konverzije zasigurno se uprošćava kod.

Možete primetiti kako svi prosti tipovi, osim tipa **boolean**, mogu biti eksplicitno konvertovani u bilo koji drugi prost tip. Ponovo napominjem, vodite računa o efektu sužavanja kada vršite eksplicitnu konverziju u manji tip, jer možete nehotice da izgubite informacije tokom konverzije.

Vežba 14: (3) Napišite metodu koja prima dva znakovna niza kao argumente i poredi ih pomoću svih **boolean** poređenja. Ispišite rezultate. Za == i **!**= obavite i test **equals**(). Pozovite svoju metodu iz **main**() s nekoliko različitih **String** objekata.

Sažetak

Ako imate iskustva s bilo kojim jezikom čija je sintaksa slična C-ovoj, videli ste kako su Javini operatori toliko slični onima iz vama znanih jezika da ih gotovo ni ne treba učiti. Ukoliko vam je poglavlje bilo tesko, pogledajte multimedijsku prezentaciju *Thinking in C*, koja je dostupna na adresi *www.MindView.net*.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.net*.

Kontrolisanje izvršavanja

Kao i svesno biće, i program mora da upravlja svojim okruženjem i da bira šta će da radi. U Javi birate pomoću naredaba za kontrolisanje toka programa.

JAVA KORISTI SVE NAREDBE JEZIKA C ZA KONTROLU IZVRŠAVANJA, STOGA ĆE VAM, AKO STE programirali na C-u ili C++-u, veći deo ovoga što sledi biti poznat. Mnogi proceduralni programski jezici imaju neku vrstu naredbe za kontrolu i među jezicima često postoje sličnosti. Među rezervisane reči za kontrolu izvršavanja u Javi spadaju **if-else**, **while**, **do-while**, **for**, **return**, **break** i naredbu izbora **switch**. Java, međutim, ne podržava prilično opasnu naredbu **goto** (koja i dalje može biti najefikasniji način za rešavanje određene vrste problema). I dalje možete da napravite skok koji je nalik na **goto**, ali je mnogo više ograničen od tipične naredbe **goto**.

Logičke vrednosti

Sve uslovne naredbe koriste tačnost i netačnost uslovnih izraza da bi odredile tok izvršavanja. Primer uslovnog izraza je **a == b**. Tu se koristi operator poređenja == da se utvrdi da li je vrednost **a** jednaka vrednosti **b**. Ovaj izraz vraća **true** ili **false**. I bilo koji operator poređenja iz prethodnog poglavlja može se koristiti za formiranje uslovnog izraza. Obratite pažnju na to da Java ne dozvoljava da kao logički tip (**boolean**) koristite broj, iako je to dozvoljeno u C-u i C++-u (gde je tačno sve različito od nule, a netačno je nula). Ako želite da koristite nelogički tip u logičkom testu, na primer **if(a)**, prvo ga morate pretvoriti u vrednost tipa **boolean** pomoću uslovnog izraza, na primer **if(a != 0)**.

Naredba **if-else**

Naredba **if-else** je najosnovniji način na koji se kontroliše tok programa. Deo **else** nije obavezan, pa stoga **if** možete da koristite u dva oblika:

```
if(logički-izraz)
  naredba
```

ili

```
if(logički-izraz)
  naredba
else
  naredba
```

Logički izraz mora da vraća rezultat tipa **boolean**. *Naredba* je jednostavna naredba koja se završava znakom tačka i zarez, ili složena naredba, odnosno grupa naredaba unutar vitičastih zagrada. Svaki put kada se koristi termin *naredba*, podrazumeva se da ona može biti jednostavna ili složena.

Kao primer za **if-else**, sledi metoda **test**() koja saopštava da li je pretpostavljena vrednost veća, manja ili jednaka ciljnoj vrednosti:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
  static int rezultat = 0;
  static void test(int vrednost, int cilj) {
    if(vrednost > cilj)
      rezultat = +1;
    else if(vrednost < cilj)
      rezultat = -1;
    else
      rezultat = 0; // Poklopile su se
    }
  public static void main(String[] args) {
    test(10, 5);
    print(rezultat);
    test(5, 10);
    print(rezultat);
    test(5, 5);
    print(rezultat);
  }
} /* Ispis:
1
-1
0
*///:~
```

Negde na sredini koda metode **test**() vidite kombinaciju **else if**, što nije nova rezervisana reč, nego jedan **else** iza kojega je nova naredba **if**.

Iako je Java jezik „slobodnog oblika", kao C i C++, uvlačenje tela naredbe za kontrolu izvršavanja je korisno kako bi čitalac mogao lakše da odredi gde ono počinje i gde se završava.

Petlje

Petlje se kontrolišu rezervisanim rečima **while**, **do-while** i **for**, koje se ponekad nazivaju i *iteracione naredbe*. *Naredba* se ponavlja sve dok *logički-izraz* za kontrolu ne dobije vrednost **false**. Oblik petlje **while** je sledeći:

```
while (logički-izraz)
  naredba
```

Logički-izraz se izračunava jednom na početku petlje i ponovo pre svakog koraka izvršavanja *naredbe*.

Sledi jednostavan primer koji generiše slučajne brojeve sve dok određeni uslov ne bude ispunjen.

```
//: control/PrimerZaWhile.java
// Prikazuje petlju while.

public class PrimerZaWhile {
   static boolean uslov()  {
   boolean rezultat = Math.random() < 0.99;
   System.out.println(rezultat +  ", ");
   return rezultat;
}
  public static void main(String[] args) {
      while(uslov() )
         System.out.println("Unutar petlje 'while'");
      System.out.println("Izašao iz petlje 'while'");
  }
} /* (Izvršite da biste videli izlaz) *///:~
```

Metoda **uslov()** koristi statičnu metodu **random()** iz biblioteke **Math**, koja generiše vrednost tipa **double** između 0 i 1 (uključujući 0, ali ne i 1). Promenljivoj **rezultat** vrednost daje operator poređenja <, čiji je rezultat tipa **boolean**. Ukoliko odštampate vrednost tipa **boolean**, automatski ćete dobiti odgovarajući znakovni niz true ili false. Uslovni izraz za **while** glasi: „Ponavljaj naredbe u telu petlje sve dok metoda **uslov()** vraća **true**".

Petlja do-while

Oblik petlje **do-while** je:

```
do
  naredba
while (logički-izraz);
```

Jedina razlika između **while** i **do-while** jeste ta da se naredba u petlji **do-while** uvek izvrši barem jednom, čak i ako izraz ima vrednost **false** i prvi put. Ako je uslovni izraz netačan pre ulaska u petlju **while**, naredba se neće izvršiti nijednom. U praksi, **do-while** se rede javlja od **while**.

Petlja for

Petlja **for** je verovatno najčešći oblik ponavljanja (ciklusa). Ona vrši inicijalizaciju pre prvog izvršavanja ciklusa. Zatim ispituje uslov (*logički-izraz*) i ako je on ispunjen, izvršava *naredbu*, a potom na kraju svakog ciklusa izvršava *korak* i ponovo ispituje *logički-izraz*. Oblik petlje **for** je:

```
for (inicijalizacija; logički-izraz; korak)
  naredba
```

Bilo koji od izraza *inicijalizacija, logički-izraz* ili *korak* može biti prazan. Logički izraz se proverava pre svakog izvršavanja naredbe. Čim izraz dobije vrednost **false**, petlja se prekida i izvršavanje se nastavlja prvom naredbom nakon petlje **for**. Na kraju svakog ciklusa (posle svakog izvršavanja naredbe), izvršava se *korak*.

Petlje **for** se obično koriste za nabrajanje:

```
//: control/:ListaZnakova.java
// Prikazuje petlju for tako što
// ispisuje sva mala ASCII slova.

public class ListaZnakova {
  public static void main(String[] args) {
     for( char c = 0; c < 128; c++)
       if (Character.isLowerCase (c))
      System.out.println("vrednost: " + (int)c +
        " znak: " + c);
     }
}  /* Ispis:
vrednost: 97 znak: a
vrednost: 98 znak: b
vrednost: 99 znak: c
vrednost: 100 znak: d
vrednost: 101 znak: e
vrednost: 102 znak: f
vrednost: 103 znak: g
vrednost: 104 znak: h
vrednost: 105 znak: i
vrednost: 106 znak: j
...
* ///:~
```

Obratite pažnju na to da je promenljiva **c** definisana na mestu gde se koristi, unutar kontrolnog izraza petlje **for**, umesto na početku metode **main**(). Oblast važenja promenljive **c** je samo naredba koju kontroliše petlja **for**.

U programu se koristi i omotačka klasa **java.lang.Character**, koja omotava prost tip **char** u objekat, a pruža i druge usluge – njena **static** metoda **isLowerCase**() otkriva da li trenutni znak pripada malim slovima.

Tradicionalni proceduralni jezici kao što je C, zahtevaju da sve promenljive budu definisane na početku bloka, tako da prevodilac, kada pravi blok, može da rezerviše prostor za njih. U Javi i C++-u definicije promenljivih možete da napišete bilo gde u bloku, tamo gde vam trebaju. Zbog toga je stil pisanja prirodniji i kôd razumljiviji.

Vežba 1: (1) Napišite program koji ispisuje brojeve od 1 do 100.

Vežba 2: (2) Napišite program koji generiše 25 slučajnih vrednosti tipa **int**. Za svaku vrednost upotrebite naredbu **if-else** da biste odredili da li je veća, manja ili jednaka drugoj slučajno generisanoj vrednosti.

Vežba 3: (1) Izmenite vežbu 2 tako da njen kôd uklopite u beskonačnu petlju **while**. Program će tada raditi sve dok ga ne prekinete s tastature (obično pritiskom na Control-C).

Vežba 4: (3) Napišite program koji pomoću dve ugneždene petlje **for** i operatora modulo deljenja (%) pronalazi i ispisuje proste brojeve (cele brojeve koji su deljivi samo sa 1 i sami sa sobom).

Vežba 5: (4) Ponovite vežbu 10 iz prethodnog poglavlja, ali tako da za prikazivanje jedinica i nula upotrebite ternarni operator, te ispitivanje bitova umesto metode **Integer.toBinaryString()**.

Operator zarez

Ranije u ovom poglavlju naveo sam da se *operator* zarez (ne *razdelnik* zarez koji služi da odvoji definicije, kao i argumente metode) koristi samo u kontrolnim izrazima petlje **for**. Kako prilikom inicijalizacije, tako i pri izvršavanju koraka kontrolnog izraza, možete navesti više naredaba razdvojenih zarezima i one će biti izvršene sekvencijalno. Pomoću operatora zarez možete definisati više promenljivih unutar naredbe **for**, ali one moraju biti istog tipa:

```
//: control/ OperatorZarez.java

public class OperatorZarez {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.out.println("i = " + i + " j = " + j);
    }
  }
} /* Ispis:
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
* ///:~
```

Definicija celobrojnih promenljivih u naredbi **for** obuhvata i **i** i **j**. Inicijalizacioni deo može imati više definicija *istog tipa*. Mogućnost da se promenljive definišu unutar kontrolnog izraza ograničena je samo na petlje **for**. Ovaj pristup ne možete da koristite ni sa jednom drugom naredbom izbora ili petlje.

Uočite da su, i u delu za inicijalizaciju kao i u delu koraka, naredbe izvršavane u sekvencijalnom poretku.

Foreach sintaksa

Java SE5 uvodi novu i sažetiju sintaksu naredbe **for** koja se upotrebljava za nizove i kontejnere (o kojima ćete više saznati u poglavljima *Nizovi* i *Detaljno razmatranje kontejnera*). Nju često nazivaju *foreach* sintaksa, jer u njoj ne morate sami da pravite celobrojnu promenljivu za brojanje prolaza kroz sekvencu stavki – foreach automatski pravi sve stavke umesto vas.

Na primer, pretpostavimo da imate niz brojeva tipa **float** i da hoćete da izaberete svaki element tog niza:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat  {
   public static void main(String[] args)  {
        Random slucajan = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
           f[i] = slucajan.nextFloat();
        for(float x : f)
           System.out.println(x);
     }
} /* Ispis:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:~
```

Niz je napunjen starom **for** petljom, jer mu se mora pristupati pomoću indeksa. Foreach sintaksu vidite u redu:

```
for(float x : f){
```

Njime se definiše promenljiva **x** tipa **float** kojoj se sekvencijalno dodeljuje svaki element niza **f**.

Za korišćenje foreach sintakse kandidat je svaka metoda koja vraća niz. Primera radi, klasa **String** ima metodu **toCharArray**() koja vraća niz tipa **char**, pa je lako izdvojiti svaki znak znakovnog niza:

```
//: control/ForEachString.java

public class ForEachString  {
   public static void main(String[] args)  {
        for(char znak : "Afrička lastavica".toCharArray() )
           System.out.print(znak + " ");
     }
} /* Ispis:
A f r i č k a   l a s t a v i c a
*///:~
```

Kao što ćete videti u poglavlju *Čuvanje objekata*, foreach sintaksa radi sa svakim objektom za koji se zna poredak njegovih članova (koji pripada klasi **Iterable**).

Mnoge **for** naredbe obuhvataju prolazak kroz sekvencu celobrojnih vrednosti, recimo ovako:

```
for(int i = 0; i < 100; i++)
```

U ovom slučaju foreach sintaksa ne bi radila, sem ukoliko unapred ne napravite niz **int** brojeva. Da bih to pojednostavio, u biblioteku **net.mindview.util.Range** stavio sam metodu **range**() koja automatski generiše odgovarajući niz. Namera mi je bila da se **range**() koristi kao **static** uvoz:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
   public static void main(String[ ] args) {
      for(int i : range(10)) // 0..9
         printnb(i + " ");
      print();
      for(int i : range(5, 10)) // 5..9
         printnb(i + " ");
      print();
      for(int i : range(5, 20, 3)) // 5..20 uz korak 3
         printnb(i + " ");
      print();
   }
} /* Ispis:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*///:~
```

Metoda **range**() bila je *preklopljena*, što znači da se uz isto ime metode mogu zadavati različite liste argumenata (o preklapanju će uskoro biti reči). Prvi preklopljeni oblik metode **range**() počinje od nule i daje brojeve do vrha opsega (engl. *range*), isključujući sam vrh. Drugi oblik počinje od prve zadate vrednosti i daje brojeve do iznosa za jedan manjeg od drugog argumenta, a treći oblik prima korak povećanja u svom trećem argumentu. **range**() je veoma prosta verzija *generatora*, kao što ćete videti u nastavku knjige.

Imajte u vidu da **range**() češće omogućava korišćenje foreach sintakse i stoga možda povećava čitljivost koda, ali donekle smanjuje efikasnost. Zato, ako pokušavate da povećate efikasnost svog programa, izmerite njegove performanse *profajlerom* (engl. *profiler*).

Možda ste primetili da je u prethodnom primeru, sem metode **print**(), upotrebljena i **printnb**(). Ona ne zadaje prelazak u novi red, pa služi za višekratno pisanje u istom redu.

Foreach sintaksa štedi nešto vremena prilikom pisanja koda, ali je važnije to što je takav kôd mnogo čitljiviji – kazuje *šta* hoćete (dobiti svaki element niza), a ne *kako* to postižete („Pravim ovaj indeks da bih mogao izabrati svaki element niza"). U ovoj knjizi će se foreach sintaksa upotrebljavati gde god je to moguće.

Rezervisana reč **return**

Za *bezuslovno skakanje* služi nekoliko rezervisanih reči: **return**, **break** i **continue**, a postoji i način skakanja na naredbu sa oznakom (engl. *label*), slično naredbi **goto** u drugim jezicima.

Rezervisana reč **return** ima dve namene: ona određuje vrednost koju će metoda vratiti (ako uopšte vraća vrednost, tj. ako povratna vrednost nije tipa **void**) i prouzrokuje trenutan izlazak iz metode. Prethodnu metodu **test()** možemo ponovo napisati tako da iskoristimo tu pogodnost:

```
//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
  static int test(int vrednost, int cilj) {
    if(vrednost > cilj)
      return +1;
    else if(vrednost < cilj)
      return -1;
    else
      return 0; // Poklopile su se
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
} /* Ispis:
1
-1
0
* ///:~
```

Sada **else** nije potrebna jer **return** završava rad metode.

Ukoliko u metodi koja vraća **void** nemate naredbu **return**, završetak metode čini implicitan **return**, pa naredbu **return** ne morate pisati. Međutim, ako metoda vraća bilo koji drugi tip sem **void**, vaš je posao da obezbedite da za sve moguće putanje kroz njen kod postoji neka povratna vrednost.

Vežba 6: (2) Prepravite metode **test()** iz oba prethodna programa tako da primaju još dva argumenta, **početak** i **kraj**. Za svaku *vrednost* treba proveriti da li je njen iznos između **početka** i **kraja** (uključivo).

Rezervisane reči **break** i **continue**

Tok petlje takođe možete da kontrolišete pomoću rezervisanih reči **break** i **continue** unutar tela bilo koje petlje. Naredba **break** izlazi iz petlje, bez izvršavanja ostalih naredaba unutar petlje. Naredba **continue** zaustavlja izvršavanje tekućeg ciklusa i vraća se na početak petlje kako bi počeo nov korak.

Sledeći program pokazuje kako se koriste **break** i **continue** unutar petlji **for** i **while**:

```
//: control/BreakIContinue.java
// Pokazuje rezervisane reči break i continue.
import static net.mindview.util.Range.*;

public class BreakIContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
        if(i == 74) break; // Izađi iz petlje
        if(i % 9 != 0) continue; // Sledeći ciklus
        System.out.print(i + " ");
    }
    System.out.println();
    // Upotreba foreach sintakse:
    for (int i : range(100))  {
         if(i == 74) break;  // Izađi iz petlje
         if(i % 9 != 0) continue; // Sledeći ciklus
        System.out.print(i + " ");
    }
    System.out.println();
    int i = 0;
    // "beskonačna" petlja:
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Izađi iz petlje
      if(i % 10 != 0) continue; // Povratak na vrh petlje
      System.out.print(i + " ");
    }
  }
} /* Ispis:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:~
```

U petlji **for** vrednost **i** nikada ne dođe do 100 jer naredba **break** prekine petlju kada **i** postane 74. Naredba **break** se na ovaj način obično koristi samo kada nije unapred poznato kada će se stvoriti uslov za izlazak. Naredba **continue** nastavlja izvršavanje od vrha petlje (stoga uvećava **i**) kad god **i** nije deljivo sa 9. Kada jeste, ta vrednost se ispisuje.

U drugoj petlji **for** prikazano je kako se upotrebljava foreach sintaksa i vidi se da ona daje jednake rezultate.

Poslednji deo prikazuje beskonačnu petlju **while**, koja bi se, teorijski, izvršavala neprekidno. Međutim, unutar petlje se nalazi naredba **break** pomoću koje se izlazi iz petlje. Pored toga, primetićete da naredba **continue** vraća izvršavanje na vrh petlje i da ne izvršava ostatak tela petlje nakon sebe. (Stoga se ispisivanje u drugoj petlji javlja samo kada je vrednost **i** deljiva sa 10.) Na izlazu se ispisuje 0 jer je vrednost izraza 0 % 9 jednaka 0.

Drugi oblik beskonačne petlje je **for(;;)**. Prevodilac tumači **while(true)** i **for(;;)** na isti način, pa je izbor isključivo pitanje programerskog ukusa.

Vežba 7: (1) Prepravite vežbu 1 tako da program završava rad pomoću rezervisane reči **break**, kada vrednost dođe do 99. Pokušajte da umesto toga koristite **return**.

Čuveni goto

Rezervisana reč **goto** postoji u programskim jezicima od samog početka. Zaista, **goto** je nastao kao posledica kontrole programa u mašinskom jeziku: „Ako je ispunjen uslov A, skoči tamo, u suprotnom, skoči ovamo". Kad pročitate asemblerski kôd koji na kraju generiše praktično bilo koji prevodilac, primetićete da se kontrola programa sastoji od više skokova. (Prevodilac Jave pravi sopstveni „asemblerski kôd", ali njega izvršava virtuelna mašina Jave (JVM), a ne hardverski procesor.)

Naredba **goto** je skok na nivou izvornog koda i to ju je iznelo na loš glas. Ako će program stalno skakati s jedne tačke na drugu, zar ne postoji način da se kôd reorganizuje tako da tok kontrole ne bude toliko pun skokova? Naredba **goto** je pala u pravu nemilost nakon objavljivanja čuvenog članka Edsgera Dijkstre „Goto se smatra štetnim" i od tada su napadi na nju postali popularan sport.

Kao što je takvoj situaciji svojstveno, srednji put je najbolji. Problem nije u korišćenju naredbe **goto**, već u njenom prečestom korišćenju – u retkim situacijama **goto** je najbolji način za strukturiranje toka izvršavanja.

Iako je reč **goto** rezervisana u Javi, ona se u jeziku ne koristi; Java nema naredbu **goto**. Ipak, ona ima nešto što liči na skok, povezano sa rezervisanim rečima **break** i **continue**. To nije skok, već više način da se izađe iz petlje. On se često povezuje s naredbom **goto** zato što koristi isti mehanizam: oznaku (engl. *label*).

Oznaka je identifikator iza koga sledi dvotačka, na primer:

```
oznaka1:
```

U Javi se oznaka koristi *samo* neposredno pre iteracione naredbe. I to znači *baš* neposredno – ne valja stavljati bilo koje druge naredbe između oznake i petlje. Oznaku treba staviti pre petlje samo ako unutar te petlje želite da ugnezdite još jednu petlju ili naredbu izbora (**switch**), o kojoj ćete uskoro saznati šta treba. Rezervisane reči **break** i **continue** obično prekidaju samo tekuću petlju, ali kada se koriste sa oznakom, one će prekinuti sve petlje do nivoa gde se nalazi oznaka:

```
oznaka1:
spoljna-petlja {
  unutrašnja-petlja {
    // ...
```

```
      break; // (1)
      // ...
      continue; // (2)
      // ...
      continue oznaka1; // (3)
      // ...
      break oznaka1; // (4)
    }
  }
```

U (1), **break** prekida unutrašnju petlju a vi nastavljate sa spoljašnjom. U (2), **continue** se vraća na početak unutrašnje petlje. Ali u (3), **continue oznaka1** prekida i unutrašnju *i* spoljašnju petlju, sve do **oznake1**. Zatim se petlja nastavlja, ali počev od spoljnog ciklusa. U (4), **break oznaka1** takođe vrši prekid sve do **oznake1**, ali ne ulazi ponovo u petlju. U stvari, ona zaista prekida obe petlje.

Evo primera s petljom **for**:

```
//: control/OznacenaPetljaFor.java
// Petlja for sa "označenom naredbom break" i "označenom naredbom continue".
import static net.mindview.util.Print.*;

public class OznacenaPetljaFor {
  public static void main(String[] args) {
    int i = 0;
    spoljasnja: // Ovde ne mogu da stoje naredbe
    for(; true ;) { // beskonačna petlja
      unutrasnja: // Ovde ne mogu da stoje naredbe
      for(; i < 10; i++) {
        print("i = " + i);
        if(i == 2) {
          print("nastavi");
          continue;
        }
        if(i == 3) {
          print("prekini");
          i++; // Inače se i nikad
               // ne uvećava.
          break;
        }
        if(i == 7) {
          print("nastavi spoljašnju");
          i++; // Inače se i nikad
               // ne uvećava.
          continue spoljasnja;
        }
        if(i == 8) {
          print("prekini spoljašnju");
          break spoljasnja;
        }
```

```
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            print("nastavi unutrašnju");
            continue unutrasnja;
          }
        }
      }
    }
    // Ovde ne možete da pozovete
    // označene naredbe break i continue
  }
}/* Ispis:
i = 0
nastavi unutrašnju
i = 1
nastavi unutrašnju
i = 2
nastavi
i = 3
prekini
i = 4
nastavi unutrašnju
i = 5
nastavi unutrašnju
i = 6
nastavi unutrašnju
i = 7
nastavi spoljašnju
i = 8
prekini spoljašnju
* ///:~
```

Obratite pažnju na to da naredba **break** prekida petlju **for** i da se izraz za uvećavanje ne izvršava sve do kraja prolaza kroz petlju **for**. Pošto **break** preskače izraz za uvećavanje, do uvećanja dolazi samo kada je **i** == 3. Naredba **continue spoljasnja** u slučaju kada je **i** == 7 takođe skače na vrh petlje i preskače uvećavanje, stoga se i ono vrši direktno.

Da nema naredbe **break spoljasnja**, ne bi postojao način da se napusti spoljašnja petlja iz unutrašnje petlje, pošto sama naredba **break** može da prekine samo trenutnu petlju. (Isto važi i za **continue**.)

Kada napuštanje petlje takođe znači izlazak iz metode, možete koristiti **return**.

Sledi primer označenih naredaba **break** i **continue** s petljama **while**:

```
//: control/OznacenWhile.java
// Petlje while sa "označenom naredbom break" i "označenom naredbom continue".
import static net.mindview.util.Print.*;

public class OznacenWhile {
  public static void main(String[] args) {
    int i = 0;
```

```
    spoljasnja:
    while(true) {
      print("Spoljašnja petlja while");
      while(true) {
        i++;
        print("i = " + i);
        if(i == 1) {
          print("nastavi");
          continue;
        }
        if(i == 3) {
          print("nastavi spoljašnju");
          continue spoljasnja;
        }
        if(i == 5) {
          print("prekini");
          break;
        }
        if(i == 7) {
          print("prekini spoljašnju");
          break spoljasnja;
        }
      }
    }
  }
}/* Ispis:
Spoljašnja petlja while
i = 1
nastavi
i = 2
i = 3
nastavi spoljašnju
Spoljašnja petlja while
i = 4
i = 5
prekini
Spoljašnja petlja while
i = 6
i = 7
prekini spoljašnju
* ///:~
```

Ista pravila važe i za **while**:

1. Obična naredba **continue** vraća kontrolu na vrh tekuće petlje i nastavlja.
2. Označena naredba **continue** vraća se na oznaku i ponovo ulazi u petlju koja se nalazi odmah iza oznake.
3. **break** izlazi iz petlje.
4. Označena naredba **break** izlazi iz petlje obeležene zadatom oznakom.

Važno je da zapamtite kako se oznaka u Javi *jedino* koristi kad imate ugnežđene petlje i želite da napravite **break** ili **continue** u više nivoa ugnežđivanja.

U svom članku „Goto se smatra štetnim", Dijkstra je posebno imao zamerki na upotrebu oznaka, a ne same naredbe **goto**. Primetio je da broj „bubica" raste s brojem oznaka u programu i da oznake i **goto** otežavaju analizu programa. Obratite pažnju na to da oznake u Javi ne pate od tog problema, pošto je mesto na kome mogu da stoje ograničeno i ne mogu da se koriste za opšti prenos kontrole. Treba primetiti i da je ovo slučaj kada neka osobina jezika postaje korisnija kada se ograniče njene mogućnosti.

Naredba **switch**

Naredba **switch** se nekad naziva i *naredba izbora*. Naredba **switch** bira između nekoliko delova koda u zavisnosti od vrednosti datog izraza. Njen opšti oblik je:

```
switch (izraz) {
  case vrednost1 : naredba; break;
  case vrednost2 : naredba; break;
  case vrednost3 : naredba; break;
  case vrednost4 : naredba; break;
  case vrednost5 : naredba; break;
  // ...
  default: naredba;
}
```

Izraz mora imati celobrojnu ili logičku vrednost. Naredba **switch** poredi rezultat *izraza* i svake *vrednosti*. Ako pronađe ekvivalentnu vrednost, odgovarajuća *naredba* (jedna ili više njih, vitičaste zagrade nisu potrebne) biće izvršena. Ako ne pronađe ekvivalentnu vrednost, biće izvršena *naredba* označena kao **default**.

Primetićete u prethodnoj definiciji da se svaki blok **case** završava sa **break**, čime se tok izvršavanja premešta na kraj tela naredbe **switch**. Ovo je uobičajen način formiranja naredbe **switch**, ali je **break** opcioni. Ako nedostaje, izvršavaće se sledeće **case** naredbe sve dok se ne naiđe na **break**. Iako obično ne želite da se to dešava, iskusnom programeru ovaj pristup može biti koristan. Obratite pažnju na to da poslednja naredba, posle **default**, nema **break** jer se izvršavanje nastavlja baš tamo gde bi se ionako odvijalo nakon **break**. Možete, bez ikakvih posledica, da stavite **break** na kraj bloka **default** ako smatrate da je to važno zbog stila.

Naredba **switch** omogućuje čistu realizaciju izbora između više različitih puteva izvršavanja, ali je za nju potreban izraz koji ima vrednost tipova kao što su **int** ili **char**. Sa naredbom **switch** ne možete kao izraz za izbor koristiti, na primer, znakovni niz ili broj u formatu pokretnog zareza. Za tipove koji nisu celobrojni ili logički morate da koristite niz naredaba **if**. Na kraju sledećeg poglavlja, videćete da to ograničenje olakšava **enum**, nova rezervisana reč koja se u Javi koristi od verzije SE5, jer **enum** lepo radi sa naredbom **switch**.

U narednom primeru određujemo da li su nasumično odabrana slova samoglasnici ili suglasnici:

```
//: control/SamoglasniciISuglasnici.java
// Prikaz naredbe switch.
import java.util.*;
import static.net.mindview.util.Print.*;

public class SamoglasniciISuglasnici {
  public static void main(String[] args) {
      Random slucajan = new Random(47);
      for(int i = 0; i < 100; i++) {
      int c = slucajan.nextInt(26) + 'a');
      printnb((char)c + ", " + c + ": ");
      switch(c) {
         case 'a':
         case 'e':
         case 'i':
         case 'o':
         case 'u': print("samoglasnik");
                   break;
      default:
                   print("suglasnik");
      }
    }
  }
}/* Ispis:
y, 121: suglasnik
n, 110: suglasnik
z, 122: suglasnik
b, 98:  suglasnik
r, 114: suglasnik
n, 110: suglasnik
y, 121: suglasnik
g, 103: suglasnik
c, 99:  suglasnik
f, 102: suglasnik
o, 111: samoglasnik
w, 119: suglasnik
z, 122: suglasnik
...
 ///:~
```

Random.nextInt(26) generiše broj između 0 i 26 (26 jer toliko ima slova u engleskoj abecedi), kojem treba dodati pomeraj slova **'a'** da bi se dobila mala slova. Znakovi u polunavodnicima u naredbama **case** takođe se pretvaraju u celobrojne veličine koje se koriste pri poređenju.

Obratite pažnju na to da naredbe **case** mogu biti naslagane jedne iznad drugih tako da se isti deo koda izvršava u više slučajeva. Vodite računa da je neophodno da stavite naredbu **break** na kraju određenog slučaja, inače će program proći dalje i početi da izvršava naredbe predviđene za naredni slučaj.

U izrazu:

```
int c = slucajan.nextInt(26) + 'a';
```

metoda **Random.nextInt()** daje slučajan ceo broj između 0 i 25, koji se dodaje numeričkoj (ASCII) vrednosti slova **'a'**. To znači da se **'a'** automatski pretvara u tip **int** da bi se izvršilo sabiranje.

Da bi se promenljiva **c** tipa **int** štampala kao znak, mora biti vraćena u tip **char** eksplicitnom konverzijom; inače bi se ispisivali brojevi.

Vežba 8: (2) Napišite naredbu **switch** koja ispisuje poruku za svaki od slučajeva (**case**), a **switch** stavite unutar petlje **for** koja isprobava svaki slučaj. Stavite naredbu **break** nakon svakog slučaja i testirajte program, a zatim izbacite naredbe **break** i vidite šta se dešava.

Vežba 9: (2) *Fibonačijev niz* čine brojevi 1, 1, 2, 3, 5, 8, 13, 21, 34 itd., gde je svaki broj (od trećeg nadalje) jednak zbiru prethodna dva. Napišite metodu koja prima ceo broj kao argument i prikazuje toliko Fibonačijevih brojeva počev od prvog, npr. ako pokrenete **java Fibonacci 5** (gde je **Fibonacci** ime klase) izlaz će biti: 1, 1, 2, 3, 5.

Vežba 10: (5) *Vampirski broj* ima paran broj cifara i dobija se množenjem para brojeva koji imaju po pola originalnih cifara proizvoda. Cifre se uzimaju iz polaznog broja proizvoljnim redosledom. Nisu dozvoljeni parovi završnih nula. Primeri:

1260 = 21 * 60
1827 = 21 * 87
2187 = 27 * 81

Napišite program koji pronalazi sve četvorocifrene vampirske brojeve. (Predložio Dan Forhan.)

Sažetak

Ovim poglavljem se završava proučavanje osnovnih osobina koje se javljaju u većini programskih jezika: računanje, prioritet operatora, eksplicitna konverzija tipova, uslovi i petlje. Sada ste spremni za korake koji će vas približiti svetu objektno orijentisanog programiranja. Naredno poglavlje će obraditi važne teme: inicijalizaciju i čišćenje objekata te veoma bitno sakrivanje realizacije.

Rešenja nekih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, a mogu se kupiti na lokaciji *www.MindView.com*.

Inicijalizacija i čišćenje

S napretkom računarske revolucije, „nesigurno" programiranje je postalo jedan od glavnih razloga za visoku cenu programiranja.

DVE BITNE OBLASTI SIGURNOSTI SU *INICIJALIZACIJA* I *ČIŠĆENJE*. MNOGE GREŠKE U C-U pojavljuju se kada programer zaboravi da inicijalizuje promenljivu. To naročito važi za biblioteke, kada korisnici ne znaju kako da inicijalizuju komponentu iz biblioteke, čak ne znaju ni da je to potrebno. Čišćenje je poseban problem jer je lako zaboraviti na element kada s njim završite – više vas se ne tiče. Zbog toga se resursi koje taj element koristi zadržavaju i lako možete doći u situaciju da vam ponestanu resursi, i to naročito memorija.

Jezik C++ je uveo koncept *konstruktora*, posebne metode koja se automatski poziva pri pravljenju objekta. Java je takođe usvojila konstruktore, a pored toga ima i sakupljač smeća koji automatski oslobađa memorijske resurse kada se ne koriste. U ovom poglavlju razmatra se inicijalizacija i čišćenje i način na koji su podržani u Javi.

Garantovana inicijalizacija pomoću konstruktora

Zamislite da za svaku klasu koju pišete, napravite metodu pod imenom **inicijalizacija**(). Samo ime nagoveštava da bi ona trebalo da bude pozvana pre korišćenja objekta. Nažalost, to znači da korisnik mora da se seti da pozove tu metodu. Projektant klase u Javi može da obezbedi garantovanu inicijalizaciju svakog objekta ako napravi *konstruktor* (engl. *constructor*). Ako klasa ima konstruktor, Java ga automatski poziva pri stvaranju objekta, pre nego što korisnik uopšte može da mu pristupi. Stoga je inicijalizacija zagarantovana.

Sledeći izazov je kako nazvati tu metodu. Tu se javljaju dva problema. Prvo: bilo koje ime može se poklopiti s nekim drugim imenom, koje želite da iskoristite za neku članicu klase. Drugo, pošto je prevodilac odgovoran za poziv konstruktora, on uvek mora da zna koju metodu da pozove. Rešenje iz C++-a deluje najlakše i najlogičnije, pa se takođe koristi i u Javi: ime konstruktora je isto kao i ime klase. Logično je da će takva metoda automatski biti pozvana tokom inicijalizacije.

Sledi jednostavna klasa sa konstruktorom:

```
//: inicijalizacija/JednostavanKonstruktor.java
// Prikazivanje jednostavnog konstruktora.

class Kamen {
  Kamen() { // Ovo je konstruktor
    System.out.print("Kamen");
  }
}

public class JednostavanKonstruktor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Kamen();
  }
```

```
} /* Ispis:
Kamen Kamen Kamen Kamen Kamen Kamen  Kamen Kamen Kamen Kamen
*///:~
```

Kada se pravi objekat:

```
new Kamen();
```

zauzima se memorija i poziva konstruktor. Garantuje se da će objekat biti pravilno inicijalizovan pre nego što bude upotrebljen.

Obratite pažnju na to da se stil pisanja po kome se prvo slovo imena svih metoda piše malim slovom ne odnosi na konstruktore, jer ime konstruktora mora da se *tačno* poklapa sa imenom klase.

Konstruktor koji ne prima argumente naziva se *podrazumevani konstruktor*. U većem delu literature o Javi koju je izdao Sun, njih nazivaju „konstruktori bez argumenata" (engl. *no-arg constructors*). Termin „podrazumevani konstruktor" upotrebljava se dugi niz godina, pa ću ga i ja koristiti. Ali kao i svaka metoda, konstruktor može da ima argumente koji omogućavaju da odredite *kako* će objekat biti napravljen. Prethodni primer lako može da se prepravi tako da konstruktor ima jedan argument:

```
//: inicijalizacija/JednostavanKonstruktor2.java
// Konstruktori mogu da imaju argumente.

class Kamen2 {
  Kamen2(int i) {
    System.out.print("Kamen " + i+"");
  }
}

public class JednostavanKonstruktor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
      new Kamen2(i);
  }
} /* Ispis:
Kamen 0 Kamen 1 Kamen 2 Kamen 3 Kamen 4 Kamen 5  Kamen 6 Kamen 7
*///:~
```

Argumenti konstruktora omogućavaju da obezbedite parametre za inicijalizaciju objekta. Na primer, ako klasa **Drvo** ima konstruktor s jednim celobrojnim argumentom koji određuje visinu drveta, objekat **Drvo** napravićete na sledeći način:

```
Drvo d = new Drvo(12);  // drvo od 12 metara
```

Ako je **Drvo(int)** jedini konstruktor, prevodilac neće dozvoliti da napravite objekat **Drvo** ni na koji drugi način.

Konstruktori eliminišu veliku grupu problema i čine programe razumljivijim. U prethodnom primeru ne postoji eksplicitan poziv nekoj metodi **inicijalizacija**() koja je konceptualno odvojena od pravljenja objekta. U Javi, pravljenje objekta i njegova inicijalizacija objedinjeni su koncepti – jedno bez drugog ne ide.

Konstruktor je neobična vrsta metode jer nema povratnu vrednost. Ovo je potpuno različito od povratne vrednosti tipa **void**, kada metoda ne vraća ništa, ali i dalje imate mogućnost da je napravite tako da vraća nešto. Konstruktori nemaju povratnu vrednost i druga mogućnost ne postoji (izraz **new** vraća referencu novonapravljenog objekta, ali sam konstruktor ne vraća ništa). Ako bi postojala povratna vrednost i ako biste mogli da je birate, prevodilac bi nekako morao da zna šta da radi s tom povratnom vrednošću.

Vežba 1: (1) Napravite klasu koja sadrži neinicijalizovanu **String** referencu. Pokažite da Java tu referencu inicijalizuje vrednošću **null**.

Vežba 2: (2) Napravite klasu s jednim **String** poljem koje se inicijalizuje na mestu definisanja i drugim koje inicijalizuje konstruktor. Po čemu se razlikuju ta dva pristupa?

Preklapanje metoda

Jedno od važnih pitanja u svakom programskom jeziku jeste upotreba imena. Praveći objekat, vi zadajete ime oblasti u memoriji. Metoda je ime neke akcije. Svim objektima i metodama obraćate se preko imena. Dobro odabrana imena čine sistem koji je lakše razumeti i menjati. To veoma liči na pisanje proze – cilj je da budete u vezi sa čitaocima.

Problem nastaje kada nijanse ljudskog jezika primenjujete na programski jezik. Često ista reč ima više različitih značenja – ona je *preklopljena* (engl. *overloaded*). To je korisno, naročito ako se radi o jednostavnim razlikama. Vi ćete reći „operi košulju", „operi kola" i „operi psa". Bilo bi budalasto ako biste morali da govorite „operiVešMašinom košulju", „operiCrevom kola" i „operiKupanjem psa" samo da bi slušalac mogao da napravi razliku između tih akcija. Većina ljudskih jezika je redundantna, pa i dalje možete da odredite značenje čak i ako ispustite nekoliko reči. Jedinstveni identifikatori nam nisu potrebni – značenje možemo shvatiti iz konteksta.

U većini programskih jezika (posebno C-u) svaka metoda (koje se u tim jezicima obično nazivaju *funkcije*) mora imati jedinstven identifikator. Tako ne možete da imate jednu funkciju pod imenom **print**() koja bi ispisivala cele brojeve i drugu istog imena za ispisivanje brojeva u formatu pokretnog zareza – svaka funkcija mora da ima jedinstveno ime.

U Javi (i C++-u) postoji još jedan činilac koji nameće preklapanje imena metoda: konstruktor. Pošto je ime konstruktora unapred određeno imenom klase, konstruktor može imati samo to jedno ime. Ali šta ako želite da objekat pravite na više načina? Na primer, pretpostavimo da pravite klasu koja može da se inicijalizuje na standardan način ili čitanjem informacija iz datoteke. Potrebna su vam dva konstruktora, jedan bez argumenata (podrazumevani konstruktor) i drugi, čiji je argument tipa **String** u kome je ime datoteke iz koje treba inicijalizovati objekat. Oba su konstruktori, pa moraju imati isto ime – ime te klase. Stoga je *preklapanje metoda* suštinski značajno u Javi, jer omogućuje da se isto ime metode koristi s drugačijim tipovima argumenata. Iako je preklapanje metoda neophodno za konstruktore, to je opšta pogodnost dostupna za bilo koju metodu.

Naredni primer pokazuje preklapanje konstruktora i preklapanje metoda:

```
//: inicijalizacija/Preklapanje.java
// Prikaz preklapanja konstruktora
// i preklapanja običnih metoda.
import static net.mindview.util.Print.*;

class Drvo {
  int visina;
  Drvo() {
    print("Sađenje mladice");
    visina = 0;
  }

  Drvo(int pocetnaVisina) {
    visina = pocetnaVisina;
    print("Pravimo novo Drvo koje je visoko "
        + visina + " m");
  }
  void info() {
    print("Drvo je visoko " + visina + " m");
  }
  void info(String s) {
    print(s + ": Drvo je visoko " + visina + " m");
  }
}

public class Preklapanje {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Drvo d = new Drvo(i);
      d.info();
      d.info("preklopljena metoda");
    }
    // Preklopljen konstruktor:
    new Drvo();
  }
} /* Ispis:
Pravimo novo Drvo koje je visoko 0 m
Drvo je visoko 0 m
preklopljena metoda: Drvo je visoko 0 m
Pravimo novo Drvo koje je visoko 1 m
Drvo je visoko 1 m
preklopljena metoda: Drvo je visoko 1 m
Pravimo novo Drvo koje je visoko 2 m
Drvo je visoko 2 m
preklopljena metoda: Drvo je visoko 2 m
Pravimo novo Drvo koje je visoko 3 m
```

```
Drvo je visoko 3 m
preklopljena metoda: Drvo je visoko 3 m
Pravimo novo Drvo koje je visoko 4 m
Drvo je visoko 4 m
preklopljena metoda: Drvo je visoko 4 m
Sađenje mladice
*///:~
```

Objekat klase **Drvo** može da bude napravljen ili kao mladica, bez argumenata, ili kao biljka odgajena u rasadniku s datom visinom. Da bi se to omogućilo, postoje dva konstruktora; jedan bez argumenata i drugi, čiji je argument postojeća visina.

Može vam zatrebati da pozovete metodu **info()** na više načina. Na primer, ako želite da ispišete još neku dodatnu poruku, onda naredite argument tipa **String**, a ako ne želite ništa više da kažete, onda vam ne treba argumenat. Bilo bi neobično da date dva različita imena nečemu što očigledno ima isti koncept. Na sreću, preklapanje metoda omogućava da isto ime koristite za obe stvari.

Razlikovanje preklopljenih metoda

Ako metode imaju isto ime, kako Java zna na koju metodu mislite? Postoji jednostavno pravilo: svaka preklopljena metoda mora da ima jedinstvenu listu tipova argumenata.

Ako razmislite na trenutak, to ima smisla: kako bi drugačije programer mogao da razlikuje dve metode koje imaju isto ime, ako ne po tipovima njihovih argumenata?

Čak je i razlika u poretku argumenata dovoljna da se dve metode razlikuju (ipak ne treba se oslanjati na ovaj pristup jer se otežava održavanje):

```
//: inicijalizacija/PreklapanjePoretkom.java
// Preklapanje zasnovano na poretku argumenata.
import static net.mindview.util.Print.*;

public class PreklapanjePoretkom {
  static void f(String s, int i) {
    print("String: " + s + ", int: " + i);
  }
  static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
  }
  public static void main(String[] args) {
    f("Prvo String", 11);
    f(99, "Prvo Int");
  }
} /* Ispis:
String: Prvo String, int: 11
int: 99, String: Prvo Int
} ///:~
```

Dve metode **f()** imaju identične argumente, ali je njihov poredak drugačiji i po tome se razlikuju.

Preklapanje i prosti tipovi

Prost tip može automatski da se proširi iz manjeg u veći, što u kombinaciji s preklapanjem može da izazove malu zabunu. Naredni primer pokazuje šta se dešava kada se prost tip prosleđuje metodi:

```
//: inicijalizacija/PreklapanjeProstihTipova.java
// Proširivanje prostih tipova i preklapanje.
import static net.mindview.util.Print.*;

public class PreklapanjeProstihTipova {
  void f1(char x) { printnb("f1(char)"); }
  void f1(byte x) { printnb("f1(byte)"); }
  void f1(short x) { printnb("f1(short)"); }
  void f1(int x) { printnb("f1(int)"); }
  void f1(long x) { printnb("f1(long)"); }
  void f1(float x) { printnb("f1(float)"); }
  void f1(double x) { printnb("f1(double)"); }

  void f2(byte x) { printnb("f2(byte)"); }
  void f2(short x) { printnb("f2(short)"); }
  void f2(int x) { printnb("f2(int)"); }
  void f2(long x) { printnb("f2(long)"); }
  void f2(float x) { printnb("f2(float)"); }
  void f2(double x) { printnb("f2(double)"); }

  void f3(short x) { printnb("f3(short)"); }
  void f3(int x) { printnb("f3(int)"); }
  void f3(long x) { printnb("f3(long)"); }
  void f3(float x) { printnb("f3(float)"); }
  void f3(double x) { printnb("f3(double)"); }

  void f4(int x) { printnb("f4(int)"); }
  void f4(long x) { printnb("f4(long)"); }
  void f4(float x) { printnb("f4(float)"); }
  void f4(double x) { printnb("f4(double)"); }

  void f5(long x) { printnb("f5(long)"); }
  void f5(float x) { printnb("f5(float)"); }
  void f5(double x) { printnb("f5(double)"); }

  void f6(float x) { printnb("f6(float)"); }
  void f6(double x) { printnb("f6(double)"); }
  void f7(double x) { printnb("f7(double)"); }
  void testKonstantom() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
  }
  void testChar() {
    char x = 'x';
```

```
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testByte() {
    byte x = 0;
    printnb("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testShort() {
    short x = 0;
    printnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testInt() {
    int x = 0;
    printnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testLong() {
    long x = 0;
    printnb("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testFloat() {
    float x = 0;
    printnb("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testDouble() {
    double x = 0;
    printnb("double: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  public static void main(String[] args) {
    PreklapanjeProstihTipova p =
      new PreklapanjeProstihTipova();
    p.testKonstantom();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
  }
} /* Ispis:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
```

```
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double)
f7(double)
* ///:~
```

Primetićete da se konstanta 5 tretira kao vrednost tipa **int**, pa se, ako postoji, poziva preklopljena metoda čiji je argument tipa **int**. U svim drugim slučajevima tip podatka se proširuje ako je manji od tipa argumenta u metodi. Kod tipa **char** javlja se nešto drugačiji efekat – ako se ne nađe tačno odgovarajući parnjak tipa **char**, on se proširuje na **int**.

Šta se dešava ako je argument *veći* od argumenta koji se očekuje u preklopljenoj metodi? Odgovor daje izmenjeni prethodni primer:

```
//: inicijalizacija/Degradacija.java
// Degradacija prostih tipova i preklapanje.
import static net.mindview.util.Print.*;

public class Degradacija {
  void f1(char x) { print("f1(char)"); }
  void f1(byte x) { print("f1(byte)"); }
  void f1(short x) { print("f1(short)"); }
  void f1(int x) { print("f1(int)"); }
  void f1(long x) { print("f1(long)"); }
  void f1(float x) { print("f1(float)"); }
  void f1(double x) { print("f1(double)"); }

  void f2(char x) { print("f2(char)"); }
  void f2(byte x) { print("f2(byte)"); }
  void f2(short x) { print("f2(short)"); }
  void f2(int x) { print("f2(int)"); }
  void f2(long x) { print("f2(long)"); }
  void f2(float x) { print("f2(float)"); }

  void f3(char x) { print("f3(char)"); }
  void f3(byte x) { print("f3(byte)"); }
  void f3(short x) { print("f3(short)"); }
  void f3(int x) { print("f3(int)"); }
  void f3(long x) { print("f3(long)"); }

  void f4(char x) { print("f4(char)"); }
  void f4(byte x) { print("f4(byte)"); }
  void f4(short x) { print("f4(short)"); }
  void f4(int x) { print("f4(int)"); }

  void f5(char x) { print("f5(char)"); }
  void f5(byte x) { print("f5(byte)"); }
  void f5(short x) { print("f5(short)"); }
```

```
  void f6(char x) { print("f6(char)"); }
  void f6(byte x) { print("f6(byte)"); }

  void f7(char x) { print("f7(char)"); }

  void testDouble() {
    double x = 0;
    print("argument tipa double:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Degradacija p = new Degradacija ();
    p.testDouble();
  }
} /* Ispis:
argument tipa double:
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*///:~
```

U ovom slučaju metode uzimaju sužene vrednosti prostog tipa. Ako je vaš argument širi, morate ga eksplicitno konvertovati u potreban tip. Ukoliko to ne uradite, prevodilac će prijaviti grešku.

Preklapanje povratnih vrednosti

Često se čuje pitanje: „Zašto samo imena klasa i liste argumenata metoda? Zašto ne bismo pravili razliku između metoda na osnovu njihovih povratnih vrednosti?“ Na primer, ove dve metode, koje imaju isto ime i argumente, jasno se razlikuju:

```
void f() { }
int f() { return 1; }
```

Ovo dobro radi kada prevodilac nedvosmisleno može da razluči značenje iz konteksta, kao u **int** x = f(). Metodu, međutim, možete da pozovete i da ignorišete povratnu vrednost; to se često naziva *pozivanje metode zbog njenog sporednog efekta*, pošto vam njena povratna vrednost nije bitna, već vam trebaju drugi efekti poziva metode. Zato, ako metodu pozovete na sledeći način:

```
f();
```

kako izvršno okruženje može da odredi koju metodu f() treba da pozove? I kako bi neko ko čita program to mogao da zna? Zbog ove vrste problema, tip povratne vrednosti ne može da se koristi za razlikovanje metoda.

Podrazumevani konstruktori

Kao što je ranije pomenuto, podrazumevani konstruktor (tj. no-arg konstruktor) nema argumente i koristi se za pravljenje podrazumevanih objekata. Ako napravite klasu koja nema konstruktore, prevodilac će umesto vas automatski napraviti podrazumevani konstruktor. Na primer:

```
//: inicijalizacija/PodrazumevaniKonstruktor.java

class Ptica {}

public class PodrazumevaniKonstruktor {
  public static void main(String[] args) {
    Ptica p = new Ptica(); // Podrazumevani!
  }
} ///:~
```

Izraz

```
new Ptica()
```

pravi nov objekat i poziva podrazumevani konstruktor, iako on ranije nije izričito naveden. Bez njega ne bi postojala nijedna metoda koju bismo mogli da pozovemo kako bismo napravili objekat. Međutim, ako definišete makar jedan konstruktor te klase (sa argumentima ili bez njih), prevodilac *neće* umesto vas napraviti podrazumevani konstruktor:

```
class Ptica2 {
  Ptica2 (int i) { }
  Ptica2 (double d) { }
}
public class NemaSinteze {
  public static void main(String[] args) {
    //! Ptica2 p = new Ptica2(); // Nema podrazumevanog konstruktora
    Ptica2 p2 = new Ptica2(1);
    Ptica2 p3 = new Ptica2(1.0);
  }
} ///:~
```

Ako sada napišete:

```
new Ptica2();
```

prevodilac će se žaliti da ne može da nađe odgovarajući konstruktor. To je kao da niste napravili nijedan konstruktor, pa prevodilac kaže: „Morate da imate *neki* konstruktor, stoga pustite mene da vam napravim jedan." Ali, ako napravite bilo kakav konstruktor, prevodilac kaže: „Napisali ste konstruktor, znači znate šta radite; ako niste stavili podrazumevan, namerno ste hteli da ga izostavite."

Vežba 3: (1) Napravite klasu s podrazumevanim konstruktorom (onim koji ne prima argumente) koji ispisuje neku poruku. Napravite objekat te klase.

Vežba 4: (1) Prethodnoj vežbi dodajte preklopljeni konstruktor koji prima jedan **String** argument i ispisuje ga uz vašu poruku.

Vežba 5: (2) Napravite klasu **Pas** s preklopljenom metodom **laje()**. Metoda treba da je preklopljena na osnovu raznih prostih tipova podataka i da ispisuje različite vrste lajanja, zavijanja itd. u zavisnosti od toga koja je preklopljena verzija pozvana. Napišite metodu **main()** koja poziva sve te verzije.

Vežba 6: (1) Prepravite prethodni program tako da dve od preklopljenih metoda primaju po dva ista argumenta (različitih tipova), ali obrnutim redosledom. Uverite se da to radi.

Vežba 7: (1) Napravite klasu bez konstruktora i zatim u metodi **main()** napravite objekat te klase, kako biste se uverili da se podrazumevani konstruktor automatski sintetizuje.

Rezervisana reč **this**

Ako imate dva objekta istog tipa, koji se zovu **a** i **b**, možda se pitate kako možete da pozovete metodu **oljusti()** za oba objekta:

```
//:  inicijalizacija/BananuOljusti.java
class Banana { void oljusti(int i) { /* ... */ } }

public class BananuOljusti {
  public static void main(String[] args) {
      Banana a = new Banana(),
             b = new Banana();
      a.oljusti(1);
      b.oljusti(2);
   }
} ///:~
```

Ako postoji samo jedna metoda pod imenom **oljusti()**, kako će ona znati da li je pozvana za objekat **a** ili **b**?

Da biste program napisali u praktičnoj objektno orijentisanoj sintaksi, pri čemu „šaljete poruku objektu", prevodilac obavlja izvestan prikriveni posao. Postoji skriveni prvi argument koji se prosleđuje metodi **oljusti()** i taj argument je referenca na objekat kojim se rukuje. Dva prethodna poziva metode postaju otprilike:

```
Banana.oljusti(a,1);
Banana.oljusti(b,2);
```

Ovo su samo interni oblici i ako ih napišete, prevodilac ih ne prihvata, ali vam daje ideju o tome šta se dešava.

Pretpostavimo da se nalazite unutar metode i da želite da koristite referencu na tekući objekat. Pošto je tu referencu prevodilac *skriveno* prosledio, za nju ne postoji identifikator. Za tu namenu, međutim, postoji rezervisana reč: **this**. Rezervisana reč **this** – koja može da se koristi samo unutar ne-**static** metode – vraća referencu na objekat za koji je metoda pozvana. Tu referencu možete da koristite kao i svaku drugu referencu na objekat. Vodite

računa o sledećem: ako pozivate metodu svoje klase iz neke druge metode te iste klase, ne treba da koristite **this**; samo pozovite metodu. Tekuća referenca **this** automatski se koristi i za drugu metodu. Znači, možete da napišete:

```
//:  inicijalizacija/Kajsija.java
public class Kajsija {
  void otrebi() { /* ... */ }
  void kostica() { otrebi(); /* ... */ }
} ///:~
```

Unutar metode **kostica**() *mogli* biste da napišete **this.otrebi**(), ali to nije potrebno.[1] Prevodilac to automatski radi umesto vas. Rezervisana reč **this** koristi se samo kada treba da izričito koristite referencu na tekući objekat. Na primer, ona se često koristi uz naredbe **return** kada treba da vratite referencu na tekući objekat:

```
//: inicijalizacija/Lista.java
// Jednostavna upotreba rezervisane reči "this".

public class Lista {
  int i = 0;
  Lista prelistaj() {
    i++;
    return this;
  }
  void ispis() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Lista x = new Lista();
    x.prelistaj().prelistaj().prelistaj().ispis();
  }
} /* Ispis:
i = 3
*///:~
```

Pošto metoda **prelistaj**() vraća referencu na tekući objekat preko rezervisane reči **this**, lako se može izvršiti više operacija nad istim objektom.

this se koristi i za prosleđivanje tekućeg objekta drugoj metodi:

```
//: inicijalizacija/ProslediPomocuThis.java

class Osoba {
  public void pojedi(Jabuka jabuka) {
```

[1] Ima ljudi koji kao opsednuti pišu **this** ispred svakog poziva metode i svake reference polja, obrazlažući da je takav kôd „jasniji i eksplicitniji". Ne činite to. Zna se zbog čega koristimo jezike visokog nivoa: oni mnogo toga rade umesto nas. Ako budete pisali **this** kada nije neophodno, zbunićete i naljutiti sve koji će čitati vaš program, pošto u svem ostalom kodu koji su pročitali **this** *nije* bio posejan posvuda. Ljudi očekuju da se **this** upotrebljava samo onda kada je neophodan. Pridržavajući se doslednog i jednostavnog stila programiranje štedite vreme i novac.

```
    Jabuka oljustena = jabuka.oljustiSe();
    System.out.println("Mljac");
  }
}

class ljustac {
  static Jabuka ljusti(Jabuka jabuka) {
    // ... ukloni ljusku
    return jabuka; // Oljustena
  }
}

class Jabuka {
  Jabuka oljustiSe() { return ljustac.ljusti(this); }
}

public class ProsledjiPomoćuThis {
  public static void main(String[] args) {
    new Osoba().pojedi(new Jabuka());
  }
} /* Ispis:
Mljac
*///:~
```

Jabuka zove **ljustac.ljusti()**, spoljnu metodu koja obavlja operaciju koja, iz nekog razloga, mora biti spoljna za objekat **Jabuka** (možda se spoljna metoda može primeniti na više različitih klasa, a vi ne želite da ponavljate njen kôd). Da bi se prosledila spoljnoj metodi, ona mora da upotrebi **this**.

Vežba 8: (1) Napravite klasu s dve metode. Unutar prve metode, dvaput pozovite drugu: prvi put bez **this**, a drugi put pomoću **this** – tek da biste videli kako radi; to ne bi trebalo da radite u praksi.

Pozivanje konstruktorâ iz konstruktorâ

Kada za klasu napišete nekoliko konstruktora, možete pozvati jedan konstruktor iz drugog da biste izbegli pisanje istih naredaba više puta. Takav poziv možete obaviti pomoću rezervisane reči **this**.

Kada napišete **this**, obično mislite na „ovaj objekat" odnosno na „tekući objekat", što samo po sebi daje referencu na tekući objekat. Ako je upotrebite s listom argumenata unutar konstruktora, rezervisana reč **this** ima drugačije značenje: ona eksplicitno poziva preklopljeni konstruktor sa odgovarajućom listom argumenata. Stoga druge konstruktore možete smesta da pozivate:

```
//: inicijalizacija/Cvet.java
// Pozivanje konstruktora pomoću rezervisane reči "this".
import static net.mindview.util.Print.*;
```

```
public class Cvet {
  int brojLatica = 0;
  String s = "početni broj";
  Cvet(int latice) {
    brojLatica = latice;
    print("Konstruktor samo sa argumentom tipa int, brojLatica= "
      + brojLatica);
  }
  Cvet(String ss) {
    print("Konstruktor samo sa argumentom tipa String, s = " + ss);
    s = ss;
  }
  Cvet(String s, int latice) {
    this(latice);
  //!    this(s); // Ne možete da pozovete dva konstruktora!
    this.s = s; // Još jedna upotreba rezervisane reči "this"
    print("Argumenti tipa String i int");
  }
  Cvet() {
    this("zdravo", 47);
    print("podrazumevani konstruktor (bez argumenata)");
  }
  void ispisBrojaLatica() {
//!    this(11); // Ne može izvan konstruktora!
    print("brojLatica = " + brojLatica + " s = "+ s);
  }
  public static void main(String[] args) {
    Cvet x = new Cvet();
    x.ispisBrojaLatica();
  }
} /* Ispis:
Konstruktor samo sa argumentom tipa int, brojLatica= 47
Argumenti tipa String i int
podrazumevani konstruktor (bez argumenata)
brojLatica = 47 s = zdravo
*///:~
```

Konstruktor **Cvet(String s, int latice)** pokazuje da pomoću rezervisane reči **this** možete pozvati jedan konstruktor ali ne i dva. Takođe, konstruktor morate da pozovete pre svih operacija – u suprotnom, biće prijavljena greška pri prevođenju.

Prethodni primer takođe pokazuje još jedan način upotrebe rezervisane reči **this**. Pošto su imena argumenta **s** i podatka člana **s** ista, to je dvosmisleno. Rešenje je da se podatku članu obraćate sa **this.s**. Takav oblik ćete često sretati u programima, kao i na brojnim mestima u ovoj knjizi.

U metodi **ispisBrojaLatica()** možete videti da prevodilac neće dozvoliti da pozovete konstruktor iz bilo koje druge metode osim iz drugog konstruktora.

Vežba 9: (1) Napravite klasu s dva (preklopljena) konstruktora. Iz prvog konstruktora pomoću **this** pozovite drugi.

Značenje rezervisane reči **static**

Pošto ste upoznali rezervisanu reč **this**, možete potpunije da shvatite šta znači kada neku metodu proglasite statičnom – za tu metodu ne postoji referenca **this**. Iz statičnih metoda ne možete da pozovete nestatične[2] metode (iako je obratno moguće). Statičnu metodu takođe možete da pozovete za samu klasu, bez objekta. To i jeste osnovna namena statične metode. To je kao da pravite ekvivalent globalne metode. Međutim, globalne metode nisu dozvoljene u Javi. Kada u klasu stavite statičnu metodu, ona može da pristupa drugim statičnim metodama i statičnim poljima.

Postoji mišljenje da statične metode nisu objektno orijentisane pošto imaju semantiku globalne metode; u statičnim metodama vi ne šaljete poruku objektu, jer ne postoji referenca **this**. To jeste razuman argument i ako primetite da *mnogo* koristite statične metode, verovatno bi trebalo da ponovo razmotrite svoj projekat. Ipak, statične metode i polja su pragmatični i ponekad zaista potrebni, pa razmatranje jesu li ili nisu „ispravno objektno orijentisani" treba prepustiti teoretičarima.

Čišćenje: finalizacija i sakupljanje smeća

Programeri su svesni značaja inicijalizacije, ali često zaboravljaju na važnost čišćenja. Naposletku, kome treba da čisti memoriju koju zauzima promenljiva tipa **int**? Ali u bibliotekama, jednostavno „napuštanje" objekta kada s njim završite nije uvek bezbedno. Naravno, Java ima sakupljač smeća koji oslobađa memoriju objekata koji se više ne koriste. Razmotrimo jedan neobičan slučaj. Pretpostavimo da je vaš objekat zauzeo „posebnu" memoriju bez upotrebe operatora **new**. Sakupljač smeća zna samo kako da oslobodi memoriju dodeljenu *pomoću* **new** i neće znati kako da oslobodi „posebnu" memoriju tog objekta. Da bi se rešio taj slučaj, u Javi je obezbeđena metoda **finalize**() koju možete da definišete u svojoj klasi. Evo kako bi ona *trebalo* da radi. Kada je sakupljač smeća spreman da oslobodi prostor koji zauzima vaš objekat, prvo će pozvati metodu **finalize**() i tek u drugom prolasku sakupljanja smeća oslobodiće i memoriju objekta. Znači, ako koristite metodu **finalize**(), ona vam omogućuje da nešto važno očistite *u trenutku sakupljanja smeća*.

Ovde se krije potencijalan izvor grešaka, jer neki programeri, posebno oni vični C++-u, u početku mogu da pomešaju metodu **finalize**() s *destruktorom* u C++-u, što je funkcija koja se uvek poziva kada se objekat uništava. Ovde je važno da napravite razliku između C++-a i Jave, jer se u C++-u *objekti uvek uništavaju* (u programu bez grešaka), dok u Javi sakupljač smeća ne počisti uvek objekte. Drugim rečima:

1. Može se desiti da sakupljač smeća ne počisti vaš objekat.
2. Sakupljanje smeća nije uništavanje.

Ako to zapamtite, nećete imati nevolje. To znači sledeće: ukoliko imate nešto da uradite pre nego što odbacite objekat, to morate uraditi ručno. Java nema destruktore niti bilo šta slično, pa morate da napravite uobičajenu metodu koja će obaviti čišćenje. Na

[2] To je moguće samo kada toj **statičnoj** metodi prosledite referencu na objekat (i statična metoda može praviti sopstvene objekte). Tada preko te reference (koja je u tom slučaju igra ulogu reference **this**) možete da pozivate **nestatične** metode i da pristupate **nestatičnim** poljima. Ipak, kada želite nešto tako da radite, napravite samo običnu **nestatičnu** metodu.

primer, pretpostavimo da se tokom stvaranja vaš objekat iscrta na ekranu. Ako eksplicitno ne obrišete njegovu sliku sa ekrana, možda ona nikad neće biti uklonjena. Ukoliko metodu **finalize()** programirate za neku vrstu brisanja, a objekat dospe pod dejstvo sakupljača smeća i **finalize()** bude pozvana (a niko ne jamči da će se to desiti), u prvoj fazi čišćenja slika će biti uklonjena sa ekrana. U suprotnom, ako sakupljač smeća ne reši da počisti vaš objekat, slika ostaje na ekranu.

Može se dogoditi da prostor koji zauzima vaš objekat nikad ne bude oslobođen, jer vašem programu nikada neće ponestati memorije. Ako se program završi, a sakupljač smeća nije imao priliku da oslobodi prostor koji je bilo koji vaš objekat zauzimao, celokupan prostor će biti *odjednom* vraćen operativnom sistemu na izlasku iz programa. Tako se izbegava dodatna upotreba računarskih resursa – ako čišćenje nije potrebno pa se poboljšavaju performanse programa.

Čemu služi metoda **finalize()**?

Dakle, ako metodu **finalize()** ne treba da koristite kao metodu čišćenja opšte namene, čemu, onda, ona služi?

Treća stvar koju treba da zapamtite glasi:

3. Sakupljanje smeća se bavi samo memorijom.

Drugim rečima, sakupljač smeća postoji samo da bi oslobađao memoriju koju vaš program više ne koristi. Zbog toga svaka aktivnost koja je povezana sa sakupljanjem smeća, posebno metoda **finalize()**, mora takođe da se bavi samo memorijom i njenim oslobađanjem.

Da li to znači da metoda **finalize()** treba eksplicitno da oslobađa objekte ukoliko ih sadrži vaš objekat? Ne, jer o oslobađanju memorije objekata vodi računa sakupljač smeća, i to bez obzira na to kako su objekti napravljeni. Metoda **finalize()** je korisna samo u posebnim slučajevima, kada vaš objekat zauzima memoriju na neki drugi način, ne pravljenjem objekata. Ali, kao što znate, u Javi je sve objekat, pa kako je, onda, taj slučaj mogući?

Ispostavlja se da metoda **finalize()** postoji zbog mogućnosti da memoriju zauzmete koristeći neki mehanizam sličan C-ovom, umesto na uobičajen način. To se prvenstveno može desiti pri pozivu *lokalnih metoda* (engl. *native methods*) koje služe da iz Jave pozovete neki drugi lokalni kod. (Lokalne metode se razmatraju u dodatku B u elektronskom 2. izdanju ove knjige, dostupnom na lokaciji *www.MindView.net*.) Od jezika za pisanje lokalnih metoda podržani su samo C i C++, ali pošto u njima možete da pozovete potprogram na nekom drugom jeziku, zapravo možete da pozovete bilo šta. Unutar lokalnog koda, memoriju možete zauzeti nekom funkcijom iz C-ove porodice funkcija **malloc()**. Ako ne pozovete i odgovarajuću funkciju **free()**, prostor neće biti oslobođen i javiće se „curenje memorije" (engl. *memory leak*). Naravno, **free()** je funkcija jezika C i C++, pa morate da je pozovete preko neke lokalne metode iz vaše metode **finalize()**.

Pošto ste sve ovo pročitali, verovatno vam je jasno da nećete često koristiti metodu **finalize()**[3]. To je zaista tako: ona nije prikladno mesto na kome se radi uobičajeno čišćenje. A gde bi, onda, ono trebalo da se radi?

---

[3] Joshua Bloch ide još dalje u odeljku pod naslovom „Izbegavajte korišćenje završnih metoda": „Finalizatori su nepredvidljivi, često opasni, i po pravilu nepotrebni." *Efikasno programiranje na Javi*, s. 18 (Mikro knjiga, 2004).

Morate da čistite sami

Da bi počistio objekat, korisnik mora da pozove metodu za čišćenje u pogodnom trenutku. To zvuči prilično jasno, ali se pomalo sudara s konceptom destruktora u C++-u. U C++-u svi objekti se uništavaju, to jest, svi objekti *bi trebalo* da budu uništeni. Ukoliko je objekat u C++-u napravljen lokalno (na primer na steku – što nije moguće u Javi), uništava se kada se naiđe na zatvorenu vitičastu zagradu koja završava oblast važenja objekta. Ako je objekat napravljen pomoću operatora **new** (kao u Javi), destruktor se poziva kada programer pozove operator **delete** (koji ne postoji u Javi). Ako programer na C++-u zaboravi da pozove **delete**, destruktor neće biti nikada pozvan i nastaće „curenje memorije", a pored toga ni drugi delovi objekta neće biti počišćeni. Ova vrsta grešaka otkriva se vrlo teško i jedan je od ubedljivih razloga za prelazak sa C++-a na Javu.

Suprotno od toga, Java ne dozvoljava da pravite lokalne objekte – uvek morate da koristite operator **new**. Međutim, u Javi ne postoji operator **delete** koji pozivate da biste oslobodili objekat, jer sakupljač smeća to radi umesto vas. Pojednostavljeno gledano, možete reći da Java nema destruktore zato što ima sakupljače smeća. Kako ova knjiga odmiče, videćete da postojanje sakupljača smeća ne eliminiše potrebu za destruktorima i njihovim korišćenjem. (A nikada ne treba direktno da pozivate **finalize**(), pa to, dakle, nije rešenje problema.) Ako pored oslobađanja memorije želite i neko drugo čišćenje, *i dalje* morate eksplicitno da pozovete odgovarajuću metodu u Javi koja je ekvivalent destruktora u C++-u.

Zapamtite da se ne garantuje ni sakupljanje smeća niti čišćenje. Ako JVM-u ne ponestane memorije, on (mudro) neće gubiti vreme na oslobađanje memorije sakupljanjem smeća.

Stanje okončanja

Uglavnom ne možete da se oslonite na to da će **finalize**() biti pozvana, već morate da napravite zasebne funkcije za čišćenje i da ih eksplicitno pozovete. Zato se čini da je **finalize**() jedino korisna za oslobađanje memorije rezervisane na neuobičajen način, koju većina programera nikada neće koristiti. Postoji, međutim, jedna vrlo korisna primena metode **finalize**() koja se ne oslanja na to da će ona biti svaki put pozvana. To je provera postojanja *stanja okončanja*[1] (engl. *termination condition*) nekog objekta.

Od trenutka kada vas objekat više ne zanima – kada je spreman da bude počišćen – taj objekat treba da bude u stanju u kome njegova memorija može bezbedno da bude oslobođena. Na primer, ako objekat predstavlja otvorenu datoteku, programer treba da zatvori tu datoteku pre nego što sakupljač smeća počisti taj objekat. Ako bilo koji deo objekta nije pravilno počišćen, imaćete grešku u programu koju je vrlo teško otkriti. Metoda **finalize**() je značajna zato što može da se iskoristi za otkrivanje takvih slučajeva i ako se ne poziva uvek. Ako neka finalizacija otkrije grešku, otkrili ste problem, što vas, uostalom, jedino i brine.

[1] Izraz koji je skovao Bill Venners (www.artima.com) tokom našeg seminara.

Evo jednostavnog primera kako da to iskoristite:

```
//: inicijalizacija/StanjeOkoncanja.java
// Otkrivanje objekta koji nije ispravno počišćen
// pomoću metode finalize().

class Knjiga {
  boolean pozajmljena = false;
  Knjiga(boolean p2) {
    pozajmljena = p2;
  }
  void vrati() {
    pozajmljena = false;
  }
  protected void finalize() {
    if(pozajmljena)
      System.out.println("Greška: knjiga nije vraćena");
      // Normalno biste uradili i sledeće:
      // super.finalize();  // Poziv verzije osnovne klase
  }
}

public class StanjeOkoncanja {
  public static void main(String[] args) {
    Knjiga roman = new Knjiga(true);
    // Pravilno čišćenje:
    roman.vrati();
    // Ispusti referencu, zaboravi da počistiš
    new Knjiga(true);
    // Zahtevaj sakupljanje smeća i finalizaciju
    System.gc();
  }
} /* Ispis:
Greška: Knjiga nije vraćena
* ///:~
```

Sve pozajmljene knjige u stanju okončanja trebalo bi da budu vraćene pre nego što ih pokupi sakupljač smeća, ali se dogodi da su metodi **main**() programer greškom ne vrati jednu od knjiga. Bez metode **finalize**() koja proverava stanje okončanja, to bi predstavljalo grešku koju je potencijalno teško otkriti.

Obratite pažnju na to da je za zahtevanje čišćenja korišćen poziv **System.gc**(). Čak i da nije, vrlo je verovatno da bi zalutala **Knjiga** jednom bila otkrivena u ponovljenim izvršavanjima programa (pod pretpostavkom da program zauzima dovoljno memorije kako bi bio pokrenut sakupljač smeća).

Po pravilu, trebalo bi da pretpostavite kako i verzija **finalize**() u osnovnoj klasi ima važnog posla i da je pozovete pomoću rezervisane reči **super**, kao što vidite u **Knjiga.finalize**(). U ovom slučaju, taj red je pretvoren u komentar zato što zahteva obradu izuzetaka, a mi ih još nismo razmotrili.

Vežba 10: (2) Napravite klasu s metodom **finalize**() koja ispisuje neku poruku. U metodi **main**() napravite objekat te klase. Objasnite ponašanje svog programa.

Vežba 11: (4) Prepravite prethodnu vežbu tako da se vaša metoda **finalize**() uvek poziva.

Vežba 12: (4) Napravite klasu **Rezervoar** koja se može puniti i prazniti, i koja ima takvo *stanje okončanja* da mora biti prazna prilikom čišćenja objekta. Napišite metodu **finalize**() koja proverava ispunjenje tog uslova okončanja. U metodi **main**() ispitajte scenarije koji se mogu odigrati prilikom upotrebe klase **Rezervoar**.

Kako radi sakupljač smeća?

Ako ste prešli s programskog jezika u kome je za pravljenje objekta u dinamičkoj memoriji neophodno mnogo procesorske snage, možete pretpostaviti da je Javin princip pravljenja svega (osim prostih tipova) u dinamičkoj memoriji takođe zahtevan. Međutim, ispostavlja se da sakupljanje smeća može značajno uticati na *povećanje* brzine pravljenja objekata. To u početku zvuči pomalo čudno – oslobađanje prostora utiče na dodeljivanje prostora – ali na taj način rade neke virtuelne mašine za Javu i to znači da dodeljivanje prostora u dinamičkoj memoriji u Javi može da bude gotovo jednako brzo kao odvajanje prostora *na steku* u drugim programskim jezicima.

Na primer, možete zamisliti da je dinamička memorija u C++-u travnjak na kom se svaki objekat pojavljuje kao mali busen. Ovaj posed može kasnije da bude napušten i zatim ponovo korišćen. U nekim Javinim virtuelnim mašinama, dinamička memorija može da bude potpuno drugačija; više kao pokretna traka koja se pomera unapred svaki put kada napravite nov objekat. To znači da je dodeljivanje prostora veoma brzo. „Dinamički pokazivač" se pomeri unapred u nedirnutu teritoriju, što odgovara dodeli prostora sa steka u C++-u. (Naravno, ima malo dodatnih troškova za knjigovodstvo, ali to nije ni nalik traženju mesta za ostavu.)

Obratite pažnju na to da dinamička memorija nije pokretna traka i ako je na taj način tretirate, ubrzo ćete doći u situaciju da vam se program raširi na više memorijskih stranica – koje se snimaju na disk i vraćaju s njega u brzu memoriju, pa će izgledati da imate više memorije nego što je zapravo slučaj. Promena stranice značajno smanjuje performanse. Na kraju, nakon što napravite dovoljan broj objekata, ponestaće vam memorije. Tu na scenu stupa sakupljač smeća – dok radi, on istovremeno sabija sve objekte u dinamičkoj memoriji čime se „dinamički pokazivač" pomera bliže početku pokretne trake i dalje od kraja stranice. Sakupljač smeća reorganizuje stvari i omogućava da se za dodeljivanje prostora koristi model brze dinamičke memorije beskonačne dužine.

Da biste shvatili kako to radi, treba da malo bolje razumete kako rade sakupljači smeća (engl. *garbage collector*, *gc*). Pri njihovom radu koristi se jednostavna ali spora tehnika – *brojanje referenci*. To znači da svaki objekat ima brojač referenci i da se taj brojač uveća za jedan svaki put kada se objektu pridruži neka referenca. Kad god referenca izađe iz oblasti važenja ili bude postavljena na vrednost **null**, brojač referenci se umanji za jedan. Prema tome, vođenje računa o brojaču referenci je mali, ali stalan režijski trošak koji se obavlja za vreme trajanja vašeg programa. Sakupljač smeća prolazi kroz celokupnu listu objekata i oslobađa memoriju kada naiđe na objekat čiji je brojač referenci nula. (Međutim, u realizacijama

brojača referenci objekat se često oslobađa čim brojač padne na nulu.) Jedna od mana ovog pristupa je sledeća: ako su objekti uzajamno povezani referencama, njihovi brojači referenci mogu biti različiti od nule, čak i ako ti objekti predstavljaju smeće. Otkrivanje takvih samoreferencirajućih grupa značajan je dodatni posao za sakupljač smeća. Brojanje referenci se često koristi da objasni jednu vrstu sakupljanja smeća, ali se ne primenjuje gotovo ni u jednoj stvarnoj virtuelnoj mašini.

U bržim realizacijama, sakupljanje smeća se ne zasniva na brojanju referenci. Umesto toga, ono se zasniva na ideji da za svaki živi objekat možemo na kraju pronaći neku referencu koja se nalazi bilo na steku ili u statičnom skladištu. Lanac veza može ići kroz nekoliko nivoa objekata. Znači, ako krenete od steka i statičnog skladišta, i prođete kroz sve reference koje se tamo nalaze, pronaći ćete sve žive objekte. Za svaku referencu koju pronađete, morate pristupiti objektu na koji ona ukazuje i zatim analizirati sve reference u *tom* objektu, dalje ka drugim objektima itd. Postupak se završava kada prođete kroz celu Mrežu koja je započela tom referencom na steku ili statičnom skladištu. Svaki objekat kroz koji prođete sigurno je živ. Obratite pažnju na to da nema problema sa izdvojenim samoreferencirajućim grupama, jer njih na ovaj način ne nalazite i stoga automatski postaju smeće.

U gore opisanom pristupu, JVM koristi *prilagodljiv* (engl. *adaptive*) način sakupljanja smeća. Šta posle radi sa pronađenim živim objektima, zavisi od varijante koja se trenutno koristi. Jedna od tih varijanata je *stani-i-kopiraj* (engl. *stop--and-copy*). To znači da se – iz razloga koji će ubrzo postati jasni – program prvo zaustavi (ovo nije sakupljanje u pozadini). Zatim se svaki živi objekat kopira iz jednog dinamičkog memorijskog prostora u drugi, pri čemu u starom ostaje sve smeće. Pored toga, objekti se pri kopiranju u novi dinamički memorijski prostor slažu jedan za drugim, čime se on sabija (i omogućava da se novi skladišni prostor samo nastavi na kraj, kao što je ranije opisano).

Kada se objekat pomera s jednog mesta na drugo, razume se da sve reference koje na njega ukazuju moraju da budu izmenjene. Referenca koja potiče iz dinamičkog memorijskog prostora ili statičkog skladišta može biti odmah promenjena, ali mogu postojati i druge reference koje ukazuju na taj objekat na koje ćemo naići kasnije. One se sređuju kako se na njih nailazi (zamislite tabelu koja prevodi stare adrese u nove).

Postoje dva činioca koja čine neefikasnim ove takozvane sakupljače s kopiranjem (engl. *copy collectors*). Prvi je da imate dva dinamička memorijska prostora i da prebacujete memoriju između ta dva odvojena prostora, trošeći dvostruko više memorije nego što vam zaista treba. Neke virtuelne mašine ovo rešavaju tako što zauzimaju dinamički memorijski prostor u komadićima, po potrebi, i kopiraju iz jednog komadića u drugi.

Drugo pitanje je sam proces kopiranja. Kada vaš program postane stabilan, praviće malo ili nimalo smeća. Uprkos tome, sakupljač s kopiranjem će i dalje kopirati memoriju s jednog mesta na drugo, što je gubljenje vremena. Da bi to sprečile, neke virtuelne mašine otkrivaju kada se više ne pravi smeće i prelaze na drugačije čišćenje (ovo je „prilagodljiv" deo). Drugi pristup se naziva *označi-i-počisti* (engl. *mark and sweep*) i njega su starije virtuelne mašine kompanije Sun koristile sve vreme. Za opštu upotrebu, tehnika označi-i-počisti je prilično spora, ali kada znate da imate malo ili nimalo dubreta, ona je brza.

Pri tehnici označi-i-počisti koristi se ista logika polaženja od steka i statičkog skladišta i praćenja svih referenci da bi se pronašli živi objekti. Svaki put kada se naiđe na živi objekat, on se označi indikatorom, ali se objekat još uvek ne sakuplja. Čišćenje se obavlja tek

kada se završi postupak označavanja. Tokom čišćenja oslobađa se memorija koju su zauzimali mrtvi objekti. Međutim, ništa se ne kopira, pa ako sakupljač odluči da sabije fragmentirani dinamički memorijski prostor, on to radi tako što pomera objekte.

Tehnika stani-i-kopiraj počiva na ideji da ovaj tip sakupljanja smeća *ne* radi u pozadini, već se program zaustavlja dok se obavlja sakupljanje smeća. U literaturi koju je objavio Sun pronaći ćete više referenci na sakupljač smeća kao pozadinski proces niskog prioriteta, ali se ispostavilo da u ranijim verzijama Sunove JVM sakupljanje smeća nije realizovano na taj način. Umesto toga, Sunov sakupljač smeća je zaustavljao program kada ponestane memorije. I tehnika označi-i--počisti zahteva zaustavljanje programa.

Kao što je ranije pomenuto, opisana virtuelna mašina zauzima memoriju u velikim blokovima. Ako napravite veliki objekat, on dobija sopstveni blok. Striktna tehnika stani-i-kopiraj zahteva kopiranje svih živih objekata sa izvorišnog dinamičkog memorijskog prostora u novi pre nego što se stari oslobodi, za šta je neophodno puno memorije. Sakupljač smeća tokom sakupljanja obično može da kopira objekte u mrtve blokove. Svaki blok ima *brojač proizvodnje* koji vodi evidenciju da li je blok živ. Obično se sabijanje vrši samo nad blokovima koji su napravljeni posle poslednjeg sakupljanja smeća; svim ostalim blokovima brojač proizvodnje se uvećava ako su odnekud referencirani. Na ovaj način se brine o uobičajenim slučajevima privremenih objekata kratkog životnog veka. Periodično se radi potpuno čišćenje – veliki objekti se i dalje ne kopiraju (već im se samo uveća brojač proizvodnje), a blokovi koji sadrže male objekte kopiraju se i slažu. JVM prati efikasnost sakupljača smeća i ako se za njegovo korišćenje neophodno troši vreme jer su svi objekti dugog životnog veka, prebacuje ga u režim označi-i-počisti. Slično tome, JVM vodi računa o uspešnosti tehnike označi-i-počisti, i ako dinamički memorijski prostor postane fragmentiran, prebacuje se nazad u režim stani-i-kopiraj. Prethodna metoda prebacivanja predstavlja pomenuti prilagodljivi deo, tj. postoji tehnika prilagodljiva stani-i-kopiraj označi-i-počisti.

Postoji više dodatnih mogućnosti za ubrzavanje virtuelne mašine. Posebno važna mogućnost odnosi se na rad programa za učitavanje (engl. *loader*) i onoga što se naziva prevodilac baš-kada-treba (engl. *Just-In-Time, JIT*). JIT prevodilac delimično ili potpuno pretvara program u lokalni mašinski kôd, tako da JVM ne mora da ga interpretira, pa je izvršavanje mnogo brže. Kada klasa mora da se učita (obično, prvi put kada želite da napravite objekat te klase), pronalazi se datoteka **.class** i bajtkod te klase se učitava u memoriju. U tom trenutku jedan pristup može biti da primenimo JIT prevođenje na sav kôd, ali to ima dve mane. Prvo, nešto više traje (što, kada se nakupi kroz životni vek programa, ne može da bude zanemarljivo). Drugo, povećava se izvršni program (bajtkod je znatno kompaktniji od raspakovanog JIT koda), pa mogu da se izmene memorijske stranice, što definitivno usporava program. Alternativni pristup je lenja procena (engl. *lazy evaluation*), što znači da se JIT prevođenje ne radi sve dok nije neophodno. Time se postiže da kôd koji se nikada ne izvrši, možda nikada ne bude JIT preveden. Tehnologije Java HotSpot u novijim razvojnim okruženjima (JDK) postupaju slično, jer optimizuju svaki deo koda svaki put kada se on izvršava – što se kôd više izvršava, postaje sve brži.

Inicijalizacija članica

Java se baš trudi da garantuje kako će sve promenljive biti pravilno inicijalizovane pre korišćenja. Kad su promenljive definisane lokalno u metodi, ovo garantuje prevodilac (jer inače prijavljuje grešku). Znači, ako napišete:

```
void f() {
  int i;
  i++;  // Greška -- i nije inicijalizovana
}
```

dobićete poruku o grešci s napomenom da promenljiva **i** možda nije inicijalizovana. Naravno, prevodilac je mogao da dodeli promenljivoj **i** podrazumevanu vrednost, ali je neinicijalizovana lokalna promenljiva verovatno greška programera, a podrazumevana vrednost bi to prikrila. Prisiljavanjem programera da obezbedi vrednosti za inicijalizaciju, povećava se verovatnoća otkrivanja grešaka.

Ako je prost tip polje neke klase, stvari su nešto drugačije. Kao što ste videli u poglavlju *Sve je objekat*, svako polje prostog tipa u svakoj klasi garantovano će dobiti inicijalnu vrednost. Evo primera koji to dokazuje i tih vrednosti:

```
//: inicijalizacija/InicijalneVrednosti.java
// Prikazuje podrazumevane inicijalne vrednosti.
import static net.mindview.util.Print.*;
public class InicijalneVrednosti {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  InicijalneVrednosti reference;
  void ispisInicijalneVrednosti() {
    print("Tip podatka      Inicijalna vrednost");
    print("boolean          " + t);
    print("char             [" + c + "] ");
    print("byte             " + b);
    print("short            " + s);
    print("int              " + i);
    print("long             " + l);
    print("float            " + f);
    print("double           " + d);
    print("reference        " + reference);
  }
  public static void main(String[] args) {
    InicijalneVrednosti iv = new InicijalneVrednosti();
```

```
    iv.ispisInicijalneVrednosti();
    /* U ovom slučaju takođe možete napisati:
    new InicijalneVrednosti().ispisInicijalneVrednosti();
    */
  }
} /* Ispis:
Tip podatka     Inicijalna vrednost
boolean         false
char            [ ]
byte            0
short           0
int             0
long            0
float           0.0
double          0.0
reference       null
* ///:~
```

Vidite da se vrednosti, iako nisu navedene, automatski inicijalizuju (podrazumevana vrednost za tip **char** je nula, što se ispisuje kao razmak.). Stoga je barem eliminisana opasnost od rada s neinicijalizovanim promenljivama.

Kada definišete referencu na objekat unutar neke klase, a ne inicijalizujete je, ta referenca dobija posebnu vrednost **null**.

Zadavanje inicijalizacije

Šta se dešava ako želite da promenljivoj dodelite početnu vrednost? Direktan način da to uradite jeste da joj dodelite vrednost na mestu gde tu promenljivu definišete u klasi. (To nije moguće u C++-u, iako početnici u jeziku C++ to uvek pokušavaju.) U sledećem primeru definicije polja u klasi **InicijalneVrednosti** promenjene su da bi obezbedile početne vrednosti:

```
//: inicijalizacija/InicijalneVrednosti2.java
// Eksplicitno zadavanje inicijalnih vrednosti.

public class InicijalneVrednosti2 {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
} ///:~
```

Objekte možete inicijalizovati na isti način. Ako je definisana klasa **Dubina**, definišite promenljivu i inicijalizujte je na sledeći način:

```
//: inicijalizacija/Mera.java
class Dubina  {}

public class Mera {
  Dubina d = new Dubina ();
  // ...
}  ///:~
```

Ako promenljivoj **d** niste dali početnu vrednost, a ipak pokušate da je koristite. dobićete grešku pri izvršavanju koja se u Javi naziva izuzetak (engl. *exception*), što je obrađeno u poglavlju *Obrada grešaka pomoću izuzetaka.*

Možete čak da pozovete metodu da biste obezbedili vrednost za inicijalizaciju:

```
//: inicijalizacija/InicijalizacijaMetodom.java
public class InicijalizacijaMetodom {
  int i = f();
  int f() { return 11; };
}  ///:~
```

Naravno, ta metoda može imati argumente, ali ti argumenti ne mogu biti ostali članovi klase koji još nisu inicijalizovani. Stoga ovo možete da uradite:

```
//: inicijalizacija/InicijalizacijaMetodom2.java
public class InicijalizacijaMetodom2 {
  int i = f();
  int j = g(i);
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

Ali ovo ne možete da uradite:

```
//: inicijalizacija/InicijalizacijaMetodom3.java
public class InicijalizacijaMetodom3 {
  //! int j = g(i); // Nedozvoljeno referenciranje unapred
  int i = f();
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

Na ovom mestu se prevodilac, sasvim opravdano, *buni* zbog referenciranja unapred. Razlog leži u redosledu inicijalizacije (promenljiva **i** se koristi pre nego što je definisana), a ne u načinu na koji se program prevodi.

Ovakav pristup inicijalizaciji jednostavan je i direktan. Ograničen je time što *svaki* objekat tipa `InicijalneVrednosti` ima iste inicijalizacione vrednosti. Ponekad vam baš to treba, ali drugi put će vam zatrebati više fleksibilnosti.

Inicijalizacija konstruktorima

Konstruktor može da se koristi za inicijalizaciju. To daje veću fleksibilnost u programiranju, jer možete da pozivate metode i izvršavate akcije tokom rada programa kako biste odredili početne vrednosti. Jednu stvar treba imati na umu: time ne sprečavate automatsku inicijalizaciju koja se odigrava pre nego što se uđe u konstruktor. Stoga, ako biste, na primer, napisali:

```
//: inicijalizacija/Brojac.java
public class Brojac {
  int i;
  Brojac() { i = 7; }
  // ...
} ///:~
```

promenljiva **i** bi prvo bila inicijalizovana vrednošću 0, a zatim vrednošću 7. Ovo važi za sve proste tipove i reference na objekte, uključujući i one koji su eksplicitno inicijalizovani na mestu definicije. Prevodilac zato ne pokušava da vas prisili da inicijalizujete elemente na bilo kom mestu u konstruktoru ili pre nego što ih koristite – inicijalizacija je već sigurno obavljena.

Redosled inicijalizacije

Redosled inicijalizacije unutar klase određen je redosledom kojim su promenljive definisane. Definicije promenljivih mogu biti razbacane unutar i između definicija metoda, ali se promenljive inicijalizuju pre poziva bilo koje metode – čak i konstruktora. Na primer:

```
//: inicijalizacija/RedosledInicijalizacije.java
// Pokazuje redosled inicijalizacije.
import static net.mindview.util.Print.*;

// Kada se pozove konstruktor za pravljenje
// objekta klase Prozor, videćete poruku:
class Prozor {
  Prozor(int marker) {
      Prozor(int marker)  { print("Prozor(" + marker + ")");  }
}

class Kuca {
  Prozor p1 = new Prozor(1); // Pre konstruktora
  Kuca() {
    // Ukazuje da smo unutar konstruktora
    print('Kuca()");
    p3 = new Prozor(33); // Ponovo inicijalizuje p3
  }
  Prozor p2 = new Prozor(2); // Nakon konstruktora
  void f() {  print("f()");  }
  Prozor p3 = new Prozor(3); // Na kraju
}
```

```
public class RedosledInicijalizacije {
  public static void main(String[] args) {
    Kuca k = new Kuca();
    k.f(); // Prikazuje da je završeno pravljenje
  }
} /* Ispis:
Prozor(1)
Prozor(2)
Prozor(3)
Kuca()
Prozor(33)
f()
* ///:-
```

U klasi **Kuca**, definicije objekata klase **Prozor** namerno su razbacane da bi se dokazalo kako će sve biti inicijalizovane pre ulaska u konstruktor. Pored toga, **p3** se ponovo inicijalizuje unutar konstruktora.

Iz izlaza vidite da se referenca **p3** inicijalizuje dvaput, jednom pre i jednom za vreme poziva konstruktora. (Prvi objekat će biti napušten i kasnije sakupljen kao smeće.) Isprva ovo može da izgleda neefikasno, ali garantuje pravilnu inicijalizaciju. Šta bi se desilo ako bi bio definisan preklopljeni konstruktor koji *ne* inicijalizuje **p3**, a „podrazumevana" inicijalizacija za **p3** pri njenoj definiciji nije zadata?

Inicijalizacija statičnih elemenata

Za statične podatke postoji jedinstveno skladište, bez obzira na to koliko je objekata napravljeno. Rezervisanu reč **static** ne možete primeniti na lokalne promenljive, pa se ona primenjuje samo na polja. I kada je statično polje prostog tipa, a vi ga ne inicijalizujete, ono će dobiti standardnu vrednost. Referenca na objekat dobija podrazumevanu početnu vrednost **null**.

Ako inicijalizaciju želite da izvršite na mestu definicije, to će izgledati isto kao i kod nestatičnih podataka.

Evo primera iz kojega ćete videti u kom trenutku se inicijalizuje statično skladište:

```
//: inicijalizacija/StatickaInicijalizacija.java
// Zadavanje početnih vrednosti u definiciji klase.
import static net.mindview.util.Print.*;

class Cinija {
  Cinija(int marker) {
    print("Cinija(" + marker + ")");
  }
  void f1(int marker) {
    print("f1(" + marker + ")");
  }
}
```

```
class Sto {
  static Cinija cinija1 = new Cinija(1);
  Sto() {
    print("Sto()");
    cinija2.f(1);
  }
  void f2(int marker) {
    print("f2(" + marker + ")");
  }
  static Cinija cinija2 = new Cinija(2);
}

class Kredenac {
  Cinija cinija3 = new Cinija(3);
  static Cinija cinija4 = new Cinija(4);
  Kredenac() {
    print("Kredenac()");
    cinija4.f1(2);
  }
  void f3(int marker) {
    print("f3(" + marker + ")");
  }
  static Cinija cinija5 = new Cinija(5);
}

public class StatickaInicijalizacija {
  public static void main(String[] args) {
    print("Pravljenje novog objekta klase Kredenac() u metodi main");
    new Kredenac();
    print(" Pravljenje novog objekta klase Kredenac() u metodi main ");
    new Kredenac();
    sto.f2(1);
    kredenac.f3(1);
  }
  static Sto sto = new Sto();
  static Kredenac kredenac = new Kredenac();
} /* Ispis:
Cinija(1)
Cinija(2)
Sto()
f1(1)
Cinija(4)
Cinija(5)
Cinija(3)
Kredenac()
f1(2)
Pravljenje novog objekta klase Kredenac() u metodi main
Cinija(3)
Kredenac()
```

```
f1(2)
Pravljenje novog objekta klase Kredenac() u metodi main
Cinija(3)
Kredenac()
f1(2)
f2(1)
f3(1)
* ///:-
```

Klasa **Cinija** omogućava da pratite pravljenje objekata, a klase **Sto** i **Kredenac** na više mesta unutar definicija imaju statične članove klase **Cinija**. Obratite pažnju na to da **Kredenac** pre statičnih definicija pravi nestatičan objekat **Cinija cinija3**.

Iz rezultata prethodnog programa vidite da se inicijalizacija statičnih elemenata javlja samo ako je neophodna. Ako ne napravite objekat klase **Sto** i ako se nikada ne obratite metodama **Sto.cinija1** ili **Sto.cinija2**, statični objekti **Cinija cinija1** i **cinija2** nikada neće biti napravljeni. Oni se inicijalizuju samo kada se pravi *prvi* objekat klase **Sto** (ili kada se prvi put pristupa statičnom podatku). Nakon toga, više se ne obavlja ponovna inicijalizacija statičnih objekata.

Prvo se inicijalizuju statični elementi, ako to već nije učinjeno, a zatim nestatični. Dokaz za to vidite u rezultatu prethodnog programa. Da bi se izvršila statična metoda **main( )**, mora biti učitana klasa **StaticInitialization**, a zatim se inicijalizuju njena statična polja **sto** i **kredenac**, što prouzrokuje učitavanje *tih* klasa; pošto obe te klase sadrže statične objekte **Cinija**, zatim se učitava **Cinija**. Tako se sve klase ovog programa učitavaju pre početka izvršavanja metode **main( )**. To najčešće nije slučaj, pošto u tipičnom programu sve neće biti povezano statičnim elementima kao u ovom primeru.

Da bismo napravili sažetak procesa pravljenja objekta, posmatrajmo klasu pod imenom **Pas**:

1. Iako se u njoj rezervisana reč **static** ne upotrebljava eksplicitno, konstruktor jeste statična metoda. Stoga, pri prvom pravljenju objekta tipa **Pas** *ili* kada se prvi put pristupa statičkoj metodi ili statičkom polju klase **Pas**, Javin interpreter mora da pronađe datoteku **Pas.class**, što čini pretraživanjem putanje klasa (engl. *classpath*).
2. Pri učitavanju **Pas.class** (pri čemu se pravi objekat klase **Class**, o čemu će kasnije biti reči) pokreće se inicijalizacija svih statičnih elemenata. Znači, oni se inicijalizuju samo jedanput, kada se odgovarajući objekat klase **Class** učitava po prvi put.
3. Kada napravite objekat operacijom **new Pas()**, proces konstrukcije objekta nalaže da se prvo zauzme dovoljno skladišnog prostora u dinamičkom memorijskom prostoru.
4. Taj prostor se popuni nulama, čime se automatski zadaje podrazumevana vrednost svih promenljivih prostog tipa u tom objektu **Pas** (nula za brojeve i njen ekvivalent za **boolean** i **char**), a sve reference dobijaju vrednost **null**.
5. Izvršavaju se sve inicijalizacije koje se javljaju na mestu definicije polja.
6. Izvršavaju se konstruktori. Kao što ćete videti u poglavlju *Ponovno korišćenje klasa*, to može da bude veliki posao, naročito pri nasleđivanju.

Eksplicitna inicijalizacija statičnih elemenata

Java omogućava da statičke inicijalizacije unutar klase grupišete posebnim „statičkim odredbama" (koje se ponekad nazivaju i *statički blokovi*). To ovako izgleda:

```
//inicijalizacija/Kasika.java
public class Kasika {
  static int i;
  static {
  i = 47;
  }
} ///:~
```

Ovo podseća na metodu, ali je to samo rezervisana reč **static** posle koje sledi blok koda. Ovaj kôd se, kao i druge inicijalizacije statičnih elemenata, izvršava samo jednom, kada prvi put pravite objekat te klase *ili* kada prvi put pristupate statičnom članu klase (čak i ako nikada ne napravite objekat te klase). Na primer:

```
//: inicijalizacija/IzricitoStaticka.java
// Eksplicitna inicijalizacija statičnih elemenata preko statičnog
bloka.
import static net.mindview.util.Print.*;
class Soljica {
  Soljica(int marker) {
    print("Soljica(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

class Soljice {
  static Soljica soljica1;
  static Soljica soljica2;
  static {
    soljica1 = new Soljica(1);
    soljica2 = new Soljica(2);
  }
  Soljice() {
    print("Soljice()");
  }
}

public class IzricitoStaticka {
  public static void main(String[] args) {
    print("Unutar metode main()");
    Soljice.soljica1.f(99);  // (1)
  }
  // static Soljice soljice1 = new Soljice();  // (2)
  // static Soljice soljice2 = new Soljice();  // (2)
```

```
} /* Ispis:
Unutar metode main()
Soljica(1)
Soljica(2)
f(99)
* ///:~
```

Statični elementi klase **Soljice** inicijalizuju se bilo da se statičnom objektu **soljica1** pristupa u redu označenom sa (1), bilo da je red (1) komentarisan a iz redova označenih sa (2) uklonjen komentar. Ako su i (1) i (2) komentarisani, statični elementi neće biti inicijalizovani, kao što vidite iz rezultata programa. Takođe nije bitno da li je komentar uklonjen ispred samo jednog reda ili oba reda označena sa (2); statični elemenati će se inicijalizovati samo jednom.

Vežba 13: (1) Proverite naredbe iz prethodnog pasusa.

Vežba 14: (1) Napravite klasu s jednim statičnim poljem tipa **String** koje se inicijalizuje na mestu definisanja i drugim poljem koje inicijalizuje statičan blok. Dodajte statičnu metodu koja ispisuje oba polja i pokazuje da se oba inicijalizuju pre upotrebe.

Inicijalizacija nestatičnih instanci

Za inicijalizaciju nestatičnih promenljivih svakog objekta, Java obezbeđuje sličnu sintaksu, nazvanu *inicijalizacija instanci*. Sledi primer:

```
//: inicijalizacija/Solje.java
// Javina "Inicijalizacija instanci."
import static net.mindview.util.Print.*;

class Solja {
  Solja(int marker) {
    print("Solja(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

public class Solje {
  Solja solja1;
  Solja solja2;
  {
    solja1 = new Solja(1);
    solja2 = new Solja(2);
    print("solja1 i solja2 inicijalizovani");
  }
  Solje() {
    print("Solje()");
  }
  Solje(int i) {
```

```
    print("Solje(int)");
  }
  public static void main(String[] args) {
    print("Unutar metode main()");
    new Solje();
    print("nove Solje() završene");
    new Solje(1);
    print("nove Solje(1) završene");
  }
} /* Ispis:
Unutar metode main()
Solja(1)
Solja(2)
solja1 i solja2 inicijalizovani
Solje()
nove Solje() završene
Solja(1)
Solja(2)
solja1 i solja2 inicijalizovani
Solje(int)
nove Solje(1) završene
* ///:~
```

Obratite pažnju na to da odredba inicijalizacije instanci:

```
{
    solja1 = new Solja(1);
    solja2 = new Solja(2);
    print("solja1 i solja2 inicijalizovani");
  }
```

izgleda gotovo istovetno kao odredba za inicijalizaciju statičnih elemenata, samo što nedostaje rezervisana reč **static**. Ova sintaksa je neophodna da bi se obezbedila podrška za inicijalizaciju *anonimnih unutrašnjih klasa* (poglavlje *Unutrašnje klase*), ali omogućuje i da jamčite kako će se određene operacije izvršiti bez obzira na to koji će eksplicitni konstruktor biti pozvan. Iz rezultata programa vidite da se odredba inicijalizacije instanci izvršava pre svih konstruktora.

Vežba 15: (1) Napravite klasu s jednim poljem tipa **String** koje se inicijalizuje pomoću inicijalizacije instanci.

Inicijalizacija nizova

Niz je sekvenca objekata ili elemenata prostog tipa koji su jednog istog tipa i upakovani zajedno pod jednim imenom za identifikaciju. Nizovi se definišu i koriste pomoću *operatora indeksiranja* [ ]. Referencu niza definišete tako što iza imena tipa stavite prazne uglaste zagrade:

```
int[] a1;
```

Ako uglaste zagrade stavite posle identifikatora, značenje će biti potpuno isto.

```
int a1[];
```

Ovo se uklapa u očekivanja C i C++ programera. Prvi stil sadrži verovatno pristupačniju sintaksu, jer pokazuje da je tip promenljive niz celih brojeva. U ovoj knjizi biće korišćen taj stil.

Prevodilac ne dozvoljava da navedete veličinu niza. To nas vraća na pitanje referenci. Sve što u ovom trenutku imate jeste referenca na niz (rezervisali ste dovoljno prostora za tu referencu) i nikakav prostor za smeštanje samog niza nije rezervisan u memoriji. Da biste napravili skladište za niz, morate da napišete izraz za inicijalizaciju. Kod nizova, inicijalizacija može da se pojavi bilo gde u kodu, ali možete da koristite i posebnu vrstu izraza za inicijalizaciju nizova koji može da se pojavi samo na mestu definicije niza. Ova specijalna inicijalizacija zadaje skup vrednosti između vitičastih zagrada. U ovom slučaju, prevodilac vodi računa o zauzimanju skladišnog prostora (kao da ste koristili **new**). Na primer:

```
int [] a1 = { 1, 2, 3, 4, 5 };
```

Pa, čemu onda služe reference na nizove bez dodeljenih nizova?

```
int [] a2;
```

Pošto je u Javi moguće da jedan niz dodelite drugom, možete i da napišete:

```
a2 = a1;
```

Pri tome se, u stvari, samo kopira referenca, što naredni kôd pokazuje:

```
//: inicijalizacija/NizoviProstihTipova.java
import static net.mindview.util.Print.*;

public class NizoviProstihTipova {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i] = a2[i] + 1;
    for(int i = 0; i < a1.length; i++)
      print("a1[" + i + "] = " + a1[i]);
  }
} /* Ispis:
 a1[0] = 2
 a1[1] = 3
 a1[2] = 4
 a1[3] = 5
 a1[4] = 6
* ///:~
```

Obratite pažnju na to da je nizu **a1** dodeljena inicijalizaciona vrednost, a nizu **a2** nije; referenci **a2** vrednost je dodeljena kasnije – u ovom slučaju, ona pokazuje na drugi niz. Pošto su **a2** i **a1** tako postali pseudonimi istog niza, izmene sprovedene preko **a2** odražavaju se i u **a1**.

Svi nizovi imaju član (bilo da su nizovi objekata ili nizovi elemenata prostog tipa) koji možete čitati – ali ne i menjati – da biste znali koliko elemenata ima u nizu. Taj član je **length**. Pošto u Javi, kao u C-u i C++-u, nizovi počinju od indeksa 0, najveći element koji možete da indeksirate je **length – 1**. Ako izađete van granica, C i C++ to ćutke prihvataju i dozvoljavaju da vršljate dalje po memoriji, što predstavlja izvor mnogih grešaka. Java vas, međutim, štiti od takvih problema tako što proizvodi grešku pri izvršavanju (*izuzetak*) ako zakoračite van granica.[5]

Šta ako ne znate koliko će vam elemenata biti potrebno u nizu dok pišete program? Samo upotrebite operator **new** za pravljenje elemenata niza. U narednom primeru koristi se **new**, iako se pravi niz čiji su elementi prostog tipa (**new** ne može da napravi samo jednu promenljivu prostog tipa):

```
//: inicijalizacija/NizoviINew.java
// Pravljenje nizova operatorom new.
import java.util.*;
import static net.mindview.util.Print.*;

public class NizoviINew {
  public static void main(String[] args) {
    int[] a;
    Random slucajan = new Random(47);
    a = new int[slucajan.nextInt(20)];
    print("duzina niza a = " + a.length);
    print(Arrays.toString(a));
  }
} /* Ispis:
duzina niza a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
} ///:~
```

Pošto se veličina niza bira pseudoslučajno (metodom **Random.nextInt**(), čiji je rezultat slučajan broj između nule i njenog argumenta), jasno je da se niz zaista pravi za vreme izvršavanja. Pored toga, iz rezultata ovog programa vidite da elementi nizova prostih tipova automatski dobijaju podrazumevane vrednosti. (Za numeričke i promenljive tipa **char** to je nula, a za **boolean** to je **false**.)

Metoda **Arrays.toString**(), koja pripada standardnoj Javinoj biblioteci **java.util**, daje verziju jednodimenzionalnog niza koja se može ispisati.

---

[5] Naravno, proveravanje svakog pristupa nizu iziskuje vreme i kod i tu proveru ne možete da isključite, što znači da pristupi nizovima mogu biti izvor neefikasnosti u vašem programu ako se javljaju na važnim mestima. Tvorci Jave su smatrali da vredi žrtvovati nešto efikasnosti na uštrb sigurnosti na Internetu i produktivnosti programera. Mada ćete možda doći u iskušenje da sami pišete kôd koji bi trebalo efikasnije da pristupa nizovima, to je gubljenje vremena, pošto će automatske optimizacije tokom prevođenja i izvršavanja ubrzati pristupanje nizovima.

Naravno, u ovom slučaju niz je mogao da bude definisan i inicijalizovan u istoj naredbi:

```
int [] a = new int [slucajan.nextInt(20)];
```

Ako možete, bilo bi bolje da radite tako.

Kada napravite niz objekata, u stvari pravite niz referenci. Razmotrimo omotački tip **Integer**, koji je klasa, a ne prost tip:

```
//: inicijalizacija/NizObjekata.java
// Pravljenje niza čiji elementi nisu prostog tipa.
import java.util.*;
import static net.mindview.util.Print.*;

public class NizObjekata {
  public static void main(String[] args) {
     Random slucajan = new Random(47);
     Integer[] a = new Integer[slucajan.nextInt(20)];
     print("duzina niza a = " + a.length);
     for(int i = 0; i < a.length; i++) {
        a[i] = slucajan.nextInt(500); // Autopakovanje
     print(Arrays.toString(a));
   }
} /* Ispis: (Primer)
duzina niza a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51,
89, 309, 278, 498, 361, 20]
* ///:~
```

U gornjem primeru, čak i kada operatorom **new** napravimo niz:

```
Integer [] a = new Integer[slucajan.nextInt(20)];
```

dobićemo samo niz referenci pa inicijalizacija neće biti potpuna sve dok ne inicijalizujemo same reference praveći nove objekte tipa **Integer** (u ovom slučaju, preko autopakovanja):

```
a[i] = new Integer (slucajan.nextInt(500));
```

Ako tokom izvršavanja zaboravite da napravite objekat, javiće se izuzetak kada pokušate da upotrebite praznu lokaciju u nizu.

Niz objekata takođe možete da inicijalizujete koristeći listu između vitičastih zagrada. Za to postoje dva oblika:

```
//: inicijalizacija/InicijalizacijaNiza.java
// Inicijalizacija niza.
import java.util.*;

public class InicijalizacijaNiza {
  public static void main(String[] args) {
    Integer[ ] a = {
      new Integer(I),
```

```
      new Integer(2),
      3, // Autopakovanje
    };
    Integer[ ] b = new Integer [ ] {
      new Integer(1),
      new Integer(2),
      3, // Autopakovanje
    };
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  };
} /* Ispis:
[1, 2, 3]
[1, 2, 3]
* ///:~
```

U oba slučaja, završni zarez u listi inicijalizatora nije obavezan. (Zato je lakše održavanje dugačkih lista.) Poslednji zarez u listi inicijalizatora je opcioni. (Ova mogućnost je namenjena lakšem održavanju dugačkih lista.)

Iako je prvi oblik koristan, njegova ograničenja su veća, jer se može upotrebljavati samo na mestu definicije niza. Drugi i treći oblik možete upotrebiti bilo gde, čak i unutar poziva neke metode. Na primer, mogli biste napraviti niz objekata tipa **String** koje ćete proslediti metodi **main**() neke druge metode, da biste na taj način zadali alternativne argumente komandne linije toj metodi **main**():

```
//: inicijalizacija/DinamickiNiz.java
// Inicijalizacija nizova.

public class DinamickiNiz {
  public static void main(String[ ] args) {
    Druga.main(new String[ ]  {"tra", "la", "la" });
  };
}
class Druga  {
  public static void main(String[ ] args) {
  for(String s : args)
    System.out.print(s + " ");
  };
} /* Ispis:
tra la la
* ///:~
```

Niz napravljen za argument metode **Druga.main**() napravljen je na mestu poziva te metode, pa alternativne argumente možete zadati čak i u trenutku tog poziva.

Vežba 16: (1) Napravite niz objekata tipa **String** i svakom elementu dodelite po jedan **String**. Ispišite niz petljom **for**.

Vežba 17: (2) Napravite klasu s konstruktorom koji prima argument tipa **String**. Tokom pravljenja ispišite taj argument. Napravite niz referenci objekata te klase, ali nemojte praviti objekte koje ćete dodeliti tom nizu. Kada pokrenete program, proverite da li se ispisuju inicijalizacione poruke iz poziva konstruktora.

Vežba 18: (2) Dopunite prethodnu vežbu pravljenjem objekata koje ćete dodeliti nizu referenci.

Liste promenljivih parametara

Drugi oblik (inicijalizacije niza) obezbeđuje pogodnu sintaksu za pravljenje i pozivanje metoda koje mogu da proizvedu isti efekat kao lista promenljivih parametara u C-u (poznata i kao varargs). Lista može imati nepoznat broj argumenata, a mogu se pojaviti i nepoznati tipovi. Pošto sve klase u osnovi nasleđuju zajedničku korensku klasu **Object** (o čemu ćete više saznati napredujući kroz ovu knjigu), možete da napravite metodu čiji je argument niz elemenata tipa **Object** i da je pozovete na sledeći način:

```
//: inicijalizacija/PromenljivaLista.java
// Korišćenje sintakse nizova za pravljenje promenljive liste parametara.

class A { }

public class PromenljivaLista {
  static void ispisiNiz(Object[ ] argumenti) {
    for(Object obj : argumenti)
      System.out.print(obj + "  ");
    System.out.println();
  }
  public static void main(String[ ] argumenti) {
    ispisiNiz(new Object[ ] {
        new Integer(47), new Float(3.14), new Double(11.11)
    });
    ispisiNiz(new Object[ ] {"jedan", "dva", "tri" });
    ispisiNiz(new Object[ ] {new A(), new A(), new A()});
  }
} /* Ispis: (Primer)
47  3.14  11.11
jedan  dva  tri
A@1a46e30  A@3e25a5  A@19821f
*///:~
```

Možete videti da **print**() uzima niz elemenata tipa **Object**, a zatim foreach sintaksom prolazi kroz niz i ispisuje sve njegove objekte. Klase standardne Javine biblioteke daju razumljiv ispis, ali objekti ovde napravljenih klasa štampaju ime klase, znak @ i heksadecimalne cifre. Dakle, podrazumevano ponašanje (ako za svoju klasu ne definišete metodu **toString**( ), što će biti opisano u nastavku knjige) jeste da se ispisuje ime klase i adresa objekta.

Ovako su se pre pojave Jave SE5 pisali Java programi da bi se dobile promenljive liste parametara. Međutim, u Javi SE5 pojavila se dugo zahtevana mogućnost da se promenljive liste parametara definišu pomoću operatora tri tačke, kao što vidite u sledećoj metodi **ispisiNiz**():

```
//: inicijalizacija/NovaPromenljivaLista.java
// Korišćenje sintakse vezane za nizove, za pravljenje promenljive liste
// parametara.
public class NovaPromenljivaLista {
  static void ispisiNiz(Object... argumenti) {
    for(Object obj : argumenti)
      System.out.print(obj + "  ");
    System.out.println();
  }
  public static void main(String[] argumenti) {
    // Može primiti pojedinačne elemente:
    ispisiNiz(new Integer(47), new Float(3.14),
      new Double(11.11));
    ispisiNiz(47, 3.14F, 11.11);
    ispisiNiz("jedan", "dva", "tri");
    ispisiNiz(new A(), new A(), new A());
    // Ili niz:
    ispisiNiz((Object[ ])new Integer[ ]{ 1, 2, 3, 4 });
    ispisiNiz(); // Prazne liste su dozvoljene
  }
} /* Ispis: (Uzorak)
47  3.14  11.11
47  3.14  11.11
jedan  dva  tri
A@1bab50a  A@c3c749  A@150bd4d
1  2  3  4
*///:~
```

Kada se radi o promenljivim parametrima, više ne morate eksplicitno da ispisujete sintaksu nizova – to će prevodilac učiniti umesto vas čim zadate promenljive parametre. Svoj niz ćete svakako dobiti, zbog toga **print**( ) može foreach sintaksom da prođe kroza sve elemente niza. Međutim, na delu je više od proste automatske konverzije liste elemenata u niz. Obratite pažnju na pretposlednji red programa, gde se niz elemenata tipa **Integer** (napravljenih autopakovanjem) eksplicitno pretvara u niz elemenata tipa **Object** (da bi se izbeglo upozorenje prevodioca) i prosleđuje metodi **ispisiNiz**( ). Jasno, prevodilac vidi da je to već niz i na njemu ne obavlja nikakvu konverziju. Dakle, ako imate grupu stavki, možete ih proslediti kao listu, a ako niz već imate, prevodilac će ga prihvatiti kao listu promenljivih argumenata (parametara).

Poslednji red programa pokazuje da se listi promenljivih parametara može proslediti nula argumenata. To je korisno kada imate neobavezne prateće argumente:

```
//: inicijalizacija/OpcioniPratećiArgumenti.java

public class OpcioniPratećiArgumenti {
  static void f(int obavezan, String... pratići) {
    System.out.print("obavezan: " + obavezan + " ");
    for(String s : pratići)
      System.out.print(s + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, "jedan");
    f(2, "dva", "tri");
    f(0);
  }
} /* Ispis:
obavezan: 1 jedan
obavezan: 2 dva tri
obavezan: 0
*///:~
```

I ovo pokazuje da promenljive parametre možete upotrebljavati i sa zadatim tipom koji nije **Object**. U prethodnom primeru, svi promenljivi parametri moraju biti **String** objekti. U promenljivim parametrima mogu se koristiti svi tipovi, i prosti. U sledećem primeru pokazujemo da lista promenljivih parametara postaje niz i da je veličina tog niza nula ukoliko ta lista nema sadržaja:

```
//: inicijalizacija/TipPromenljivogParametra.java

public class TipPromenljivogParametra {
  static void f(Character... argumenti) {
    System.out.print(argumenti.getClass());
    System.out.println(" length " + argumenti.length);
  }
  static void g(int... argumenti) {
    System.out.print(argumenti.getClass());
    System.out.println(" duzina " + argumenti.length);
  }
  public static void main(String[] argumenti) {
    f('a');
    f();
    g(1);
    g();
    System.out.println("int[]: " + new int[0].getClass());
  }
} /* Ispis:
class [Ljava.lang.Character; duzina 1
class [Ljava.lang.Character; duzina 0
class [I duzina 1
class [I duzina 0
int[]: class [I
*///:~
```

Metoda **getClass**() je deo **Object**a i biće potpuno razmotrena u poglavlju *Podaci o tipu.* Ona daje klasu objekta, i kada je ispišete, dobijate kodiran znakovni niz koji predstavlja tip te klase. Vodeće [pokazuju da se radi o nizu tipa koji sledi. I naznačuje prosti **int**; radi još jedne provere, u poslednjem redu sam napravio niz tipa **int** i ispisao njegov tip. Time je dokazano da upotreba promenljivih parametara ne zavisi od autopakovanja, nego da zaista radi s prostim tipovima.

Međutim, promenljivi parametri lepo rade sa autopakovanjem. Na primer:

```
//: inicijalizacija/AutopakovanjePromenljivihParametara.java

public class AutopakovanjePromenljivihParametara {
  public static void f(Integer... argumenti) {
    for(Integer i : argumenti)
      System.out.print(i + " ");
    System.out.println();
  }
  public static void main(String[] argumenti) {
    f(new Integer(1), new Integer(2));
    f(4, 5, 6, 7, 8, 9);
    f(10, new Integer(11), 12);
  }
} /* Ispis:
1 2
4 5 6 7 8 9
10 11 12
*///:~
```

Imajte u vidu da u istoj listi argumenata možete mešati tipove i da autopakovanje selektivno unapređuje **int** argumente u tip **Integer**.

Promenljivi parametri komplikuju postupak preklapanja, iako to na prvi pogled izgleda sasvim bezopasno:

```
//: inicijalizacija/PreklapanjePromenljivihParametara.java

public class PreklapanjePromenljivihParametara {
  static void f(Character... argumenti) {
    System.out.print("prvi");
    for(Character c : argumenti)
      System.out.print(" " + c);
    System.out.println();
  }
  static void f(Integer... argumenti) {
    System.out.print("drugi");
    for(Integer i : argumenti)
      System.out.print(" " + i);
    System.out.println();
  }
```

```
  static void f(Long... argumenti) {
    System.out.println("treći");
  }
  public static void main(String[] argumenti) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //! f(); // Neće se prevesti -- višeznačno
  }
} /* Ispis:
prvi a b c
drugi 1
drugi 2 1
drugi 0
treći
*///:~
```

U svakom od prethodnih slučajeva, prevodilac upotrebljava autopakovanje da bi se prilagodio preklopljenoj metodi i poziva najprikladniju metodu.

Ali kada pozovete **f()** bez argumenata, on ne zna koju metodu da pozove. Iako je ta greška razumljiva, ona će verovatno iznenaditi programera klijenta.

Problem biste mogli pokušati da rešite dodavanjem nepromenljivog argumenta jednoj od metoda:

```
//: inicijalizacija/PreklapanjePromenljivihParametara2.java
// {CompileTimeError} (Neće se prevesti)

public class PreklapanjePromenljivihParametara2 {
  static void f(float i, Character... argumenti) {
    System.out.println("prvi");
  }
  static void f(Character... argumenti) {
    System.out.print("drugi");
  }
  public static void main(String[] argumenti) {
    f(1, 'a');
    f('a', 'b');
  }
} ///:~
```

Oznaka **{CompileTimeError}** u komentaru uklanja ovu datoteku iz skripta za prevođenje primera iz ove knjige. Ako je prevedete ručno, dobićete poruku o grešci:

reference to f is ambiguous, both method f(float,java.lang.Character...) in PreklapanjePromenljivihParametara2 and method f(java.lang.Character...) in PreklapanjePromenljivihParametara2 match

Ako obema metodama date nepromenljiv argument, upaliće:

```
//: inicijalizacija/PreklapanjePromenljivihParametara3.java

public class PreklapanjePromenljivihParametara3 {
  static void f(float i, Character... argumenti) {
    System.out.println("prvi");
  }
  static void f(char c, Character... argumenti) {
    System.out.println("drugi");
  }
  public static void main(String[] argumenti) {
    f(1, 'a');
    f('a', 'b');
  }
} /* Ispis:
prvi
drugi
*///:~
```

Po pravilu, listu promenljivih argumenata trebalo bi da upotrebljavate samo u jednoj od preklopljenih metoda. Ili, uopšte ne upotrebljavajte.

Vežba 19: (2) Napišite metodu koja prima **String** niz promenljivih argumenata. Uverite se da toj metodi možete proslediti listu elemenata tipa **String** razdvojenih zarezima ili **String**[].

Vežba 20: (1) Napišite metodu **main**() koja umesto uobičajene sintakse upotrebljava promenljive argumente. Ispišite sve elemente rezultujućeg niza **args**. Ispitajte tu metodu pomoću različitog broja argumenata s komandne linije.

Nabrojani tipovi

Naizgled mali dodatak u Javi SE5 jeste rezervisana reč **enum** koja mnogo olakšava život kada treba da grupišete i upotrebite skup *nabrojanih tipova* (engl. *enumerated types*). Nekada biste napravili skup konstantnih celobrojnih vrednosti, ali one „ne znaju" da čine skup, pa se rizičnije i teže upotrebljavaju. Inače, nabrojani tipovi su toliko često potrebni da su oduvek postojali u C-u, C++-u i u više drugih jezika. Pre Jave SE5, programeri na Javi morali su mnogo da znaju i da paze kako bi valjano proizveli **enum** efekat. Sada i Java ima **enum**, a on je čak i moćniji od onoga što imaju C/C++. Evo jednostavnog primera:

```
//: inicijalizacija/Ljuto.java
public enum Ljuto {
  NE, BLAGO, SREDNJE, MNOGO, PALI
} ///:~
```

Ovime je napravljen nabrojani tip **Ljuto** s pet imenovanih vrednosti. Pošto su primerci nabrojanih tipova konstante, po konvenciji ih pišemo velikim slovima (ako ime sadrži više reči, razdvajamo ih donjom crticom – podvlakom).

Kada vam zatreba **enum**, napravite referencu tog tipa i dodelite je nekom primerku (instanci):

```
//: inicijalizacija/JednostavnaUpotrebaNabrojanihTipova.java
public class JednostavnaUpotrebaNabrojanihTipova {
  public static void main(String[] args) {
    Ljuto kolikoljuto = Ljuto.SREDNJE;
    System.out.println(kolikoljuto);
  }
} /* Ispis:
SREDNJE
*///:~
```

Kada napravite **enum**, prevodilac mu automatski dodaje korisne mogućnosti. Na primer, pravi metodu **toString()** kojom možete lako da prikažete ime **enum** instance. (Upravo tako je prethodna naredba **print** dala rezultat. Prevodilac pravi i metodu **ordinal()** koja daje redni broj deklarisanja odredene **enum** konstante u njenoj klasi, i statičnu metodu **values()** koja daje niz vrednosti **enum** konstanti poređanih po redosledu kojim su bile deklarisane:

```
//: inicijalizacija/RedosledNabrojanihTipova.java
public class RedosledNabrojanihTipova {
  public static void main(String[] args) {
    for(Ljuto s : Ljuto.values())
      System.out.println(s + ", " + s.ordinal() + ". po redu");
  }
} /* Ispis:
NE, 0. po redu
BLAGO, 1. po redu
SREDNJE, 2. po redu
MNOGO, 3. po redu
PALI, 4. po redu
*///:~
```

Iako **enum** izgleda kao nov tip podataka, ta rezervisana reč samo određuje ponašanje prevodioca tokom generisanja klase za **enum**, pa možete umnogome da ga tretirate kao i svaku drugu klasu. U stvari, nabrojani tipovi i jesu klase sa sopstvenim metodama.

Posebno je zgodno kako se nabrojani tipovi mogu upotrebljavati unutar naredbe **switch**:

```
//: inicijalizacija/Pljeskavica.java

public class Pljeskavica {
  Spiciness stepen;
  public Pljeskavica(Ljuto stepen) { this.stepen = stepen;}
```

```
  public void opiši() {
    System.out.print("Ova pljeskavica je ");
    switch(stepen) {
      case NE:    System.out.println("potpuno nezačinjena.");
                  break;
      case BLAGO:
      case SREDNJE: System.out.println("ljutkasta.");
                    break;
      case MNOGO:
      case PALI:
      default:     System.out.println("preljuta.");
    }
  }
  public static void main(String[] args) {
    Pljeskavica
      bez = new Pljeskavica(Ljuto.NE),
      biber = new Pljeska(Začinjenost.SREDNJE),
      feferon = new Pljeska(Začinjenost.MNOGO);
    bez.opiši();
    biber.opiši();
    feferon.opiši();
  }
} /* Ispis:
Ova pljeskavica je potpuno nezačinjena.
Ova pljeskavica je ljutkasta.
Ova pljeskavica je preljuta.
*///:~
```

Pošto **switch** služi za izbor iz ograničenog skupa mogućih vrednosti, idealno odgovara definisanju tipova nabrajanjem. Obratite pažnju na to kako imena nabrojanih tipova mnogo jasnije pokazuju šta će program uraditi.

Uopšte uzev, **enum** možete da koristite kao drugi način definisanja tipova podataka. To je i bio cilj, pa na njih više ne morate obraćati posebnu pažnju. Pre uvođenja nabrojanih tipova u Javu SE5, trebalo je uložiti dosta truda za pravljenje ekvivalentnog nabrojanog tipa koji je bezbedno upotrebljavati.

Ovo je dovoljno za shvatanje i korišćenje jednostavnih nabrojanih tipova. Njima smo posvetili posebno poglavlje, *Nabrojani tipovi*.

Vežba 21: (1) Definišite **enum** od šest najmanje vrednih tipova papirnih novčanica. Metodom **values**() prodite kroz sve nabrojane vrednosti, ispišite svaku vrednost i (metodom **ordinal**()) njen redni broj.

Vežba 22: (2) Napišite naredba **switch** za **enum** iz prethodnog primera. Za svaki **case**, ispišite opis te novčanice.

Sažetak

Prilično razrađen mehanizam inicijalizacije, kakav je konstruktor, trebalo bi da ukaže na ogromnu važnost koja je u ovom jeziku data inicijalizaciji. Jedno od prvih zapažanja o produktivnosti u C-u, koje je Stroustrup izneo dok je projektovao jezik C++, bilo je da nepravilna inicijalizacija promenljivih prouzrokuje značajan deo problema pri programiranju. Ovu vrstu grešaka je teško pronaći, a slični problemi javljaju se i pri nepravilnom čišćenju. Pošto konstruktori omogućuju da *garantujete* pravilnu inicijalizaciju (prevodilac neće dozvoliti da objekat bude napravljen bez valjanog poziva konstruktora), dobijate potpunu kontrolu i sigurnost.

U C++-u uništavanje je veoma bitno, jer objekti napravljeni operatorom **new** moraju eksplicitno da budu uništeni. Sakupljač smeća u Javi automatski oslobađa memoriju, tako da u većini slučajeva odgovarajuće metode za čišćenje nisu potrebne. (Ali kada jesu, morate sami sve da uradite.) Kada vam ne treba ponašanje nalik na destruktore, Javin sakupljač smeća umnogome pojednostavljuje programiranje i dodaje preko potrebnu sigurnost u upravljanju memorijom. Neki sakupljači smeća mogu čak da čiste i druge resurse, kao što su identifikatori datoteka i grafičkih objekata. Međutim, sakupljač smeća dodaje režijske troškove tokom izvršavanja programa, čiju je cenu teško proceniti zbog tradicionalne sporosti Javinih interpretatora. Iako su se performanse Jave znatno poboljšale tokom vremena, taj jezik se, zbog svoje sporosti, i dalje ne primenjuje u nekim situacijama.

O konstruktorima postoji još stvari koje nisu razmotrene u ovom poglavlju. Pošto garancija konstrukcije važi i kada pravite nove klase *kompozicijom* ili *nasleđivanjem*, potrebna je dodatna sintaksa da bi se to podržalo. O kompoziciji, nasleđivanju i tome kako te operacije utiču na konstruktore, naučićete u narednim poglavljima.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Kontrola pristupa

Kontrola pristupa (ili sakrivanje realizacije) znači „moći popraviti ono što prvi put nije bilo dobro urađeno".

SVI DOBRI PISCI – UKLJUČUJUĆI TU I ONE KOJI PIŠU SOFTVER – ZNAJU DA NIŠTA NIJE DOBRO napisano dok se više puta ne preradi. Ako deo svog koda na neko vreme ostavite u fioku i zatim ga ponovo pročitate, možda ćete otkriti mnogo bolji način da ga uradite. To je jedan od glavnih razloga za *ponovnu podelu na proste faktore*, dakle za preradu funkcionalnog koda da bi se učinio čitljiviji i razumljiviji, a samim tim će se i lakše održavati.[1]

Međutim, postoje i problemi zbog nastojanja da se postojeći kôd menja i poboljšava. Često korisnici (*programeri klijenti*) računaju na to da će određeni aspekat koda ostati neizmenjen. Dakle, vi hoćete da ga menjate, a oni da on ostane neizmenjen. Zato je jedan od osnovnih ciljeva objektno orijentisanog dizajna „razdvojiti ono što se menja od stvari koje ostaju iste".

Ovo je posebno važno za biblioteke. Korisnici biblioteke moraju biti sigurni kako se mogu osloniti na deo koji koriste i kako neće morati ponovo da pišu programe kada se pojavi nova verzija biblioteke. S druge strane, tvorac biblioteke mora imati slobodu da pravi izmene i unapređenja a da pri tom bude siguran kako te izmene neće uticati na kôd klijentskog programa.

Sve to može da se ostvari preko konvencija. Na primer, programer biblioteke mora da se složi kako neće izbacivati postojeće metode kada menja klasu u biblioteci, jer bi time narušio kôd programera klijenta. Obrnuta situacija je znatno složenija. Ako je reč o polju, kako tvorac biblioteke može da zna kojim su poljima pristupali programeri klijenti? Isto važi i za metode koje su samo deo realizacije klase i nisu namenjene da ih direktno upotrebljavaju programeri klijenti. Šta ako tvorac biblioteke želi da potpuno izbaci staru realizaciju i napiše novu? Menjanje bilo koje od gore pomenutih članica moglo bi da naruši kôd programera klijenta. Stoga su tvorcu biblioteke vezane ruke i ne može ništa da menja.

Da bi se rešio ovaj problem, u Javi postoje *specifikatori pristupa* koji tvorcu biblioteke omogućavaju da naznači šta je programerima klijentima dostupno, a šta nije. Nivoi dozvoljenog pristupa od „potpunog pristupa" do „najmanjeg pristupa" jesu **public**, **protected**, paketni pristup (za koji ne postoji rezervisana reč) i **private**. Na osnovu prethodnog možete pomisliti da kao projektant biblioteke treba sve da čuvate što „privatnije" moguće i da izložite samo metode koje želite da programeri klijenti koriste. To je sasvim tačno, iako je često suprotno intuiciji stručnjaka koji programiraju na drugim jezicima (posebno na C-u) i koji su navikli da svemu pristupaju bez ograničenja. Do kraja ovog poglavlja trebalo bi da se uverite u značaj kontrole pristupa u Javi.

---

[1] Videti *Refactoring: Improving the Design of Existing Code*, koji su napisali Martin Fowler i dr. (Addison-Wesley, 1999). Povremeno se neko pobuni protiv ponovne podele na proste faktore, sa argumentom da je kôd koji radi savršeno dobar i da preradivati ga znači gubiti vreme. Takav argument nije valjan zato što lavovski deo vremena i novca koji se uloži u projekat ne otpada na prvo pisanje koda, nego na njegovo održavanje. Učiniti kôd razumljivijim, znači uštedeti mnogo novca.

Koncept biblioteke komponenata i kontrole pristupa ovim nije završen. I dalje postoji pitanje kako se komponente povezuju u jednu koherentnu biblioteku. U Javi se paketi prave pomoću rezervisane reči **package**, a na specifikatore pristupa utiče da li je klasa u istom ili u nekom drugom paketu. Na početku ovog poglavlja naučićete kako se komponente biblioteke stavljaju u pakete, a nakon toga moći ćete da razumete potpuno značenje specifikatora pristupa.

Paket: bibliotečka jedinica

Paket sadrži grupu klasa, objedinjenu u istom *prostoru imena.*

Na primer, biblioteka sa alatkama **java.util** deo je standardne distribucije Jave. **ArrayList** je jedna od klasa u paketu **java.util**. Jedan od načina upotrebe klase **ArrayList** jeste da navedete njeno puno ime **java.util.ArrayList**.

```
//: pristup/PunoIme.java

public class PunoIme {
  public static void main(String[] args) {
    java.util.ArrayList lista = new java.util.ArrayList();
  }
} ///:~
```

Ovo brzo postaje zamorno, pa ćete umesto navođenja punog imena klase verovatno radije upotrebljavati rezervisanu reč **import**. Ako želite da uvezete samo jednu klasu, njeno ime možete da navedete u naredbi **import**:

```
//: pristup/JedanImport.java
import java.util.ArrayList;

public class JedanImport {
  public static void main(String[] args) {
    ArrayList lista = new java.util.ArrayList();
  }
} ///:~
```

Sada **ArrayList** možete da koristite bez imena paketa. Međutim, nije dostupna nijedna druga klasa iz paketa **java.util**. Da biste uvezli sve klase biblioteke, upotrebite * kao što je uradeno u ostalim primerima u ovoj knjizi:

```
import java.util.*;
```

Ovo uvoženje se radi da bi se obezbedio mehanizam za upravljanje imenskim prostorima (engl. *name spaces*). Imena svih vaših članica klasa su izolovana jedna od drugih. Metoda **f()** iz klase **A** neće se sudariti s metodom f() koja ima isti potpis (listu argumenata) iz klase **B**. Ali, kako stoje stvari sa imenima klasa? Pretpostavimo da napravite klasu **Stek** koja se instalira na računar na kome već postoji klasa **Stek** koju je neko drugi napisao. Zbog potencijalnih sukoba imena bitno je da imate potpunu kontrolu imenskih prostora u Javi i da za svaku klasu pravite jedinstvenu kombinaciju identifikatora.

Do sada se veći deo primera u ovoj knjizi nalazio u jednoj datoteci i bio je napisan za lokalnu upotrebu, te se nije bavio imenima paketa. U tim primerima klase su ipak bile u paketu, tzv. neimenovanom ili *podrazumevanom paketu.* To se svakako može raditi, i u nastavku knjige takav pristup biće korišćen gde god je moguće, jednostavnosti radi. Ali ako nameravate da pravite biblioteke ili programe koje će voditi računa i o drugim programima na istom računaru, morate da vodite računa o sprečavanju sukoba imena klasa.

Kada pravite datoteku sa izvornim kodom za Javu, ta datoteka se često naziva i *jedinica za prevođenje* (engl. *compilation unit, translation unit*). Ime svake jedinice za prevođenje mora da se završava sa **.java**, a unutar nje može da postoji jedna javna (**public**) klasa istog imena kao i datoteka (uključujući i velika i mala slova, ali bez nastavka **.java**). U svakoj jedinici za prevođenje može da se nalazi samo *jedna* javna klasa, inače se prevodilac buni. Ostale klase iz te jedinice za prevođenje, ako ih uopšte ima, skrivene su od spoljnog sveta izvan tog paketa, jer one *nisu* javne i predstavljaju klase za „podršku" glavnoj javnoj klasi.

Organizacija koda

Pri prevođenju datoteke **.java**, dobijate po jednu izlaznu datoteku *za svaku klasu* iz datoteke **.java**. Svaka od tih izlaznih datoteka ima isto ime kao odgovarajuća klasa u datoteci **.java**, ali s nastavkom **.class**. Stoga iz malog broja datoteka **.java** možete da dobijete prilično broj datoteka **.class**. Ako ste programirali na jeziku koji se prevodi u izvršni oblik, verovatno ste navikli da prevodilac pravi međuoblik (obično datoteku **.obj**) koji zatim s drugim datotekama istog oblika pakujete pomoću povezivača (engl. *linker*) da biste napravili izvršnu datoteku ili to činite pomoću bibliotekara (engl. *librarian*) da biste napravili biblioteku. Java ne radi na taj način. Program koji radi je skup datoteka **.class** koje mogu da budu spakovane i komprimovane u JAR arhivu (pomoću Javinog arhivara **jar.**) Posao Javinog interpretatora je da pronađe, učita i interpretira[2] te datoteke.

Biblioteka je skup takvih datoteka sa klasama. Svaka izvorna datoteka obično ima jednu klasu koja je javna i proizvoljan broj nejavnih klasa, pa za svaku izvornu datoteku postoji jedna javna (**public**) komponenta. Ako želite da naznačite da sve te komponente (koje se nalaze u odvojenim datotekama **.java** i **.class**) idu zajedno, koristi se rezervisana reč **package.**

Ako koristite naredbu **package**, ona *mora* da bude u prvom redu koji nije komentar u datoteci. Kada napišete:

```
package pristup;
```

naznačavate da je ta jedinica za prevođenje deo biblioteke pod imenom **pristup.** Odnosno, drugačije rečeno, naznačavate da je javno ime klase, iz te jedinice za prevođenje, pod kišobranom imena **pristup**. Ako neko želi da koristi to ime, mora ili da potpuno navede dato ime ili da upotrebi rezervisanu reč **import** u kombinaciji s imenom **pristup,** kao što je ranije objašnjeno. (Imajte u vidu da konvencija za davanje imena Javinim paketima nalaže da se koriste samo mala slova, čak i za reči u sredini imena.)

[2] Java baš ni sa čim ne nameće upotrebu interpretatora. Postoje Java prevodioci u lokalni kôd koji prave jednu izvršnu datoteku.

Na primer, pretpostavimo da je ime datoteke **MojaKlasa.java**. To znači da u njoj može da postoji jedna i samo jedna javna klasa i da ime te klase mora da bude **MojaKlasa** (s tom kombinacijom velikih i malih slova):

```
//: pristup/mojpaket/MojaKlasa.java
package pristup.mojpaket;

public class MojaKlasa {
  // ...
} ///:~
```

Ako neko želi da koristi klasu **MojaKlasa** ili neku drugu javnu klasu iz paketa **pristup**, mora da upotrebi rezervisanu reč **import** da bi ime ili imena iz paketa **pristup** postala dostupna. Takođe može da upotrebi potpuno određeno ime:

```
//: pristup/OdređenaMojaKlasa.java

public class OdređenaMojaKlasa {
  public static void main(String[] args) {
    pristup.mojpaket.MojaKlasa m =
      new pristup.mojpaket.MojaKlasa();
  }
} ///:~
```

Rezervisana reč **import** znatno doprinosi jednostavnosti:

```
//: pristup/OdređenaMojaKlasa.java
import pristup.mojpaket.*;

public class UvezenaMojaKlasa {
  public static void main(String[] args) {
    MojaKlasa m = new MojaKlasa();
  }
} ///:~
```

Vredi imati na umu da projektantu biblioteke, rezervisane reči **package** i **import** omogućavaju da podeli jedan globalni imenski prostor i izbegne višesmislenost imena, bez obzira na to koliko se ljudi poveže na Internet i počne da piše klase na Javi.

Pravljenje jedinstvenih imena paketa

Budući da paketi nikada zaista nisu „upakovani" u jednu datoteku, jedan paket može da se sastoji od više datoteka **.class** i stvari mogu da postanu prilično komplikovane. Logičan potez bio bi da stavite sve datoteke **.class** jednog paketa u poseban direktorijum; odnosno, da iskoristite hijerarhijsku strukturu sistema datoteka operativnog sistema. To je jedan način na koji Java rešava pomenuti problem; drugi način ćete videti kasnije, kada bude predstavljana alatka **jar**.

Grupisanje datoteka jednog paketa u zaseban direktorijum rešava i još dva problema: pravljenje jedinstvenih imena paketa i pronalaženje klasa koje mogu biti zakopane negde u dubini strukture direktorijuma. To se postiže tako što se putanja do datoteka **.class** ubacuje u ime paketa. Po konvenciji, prvi deo imena paketa je ime Internet domena tvorca klase, u obrnutom redosledu. Pošto su imena Internet domena garantovano jedinstvena, *ako* primenjujete ovu konvenciju, ime vašeg paketa će sigurno biti jedinstveno i neće doći do sukoba imena (tj. dok neko drugi ne preuzme vaš domen i ne počne da piše kôd na Javi sa istim putanjama koje ste i vi koristili). Naravno, ako nemate svoj domen, smislite malo verovatnu kombinaciju (kao što je vaše ime i prezime) da biste napravili jedinstveno ime paketa. Ako ste odlučili da objavljujete kôd pisan na Javi, isplati se uložiti relativno mali napor u pribavljanje ličnog domena.

Drugi deo ove tehnike je da preslikate ime paketa na ime direktorijuma na vašem računaru. Kada izvršni sistem treba da učita datoteku **.class**, može da pronađe direktorijum u kome se ona nalazi.

Javin interpretator radi sledeće: prvo pronalazi sistemsku promenljivu CLASSPATH[3] (koja se postavlja u operativnom sistemu, što ponekad uradi instalacioni program koji na računaru instalira Javu ili na Javi zasnovanu alatku). CLASSPATH sadrži jedan ili više direktorijuma koji se koriste kao korenski za traženje datoteka **.class**. Interpretator u imenu paketa zamenjuje svaku tačku kosom crtom da bi generisao putanju od korena definisanog promenljivom CLASSPATH (tako paket **foo.bar.baz** postaje **foo\bar\ baz** ili **foo/bar/baz** ili možda nešto drugo, u zavisnosti od vašeg operativnog sistema). To se zatim spaja s raznim direktorijumima iz putanje CLASSPATH. Na tim mestima interpretator traži datoteku **.class** čije ime odgovara klasi koju pokušavate da napravite. (On takođe pretražuje i neke standardne direktorijume u zavisnosti od mesta na kome se interpretator nalazi.)

Da biste ovo shvatili, razmotrite ime mog domena – **MindView.net**. Kada obrnemo redosled i sve ispišemo malim slovima, **net.mindview** ustanovljava se jedinstveno globalno ime za moje klase. (Nastavci com, edu, org itd. ranije su u Javinim paketima pisani velikim slovima, ali je to promenjeno u Javi 2, tako da se celo ime piše malim slovima.) Ako odlučim da napravim biblioteku pod imenom **simple**, uneću podelu u globalno ime i dobiti ime paketa:

```
package net.mindview.simple;
```

Sada ovo ime paketa može da se koristi kao imenski prostor za sledeće dve datoteke:

```
//: net/mindview/simple/Vector.java
// Pravljenje paketa
package net.mindview.simple;

public class Vector {
  public Vector() {
    System.out.println("net.mindview.simple.Vector");
  }
} ///:~
```

[3] Sistemske promenljive ćemo pisati velikim slovima (npr. CLASSPATH).

Kao što je već bilo rečeno, naredba **package** mora da bude u prvom redu koji nije komentarisan. Druga datoteka je veoma slična:

```
//: net/mindview/simple/List.java
// Pravljenje paketa.
package net.mindview.simple;

public class List {
  public List() {
    System.out.println("net.mindview.simple.List");
  }
} ///:~
```

Obe datoteke se smeštaju u poddirektorijum na mom sistemu:

```
C:\DOC\JavaT\net\mindview\simple
```

(Obratite pažnju na to da se u prvom redu komentara u svakoj datoteci u ovoj knjizi daje mesto direktorijuma te datoteke u stablu izvornog koda. To je urađeno zbog alatke koja u ovoj knjizi automatski pronalazi kôd.)

Ako krenete od kraja ove putanje, vidite ime paketa **net.mindview.simple**, ali šta je s prvim delom putanje? O tome je vođeno računa u sistemskoj promenljivoj CLASSPATH koja, na mom računaru, ima sledeći sadržaj:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Vidite da promenljiva CLASSPATH može da sadrži više alternativnih putanja za pretraživanje.

Međutim, kada koristite JAR datoteke, postoje izvesne izmene. U CLASSPATH morate da stavite i ime JAR datoteke, a ne samo putanju do nje. Znači, ako bi se JAR datoteka zvala **grozd.jar**, u vašoj promenljivoj CLASSPATH trebalo bi da piše:

```
CLASSPATH=.; D:\JAVA\LIB;C:\ukusi\grozd.jar
```

Kada promenljivu CLASSPATH podesite kako treba, narednu datoteku možete da stavite u bilo koji direktorijum:

```
//: pristup/LibTest.java
// Koristi biblioteku.
import net.mindview.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} /* Ispis:
net.mindview.simple.Vector
net.mindview.simple.List
* ///:~
```

Kada prevodilac naiđe na naredbu **import** za biblioteku **simple**, on počinje da pretražuje direktorijume navedene u promenljivoj CLASSPATH, tražeći poddirektorijum **net\mindview\simple** a zatim i prevedene datoteke odgovarajućih imena (**Vector.class** za klasu **Vector** i **List.class** za klasu **List**). Zapamtite da obe klase kao i potrebne metode u klasama **Vector** i **List** moraju da budu javne.

Pravilno podešavanje promenljive CLASSPATH bio je veliki problem početnicima u Javi (meni takođe, kada sam počinjao), pa je SUN u kasnijim verzijama Jave „opametio" razvojno okruženje. Kada instalirate Javu, primetićete da ćete moći da prevodite i pokrećete osnovne programe u Javi, čak i ako ne podesite promenljivu CLASSPATH. Međutim, da biste preveli i izvršavali izvorni kôd iz ove knjige (dostupan na lokaciji *www.MindView.net*), promenljivoj CLASSPATH moraćete dodati osnovni direktorijum stabla koda knjige.

Vežba 1: (1) Napravite klasu u nekom paketu. Napravite primerak te klase izvan tog paketa.

Dvosmislenost i sukobi imena

Šta se dešava ako se preko * uvezu dve biblioteke koje sadrže ista imena? Na primer, pretpostavimo da u programu piše:

```
import net.mindview.simple.*;
import java.util.*;
```

Pošto i **java.util.*** takođe sadrži klasu **Vector**, to može da izazove sukob imena. Međutim, dokle god ne napišete kôd koji izaziva dvosmislenost, sve će biti u redu – to je dobro, inače biste morali mnogo da pišete da biste izbegli sukob koji se nikada neće dogoditi.

Sukob *nastaje* ako sada pokušate da napravite objekat klase **Vector**:

```
Vector v = new Vector();
```

Na koju se klasu **Vector** ovo odnosi? Prevodilac to ne može da zna, kao ni čitalac. Stoga se prevodilac žali i zahteva da budete izričiti. Ako, na primer, želite standardni Javin **Vector**, morate da napišete:

```
java.util.Vector v = new java.util.Vector();
```

Pošto ovo (zajedno s promenljivom CLASSPATH) potpuno određuje lokaciju klase **Vector**, naredba **import.java.util.*** postaje neophodna samo ako hoćete da koristite i nešto drugo iz biblioteke **java.util**.

Sukobe imena biste mogli sprečiti tako što ćete uvesti samo jednu klasu umesto celog paketa, ukoliko u istom programu ne upotrebite oba sukobljena imena. (U tom slučaju, morali biste da navedete njihova celokupna, tj. potpuno određena imena.)

Vežba 2: (1) Pretvorite delove koda u ovom odeljku u program i uverite se da se sukobi zaista dešavaju.

Lična biblioteka sa alatkama

Pošto sada sve ovo znate, u stanju ste da napravite svoje biblioteke sa alatkama kako biste smanjili ili ukinuli ponavljanje koda. Na primer, pogledajte pseudonim za naredbu **System.out.println()** koji smo koristili da bismo smanjili količinu teksta koji moramo da pišemo. Neka bude deo klase pod imenom **Print**, tako da na kraju dobijemo čitljiv statički uvoz:

```
//: net/mindview/util/Print.java
// Metode ispisivanja koje se mogu upotrebljavati bez određenja
// paketa i putanje, kada ih u Javu SE5 uvezemo kao statičke:
package net.mindview.util;
import java.io.*;

public class Print {
  // Ispiši i pređi u nov red:
  public static void print (Object obj) {
    System.out.println(obj);
  }
  // Samo pređi u nov red:
  public static void print () {
    System.out.println();
  }
  // Ispiši bez prelaska u nov red:
  public static void printnb (Object obj) {
    System.out.print(obj);
  }
  // Nova metoda printf() Jave SE5 (kao u C-u):
  public static PrintStream
  printf(String format, Object... argumenti) {
    return System.out.printf(format, argumenti);
  }
}///:~
```

Ove skraćenice možete da koristite za ispisivanje svega, bilo s prelaskom u novi red (**print()**) ili bez prelaska u novi red (**printnb()**).

Ova datoteka mora da se nalazi u direktorijumu koji počinje na nekoj lokaciji iz promenljive CLASSPATH i zatim se nastavlja sa **net/mindview**. Nakon prevođenja, statičke metode **print()** i **printnb()** možete da koristite bilo gde na svom sistemu pomoću naredbe **import static**:

```
//: pristup/PrintTest.java
// Koristi statičke metode za ispisivanje, koje pripadaju klasi
// Print.java.
import net.mindview.util.Print.*;

public class PrintTest {
  public static void main(String[] args) {
    print("Dostupna od sada pa nadalje!");
    print(100);
```

```
    print(100L);
    print(3.14159);
  }
} /* Ispis:
Dostupna od sada pa nadalje!
100
100
3.14159
* ///:~
```

Druga komponenta ove biblioteke mogu biti metode **range( )** koje ste upoznali u poglavlju *Kontrolisanje izvršavanja*. One omogućavaju zadavanje jednostavnih nizova celih brojeva foreach sintaksom:

```
//: net/mindview/util/Range.java
// Metode za pravljenje nizova koje se mogu upotrebljavati
// bez potpuno određenih imena, koristeći Java SE5 naredbe static import:
package net.mindview.util;

public class Range {
  // Napravi niz [0..n)
  public static int[] range(int n) {
    int[] rezultat = new int[n];
    for(int i = 0; i < n; i++)
      rezultat[i] = i;
    return rezultat;
  }
  // Napravi niz [početak..kraj)
  public static int[] range(int početak, int kraj) {
    int sz = kraj - početak;
    int[] rezultat = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = početak + i;
    return rezultat;
  }
  // Napravi niz [start..end) uz inkrement korak
  public static int[] range(int početak, int kraj, int korak) {
    int sz = (kraj - početak)/korak;
    int[] rezultat = new int[sz];
    for(int i = 0; i < sz; i++)
      rezultat[i] = početak + (i * korak);
    return rezultat;
  }
} ///:~
```

Od sada možete u svoju biblioteku dodati svaku korisnu novu alatku kada god naiđete na nju. Biblioteci **net.mindview.util** dodavaćemo komponente kroz celu knjigu.

Korišćenje uvoženja za promenu ponašanja programa

U Javi nedostaje *uslovno prevođenje* iz C-a koje omogućava da promenom vrednosti nekog semafora promenite ponašanje programa, bez drugih izmena u kodu. Takva mogućnost je izostavljena iz Jave verovatno zbog toga što je u C-u najčešće korišćena za rešavanje pitanja različitih platformi: u zavisnosti od platforme za koju se program prevodi, prevode se različiti delovi koda. Pošto je predviđeno da Java automatski pravi prenosiv kôd, takva mogućnost je smatrana nepotrebnom.

Postoje, međutim, i druge korisne upotrebe uslovnog prevođenja. Ono se najčešće koristi za ispravljanje grešaka u programu. Funkcije za otklanjanje grešaka su uključene u razvojnoj fazi, a u finalnom proizvodu su onemogućene. Promenom paketa koji uvozite, birate verziju za otklanjanje grešaka ili verziju za isporuku. Ova tehnika može da se koristi za bilo koju vrstu uslovnog prevođenja.

Vežba 3: (2) Napravite dva paketa: **debug** i **debugoff** koji sadrže identičnu klasu s metodom **debug( )**. Prva verzija prikazuje na konzoli svoj argument tipa **String**, a druga ne radi ništa. Upotrebite naredbu **static import** da biste u program za ispitivanje uvezli tu klasu i pokažite na delu uslovno prevođenje.

Upozorenje pri radu s paketima

Treba zapamtiti sledeće: svaki put kada pravite paket i date mu ime, implicitno određujete strukturu direktorijuma. Paket *mora* da se nalazi u direktorijumu na koji ukazuje ime paketa i taj direktorijum mora da bude dostupan pretraživanjem promenljive CLASSPATH. Eksperimentisanje rezervisanom reči **package** u početku može da zbuni jer će se, ako se ne držite pravila o imenu paketa i imenu direktorijuma, pojaviti mnoštvo tajanstvenih grešaka pri izvršavanju. Na primer, možete da dobijete poruku kako nije bilo moguće naći određenu klasu i ako se ona nalazi u tekućem direktorijumu. Ako dobijete takvu poruku o grešci, stavite naredbu **package** pod komentar i ako program proradi, znaćete gde leži problem.

Povedite računa o tome da se prevedeni kôd često ne smešta u direktorijum u kojem je izvorni kôd, ali izvršnom okruženju Jave morate omogućiti da prevedeni kôd pronađe pomoću promenljive CLASSPATH.

Specifikatori pristupa u Javi

Specifikatori pristupa (engl. *access specifiers*) u Javi su **public**, **protected** i **private**. Stavljaju se ispred svake definicije svakog člana klase, bilo da je u pitanju polje ili metoda. Svaki specifikator pristupa kontroliše pristup samo pojedinoj definiciji.

Ukoliko ne zadate specifikator pristupa, podrazumeva se „paketni pristup". Na jedan ili na drugi način, za sve je definisana neka vrsta pristupa. U narednim odeljcima, naučićete sve o raznim vrstama pristupa.

Paketni pristup

U svim primerima do ovog poglavlja specifikator pristupa uopšte nije bio naveden. Za podrazumevani pristup ne postoji rezervisana reč, ali se on obično naziva *paketni* (katkada i „prijateljski"). To znači da sve ostale klase iz tekućeg paketa imaju pristup tom članu, ali sve klase van tog paketa taj član vide kao privatni (i ne mogu da mu pristupe). Pošto jedinica za prevođenje – datoteka – može da pripada samo jednom paketu, sve klase unutar nje automatski su dostupne jedna drugoj, zato što imaju paketni pristup.

Paketni pristup omogućava da grupišete srodne klase u jedan paket tako da mogu lako međusobno da komuniciraju. Kada klase grupišete u paket, čime paketnim članovima dozvoljavate međusobni pristup, vi postajete „vlasnik" koda u tom paketu. Ima logike u tome da samo kôd u vašem vlasništvu treba da ima paketni pristup ostalom kodu u vašem vlasništvu. Može se reći da paketni pristup opravdava grupisanje klasa u pakete. U mnogim jezicima, definicije možete svakojako da organizujete, ali ste u Javi prisiljeni da ih organizujete na razuman način. Pored toga, verovatno ćete odvojiti klase koje ne treba da imaju pristup klasama definisanim u tekućem paketu.

Klasa kontroliše odakle se može pristupati njenim članovima. Kôd iz drugog paketa ne može da se pojavi, kaže: „Zdravo, ja sam Perin prijatelj" i očekuje da će dobiti pristup zaštićenim, paketnim i privatnim članovima klase **Pera**. Članu se dozvoljava pristup samo na sledeće načine:

1. Član proglasite javnim (**public**). Tada svako i sa svakog mesta može da mu pristupi.
2. Članu omogućite paketni pristup izostavljajući bilo kakve specifikatore pristupa, a ostale klase stavite u isti paket s njim. Tada ostale klase tog paketa mogu da pristupaju tom članu.
3. Kao što ćete videti u poglavlju *Ponovno korišćenje klasa* posvećenom nasleđivanju, potomak klase može da pristupa i zaštićenim i javnim članovima (ali ne i privatnim članovima) roditeljske klase. Članovima s paketnim pristupom može da pristupa samo ako su te dve klase u istom paketu. O nasleđivanju i zaštićenim članovima nemojte sada da brinete.
4. Obezbedite metode koje čitaju i menjaju željenu vrednost (koje se takođe nazivaju i metode pročitaj/postavi – engl. *get/set methods*). U OOP-u je to najpametniji pristup i suštinski je značajan za zrna Jave, kao što ćete videti u poglavlju *Grafička korisnička okruženja*.

public: interfejs za pristup

Kada upotrebite rezervisanu reč **public**, označavate da je definicija člana koji sledi odmah iza nje dostupna svima, posebno programeru klijentu koji koristi biblioteku. Pretpostavimo da definišete paket **desert** koji sadrži sledeću datoteku:

```
//: pristup/desert/Kolacic.java
// Pravi biblioteku
package pristup.desert;
```

```
public class Kolacic {
  public Kolacic() {
   System.out.println("Konstruktor klase Kolacic");
  }
  void zagrizi() { System.out.println("gric"); }
} ///:~
```

Zapamtite, datoteka klase koju će napraviti **Kolacic.java** mora da se nalazi u poddirektorijumu **desert** direktorijuma **pristup** (ukazuje na poglavlje *Kontrola pristupa* ove knjige) koji mora da bude unutar jednog od direktorijuma definisanih u promenljivoj CLASSPATH. Pogrešićete ako pomislite da će Java uvek da pogleda u tekući direktorijum kao jednu od početnih tačaka za pretraživanje. Ako ne navedete tačku kao jednu od putanja u svojoj promenljivoj CLASSPATH, Java neće uzeti u razmatranje tekući direktorijum.

Ako sada napravite program koji koristi klasu **Kolacic**:

```
//: pristup/Vecera.java
// Koristi biblioteku.
import pristup.desert.*;

public class Vecera {
  public static void main(String[] args) {
    Kolacic x = new Kolacic ();
    System.out.println("Konstruktor klase Kolacic ");
    //! x.zagrizi(); // Pristup nije moguć
  }
} /* Ispis:
Konstruktor klase Kolacic
* ///:~
```

možete da napravite objekat klase **Kolacic**, jer je njegov konstruktor javni i klasa je javna. (Koncept javnih klasa detaljnije ćemo razmotriti kasnije.) Međutim, metoda članica **zagrizi()** nije dostupna klasama iz datoteke **Vecera.java**, jer metoda **zagrizi()** omogućava pristup samo klasama iz paketa **desert**, pa će vas prevodilac sprečiti da je upotrebite.

Podrazumevani paket

Možda će vas iznenaditi kada otkrijete da će naredni program biti ispravno preveden iako na prvi pogled izgleda da krši pravila:

```
//: pristup/Torta.java
// Pristupa klasi iz druge (zasebne) jedinice za prevođenje (datoteke).

class Torta {
  public static void main(String[] args) {
    Pita x = new Pita();
    x.f();
  }
} /* Ispis:
Pita.f()
* ///:~
```

Druga datoteka treba da bude u istom direktorijumu:

```
//: pristup/Pita.java
// Druga klasa.

class Pita {
  void f() { System.out.println("Pita.f()"); }
} ///:~
```

U prvom trenutku, ove dve datoteke mogli biste da posmatrate kao potpuno odvojene pa ipak objekat klase **Torta** može da napravi objekat klase **Pita** i da pozove njenu metodu **f()**! (Vodite računa o tome da u svojoj promenljivoj CLASSPATH morate imati tačku da bi se ove datoteke pravilno prevele.) Pomislili biste da klasa **Pita** i **f()** imaju paketni pristup i da su stoga nedostupne klasi **Torta**. One *imaju* paketni pristup – taj deo zaključka je ispravan. Dostupne su klasi u datoteci **Torta.java** zato što se nalaze u istom direktorijumu i nemaju izričito ime paketa. Takve datoteke Java tretira kao deo „podrazumevanog paketa" za taj direktorijum i stoga omogućava paketni pristup svim ostalim klasama u istom direktorijumu.

private: to ne sme da se dira!

Rezervisana reč **private** označava da nekom članu niko ne sme da pristupa osim klase koja sadrži taj član, i to unutar njenih metoda. Takvi članovi se zovu privatni. Druge klase iz istog paketa ne mogu da pristupaju privatnim članovima, pa kao da ste izolovali klasu. S druge strane, nije neobično da paket pravi nekoliko ljudi, pa vam rezervisana reč **private** omogućava da slobodno menjate taj član bez brige kako će to da utiče na druge klase iz istog paketa.

Podrazumevani paketni pristup često obezbeđuje dovoljno sakrivanje; setite se, član kojem se pristupa paketno nije dostupan programeru klijentu koji tu klasu upotrebljava. To je dobro, jer obično i koristite podrazumevani pristup (a i njega dobijate ako zaboravite da dodate neki specifikator pristupa). Stoga ćete obično da razmišljate o pristupu članovima koje izričito želite da proglasite javnim i dostupnim programeru klijentu. U početku verovatno nećete često razmatrati upotrebu rezervisane reči **private**, jer možete da prođete i bez nje. Ispostavlja se, međutim, da je dosledna upotreba rezervisane reči **private** veoma važna, posebno za višenitni rad (kao što ćete videti u poglavlju *Paralelno izvršavanje*).

Sledi primer upotrebe rezervisane reči **private**:

```
//: pristup/SladoledSaVocem.java
// Prikazuje rezervisanu reč "private".

class SladoledSaVocem {
  private SladoledSaVocem() {}
  static SladoledSaVocem napraviSladoledSaVocem() {
    return new SladoledSaVocem();
  }
}
```

```
public class Sladoled {
  public static void main(String[] args) {
    //! SladoledSaVocem x = new SladoledSaVocem();
    SladoledSaVocem x = SladoledSaVocem.napraviSladoledSaVocem();
  }
} ///:~
```

Prethodni program predstavlja primer kada rezervisana reč **private** može zgodno da posluži: možete da poželite da kontrolišete pravljenje objekta i da sprečite nekog da direktno pristupi određenom konstruktoru (ili svim konstruktorima). U prethodnom primeru, objekat klase **SladoledSaVocem** ne možete da napravite preko konstruktora; umesto toga morate da pozovete metodu **napraviSladoledSaVocem()** koja će to da uradi umesto vas.[4]

Svaku metodu, za koju ste sigurni da je samo „pomoćna" metoda te klase, možete da proglasite privatnom kako je ne biste slučajno upotrebili negde u paketu – tako ćete predupredili eventualnu kasniju izmenu ili izbacivanje te metode. Ako metodu proglasite privatnom, tu mogućnost garantovano zadržavate.

Isto važi i za privatna polja unutar klase. Osim kada osnovna realizacija mora da bude vidljiva (što je mnogo manje verovatno nego što možete da pomislite), sva polja treba da proglasite privatnim. Međutim, to što je referenca na neki objekat privatna unutar klase, ne znači da neki drugi objekat ne može da ima javnu referencu na taj isti objekat. (Pseudonimi su obrađeni u dodacima knjige dostupnim na Mreži.)

protected: „pristup nasleđivanjem"

Da biste razumeli specifikator pristupa **protected**, pođimo malo unapred. Kao prvo, pomirite se s tim da ne morate da razumete ovaj deo da biste mogli da nastavite do nasleđivanja (poglavlje *Ponovno korišćenje klasa*). Ali, opšte slike radi, biće dat kratak opis i primer korišćenja rezervisane reči **protected**.

Rezervisana reč **protected** odnosi se na koncept koji se naziva *nasleđivanje*, u kome se uzima postojeća klasa i njoj dodaju novi članovi, bez izmena na postojećoj klasi (koju ćemo zvati *osnovna klasa*). Takođe, možete da izmenite ponašanje postojećih članica klase kada su u novoj klasi. Da biste nasledili određenu klasu, kažite da nova klasa proširuje (engl. *extends*) tu postojeću klasu, na primer:

```
class Naslednik extends Osnova {
```

Ostatak definicije klase je isti kao i ranije.

Ako napravite nov paket i ako nasledite neku klasu iz drugog paketa, imaćete pristup samo javnim članovima iz originalnog paketa. (Naravno, ako nasleđivanje uradite u *istom* paketu, moći ćete da pristupate svim njegovim članovima koji imaju paketni pristup.) Ponekad autor osnovne klase poželi da omogući pristup određenom članu iz izvedene klase, ali ne iz ostalih klasa. To se postiže rezervisanom rečju **protected**. **protected** daje i paketni pristup, tj. ostale klase u istom paketu mogu pristupati zaštićenim (engl. *protected*) članovima.

[4] U ovom slučaju nastaje još jedna posledica: pošto je definisan jedino podrazumevani konstruktor koji je privatni, nasleđivanje klase neće biti moguće. (O tome će kasnije još biti reči).

Ako ponovo razmotrite datoteku **Kolacic.java**, videćete da sledeća klasa *ne može* da pristupi članu **zagrizi( )** s paketnim pristupom:

```
//: pristup/CokoladniKeks.java
// Nije moguć pristup članu s paketnim pristupom u drugom paketu
import pristup.desert.*;

public class CokoladniKeks extends Kolacic {
  public CokoladniKeks () {
   System.out.println("Konstruktor klase CokoladniKeks");
  }
  public void njam()  {
      //! zagrizi(); // Nije moguć pristup metodi zagrizi
  }
  public static void main(String[] args) {
    CokoladniKeks x = new CokoladniKeks();
    x.njam();
  }
}  /* Ispis:
Konstruktor klase Kolacic
Konstruktor klase CokoladniKeks
* ///:~
```

Jedna od važnih osobina nasleđivanja je sledeća: ako metoda **zagrizi()** postoji u klasi **Kolacic**, onda ona postoji i u svakoj klasi nasleđenoj iz klase **Kolacic**. Ali pošto metoda **zagrizi()** ima paketni pristup i nalazi se u drugom paketu, ona nam nije dostupna u tekućem paketu. Mogli biste, naravno, da je proglasite za javnu, ali bi onda svi imali pristup, a to možda niste želeli. Ako klasu **Kolacic** izmenimo na sledeći način:

```
//: pristup/kolacic2/Kolacic.java
package pristup.kolacic2;

public class Kolacic {
  public Kolacic() {
    System.out.println("Konstruktor klase Kolacic");
  }
  protected void zagrizi() {
    System.out.println("gric");
  }
} ///:~
```

sada je metoda **zagrizi()** dostupna svima koji nasleđuju klasu **Kolacic**:

```
//: pristup/CokoladniKeks2.java
import pristup.kolacic2.*;

public class CokoladniKeks2 extends Kolacic {
  public CokoladniKeks2 () {
   System.out.println("Konstruktor klase CokoladniKeks2");
  }
```

```
  public void njam()  { zagrizi(); } // Zaštićena metoda
  public static void main(String[] args) {
    CokoladniKeks2 x = new CokoladniKeks2();
    x.njam();
  }
} /* Ispis:
Konstruktor klase Kolacic
Konstruktor klase CokoladniKeks2
zagrizi
* ///:-
```

Povedite računa o tome da **zagrizi( )** doduše ima paketni pristup, ali *nije* javna.

Vežba 4: (2) Pokažite da zaštićene metode imaju paketni pristup iako nisu javne.

Vežba 5: (2) Napravite klasu s javnim, privatnim, zaštićenim članovima i članovima (poljima i metodama) s paketnim pristupom. Napravite objekat te klase i proverite kakve poruke dobijate kada pokušate da pristupite svim članovima klase. Imajte u vidu da su klase u istom direktorijumu deo „podrazumevanog" paketa.

Vežba 6: (1) Napravite klasu sa zaštićenim podacima. Napravite drugu klasu u istoj datoteci s metodom koja rukuje zaštićenim podacima u prvoj klasi.

Interfejs i realizacija

Kontrola pristupa se često naziva i *sakrivanje realizacije*. Pakovanje podataka i metoda unutar klasa u kombinaciji sa skrivanjem realizacije često se naziva *kapsuliranje*.[5] Kao rezultat dobija se tip podataka sa svojstvenim obeležjima i određenim ponašanjem.

Kontrola pristupa nameće ograničenja unutar tipa podataka iz dva bitna razloga. Prvi je da odredite šta programeri klijenti smeju da koriste, a šta ne. Interne mehanizme možete slobodno da ugradite u strukturu, bez brige o tome da će programeri klijenti slučajno koristiti interne delove interfejsa.

To nas uvodi direktno u drugi razlog, a to je razdvajanje interfejsa i realizacije. Ako se struktura koristi u više programa, a programeri klijenti samo mogu da šalju poruke javnom interfejsu, tada možete da menjate sve što *nije* javno (znači sve to je prijateljsko, zaštićeno ili privatno), a da ne pokvarite klijentske programe.

Radi boljeg razumevanja, kao stil pravljenja klasa možete usvojiti da na početku stavite javne članove, a zatim zaštićene, paketne i privatne. Prednost ovog načina je to što korisnik klase može da krene od vrha i prvo vidi ono što mu je bitno (javne članove, jer njima može da pristupi izvan date datoteke) a zatim će se zaustaviti kada naiđe na ostale članove koji su deo interne realizacije:

```
//: pristup/RasporedPoNacinuPristupa.java

public class RasporedPoNacinuPristupa {
  public void jav1( ) { /* ... */ }
  public void jav2( ) { /* ... */ }
  public void jav3( ) { /* ... */ }
```

[5] Pod kapsuliranjem neki često podrazumevaju samo sakrivanje realizacije.

```
  private void priv1( ) { /* ... */ }
  private void priv2( ) { /* ... */ }
  private void priv3( ) { /* ... */ }
  private int i;
  // ...
} ///:~
```

Ovo će samo delimično olakšati čitanje jer su interfejs i realizacija i dalje zajedno. Odnosno, i dalje vidite izvorni kôd – realizaciju – jer se ona nalazi tu, u klasi. Pored toga, dokumentacioni komentari koje podržava alatka **Javadoc** umanjuju značaj čitljivosti programa za programera klijenta. Prikazivanje interfejsa korisniku klase u stvari je posao *čitača klasa* (engl. *class browser*), alatke čiji je zadatak da napravi pregled svih raspoloživih klasa i pokaže šta se s njima korisno može uraditi (tj. koje članice su dostupne). Prikazivanje dokumentacije razvojnog okruženja za Javu pomoću čitača Weba daje isti rezultat kao i čitač klasa.

Pristup klasama

Specifikatori pristupa u Javi takođe se mogu upotrebiti da odrede koje će klase *unutar* biblioteke biti dostupne njenom korisniku. Ako želite da klasa bude dostupna programeru klijentu, stavite rezervisanu reč **public** na čelo definicije klase. Time određujete da li programer klijent uopšte može da napravi objekat date klase.

Da bi kontrolisao pristup klasi, specifikator mora da se nađe pre rezervisane reči **class**. Stoga možete da napišete:

```
public class Spravica {
```

Ako je ime vaše biblioteke **pristup**, svaki programer klijent može da pristupi klasi **Spravica** kada napiše:

```
import pristup.Spravica;
```

ili

```
import pristup.*;
```

Međutim, postoje dodatna ograničenja:

1. U svakoj jedinici za prevođenje (datoteci) može da postoji samo jedna javna klasa. Ideja je da svaka jedinica za prevođenje ima samo jedan javni interfejs koji ta javna klasa predstavlja. Jedinica za prevođenje može da ima proizvoljan broj pomoćnih klasa s paketnim pristupom. Ako unutar jedinice za prevođenje imate više od jedne javne klase, prevodilac će prijaviti grešku.
2. Ime javne klase mora tačno da se poklapa sa imenom datoteke koja sadrži jedinicu za prevođenje, uključujući mala i velika slova. Znači, za klasu **Spravica** ime datoteke mora da bude **Spravica.java**, nikako **spravica.java** ili **SPRAVICA.java**. Ako se imena ne slažu, ponovo ćete dobiti poruku o grešci.
3. Moguće je, iako nije uobičajeno, da imate jedinicu za prevođenje u kojoj uopšte nema javne klase. U tom slučaju, datoteku možete da nazovete kako god želite (iako će proizvoljno imenovanje zbuniti one koji čitaju i održavaju taj kôd).

Šta ako unutar biblioteke **pristup** imate klasu koju koristite samo za zadatke koje izvršava klasa **Spravica** ili neka druga javna klasa iz te biblioteke? Nećete da pišete dokumentacije za programera klijenta i mislite kako ćete kasnije verovatno hteti da potpuno izmenite stvari i celu tu klasu izbacite i zamenite je nekom novom. Da biste zadržali tu mogućnost, morate osigurati da nijedan programer klijent ne postane zavisan od određenih detalja realizacije skrivenih unutar biblioteke **pristup**. Da biste to postigli, izostavite rezervisanu reč **public** ispred klase, pri čemu ona dobija paketni pristup. (Ta klasa može da se koristi samo unutar paketa.)

Vežba 7: (1) Napravite biblioteku **pristup** na osnovu prethodnih delova koda koji opisuju nju i klasu **Spravica**. Zatim napravite objekat klase **Spravica** u klasi koja nije deo paketa **pristup**.

Kada napravite klasu s paketnim pristupom, ima smisla da njena polja budu privatna – polja bi uvek trebalo da budu privatna – ali je po pravilu uputno dati metodama jednaki (paketni) pristup kao klasi. Pošto se klasa s paketnim pristupom obično upotrebljava samo unutar svog paketa, metode takve klase treba da učinite javnim samo ako morate, a to će vam reći prevodilac.

Zapamtite da klasa ne može da bude privatna (time bi postala dostupna samo sebi) niti zaštićena.[6] Za pristup klasi, stoga, imate samo dva izbora: paketni ili javni. Ako želite da niko drugi nema pristup nekoj klasi, sve konstruktore možete da proglasite privatnim, čime ćete sprečiti sve osim sebe da naprave objekat te klase. (I vi ćete to moći samo unutar statične članice te klase.) Evo primera:

```
//: pristup/Rucak.java
// Pokazuje specifikatore pristupa klasi. Klasu pretvarate
// u privatnu proglašavajući njene konstruktore privatnim:

class Supa1 {
  private Supa1() {}
  // (1) Dozvoljava pravljenje statičkom metodom:
  public static Supa1 napraviSupu() {
    return new Supa1();
  }
class Supa2 {
  private Supa2() {}
  // (2) Napravi statični objekat i vrati referencu ako je zahtevano.
  // ("Singularni" projektni obrazac):
  private static Supa2 ps1 = new Supa2();
  public static Supa2 pristup() {
    return ps1;
  }
  public void f() {}
}
```

[6] Unutrašnja (engl. *inner*) klasa može biti privatna ili zaštićena, ali to je specijalan slučaj. Te klase će biti objašnjene u poglavlju *Unutrašnje klase*.

```
// Dozvoljena je samo jedna javna klasa u datoteci:
public class Rucak {
  void testPrivate() {
    // Ovo ne može da se uradi! Konstruktor je privatan:
    //! Supa1 supa = new Supa1();
  }
  void testStaticni()  {
    Supa1 supa = Supa1.napraviSupu();
  }
  void testSingularni()   {
     Supa2.pristup().f();
  }
} ///:~
```

Do sada, većina metoda je vraćala ili **void** ili prost tip, pa definicija:

```
public static Supa1 napraviSupu() {
  return new Supa1();
}
```

isprva može malo da zbuni. Reč **Supa1** pre imena metode (**napraviSupu**) govori šta metoda vraća. Do sada je u ovoj knjizi to najčešće bila reč **void**, što znači da ne vraća ništa. Ali takode možete da vratite i referencu na objekat, što se ovde i dešava. Metoda vraća referencu na objekat klase **Supa1**.

Klase **Supa1** i **Supa2** pokazuju kako se proglašavanjem konstruktora privatnim sprečava direktno stvaranje objekata. Zapamtite, ako izričito ne napravite nijedan konstruktor, automatski će biti napravljen podrazumevani konstruktor (konstruktor bez argumenata). Ako sami napišete podrazumevani konstruktor, sprečićete da se on automatski napravi. Ukoliko ga zatim proglasite privatnim, niko neće moći da napravi objekat te klase. Ali, kako će sada iko tu klasu da koristi? Prethodni primer pokazuje dva načina. U **Supa1** napravljena je statična metoda koja pravi novi objekat klase **Supa1** i vraća referencu na njega. To može da bude korisno ako želite da sa objektom klase **Supa1** uradite još neke operacije pre nego što vratite referencu ili ako želite da vodite računa o broju napravljenih objekata klase **Supa1** (recimo da biste ograničili njihovu populaciju).

U **Supa2** koristi se *projektni obrazac*, o čemu više možete da pročitate u knjizi *Thinking in Patterns (with Java)*, na adresi *www.MindView.net*. Upotrebljeni obrazac u primeru naziva se „singularan", jer dozvoljava da bude napravljen samo jedan objekat. Objekat klase **Supa2** je napravljen kao statični i privatni član klase **Supa2**, pa postoji jedan i samo jedan i njemu možete da pristupite samo preko javne metode **pristup**().

Kao što je ranije napomenuto, ako ne stavite nijedan specifikator pristupa za klasu, podrazumeva se da je u pitanju paketni pristup. To znači da bilo koja druga klasa u paketu može da napravi objekat date klase, dok one izvan paketa to ne mogu. (Zapamtite da su sve datoteke koje se nalaze u istom direktorijumu i koje nemaju izričitu paketnu deklaraciju, eksplicitno svrstane u podrazumevani paket za taj direktorijum.) Ako je, međutim, statični član te klase javni, programer klijent još uvek može da pristupa statičnim članovima, iako ne može da napravi objekat te klase.

Vežba 8: (4) Prateći formu iz primera **Rucak.java**, napravite klasu pod imenom **UpravljacVezama** koja upravlja fiksnim nizom objekata klase **Veza.** Programer klijent ne sme da bude u mogućnosti da izričito pravi objekte klase **Veza**; trebalo bi da ih dobija samo preko statične metode iz klase **UpravljacVezama**. Kada klasi **UpravljacVezama** ponestane objekata, ona treba da vraća referencu **null.** Klase testirajte u metodi **main()**.

Vežba 9: (2) Napravite sledeću datoteku u direktorijumu **pristup/lokal** (koja bi trebalo da je u vašoj promenljivoj CLASSPATH):

```
///: pristup/lokal:UpakovanaKlasa.java
package pristup.lokal;

class UpakovanaKlasa {
   public UpakovanaKlasa () {
      System.out.println("Pravljenje upakovane klase");
   }
}
```

Zatim napravite sledeću datoteku u nekom drugom direktorijumu:

```
///: pristup/strana/Strana.java
package pristup.strana;
import pristup.lokal.*;

public class Strana {
   public static void main (String [] args) {
      UpakovanaKlasa uk = new UpakovanaKlasa();
   }
}
```

Objasnite zašto prevodilac prijavljuje grešku. Da li bi se išta promenilo ako bi klasa **Strana** postala deo paketa **pristup.lokal**?

Sažetak

U svakoj vezi bitno je da sve uključene strane poštuju izvesne granice. Kada pravite biblioteku, vi uspostavljate vezu s korisnikom te biblioteke – programerom klijentom – koji je takođe programer, ali koji koristi vašu biblioteku da bi napravio neku aplikaciju ili još veću biblioteku.

Kad ne bi bilo pravila, programeri klijenti mogli bi da rade šta im je volja sa svim članovima klase, čak i kada biste vi više voleli da ne rade direktno s nekim od njih. Sve bi bilo potpuno izloženo.

Ovo poglavlje se odnosilo na pravljenje klase da bi se formirala biblioteka: prvo, na koji način se klase pakuju u jednu biblioteku i drugo, način na koji klasa kontroliše pristup svojim članovima.

Procena je da projekti pisani na jeziku C počinju da se urušavaju kada dostignu između 50 K i 100 K redova koda. C ima jedinstven imenski prostor, pa imena počinju da se sudaraju te ih morate raspetljavati. U Javi, rezervisana reč **package**, tehnika imenovanja paketa i rezervisana reč **import**, obezbeđuju potpunu kontrolu nad imenima, čime se problem sa sudaranjem imena lako zaobilazi.

Postoje dva razloga za kontrolu pristupa članovima. Prvi je obezbediti da korisnici drže ruke podalje od delova koje ne bi smeli da diraju. Ti delovi su neophodni za interni rad klase, ali nisu deo interfejsa koji je potreban programerima klijentima. Zato proglašavanjem metoda i polja privatnim pravite uslugu programerima klijentima – lako mogu da razluče šta je za njih bitno, a šta mogu da ignorišu. Zbog toga bolje razumeju klase.

Drugi i najvažniji razlog kontrole pristupa jeste omogućiti projektantu biblioteke da promeni unutrašnji izgled klase bez brige o tome kako će to uticati na programera klijenta. Na primer, klasu isprva napravite na jedan način, a zatim otkrijete kako će se rekonstrukcijom koda bitno ubrzati izvršavanje. Ako su interfejs i realizacija jasno razdvojeni i zaštićeni, to možete da postignete bez primoravanja programera klijenta da ponovo pišu svoje programe. Kontrola pristupa osigurava da nijedan programer klijent ne postane zavisan od realizacije koja se nalazi u osnovi klase.

Kada imate mogućnost da menjate osnovnu realizaciju, ne samo da možete da unapređujete projekat, već i da pravite greške. Bez obzira na to koliko pažljivo planirate i projektujete, napravićete greške. Ako znate da greške neće izazvati mnogo štete, bićete više raspoloženi za eksperimente, brže ćete učiti i brže završavati projekte.

Korisnik *vidi* javni interfejs, pa je to, prilikom analize i projektovanja programa, najvažniji deo klase koji treba da uradite kako valja. Čak vam je i tu data izvesna sloboda da menjate. Ako interfejs ne „pogodite“ iz prve, uvek možete da *dodate* druge metode, dokle god ne izbacite neke koje je programer klijent već upotrebio u svom programu.

Obratite pažnju na to da se kontrola pristupa usredsređuje na odnos – i neku vrstu komunikacije – između autora biblioteke i njenih spoljnih klijenata. Ima mnogo situacija u kojima to nije slučaj. Primera radi, sami pišete sav kôd ili radite zajedno s malom ekipom i sve što napišete ide u isti paket. U tim situacijama potrebna je drugačija komunikacija, pa kruto pridržavanje pravila pristupa može da smeta. Tada bi podrazumevani (paketni) pristup mogao da bude optimalan.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking In Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.net*

Ponovno korišćenje klasa

Jedna od najlepših mogućnosti u Javi jeste ponovno korišćenje koda. Ali, da biste u tom poslu bili revolucionarni, sa kodom morate umeti da uradite mnogo više od pukog kopiranja i menjanja.

U PROCEDURALNIM JEZICIMA KAO ŠTO JE C, KÔD SE PONOVO KORISTI TAKO ŠTO SE KOPIRA i izmeni, ali se takav pristup i nije baš najbolje pokazao. Kao i sve ostalo u Javi, rešenje se vrti oko klase. Kôd ponovo koristite tako što pravite nove klase ali, umesto da ih pravite „od nule", uzimate postojeće klase koje je neko već napravio i očistio od grešaka.

Rešenje je da se upotrebe klase bez menjanja postojećeg kôda. U ovom poglavlju naučićete dva načina da to postignete. Prvi je prilično jasan: pravite objekte postojeće klase unutar nove klase. To se naziva *kompozicija*, jer je nova klasa sastavljena od objekata postojećih klasa. Ponovo se koristi funkcionalnost koda, a ne njegova forma.

Drugi pristup je suptilniji. Nova klasa se pravi kao *tip* postojeće klase. Vi bukvalno uzimate oblik postojeće klase, dodajete joj kôd i ništa ne menjate na njoj samoj. Ova tehnika se naziva *nasleđivanje* i prevodilac tu obavlja veći deo posla. Nasleđivanje (engl. *inheritance*) predstavlja jedan od kamena temeljaca objektno orijentisanog programiranja i podrazumeva još neke postupke koji će biti istraženi u poglavlju *Polimorfizam*.

Ispostavlja se da je dobar deo sintakse i osobina kompozicije i nasleđivanja sličan (što ima smisla, jer se oba koriste za pravljenje novih tipova od postojećih). U ovom poglavlju učićete o tim mehanizmima ponovnog korišćenja koda.

Sintaksa kompozicije

Do ovog mesta u knjizi često smo koristili kompoziciju – smeštanje referenci na objekte unutar nove klase. Pretpostavimo, na primer, da želite da napravite objekat koji čuva nekoliko objekata tipa **String**, nekoliko prostih tipova i objekat neke druge klase. Kod složenih tipova, unutar nove klase stavljate reference, dok proste tipove direktno definišete:

```
//: ponovnaupotreba/Prskalica.java
//  Ponovna upotreba koda kompozicijom.

class Izvor {
  private String s;
  Izvor() {
    System.out.println("Izvor()");
    s = "Konstruisan";
  }
  public String toString() { return s; }
}

public class Prskalica {
  private String ventil1, ventil2, ventil3, ventil4;
  private Izvor izvor = new Izvor();
```

```
  private int i;
  private float f;
  public String toString() {
    return
      "ventil1 = " + ventil1 + " " +
      "ventil2 = " + ventil2 + " " +
      "ventil3 = " + ventil3 + " " +
      "ventil4 = " + ventil4 + "\n" +
      "i = " + i + " " + "f = " + f + " " +
      "izvor = " + izvor;
  }
  public static void main(String[] args) {
    Prskalica prskalice = new Prskalica();
    System.out.println(prskalice);
  }
} /* Ispis:
Izvor()
ventil1 = null ventil2 = null ventil3 = null ventil4 = null
i = 0 f = 0.0 izvor = Konstruisan
* ///:~
```

Jedna od metoda definisanih u obe klase je posebna: to je metoda **toString()**. Svaki objekat ima metodu **toString()** koja se poziva u posebnim situacijama, ako prevodiocu treba objekat klase **String**, a umesto toga mu je prosleđen neki drugi objekat. Stoga u izrazu u **Prskalica.toString()**:

```
"izvor = " + izvor;
```

prevodilac primećuje kako pokušavate da na objekat tipa **String** (**"izvor = "**) nadovežete objekat tipa **Izvor**. Pošto na objekat tipa **String** možete da nadovežete samo drugi takav objekat, on kaže: „pretvoriću **Izvor** u tip **String** pozivanjem metode **toString()**". Nakon toga, on može da kombinuje dva objekta klase **String** i da rezultat prosledi metodi **System.out. println**() (ili prethodno u knjizi definisanim statičnim metodama **print**() i **printnb**()). Kad god želite da klasi koju pravite omogućite takvo ponašanje, samo napišite metodu **toString()**.

Promenljive prostog tipa koje su polja klase, automatski su inicijalizovane vrednošću nula, kao što je napomenuto u poglavlju *Sve je objekat*. Međutim, reference na objekte inicijalizovane su vrednošću **null** i ako preko bilo koje od tih referenci pokušate da pozovete neku metodu, izazvaćete izuzetak – grešku tokom izvršavanja. Dobro je što ipak možete da ispišete **null** referencu, a da se pri tom ne javi izuzetak.

Ima smisla to što prevodilac ne pravi podrazumevani objekat za svaku referencu, jer bi to u mnogim slučajevima izazvalo nepotrebne režijske troškove. Ako želite da reference budu inicijalizovane, to morate da uradite sami:

1. Na mestu gde se objekti definišu. To znači da će uvek biti inicijalizovani pre nego što konstruktor bude pozvan.
2. U konstruktoru te klase.

3. Neposredno pre nego što vam je zaista taj objekat potreban. To se često naziva i *lenja inicijalizacija*. Ona može da smanji režijske troškove onda kada je pravljenje objekta skupo, a ne treba ga praviti svaki put.
4. *Inicijalizacijom instance* (primerka).

Sva četiri pristupa prikazana su ovde:

```
//: ponovnaupotreba/Kada.java
// Inicijalizacija u konstruktoru i kompozicija.
import static net.mindview.util.Print.*;

class Sapun {
  private String s;
  Sapun() {
    print("Sapun()");
    s = "Konstruisan";
  }
  public String toString() { return s; }
}

public class Kada {
  private String     // Inicijalizacija na mestu definicije:
    s1 = "Srećan",
    s2 = "Srećan",
    s3, s4;
  private Sapun sapuncic;
  private int i;
  private float igracka;
  public Kada() {
    print("Unutar klase Kada()");
    s3 = "Radostan";
    igracka = 3.14f;
    sapuncic = new Sapun();
  }
  // Inicijalizacija instance (primerka):
  { i = 47; }
  public String toString()  {
    if(s4 == null)  // Odložena inicijalizacija:
      s4 = "Radostan";
    return
      "s1 = " + s1 + "\n" +
      "s2 = " + s2 + "\n" +
      "s3 = " + s3 + "\n" +
      "s4 = " + s4 + "\n" +
      "i = " + i + "\n" +
      "igracka = " + igracka + "\n" +
      "sapuncic = " + sapuncic;
  }
  public static void main(String[] args) {
    Kada b = new Kada();
```

```
    print(b);
  }
} /* Ispis:
Unutar klase Kada()
Sapun()
s1 = Srećan
s2 = Srećan
s3 = Radostan
s4 = Radostan
i = 47
igracka = 3.14
sapuncic = Konstruisan
* ///:~
```

Obratite pažnju na to da se u konstruktoru klase **Kada** naredba izvršava pre nego što se uradi bilo koja inicijalizacija. Kada inicijalizaciju ne uradite na mestu definicije, ne postoji garancija da će objekat ili referenca biti inicijalizovani pre nego što im pošaljete poruku – izuzev neizbežnog izuzetka prilikom izvršavanja.

Kada se pozove **toString**(), on popunjava **s4** tako da su sva polja pravilno inicijalizovana u trenutku njihovog korišćenja.

Vežba 1: (2) Napravite jednostavnu klasu. Unutar druge klase definišite referencu objekta prve klase. Za instanciranje tog objekta koristite lenju inicijalizaciju.

Sintaksa nasleđivanja

Nasleđivanje je sastavni deo Jave (i uopšte objektno orijentisanih jezika). U stvari, kada pravite klasu, ispostavlja se da uvek primenjujete nasleđivanje. Ako eksplicitno ne nasledite neku klasu, implicitno ćete naslediti Javinu podrazumevanu korensku klasu **Object.**

Sintaksa kompozicije je očigledna, ali se za nasleđivanje upotrebljava posebna sintaksa. Pri nasleđivanju kažete „nova klasa je kao ta stara klasa". U programu to navodite pre početne vitičaste zagrade tela klase, pišući rezervisanu reč **extends** iza koje sledi ime *osnovne klase*. Kada to učinite, automatski preuzimate sva polja i metode osnovne klase. Evo primera:

```
//: ponovnaupotreba/Deterdzent.java
// Sintaksa i osobine nasleđivanja.
import static net.mindview.util.Print.*;

class Cistac {
  private String s = "Cistac";
  public void dodaj(String a) { s += a; }
  public void razredi() { dodaj(" razredi()"); }
  public void sipaj() { dodaj(" sipaj()"); }
  public void ribaj() { dodaj(" ribaj()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cistac x = new Cistac();
```

```
    x.razredi(); x.sipaj(); x.ribaj();
    print(x);
  }
}
public class Deterdzent extends Cistac {
  // Promeni metodu:
  public void ribaj() {
    dodaj(" Deterdzent.ribaj()");
    super.ribaj(); // Poziva verziju iz osnovne klase
  }
  // Dodaje metode klasi:
  public void zapeni() { dodaj(" zapeni()"); }
  // Testira novu klasu:
  public static void main(String[] args) {
    Deterdzent x = new Deterdzent();
    x.razredi();
    x.sipaj();
    x.ribaj();
    x.zapeni();
    print(x);
    print("Testiranje osnovne klase:");
    Cistac.main(args);
  }
} /* Ispis:
Cistac razredi() sipaj() Deterdzent.ribaj() ribaj() zapeni()
Testiranje osnovne klase:
Cistac razredi() sipaj() ribaj()
* ///:~
```

Ovim se pokazuje više osobina. Prvo, u metodi **dodaj** klase **Cistac**, objekti klase **String** se nadovezuju na sadržaj promenljive **s** pomoću operatora +=. Taj operator (kao i operator +) autori Jave su „preklopili" tako da radi i sa objektima klase **String**.

Drugo, obe klase, **Cistac** i **Deterdzent** sadrže metodu **main**(). Svaka od vaših klasa može da sadrži metodu **main**(); tehnikom stavljanja metode **main**() u svaku klasu, omogućavate lako testiranje svake klase. Kada ga završite, metodu **main**() ne morate da uklanjate; možete da je ostavite za kasnije testiranje.

I kada imate puno klasa u programu, biće pozvana samo metoda **main**() klase pozvane s komandne linije. Stoga će u ovom slučaju biti pozvana metoda **Deterdzent.main**() ako napišete **java Deterdzent**. Međutim, takođe možete da napišete **java Cistac** čime pozivate metodu **Cistac.main**(), iako klasa **Cistac** nije javna. Čak i ako klasa ima paketni pristup, javna metoda **main**() je dostupna.

U ovom slučaju obratite pažnju na to da metoda **Deterdzent.main**() izričito poziva metodu **Cistac.main**(), pri čemu joj prosleđuje iste argumente koje je sama dobila s komandne linije (u principu, možete da prosledite bilo koji niz elemenata tipa **String**).

Važno je da sve metode klase **Cistac** budu javne. Setite se da se, ako izostavite specifikator pristupa, podrazumeva da član ima paketni pristup koji dozvoljava pristup samo članovima istog paketa. Stoga bi, ako ne navedete specifikator pristupa, svaki *član tog paketa* mogao da koristi te metode. Za klasu **Deterdzent**, na primer, ne bi bilo problema.

Međutim, ako bi neka klasa iz nekog drugog paketa nasledila klasu **Cistac**, ona bi mogla da pristupa samo njenim javnim članovima. Dakle, da biste omogućili nasleđivanje klase, opšte pravilo je da sva polja budu privatna, a sve metode javne (izvedena klasa takođe može da pristupa zaštićenim članovima; o tome nešto kasnije). Naravno da u određenim slučajevima morate učiniti izvesna prilagođavanja, ali ovo je korisna smernica.

Klasa **Cistac** ima grupu metoda u svom interfejsu: **dodaj()**, **razredi()**, **sipaj()**, **ribaj()** i **toString()**. Pošto je klasa **Deterdzent** *izvedena* iz klase **Cistac** (pomoću rezervisane reči **extends**), njen interfejs automatski sadrži i navedene metode, iako ne vidite da su one u njoj izričito definisane. Znači da nasleđivanje možete da posmatrate kao ponovno korišćenje klase.

Na primeru metode **ribaj()** vidi se da je moguće menjati metodu koja je već bila definisana u osnovnoj klasi. U ovom slučaju, iz nove verzije je pozvana i metoda iz osnovne klase. Međutim, u metodi **ribaj()** nije ispravno da se pozove samo metoda **ribaj()**, jer bi to bio rekurzivan poziv, a to niste nameravali. Da bi rešila ovaj problem, Javina rezervisana reč **super** označava natklasu (engl. *superclass*) koju tekuća klasa nasleđuje. Stoga naredba **super.ribaj()** poziva verziju metode **ribaj()** iz osnovne klase.

Pri nasleđivanju ne morate koristiti samo metode osnovne klase. Izvedenoj klasi možete da dodajete nove metode na isti način na koji biste to radili i s bilo kojom drugom klasom: definišite ih. Primer za to je metoda **zapeni()**.

U metodi **Deterdzent.main()** vidite da preko objekta klase **Deterdzent** možete da pozivate metode koje su raspoložive za klasu **Cistac** i one za klasu **Deterdzent** (na primer, **zapeni()**).

Vežba 2: (2) Iz klase **Deterdzent** nasledite novu klasu. Redefinišite metodu **ribaj()**, a dodajte i novu metodu pod imenom **sterilisi()**.

Inicijalizacija osnovne klase

Pošto su sada u priču uključene dve klase – osnovna i izvedena – umesto samo jedne, pomalo zbunjuje pomisao na to kako izgleda objekat koji se dobija od izvedene klase. Spolja izgleda kao da nova klasa ima isti interfejs kao osnovna klasa, možda uz poneku dodatnu metodu ili polje. Ali nasleđivanje ne kopira samo interfejs osnovne klase. Kada napravite objekat izvedene klase, on sadrži *podobjekat* (engl. *subobject*) osnovne klase. Taj podobjekat izgleda isto kao da ste napravili sâm objekat osnovne klase. Jedino je, spolja gledano, podobjekat osnovne klase upakovan u objekat izvedene klase.

Naravno, neophodno je da podobjekat osnovne klase bude pravilno inicijalizovan i postoji samo jedan način da se to i garantuje: da se inicijalizacija obavi u konstruktoru pozivanjem konstruktora osnovne klase. Taj konstruktor ima sve potrebno znanje i privilegije da inicijalizuje osnovnu klasu. Java automatski dodaje poziv konstruktora osnovne klase u konstruktor izvedene klase. Naredni primer pokazuje rad s tri nivoa nasleđivanja:

```
//: ponovnaupotreba/Karikatura.java
// Poziv konstruktora pri nasleđivanju.
import static net.mindview.util.Print.*;
```

```
class UmetnickoDelo {
  UmetnickoDelo() { print("Konstruktor klase UmetnickoDelo"); }
}

class Crtez extends UmetnickoDelo {
  Crtez() { print("Konstruktor klase Crtez");  }
}

public class Karikatura extends Crtez {
  public Karikatura() { print("Konstruktor klase Karikatura");  }
  public static void main(String[] args) {
    Karikatura x = new Karikatura();
  }
} /* Ispis:
Konstruktor klase UmetnickoDelo
Konstruktor klase Crtez
Konstruktor klase Karikatura
 ///:~
```

Vidite da se konstrukcija obavlja od osnovne ka izvedenim klasama, pa je osnovna klasa inicijalizovana pre nego što konstruktor izvedene klase može da joj pristupi. Čak i ako za klasu **Karikatura** ne napravite konstruktor, prevodilac će umesto vas napraviti podrazumevani konstruktor koji će pozvati konstruktor osnovne klase.

Vežba 3: (2) Dokažite prethodnu tvrdnju (da prevodilac pravi podrazumevane konstruktore umesto vas).

Vežba 4: (2) Dokažite da se konstruktori osnovne klase (a) uvek pozivaju i (b) pozivaju pre konstruktora izvedene klase.

Vežba 5: (1) Napravite dve klase, **A** i **B**, sa podrazumevanim konstruktorima (bez argumenata) koje javljaju svoje postojanje. Iz klase **A** nasledite novu klasu **C** i u njoj napravite objekat član klase **B**. Nemojte da pravite konstruktor klase **C**. Napravite objekat te klase i posmatrajte rezultate.

Konstruktori sa argumentima

U prethodnom primeru su korišćeni podrazumevani konstruktori; odnosno, oni nisu imali nikakve argumente. Prevodiocu je lako da ih poziva, jer nema sumnje koje argumente treba proslediti. Ako osnovna klasa nema podrazumevani konstruktor ili ako želite da pozovete konstruktor osnovne klase kome su potrebni argumenti, morate izričito pozvati konstruktor osnovne klase koristeći rezervisanu reč **super** i proslediti mu odgovarajuću listu argumenata:

```
//: ponovnaupotreba/Sah.java
// Nasleđivanje, konstruktori i argumenti.
import static net.mindview.util.Print.*;
```

```
class Igra {
  Igra(int i) {
    print("Konstruktor klase Igra");
  }
}

class IgraNaTabli extends Igra {
  IgraNaTabli(int i) {
    super(i);
    print("Konstruktor klase IgraNaTabli");
  }
}

public class Sah extends IgraNaTabli {
  Sah() {
    super(11);
    print("Konstruktor klase Sah");
  }
  public static void main(String[] args) {
    Sah x = new Sah();
  }
} /* Ispis:
Konstruktor klase Igra
Konstruktor klase IgraNaTabli
Konstruktor klase Sah
* ///:~
```

Ako u konstruktoru klase **IgraNaTabli()** ne pozovete konstruktor osnovne klase, prevodilac će se žaliti da ne može da pronađe konstruktor oblika **Igra()**. Pored toga, prvo se *mora* pozvati konstruktor osnovne klase, i to najpre *morate* da uradite u konstruktoru izvedene klase. (Prevodilac će vas podsetiti ako pogrešite.)

Vežba 6: (1) Pomoću datoteke **Sah.java** dokažite tvrdnje iz prethodnog pasusa.

Vežba 7: (1) Izmenite vežbu 5 tako da **A** i **B** umesto podrazumevanih konstruktora imaju konstruktore sa argumentima. Napišite konstruktor klase **C** i svu inicijalizaciju obavite unutar njega.

Vežba 8: (1) Napravite osnovnu klasu koja ima samo jedan nepodrazumevani konstruktor i izvedenu klasu koja ima i podrazumevani i nepodrazumevani konstruktor. Pozovite konstruktor osnovne klase iz konstruktora izvedene klase.

Vežba 9: (2) Napravite klasu pod imenom **Koren** koja sadrži po jednu instancu klasa (koje takođe treba da napravite) pod imenima **Komponenta1**, **Komponenta2** i **Komponenta3**. Iz klase **Koren** izvedite klasu **Stabljika** koja takođe ima po instancu svake komponente. Svaka klasa treba da ima podrazumevani konstruktor koji ispisuje poruku o njoj.

Vežba 10: (1) Izmenite prethodnu vežbu tako da svaka klasa ima i nepodrazumevani konstruktor.

Delegiranje

Java ne podržava direktno treći odnos koji se naziva *delegiranje*. To je nešto na pola puta između nasleđivanja i kompozicije, pošto u klasu koju pravite smeštate objekat član (kao kompozicija), ali u novoj klasi istovremeno eksponirate sve metode tog objekta člana (kao nasleđivanje). Na primer, svemirskom brodu treba modul za upravljanje:

```
//: ponovnaupotreba/UpravljačkiUređajiSvemirskogBroda.java

public class UpravljačkiUređajiSvemirskogBroda  {
   void naviše(int brzina)  {}
   void naniže(int brzina)  {}
   void nalevo(int brzina)  {}
   void nadesno(int brzina)  {}
   void napred(int brzina)  {}
   void nazad(int brzina)  {}
   void turboPotisak(int brzina)  {}
} ///: ~
```

Jedan od načina da se napravi svemirski brod bilo bi nasleđivanje:

```
//: ponovnaupotreba/SvemirskiBrod.java

public class SvemirskiBrod extends UpravljačkiUređajiSvemirskogBroda  {
   private String ime;
   public SvemirskiBrod(String ime)  {  this.ime = ime; }
   public String toString()  {  return ime; }
   public static void main(String[] args) {
     SvemirskiBrod štit  = new SvemirskiBrod("PVO  tit");
     štit.napred(100);
   }
} ///:~
```

Naravno, **SvemirskiBrod** u stvari nije „jedan od objekata" klase **UpravljačkiUređajiSvemirskogBroda**, čak i ako objektu klase **SvemirskiBrod**, primera radi, možete „narediti" da ide **napred**(). Tačnije je reći da **SvemirskiBrod** *sadrži* **UpravljačkiUređajiSvemirskogBroda**. Istovremeno, u objektu klase **SvemirskiBrod** eksponirane su sve metode objekta klase **UpravljačkiUredajiSvemirskogBroda**.

Ovu nedoumicu rešava delegiranje:

```
//: ponovnaupotreba/DelegiranjeSvemirskogBroda.java

public class DelegiranjeSvemirskogBroda  {
  private String ime;
  private UpravljačkiUređajiSvemirskogBroda upravljačkiUređaji =
    new UpravljačkiUređajiSvemirskogBroda();
  public DelegiranjeSvemirskogBroda(String ime)  {
    this.ime  = ime;
  }
  //  Delegirane metode:
  public void nazad(int brzina)  {
```

```
    upravljačkiUređaji.nazad(brzina);
  }
  public void naniže(int brzina)  {
    upravljačkiUređaji.naniže(brzina);
  }
  public void napred(int brzina)  {
    upravljačkiUređaji.napred(brzina);
  }
  public void nalevo(int brzina)  {
    upravljačkiUređaji.nalevo(brzina);
  }
  public void nadesno(int brzina)  {
    upravljačkiUređaji.nadesno(brzina);
  }
  public void turboPotisak(int brzina)  {
    upravljačkiUređaji.turboPotisak(brzina);
  }
  public void naviše(int brzina)  {
    upravljačkiUređaji.naviše(brzina);
  }
  public static void main(String[] args) {
    DelegiranjeSvemirskogBroda štit  =
      new DelegiranjeSvemirskogBroda("PVO  tit");
  štit.napred(100);
  }
} ///:~
```

Vidi se kako su metode prosleđene pripadnom objektu tipa **upravljačkiUređaji**, pa je interfejs zato jednak kao pri nasleđivanju. Međutim, delegiranje omogućuje više kontrole, pošto u objekat član možete preneti proizvoljan podskup svih metoda.

Iako Java ne podržava delegiranje, ima mnogo razvojnih alatki koje ga podržavaju. Recimo, gornji primer je automatski generisan pomoću razvojnog okruženja JetBrains Idea.

Vežba 11: (3) Izmenite **Deterdzent.java** tako da se u njoj koristi delegiranje.

Kombinovanje kompozicije i nasleđivanja

Kompozicija i nasleđivanje se često koriste zajedno. Naredni primer pokazuje pravljenje složenije klase, korišćenjem kompozicije i nasleđivanja, kao i neophodne inicijalizacije konstruktorima:

```
//: ponovnaupotreba/PostavkaStola.java
// Kombinovanje kompozicije i nasleđivanja.
import static net.mindview.util.Print.*;
class Tanjir {
  Tanjir(int i) {
    print("Konstruktor klase Tanjir");
  }
}
```

```
class TanjirZaVeceru extends Tanjir {
  TanjirZaVeceru(int i) {
    super(i);
    print("Konstruktor klase TanjirZaVeceru");
  }
}

class Pribor {
  Pribor(int i) {
    print("Konstruktor klase Pribor");
  }
}

class Kasika extends Pribor {
  Kasika(int i) {
    super(i);
    print("Konstruktor klase Kasika");
  }
}

class Viljuska extends Pribor {
  Viljuska(int i) {
    super(i);
    print("Konstruktor klase Viljuska");
  }
}
class Noz extends Pribor {
  Noz(int i) {
    super(i);
    print("Konstruktor klase Noz");
  }
}
// Kulturan način da se nešto uradi:
class Obicaj {
  Obicaj(int i) {
    print("Konstruktor klase Obicaj");
  }
}

public class PostavkaStola extends Obicaj {
  private Kasika ks;
  private Viljuska vilj;
  private Noz nz;
  private TanjirZaVeceru tv;
  public PostavkaStola(int i) {
    super(i + 1);
    ks = new Kasika(i + 2);
    vilj = new Viljuska(i + 3);
    nz = new Noz(i + 4);
    tv = new TanjirZaVeceru(i + 5);
    print("Konstruktor klase PostavkaStola");
```

```
  }
  public static void main(String[] args) {
    PostavkaStola x = new PostavkaStola(9);
  }
} /* Ispis:
Konstruktor klase Obicaj
Konstruktor klase Pribor
Konstruktor klase Kasika
Konstruktor klase Pribor
Konstruktor klase Viljuska
Konstruktor klase Pribor
Konstruktor klase Noz
Konstruktor klase Tanjir
Konstruktor klase TanjirZaVeceru
Konstruktor klase PostavkaStola
* ///:~
```

Iako vas prevodilac prisiljava da inicijalizujete osnovne klase i zahteva da to učinite odmah na početku konstruktora, on ne vodi računa o tome da li ste inicijalizovali objekte članove, pa morate sami o tome voditi računa.

Prilično je zadivljujuće kako su klase jasno razdvojene. Nije vam čak potreban ni izvorni kôd metoda da biste ih ponovo koristili. U većini slučajeva treba samo da uvezete paket. (Ovo važi i za nasleđivanje i za kompoziciju.)

Garantovanje pravilnog čišćenja

Java nema ništa slično *destruktoru* iz C++-a, tj. nema metodu koja se automatski poziva pri uništavanju objekta, verovatno zato što je u Javi praksa da objekte zaboravite umesto da ih uništavate i da pustite sakupljač smeća da povrati memoriju kada je potrebno.

Često je to sasvim u redu, ali u nekim slučajevima klasa može da obavlja izvesne aktivnosti koje zahtevaju čišćenje. Kao što je pomenuto u poglavlju *Inicijalizacija i čišćenje*, vi ne možete znati kada će sakupljač smeća biti pozvan ili da li će uopšte biti pozvan. Stoga, ako je za neku klasu potrebno čišćenje, morate izričito da napišete posebnu metodu i obezbedite da programer klijent zna da mora pozvati tu metodu. Povrh svega – kao što će biti opisano u poglavlju *Obrada grešaka pomoću izuzetaka* – od izuzetaka morate da se zaštitite tako što ćete odgovarajuću proceduru za čišćenje ugraditi u blok **finally**.

Razmotrimo primer sistema za projektovanje pomoću računara koji na ekranu crta slike:

```
//: ponovnaupotreba/CADSistem.java
// Obezbeđivanje pravilnog čišćenja
package ponovnaupotreba;
import static net.mindview.util.Print.*;

class Oblik {
  Oblik(int i) { print("Konstruktor klase Oblik"); }
  void ciscenje() { print("Čišćenje klase Oblik"); }
}
```

```
class Krug extends Oblik {
  Krug(int i) {
    super(i);
    print("Crtanje kruga");
  }
  void ciscenje() {
    print("Brisanje kruga");
    super.ciscenje();
  }
}

class Trougao extends Oblik {
  Trougao(int i) {
    super(i);
    print("Crtanje trougla");
  }
  void ciscenje() {
    print("Brisanje trougla");
    super.ciscenje();
  }
}

class Linija extends Oblik {
  private int pocetak, kraj;
  Linija(int pocetak, int kraj) {
    super(pocetak);
    this.pocetak = pocetak;
    this.kraj = kraj;
    print("Crtanje linije: " + pocetak + ", " + kraj);
  }
  void ciscenje() {
    print("Brisanje linije: " + pocetak + ", " + kraj);
    super.ciscenje();
  }
}

public class CADSistem extends Oblik {
  private Krug c;
  private Trougao t;
  private Linija[] linije = new Linija[3];
  public CADSistem(int i) {
    super(i + 1);
    for(int j = 0; j < linije.length; j++)
      linije[j] = new Linija(j, j*j);
    c = new Krug(1);
    t = new Trougao(1);
    print("Kombinovani konstruktor");
  }
  public void ciscenje() {
    print("CADSistem.ciscenje()");
```

```
    // Poredak čišćenja je inverzan u odnosu na poredak inicijalizacije
    t.ciscenje();
    c.ciscenje();
    for(int i = linije.length - 1; i >= 0; i--)
      linije[i].ciscenje();
    super.ciscenje();
  }
  public static void main(String[] args) {
    CADSistem x = new CADSistem(47);
    try {
      // Kôd i obrada izuzetaka...
    } finally {
      x.ciscenje();
    }
  }
} /* Ispis:
Konstruktor klase Oblik
Konstruktor klase Oblik
Crtanje linije: 0, 0
Konstruktor klase Oblik
Crtanje linije: 1, 1
Konstruktor klase Oblik
Crtanje linije: 2, 4
Konstruktor klase Oblik
Crtanje kruga
Konstruktor klase Oblik
Crtanje trougla
Kombinovani konstruktor
CADSistem.ciscenje()
Brisanje trougla
Čišćenje klase Oblik
Brisanje kruga
Čišćenje klase Oblik
Brisanje linije: 2, 4
Čišćenje klase Oblik
Brisanje linije: 1, 1
Čišćenje klase Oblik
Brisanje linije: 0, 0
Čišćenje klase Oblik
Čišćenje klase Oblik
*///:~
```

U ovom sistemu sve je neka vrsta **Oblika** (koji je i sam tipa **Object**, jer implicitno nasleđuje korensku klasu). Svaka klasa redefiniše metodu **ciscenje**() klase **Oblik** i pored ostalog, poziva i verziju te metode iz osnovne klase pomoću rezervisane reči **super**. Pojedine vrste klase **Oblik** – **Krug**, **Trougao** i **Linija** – imaju svoje konstruktore koji „crtaju", premda bilo koja metoda tokom životnog veka objekta može da uradi nešto što zahteva čišćenje. Svaka klasa ima svoju metodu **ciscenje**() koja vraća stvari bez ikakve veze s memorijom, u stanje u kome su bile pre nego što je objekat napravljen.

U metodi **main**() možete da primetite dve nove rezervisane reči koje neće biti detaljno objašnjene sve do poglavlja *Obrada grešaka pomoću izuzetaka*, a to su **try** i **finally**. Rezervisana reč **try** naznačava da je blok koji sledi (unutar vitičastih zagrada) *zaštićeni region*, što znači da ima poseban tretman. Jedan deo tog posebnog tretmana odnosi se na kôd unutar bloka **finally** koja sledi iza tog zaštićenog regiona. Taj kôd se *uvek* izvršava, bez obzira na to kako se iz bloka **try** izlazi. (Pri obradi izuzetaka, blok **try** može se napustiti na više neuobičajenih načina.) U ovom slučaju, blok **finally** kaže: „Za **x** uvek pozovi metodu **ciscenje**(), bez obzira na to šta se događa".

Ukoliko jedan podobjekat zavisi od drugog, zapamtite da u svojoj metodi za čišćenje (u ovom slučaju, metodi **ciscenje**()) takođe morate da vodite računa o poretku čišćenja osnovne klase i objekata članova. Po pravilu, treba da pratite princip koji je uveo prevodilac jezika C++ za svoje destruktore: prvo obavite sve čišćenje koje je specifično za vašu klasu, i to redosledom koji je obrnut od redosleda pravljenja. (U opštem slučaju, za to je potrebno da elementi osnovne klase i dalje budu u životu.) Zatim pozovite metodu za čišćenje osnovne klase, kao što je u našem primeru pokazano.

U mnogim slučajevima čišćenje nije problem i možete da pustite sakupljača smeća da obavi posao. Ali, kada je izričito čišćenje potrebno, njemu morate da pristupite marljivo i pažljivo, pošto na sakupljanje smeća ne možete previše da se oslonite. Sakupljanje smeća uopšte ne mora da se dogodi, a ako se dogodi, može da čisti objekte bilo kojim redosledom. Najbolje je ne oslanjati se na sakupljač smeća ni za šta osim za oslobađanje memorije. Ako vam je čišćenje potrebno, napravite svoje metode za njega i ne oslanjajte se na metodu **finalize**().

Vežba 12: (3) Dodajte odgovarajuću hijerarhiju metoda **ciscenje**() svim klasama iz vežbanja 9.

Sakrivanje imena

Ako je u osnovnoj klasi u Javi definisano ime metode koje je kasnije nekoliko puta preklopljeno, redefinisanje tog imena metode u izvedenoj klasi *neće* sakriti nijednu verziju iz osnovne klase (za razliku od C++-a). Stoga preklapanje funkcioniše bez obzira na to da li je metoda definisana na tom nivou ili u osnovnoj klasi:

```
//: ponovnaupotreba/Sakrivanje.java
//  Preklapanje imena metode osnovne klase u izvedenoj klasi
//  ne sakriva verzije iz osnovne klase.
import static net.mindview.util.Print.*;
class Homer {
  char doh(char c) {
    print("doh(char)");
    return 'd';
  }
  float doh(float f) {
    print("doh(float)");
    return 1.0f;
  }
}
```

```
class Milhaus {}

class Bart extends Homer {
  void doh(Milhaus m) {}
    print("doh(Milhaus)");
  }
}

public class Sakrivanje {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1);
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhaus());
  }
} /* Ispis:
doh(float)
doh(char)
doh(float)
doh(Milhaus)
*///:~
```

Vidite da su sve preklopljene metode klase **Homer** dostupne u klasi **Bart**, iako **Bart** uvodi novu preklopljenu metodu (time bi se u C++-u sakrile metode osnovne klase). Kao što ćete pročitati u narednom poglavlju, metode istog imena se mnogo češće redefinišu pomoću potpuno istog potpisa i povratnog tipa kao i u osnovnoj klasi. Inače rezultati mogu biti zbunjujući (zbog čega to u C++-u i nije dozvoljeno, da ne biste napravili grešku).

U Javi SE5 dodata je anotacija **@Override**, što nije rezervisana reč, ali se može upotrebljavati kao da jeste. Kada hoćete da redefinišete metodu, možete dodati tu anotaciju i prevodilac će prijaviti grešku ukoliko je nenamerno preklopite umesto da je redefinišete:

```
//: ponovnaupotreba/Lisa.java
//  {CompileTimeError}  (Neće se prevesti)

class Lisa extends Homer {
  @Override void doh(Milhaus m) {
    System.out.println("doh(Milhaus)");
  }
} ///:~
```

Oznaka **{CompileTimeError}** uklanja ovu datoteku iz Ant skripta za prevođenje primera iz knjige, ali ako probate ručno da prevedete taj primer dobićete poruku o grešci:

```
method does not override a method from its superclass
```

Time će anotacija **@Override** sprečiti da slučajno preklopite metodu koja nije bila predviđena za to.

Vežba 13: (2) Napravite klasu s metodom koja je preklopljena triput. Neka je nasledi nova klasa, dodajte još jedno preklapanje metode i pokažite da su sve četiri metode dostupne u izvedenoj klasi.

Izbor između kompozicije i nasleđivanja

I kompozicija i nasleđivanje dozvoljavaju da umetnete podobjekte u novu klasu (u kompoziciji to morate da uradite eksplicitno, a u nasleđivanju je to implicitno). Možete da se zapitate kakva je razlika između njih i kada se koja tehnika koristi.

Kompozicija se po pravilu koristi kada u novoj klasi hoćete da imate mogućnosti postojeće klase, ali ne i njen interfejs. Odnosno, vi ugradite objekat tako da ga možete koristiti za realizovanje funkcionalnosti u novoj klasi, ali korisnik te nove klase vidi interfejs koji ste vi za nju definisali, a ne interfejs ugrađenog objekta. Da biste to postigli, u novu klasu treba da ugradite privatni objekat postojeće klase.

Ponekad ima smisla da korisniku klase dozvolite direktan pristup kompoziciji nove klase, odnosno da objekte članove proglasite javnim. Pošto objekti članovi i sami koriste sakrivanje realizacije, to možete bez bojazni da uradite. Kada korisnik zna da je posredi sastavljanje od više delova, lakše će razumeti interfejs. Objekat **kola** je dobar primer:

```
//: ponovnaupotreba/Kola.java
// Kompozicija pomoću javnih objekata.
class Motor {
  public void start() {}
  public void rikverc() {}
  public void stop() {}
}

class Tocak {
  public void naduvaj(int psi) {}
}

class Prozor {
  public void podigni() {}
  public void spusti() {}
}

class Vrata {
  public Prozor prozor = new Prozor();
  public void otvori() {}
  public void zatvori() {}
}

public class Kola {
  public Motor motor = new Motor();
  public Tocak[] tockovi = new Tocak[4];
  public Vrata
    leva = new Vrata(),
    desna = new Vrata(); // sa dvoja vrata
```

```
  public Kola() {
    for(int i = 0; i < 4; i++)
      tockovi[i] = new Tocak();
  }
  public static void main(String[] args) {
    Kola kola = new Kola();
    kola.leva.prozor.podigni();
    kola.tockovi[0].naduvaj(72);
  }
} ///:~
```

Pošto je kompozicija kola u ovom slučaju deo analize problema (a ne samo deo osnovnog projekta), proglašavanjem članova javnim pomažete programeru klijentu da shvati kako da koristi klasu, a time se i smanjuje složenost koda koji autor klase mora da napiše. Imajte na umu da je ovo specijalan slučaj i da polja uglavnom treba da budu privatna.

Pri nasleđivanju polazite od postojeće klase i pravite njenu specijalnu verziju. U opštem slučaju, to znači da uzimate klasu opšte namene i specijalizujete je za određene potrebe. Ako malo razmislite, uvidećete kako nema smisla da klasu **auto** pravite kompozicijom od klase **vozilo** – auto ne sadrži vozilo, on *jeste* vozilo. Relacija *jeste* izražava se nasleđivanjem a relacija *sadrži* izražava se kompozicijom.

Vežba 14: (1) U datoteci **Kola.java** klasi **Motor** dodajte metodu **servis()** i pozovite je iz metode **main()**.

Rezervisana reč **protected**

Pošto ste savladali nasleđivanje, rezervisana reč **protected** napokon dobija značenje. U idealnom svetu, privatni članovi bi uvek bili potpuno privatni, ali u stvarnim projektima nekad hoćete da nešto sakrijete od šire javnosti, a da ipak izvedenim klasama dozvolite pristup.

Rezervisana reč **protected** doprinosi pragmatizmu. Ona kaže: „Ovo je privatno što se tiče korisnika klase, ali je dostupno svakom ko ovu klasu nasledi ili bilo kom drugom u istom paketu". Odnosno, u Javi zaštićeno (engl. *protected*) automatski podrazumeva i paketni pristup.

Iako možete praviti zaštićena polja, najbolje je da polja budu privatna – uvek treba da zadržite pravo da menjate osnovnu realizaciju. Tada pomoću zaštićenih metoda možete da dozvolite kontrolisan pristup naslednicima vaše klase:

```
//: ponovnaupotreba/Ork.java
// Rezervisana reč protected.
import static net.mindview.util.Print.*;

class Nitkov {
  private String ime;
  protected void postavi(String im) { ime = im; }
  public Nitkov(String ime) { this.ime = ime; }
  public String toString()  {
```

```
    return "Ja sam Nitkov i zovem se " + ime;
  }
}

public class Ork extends Nitkov {
  private int orkBroj;
  public Ork(String ime, int orkBroj) {
    super(ime);
    this.orkBroj = orkBroj;
  }
  public void promeni(String ime, int orkBroj) {
    postavi(ime);  // Dostupna zato što je zaštićena
    this.orkBroj = orkBroj;
 }
  public String toString()  {
    return "Ork" + orkBroj + ": " + super.toString();
  }
  public static void main(String[] args) {
    Ork ork = new Ork("Limburger", 12);
    print(ork);
    ork.promeni("Bob", 19);
    print(ork);
  }
} /* Ispis:
Ork 12: Ja sam Nitkov i zovem se Limburger
Ork 19: Ja sam Nitkov i zovem se Bob
* ///:~
```

Možete videti da metoda **promeni**() ima pristup metodi **postavi**() jer je ona zaštićena. Obratite pažnju i na to da je metoda **toString**() klase **Ork** definisana pomoću odgovarajuće verzije istoimene metode natklase.

Vežba 15: (2) Napravite klasu sa zaštićenom metodom unutar paketa. Pokušajte da pozovete tu zaštićenu metodu van paketa i objasnite rezultate. Sada nasledite tu klasu i pozovite zaštićenu metodu iz neke druge metode te nasleđene klase.

Svođenje naviše

Najbitniji vid nasleđivanja nije obezbeđivanje metoda novim klasama. Nasleđivanje predstavlja relaciju između nove i osnovne klase. Relacija može da se sažme kao „nova klasa *je tipa* postojeće klase".

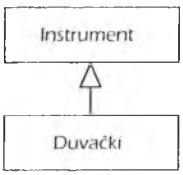
Prethodni opis nije izmišljen samo da bi se objasnilo nasleđivanje – jezik ga direktno podržava. Na primer, posmatrajte osnovnu klasu pod imenom **Instrument**, i izvedenu klasu pod imenom **Duvacki**. Pošto nasleđivanje podrazumeva da su sve metode osnovne klase dostupne i u izvedenoj klasi, svaku poruku koju možete da pošaljete osnovnoj klasi, možete da pošaljete i izvedenoj klasi. Ako klasa **Instrument** ima metodu **sviraj**(), imaće je i **Duvacki** instrumenti. To znači da, bez greške, možemo reći kako je objekat tipa **Duvacki** takođe i tipa **Instrument**. Naredni primer pokazuje kako prevodilac podržava tu notaciju:

```
//: ponovnaupotreba/Duvacki.java
// Nasleđivanje i svođenje naviše.
class Instrument {
  public void sviraj() {}
  static void melodija(Instrument i) {
    // ...
    i.sviraj();
  }
}

// Objekti klase Duvacki su instrumenti
// jer imaju isti interfejs:
public class Duvacki extends Instrument {
  public static void main(String[] args) {
    Duvacki flauta = new Duvacki();
    Instrument.melodija(flauta); // Svođenje naviše
  }
} ///:~
```

U ovom primeru važna je metoda **melodija()**, čiji argument je referenca na **Instrument**. Međutim, u metodi **Duvacki.main**(), metoda **melodija**() se poziva uz prosleđivanje reference na objekat klase **Duvacki**. Znajući da Java strogo proverava tipove, čudno je da metoda koja prihvata jedan tip spremno prihvata neki drugi, sve do trenutka dok ne shvatite da je objekat tipa **Duvacki** takođe i objekat tipa **Instrument.** Zbog toga ne postoji metoda **melodija**() kojoj može da se prosledi objekat klase **Instrument**, ali ne i objekat klase **Duvacki**. Kôd unutar metode **melodija**() radi za klasu **Instrument** i za sve što je iz nje izvedeno, a sam čin konverzije iz reference klase **Duvacki** u referencu klase **Instrument** naziva se *svođenje naviše* (engl. *upcasting*).

Zašto „svođenje naviše"?

Ovaj naziv je zasnovan na tradicionalnom načinu crtanja dijagrama nasleđivanja: koren je u vrhu, a dijagram se širi nadole. (Naravno, dijagrame možete da crtate na bilo koji način.) Dijagram nasleđivanja za **Duvacki.java** je:



Konverzijom od izvedene ka osnovnoj klasi krećete se *naviše* kroz dijagram nasleđivanja, pa se to često zove *svođenje naviše*. Svođenje naviše je uvek bezbedno, jer se sa određenijeg tipa vraćate na opštiji; odnosno, izvedena klasa je natklasa osnovne klase. Ona može da sadrži više metoda nego osnovna klasa, ali zato *mora* da sadrži one metode koje postoje u osnovnoj klasi. Prilikom svođenja naviše, jedino se može dogoditi da interfejs

klase izgubi neke metode, a ne da ih dobije. Zato prevodilac dozvoljava svođenje naviše bez eksplicitne konverzije ili neke druge posebne notacije.

Osim svođenja naviše, možete da uradite i suprotan proces, *svođenje naniže*, ali se pri tome javljaju određene dileme, o čemu će biti reči u narednom poglavlju i poglavlju *Podaci o tipu*.

Ponovo o biranju između kompozicije i nasleđivanja

Najčešći način za pravljenje i korišćenje objektno orijentisanih programa jeste pakovanje podataka i metoda u klase i upotreba objekata te klase. Koristićete i postojeće klase za izgrađivanje novih klasa kompozicijom. Nasleđivanje ćete ređe koristiti. Iako je pri podučavanju OOP-a prilično naglašeno nasleđivanje, to ne znači da ga treba koristiti gde god je moguće. Baš suprotno, treba ga koristiti štedljivo, samo kada je jasno da je nasleđivanje korisno. Najsigurnije ćete odrediti da li da koristite kompoziciju ili nasleđivanje ako se zapitate hoće li vam ikada biti potrebno svođenje naviše iz nove klase u osnovnu klasu. Ako morate da koristite svođenje naviše, nasleđivanje je neophodno, ali ako ne morate, pažljivo proučite da li vam nasleđivanje zaista treba. U poglavlju *Polimorfizam* opisuje se jedan od osnovnih razloga za svođenje naviše. Ako zapamtite ove principe i upitate se: „Da li mi je potrebno svođenje naviše?", u rukama ćete imati dobru alatku za odlučivanje između kompozicije i nasleđivanja.

Vežba 16: (2) Napravite klasu pod imenom **Vodozemac**. Iz nje nasledite klasu **Zaba**. Stavite odgovarajuće metode u osnovnu klasu. Iz metode **main**() napravite objekat klase **Zaba**, svedite ga naviše do klase **Vodozemac** i pokažite da sve metode i dalje rade.

Vežba 17: (1) Izmenite prethodnu vežbu tako da u klasi **Zaba** redefinišete metodu iz osnovne klase (napravite nove definicije koristeći iste potpise metoda). Pogledajte šta se dešava u metodi **main**().

Rezervisana reč **final**

Značenje Javine rezervisane reči **final** ima nekoliko nijansi, u zavisnosti od konteksta u kom je upotrebljena, ali opšte je: „Ovo ne može da se menja". Možete poželeti da promene sprečite zbog projekta ili efikasnosti. Pošto su ti principi sasvim različiti, rezervisana reč **final** može se i pogrešno upotrebiti.

U narednom odeljku razmatraju se tri slučaja u kojima može da se upotrebi rezervisana reč **final**: za podatke, metode i klase.

Finalni podaci

Mnogi programski jezici imaju način da prevodiocu naznače kako je neki podatak „konstantan". Konstante su korisne iz dva razloga:

1. Mogu da zadaju *konstantne vrednosti pri prevođenju* koje se nikada neće promeniti.
2. Mogu da predstavljaju vrednosti koje inicijalizujemo tokom izvršavanja, a ne želimo da budu menjane.

Kada se koristi konstanta pri prevođenju, prevodiocu je dozvoljeno da konstantnu vrednost „uklopi" u sva izračunavanja u kojima se pojavljuje, to jest računanje može da se obavi tokom prevođenja, čime se ubrzava izvršavanje. Ova vrsta konstanti u Javi mora da bude prostog tipa i označava se pomoću rezervisane reči **final**. Njena vrednost mora biti zadata na mestu definicije.

Za polje koje je i statično i finalno postoji samo jedno mesto za skladištenje i ono ne može da se menja.

Kada se rezervisana reč **final** koristi uz reference na objekte umesto uz proste tipove, njeno značenje može da zbuni. Uz prost tip, **final** označava da je *vrednost* konstantna, a uz referencu na objekat da je *referenca* konstantna. Kada se referenca jednom inicijalizuje i poveže sa objektom, ona više ne može da bude promenjena tako da pokazuje na neki drugi objekat. Sam objekat, međutim, može da se izmeni – Java ne obezbeđuje način da neki proizvoljan objekat pretvorite u konstantan. (Možete, ipak, svoju klasu da napišete tako da se njeni objekti efektivno ponašaju kao da su konstantni.) Ovo ograničenje se odnosi i na nizove, koji su takođe objekti.

Sledi primer finalnih polja. Obratite pažnju na to da se statična i finalna polja (tj. konstante pri prevođenju) po konvenciji pišu velikim slovima, s tim da se između reči piše potcrta.

```
//: ponovnaupotreba/FinalniPodaci.java
// Efekat rezervisane reči final na polja.
import static java.util.*;
import static net.mindview.util.Print.*;

class Vrednost {
  int i; // Paketni pristup
  public Vrednost(int i)  { this.i = i }
}

public class FinalniPodaci {
  private static Random slucajan = new Random(47);
  private String id;
  public FinalniPodaci(String id)   { this.id = id }
  // Mogu da budu konstante pri prevođenju
  private final int vrednostJedan = 9;
  private static final int VREDNOST_DVA = 99;
  // Tipična javna konstanta:
  public static final int VREDNOST_TRI = 39;
  // Ne mogu da budu konstante pri prevođenju:
  private final int i4 = slucajan.nextInt(20);
  static final int INT_5 = slucajan.nextInt(20);
  private Vrednost v1 = new Vrednost(11);
  private final Vrednost v2 = new Vrednost(22);
  private static final Vrednost VRE_3 = new Vrednost(33);
  // Nizovi:
  private final int[] a = { 1, 2, 3, 4, 5, 6 };
  public String toString() {
```

```
    return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
  }
  public static void main(String[] args) {
    FinalniPodaci fd1 = new FinalniPodaci ("fd1");
    //! fd1.vrednostJedan++; // Greška: vrednost ne može da bude promenjena
    fd1.v2.i++; // Objekat nije konstantan!
    fd1.v1 = new Vrednost(9); // OK -- nije finalna
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Objekat nije konstantan!
    //! fd1.v2 = new Vrednost(0); // Greška: ne možete da
    //! fd1.VAL_3 = new Vrednost(1); // promenite referencu
    //! fd1.a = new int[3];
    print(fd1);
    print("Pravljenje novog objekta FinalniPodaci");
    FinalniPodaci fd2 = new FinalniPodaci("fd2");
    print(fd1);
    print(fd2);
  }
} /* Ispis:
fd1: i4 = 15, INT_5 = 18
Pravljenje novog objekta FinalniPodaci
fd1: i4 = 15, INT_5 = 18
fd1: i4 = 13, INT_5 = 18
* ///:~
```

Pošto su promenljive **vrednostJedan** i **VREDNOST_DVA** prostog tipa, finalne su i dobijaju vrednost prilikom prevođenja, obe mogu da se upotrebe kao konstante pri prevođenju i bitno se ne razlikuju. **VREDNOST_TRI** je uobičajeniji način za definisanje takvih konstanti: kao *javne*, zbog čega mogu da se koriste i izvan paketa, *statične*, da bi se naglasilo kako postoji samo jedna kopija i *finalne*, da se naznači da su konstantne. Obratite pažnju na to da su finalne i statične promenljive prostog tipa s konstantnim inicijalnim vrednostima (tj. konstante pri prevođenju), po konvenciji napisane velikim slovima, a reči su razdvojene potcrtom (kao i konstante u C-u, odakle i potiče ova konvencija).

Samo zato što je nešto finalno, ne znači da je njegova vrednost poznata u trenutku prevođenja. U primeru, to je pokazano na promenljivama **i4** i **INT_5** koje su inicijalizovane slučajnim brojevima tokom izvršavanja. Taj deo primera takođe pokazuje u čemu je razlika između statičnih i nestatičnih finalnih vrednosti. Razlika se javlja samo kada se vrednosti inicijalizuju prilikom izvršavanja, jer sve konstante pri prevođenju prevodilac isto tretira. (I verovatno optimizuje i eliminiše, jer više nisu potrebne.) Razlika se vidi iz rezultata izvršavanja primera. Vrednosti **i4** za **fd1** i **fd2** jedinstvene su, dok se vrednost **INT_5** ne menja nakon pravljenja drugog objekta **FinalniPodaci**. Ona je statična, pa se inicijalizuje samo jednom pri učitavanju klase, a ne svaki put kada se pravi novi objekat.

Promenljive od **v1** do **VRE_3** pokazuju značenje finalnih referenci. Kao što možete videti u metodi **main**(), to što je **v2** finalna, ne znači da ne možete menjati vrednost objekta na koji ukazuje. Ali pošto je **v2** referenca, rezervisana reč **final** znači da **v2** ne možete da povežete s nekim drugim objektom. Isto važi i za niz, koji je u suštini samo druga vrsta

reference. (Koliko ja znam, ne postoji način da same reference u nizu proglasite finalnim.) Proglašavanje referenci finalnim čini se manje korisnim od proglašavanja promenljivih prostog tipa finalnim.

Vežba 18: (2) Napravite klasu koja ima statično finalno polje i finalno polje i pokažite razlike između njih.

Prazna finalna polja

Java dozvoljava pravljenje *praznih finalnih polja* (engl. *blank finals*), što označava polja koja su deklarisana kao finalna, ali im nije zadata inicijalizaciona vrednost. U svim slučajevima, prazna finalna polja *moraju* da budu inicijalizovana pre nego što se upotrebe i prevodilac to proverava. Prazna finalna polja su, međutim, mnogo fleksibilnija. Na primer, finalno polje unutar klase u tom slučaju može da bude različito za svaki objekat, a da ipak pri tom zadrži nepromenljivost. Evo primera:

```
//: ponovnaupotreba/PraznoFinalno.java
// "Prazna" finalna polja.

class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}
public class PraznoFinalno {
  private final int i = 0; // Inicijalizovana finalna promenljiva
  private final int j; // prazna finalna promenljiva
  private final Poppet p; // prazna finalna referenca
  // Prazna finalna polja MORAJU da budu inicijalizovana u konstruktoru:
  public PraznoFinalno () {
    j = 1; // Inicijalizovanje praznog finalnog polja
    p = new Poppet(1);  // Inicijalizovanje prazne finalne reference
  }
  public PraznoFinalno (int x) {
    j = x; // Inicijalizovanje praznog finalnog polja
    p = new Poppet(x);  // Inicijalizovanje prazne finalne reference
  }
  public static void main(String[] args) {
    new PraznoFinalno ();
    new PraznoFinalno (47);
  }
} ///:~
```

Finalnim poljima morate da dodelite vrednosti pomoću izraza na mestu definicije ili u svakom konstruktoru. Na taj način se garantuje da su finalna polja uvek inicijalizovana pre upotrebe.

Vežba 19: (2) Napravite klasu s praznom finalnom referencom na objekat. Unutar svih konstruktora inicijalizujte prazan finalni element. Dokažite da finalni element mora biti inicijalizovan pre upotrebe i da nakon toga više ne može da se menja.

Finalni argumenti

Argumente možete da proglasite finalnim tako što ih deklarišete kao finalne u listi argumenata. To znači da unutar metode ne možete menjati ono na šta taj argument kao referenca ukazuje:

```
//: ponovnaupotreba/FinalniArgumenti.java
// Upotreba rezervisane reči "final" uz argumente metode.

class Stvar {
  public void zavrti() {}
}

public class FinalniArgumenti {
  void sa(final Stvar g) {
    //! g = new Stvar(); // Nedozvoljeno -- g je finalno
  }
  void bez(Stvar g) {
    g = new Stvar(); // OK -- g nije finalno
    g.zavrti();
  }
  // void f(final int i) { i++; } // Ne možete da promenite
  // Finalnu promenljivu prostog tipa možete samo da čitate:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalniArgumenti bf = new FinalniArgumenti();
    bf.bez(null);
    bf.sa(null);
  }
} ///:~
```

Metode **f()** i **g()** pokazuju šta se dešava kada su argumenti prostog tipa finalni: argument možete čitati, ali ga ne možete promeniti. To se koristi prvenstveno za prosleđivanje podataka anonimnim unutrašnjim klasama koje ćete upoznati u poglavlju *Unutrašnje klase.*

Finalne metode

Za upotrebu finalnih metoda postoje dva razloga. Prvi je da se metoda „zaključa" kako bi se sprečilo da bilo koja klasa naslednica promeni njeno značenje. To se radi zbog projektnih razloga, kada želimo da budemo sigurni kako će ponašanje metode biti nepromenjeno prilikom nasleđivanja, odnosno kako metoda ne može biti redefinisana.

Ranije se kao drugi razlog za korišćenje finalnih metoda navodila efikasnost. U prethodnim realizacijama Jave, kada biste metodu proglasili finalnom, dozvoljavali ste prevodiocu da sve pozive toj metodi ugradi direktno u kôd (engl. *inline call).* Kada bi prevodilac naišao na poziv finalnoj metodi, mogao je (po svom nahođenju) da preskoči uobičajen poziv (stavi argumente na stek, skoči na kôd metode i izvrši ga, vrati se, počisti argumente sa steka i razmotri povratnu vrednost). Umesto toga, poziv metode mogao je da se zameni kopijom stvarnog koda iz njenog tela. Time su se eliminisali režijski troškovi samog poziva. Naravno, ako je metoda velika, kôd počinje da se povećava i verovatno ne

biste dobiti poboljšanje performansi, jer bi sva unapređenja bila zanemariva u odnosu na vreme izvršavanja metode.

U novijim verzijama Jave, virtuelna mašina (konkretno, tehnologije vrućih tačaka) može da prepozna takve situacije i pametno odabere da li da koristi umetanje koda na mestu poziva za finalnu metodu, pa više ne treba da pomažete optimizatoru tako što ćete koristiti rezervisanu reč **final** (po pravilu, to ne bi trebalo da radite). U Javi SE5/6, pustite da se prevodilac i izvršno okruženje (JVM) staraju o efikasnosti, a metodu proglasite finalnom samo ako želite izričito da sprečite njeno redefinisanje.[1]

Finalno i privatno

Svaka privatna metoda u klasi implicitno je i finalna. Pošto privatnoj metodi ne možete da pristupite iz nasleđenih klasa, ne možete ni da je redefinišete. Privatnoj metodi možete da dodelite specifikator **final**, ali njoj to ne daje nikakvo dodatno značenje.

Ovo stanje može da izazove zabunu – ako pokušate da redefinišete privatnu metodu (koja je implicitno finalna), to ćete samo naizgled uspeti, jer prevodilac neće prijaviti grešku:

```
//: ponovnaupotreba/IluzijaRedefinisanjaFinalnih.java
// Privatnu ili privatnu finalnu metodu samo naizgled možete da redefinišete.
import static net.mindview.util.Print.*;

class SadrziFinalne {
  // Identično kao i kada stoji samo "private":
  private final void f() {  print("SadrziFinalne.f()");  }
  // Takođe automatski finalna:
  private void g() {  print("SadrziFinalne.g()");  }
}

class RedefinisanjePrivatnih extends SadrziFinalne {
  private final void f() {
    print("RedefinisanjePrivatnih.f()");
  }
  private void g() {
    print("RedefinisanjePrivatnih.g()");
  }
}

class RedefinisanjePrivatnih2 extends RedefinisanjePrivatnih {
  public final void f() {
    print("RedefinisanjePrivatnih2.f()");
  }
  public void g() {
    print("RedefinisanjePrivatnih2.g()");
  }
}
```

[1] Nemojte podleći iskušenju da prerano optimizujete. Ako postignete da vaš sistem radi, ali je previše spor, pitanje je možete li to popraviti pomoću rezervisane reči **final**. Na *http://MindView.net/Books/NBetterJava* imate podatke o profilisanju – to zaista *može* da ubrza vaš program.

```
public class IluzijaRedefinisanjaFinalnih {
  public static void main(String[] args) {
    RedefinisanjePrivatnih2 op2 = new RedefinisanjePrivatnih2 ();
    op2.f();
    op2.g();
    // Možete da obavite svođenje naviše:
    RedefinisanjePrivatnih op = op2;
    // Ali ne možete da pozovete metode:
    //! op.f();
    //! op.g();
    // Isto važi i ovde:
    SadrziFinalne sf = op2;
    //! sf.f();
    //! sf.g();
  }
} /* Ispis:
RedefinisanjePrivatnih2.f()
RedefinisanjePrivatnih2.g()
* ///:~
```

Redefinisanje se javlja samo ako se menja deo interfejsa osnovne klase. Odnosno, morate biti u mogućnosti da objekat svedete naviše do njegovog prostog tipa i da pozovete istu metodu (poenta ovoga će postati jasna u narednom poglavlju). Ukoliko je metoda privatna, ona nije deo interfejsa osnovne klase. Ona je samo neki kôd koji je sakriven unutar klase i samo slučajno ima isto ime kao metoda u izvedenoj klasi. Ako napravite javnu metodu, zaštićenu metodu ili metodu s paketnim pristupom u izvedenoj klasi, ne postoji veza s privatnom metodom koja može imati takvo isto ime u osnovnoj klasi. Niste redefinisali postojeću metodu, nego napravili novu. Pošto je privatna metoda nedostupna i efektivno nevidljiva, ona ne utiče ni na šta sem na organizaciju koda klase u kojoj je definisana.

Vežba 20: (1) Pokažite da anotacija **@Override** rešava problem razmotren u ovom odeljku.

Vežba 21: (1) Napravite klasu s finalnom metodom. Nasledite klasu i pokušajte da tu metodu redefinišete.

Finalne klase

Kada označite da je cela klasa finalna (stavljajući rezervisanu reč **final** pre njene definicije), saopštavate da tu klasu ne želite da nasleđujete niti dozvoljavate bilo kome da to učini. Drugim rečima, iz nekog razloga projekat vaše klase nikad se neće menjati ili iz bezbednosnih razloga ne želite da omogućite nasleđivanje.

```
//: ponovnaupotreba/Preistorijski.java
// Proglašavanje cele klase finalnom.

class MaliMozak {}

final class Dinosaurus {
  int i = 7;
  int j = 1;
```

```
  MaliMozak x = new MaliMozak();
  void f() {}
}

//! class Nastavak extends Dinosaurus {}
// greška: Klasa 'Dinosaurus' ne može da bude proširena

public class Preistorijski {
  public static void main(String[] args) {
    Dinosaurus n = new Dinosaurus();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Polja finalne klase mogu da budu finalna, ali i ne moraju, kako god izaberete. Za finalna polja važe ista pravila kao i ranije, bez obzira na to da li je sama klasa finalna. Kako je nasleđivanje sprečeno, sve *metode* u finalnoj klasi su implicitno (automatski) finalne, pošto ne postoji način za njihovo redefinisanje. Metodama u finalnoj klasi možete da dodate specifikator **final**, ali on ne dodaje nikakvo novo značenje.

Vežba 22: (1) Napravite finalnu klasu i pokušajte da je nasledite.

Pažljivo sa **final**

Kada projektujete klasu, može vam se učiniti kao dobra ideja da metodu proglasite finalnom. Možda pomislite da ionako nema šanse da iko ikada redefiniše vaše metode. Ponekad je to zaista tako.

Ipak, pažljivo razmislite. Uglavnom je teško predvideti kako će se klasa ponovo koristiti, naročito klase opšte namene. Ako metodu proglasite finalnom, sprečićete da vašu klasu putem nasleđivanja upotrebi neki drugi programer, i to samo zato što niste mogli zamisliti da se ona može koristiti na taj način.

Standardna Javina biblioteka je dobar primer. Naročito klasa **Vector** iz Jave 1.0/1.1 koja se često koristi i koja je mogla da bude još korisnija da nisu, u ime efikasnosti (što je gotovo sigurno bila zabluda) sve metode proglašene finalnim. Lako je pretpostaviti kako želite da nasledite i redefinišete tako fundamentalno korisnu klasu, ali su projektanti nekako odlučili da to nije u redu. Ironično – iz dva razloga. Prvo, klasa **Stack** je nasleđena iz klase **Vector**, što govori da **Stack** *jeste* **Vector**, mada to i nije baš tačno s logičke tačke gledišta. Uprkos tome, sami projektanti Jave izveli su tu klasu iz klase **Vector**. Čim su na taj način napravili **Stack**, trebalo je da shvate kako su finalne metode previše ograničene.

Drugo, mnoge najvažnije metode klase **Vector**, kao što su **addElement**() i **elementAt**(), sinhronizovane su (rezervisana reč **synchronized**). Kao što ćete videti u poglavlju *Paralelno izvršavanje*, to pravi značajne režijske troškove koji verovatno poništavaju svu korist od toga što su metode finalne. Ova priča potvrđuje teoriju da programeri loše pogađaju mesta na kojima treba optimizovati. Zaista je loše što je tako nespretan projekat ušao u standardnu biblioteku pa svi moramo da se borimo s njim. (Na sreću, kontejnerska biblioteka

savremene Jave zamenjuje klasu **Vector** klasom **ArrayList**, koja se ponaša mnogo primerenije. Nažalost, i dalje dosta novih programa koriste staru kontejnersku biblioteku.)

Važno je napomenuti da klasa **Hashtable**, još jedna važna klasa iz standardne biblioteke Jave 1.0/1.1, *nema* nijednu finalnu metodu. Kao što je već rečeno na drugim mestima u ovoj knjizi, prilično je očigledno da su neke klase projektovali sasvim različiti ljudi. (Videćete i da su imena metoda u klasi **Hashtable** mnogo kraća od imena u klasi **Vector**, što je još jedan dokaz.) Baš to korisnicima biblioteke klasa *ne bi* trebalo da bude očigledno. Kada stvari nisu dosledne, korisnik samo ima više posla. To govori da projekat i kôd treba pregledati više puta. (U kontejnerskoj biblioteci savremene Jave, klasa **Hashtable** je zamenjena klasom **Hashmap**.)

Inicijalizacija i učitavanje klasa

Programi pisani na većini jezika učitavaju se u celini u okviru pokretanja programa. Nakon toga sledi inicijalizacija, a zatim otpočinje program. Proces inicijalizacije u takvim jezicima mora da bude pažljivo kontrolisan, kako poredak inicijalizacije statičnih elemenata ne bi prouzrokovao probleme. Primera radi, u C++-u se javlja problem kada jedan statični element očekuje da drugi bude ispravan pre nego što je taj drugi statični element inicijalizovan.

U Javi se taj problem ne pojavljuje, jer ona ima drugi pristup učitavanju. Ovo je jedna od aktivnosti koje su postale lakše zato što je u Javi sve objekat. Setite se da se prevedeni kôd svake klase nalazi u zasebnoj datoteci. Ta datoteka se ne učitava sve dok taj kôd ne bude potreban. Uopšteno, možemo reći da se „kôd klase učitava na mestu prve upotrebe". Obično se klasa učitava prilikom konstrukcije prvog objekta te klase, ali se učitavanje obavlja i prilikom pristupanja statičkom polju ili statičkoj metodi.[1]

Prilikom prvog korišćenja klase, obavlja se i statička inicijalizacija. Svi statični objekti i statični delovi koda inicijalizuju se u trenutku učitavanja po „redosledu pisanja" (tj. u poretku kojim ste ih napisali u definiciji klase). Statični elementi se, naravno, inicijalizuju samo jednom.

Inicijalizacija i nasleđivanje

Da bismo dobili celovitu sliku o svemu ovome, pogledajmo ceo proces inicijalizacije uključujući i nasleđivanje. Razmotrite naredni program:

```
//: ponovnaupotreba/Buba.java
// Potpuni proces inicijalizacije.
import static net.mindview.util.Print.*;

class Insekt {
  int i = 9;
  int j;
  Insekt() {
```

[1] I konstruktor je statična metoda, mada se rezervisana reč **static** ne mora eksplicitno navesti. Dakle, preciznije je reći da se klasa učitava onda kada se pristupi bilo kojem njenom statičnom članu.

```
    print("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    printInit("inicijalizovana statična promenljiva Insect.x1");
  static int printInit(String s) {
    print(s);
    return 47;
  }
}

public class Buba extends Insekt {
  private int k = printInit("Buba.k inicijalizovana");
  public Buba() {
    print("k = " + k);
    print("j = " + j);
  }
  private static int x2 =
    printInit("inicijalizovana statična promenljiva Buba.x2 ");
  public static void main(String[] args) {
    print("konstruktor klase Buba");
    Buba b = new Buba();
  }
} /* Ispis:
inicijalizovana statična promenljiva Insekt.x1
inicijalizovana statična promenljiva Buba.x2
konstruktor klase Buba
i = 9, j = 0
Buba.k inicijalizovana
k  = 47
j = 39
* ///:~
```

Kada pokrenete klasu **Buba**, najpre pokušavate da pristupite (statičnoj) metodi **Buba.main**(), zbog čega prevodilac (u datoteci **Buba.class**) traži prevedeni kôd klase **Buba**. Program za učitavanje (engl. *loader*) primećuje da ona ima osnovnu klasu (što označava rezervisana reč **extends**), pa zatim učita i nju. To će se dogoditi bez obzira na to da li ćete praviti objekat te osnovne klase. (Kao dokaz, probajte da stavite u komentar pravljenje objekta.)

Ako osnovna klasa ima sopstvenu osnovnu klasu, i ona će biti učitana i tako redom. Zatim se obavlja statička inicijalizacija u korenskoj osnovnoj klasi (u našem slučaju klasi **Insekt**), zatim u sledećoj izvedenoj i tako redom. Ovaj redosled je bitan, jer statička inicijalizacija izvedene klase može da zavisi od pravilne inicijalizacije nekog statičkog člana osnovne klase.

U tom trenutku, učitane su sve neophodne klase i objekat može da bude napravljen. Prvo, sve promenljive prostog tipa dobijaju podrazumevane vrednosti, a reference na objekte dobijaju vrednost **null** – to se obavlja odjednom, tako što se memorija u objektu popuni binarnim nulama. Zatim se poziva konstruktor osnovne klase. U našem slučaju,

poziv se obavlja automatski, ali takođe možete da odredite koji konstruktor osnovne klase treba da bude pozvan (prva operacija u konstruktoru klase **Buba**) pomoću rezervisane reči **super**. Konstruktor osnovne klase prolazi kroz isti proces i istim redosledom kao konstruktor izvedene klase. Nakon obavljene konstrukcije osnovne klase, promenljive instance inicijalizuju se po poretku pisanja. Konačno, izvršava se ostatak tela konstruktora.

Vežba 23: (2) Dokažite da se klasa učitava samo jednom. Dokažite da učitavanje može da bude prouzrokovano bilo pravljenjem prve instance te klase bilo pristupom statičnom članu.

Vežba 24: (2) U datoteci **Buba.java**, iz klase **Buba** nasledite specifičnu vrstu bube, držeći se istog formata kao kod postojećih klasa. Pratite i objasnite rezultat programa.

Sažetak

I nasleđivanje i kompozicija omogućavaju da od postojećih tipova napravite nove. Kompozicija upotrebljava postojeći tip kao deo osnovne realizacije novog tipa, a nasleđivanje ponovo upotrebljava interfejs.

U nasleđivanju, izvedena klasa obuhvata interfejs osnovne klase, pa može da bude *svedena naviše* ka osnovi, što je veoma bitno za polimorfizam (kao što ćete videti u narednom poglavlju).

Uprkos veoma naglašenom nasleđivanju u objektno orijentisanom programiranju, u prvom prolazu kroz projekat trebalo bi da se koncentrišete na kompoziciju (ili na delegiranje), a nasleđivanje koristite samo kada je neophodno. Kompozicija ume da bude fleksibilnija; lukavo primenjujući nasleđivanje na objekte članove, možete im promeniti tačan tip, a samim tim i ponašanje tokom izvršavanja. Drugim rečima, ponašanje objekta nastalog pomoću kompozicije možete da menjate tokom izvršavanja.

Kada projektujete sistem, vaš cilj je da pronađete ili napravite skup klasa u kome svaka klasa ima specifičnu namenu. Uz to, nijedna klasa ne treba da bude ni prevelika (s toliko mogućnosti da postaje nezgrapna da bi se ponovo koristila) ni premala (toliko da ni vi ne možete da je koristite takvu kakva je već je morate proširiti). Ukoliko projekat postane previše složen, često pomaže kada dodate nove objekte nastale cepanjem postojećih na manje delove.

Kada počnete da projektujete sistem, važno je da shvatite kako je razvoj programa postupan proces, baš kao i čovekovo učenje. Njegova osnova je eksperimentisanje; možete beskrajno dugo da analizirate stvari, ali i dalje nećete znati sve odgovore kada se upustite u projekat. Imaćete mnogo više uspeha – i više neposrednih podataka o rezultatima – ako „uzgajate" svoj projekat kao organsko biće koje se razvija, umesto da ga odjednom pravite kao stakleni soliter. Nasleđivanje i kompozicija spadaju među najvažnije alatke u objektno orijentisanom programiranju koje omogućavaju takvo eksperimentisanje.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solution Guide*, koji se može kupiti na Web lokaciji *www.MindView.net*.

Polimorfizam

„Pitali su me: 'Molim vas, gospodine Bebidže, recite hoće li iz vaše mašine izaći tačni odgovori čak i ako u nju unesete pogrešne brojeve?' Ne mogu da pojmim kakva je zbrka u glavama ljudi koji pitaju tako šta". Čarls Bebidž (1791-1871)

POLIMORFIZAM PREDSTAVLJA TREĆU SUŠTINSKU OSOBINU OBJEKTNO ORIJENTISANOG programskog jezika, pored apstrakcije podataka i nasleđivanja.

Polimorfizam obezbeđuje još jednu dimenziju u razdvajanju interfejsa i realizacije, tj. razdvaja *šta* i *kako*. Pored toga što omogućava napredniju organizaciju koda i poboljšava čitljivost, polimorfizam omogućava i pisanje *proširivih* programa koji mogu da „rastu" i tokom početnog pravljenja projekta i kada zatrebaju nove mogućnosti.

Kapsulacija daje nove tipove kombinovanjem karakteristika i ponašanja. Sakrivanje realizacije odvaja interfejs od realizacije tako što detalje proglašava privatnim. Ovakva mehanička organizacija je potpuno jasna nekome ko ima iskustva u proceduralnom programiranju. Polimorfizam, međutim, razdvaja klase po *tipovima*. U prethodnom poglavlju ste videli kako nasleđivanje omogućava da objekat tretirate kao objekat sopstvenog tipa *ili* kao objekat natklase. To je veoma bitno, jer omogućava da mnogo tipova (izvedenih iz istog prostog tipa) bude obrađivano kao da su svi jednog istog tipa i da jedan isti kôd radi jednako sa svim tim različitim tipovima. Polimorfni poziv metode omogućava da jedan tip izrazi svoju različitost od drugog sličnog tipa, dokle god su oba izvedena iz istog prostog tipa. Ta različitost se ispoljava u ponašanju metoda koje možete da pozovete preko osnovne klase.

U ovom poglavlju o polimorfizmu (koji se takođe naziva *dinamičko vezivanje*, kao i *kasno vezivanje* ili *vezivanje prilikom izvršavanja*) poći ćete od osnova, uz jednostavne primere koji ne sadrže ništa drugo sem polimorfnog ponašanja programa.

Ponovo o svođenju naviše

U prethodnom poglavlju videli ste kako objekat može da bude upotrebljen kao objekat sopstvene klase ili kao objekat natklase. Kada referencu na objekat tretirate kao referencu natklase, to se naziva *svođenje naviše* (engl. *upcasting*) zbog načina crtanja stabala nasleđivanja – sa osnovnom klasom u vrhu.

Videli ste takođe da se pojavljuje i problem prikazan u sledećem primeru s muzičkim instrumentima.

Prvo, pošto se u više ovih primera sviraju note, trebalo bi da u nekom paketu napravimo zaseban tip **Nota** definisan nabrajanjem:

```
//: polimorfizam/muzika/Nota.java
// Note koje će se svirati na muzičkim instrumentima.
package polimorfizam.muzika

public enum Nota {
    SREDNJE_C, C_POVISENO,  B_SNIZENO ; // Itd.
} ///:~
```

Nabrojane tipove predstavili smo u poglavlju *Inicijalizacija i čišćenje.*
Ovde je **Duvački** tip **Instrumenta**; zato je **Duvački** nasleđen od **Instrumenta**:

```
//: polimorfizam/muzika/Instrument.java
package polimorfizam.muzika
import static net.mindview.util.Print.*;

class Instrument {
  public void sviraj(Nota n) {
    print("Instrument.sviraj()");
  }
}
  ///:~

//: polimorfizam/muzika/Instrument.java
package polimorfizam.muzika

// Objekti klase Duvacki su instrumenti
// jer imaju isti interfejs:
public class Duvacki extends Instrument {
  // Redefinisanje metode interfejsa:
  public void sviraj(Nota n) {
    System.out.println("Duvacki.sviraj()" + n);
  }
}  ///:~

//: polimorfizam/muzika/Instrument.java
//: Nasleđivanje i svođenje naviše
package polimorfizam.muzika

public class Muzika {
  public static void melodija(Instrument i) {
    // ...
    i.sviraj(Nota.SREDNJE_C);
  }
  public static void main(String[] args) {
    Duvacki flauta = new Duvacki();
    melodija(flauta); // Svođenje naviše
  }
} /* Ispis:
Duvacki.sviraj() SREDNJE_C
* ///:~
```

Metoda **Muzika.melodija()** prihvata reference na objekte klase **Instrument**, ali takođe i na bilo šta što je iz te klase izvedeno. U to možete da se uverite u metodi **main**(), u kojoj se metodi **melodija()** prosleđuje referenca na objekat klase **Duvacki**, a da nema potrebe za eksplicitnom konverzijom. To je prihvatljivo, jer je klasa **Duvacki** nasleđena iz klase **Instrument**, pa u njoj mora da postoji interfejs klase **Instrument**. Svođenje naviše iz klase **Duvacki** na klasu **Instrument** može da „suzi" interfejs, ali on neće biti manji od potpunog interfejsa klase **Instrument**.

Zanemarivanje tipa objekta

Možda vam program **Muzika.java** izgleda neobično. Zašto bi iko namerno *zanemario* tip objekta? Baš to se dešava pri svođenju naviše. Naizgled je mnogo logičnije da argument metode **melodija**() bude referenca na klasu **Duvacki**. To nas dovodi do ključnog dela: u tom slučaju, za svaki tip instrumenta morali biste da pišete novu metodu **melodija**(). Pretpostavimo da takav način razmišljanja primenite i na ostale instrumente:

```
//: polimorfizam/muzika/Muzika2.java
// Preklapanje umesto svođenja naviše
package polimorfizam.muzika
import static net.mindview.util.Print.*;

class Zicani extends Instrument {
  public void sviraj(Nota n) {
    print("Zicani.sviraj()" + n);
  }
}

class LimeniDuvacki extends Instrument {
  public void sviraj(Nota n) {
    print("LimeniDuvacki.sviraj()" + n);
  }
}

public class Muzika2 {
  public static void melodija(Duvacki i) {
    i.sviraj(Nota.SREDNJE_C);
  }
  public static void melodija(Zicani i) {
    i.sviraj(Nota.SREDNJE_C);
  }
  public static void melodija(LimeniDuvacki i) {
    i.sviraj(Nota.SREDNJE_C);
  }
  public static void main(String[] args) {
    Duvacki flauta = new Duvacki();
    Zicani violina = new Zicani();
    LimeniDuvacki francuskiRog = new LimeniDuvacki();
    melodija(flauta); // Nema svođenja naviše
    melodija(violina);
    melodija(francuskiRog);
  }
} /* Ispis:
Duvacki.sviraj() SREDNJE_C
Zicani.sviraj() SREDNJE_C
LimeniDuvacki.sviraj() SREDNJE_C
* ///:~
```

Ovaj pristup dovodi do željenih rezultata, ali uz veliku manu: za svaki novi **Instrument** koji dodate, morate da napišete posebnu metodu. To na samom početku zahteva više programiranja, ali takođe znači da ćete imati i puno posla ako želite da dodate novu metodu sličnu metodi **melodija()** ili novu potklasu klase **Instrument**. Dodajte tome činjenicu da prevodilac neće prijaviti poruku o grešci ako zaboravite da preklopite neku metodu, pa ceo proces rada s tipovima postaje veoma nepodesan.

Zar ne bi bilo mnogo lepše ako biste mogli da napišete samo jednu metodu, čiji bi argument bila osnovna klasa, a ne neka posebno izvedena klasa? Odnosno, zar ne bi bilo fino ako biste mogli zanemariti činjenicu da su to izvedene klase i mogli da napišete program tako da se obraća samo osnovnoj klasi?

Polimorfizam vam omogućava baš to. Međutim, većina programera sa iskustvom u proceduralnom programiranju malo teže razume način na koji polimorfizam deluje.

Vežba 1: (2) Napravite klasu **Cikl** s potklasama **Unicikl**, **Bicikl** i **Tricikl**. Pokažite da se primerak svakog od tih tipova metodom **voziti( )** može svesti naviše na **Cikl**.

Začkoljica

Problem s programom **Muzika.java** može da se uoči nakon pokretanja programa. Rezultat će biti poziv metode **Duvacki.sviraj()**. Dobijen je željeni rezultat, ali ne izgleda logično zašto to tako radi. Posmatrajte metodu **melodija()**:

```
public static void melodija(Instrument i) {
  // ...
  i.sviraj(Nota.SREDNJE_C);
}
```

Ona dobija referencu na **Instrument**. Kako, onda, prevodilac uopšte može znati da referenca na **Instrument** u stvari pokazuje na objekat klase **Duvacki**, a ne na objekat klase **LimeniDuvacki** ili **Zicani**? Prevodilac i ne zna. Da biste ovo pitanje bolje razumeli, korisno je proučiti *vezivanje*.

Vezivanje poziva metoda

Povezivanje poziva metode i tela metode naziva se *vezivanje* (engl. *binding*). Kada se vezivanje obavi pre pokretanja programa (što rade prevodilac i program za povezivanje – engl. *linker*, ako postoji), to se naziva *rano vezivanje* (engl. *early binding*). Moguće je da za ovaj termin uopšte niste čuli zato što u proceduralnim jezicima i nije bilo druge mogućnosti. Primera radi, C ima samo jedan način za pozivanje metoda, a to je rano vezivanje.

Ono što u prethodnom primeru zbunjuje, odnosi se na rano vezivanje. Pošto prevodilac ima samo referencu na **Instrument**, ne može da zna koju metodu treba da pozove.

Rešenje tog problema naziva se *kasno vezivanje* (engl. *late binding*), čime se označava da se vezivanje obavlja prilikom izvršavanja, u zavisnosti od tipa objekta. Kasno vezivanje se još naziva i *dinamičko vezivanje* (engl. *dynamic binding*) ili *vezivanje prilikom izvršavanja* (engl. *runtime binding*). Kada je u nekom jeziku omogućeno kasno vezivanje, mora

postojati mehanizam koji određuje tačan tip objekta prilikom izvršavanja i koji će potom pozvati odgovarajuću metodu. Odnosno, prevodilac i dalje ne zna tip objekta, ali mehanizam za pozivanje to otkriva i poziva odgovarajuće telo metode. Mehanizam kasnog vezivanja varira od jezika do jezika, ali je jasno da u objekte mora da bude ugrađena nekakva informacija o tipu.

U Javi se svi pozivi metoda obavljaju preko kasnog vezivanja, osim ako je metoda proglašena statičnom ili finalnom (privatne metode su implicitno finalne). To znači da obično ne morate odlučivati o tome da li će se koristiti kasno vezivanje – ono se upotrebljava automatski.

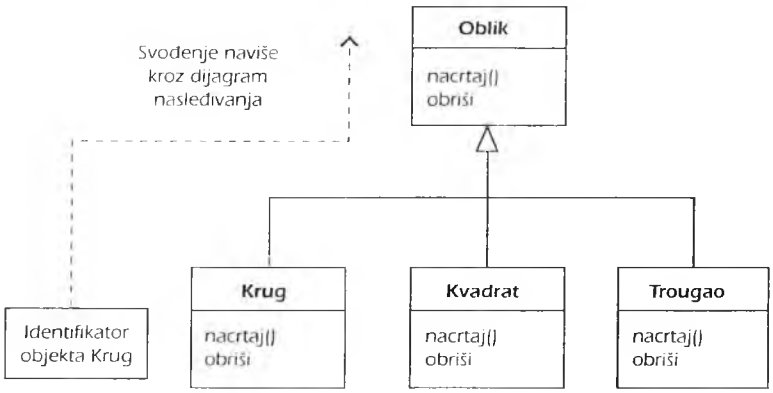
Zbog čega biste neku metodu proglasili finalnom? Kao što je u prethodnom poglavlju napomenuto, time sprečavate redefinisanje. Verovatno da je još bitnije što time efektivno „isključujete" dinamičko vezivanje, odnosno ukazujete prevodiocu kako dinamičko vezivanje nije neophodno. To omogućava nešto efikasnije pozivanje finalnih metoda. U većini slučajeva, međutim, tako se neće ostvariti opšte poboljšanje performansi programa, pa je najbolje da rezervisanu reč **final** koristite samo kao posledicu projekta, a ne kao pokušaj za poboljšavanje performansi.

Dobijanje pravilnog ponašanja

Imajući na umu da se svi pozivi metoda u Javi obavljaju polimorfno, preko kasnog vezivanja, program možete da pišete tako da se obraća osnovnoj klasi pa će pouzdano ispravno raditi i sa svim izvedenim klasama. Drugim rečima, „pošaljite poruku objektu i pustite ga da sam otkrije šta treba da uradi".

Klasičan primer u OOP-u je onaj sa oblicima. On se često koristi jer se lako prikazuje ali, nažalost, on programere početnike može navesti da pogrešno pomisle kako OOP služi samo za programiranje grafike, što, naravno, nije slučaj.

Primer sa oblicima ima osnovnu klasu pod imenom **Oblik** i razne izvedene klase: **Krug**, **Kvadrat**, **Trougao** itd. Ovaj primer je veoma pogodan, jer je lako pojmljivo kada kažemo „krug je oblik". Međusobni odnosi su pokazani na dijagramu nasleđivanja:



Svođenje naviše može da se pojavi čak i u ovako jednostavnoj naredbi:

```
Oblik o = new Krug();
```

U ovom slučaju, pravi se objekat klase **Krug**, a njegova referenca se odmah dodeljuje promenljivoj tipa **Oblik**, što je naizgled greška (jer jedan tip dodeljujemo drugom), a opet to je u redu jer, zbog nasleđivanja, **Krug** *jeste* **Oblik**. Stoga se prevodilac slaže s prethodnom naredbom i ne prijavljuje poruku o grešci.

Pretpostavimo da pozovete neku metodu osnovne klase (koja je redefinisana u izvedenim klasama):

```
o.nacrtaj();
```

Mogli biste, ponovo, očekivati kako će biti pozvana metoda **nacrtaj()** klase **Oblik**, pošto je to ipak referenca na **Oblik** – kako bi, onda, prevodilac mogao bilo šta drugo da uradi? A ipak, zbog kasnog povezivanja (polimorfizma) poziva se odgovarajuća metoda **Krug.nacrtaj()**.

Naredni primer pokazuje taj princip na nešto drugačiji način. Najpre ćemo napraviti višekratno upotrebljivu biblioteku tipova **Oblika**:

```
//: polimorfizam/oblik/Oblik.java
package polimorfizam.oblik;

public class Oblik {
  public void nacrtaj() {}
  public void obrisi() {}
} ///:~

//: polimorfizam/oblik/Krug.java
package polimorfizam.oblik;
import static net.mindview.util.Print.*;

public class Krug extends Oblik {
  public void nacrtaj() { print("Krug.nacrtaj()"); }
  public void obrisi() { print("Krug.obrisi()"); }
} ///:~

//: polimorfizam/oblik/Kvadrat.java
package polimorfizam.oblik;
import static net.mindview.util.Print.*;

public class Kvadrat extends Oblik {
  public void nacrtaj() { print("Kvadrat.nacrtaj()"); }
  public void obrisi() { print("Kvadrat.obrisi()"); }
} ///:~

//: polimorfizam/oblik/Trougao.java
package polimorfizam.oblik;
import static net.mindview.util.Print.*;
```

```
public class Trougao extends Oblik {
  public void nacrtaj() { print("Trougao.nacrtaj()"); }
  public void obrisi() { print("Trougao.obrisi()"); }
} ///:~

//: polimorfizam/oblik/GeneratorSlucajnihOblika.java
//: "Proizvodjac" koji nasumično proizvodi oblike.
package polimorfizam.oblik;
import java.util.*;

public class GeneratorSlucajnihOblika {
  private Random slucajan = new Random(47);
  public Oblik next() {
    switch(slucajan.nextInt(3)) {
      default:
      case 0: return new Krug();
      case 1: return new Kvadrat();
      case 2: return new Trougao();
    }
  }
} ///:~

//: polimorfizam/Oblici.java
// Polimorfizam u Javi
import polimorfizam.oblik.*;

public class Oblici  {
  private static GeneratorSlucajnihOblika gen =
    new GeneratorSlucajnihOblika();
  public static void main(String[] args) {
    Oblik[] o = new Oblik[9];
    // Popuni niz oblicima:
    for(int i = 0; i < o.length; i++)
      o[i] = gen.next();
    // Obavi polimorfne pozive metoda:
    for(Oblik obl : o)
      oblik.nacrtaj();
  }
} /* Ispis:
Trougao.nacrtaj()
Trougao.nacrtaj()
Kvadrat.nacrtaj()
Trougao.nacrtaj()
Kvadrat.nacrtaj()
Trougao.nacrtaj()
Kvadrat.nacrtaj()
Trougao.nacrtaj()
Krug.nacrtaj()
*///:~
```

Osnovna klasa **Oblik** ustanovljava zajednički interfejs za sve što je iz nje nasleđeno – odnosno, svi oblici mogu da budu nacrtani i obrisani. Izvedene klase redefinišu ove metode da bi obezbedile odgovarajuće ponašanje za specifičan oblik.

Klasa **GeneratorSlucajnihOblika** je neka vrsta „proizvođača" koja vraća referencu na slučajno odabran tip oblika, svaki put kada pozovete njenu metodu **next()**. Pri svakoj naredbi **return** obavlja se svođenje naviše, pri čemu se referenca na **Krug**, **Kvadrat** ili **Trougao** šalje iz metode **next()** i pretvara u povratni tip **Oblik**. Zato kadgod pozovete **next()**, nikada ne vidite koji je tip odabran, jer kao povratnu vrednost uvek dobijate referencu na **Oblik**.

Metoda **main()** sadrži niz referenci na objekte klase **Oblik**; taj niz se popunjava pozivima metodi **GeneratorSlucajnihOblika.next()**. U tom trenutku znate da imate neke objekte klase **Oblik** ali ne znate ništa preciznije (niti to zna prevodilac). Međutim, kada prođete kroz taj niz i za svaki element pozovete metodu **nacrtaj()**, zbog neke čarolije, svaki tip počinje da se pravilno ponaša. U to se možete uveriti iz rezultata programa kada ga pokrenete.

Nasumično pravljenje oblika treba da vam dokaže kako prevodilac ne može znati išta što bi mu pomoglo da pravilno poziva metode prilikom prevođenja. Svi pozivi metodi **nacrtaj()** moraju biti dinamički vezani.

Vežba 2: (1) U primer sa oblicima dodajte anotaciju **@Override**.

Vežba 3: (1) Dodajte novu metodu osnovnoj klasi u datoteci **Oblici.java** koja ispisuje poruku, ali nemojte da je redefinišete u izvedenim klasama. Objasnite šta se dešava. Sada je redefinišite u samo jednoj izvedenoj klasi i vidite šta se dešava. Konačno, redefinišite je u svim izvedenim klasama.

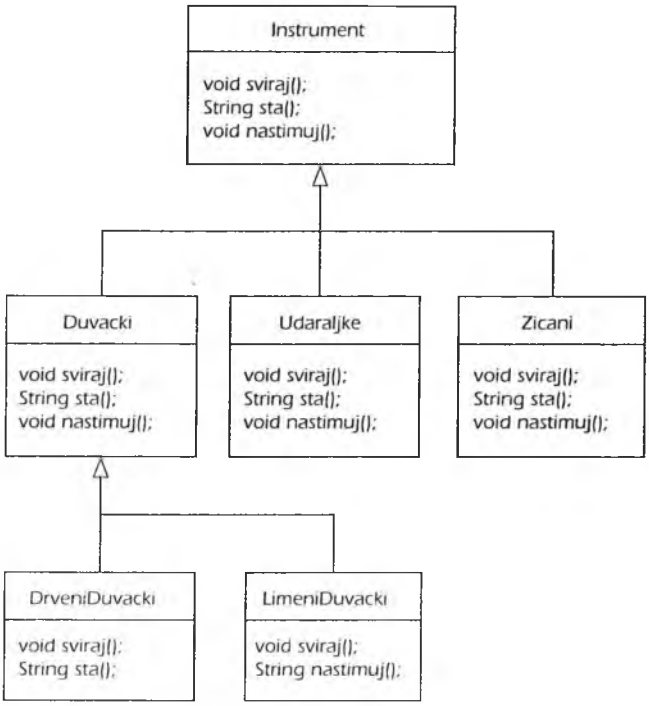
Vežba 4: (2) Dodajte novu potklasu klase **Oblik** u datoteci **Oblici.java** i potvrdite u metodi **main()** da polimorfizam radi za novi tip kao i za stare tipove.

Vežba 5: (1) U vežbi 1, klasi **Cikl** dodajte metodu **tockovi()** koja vraća broj točkova. Izmenite metodu **voziti()** tako da poziva metodu **tockovi()** i uverite se da polimorfizam radi.

Proširivost

Vratimo se sada primeru s muzičkim instrumentima. Polimorfizam omogućuje da tom sistemu dodate koliko god želite novih instrumenata, bez potrebe da menjate metodu **melodija()**. U dobro projektovanom OOP programu, većina metoda ili sve naše metode pratile bi model primenjen na metodu **melodija()** i komunikaciju obavljale samo preko interfejsa osnovne klase. Takav program je *proširiv*, jer možete da mu dodate novu funkcionalnost ako nove tipove podataka nasledite iz zajedničke osnovne klase. Metode koje obavljaju neki posao preko interfejsa osnovne klase, uopšte ne treba da menjate da bi prihvatile i nove klase.

Razmotrite šta se dešava ako u primeru sa instrumentima dodamo nove metode u osnovnu klasu, kao i izvestan broj novih klasa. Dijagram hijerarhije nasleđivanja dat je na slici:



Sve ove nove klase pravilno rade sa starom i neizmenjenom metodom **melodija**(). Čak i ako se metoda **melodija**() nalazi u posebnoj datoteci i ako se interfejsu klase **Instrument** dodaju nove metode, **melodija**() će, bez ponovnog prevođenja, pravilno raditi. Evo i realizacije dijagrama:

```
//: polimorfizam/muzika3/Muzika3.java
// Proširiv program
package polimorfizam.muzika3;
import polimorfizam.muzika.Nota;
import static net.mindview.util.Print.*;

class Instrument {
  void sviraj(Nota n) { print("Instrument.sviraj() " + n); }
  String sta() { return "Instrument"; }
  void nastimuj() { print("Štimovanje instrumenta"); }
}

class Duvacki extends Instrument {
  void sviraj(Nota n) { print("Duvacki.sviraj() " + n); }
  String sta() { return "Duvacki"; }
  void nastimuj() { print("Štimovanje duvačkog"); }
```

```
}

class Udaraljke extends Instrument {
  void sviraj(Nota n) { print("Udaraljke.sviraj() " + n); }
  String sta() { return "Udaraljke"; }
  void nastimuj() { print("Štimovanje udaraljke"); }
}

class Zicani extends Instrument {
  void sviraj(Nota n) { print("Zicani.sviraj() " + n); }
  String sta() { return "Zicani"; }
  void nastimuj() { print("Štimovanje žičanog"); }
}

class LimeniDuvacki extends Duvacki {
  void sviraj(Nota n) { print("LimeniDuvacki.sviraj() " + n); }
  void nastimuj() { print("LimeniDuvacki.nastimuj()"); }
}

class DrveniDuvacki extends Duvacki {
  void sviraj(Nota n) { print("DrveniDuvacki.sviraj() " + n); }
  String sta() { return "DrveniDuvacki"; }
}

public class Muzika3 {
  // Ne vodi računa o tipovima, pa će novi tipovi
  // koje dodamo sistemu i dalje pravilno raditi:
  public static void melodija(Instrument i) {
    // ...
    i.sviraj(Nota.SREDNJE_C);
  }
  public static void sveMelodije(Instrument[] e) {
    for(Instrument i : e)
      melodija(i);
  }
  public static void main(String[] args) {
    // Svođenje naviše prilikom dodavanja u niz:
    Instrument[] orkestar = {
      new Duvacki();
      new Udaraljke();
      new Zicani();
      new LimeniDuvacki();
      new DrveniDuvacki();
    };
    sveMelodije(orkestar);
  }
} /* Ispis:
Duvacki.sviraj() SREDNJE_C
Udaraljke.sviraj() SREDNJE_C
```

```
Zicani.sviraj() SREDNJE_C
LimeniDuvacki.sviraj() SREDNJE_C
DrveniDuvacki.sviraj() SREDNJE_C
* ///:~
```

Nove su metoda **sta()**, koja vraća referencu na objekat klase **String** u kojem je opis klase, i metoda **nastimuj()** koja omogućava da se svaki instrument na neki način naštimuje.

Svaki put kada nešto ubacite u niz **orkestar** u metodi **main()**, automatski obavljate svođenje naviše ka klasi **Instrument.**

Metoda **melodija()** je potpuno imuna na sve izmene u programu i radi pravilno. Polimorfizam baš to omogućava. Izmene u kodu ne remete delove programa na koje i ne bi trebalo da utiču. Drugim rečima, polimorfizam je važna tehnika koja programeru omogućava da „razdvoji ono što se menja od onog što ostaje isto".

Vežba 6: (1) Promenite datoteku **Muzika3.java** tako da metoda **sta()** postane metoda **toString()** iz korenske klase **Object.** Probajte da objekte klase **Instrument** ispisujete pomoću metode **System.out.println()** (bez ikakve konverzije).

Vežba 7: (2) U datoteku **Muzika3.java** dodajte novu potklasu klase **Instrument** i uverite se da polimorfizam radi i za novi tip.

Vežba 8: (2) Izmenite datoteku **Muzika3.java** tako da objekte klase **Instrument** pravi na slučajan način kao u datoteci **Oblici.java**.

Vežba 9: (3) Napravite hijerarhiju nasleđivanja klase **Glodar**: **Mis**, **Pacov**, **Hrcak** itd. U osnovnoj klasi obezbedite metode koje su zajedničke za sve glodare i redefinišite ih u izvedenim klasama, tako da se različito ponašaju za razne tipove glodara. Napravite niz objekata klase **Glodar**, popunite ga raznim objektima potklasa klase **Glodar**, pozovite metode definisane u osnovnoj klasi i vidite šta se dogada.

Vežba 10: (3) Napravite osnovnu klasu s dve metode. Pozovite drugu metodu iz prve metode. Nasledite tu klasu i redefinišite drugu metodu. Napravite objekat izvedene klase, svedite ga naviše do njegovog prostog tipa i pozovite prvu metodu. Objasnite šta se dešava.

Greška: „redefinisanje" privatnih metoda

Evo nečega što biste u blaženom neznanju mogli pokušati da uradite:

```
//: polimorfizam/RedefinisanjePrivatnih.java
// Pokušaj redefinisanja privatne metode.
package polimorfizam
import static net.mindview.util.Print.*;

public class RedefinisanjePrivatnih {
  private void f()  { print("privatna f()"); }
  public static void main(String[] args) {
    RedefinisanjePrivatnih rp = new Izvedena();
    rp.f();
  }
}
```

```
class Izvedena extends RedefinisanjePrivatnih {
  public void f()  { print("javna f()"); }
} /* Ispis:
privatna f()
* ///:-
```

Mogli biste razložno očekivati da rezultat bude **javna f()**, ali je privatna metoda automatski **finalna**, a i sakrivena je od izvedene klase. Zato je u ovom slučaju metoda **f()** klase **Izvedena** potpuno nova metoda; ona nije čak ni preklopljena, pošto u klasi **Izvedena** nije vidljiva verzija **f()** iz osnovne klase.

Rezultat svega ovoga jeste da samo neprivatne metode mogu biti redefinisane, ali se morate čuvati i od redefinisanja privatnih metoda – prevodilac vas ničim neće upozoriti na to, ali se program verovatno neće ponašati onako kako ste hteli. Da budemo potpuno jasni, u izvedenoj klasi upotrebljavajte imena drugačija od imena privatne metode iz osnovne klase.

Greška: polja i statične metoda

Pošto ste sada naučili šta je polimorfizam, mogli biste pomisliti da se sve dešava polimorfno. Međutim, polimorfni mogu biti samo obični pozivi metoda. Na primer, ako polju pristupite neposredno, taj pristup će biti razrešen u vreme prevođenja, kao što pokazuje[1] sledeći primer:

```
//: polimorfizam/PristupPolju.java
// Neposredno pristupanje polju biva razrešeno u trenutku prevođenja.

class Nad  {
  public int polje = 0;
  public int pribaviPolje ()  { return polje; }
}

class Pod extends Nad  {
  public int polje = 1;
  public int pribaviPolje ()  { return polje; }
  public int pribaviNadPolje () { return nad.polje; }
}

public class PristupPolju  {
  public static void main(String[] args)  {
    Nad nad = new Pod();  // Svođenje naviše
    System.out.println("nad.polje = " + nad.polje +
      ", nad.pribaviPolje() = " + nad.pribaviPolje());
    Pod pod = new Pod();
    System.out.println("pod.polje = " + pod.polje +
      ", pod.pribaviPolje() = " + pod.pribaviPolje() +
      ", pod.pribaviNadPolje() = " + pod.pribaviNadPolje());
  }
```

[1] Zahvaljujem Rendiju Nikolsu koji se setio da postavi to pitanje.

```
} /* Ispis:
nad.polje = 0, nad.pribaviPolje() = 1
pod.polje = 1, pod.pribaviPolje() = 1, pod.pribaviNadPolje = 0
* ///:~
```

Kada se **Pod** objekat svede naviše na **Nad** referencu, sva pristupanja poljima razrešava prevodilac, pa ona zato nisu polimorfna. U ovom primeru, **Nad.polje** i **Pod.polje** ne dobijaju isti memorijski prostor. Zato **Pod** zapravo sadrži dva polja **Polje**: sopstveno i ono koje dobija od klase **Nad**. Međutim, **Nad** verzija nije ona podrazumevana koja se dobija kada izraz upućuje na **polje** u klasi **Pod**; ako hoćete **Nad polje**, morate izričito napisati **nad.polje**.

Iako možda izgleda da bi ovo moglo izazvati zabunu, u praksi se to gotovo nikada ne dešava. Prvo, po pravilu sva polja treba da budu privatna, pa im nećete pristupati neposredno, nego samo kao sporedni efekat pozivanja metoda. Uz to, verovatno nećete dati isto ime polju osnovne klase i polju izvedene klase, jer biste tako izazvali zbrku.

Ako je metoda statična, ona se ne ponaša polimorfno:

```
//: polimorfizam/StaticniPolimorfizam.java
// Staticne metode nisu polimorfne.

class StaticnaNad {
  public static String staticnaPribavi() {
    return "Osnovna staticnaPribavi()";
  }
  public String dinamicnaPribavi() {
    return "Osnovna dinamicnaPribavi()";
  }
}

class StaticnaPod extends StaticnaNad {
  public static String staticnaPribavi() {
    return "Izvedena staticnaPribavi()";
  }
  public String dinamicnaPribavi() {
    return "Izvedena dinamicnaPribavi()";
  }
}

public class PolimorfizamStaticnih {
  public static void main(String[] args) {
    StaticnaNad nad = new StaticnaPod(); // Svođenje naviše
    System.out.println(nad.staticnaPribavi());
    System.out.println(nad.dinamicnaPribavi());
  }
} /* Ispis:
Osnovna staticnaPribavi()
Izvedena dinamicnaPribavi()
*///:~
```

Statične metode su pridružene klasama, a ne pojedinačnim objektima.

Konstruktori i polimorfizam

Konstruktori se, kao i obično, razlikuju od ostalih metoda. To važi i u slučaju polimorfizma. Iako konstruktori nisu polimorfni (to su zapravo statične metode, ali je njihova deklaracija statičnosti implicitna), bitno je da shvatite način na koji oni rade u složenim hijerarhijama i s polimorfizmom. To znanje će vam pomoći da izbegnete neprijatna zapetljavanja.

Redosled poziva konstruktora

Redosled poziva konstruktora je kratko razmatran u poglavlju *Inicijalizacija i čišćenje* i ponovo u poglavlju *Ponovno korišćenje klasa*, ali je to bilo pre nego što smo uveli polimorfizam.

U procesu konstrukcije izvedene klase uvek se poziva konstruktor osnovne klase. Taj poziv se automatski pomera naviše po stablu hijerarhije, tako da bivaju pozvani svi konstruktori osnovnih klasa. To ima smisla, jer konstruktor ima poseban zadatak: da se pobrine da objekat bude pravilno napravljen. Izvedena klasa ima pristup samo svojim članovima, ali ne i članovima osnovne klase (koji su obično privatni). Samo konstruktor osnovne klase ima odgovarajuće znanje i prava pristupa za inicijalizaciju elemenata te klase. Stoga je neophodno da budu pozvani svi konstruktori, jer, u suprotnom, neće biti napravljen ceo objekat. Zbog toga prevodilac insistira da bude pozvan konstruktor za svaki deo izvedene klase. Ukoliko u telu konstruktora izvedene klase izričito ne pozovete konstruktor osnovne klase, prevodilac će automatski da pozove njen podrazumevani konstruktor. Ako takav kostruktor ne postoji, prevodilac će prijaviti grešku. (Kada klasa nema nijedan konstruktor, prevodilac automatski pravi podrazumevani.)

Pogledajmo primer koji pokazuje efekte kompozicije, nasleđivanja i polimorfizma na poredak konstrukcije:

```
//: polimorfizam/Sendvic.java
// Poredak poziva konstruktora
package polimorfizam
import static net.mindview.util.Print.*;

class Obrok {
  Obrok() { print("Obrok()"); }
}

class Hleb {
  Hleb() { print("Hleb()"); }
}

class Sir {
  Sir() { print("Sir()"); }
}

class Salata {
  Salata() { print("Salata()"); }
}
```

```
class Rucak extends Obrok {
  Rucak() { print("Rucak()");}
}

class RucakZaNosenje extends Rucak {
  RucakZaNosenje() { print("RucakZaNosenje()");  }
}

public class Sendvic extends RucakZaNosenje {
  private Hleb h = new Hleb();
  private Sir s = new Sir();
  private Salata sl = new Salata();
  public Sendvic () { print("Sendvic ()");  }
  public static void main(String[] args) {
    new Sendvic ();
  }
} /* Ispis:
Obrok()
Rucak()
RucakZaNosenje()
Hleb()
Sir()
Salata()
Sendvic()
*///:~
```

U ovom primeru, složena klasa se pravi od drugih klasa. Svaki konstruktor ispisuje poruku kada je pozvan. Nama je bitna klasa **Sendvic** koja objedinjuje tri nivoa nasleđivanja (četiri, ako brojite i implicitno nasleđivanje klase **Object**) i tri objekta člana. Rezultat se vidi nakon pravljenja objekta klase **Sendvic** u metodi **main()**. To znači da je redosled poziva konstruktora složenog objekta sledeći:

1. Poziva se konstruktor osnovne klase. Ovaj korak se rekurzivno ponavlja tako da se prvo konstruiše koren u hijerarhiji, nakon njega prva izvedena klasa itd., sve dok se ne dođe do krajnje izvedene klase.
2. Pozivaju se inicijalizatori članova i to po redu njihove deklaracije.
3. Poziva se telo konstruktora izvedene klase.

Redosled poziva konstruktora je veoma bitan. Prilikom nasleđivanja, o osnovnoj klasi znate sve i možete da pristupite svim njenim javnim i zaštićenim članovima. To znači da možete pretpostaviti kako su, kada se nađete u izvedenoj klasi, svi članovi osnovne klase ispravni. Kada se nađete u običnoj metodi, konstrukcija je već obavljena, pa su svi članovi svih delova objekta napravljeni. Ali kada ste u konstruktoru, morate biti sigurni da su svi članovi klase koju nasleđujete već napravljeni. Jedini način da se to garantuje jeste da se prvo pozove konstruktor osnovne klase. Nakon toga, kada se nađete u konstruktoru izvedene klase, svi članovi osnovne klase kojima možete da pristupite pravilno su inicijalizovani. To što morate „znati da su svi članovi ispravni" unutar konstruktora takođe je razlog da, kada god to možete, inicijalizujete sve objekte članove (tj. objekte koje u klasu umećete

kompozicijom) na mestu njihove definicije u klasi (npr. **h, s i sl** u prethodnom primeru). Ako ovaj postupak poštujete, bićete sigurniji da su svi članovi klase *i* objekti članovi tekućeg objekta inicijalizovani. Nažalost, time ne pokrivate sve slučajeve, kao što ćete videti u narednom odeljku.

Vežba 11: (1) Datoteci **Sendvic.java** dodajte klasu **KiseliKrastavcic**.

Nasleđivanje i čišćenje

Kada novu klasu pravite kompozicijom i nasledivanjem, uglavnom nećete morati da brinete o **ciscenju**; podobjekti se obično mogu prepustiti sakupljaču smeća. Ukoliko čišćenje ipak treba da obavite, za svoju novu klasu morate sami da napravite neku metodu **ciscenje()** (ako imate bolje ime, nazovite je drugačije). I prilikom nasleđivanja, u izvedenoj klasi morate da redefinišete metodu **ciscenje()** ako je prilikom sakupljanja smeća potrebno nekakvo posebno čišćenje. Kada u izvedenoj klasi redefinišete metodu **ciscenje()**, bitno je da se setite da pozovete verziju te metode iz osnovne klase, inače osnovna klasa neće biti očišćena. To dokazuje naredni primer:

```
//: polimorfizam/Zaba.java
// Čišćenje i nasleđivanje.
package polimorfizam;
import static net.mindview.util.Print.*;

class Karakteristika {
  private String s;
  Karakteristika(String s) {
    this.s = s;
    print( "Pravljenje Karakteristike " + s);
  }
  protected void ciscenje() {
    print("čišćenje Karakteristike " + s); }
  }
}

class Opis  {
  private String s;
  Opis(String s)  {
    this.s = s;
    print("Pravljenje Opisa " + s);
  }
  prottected void ciscenje() {
    print("čišćenje Opisa " + s);
  }
}

class ZivoBice {
  private Karakteristika p =
    new Karakteristika("živo je");
  private Opis t =
```

```
    new Opis("Elementarno živo biće");
  ZivoBice() {
    print("ZivoBice()");
  }
  protected void ciscenje() {
    print("ZivoBice - čišćenje");
    t.ciscenje()
    p.ciscenje()
  }
}

class Zivotinja extends ZivoBice {
  private Karakteristika p =
    new Karakteristika("ima srce");
  private Opis t =
      new Opis("Životinja je, nije biljka");
  Zivotinja() { print("Životinja()"); }
  protected void ciscenje() {
    print("Zivotinja - čišćenje");
    t.ciscenje();
    p.ciscenje();
    super.ciscenje();
  }
}

class Vodozemac extends Zivotinja {
  private Karakteristika p =
    new Karakteristika("može da živi u vodi");
  private Opis t =
    new Opis("I u vodi i na zemlji");
  Vodozemac() {
    print("Vodozemac()");
  }
  protected void ciscenje() {
    print("Vodozemac - čišćenje");
    t.ciscenje();
    p.ciscenje();
    super.ciscenje();
  }
}

public class Zaba extends Vodozemac {
  private Karakteristika p = new Karakteristika("Krekeće");
  private Opis t = new Opis("Jede bube");
  public Zaba() { print("Zaba()"); }
  protected void ciscenje() {
    print("Zaba - čišćenje");
    t.ciscenje();
    p.ciscenje();
    super.ciscenje();
  }
```

```
  public static void main(String[] args) {
    Zaba zaba = new Zaba();
    print("Zdravo!");
    Zaba.ciscenje();
  }
} /* Ispis:
Pravljenje Karakteristike živo je
Pravljenje Opisa Elementarno živo biće
ZivoBice()
Pravljenje Karakteristike ima srce
Pravljenje Opisa Zivotinja je, nije biljka
Zivotinja()
Pravljenje Karakteristike može da živi u vodi
Pravljenje Opisa I u vodi i na zemlji
Vodozemac()
Pravljenje Karakteristike Krekeće
Pravljenje Opisa Jede bube
Zaba()
Zdravo!
Zaba - čišćenje
čišćenje Opisa Jede bube
čišćenje Karakteristike Krekeće
Vodozemac - čišćenje
čišćenje Opisa I u vodi i na zemlji
čišćenje Karakteristike može da živi u vodi
Zivotinja - čišćenje
čišćenje Opisa Zivotinja je, nije biljka
čišćenje Karakteristike  ima srce
ZivoBice - čišćenje
čišćenje Opisa Elementarno živo biće
čišćenje Karakteristike  živo je
*///:-
```

Svaka klasa u hijerarhiji sadrži i objekte članove tipova **Karakteristika** i **Opis**, koji takođe moraju biti očišćeni. Redosled čišćenja treba da je suprotan redosledu inicijalizacije, za slučaj da jedan podobjekat zavisi od drugog. Za polja to znači da redosled čišćenja treba da je suprotan redosledu deklarisanja (pošto se polja inicijalizuju onim redom kojim su deklarisana). Za osnovne klase (prateći formu upotrebljenu za destruktore u C++-u) prvo treba da očistite izvedene klase, a zatim i osnovne klase. Razlog je to što pri čišćenju izvedene klase možete da pozovete neke metode iz osnovne klase koje zahtevaju da komponente osnovne klase još budu žive, pa ne smete da ih uništite pre vremena. Iz rezultata prethodnog programa vidite da se delovi objekta **Zaba** čiste redom suprotnim redosledu kojim su pravljeni.

Iz ovog primera vidite sledeće: iako ne morate uvek sami da čistite, onda kada morate, taj proces zahteva pažnju i znanje.

Vežba 12: (3) Izmenite vežbu 9 tako da prikazuje redosled inicijalizacije u osnovnoj klasi i u izvedenim klasama. Sada i osnovnoj klasi i izvedenim klasama dodajte objekte članove i pokažite kojim redosledom se inicijalizuju prilikom konstrukcije.

Povedite računa i u tome da u gornjem primeru objekat **Zaba** „poseduje" svoje objekte članove. On ih pravi i zna koliko dugo treba da žive (koliko i **Zaba**), pa zna kada da pozove **ciscenje**() za te objekte članove. Međutim, ukoliko jedan ili više drugih objekata poseduju neki od tih objekata članova, zadatak postaje složeniji, pa više ne možete prosto da pretpostavite kako je dovoljno pozvati **ciscenje**(). U tom slučaju, možda će vam biti potrebno *brojanje referenci* da biste znali koliko objekata još uvek pristupa deljenom(zajedničkom) objektu. Evo kako to izgleda:

```
//: polimorfizam/BrojanjeReferenci.java
// Čišćenje deljenih objekata članova.
import static net.mindview.util.Print.*;

class Deljena {
  private int brojacref = 0;
  private static long brojac = 0;
  private final long id = brojac++;
  public Deljena() {
    print("Pravljenje " + this);
  }
  public void dodajRef() { brojacref++; }
  protected void ciscenje() {
    if(--brojacref == 0)
      print("Čišćenje " + this);
  }
  public String toString() { return "Deljena" + id;}
}

class Kompozicija {
  private Deljena deljena;
  private static long brojac = 0;
  private final long id = brojac++;
  public Kompozicija(Deljena deljena) {
    print("Pravljenje " + this);
    this.deljena = deljena;
    this.deljena.dodajRef();
  }
  protected void ciscenje() {
    print("čišćenje " + this);
    deljena.ciscenje()
  }
  public String toString() { return "Kompozicija " + id;}
}

public class BrojanjeReferenci {
  public static void main(String[] args) {
    Deljena deljena = new Deljena();
    Kompozicija[] kompozicija = { new Kompozicija(deljena),
      new Kompozicija(deljena), new Kompozicija(deljena),
      new Kompozicija(deljena), new Kompozicija(deljena) };
```

```
    for(Kompozicija c : kompozicija)
      c.ciscenje();
  }
} /* Ispis:
Pravljenje Deljena 0
Pravljenje Kompozicija 0
Pravljenje Kompozicija 1
Pravljenje Kompozicija 2
Pravljenje Kompozicija 3
Pravljenje Kompozicija 4
čišćenje Kompozicija 0
čišćenje Kompozicija 1
čišćenje Kompozicija 2
čišćenje Kompozicija 3
čišćenje Kompozicija 4
čišćenje Deljena 0
*///:~
```

Statičan **brojac** tipa **long** prati broj napravljenih primeraka objekta tipa **Deljena** i daje vrednost promenljivoj **id**. Tip **brojac**a je **long** umesto **int** da bi se izbeglo prekoračenje (u ovom slučaju, to je samo dobra praksa; verovatno nijedan primer iz ove knjige neće prouzrokovati prekoračenje takvog brojača). Promenljiva **id** je **final**na, zato što ne očekujemo da će menjati vrednost tokom životnog veka ovog objekta.

Kada svojoj klasi pridružite deljeni objekat, ne smete zaboraviti da pozovete **dodajRef()**, ali će metoda **ciscenje()** pratiti broj referenci i odlučiti kada da obavi čišćenje. Ova tehnika zahteva dodatan trud, ali ako su objekti koji zahtevaju čišćenje deljeni, nemate mnogo izbora.

Vežba 13: (3) Datoteci **BrojanjeReferenci.java** dodajte metodu **finalize()** da biste proverili postoji li *stanja okončanja* (videti poglavlje *Inicijalizacija i čišćenje*).

Vežba 14: (4) Izmenite vežbu 12 tako da jedan od objekata članova bude deljen s brojanjem referenci i pokažite da to ispravno radi.

Ponašanje polimorfnih metoda unutar konstruktora

Hijerarhija poziva konstruktora izaziva zanimljivu nedoumicu. Šta će se desiti ako se nalazite unutar konstruktora i pozovete dinamički povezanu metodu objekta čija je konstrukcija u toku?

Unutar obične metode, dinamički povezan poziv bio bi razrešen prilikom izvršavanja, jer objekat ne može da zna da li on pripada klasi u kojoj se metoda nalazi ili nekoj drugoj klasi koja je iz nje izvedena.

Ako dinamički povezanu metodu pozovete iz konstruktora, biće upotrebljena redefinisana verzija te metode. *Efekat* može da bude prilično neočekivan, zato što se redefinisana verzija metode poziva pre završetka konstruisanja objekta, a to može izazvati greške koje se veoma teško pronalaze.

Po konceptu, posao konstruktora je da objekat dovede u život (što je priličan podvig). Unutar svakog konstruktora, ceo objekat može biti samo delimično formiran – možete da znate samo da su inicijalizovani objekti osnovne klase. Ako je izrada konstruktora samo

jedan korak u pravljenju objekta klase koja je izvedena iz klase tog konstruktora, u vreme poziva tekućeg konstruktora izvedeni delovi još nisu inicijalizovani. Međutim, dinamički povezan poziv seže ka „spoljašnjosti" hijerarhije nasleđivanja. On poziva metodu izvedene klase. Ako ga iskoristite unutar konstruktora, pozvaćete metodu koja možda radi sa članovima koji još uvek nisu inicijalizovani, što je siguran recept za katastrofu.

Problem možete da sagledate u narednom primeru:

```
//: polimorfizam/PoliKonstruktori.java
// Konstruktori i polimorfizam
// ne pružaju ono što možda očekujete.
import static net.mindview.util.Print.*;

class Glif {
  void crtaj() {  print("Glif.crtaj()");  }
  Glif() {
    print("Glif() pre metode crtaj()");
    crtaj();
    print("Glif() posle metode crtaj()");
  }
}

class OkrugliGlif extends Glif {
  private int poluprecnik = 1;
  OkrugliGlif(int r) {
    poluprecnik = r;
    print("OkrugliGlif. OkrugliGlif(), poluprecnik = " + poluprecnik);
  }
  void crtaj() {
    print("OkrugliGlif.crtaj(), poluprecnik = " + poluprecnik);
  }
}

public class PoliKonstruktori {
  public static void main(String[] args) {
    new OkrugliGlif (5);
  }
} /* Ispis:
Glif() pre metode crtaj()
OkrugliGlif.crtaj(), poluprecnik = 0
Glif() posle metode crtaj()
OkrugliGlif. OkrugliGlif(), poluprecnik = 5
*///:~
```

U klasi **Glif**, metoda **crtaj**() može da bude redefinisana, što je urađeno u klasi **OkrugliGlif**. Tu metodu poziva konstruktor klase **Glif**, te će biti napravljen skok u metodu **OkrugliGlif.crtaj**(), što kao da nam je i bila namera. Ali, pogledajte rezultat i videćete sledeće: kada konstruktor klase **Glif** pozove metodu **crtaj**(), promenljiva **poluprecnik** nije još dobila ni podrazumevanu početnu vrednost 1. Njena vrednost je 0. Stoga bi na ekranu bila nacrtana tačka ili možda ne bi bilo nacrtano ništa, a vi biste ostali zbunjeni pokušavajući da otkrijete zašto program ne radi.

Poredak inicijalizacije koji je opisan u ranijem odeljku nije baš kompletan i to je ključ za rešenje misterije. Proces inicijalizacije zaista teče na sledeći način:

1. Prostor za smeštanje objekta popunjava se binarnim nulama pre nego što se bilo šta drugo odigra.
2. Pozivaju se konstruktori osnovnih klasa kao što je ranije opisano. Na tom mestu poziva se redefinisana metoda **crtaj()** (da, *pre* nego što se pozove konstruktor klase **OkrugliGlif**), koja zbog koraka 1 kao vrednost poluprečnika dobija nulu.
3. Pozivaju se inicijalizatori članova po redosledu deklaracije.
4. Izvršava se telo konstruktora izvedene klase.

Postoji jedna dobra strana svega ovoga – sve je barem inicijalizovano nulama (ili šta god nula značila za određeni tip) i nije ostavljeno sa slučajno zatečenim vrednostima koje su beskorisne. To uključuje i reference na objekte koje su ugrađene u klasu preko kompozicije i dobijaju vrednost **null**. Stoga, ako takvu referencu zaboravite da inicijalizujete, javiće se izuzetak prilikom izvršavanja. Sve drugo dobija vrednost nula, što je obično siguran znak da ćete u rezultatu videti grešku.

S druge strane, trebalo bi da vas prilično užasne rezultat ovog programa. Postupili ste logično, a ipak je njegovo ponašanje tajanstveno pogrešno, a prevodilac se nije bunio. (U ovakvim situacijama se C++ ponaša razboritije.) Ovakve greške lako mogu da budu duboko ukopane i njihovo pronalaženje iziskuje puno vremena.

Zbog toga, dobar savet za pisanje konstruktora glasi: „Uradite što manje posla da objekat dovedete u ispravno stanje i ako ikako možete, izbegavajte da pozivate bilo kakve druge metode u toj klasi". Sa sigurnošću možete da pozivate samo finalne metode osnovne klase. (Ovo se automatski odnosi i na privatne metode, koje su automatski i finalne.) One ne mogu da budu redefinisane i stoga ne kriju ovakvu vrstu iznenađenja. Nećete uvek moći da se pridržavate ovog saveta, ali to je ideal kojem treba težiti.

Vežba 15: (2) Datoteci **PoliKonstruktori.java** dodajte **PravougaoniGlif** i pokažite problem opisan u ovom odeljku.

Kovarijantni povratni tipovi

Java SE5 je donela i *kovarijantne povratne tipove*, što znači da redefinisana metoda izvedene klase može vraćati tip izveden iz tipa koji vraća metoda osnovne klase:

```
//: polimorfizam/KovarijantnoVracanje.java

class Zito {
  public String toString() { return "Žito"; }
}

class Psenica extends Zito {
  public String toString() { return "Pšenica"; }
}
```

```
class Mlin {
  Zito obradi() { return new Zito(); }
}

class PsenicaMlin extends Mlin {
  Psenica obradi() { return new Psenica(); }
}

public class KovarijantnoVracanje {
  public static void main(String[] args) {
    Mlin m = new Mlin();
    Zito z = m.obradi();
    System.out.println(z);
    m = new PšenicaMlin();
    z = m.obradi();
    System.out.println(z);
  }
} /* Ispis:
Žito
Pšenica
*///:~
```

Ključna razlika između Jave SE5 i ranijih verzija Jave leži u sledećem: ranije verzije bi naterale redefinisanu verziju metode **obradi**() da vrati objekat klase **Zito**, a ne **Psenica**, iako je **Psenica** izvedena iz klase **Zito** i stoga je i dalje legitiman povratni tip. Kovarijantni povratni tipovi omogućavaju specifičniji povratni tip **Psenica**.

Projektovanje pomoću nasleđivanja

Kada shvatite polimorfizam, može vam se učiniti da bi sve trebalo da nasleđujete jer je polimorfizam tako pametna alatka. To može da optereti projekat; naime, ako odmah upotrebite nasleđivanje kada od postojeće klase pravite novu, stvari mogu bez potrebe da se komplikuju.

Bolji pristup je da najpre izaberete kompoziciju, naročito ako nije očigledno šta treba da upotrebite. Kompozicija ne nameće uvođenje hijerarhije nasleđivanja u projekat. Kompozicija je takođe fleksibilnija, jer ste u mogućnosti da tip odaberete dinamički (a time i ponašanje), dok pri nasleđivanju morate već prilikom prevođenja da znate tačan tip. To ilustruje naredni primer:

```
//: polimorfizam/Kompozicija.java
// Dinamička promena ponašanja objekta
// pomoću kompozicije (projektni obrazac "stanje")
import static net.mindview.util.Print.*;

class Glumac {
  public void glumi();
}
```

```
class SrecanGlumac extends Glumac {
  public void glumi() { print("SrecanGlumac"); }
}

class TuzanGlumac extends Glumac {
  public void glumi() { print("TuzanGlumac");  }
}

class Pozornica {
  private Glumac glumac = new SrecanGlumac();
  public void promeni() { glumac = new TuzanGlumac(); }
  public void kreni() { glumac.glumi(); }
}

public class Kompozicija {
  public static void main(String[] args) {
    Pozornica pozornica = new Pozornica();
    pozornica.kreni();
    pozornica.promeni();
    pozornica.kreni();
  }
} /* Ispis:
SrecanGlumac
TuzanGlumac
*///:~
```

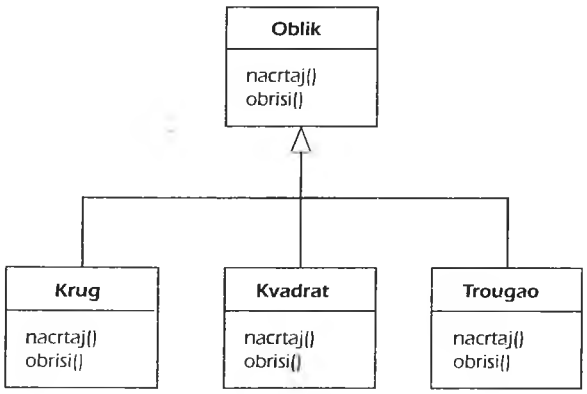
Objekat **Pozornica** sadrži referencu na klasu **Glumac** koja je inicijalizovana objektom klase **SrecanGlumac**. To određuje ponašanje metode **kreni**(). Ali pošto referenca prilikom izvršavanja može da bude povezana s drugačijim objektom, referenca **glumac** može da bude izmenjena tako da pokazuje na objekat klase **TuzanGlumac**, čime se ponašanje metode **kreni**() menja. Znači, dobili ste dinamičku fleksibilnost pri izvršavanju. (Ovo se takođe naziva i obrazac *State* (Stanje). Pogledajte knjigu *Thinking in Patterns (with Java)* na adresi *www.MindView.net*.) Nasuprot tome, prilikom izvršavanja ne možete da menjate šta ćete naslediti; to mora potpuno da bude određeno već prilikom prevođenja.

Opšta smernica je „upotrebljavajte nasleđivanje da izrazite razlike u ponašanju, a polja da izrazite promene stanja". U prethodnom primeru koristi se i jedno i drugo: dve različite klase su nasleđene kako bi iskazale razliku u metodi **glumi**(), a u klasi **Pozornica** upotrebljava se kompozicija kako bi njeno stanje moglo da se menja. U ovom slučaju, promena stanja povlači i promenu ponašanja.

Vežba 16: (3) Prateći primer iz datoteke **Kompozicija.java**, napravite klasu **SvemirskiBrod** u kojoj se nalazi referenca na klasu **Uzbuna** koja može da prikaže tri različita stanja uzbune. Napravite i metode koje mogu da menjaju ta stanja.

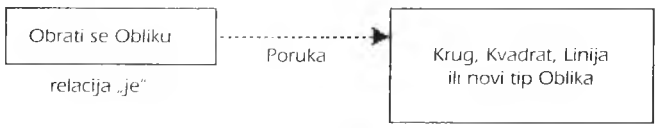
Poređenje zamene i proširivanja

Čini se da je najbolji pristup za pravljenje hijerarhije nasleđivanja upotreba „potpunog" pristupa. Odnosno, u izvedenoj klasi biće redefinisane samo metode ustanovljene u osnovnoj klasi, kao što je pokazano na narednom dijagramu:



Ovo može da bude označeno kao potpuna relacija „je", jer interfejs klase određuje šta ova hijerarhija predstavlja. Nasleđivanje garantuje da će bilo koja izvedena klasa imati interfejs osnovne klase i ništa manje. Ako pratite ovaj dijagram, uočićete da izvedene klase takođe neće imati *ništa više* osim interfejsa osnovne klase.

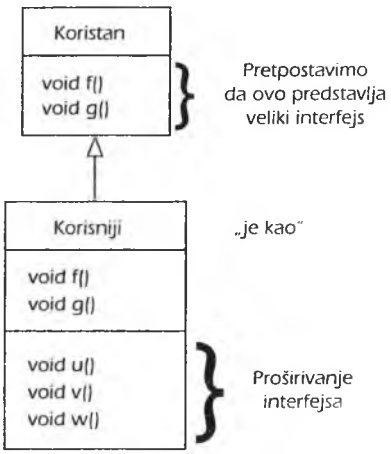
To može da se posmatra kao *potpuna zamena*, jer objekti izvedenih klasa mogu savršeno da se zamene osnovnom klasom, pa kada ih koristite ne morate o njima imati nikakve dodatne informacije:



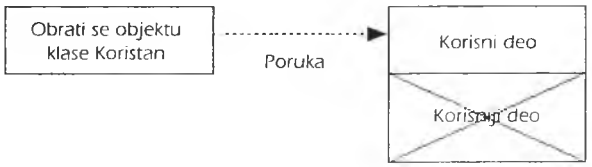
Osnovna klasa može da primi svaku poruku koju možete da pošaljete izvedenoj klasi, jer one imaju potpuno isti interfejs. Treba samo da uradite svođenje naviše iz izvedene klase pa nikada više nećete morati da razmišljate o tome s kojim tipom radite. Polimorfizam se stara o svemu drugom.

Kada to na ovaj način posmatrate, čini se da je potpuna relacija „je" jedini logični način da takve stvari uradite i da je bilo koji drugi način projektovanja nejasan i samim tim loš. To je takođe zamka. Čim počnete da razmišljate na taj način, pogledaćete oko sebe i ustanoviti da je proširivanje interfejsa (što nažalost rezervisana reč **extends** podstiče) savršeno rešenje određenog problema. To se može nazvati relacijom „je kao", jer je izvedena

klasa *poput* osnovne klase – ima isti osnovni interfejs – ali ima i druge mogućnosti koje zahtevaju realizaciju dodatnih metoda:



Iako je ovo koristan i razuman pristup (u zavisnosti od situacije), on ima svoje mane. Prošireni deo interfejsa izvedene klase nije dostupan osnovnoj klasi, stoga nakon svođenja naviše ne možete da pozivate nove metode:



Ako u ovom slučaju ne obavljate svođenje naviše, to vas neće opterećivati, ali često možete da dođete u situaciju kada treba ponovo otkriti tačan tip objekta kako biste mogli da pristupite njegovim dodatnim metodama. U narednom odeljku pokazujemo kako se to radi.

Svođenje naviše i podaci o tipu prilikom izvršavanja

Pošto prilikom *svođenja naviše* (kretanja uz hijerarhiju nasleđivanja) gubite informaciju o tačnom tipu, ima smisla da za ponovno dobijanje informacije o tipu – odnosno za kretanje nazad niz hijerarhiju nasleđivanja – upotrebljavate *svođenje naniže* (engl. *downcast*). Naučili ste da je svođenje naviše uvek sigurno, jer osnovna klasa ne može da ima širi interfejs od izvedene klase, pa će stoga svaka poruka koju pošaljete preko interfejsa osnovne klase biti prihvaćena. Ali prilikom svođenja naniže ne znate da je neki oblik (na primer) zaista krug – mogao bi da bude trougao, kvadrat ili nešto drugo.

Da bi se ovaj problem rešio, mora da postoji način za garantovanje ispravnog svođenja naniže, kako ne biste greškom uradili svođenje u pogrešan tip i zatim poslali poruku koju objekat ne može da primi. To bi bilo veoma opasno.

U nekim jezicima (poput C++-a), morate da upotrebite posebnu operaciju da biste dobili sigurno svođenje naniže. Međutim, u Javi se proveravaju *sva svođenja*! Iako naizgled obavljate običnu konverziju, prilikom izvršavanja se utvrđuje da li je to zaista onaj tip koji vi mislite da jeste. Ako to nije slučaj, doći će do izuzetka tipa **ClassCastException**. Čin proveravanja tipova prilikom izvršavanja naziva se *podaci o tipovima prilikom izvršavanja* (engl. *RTTI – run-time type information*). Naredni primer pokazuje kako RTTI radi:

```
//: polimorfizam/RTTI.java
// Svođenje naniže i podaci o tipovima prilikom izvršavanja (RTTI).
//  {ThrowsException}

class Koristan {
  public void f() {}
  public void g() {}
}

class Korisniji extends Koristan {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Koristan[] x = {
      new Koristan(),
      new Korisniji()
    };
    x[0].f();
    x[1].g();
    // Prilikom prevođenja: metoda nije pronađena u klasi Koristan
    //! x[1].u();
    ((Korisniji)x[1]).u(); // Svođenje naniže/RTTI
    ((Korisniji)x[0]).u(); // biće bačen izuzetak
  }
} ///:~
```

Kao što se vidi s prethodnog dijagrama, klasa **Korisniji** proširuje interfejs klase **Koristan**. Pošto je nasleđena, ona takođe može da bude svedena naviše ka klasi **Koristan**. To možete da vidite u okviru inicijalizacije niza **x** u metodi **main()**. Pošto oba objekta u nizu pripadaju klasi **Koristan**, za oba možete da pozovete metode **f()** i **g()**, a ako pokušate da pozovete metodu **u()** (koja postoji samo u klasi **Korisniji**), dobićete grešku pri prevođenju.

Ukoliko želite da pristupite proširenom interfejsu objekta klase **Korisniji**, primenite svođenje naniže. Ako je tip odgovarajući, ono će biti uspešno. U suprotnom, dobićete izuzetak tipa **ClassCastException**. Za obradu ovog izuzetka nije neophodno da pišete poseban kôd, jer on naznačava programersku grešku koja je mogla da se desi bilo gde u programu. Komentar – oznaka **{ThrowsException}** – kazuje sistemu za automatizovano prevođenje i pakovanje (engl. *build*) primera iz ove knjige kako očekuje da će ovaj program prilikom izvršavanja generisati izuzetak.

RTTI je više od obične konverzije. Postoji, na primer, način da ustanovite tip *pre* nego što pokušate da obavite njegovo svođenje naniže. Kompletno poglavlje *Podaci o tipovima* posvećeno je proučavanju različitih vidova podataka o tipovima prilikom izvršavanja u Javi.

Vežba 17: (2) Koristeći hijerarhiju **Cikl** iz primera 1, metodu **ravnoteža**() dodajte klasama **Unicikl** i **Bicikl**, ali ne i klasi **Tricikl**. Napravite primerke sva tri tipa i svedite ih naviše na niz tipa **Cikl**. Pozovite metodu **ravnoteža**() za svaki element tog niza i pogledajte rezultate. Svedite naniže, pozovite metodu **ravnoteža**() i pogledajte šta se dešava.

Sažetak

Polimorfizam znači „različiti oblici". U objektno orijentisanom programiranju imate isti interfejs osnovne klase i različite oblike koji upotrebljavaju taj interfejs: različite verzije dinamički povezanih metoda.

U ovom poglavlju videli ste da nije moguće razumeti niti primeniti polimorfizam bez upotrebe apstrakcije podataka i nasleđivanja. Polimorfizam je osobina koja ne može da se posmatra izolovano (kao što može, na primer, naredba **switch**), već samo kao deo „veće slike" relacija između klasa.

Da biste u programima efikasno koristili polimorfizam – a samim tim i objektno orijentisane tehnike – morate da proširite svoje poimanje programiranja. Pored članova i poruka zasebne klase, uzmite u obzir i zajedničke osobine među klasama kao i njihove međusobne relacije. Iako to iziskuje značajan napor, on se i te kako isplati. Postiže se brži razvoj programa, bolja organizacija koda, prošireni programi i lakše održavanje koda.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.net*.

Interfejsi

Interfejsi i apstraktne klase pružaju strukturiraniji način razdvajanja interfejsa od realizacije.

Takvi mehanizmi nisu tako česti u programskim jezicima. Primera radi, C++ ima samo posrednu podršku za te koncepte. Činjenica da oni postoje u Javi, govori da su ih projektanti jezika smatrali dovoljno bitnim da se za njih obezbedi direktna podrška kroz rezervisane reči.

Najpre ćemo razmotriti *apstraktne klase* koje su otprilike na pola puta između obične klase i interfejsa. Iako ćete biti skloniji pravljenju interfejsa, apstraktna klasa je važna i neophodna alatka za pravljenje klasa koje imaju neke nerealizovane metode. Ne možete uvek koristiti čist interfejs.

Apstraktne klase i metode

U primeru sa instrumentima u prethodnom poglavlju, metode u osnovnoj klasi **Instrument** uvek su bile „lažne" metode. Ako bi one ikad bile pozvane, posredi bi bila neka greška, zato što je namena klase **Instrument** da obezbedi *zajednički interfejs* za sve iz nje izvedene klase.

U tim primerima, jedini razlog za pravljenje zajedničkog interfejsa jeste da se omogući različito ponašanje svakog pojedinačnog podtipa. On uspostavlja osnovni izgled, tako da znate šta je zajedničko za sve izvedene klase. Drugi način da to iskažete jeste da proglasite klasu **Instrument** *apstraktnom osnovnom klasom* (ili jednostavno *apstraktnom klasom*).

Ako imate apstraktnu klasu, poput klase **Instrument**, njeni objekti gotovo nikad nemaju nikakvo značenje. Apstraktnu klasu pravite kada želite da s grupom klasa radite preko zajedničkog interfejsa. Tako **Instrument** služi samo da se opiše interfejs, a ne određena realizacija – stoga pravljenje objekata te klase nema smisla i verovatno ćete poželeti da sprečite korisnika da to uradi. To možete da postignete tako što ćete svim metodama u klasi **Instrument** zadati da ispisuju poruku o grešci, ali time informaciju odlažete sve do izvršavanja i od korisnika zahtevate prilično temeljno testiranje. Uvek je bolje prijaviti probleme u vreme prevođenja.

Java za to obezbeđuje mehanizam koji se naziva *apstraktna metoda*.[1] To je nepotpuna metoda, ona ima samo deklaraciju bez tela. Sintaksa za deklaraciju apstraktne metode je sledeća:

```
abstract void f();
```

Klasa koja sadrži apstraktne metode naziva se *apstraktna klasa*. Ako klasa sadrži jednu ili više apstraktnih metoda, ona mora biti kvalifikovana kao apstraktna. (U suprotnom, prevodilac će prijaviti grešku.)

Pošto je apstraktna klasa nepotpuna, šta bi prevodilac trebalo da uradi kada neko pokuša da napravi objekat takve klase? Pošto ne može da napravi objekat apstraktne klase na siguran način, prevodilac će prijaviti grešku. Na ovaj način se obezbeđuje potpunost apstraktne klase i vi ne morate da brinete o njenoj eventualnoj pogrešnoj upotrebi.

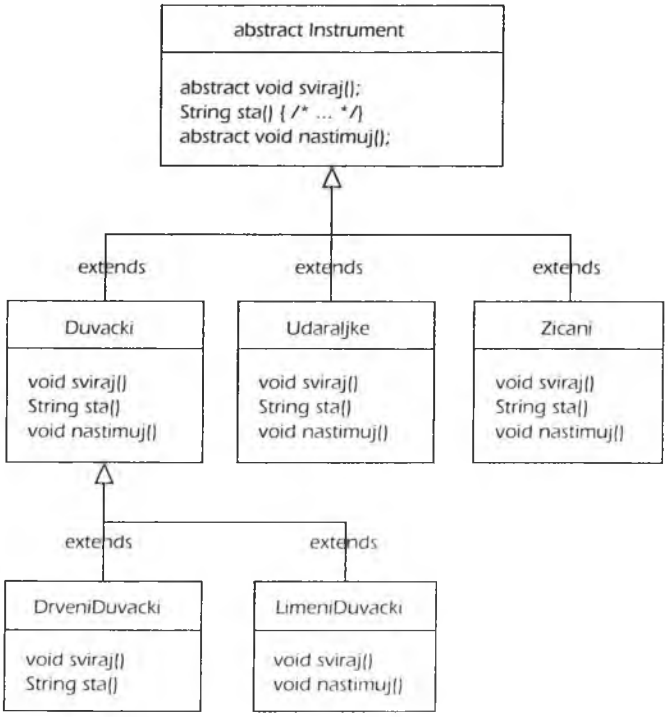
---

[1] Za programere na C++-u one su jednake *potpuno virtuelnim funkcijama*.

Ako apstraktnu klasu nasledite i želite da pravite objekte tog novog tipa, morate da obezbedite definicije za sve apstraktne metode osnovne klase. Ukoliko to ne uradite (a možete i tako da odlučite), tada i izvedena klasa ostaje apstraktna, te će vas prevodilac prisiliti i da *tu* klasu označite rezervisanom reči **abstract**.

Apstraktnu klasu možete da napravite i tako da ne sadrži nijednu apstraktnu metodu. To je korisno kada u klasi nema smisla praviti bilo kakve apstraktne metode, ali ipak želite da sprečite pravljenje objekata te klase.

Klasa **Instrument** iz prethodnog poglavlja lako može da bude pretvorena u apstraktnu klasu. Samo neke metode će biti apstraktne – ne moraju sve. Evo kako to izgleda:



Sledi izmenjeni primer sa orkestrom uz upotrebu apstraktnih klasa i metoda:

```
//: interfejsi/muzika4/Muzika4.java
// Apstraktne klase i metode
package interfejsi.muzika4;
import polimorfizam.muzika.Nota;
import static net.mindview.util.Print.*;

abstract class Instrument {
  private int i; // Zauzima se memorija za svaki objekat klase Instrument
  public abstract void sviraj(Nota n);
  public String sta() { return "Instrument"; }
```

```
  public abstract void nastimuj();
}

class Duvacki extends Instrument {
  public void sviraj(Nota n) {
    print("Duvacki.sviraj() " + n); {
  }
  public String sta() { return "Duvacki"; }
  public void nastimuj() {}
}

class Udaraljke extends Instrument {
  public void sviraj(Nota n) {
    print("Udaraljke.sviraj() " + n);
  }
  public String sta() { return "Udaraljke"; }
  public void nastimuj() {}
}

class Zicani extends Instrument {
  public void sviraj(Nota n) {
    print("Zicani.sviraj() " + n);
  }
  public String sta() { return "Zicani"; }
  public void nastimuj() {}
}
class LimeniDuvacki extends Duvacki {
  public void sviraj(Nota n) {
    print("LimeniDuvacki.sviraj() " + n);
  }
  public void nastimuj() {  print("LimeniDuvacki.nastimuj()");  }
}

class DrveniDuvacki extends Duvacki {
  public void sviraj(Nota n) {
    print("DrveniDuvacki.sviraj() " + n);
  }
  public String sta() { return "DrveniDuvacki"; }
}

public class Muzika4 {
  // Ne vodi računa o tipovima pa će novi tipovi
  // koje dodamo sistemu i dalje pravilno da rade:
  static void melodija(Instrument i) {
    // ...
    i.sviraj(Nota.SREDNJE_C);
  }
  static void sveMelodije(Instrument[] e) {
    for(Instrument i : e)
      melodija(i);
  }
```

```
  public static void main(String[] args) {
    // Svođenje naviše prilikom dodavanja u niz:
    Instrument[] orkestar = {
      new Duvacki();
      new Udaraljke();
      new Zicani();
      new LimeniDuvacki();
      new DrveniDuvacki();
    };
    sveMelodije(orkestar);
  }
} /* Ispis:
Duvacki.sviraj() SREDNJE_C
Udaraljke.sviraj() SREDNJE_C
Zicani.sviraj() SREDNJE_C
LimeniDuvacki.sviraj() SREDNJE_C
DrveniDuvacki.sviraj() SREDNJE_C
*///:-
```

Obratite pažnju na to da je izmenjena samo osnovna klasa.

Pravljenje apstraktnih metoda i klasa je korisno jer izričito određuje apstraktnost klase i objašnjava njenu namenu i korisniku i prevodiocu. Takođe, apstraktne klase su korisne alatke za ponovnu podelu na proste faktore, jer omogućavaju lako pomeranje zajedničkih metoda naviše u hijerarhiji nasleđivanja.

Vežba 1: (1) Izmenite vežbu 9 iz prethodnog poglavlja tako da klasa **Glodar** postane apstraktna klasa. Gde god je moguće, pretvorite metode klase **Glodar** u apstraktne.

Vežba 2: (1) Napravite apstraktnu klasu koja nema nijednu apstraktnu metodu i uverite se da ne nije moguće napraviti instance te klase.

Vežba 3: (2) Napravite osnovnu klasu s apstraktnom metodom **ispisi()** koju ćete redefinisati u izvedenoj klasi. Redefinisana verzija te metode treba da ispisuje vrednost celobrojne promenljive definisane u izvedenoj klasi. Promenljivoj zadajte neku vrednost različitu od nule na mestu definicije. Pozovite tu metodu u konstruktoru osnovne klase. U metodi **main()** napravite objekat izvedenog tipa i potom pozovite njegovu metodu **ispisi()**. Objasnite rezultate.

Vežba 4: (3) Napravite apstraktnu klasu koja nema nijednu metodu. Nasledite je i dodajte jednu metodu. Napravite statičnu metodu čiji je argument referenca na osnovnu klasu. Svedite tu referencu naniže ka izvedenoj klasi i pozovite metodu. U metodi **main()** pokažite da to radi. Sada ubacite apstraktnu deklaraciju u osnovnu klasu i eliminišite potrebu za svođenjem naniže.

Interfejsi

Rezervisana reč **interface** unapređuje koncept apstrakcije. Rezervisana reč **abstract** omogućava da u klasi napravite jednu ili više metoda koje nemaju definicije – time obezbeđujete deo interfejsa, ali izostavljate odgovarajuću realizaciju, koju će napraviti naslednici te klase. Rezervisana reč **interface** označava potpuno apstraktnu klasu, u kojoj uopšte nema

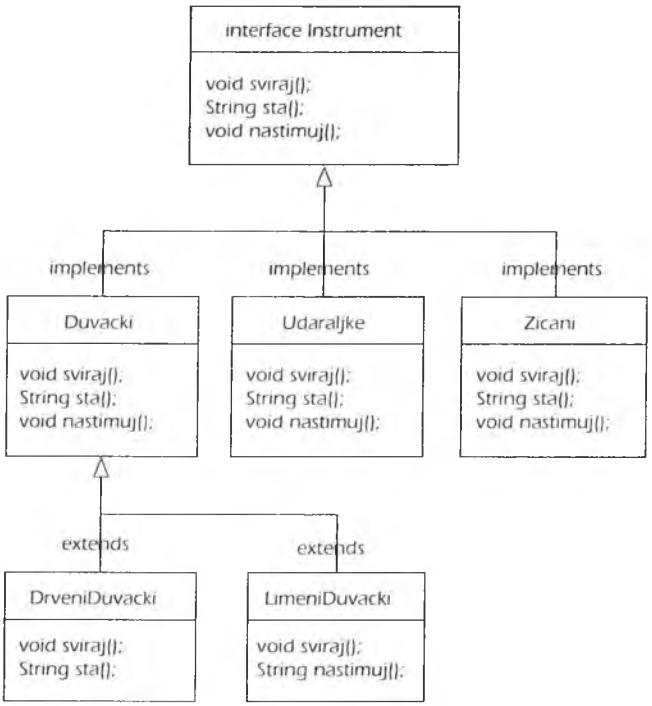
nikakve realizacije. Interfejs autoru omogućava da definiše izgled klase: imena metoda, liste argumenata i povratne tipove, ali ne i tela metoda. On obezbeđuje samo formu, ali ne i realizaciju.

Interfejs nam govori: „Ovako će izgledati sve klase koje *realizuju* ovaj interfejs". Stoga svaki kôd koji upotrebljava određeni interfejs zna koje metode interfejsa može da koristi i to je sve. Interfejs se, znači, koristi da ustanovi „komunikacioni protokol" između klasa. (Neki objektno orijentisani programski jezici za istu namenu koriste rezervisanu reč *protocol*.)

Međutim, **interface** nije samo apstraktna klasa dovedena do krajnjih granica. On vam omogućava da dobijete varijantu „višestrukog nasleđivanja" C++-a, jer pomoću njega možete da napravite klasu koja se može svesti naviše prema više osnovnih tipova.

Da biste napravili interfejs, umesto rezervisane reči **class** upotrebite rezervisanu reč **interface**. Kao i kod klasa, ispred rezervisane reči **interface** možete da dodate i rezervisanu reč **public** (ali samo ako je taj interfejs definisan u datoteci istog imena). Ukoliko je izostavite, dobijate paketni pristup, tako da taj interfejs možete da upotrebljavate samo unutar istog paketa. Interfejs može da sadrži i polja, ali su ona implicitno (automatski) definisana kao statična i finalna.

Da biste napravili klasu koja zadovoljava određeni interfejs (ili grupu interfejsa), upotrebite rezervisanu reč **implements**. Time govorite: „njen izgled je definisan datim interfejsom, a sada ću definisati kako će ona da *radi*". Izuzev te izmene, sve ostalo je kao kod nasleđivanja. Izloženi koncept je prikazan na dijagramu primera s instrumentima:



Iz klasa **DrveniDuvacki** i **LimeniDuvacki** možete videti sledeće: kada realizujete interfejs, ta realizacija postaje obična klasa koja može da se proširuje na uobičajene načine.

Deklaracije metoda u interfejsu možete eksplicitno da proglasite javnim. One su, međutim, javne čak i ako to ne učinite. Stoga prilikom realizacije interfejsa, sve metode iz interfejsa moraju da budu proglašene javnim. U suprotnom bi dobile podrazumevani paketni pristup, čime biste umanjili nivo pristupa metodama prilikom nasleđivanja, što Java prevodilac ne dozvoljava.

To možete da vidite u izmenjenoj verziji primera sa instrumentima. Zapamtite da je svaka metoda u interfejsu samo deklaracija i prevodilac neće dopustiti ništa drugo. Pored toga, iako nijedna metoda interfejsa **Instrument** nije deklarisana kao javna, to se automatski obavlja:

```
//: interfejsi/muzika5/Muzika5.java
// Interfejsi.
package interfejsi.muzika5;
import polimorfizam.muzika.Nota;
import static net.mindview.util.Print.*;

interface Instrument {
  // Konstanta pri prevođenju:
  int IZNOS = 5; // statična i finalna
  // Ne može da ima definicije metoda, već samo deklaracije:
  void sviraj(Nota n); // Automatski javne
  void nastimuj();
}

class Duvacki implements Instrument {
  public void sviraj(Nota n) {
    print(this + ".sviraj() " + n);
  }
  public String toString() { return "Duvacki"; }
  public void nastimuj() { print(this + ".nastimuj()"); }
}

class Udaraljke implements Instrument {
  public void sviraj(Nota n) {
    print(this + ".sviraj() " + n);
  }
  public String toString() { return "Udaraljke"; }
  public void nastimuj() { print(this + ".nastimuj()"); }
}
class Zicani implements Instrument {
  public void sviraj(Nota n) {
    print(this + ".sviraj() " + n);
  }
  public String toString() { return "Zicani"; }
  public void nastimuj() { print(this + ".nastimuj()"); }
}
```

```
class LimeniDuvacki extends Duvacki {
  public String toString() { return "LimeniDuvacki"; }
}

class DrveniDuvacki extends Duvacki {
  public String toString() { return "DrveniDuvacki"; }
}

public class Muzika5 {
  // Ne vodi računa o tipovima pa će i novi tipovi
  // koje dodate u sistem ispravno raditi:
  static void melodija(Instrument i) {
    // ...
    i.sviraj(Nota.SREDNJE_C);
  }
  static void sveMelodije(Instrument[] e) {
    for(Instrument i : e)
      melodija(i);
  }
  public static void main(String[] args) {
    // Svođenje naviše prilikom dodavanja u niz:
    Instrument[] orkestar = new {
      new Duvacki();
      new Udaraljke();
      new Zicani();
      new LimeniDuvacki();
      new DrveniDuvacki();
    };
    sveMelodije(orkestar);
  }
} /* Ispis:
Duvacki.sviraj() SREDNJE_C
Udaraljke.sviraj() SREDNJE_C
Zicani.sviraj() SREDNJE_C
LimeniDuvacki.sviraj() SREDNJE_C
DrveniDuvacki.sviraj() SREDNJE_C
*///:~
```

U ovoj verziji primera učinjena je još jedna izmena: metoda **sta**() prepravljena je u **toString**(), jer se koristila upravo na taj način. Pošto je **toString**() deo korenske klase **Object**, ne mora da se pojavi u ovom interfejsu.

Ostatak programa radi kao i ranije. Nije bitno da li obavljate svođenje naviše ka običnoj klasi **Instrument**, apstraktnoj klasi **Instrument** ili interfejsu **Instrument**. Ponašanje je identično. U metodi **melodija**(), u stvari, ne postoji nijedan trag o tome da li je **Instrument** „obična" klasa, apstraktna klasa ili interfejs.

Vežba 5: (2) U zasebnom paketu napravite interfejs koji sadrži tri metode. Realizujte taj interfejs u nekom drugom paketu.

Vežba 6: (2) Dokažite da su sve metode interfejsa automatski javne.

Vežba 7: (1) Izmenite vežbu 9 iz poglavlja *Polimorfizam* tako da klasu **Glodar** pretvorite u interfejs.

Vežba 8: (2) U datoteci **polimorfizam.Sendvic.java** napravite interfejs pod imenom **BrzaHrana** (sa odogovarajućim metodama) i promenite **Sendvic** tako da realizuje i interfejs **BrzaHrana**.

Vežba 9: (3) Izmenite program **Muzika5.java** praveći apstraktnu klasu koja sadrži zajedničke metode potklasa **Duvacki**, **Udaraljke** i **Zicani**.

Vežba 10: (3) Izmenite datoteku **Muzika5.java** dodajući interfejs **MozeDaSvira**. Deklaraciju metode **sviraj( )** prebacite iz interfejsa **Instrument** u **MozeDaSvira**. Izvedenim klasama dodajte i interfejs **MozeDaSvira**, ubacujući ga u listu iza rezervisane reči **implements**. Izmenite metodu **melodija( )** tako da njen argument bude interfejs **MozeDaSvira**, umesto interfejsa **Instrument**.

Potpuno razdvajanje

Kad god metoda radi s klasom umesto sa interfejsom, ograničeni ste na upotrebu te klase i njenih podklasa. Ukoliko biste baš hteli da primenite tu metodu na neku klasu koja nije u toj hijerarhiji, nemate sreće. Interfejs znatno slabi to ograničenje. Stoga možete da pišete kôd koji je lakše upotrebiti više puta.

Na primer, pretpostavimo da imate klasu **Procesor** s metodama **ime( )** i **obradi( )** koja prima ulazne podatke, modifikuje ih i šalje na izlaz. Osnovna klasa se proširuje prilikom pravljenja više različitih vrsta **Procesor**a. U ovom slučaju, podtipovi **Procesor**a modifikuju **String** objekte (vodite računa o tome da kovarijantni mogu biti povratni tipovi, ali ne i tipovi argumenata):

```
//: interfejsi/klasaprocesor/Primeni.java
package interfejsi.klasaprocesor;
import java.util.*;
import static net.mindview.util.Print.*;

class Procesor {
  public String ime() {
    return getClass().getSimpleName();
  }
  Object obradi(Object ulaz) { return ulaz; }
}

class VelikimSlovima extends Procesor {
  String obradi(Object ulaz) { // Kovarijantno vraćanje
    return ((String)ulaz).toUpperCase();
  }
}

class MalimSlovima extends Procesor {
  String obradi(Object ulaz) {
```

```
    return ((String)ulaz).toLowerCase();
  }
}

class Delilac extends Procesor {
  String obradi(Object ulaz) {
    //  Argument metode split() je graničnik po kojem se znakovni niz deli
   //  na sastavne delove:
    return Arrays.toString(((String)ulaz).split(" "));
  }
}

public class Primeni {
  public static void process(Procesor p, Object s) {
    print("Uključen procesor " + p.ime());
    print(p.obradi(s));
  }
  public static String s =
    "Disagreement with beliefs is by definition incorrect";
  public static void main(String[] args) {
    obradi(new VelikimSlovima(), s);
    obradi(new MalimSlovima(), s);
    obradi(new Delilac(), s);
  }
} /* Ispis:
Uključen procesor VelikimSlovima
DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
Uključen procesor MalimSlovima
disagreement with beliefs is by definition incorrect
Uključen procesor Delilac
[Disagreement, with, beliefs, is, by, definition, incorrect]
*///:~
```

Metoda **Primeni.obradi**() prima **Procesor** bilo koje vrste i primenjuje ga na **Object**, a zatim ispisuje rezultat. Pravljenje metode koja se ponaša različito, u zavisnosti od argumenta objekta koji joj je prosleđen, naziva se projektni obrazac *Strategy* (strategija). Ta metoda sadrži nepromenljiv deo algoritma koji treba izvršiti, dok promenljiv deo sadrži Strategija. Strategija je objekat koji prosledujete; on sadrži kôd koji treba izvršiti. Ovde je Strategija objekat tipa **Procesor**, a u metodi **main**() vidite tri različite Strategije primenjene na objekte tipa **Strings**.

Metoda **split**() je deo klase **String**. Ona prima **String** objekat i deli ga koristeći svoj argument kao graničnik, a vraća **String**[]. Ovde je upotrebljena za brže pravljenje niza objekata tipa **String**.

Pretpostavimo da ste pronašli skup elektronskih filtara koji su prividno prikladni za metodu **Primeni.obradi**():

```
//: interfejsi/filtri/Talasnioblik.java
package interfejsi.filtri;
```

```
public class Talasnioblik {
  private static long brojac;
  private final long id = brojac++;
  public String toString() { return "Talasnioblik " + id; }
} ///:~

//: interfejsi/filtri/Filtar.java
package interfejsi.filtri;
public class Filtar {
  public String ime() {
    return getClass().getSimpleName();
  }
  public Talasnioblik obradi(Talasnioblik ulaz) { return ulaz; }
} ///:~

//: interfejsi/filtri /Niskopropusni.java
package interfejsi.filtri;

public class Niskopropusni extends Filtar {
  double lom;
  public Niskopropusni(double lom) { this.lom = lom; }
  public Talasnioblik obradi(Talasnioblik ulaz) {
    return ulaz; // Lažna obrada
  }
} ///:~

//: interfejsi/filtri/Visokopropusni.java
package interfejsi.filtri;

public class Visokopropusni extends Filtar {
  double lom;
  public Visokopropusni(double lom) { this.lom = lom; }
  public Talasnioblik obradi(Talasnioblik ulaz) { return ulaz; }
} ///:~

//: interfejsi/filtri/PojasniPropust.java
package interfejsi.filtri;

public class PojasniPropust extends Filtar {
  double donjiLom, gornjiLom;
  public PojasniPropust(double donjiL, double gornjiL) {
    donjiLom = donjiL;
    gornjiLom = gornjiL;
  }
  public Talasnioblik obradi(Talasnioblik ulaz) { return ulaz; }
} ///:~
```

Filtar ima iste elemente interfejsa kao **Procesor**, ali pošto nije nasledio **Procesor** – jer tvorac klase **Filtar** nije ni pomislio kako biste vi heli da ga upotrebite kao **Procesor** – **Filtar**

ne možete upotrebiti s metodom **Primeni.obradi()**, iako bi to bilo lepo. U suštini, veza između **Primeni.obradi()** i **Procesora** jača je nego što je potrebno, a to sprečava kôd iz **Primeni.obradi()** da se ponovo upotrebljava tamo gde bi to trebalo. Obratite pažnju na to da su i ulazi i izlazi tipa **Talasnioblik**.

Da je **Procesor** interfejs, ograničenja bi bila toliko oslabljena da biste mogli koristiti metodu **Primeni.obradi()** koja uzima taj interfejs. Evo modifikovanih verzija **Procesora** i **Primeni**:

```
//: interfejsi/interfejsprocesor/Procesor.java
package interfejsi.interfejsprocesor;

public interface Procesor {
  String ime();
  Object obradi(Object ulaz);
} ///:~

//: interfejsi/interfejsprocesor/Primeni.java
package interfejsi.interfejsprocesor;
import static net.mindview.util.Print.*;

public class Primeni {
  public static void obradi(Procesor p, Object s) {
    print("Uključen procesor " + p.ime());
    print(p.obradi(s));
  }
} ///:~
```

Prvi način na koji možete ponovo upotrebiti kôd jeste onaj kada programeri klijenti napišu svoje klase tako da budu usaglašene sa ovim interfejsom; na primer:

```
//: interfejsi/interfejsprocesor/ProcesorZnakovnihNizova.java
package interfejsi.interfejsprocesor;
import java.util.*;

public abstract class ProcesorZnakovnihNizova implements Procesor{
  public String ime() {
    return getClass().getSimpleName();
  }
  public abstract String obradi(Object ulaz);
  public static String s =
    "If she weighs the same as a duck, she's made of wood";
  public static void main(String[] args) {
    Primeni.obradi(new VelikimSlovima(), s);
    Primeni.obradi(new MalimSlovima(), s);
    Primeni.obradi(new Delilac (), s);
  }
}
```

```
class VelikimSlovima extends ProcesorZnakovnihNizova {
  public String obradi(Object ulaz) { // Kovarijantno vraćanje
    return ((String)ulaz).toUpperCase();
  }
}

class MalimSlovima extends ProcesorZnakovnihNizova {
  public String obradi(Object ulaz) {
    return ((String)ulaz).toLowerCase();
  }
}

class Delilac extends ProcesorZnakovnihNizova {
  public String obradi(Object ulaz) {
    return Arrays.toString(((String)ulaz).split(" "));
  }
} /* Ispis:
Uključen procesor VelikimSlovima
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Uključen procesor MalimSlovima
if she weighs the same as a duck, she's made of wood
Uključen procesor Delilac
[If, she, weighs, the, same, as, a, duck,, she's, made, of, wood]
*///:~
```

Međutim, često ne možete da modifikujete klase koje hoćete da upotrebite. Za elektronske filtre, na primer, pronađena je gotova biblioteka, a nije pravljena nova. U takvim slučajevima možete upotrebiti projektni obrazac *Adapter* (adapter). Navodimo način kako u Adapteru pišete kod koji će preuzeti postojeći interfejs i napraviti interfejs koji vam je potreban:

```
//: interfejsi/interfejsprocesor/ProcesorFiltar.java
package interfejsi.interfejsprocesor;
import interfejsi.filtri.*;

class FiltarAdapter implements Procesor {
  Filtar filtar;
  public FiltarAdapter(Filtar filtar) {
    this.filtar = filtar;
  }
  public String ime() { return filtar.ime(); }
  public Talasnioblik obradi(Object ulaz) {
    return filter.obradi((Talasnioblik)ulaz);
  }
}

public class FiltarProcesor {
  public static void main(String[] args) {
    Talasnioblik w = new Talasnioblik();
    Primeni.obradi(new FiltarAdapter(new Niskopropusni(1.0)), w);
```

```
    Primeni.obradi(new FiltarAdapter(new Visokopropusni(2.0)), w);
    Primeni.obradi(
      new FiltarAdapter(new PojasniPropust(3.0, 4.0)), w);
  }
} /* Ispis:
Uključen procesor Niskopropusni
Talasnioblik 0
Uključen procesor Visokopropusni
Talasnioblik 0
Uključen procesor PojasniPropust
Talasnioblik 0
*///:~
```

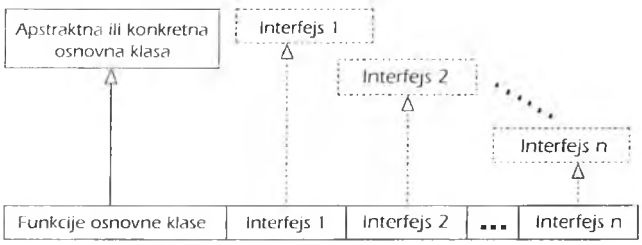
U ovom pristupu Adapteru, konstruktor klase **FiltarAdapter** uzima interfejs koji imate – **Filtar** – i proizvodi objekat koji ima interfejs **Procesor** koji vam treba. Možda ste uočili i delegiranje u klasi **FiltarAdapter**.

Razdvajanje interfejsa od realizacije omogućava da isti interfejs bude primenjen na više različitih realizacija, čime kôd postaje lakše ponovo upotrebljiv.

Vežba 11: (4) Napravite klasu s metodom koja uzima argument tipa **String** i daje rezultat u kojem su zamenjena mesta svakog para znakova u tom argumentu. Prilagodite klasu tako da radi s metodom **interfejsprocesor.Primeni.obradi()**.

Višestruko nasleđivanje u Javi

Pošto interfejs uopšte nema nikakvu realizaciju – odnosno za njega se ne odvaja nikakvo mesto za skladištenje – ništa nas ne sprečava da kombinujemo više interfejsa. To je korisno, jer ponekad treba reći da je „**x** tipa *i* **a** *i* **b** *i* **c**". Ovo kombinovanje interfejsa više klasa, u C++-u se naziva *višestruko nasleđivanje* i sa sobom nosi prilično obiman prtljag, jer svaka klasa može da ima različitu realizaciju neke metode. U Javi možete da uradite isto, ali realizaciju može da ima samo jedna klasa, pa se u Javi prilikom kombinovanja više interfejsa ne javljaju problemi koje smo susretali u C++-u:



Osnovna klasa neke izvedene klase ne mora da bude ni apstraktna ni „konkretna" (klasa koja nema apstraktne metode). Ako nasleđujete nešto *što nije interfejs*, možete da nasledite samo jednu takvu klasu. Svi ostali nasleđeni elementi moraju da budu interfejsi. Sva imena interfejsa moraju biti razdvojena zarezima posle rezervisane reči **implements**.

Možete da navedete koliko god želite interfejsa – svaki postaje nezavisan tip, ka kojem možete da svedete naviše novu klasu. U narednom primeru prikazana je konkretna klasa koja uz nekoliko interfejsa daje novu klasu:

```
//: interfejsi/Avantura.java
// Više interfejsa

interface MozeDaSeBori {
  void boriSe();
}
interface MozeDaPliva {
  void plivaj();
}

interface MozeDaLeti {
  void leti();
}

class AkcioniJunak {
  public void boriSe() {}
}
class Heroj extends AkcioniJunak
    implements MozeDaSeBori, MozeDaPliva, MozeDaLeti {
  public void plivaj() {}
  public void leti() {}
}

public class Avantura {
  public static void t(MozeDaSeBori x) { x.boriSe(); }
  public static void u(MozeDaPliva x) { x.plivaj(); }
  public static void v(MozeDaLeti x) { x.leti(); }
  public static void w(AkcioniJunak x) { x.boriSe(); }
  public static void main(String[] args) {
    Heroj h = new Heroj();
    t(h); // Tretiraj ga kao tip MozeDaSeBori
    u(h); // Tretiraj ga kao tip MozeDaPliva
    v(h); // Tretiraj ga kao tip MozeDaLeti
    w(h); // Tretiraj ga kao tip AkcioniJunak
  }
} ///:~
```

U gornjem primeru vidite da klasa **Heroj** kombinuje klasu **AkcioniJunak** sa interfejsima **MozeDaSeBori**, **MozeDaPliva** i **MozeDaLeti**. Kada na ovaj način konkretnu klasu kombinujete sa interfejsima, prvo morate da navedete tu konkretnu klasu, a posle nje interfejse. (U suprotnom, prevodilac prijavljuje grešku.)

Potpis metode **boriSe**() isti je u interfejsu **MozeDaSeBori** i u klasi **AkcioniJunak**, a metoda **boriSe**() *ne postoji* u definiciji klase **Heroj**. Interfejs možete da nasledite (kao što ćete ubrzo videti), ali pri tom ćete dobiti još jedan interfejs. Ako želite da napravite objekat

novog tipa, morate da napravite klasu u kojoj postoje definicije svih metoda. Iako metoda **boriSe()** nije izričito definisana u klasi **Heroj**, ona dolazi uz klasu **AkcioniJunak**, čime automatski postaje i deo klase **Heroj**, pa je moguće napraviti objekat te klase.

U klasi **Avantura** postoje četiri metode čiji su argumenti razni interfejsi konkretne klase. Kada se napravi objekat klase **Heroj**, on može da se prosledi bilo kojoj od tih metoda, što znači da se može svesti naviše ka svakom od tih interfejsa. To se obavlja bez prepreka i bez posebnog angažovanja programera, zahvaljujući načinu na koji su interfejsi u Javi projektovani.

Zapamtite da je u gornjem primeru pokazan suštinski razlog postojanja interfejsa: mogućnost da se obavi svođenje naviše ka više osnovnih tipova. Međutim, drugi razlog za korišćenje interfejsa isti je kao i za korišćenje apstraktne osnovne klase: da programer klijent ne može da napravi objekat te klase i da bude svestan kako je to samo interfejs.

Zbog toga se javlja pitanje: da li upotrebiti interfejs ili apstraktnu klasu? Ako osnovnu klasu možete da napravite bez definicija metoda i bez promenljivih, uvek odaberite interfejse umesto apstraktnih klasa. U suštini, ako znate da će nešto biti osnovna klasa, prvo pokušajte da to napravite kao interfejs. (O tome ćemo još govoriti u sažetku ovog poglavlja.)

Vežba 12: (2) U datoteci **Avantura.java** dodajte interfejs pod imenom **MozeDaSePenje** prateći oblik ostalih interfejsa u njoj.

Vežba 13: (2) Napravite interfejs koji nasleđuju dva druga interfejsa. Neka treći interfejs nasledi ta dva.[2]

Proširivanje interfejsa nasleđivanjem

Pomoću nasleđivanja, interfejsu lako dodajete nove deklaracije metoda, i kako kombinujete nekoliko interfejsa u jedan nov interfejs. U oba slučaja dobićete nov interfejs, kao što se vidi iz narednog primera:

```
//: interfejsi/HororPredstava.java
// Proširivanje interfejsa nasleđivanjem.

interface Cudoviste {
  void zapreti();
}

interface OpasnoCudoviste extends Cudoviste {
  void unisti();
}

interface Smrtonosno {
  void ubij();
}
```

[2] To će pokazati kako interfejsi sprečavaju „Problem romba" koji se javlja u C++-u prilikom višestrukog nasleđivanja.

```
class Zmaj implements OpasnoCudoviste {
  public void zapreti() {}
  public void unisti() {}
}

interface Vampir extends OpasnoCudoviste, Smrtonosno {
  void pijeKrv();
}
class VeomaZaoVampir implements Vampir  {
  public void zapreti() {}
  public void unisti() {}
  public void ubij() {}
  public void pijeKrv() {}
}

public class HororPredstava {
  static void u(Cudoviste b) { b.zapreti(); }
  static void v(OpasnoCudoviste d) {
    d.zapreti();
    d.unisti();
  }
  static void w(Smrtonosno s) { s.ubij(); }
  public static void main(String[] args) {
    OpasnoCudoviste barni = new Zmaj();
    u(barni);
    v(barni);
    Vampir vlad = new VeomaZaoVampir();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

Interfejs **OpasnoCudoviste** je jednostavno proširenje interfejsa **Cudoviste**. On je realizovan u klasi **Zmaj**.

Sintaksa upotrebljena za interfejs **Vampir** može da se koristi *samo* kada nasleđujete interfejse. Uz rezervisanu reč **extends** možete da upotrebite samo jednu klasu, ali pošto interfejs može da se sastoji od više drugih, prilikom pravljenja novog interfejsa rezervisana reč **extends** može da se odnosi na više interfejsa. Kao što vidite, imena interfejsa samo su razdvojena zarezima.

Vežba 14: (2) Napravite tri interfejsa od kojih svaki ima po dve metode. Nasledite novi interfejs iz ta tri uz dodavanje još jedne nove metode. Napravite klasu realizujući novoformirani interfejs i istovremeno nasleđujući konkretnu klasu. Sada napišite četiri metode od kojih će svaka za argument imati jedan od datih interfejsa. U metodi **main( )** napravite objekat te klase i prosledite ga svakoj metodi.

Vežba 15: (2) Izmenite prethodnu vežbu praveći apstraktnu klasu i iz nje izvedite klasu.

Sukobljavanje imena prilikom kombinovanja interfejsa

Prilikom kombinovanja više interfejsa možete da se suočite s malom začkoljicom. U prethodnom primeru, interfejs **MozeDaSeBori** i klasa **AkcioniJunak** imaju metodu identičnog imena – **void boriSe()**. U našem primeru to nije problem, jer su metode iste u obe klase, ali šta ako se metode razlikuju po potpisu ili povratnom tipu? Sledi primer:

```
//: interfejsi/SukobljavanjeInterfejsa.java
package interfejsi;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // preklopljena
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // preklopljena
}

class C4 extends C implements I3 {
  // Identična, nije problem:
  public int f() { return 1; }
}

// Metode se razlikuju samo po povratnom tipu:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:-
```

Problem se javlja jer su redefinisanje, realizacija i preklapanje nezgodno izmešani, a preklopljene funkcije ne mogu da se razlikuju samo po povratnom tipu. Kada se ukloni komentar ispred poslednja dva reda primera, poruka o grešci sama sve govori:

SukobljavanjeInterfejsa.java:23: f() in C cannot implement f() in I1; attempting to use incompatible return type
found : int
required: void
SukobljavanjeInterfejsa.java:24: interfaces I3 and I1 are incompatible; both define f(), but with different return type

Ako upotrebite ista imena metoda u različitim interfejsima koje nameravate da kombinujete, uglavnom ćete uneti konfuziju i smanjiti čitljivost programa. Trudite se da to izbegnete.

Prilagođavanje interfejsu

Jedan od najvažnijih razloga za upotrebu interfejsa jeste mogućnost postojanja više realizacija istog interfejsa. U jednostavnim slučajevima, to se radi u obliku metode koja prihvata interfejs, a vama ostavlja da ga realizujete i metodi prosledite svoj objekat.

Stoga se interfejsi obično upotrebljavaju u prethodno spomenutom projektnom obrascu *Strategy*. Napišete metodu koja obavlja izvesne operacije, a ta metoda prihvata i interfejs koji vi zadate. U suštini, vi kažete: „Možeš upotrebljavati moju metodu s bilo kojim objektom koji je usaglašen s mojim interfejsom". Time vaša metoda postaje fleksibilnija, opštija i upotrebljivija.

Na primer, konstruktor Java SE5 klase **Scanner** (o kojoj ćete više saznati u poglavlju *Znakovni nizovi*) prima interfejs **Readable**. Videćete da **Readable** nije argument nijedne druge metode u standardnoj biblioteci Jave – napravljen je samo za **Scanner**, tako da **Scanner** ne mora da ograniči svoj argument na određenu klasu. Na taj način, **Scanner** može da radi s više tipova. Ako pravite novu klasu i hoćete da ona bude upotrebljiva s klasom **Scanner**, neka bude **Readable** (čitljiva):

```
//: interfejsi/SlucajneReci.java
// Realizovanje interfejsa tako da bude usaglašen sa određenom metodom.
import java.nio.*;
import java.util.*;

public class SlucajneReci implements Readable {
  private static Random slucajno = new Random(47);
  private static final char[] velikaslova =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static final char[] malas =
    "abcdefghijklmnopqrstuvwxyz".toCharArray();
  private static final char[] samoglasnici =
    "aeiou".toCharArray();
  private int brojac;
  public SlucajneReci(int brojac) { this.brojac = brojac; }
  public int read(CharBuffer cb) {
    if(brojac-- == 0)
      return -1; // Pokazuje kraj ulaza
    cb.append(velikaslova [slucajno.nextInt(velikaslova.length)]);
    for(int i = 0; i < 4; i++) {
      cb.append(samoglasnici[slucajno.nextInt(samoglasnici.length)]);
      cb.append(malas[slucajno.nextInt(malas.length)]);
    }
    cb.append(" ");
    return 10; // Broj dodatih znakova
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new SlucajneReci(10));
    while(s.hasNext())
      System.out.println(s.next());
  }
```

```
} /* Ispis:
Yazeruyac
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
*///:~
```

Interfejs **Readable** zahteva samo realizaciju metode **read()**. U metodi **read()**, argumentu tipa **CharBuffer** dodajete znakove (to se može uraditi na nekoliko načina; videti dokumentaciju klase **CharBuffer**) ili vraćate -1 kada ulaza više nema.

Pretpostavimo da imate klasu koja nije realizovala **Readable** – kako ćete postići da ona radi s klasom **Scanner**? Evo jedne klase koja proizvodi slučajne realne brojeve (s pokretnim zarezom):

```
//: interfejsi/SlucajniDouble.java
import java.util.*;

public class SlucajniDouble {
  private static Random slucajan = new Random(47);
  public double next() { return slucajan.nextDouble(); }
  public static void main(String[] args) {
    SlucajniDoubles sc = new SlucajniDouble();
    for(int i = 0; i < 7; i ++)
      System.out.print(sc.next() + " ");
  }
} /* Ispis:
0.7271157860730044 0.5309454508634242 0.16020656493302599
0.18847866977771732 0.5166020801268457 0.2678662084200585
0.2613610344283964
*///:~
```

Ponovo smo upotrebili projektni obrazac *Adapter*, ali se u ovom slučaju prilagođena klasa može napraviti nasleđivanjem i realizacijom interfejsa **Readable**. Dakle, koristeći pseudovišestruko nasleđivanje koje omogućava rezervisana reč **interface**, napravili smo novu klasu koja je i tipa **SlucajniDouble** i tipa **Readable**:

```
//: interfejsi/PrilagodjeniSlucajniDouble.java
// Pravljenje adaptera s nasleđivanjem.
import java.nio.*;
import java.util.*;

public class PrilagodjeniSlucajniDouble extends SlucajniDouble
implements Readable {
  private int brojac;
```

```
  public PrilagodjeniSlucajniDouble (int brojac) {
    this.brojac = brojac;
  }
  public int read(CharBuffer cb) {
    if(brojac-- == 0)
      return -1;
    String rezultat = Double.toString(next()) + " ";
    cb.append(rezultat);
    return rezultat.length();
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new PrilagodjeniSlucajniDouble(7));
    while(s.hasNextDouble())
      System.out.print(s.nextDouble() + " ");
  }
} /* Ispis:
0.7271157860730044 0.5309454508634242 0.16020656493302599
0.18847866977771732 0.5166020801268457 0.2678662084200585
 0.2613610344283964
*///:~
```

Pošto na ovaj način svakoj postojećoj klasi možete dodati interfejs, znači da metoda koja prima interfejs obezbeđuje i način prilagođenja bilo koje klase za rad s tom metodom. U tome leži moć korišćenja interfejsa umesto klasa.

Vežba 16: (3) Napravite klasu koja proizvodi niz znakova (objekata tipa **char**). Prilagodite tu klasu tako da se može upotrebiti za ulaz objekta tipa **Scanner**.

Polja u interfejsima

Pošto su sva polja koja stavite u interfejs automatski statična i finalna, interfejs je korisna alatka za pravljenje grupa konstantnih vrednosti. Pre Jave SE5, to je bio jedini način da se postigne efekat koji daje rezervisana reč **enum** u C-u ili C++-u. Dakle, naići ćete na ovakav kôd napisan pre Jave SE5:

```
//: interfejsi/Meseci.java
// Upotreba interfejsa za pravljenje grupa konstanti.
package interfejsi;

public interface Meseci {
  int
    JANUAR = 1, FEBRUAR = 2, MART = 3,
    APRIL = 4, MAJ = 5, JUNI = 6, JULI = 7,
    AVGUST = 8, SEPTEMBAR = 9, OKTOBAR = 10,
    NOVEMBAR = 11, DECEMBAR = 12;
} ///:~
```

Obratite pažnju na to da je i ovde upotrebljen uobičajeni stil pisanja statičnih finalnih polja u Javi, čije su vrednosti inicijalizovane konstantama – sve se piše velikim slovima,

dok se više reči u jednom identifikatoru razdvaja potcrtom. Polja u interfejsu su automatski javna, pa to nije potrebno eksplicitno zadati.

Od pojave Jave SE5, na raspolaganju vam je mnogo moćnija i fleksibilnija rezervisana reč **enum**, pa više nema smisla upotrebljavati interfejse za konstante. Međutim, prilikom čitanja starog koda verovatno ćete često nailaziti na taj stari idiom (dodaci ove knjige na *www.MindView.net* sadrže potpun opis starog načina pravljenja nabrojanih tipova pomoću interfejsa). Više pojedinosti o upotrebi rezervisane reči **enum** naći ćete u poglavlju *Nabrojani tipovi.*

Vežba 17: (2) Dokažite da su polja interfejsa automatski statična i finalna.

Inicijalizacija polja u interfejsima

Polja definisana u interfejsima ne mogu biti prazna finalna polja, ali mogu da budu inicijalizovana nekonstantnim izrazom. Na primer:

```
//: interfejsi/SlucajnePromenljive.java
// Inicijalizacija polja u interfejsu pomoću
// nekonstantnih inicijalizacionih vrednosti
import java.util.*;

public interface SlucajnePromenljive {
  Random SLUCAJAN = new Random(47);
  int SLUCAJAN_INT = SLUCAJAN.nextInt(10);
  long SLUCAJAN_LONG = SLUCAJAN.nextLong() * 10;
  float SLUCAJAN_FLOAT = SLUCAJAN.nextLong() * 10;
  double SLUCAJAN_DOUBLE = SLUCAJAN.nextDouble() * 10;
} ///:~
```

Pošto su polja statična, biće inicijalizovana kada klasa bude prvi put učitana, što će se desiti kada prvi put budete pristupili bilo kom polju. Evo i jednostavnog testa:

```
//: interfejsi/TestirajSlucajnePromenljive.java
import static net.mindview.util.Print.*;

public class TestirajSlucajnePromenljive {
  public static void main(String[] args) {
    print(SlucajnePromenljive.SLUCAJAN_INT);
    print(SlucajnePromenljive.SLUCAJAN_LONG);
    print(SlucajnePromenljive.SLUCAJAN_FLOAT);
    print(SlucajnePromenljive.SLUCAJAN_DOUBLE);
  }
} /* Ispis:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*///:~
```

Ova polja nisu deo interfejsa, već se čuvaju u statičnom skladištu tog interfejsa.

Ugnežđivanje interfejsa

Interfejsi[3] mogu da budu ugnežđeni unutar drugih klasa ili drugih interfejsa. Time se otkriva nekoliko veoma korisnih osobina:

```
//: interfejsi/ugnezdjivanje/UgnezdjivanjeInterfejsa.java
package interfejsi.ugnezdjivanje;

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D uzmiD() { return new DImp2(); }
  private D dRef;
  public void primiD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
```

[3] Zahvaljujem Martinu Daneru što je postavio ovo pitanje tokom seminara.

```
    // Suvišan "public":
    public interface H {
      void f();
    }
    void g();
    // Ne može da bude privatan unutar interfejsa:
    //! private interface I {}
  }

  public class UgnezdjivanjeInterfejsa {
    public class BImp implements A.B {
      public void f() {}
    }
    class CImp implements A.C {
      public void f() {}
    }
    // Privatni interfejs može da se realizuje samo u klasi koja ga je definisala.
    //! class DImp implements A.D {
    //! public void f() {}
    //! }
    class EImp implements E {
      public void g() {}
    }
    class EGImp implements E.G {
      public void f() {}
    }
    class EImp2 implements E {
      public void g() {}
      class EG implements E.G {
        public void f() {}
      }
    }
    public static void main(String[] args) {
      A a = new A();
      // Nije moguć pristup A.D:
      //! A.D ad = a.uzmiD();
      // Vraća samo A.D:
      //! A.DImp2 di2 = a.uzmiD();
      // Ne može da se pristupi članu interfejsa:
      //! a.uzmiD().f();
      // Samo drugi A može nešto da uradi sa uzmiD():
      A a2 = new A();
      a2.primiD(a.uzmiD());
    }
  } ///:~
```

Sintaksa ugnežđivanja interfejsa unutar klase prilično je očigledna i, poput neugnežđenih interfejsa, oni takođe mogu imati javni ili paketni pristup.

Postoji nova začkoljica: interfejsi takođe mogu da budu privatni, kao što je slučaj s **A.D** (za ugnežđene interfejse koristi se ista sintaksa kao i za ugnežđene klase). Čemu služi pri-

vatni ugnežđeni interfejs? Mogli biste pretpostaviti kako on može samo da bude realizovan kao privatna ugnežđena klasa u klasi **DImp**, ali nam klasa **A.DImp2** pokazuje da on takođe može biti realizovan i kao javna klasa. Klasa **A.DImp2**, medutim, može da se upotrebljava samo kao klasa tog određenog tipa. Pošto ne postoji način da se spolja vidi kako ona, u stvari, realizuje privatni interfejs **D**, to je realizovanje privatnog interfejsa način da nametnete primenu definicija nekog interfejsa, a da pri tome ne dodate i informaciju o tipu (odnosno, time nećete dozvoliti svođenje naviše).

Metoda **uzmiD()** unosi dodatnu dilemu po pitanju privatnog interfejsa: ona je javna metoda koja vraća referencu na privatni interfejs. Šta možete da uradite s povratnom vrednošću ove metode? U metodi **main()** možete da uočite nekoliko pokušaja da se ta povratna vrednost upotrebi, ali se svi oni završavaju neuspehom. Povratnu vrednost jedino možete da predate drugom objektu koji ima ovlašćenja da je koristi – u ovom slučaju, drugom objektu klase **A** preko metode **primiD( )**.

Interfejs **E** pokazuje da interfejsi mogu da budu ugnežđeni jedan unutar drugog. Pravila o interfejsima – posebno ono da svi elementi interfejsa moraju da budu javni – ovde se strogo poštuju, pa je interfejs ugnežđen unutar drugog interfejsa automatski javan i ne može biti proglašen privatnim.

Klasa **UgnezdjivanjeInterfejsa** pokazuje različite načine na koje ugnežđeni interfejsi mogu da budu realizovani. Prilikom realizovanja interfejsa ne morate da realizujete i interfejse koji su u njega ugnežđeni. Takođe, privatni interfejsi ne mogu da budu realizovani van klase u kojoj su definisani.

U početku vam se može učiniti da ove mogućnosti postoje samo zbog konsistentnosti sintakse. U principu, kada znate da neka mogućnost postoji, smatram da možete naići na mesto na kome će korisno da vam posluži.

Interfejsi i Proizvođači

Interfejs bi trebalo da posluži kao put do više realizacija, a tipičan način pravljenja objekata usklađenih sa interfejsom jeste projektni obrazac *Factory Method* (Proizvodna metoda). Umesto da konstruktor pozivate neposredno, vi pozivate metodu za pravljenje proizvodnog objekta koji proizvodi realizaciju tog interfejsa – time bi, teorijski, vaš kod bio potpuno izolovan od realizacije interfejsa, što bi omogućilo neprimetnu zamenu jedne realizacije drugom. Evo jednog primera koji će pokazati strukturu obrasca *Factory Method*:

```
//: interfejsi/Proizvodjaci.java
import static net.mindview.util.Print.*;

interface Usluga {
  void metoda1();
  void metoda2();
}

interface ProizvodjacUsluga {
  Usluga uzmiUslugu ();
}
```

```
class Realizacija1 implements Usluga {
  Realizacija1() {} // Paketni pristup
  public void metoda1() {print("Realizacija1 metoda1");}
  public void metoda2() {print("Realizacija1 metoda2");}
}

class Realizacija1Proizvodjac implements ProizvodjacUsluga {
  public Usluga uzmiUslugu() {
    return new Realizacija1();
  }
}
class Realizacija2 implements Usluga {
  Realizacija2() {} // Paketni pristup
  public void metoda1() {print("Realizacija2 metoda1");}
  public void metoda2() {print("Realizacija2 metoda2");}
}

class Realizacija2Proizvodjac implements ProizvodjacUsluga {
  public Usluga uzmiUslugu() {
    return new Realizacija2();
  }
}

public class Proizvodjaci {
  public static void korisnikUsluge(ProizvodjacUsluga fabr) {
    Usluga s = fabr.uzmiUslugu();
    s.metoda1();
    s.metoda2();
  }
  public static void main(String[] args) {
    korisnikUsluge(new Realizacija1Proizvodjac ());
    // Realizacije su potpuno zamenljive:
    korisnikUsluge(new Realizacija2Proizvodjac ());
  }
} /* Ispis:
Realizacija1 metoda1
Realizacija1 metoda2
Realizacija2 metoda1
Realizacija2 metoda2
*///:~
```

Da nema obrasca *Factory Method*, negde u kodu morali biste da navedete tačan tip interfejsa **Usluga** koji se pravi, da bi on mogao da pozove odgovarajući konstruktor.

Zašto da čovek dodaje još jedan nivo posredovanja? Jedan od uobičajenih razloga je pravljenje zajedničkog kostura. Pretpostavimo da pravite sistem za igranje igara; na primer, za igranje šaha i dame na istoj tabli:

```
//: interfejsi/Igre.java
// Kostur za Igre napravljen pomoću Proizvodnih metoda.
import static net.mindview.util.Print.*;
```

```
interface Igra { boolean potez(); }
interface ProizvodjacIgara { Igra uzmiIgru(); }

class Dama implements Igra {
  private int potezi = 0;
  private static final int POTEZI = 3;
  public boolean potez() {
    print("Dama potez " + potez);
    return ++potezi != POTEZI;
  }
}

class ProizvodjacDama implements ProizvodjacIgara {
  public Igra uzmiIgru() { return new Dama(); }
}

class sah implements Igra {
  private int potezi = 0;
  private static final int POTEZI = 4;
  public boolean potez() {
    print("šah potez " + potezi);
    return ++potezi != POTEZI;
  }
}

class Proizvodjacsah implements ProizvodjacIgara {
  public Igra uzmiIgru() { return new sah(); }
}

public class Igre {
  public static void igrajIgru(ProizvodjacIgara proizvodjac) {
    Igra s = proizvodjac.uzmiIgru();
    while(s.potez())
      ;
  }
  public static void main(String[] args) {
    igrajIgru(new ProizvodjacDama());
    igrajIgru(new Proizvodjacsah());
  }
} /* Ispis:
Dama potez 0
Dama potez 1
Dama potez 2
Šah potez 0
Šah potez 1
Šah potez 2
Šah potez 3
*///:~
```

Ukoliko klasa **Igre** predstavlja složen deo koda, ovaj pristup omogućava ponovnu upotrebu tog koda s različitim vrstama igara. Postoje i komplikovanije igre za koje bi se mogao iskoristiti taj obrazac.

U sledećem poglavlju videćete elegantniji način za realizovanje Proizvođača pomoću anonimnih unutrašnjih klasa.

Vežba 18: (2) Napravite interfejs **Cikl** s realizacijama **Unicikl**, **Bicikl** i **Tricikl**. Napravite Proizvođače za svaki tip **Cikl**a i kôd koji upotrebljava te Proizvođače.

Vežba 19: (3) Pomoću proizvodnih metoda napravite zajednički kostur za bacanje novčića i bacanje kocke.

Sažetak

Lako je pasti u zamku i sve interfejse proglasiti dobrim pa ih zbog toga uvek treba pretpostaviti konkretnim klasama. Naravno, umesto svake klase gotovo uvek se može napraviti interfejs i Proizvođač.

Mnogi su podlegli tom iskušenju i napravili interfejse i Proizvođače gde god su mogli. Razlog: možda će zatrebati drugačija realizacija, pa tu apstrakciju treba dodati uvek. To se pretvorilo u svojevrsnu preranu optimizaciju projekta.

Svaku apstrakciju treba da motiviše određena stvarna potreba. Ako vam zatrebaju, interfejse treba da dodate tek prilikom ponovne podele na proste faktore. Nemojte taj dodatni nivo indirekcije stavljati odmah i svugde, jer time povećavate složenost. Dodatna složenost je bitna, i ako me neko natera da prođem kroz nju samo da bih na kraju shvatio kako je interfejs dodat „za svaki slučaj", a bez pravog razloga – posumnjaću u sve projekte te osobe.

Umesna smernica je *radije koristite klase nego interfejse*. Počnite od klasa, pa ako se ispostavi da su interfejsi neophodni, ponovo sve podelite na proste faktore. Interfejsi su odlične alatke, ali je lako preterati s njihovom upotrebom.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.net*.

Unutrašnje klase

Definiciju klase možete da stavite unutar definicije neke druge klase. Takva klasa se naziva unutrašnja klasa *(engl. inner class).*

UNUTRAŠNJA KLASA JE VREDNA, ZATO ŠTO OMOGUĆUJE DA KLASE LOGIČKI GRUPIŠETE I DA kontrolišete njihovu međusobnu vidljivost. Važno je shvatiti da su unutrašnje klase bitno različite od kompozicije.

Na prvi pogled, unutrašnje klase izgledaju kao jednostavan mehanizam za sakrivanje koda: preko njih ugrađujete klasu unutar druge klase. Naučićete da unutrašnja klasa pruža i više od toga – ona je svesna prisustva klase koja je okružuje i može s njom da komunicira. Iako su za većinu unutrašnje klase sasvim nov koncept, pomoću njih možete da pišete finiji i jasniji kôd, mada niko ne jamči da će takav i biti.

Dok učite o unutrašnjim klasama, njihova svrha često nije očigledna, pa navikavanje na njih može da potraje. Nakon opisa osnovne sintakse i semantike unutrašnjih klasa, u odeljku „Zašto unutrašnje klase?" naći ćete primere koji će objasniti prednosti unutrašnjih klasa.

Nakon tog odeljka, u ostatku poglavlja detaljnije je razmotrena sintaksa unutrašnjih klasa. Te mogućnosti su dodate radi potpunosti jezika, ali vam verovatno neće trebati, naročito ne isprva. Zato će vam početni delovi poglavlja zasad verovatno biti dovoljni, a detaljnije istraživanje možete smatrati referentnim materijalom.

Pravljenje unutrašnjih klasa

Unutrašnju klasu pravite baš kako biste očekivali – tako što definiciju klase umetnete u spoljnu klasu:

```
//: unutrasnjeklase/Posiljka1.java
// Pravljenje unutrašnjih klasa.

public class Posiljka1 {
  class Sadrzina {
    private int i = 11;
    public int value() { return i; }
  }
  class Odrediste {
    private String oznaka;
    Odrediste(String gde) {
      oznaka = gde;
    }
    String procitajOznaku() { return oznaka; }
  }
  // Upotreba unutrašnjih klasa izgleda kao i upotreba
  // bilo koje druge klase unutar klase Posiljka1:
  public void posalji(String odr) {
    Sadrzina c = new Sadrzina();
```

```
    Odrediste d = new Odrediste(odr);
    System.out.println(d.procitajOznaku());
  }
  public static void main(String[] args) {
    Posiljka1 p = new Posiljka1();
    p.posalji("Tanzanija");
  }
} /* Ispis:
Tanzanija
*///:~
```

Upotreba unutrašnje klase unutar metode **posalji**() izgleda kao i upotreba bilo koje druge klase. U ovom slučaju, jedina važna razlika je to što su imena ugnežđena unutar klase **Posiljka1**. Uskoro ćete videti da to nije jedina razlika.

Spoljna klasa će češće imati metodu koja vraća referencu na unutrašnju klasu, kao što se vidi u metodama **za**() i **sadr**():

```
//: unutrasnjeklase/Posiljka2.java
// Vraćanje reference na unutrašnju klasu

public class Posiljka2 {
  class Sadrzina {
    private int i = 11;
    public int vrednost() { return i; }
  }
  class Odrediste {
    private String oznaka;
    Odrediste(String gde) {
      oznaka = gde;
    }
    String procitajOznaku() { return oznaka; }
  }
  public Odrediste za(String s) {
    return new Odrediste(s);
  }
  public Sadrzina sadr() {
    return new Sadrzina();
  }
  public void posalji(String odr) {
    Sadrzina c = sadr();
    Odrediste d = za(odr);
    System.out.println(d.procitajOznaku());
  }
  public static void main(String[] args) {
    Posiljka2 p = new Posiljka2();
    p.posalji("Tanzanija");
    Posiljka2 q = new Posiljka2();
    // Definisanje referenci na unutrašnje klase:
    Posiljka2.Sadrzina c = q.sadr();
    Posiljka2.Odrediste d = q.za("Borneo");
  }
```

```
} /* Ispis:
Tanzanija
*///:~
```

Ako želite da objekat unutrašnje klase napravite izvan neke nestatične metode spoljne klase, kao tip tog objekta morate da navedete *ImeSpoljneKlase.ImeUnutrašnjeKlase*, kao što se vidi u metodi **main**().

Vežba 1: (1) Napišite klasu **Spoljna** koja sadrži unutrašnju klasu **Unutrasnja**. Klasi **Spoljna** dodajte metodu koja vraća objekat tipa **Unutrasnja**. U metodi **main**( ) napravite i inicijalizujte referencu na objekat klase **Unutrasnja**.

Veza ka spoljnoj klasi

Do sada izgleda da su unutrašnje klase samo tehnika za sakrivanje imena i za organizaciju koda, koja je korisna, ali ne i neophodna. Postoji, međutim, još jedna začkoljica. Kada napravite unutrašnju klasu, njen objekat *ima vezu sa objektom okolne klase koja ga je i napravila*, pa objekat unutrašnje klase može da pristupa članovima objekta koji ga okružuje – bez nekih posebnih kvalifikatora. Unutrašnje klase, pored toga, imaju prava da pristupaju svim elementima okružujuće klase.[1] To pokazuje naredni primer:

```
//: unutrasnjeklase/Sekvenca.java
// Čuva niz objekata

interface Selektor {
  boolean kraj();
  Object tekuci();
  void sledeci();
}

public class Sekvenca {
  private Object[] obs;
  private int sledeci = 0;
  public Sekvenca(int velicina) {obs = new Object[velicina]; }
  public void dodaj(Object x) {
    if(sledeci < obs.length) {
    obs[sledeci++] = x;
  }
  private class SelektorSekvence implements Selektor {
    private int i = 0;
    public boolean kraj() {return i == obs.length;}
    public Object tekuci() {return obs[i];}
    public void sledeci() {if(i < obs.length) i++; }
  }
  public Selektor selektor() {
    return new SelektorSekvence();
  }
```

[1] U ovom pogledu, *ugnežđene klase* u C++-u veoma se razlikuju, jer su one samo mehanizam za sakrivanje imena. U C++-u ne postoji veza sa okolnim objektom niti podrazumevana prava pristupa.

```
  public static void main(String[] args) {
    Sekvenca sekvenca = new Sekvenca(10);
    for(int i = 0; i < 10; i++)
      sekvenca.dodaj(Integer.toString(i));
    Selektor selektor = sekvenca.selektor();
    while(!selektor.kraj()) {
      System.out.println(selektor.tekuci() + " ");
      selektor.sledeci();
    }
  }
} /* Ispis:
0 1 2 3 4 5 6 7 8 9
*///:~
```

Klasa **Sekvenca** obuhvata samo niz fiksne dužine elemenata tipa **Object**. Možete da pozovete metodu **dodaj()** kako biste dodali nov element tipa **Object** na kraj niza (ako u njemu ima mesta). Za traženje elemenata niza postoji interfejs **Selektor**. To je primer projektnog obrasca *Iterator* (iterator), o kojem ćete više saznati u nastavku knjige. **Selektor** omogućava da proverite da li ste na kraju niza (metoda **kraj**()), da pogledate tekući objekat (metoda **tekuci**()) i da se pomerite na sledeći element u nizu (metoda **sledeci**()). Pošto je **Selektor** interfejs, njega mogu da realizuju i mnoge druge klase na neki svoj način, a taj interfejs može biti argument mnogih metoda, čime se pravi generički kôd.

U našem primeru, privatna klasa **SelektorSekvence** pruža funkcionalnost interfejsa **Selektor**. U metodi **main**() se, nakon pravljenja objekta klase **Sekvenca**, u taj objekat dodaje nekoliko objekata tipa **String**. Zatim se pozivanjem metode **selektor** pravi **Selektor** koji se koristi za pomeranje po objektu klase **Sekvenca** i za pristup svakom elementu.

Na prvi pogled, pravljenje klase **SelektorSekvence** izgleda poput još jedne unutrašnje klase. Proučimo je pažljivije. Obratite pažnju na to da se sve metode – **kraj**(), **tekuci**() i **sledeci**() – obraćaju referenci **obs** koja nije deo klase **SelektorSekvence**, već je privatno polje okolne spoljne klase. Unutrašnja klasa, međutim, može da pristupa metodama i poljima spoljne klase kao da pripadaju njoj. Ta osobina može da bude veoma pogodna, kao što se vidi u prethodnom primeru.

Znači, unutrašnja klasa ima automatski pristup članovima klase koja je okružuje. Kako se to dešava? Unutrašnja klasa skriveno čuva referencu na onaj objekat spoljne klase koji je tu unutrašnju klasu napravio. Prilikom pristupanja članovima spoljne klase upotrebljava se ta (skrivena) referenca. Prevodilac se, na sreću, brine o svim detaljima umesto vas. Sada konačno možete da shvatite zašto unutrašnja klasa može da se napravi samo zajedno sa objektom spoljne klase (kada je, kao što ćete videti, unutrašnja klasa nestatična). Za konstrukciju objekta unutrašnje klase potrebna je referenca na objekat spoljne klase i prevodilac će se pobuniti ako toj referenci ne može da pristupi. U većini slučajeva, ovaj scenario se odigrava bez intervencije programera.

Vežba 2: (1) Napravite klasu koja čuva jedan **String** i ima metodu **toString**() koja taj **String** prikazuje. Dodajte nekoliko primeraka te klase objektu tipa **Sekvenca** i prikažite ih.

Vežba 3: (1) Izmenite vežbu 1 tako da **Spoljna** ima privatno polje tipa **String** (koje inicijalizuje konstruktor), a **Unutrasnja** metodu **toString**() koja to polje prikazuje. Napravite objekat tipa **Unutrasnja** i prikažite ga.

Upotreba sintaksi .this i .new

Ako vam u unutrašnjoj klasi treba referenca na objekat spoljne klase, navedite ime spoljne klase i iza nje tačku i rezervisanu reč **this**. Dobijena referenca će automatski biti ispravnog tipa, što se zna i proverava prilikom prevođenja, a ne prilikom izvršavanja, pa nema režijskih troškova za to. Evo primera koji pokazuje upotrebu **.this**:

```
//: unutrasnjeklase/TackaThis.java
// Kvalifikovanje pristupa objektu spoljne klase.
public class TackaThis {
  void f() { System.out.println("TackaThis.f()"); }
  public class Unutrasnja {
    public TackaThis spoljna() {
      return TackaThis.this;
      // Golo "this" bi bilo "this" od klase Unutrasnja
    }
  }
  public Unutrasnja unutrasnja() { return new Unutrasnja(); }
  public static void main(String[] args) {
    TackaThis dt = new TackaThis();
    TackaThis.Unutrasnja dti = dt.unutrasnja();
    dti.spoljna().f();
  }
} /* Ispis:
TackaThis.f()
*///:~
```

Ponekad nekom drugom objektu treba narediti da napravi objekat neke od svojih unutrašnjih klasa. Da biste to postigli, sintaksom **.new** prosledite referencu na taj drugi objekat spoljne klase, kao u narednom primeru:

```
//: unutrasnjeklase/TackaNova.java
// Neposredno pravljenje unutrašnje klase operatorom .new.

public class TackaNova {
  public class Unutrasnja {}
  public static void main(String[] args) {
    TackaNova dn = new TackaNova();
    TackaNova.Unutrasnja dni = dn.new Unutrasnja();
  }
} ///:~
```

Nasuprot očekivanju, da biste direktno napravili objekat unutrašnje klase ne treba da pratite uobičajenu metodu i da se obratite imenu spoljne klase **TackaNova** – umesto toga morate da upotrebite *objekat* spoljne klase, kao što vidite u prethodnom primeru. Time se rešavaju i pitanja oblasti važenja za unutrašnju klasu, pa se ne kaže **dn.new TackaNova.Unutrasnja( )**. (To se čak *ne može* reći.)

Znači, ako nemate objekat spoljne klase, ne možete da napravite ni objekat unutrašnje klase, zbog toga što objekat unutrašnje klase mora biti povezan sa spoljnom klasom. Međutim, ukoliko pravite *ugnežđenu klasu* (statičnu unutrašnju klasu), njoj referenca na objekat spoljne klase nije potrebna.

Evo kako bi se operator **.new** primenio u primeru Pošiljka:

```
//: unutrasnjeklase/Posiljka3.java
// Pravljenje instanci unutrašnje klase operatorom .new

public class Posiljka3 {
  class Sadrzina {
    private int i = 11;
    public int vrednost() { return i; }
  }
  class Odrediste {
    private String oznaka;
    Odrediste(String gde) {  oznaka = gde;  }
    String procitajOznaku() { return oznaka; }
  }
  public static void main(String[] args) {
    Posiljka3 p = new Posiljka3();
    // Morate da upotrebite instancu spoljne klase
    // da biste napravili instancu unutrašnje klase:
    Posiljka3.Sadrzina c = p.new Sadrzina();
    Posiljka3.Odrediste d = p.new Odrediste("Tanzanija");
  }
} ///:~
```

Vežba 4: (2) Klasi **Sekvenca.SelektorSekvence** dodajte metodu koja proizvodi referencu na spoljnu klasu **Sekvenca**.

Vežba 5: (1) Napravite klasu koja ima unutrašnju klasu. U odvojenoj klasi napravite instancu unutrašnje klase.

Unutrašnje klase i svođenje naviše

Unutrašnje klase dolaze do izražaja kada počnete da primenjujete svođenje naviše ka osnovnoj klasi, a posebno ka interfejsu. (Efekat dobijanja reference na interfejs od objekta koji ga realizuje, u suštini je isti kao i svođenje naviše ka osnovnoj klasi.) Ovo biva zato što unutrašnja klasa koja realizuje interfejs može biti potpuno nevidljiva i nedostupna, što je pogodno za sakrivanje realizacije. Vraća vam se samo referenca na osnovnu klasu ili interfejs.

Prvo ćemo definisati zajedničke interfejse za prethodne primere:

```
//: unutrasnjeklase/Odrediste.java
public interface Odrediste {
  String procitajOznaku();
} ///:~
```

```
//: unutrasnjeklase/Sadrzina.java
public interface Sadrzina {
  int vrednost();
} ///:~
```

Sada **Sadrzina** i **Odrediste** predstavljaju interfejse dostupne programeru klijentu koji koristi našu biblioteku klasa. (Setite se da su svi članovi interfejsa automatski javni.)

Kada dobijete referencu na osnovnu klasu ili na interfejs, moguće je da uopšte ne možete da otkrijete stvarni tip objekta, kao što je prikazano u sledećem primeru:

```
//: unutrasnjeklase/ProbnaPosiljka.java

public class Posiljka4 {
  private class PSadrzina implements Sadrzina {
    private int i = 11;
    public int vrednost() { return i; }
  }
  protected class POdrediste implements Odrediste {
    private String oznaka;
    private POdrediste(String gde) {
      oznaka = gde;
    }
    public String procitajOznaku() { return oznaka; }
  }
  public Odrediste odrediste(String s) {
    return new POdrediste(s);
  }
  public Sadrzina sadr() {
    return new PSadrzina();
  }
}
public class ProbnaPosiljka {
  public static void main(String[] args) {
    Posiljka4 p = new Posiljka4();
    Sadrzina c = p.sadr();
    Odrediste d = p.odrediste("Tanzanija");
    // Nije dozvoljeno -- ne možete da pristupite privatnoj klasi:
    //! Posiljka4.PSadrzina pc = p.new PSadrzina();
  }
} ///:~
```

U klasi **Posiljka4** dodato je nešto novo: unutrašnja klasa **PSadrzina** je privatna tako da niko osim klase **Posiljka4** ne može da joj pristupi. Obične klase (koje nisu unutrašnje) ne mogu da budu privatne niti zaštićene – mogu imati samo javni ili paketni pristup. Klasa **POdrediste** je zaštićena, pa ne može da joj pristupa niko osim klase **Posiljka4**, klasa koje su u istom paketu kao i **Posiljka4** (pošto rezervisana reč **protected** takođe dozvoljava paketni pristup) i naslednika klase **Posiljka4**. To znači da programer klijent ne može potpuno da

poznaje te članove, niti može da im pristupa bez ograničenja. Ka privatnoj unutrašnjoj klasi (ili ka zaštićenoj unutrašnjoj klasi ako niste neki od naslednika), u stvari, ne možete da obavite ni svođenje naniže, jer ne možete da pristupite njenom imenu, kao što se vidi u klasi **ProbnaPosiljka**. Privatne unutrašnje klase stoga omogućavaju projektantu klase da potpuno spreči zavisnost od tipa i da potpuno sakrije podatke o realizaciji. Pored toga, proširivanje interfejsa je beskorisno, iz perspektive programera klijenta, jer on ne može da pristupi ostalim metodama koje nisu deo javnog interfejsa. Time se takođe Javinom prevodiocu omogućava da pravi efikasniji kod.

Vežba 6: (2) Napravite interfejs koji sadrži barem jednu metodu u sopstvenom paketu. Dodajte zaštićenu unutrašnju klasu koja realizuje taj interfejs. U trećem paketu nasledite svoju klasu i, unutar metode, vratite objekat zaštićene unutrašnje klase, svodeći naviše na interfejs tokom vraćanja.

Vežba 7: (2) Napravite klasu koja ima privatno polje i privatnu metodu. Napravite unutrašnju klasu čija metoda menja polje spoljne klase i poziva metodu spoljne klase. U drugoj metodi spoljne klase napravite objekat unutrašnje klase i pozovite njenu metodu, a zatim pokažite dejstvo na objekat spoljne klase.

Vežba 8: (2) Utvrdite da li spoljna klasa ima pristup privatnim elementima svoje unutrašnje klase.

Unutrašnje klase u metodama i oblastima važenja

Na osnovu onoga što ste dosad videli, stekli ste predstavu o uobičajenoj upotrebi unutrašnjih klasa. Programi koje ćete pisati i čitati, a u kojima se pojavljuju unutrašnje klase, po pravilu biće sa „običnim" unutrašnjim klasama koje su jednostavne i lako se razumeju. Koncept unutrašnjih klasa obuhvata još dosta toga i postoji više drugih, skrivenih načina kako da ih, ako želite, koristite: unutrašnje klase mogu da budu napravljene unutar metoda ili čak u proizvoljnoj oblasti važenja. Tako nešto možete da uradite iz dva razloga, ukoliko:

1. Kao što je ranije pokazano, realizujete interfejs tako da možete da napravite i vratite referencu.
2. Rešavate složen problem i želite da napravite klasu koja će vam pomoći u njegovom rešavanju, ali ne želite da ona bude javno dostupna.

U narednim primerima, prethodni program će biti izmenjen tako da koristi:

1. Klasu definisanu unutar metode.
2. Klasu definisanu unutar neke oblasti važenja u metodi.
3. *Anonimnu* klasu koja realizuje interfejs.
4. Anonimnu klasu koja proširuje klasu koja nema podrazumevani konstruktor.
5. Anonimnu klasu koja inicijalizuje polja.
6. Anonimnu klasu koja obavlja konstrukciju pomoću inicijalizacije instance (anonimne unutrašnje klase ne mogu da imaju konstruktore).

Prvi primer prikazuje pravljenje cele klase čija je oblast važenja jedna metoda (umesto da je oblast važenja druga klasa). To je *lokalna unutrašnja klasa*:

```
//: unutrasnjeklase/Posiljka5.java
// Ugnežđivanje klase unutar metode.

public class Posiljka5 {
  public Odrediste odr(String s) {
    class POdrediste implements Odrediste {
      private String oznaka;
      private POdrediste(String gde) {
        oznaka = gde;
      }
      public String procitajOznaku() { return oznaka; }
    }
    return new POdrediste(s);
  }
  public static void main(String[] args) {
    Posiljka5 p = new Posiljka5();
    Odrediste d = p.odr("Tanzanija");
  }
} ///:~
```

Klasa **POdrediste** je deo metode **odr**(), a ne deo klase **Posiljka5**. Zbog toga klasi **POdrediste** ne možete da pristupite izvan metode **odr**(). Obratite pažnju na svođenje naviše u naredbi **return** – metoda **odr**() vraća samo referencu na osnovnu klasu, klasu **Odrediste**. Naravno, to što je ime klase **POdrediste** navedeno unutar metode **odr**() ne znači da objekti klase **POdrediste** nakon povratka iz te metode neće biti ispravni.

Obratite pažnju na to da ste za imena unutrašnjih klasa u svim klasama u istom poddirektorijumu mogli da koristite identifikator **POdrediste**, a da ne dođe do sukoba imena.

Naredni primer pokazuje kako unutrašnju klasu možete da ugnezdite unutar proizvoljne oblasti važenja:

```
//: unutrasnjeklase/Posiljka6.java
// Ugnežđivanje klase unutar oblasti važenja

public class Posiljka6 {
  private void unutrasnjePracenje(boolean b) {
    if(b) {
      class KarticaZaPracenje {
        private String id;
        KarticaZaPracenje(String s) {
          id = s;
        }
        String uzmiKarticu() { return id; }
      }
      KarticaZaPracenje ts = new KarticaZaPracenje("slip");
      String s = ts.uzmiKarticu();
    }
```

```
    // Ovde ne možete da je koristite! Izvan je oblasti važenja:
    //! KarticaZaPracenje ts = new KarticaZaPracenje("x");
  }
  public void prati() { unutrasnjePracenje(true); }
  public static void main(String[] args) {
    Posiljka6 p = new Posiljka6();
    p.prati();
  }
} ///:~
```

Klasa **KarticaZaPracenje** je ugnežđena unutar oblasti važenja naredbe **if**. To ne znači da se *klasa* uslovno pravi – ona će biti prevedena zajedno sa svim ostalim. Ona, međutim, nije dostupna izvan oblasti važenja unutar koje je definisana. Osim toga, ista je kao svaka druga obična klasa.

Vežba 9: (1) Napravite interfejs s najmanje jednom metodom i realizujte ga definišući unutrašnju klasu unutar neke metode koja vraća referencu na vaš interfejs.

Vežba 10: (1) Ponovite prethodnu vežbu, ali definišite unutrašnju klasu u nekoj oblasti važenja u okviru metode.

Vežba 11: (2) Napravite privatnu unutrašnju klasu koja realizuje javni interfejs. Napišite metodu koja vraća referencu na instancu privatne unutrašnje klase, svedenu naviše ka interfejsu. Pokažite da je unutrašnja klasa potpuno sakrivena tako što ćete pokušati da svedete naniže ka njoj.

Anonimne unutrašnje klase

Naredni primer izgleda pomalo čudno:

```
//: unutrasnjeklase/Posiljka7.java
// Metoda koja vraća anonimnu unutrašnju klasu.

public class Posiljka7 {
  public Sadrzina sadr() {
    return new Sadrzina() {  // Umetanje definicije klase
      private int i = 11;
      public int vrednost() { return i; }
    }; // U ovom slučaju je potrebna tačka i zarez.
  }
  public static void main(String[] args) {
    Posiljka7 p = new Posiljka7();
    Sadrzina c = p.sadr();
  }
} ///:~
```

Metoda **sadr**() kombinuje generisanje povratne vrednosti i definiciju klase koja predstavlja tu povratnu vrednost! Pored toga, klasa je *anonimna* – nema ime. Da stvari budu još gore, izgleda kao da počinjete da pravite nov objekat klase **Sadrzina**, ali tada, pre nego što dođete do tačke i zareza, kažete: „Samo trenutak, sada ću da ubacim definiciju klase".

Ova čudna sintaksa znači: „Napravi objekat anonimne klase koja nasleđuje klasu **Sadrzina**". Referenca koju vraća operator **new** automatski se svodi naviše na referencu klase **Sadrzina**. Sintaksa za anonimnu unutrašnju klasu je skraćeni oblik za:

```
//: unutrasnjeklase/Posiljka7b.java
// Proširena verzija programa Posiljka7.java

public class Posiljka7b {
  class MojaSadrzina implements Sadrzina {
    private int i = 11;
    public int vrednost() { return i; }
  }
  public Sadrzina sadrzina() { return new MojaSadrzina(); }
  public static void main(String[] args) {
    Posiljka7b p = new Posiljka7b();
    Sadrzina c = p.sadrzina();
  }
} ///:~
```

U anonimnoj unutrašnjoj klasi, klasa **Sadrzina** je napravljena pomoću podrazumevanog konstruktora.

Naredni program pokazuje šta da radite ako vaša osnovna klasa zahteva konstrukciju sa argumentom:

```
//: unutrasnjeklase/Posiljka8.java
// Anonimna unutrašnja klasa koja poziva konstruktor osnovne klase.

public class Posiljka8 {
  public Omotac omot(int x) {
    // Poziv konstruktoru osnovne klase:
    return new Omotac(x) { // Prosleđivanje argumenta konstruktoru.
      public int vrednost() {
        return super.vrednost() * 47;
      }
    }; // Potrebna je tačka i zarez
  }
  public static void main(String[] args) {
    Posiljka8 p = new Posiljka8();
    Omotac w = p.omot(10);
  }
} ///:~
```

Odgovarajući argument samo prosledite konstruktoru osnovne klase, kao što je u gornjem primeru parametar **x** prosleđen naredbom **new Omotac(x)**. Iako je **Omotac** obična klasa s realizacijom, ona se upotrebljava i kao zajednički interfejs za svoje izvedene klase:

```
//: unutrasnjeklase/Omotac.java
public class Omotac {
  private int i;
```

```
  public Omotac(int x) { i = x; }
  public int vrednost() { return i; }
} ///:~
```

Primetili ste kako **Omotac** ima konstruktor koji zahteva argument, da bi bilo malo zanimljivije.

Znak tačka i zarez na kraju anonimne unutrašnje klase ne označava kraj tela klase. On označava kraj izraza koji u datom slučaju sadrži i anonimnu klasu. Dakle, upotreba tačke i zareza potpuno je ista kao i bilo gde drugde u programu.

Inicijalizaciju objekta anonimne klase možete da obavite na mestu na kome definišete polja:

```
//: unutrasnjeklase/Posiljka9.java
// Anonimna unutrašnja klasa koja obavlja inicijalizaciju.
// Kraća verzija datoteke Posiljka5.java.

public class Posiljka9 {
  // Argument koji koristite u anonimnoj unutrašnjoj
  // klasi mora da bude finalan:
  public Odrediste odr(final String odr) {
    return new Odrediste() {
      private String oznaka = odr;
      public String procitajOznaku() { return oznaka; }
    };
  }
  public static void main(String[] args) {
    Posiljka9 p = new Posiljka9();
    Odrediste d = p.odr("Tanzanija");
  }
} ///:~
```

Ako definišete anonimnu unutrašnju klasu u kojoj želite da koristite objekat definisan izvan nje, prevodilac zahteva da taj objekat bude finalan. Zbog toga je argument metode **odr**() finalan. Ukoliko to zaboravite, dobićete grešku prilikom prevođenja.

Dokle god samo dodeljujete vrednosti poljima, gornji pristup je u redu. Ali šta ako treba da obavite neku aktivnost sličnu konstruktoru? Pošto je klasa anonimna, konstruktor u njoj ne može da ima ime (jer ono ne postoji!). Na sledeći način možete efektivno da napravite konstruktor za anonimnu unutrašnju klasu pomoću bloka za *inicijalizaciju instanci*:

```
//: unutrasnjeklase/KonstruktorAnonimne.java
// Pravljenje konstruktora za anonimnu unutrašnju klasu.
import static net.mindview.util.Print.*;
abstract class Osnovna {
  public Osnovna(int i) {
    print("Konstruktor klase Osnovna, i = " + i);
  }
  public abstract void f();
}
```

```
public class KonstruktorAnonimne {
  public static Osnovna uzmiOsnovnu(int i) {
    return new Osnovna(i) {
      { print("Unutar inicijalizatora instance"); }
      public void f() {
        print("U anonimnoj f()");
      }
    };
  }
  public static void main(String[] args) {
    Osnovna osnovna = uzmiOsnovnu(47);
    osnovna.f();
  }
} /* Ispis:
Konstruktor klase Osnovna, i = 47
Unutar inicijalizatora instance
U anonimnoj f()
*///:~
```

U ovom slučaju, promenljiva **i** *nije* morala da bude finalna. Iako se **i** prosleđuje osnovnom konstruktoru anonimne klase, nikada se ne upotrebljava neposredno *unutar* te anonimne klase.

Evo kako izgleda naša Pošiljka sa inicijalizacijom instance. Imajte u vidu da argumenti metode **odr**() moraju biti finalni, pošto se upotrebljavaju unutar anonimne klase.

```
//: unutrasnjeklase/Posiljka10.java
// Upotreba "inicijalizacije instanci" za konstrukciju
// anonimne unutrašnje klase.

public class Posiljka10 {
  public Odrediste
  odr(final String odr, final float cena) {
    return new Odrediste() {
      private int troskovi;
      // Inicijalizacija instanci za svaki objekat:
      {
        troskovi = Math.round(cena);
        if(troskovi > 100)
          System.out.println("Prekoračen budžet!");
      }
      private String oznaka = odr;
      public String procitajOznaku() { return oznaka; }
    };
  }
  public static void main(String[] args) {
    Posiljka10 p = new Posiljka10();
    Odrediste d = p.odr("Tanzanija", 101.395F);
  }
} /* Ispis:
Prekoračen budžet!
*///:~
```

Unutar bloka za inicijalizaciju instanci uočavate kôd koji ne može da bude upotrebljen za inicijalizaciju polja (preciznije, naredbu **if**). Stoga blok za inicijalizaciju instanci efektivno predstavlja konstruktor anonimne unutrašnje klase. Ovakav pristup, naravno, ima ograničenja; blok za inicijalizaciju instanci ne možete da preklopite pa ćete imati samo jedan takav konstruktor.

Anonimne unutrašnje klase su ponešto ograničenije nego obično nasleđivanje, pošto one mogu da prošire klasu ili da realizuju interfejs, ali ne oboje. A ako realizujete interfejs, možete da realizujete samo jedan.

Vežba 12: (1) Ponovite vežbu 7 koristeći anonimnu unutrašnju klasu.

Vežba 13: (1) Ponovite vežbu 9 koristeći anonimnu unutrašnju klasu.

Vežba 14: (1) Prepravite program **interfejsi/HororPredstava.java** tako da interfejse **OpasnoCudoviste** i **Vampir** realizujete pomoću anonimnih klasa.

Vežba 15: (2) Napravite klasu koja ima nepodrazumevani konstruktor, ali ne i podrazumevani. Napravite drugu klasu koja ima metodu koja vraća referencu na prvu klasu. Napravite objekat koji ćete vratiti koristeći anonimnu unutrašnju klasu koja nasleđuje prvu klasu.

Ponovo o proizvodnim metodama

Pogledajte koliko lepše izgleda primer **interfejsi/Proizvodjaci.java** kada se upotrebe anonimne unutrašnje klase:

```
//: unutrasnjeklase/Proizvodjaci.java
import static net.mindview.util.Print.*;

interface Usluga {
  void metoda1();
  void metoda2();
}

interface ProizvodjacUsluga {
  Usluga uzmiUslugu ();
}

class Realizacija1 implements Usluga {
  private Realizacija1() {}
  public void metoda1() {print("Realizacija1 metoda1");}
  public void metoda2() {print("Realizacija1 metoda2");}
  public static ProizvodjacUsluga proizvodjac =
    new ProizvodjacUsluga()  {
      public Usluga uzmiUslugu ()  {
        return new Realizacija1();
      }
    };
}
```

```
class Realizacija2 implements Usluga {
  private Realizacija2() {}
  public void metoda1() {print("Realizacija2 metoda1");}
  public void metoda2() {print("Realizacija2 metoda2");}
  public static ProizvodjacUsluga proizvodjac =
    new ProizvodjacUsluga() {
      public Usluga uzmiUslugu () {
        return new Realizacija2();
      }
    };
}

public class Proizvodjaci {
  public static void korisnikUsluge(ProizvodjacUsluga fabr) {
    Usluga s = fabr.uzmiUslugu();
    s.metoda1();
    s.metoda2();
  }
  public static void main(String[] args) {
    korisnikUsluge(new Realizacija1.proizvodjac ());
    // Realizacije su međusobno potpuno zamenljive:
    korisnikUsluge(new Realizacija2.proizvodjac ());
  }
} /* Ispis:
Realizacija1 metoda1
Realizacija1 metoda2
Realizacija2 metoda1
Realizacija2 metoda2
*///:~
```

Sada konstruktori za klase **Realizacija1** i **Realizacija2** mogu biti privatni, i nema potrebe da se imenovana klasa pravi kao Proizvođač. Pored toga, često je potreban samo jedan proizvodni objekat, pa je ovde napravljen kao statično polje u realizaciji interfejsa **Usluga**. Dobija se i smislenija sintaksa.

I primer **interfejsi/Igre.java** može biti poboljšan pomoću anonimnih unutrašnjih klasa:

```
//: unutrasnjeklase/Igre.java
// Kostur za Igre napravljen pomoću unutrašnjih klasa.
import static net.mindview.util.Print.*;

interface Igra { boolean potez(); }
interface ProizvodjacIgara { Igra uzmiIgru(); }

class Dama implements Igra {
  private Dama() {}
  private int potezi = 0;
  private static final int POTEZI = 3;
  public boolean potez() {
```

```
    print("Dama potez " + potez);
    return ++potezi != POTEZI;
  }
  public static ProizvodjacIgara proizvodjac = new ProizvodjacIgara() {
    public Igra uzmiIgru() { return new Dama(); }
  };
}
class Sah implements Igra {
  private Sah() {}
  private int potezi = 0;
  private static final int POTEZI = 4;
  public boolean potez() {
    print("Šah potez " + potezi);
    return ++potezi != POTEZI;
  }
  public static ProizvodjacIgara proizvodjac = new ProizvodjacIgara() {
    public Igra uzmiIgru() { return new Sah(); }
  };
}

public class Igre {
  public static void igrajIgru(ProizvodjacIgara proizvodjac) {
    Igra s = proizvodjac.uzmiIgru();
    while(s.potez())
      ;
  }
  public static void main(String[] args) {
    igrajIgru(Dama.proizvodjac);
    igrajIgru(Sah.proizvodjac);
  }
} /* Ispis:
Dama potez 0
Dama potez 1
Dama potez 2
Šah potez 0
Šah potez 1
Šah potez 2
Šah potez 3
*///:~
```

Ne zaboravite savet s kraja prethodnog poglavlja: *pravite klase, a ne interfejse.* Ukoliko projekat zahteva interfejs, to će vam biti očigledno. Ne koristite interfejse ako baš ne morate.

Vežba 16: (1) Prepravite rešenje vežbe 18 iz poglavlja 9 koristeći anonimnu unutrašnju klasu.

Vežba 17: (1) Prepravite rešenje vežbe 19 iz poglavlja 9 koristeći anonimnu unutrašnju klasu.

Ugnežđene klase

Ako vam ne treba veza između njenih objekata i objekata spoljne klase, unutrašnju klasu možete da proglasite statičnom. Ona se obično naziva *ugnežđena klasa*[2]. Da biste shvatili značenje kvalifikatora **static** kada se primeni na unutrašnje klase, setite se da objekat obične unutrašnje klase implicitno čuva referencu na objekat spoljne klase koja ga je napravila. Kada kažete da je unutrašnja klasa statična, ovo ne važi. Ugnežđena klasa znači da:

1. Za pravljenje objekta ugnežđene klase nije potreban objekat spoljne klase.
2. Iz objekta ugnežđene klase ne možete da pristupite objektu spoljne klase.

Ugnežđene klase se još po nečemu razlikuju od nestatičnih, tj. običnih unutrašnjih klasa. Polja i metode nestatične unutrašnje klase mogu da budu samo na spoljnom nivou klase, odnosno nestatična unutrašnja klasa ne može da ima statične podatke, statična polja niti statične unutrašnje klase. Međutim, ugnežđene (statične unutrašnje) klase mogu sve to da sadrže:

```
//: unutrasnjeklase/Posiljka11.java
// Ugnežđene klase (statične unutrašnje klase)

public class Posiljka11 {
  private static class PSadrzina implements Sadrzina {
    private int i = 11;
    public int vrednost() { return i; }
  }
  protected static class POdrediste
  implements Odrediste {
    private String oznaka;
    private POdrediste(String gde) {
      oznaka = gde;
    }
    public String procitajOznaku() { return oznaka; }
    // Statična unutrašnja klasa može da sadrži druge statične elemente:
    public static void f() {}
    static int x = 10;
    static class DrugiNivo {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Odrediste odr(String s) {
    return new POdrediste(s);
  }
  public static Sadrzina sadr() {
    return new PSadrzina();
  }
```

[2] Otprilike slično ugnežđenim klasama u C++-u, sem što te klase ne mogu pristupati privatnim članovima kao u Javi.

```
  public static void main(String[] args) {
    Sadrzina c = sadr();
    Odrediste d = odr("Tanzanija");
  }
} ///:~
```

U metodi **main**() nije potreban objekat klase **Posiljka11**; umesto toga upotrebljavate normalnu sintaksu za pristup statičnom članu da biste pozvali metode koje vraćaju reference na objekte klase **Sadrzina** i **Odrediste**.

Kao što ste videli u prethodnom delu poglavlja, u običnoj (nestatičnoj) unutrašnjoj klasi veza ka spoljnoj klasi ostvaruje se preko posebne reference **this**. Statična unutrašnja klasa nema tu posebnu referencu, zbog čega je analogna statičnim metodama.

Vežba 18: (1) Napravite klasu koja sadrži ugnežđenu klasu. U metodi **main**( ) napravite instancu te ugnežđene klase.

Vežba 19: (2) Napravite klasu koja sadrži unutrašnju klasu koja i sama sadrži unutrašnju klasu. Nastavite dalje ovako, koristeći ugnežđene klase. Pogledajte imena datoteka **.class** koje prevodilac pravi.

Klase unutar interfejsa

Unutar interfejsa ne možete da pišete bilo kakav kôd, ali statična unutrašnja klasa *može* da bude deo nekog interfejsa. Svaka klasa koju stavite unutar interfejsa automatski postaje javna i statična. Pošto je klasa statična, ona ne narušava pravila o interfejsima – statična unutrašnja klasa se samo umeće u imenski prostor interfejsa. U unutrašnjoj klasi čak možete da realizujete okružujući interfejs, i to na ovaj način:

```
//: unutrasnjeklase/KlasaUInterfejsu.java
// {main: KlasaUInterfejsu$Test}

public interface KlasaUInterfejsu {
  void zdravo();
  class Test implements KlasaUInterfejsu  {
    public void zdravo()  {
      System.out.println("Zdravo!");
    }
    public static void main(String[] args) {
      new Test().zdravo();
    }
  }
} /* Ispis:
Zdravo!
*///:~
```

Klasu je umesno staviti u interfejs onda kada hoćete da napravite zajednički kôd koji će biti korišćen sa svim različitim realizacijama tog interfejsa.

Na početku ove knjige savetovao sam vam da u svaku klasu ubacite metodu **main**() za njeno testiranje. Mana takvog pristupa je dodatna količina prevedenog koda koji morate da vučete naokolo. Ako to predstavlja problem, za čuvanje koda za testiranje koristite statičnu unutrašnju klasu:

```
//: unutrasnjeklase/ ZaTestiranje.java
// Stavljanje koda za testiranje u statičnu unutrašnju klasu
// {main: ZaTestiranje$Tester}

public class ZaTestiranje {
  public void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      ZaTestiranje t = new ZaTestiranje();
      t.f();
    }
  }
} /* Ispis:
f()
*///:~
```

Ovim pravite zasebnu klasu pod imenom **ZaTestiranje$Tester** (da biste pokrenuli program, napišite **java ZaTestiranje$Tester**, ali u Unixu/Linuxu pretvorite **$** u izlaznu sekvencu). Ovu klasu možete da koristite za testiranje, ali ne morate da je uključite u gotov proizvod; pre nego što sve upakujete, samo uklonite **ZaTestiranje$Tester.class**.

Vežba 20: (1) Napravite interfejs koji sadrži ugnežđenu klasu. Realizujte interfejs i napravite instancu te ugnežđene klase.

Vežba 21: (2) Napravite interfejs koji sadrži unutrašnju klasu sa statičnom metodom što poziva metode tog interfejsa i prikazuje rezultate. Realizujte interfejs i metodi prosledite instancu te realizacije.

Pristupanje spoljašnjosti iz višestruko ugnežđenih klasa

Bez obzira na to koliko duboko je unutrašnja klasa ugnežđena, ona može transparentno da pristupi svim članovima svih klasa u kojima je ugnežđena, kao što se vidi iz narednog primera:[3]

```
//: unutrasnjeklase/PristupPriVisestrukomUgnezdjivanju.java
// Ugnežđene klase mogu da pristupe svim članovima
// na svim nivoima klasa u koje su ugnežđene.

class PVU {
  private void f() {}
  class A {
    private void g() {}
```

[3] Još jednom zahvaljujem Martinu Daneru.

```
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}
public class PristupPriVisestrukomUgnezdjivanju {
  public static void main(String[] args) {
    PVU pvu = new PVU();
    PVU.A pvua = pvu.new A();
    PVU.A.B pvuab = pvua.new B();
    pvuab.h();
  }
} ///:~
```

Iz primera vidite da u klasi **PVU.A.B** metode **g()** i **f()** mogu da budu pozvane bez ikakvih kvalifikatora (i pored činjenice da su privatne). Ovaj primer takođe pokazuje sintaksu koja je potrebna za građenje objekata višestruko ugnežđenih unutrašnjih klasa pri pravljenju objekata različitih klasa. Sintaksa **.new** daje odgovarajuću oblast važenja, pa pri pozivu konstruktora ne morate da naznačite ime klase.

Zašto unutrašnje klase?

Dosad je bilo puno sintakse i semantike koje opisuju način na koji unutrašnje klase rade, ali to i dalje nije odgovor na pitanje: zbog čega one postoje? Zbog čega su se projektanti Jave toliko pomučili da jeziku dodaju ovu suštinsku mogućnost?

Unutrašnja klasa obično nasleđuje klasu ili realizuje interfejs i radi sa objektom spoljne klase unutar koje je napravljena. Moglo bi se reći da je unutrašnja klasa nalik prozoru u spoljašnju klasu.

Pitanje koje zadire u srž unutrašnjih klasa glasi: ako mi je potrebna samo referenca na interfejs, zašto ne napravim spoljašnju klasu tako da realizuje taj interfejs? Odgovor je: „Ako vam je samo to potrebno, onda tako i uradite". Po čemu se, onda, unutrašnja klasa koja realizuje interfejs razlikuje od spoljne klase koja realizuje taj isti interfejs? Odgovor glasi: ne možete uvek da koristite pogodnosti interfejsa – ponekad morate da radite s realizacijama. Dakle, najznačajniji razlog za postojanje unutrašnjih klasa može da se formuliše na sledeći način:

Svaka unutrašnja klasa može nezavisno od drugih da nasledi realizaciju. Stoga, unutrašnja klasa nije ograničena time što je spoljna klasa već nasledila neku realizaciju.

Bez mogućnosti koje pružaju unutrašnje klase – da se istovremeno nasledi više od jedne konkretne ili **apstraktne** klase – neki problemi pri projektovanju i programiranju bili bi nerešivi. Jedan način za posmatranje unutrašnjih klasa je upotpunjavanje rešenja za problem višestrukog nasleđivanja. Deo problema rešavaju interfejsi, dok unutrašnje klase efektivno dozvoljavaju „višestruko nasleđivanje realizacije". Unutrašnje klase, dakle, efektivno omogućavaju da nasledite više klasa koje nisu interfejsi.

Da bismo ovo detaljnije proučili, razmotrimo situaciju u kojoj imamo dva interfejsa koja nekako moramo da realizujemo u jednoj klasi. Zbog fleksibilnosti interfejsa možete da birate jednu od dve mogućnosti: jedna klasa ili unutrašnje klase:

```
//: unutrasnjeklase/VisestrukiInterfejsi.java
// Dva načina na koje klasa može da realizuje više interfejsa.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B napraviB() {
    // Anonimna unutrašnja klasa:
    return new B() {};
  }
}
public class VisestrukiInterfejsi {
  static void uzimaA(A a) {}
  static void uzimaB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    uzimaA(x);
    uzimaA(y);
    uzimaB(x);
    uzimaB(y.napraviB());
  }
} ///:~
```

Pri tome se, naravno, podrazumeva da je struktura vašeg programa takva da oba načina budu logična. Obično sama priroda problema upućuje na pravu odluku: da li da upotrebite jednu klasu ili unutrašnje klase. U gornjem primeru, s tačke gledišta realizacije, gotovo da i nema nikakve razlike između ta dva pristupa. I jedan i drugi će raditi.

Međutim, ako umesto interfejsa imate apstraktne ili konkretne klase i ako vaša klasa mora nekako da realizuje i jedne i druge, morate upotrebiti unutrašnje klase:

```
//: unutrasnjeklase/VisestrukaRealizacija.java
// Ako su u pitanju konkretne ili apstraktne klase unutrašnje klase su
// jedini način za dobijanje efekta "višestrukog nasleđivanja realizacije".
package unutrasnjeklase;

class D {}
abstract class E {}

class Z extends D {
  E napraviE() { return new E() {}; }
}
```

```
public class VisestrukaRealizacija {
  static void uzimaD(D d) {}
  static void uzimaE(E e) {}
  public static void main(String[] args) {
    Z z = new Z();
    uzimaD(z);
    uzimaE(z.napraviE());
  }
} ///:~
```

Ako se niste susreli s problemom „višestrukog nasleđivanja realizacije", sve ostalo ste sasvim logično mogli da programirate i bez upotrebe unutrašnjih klasa. Ukoliko koristite unutrašnje klase, dobijate sledeće dodatne mogućnosti:

1. U jednom objektu spoljne klase možete da imate više instanci unutrašnje klase, od kojih svaka čuva svoje informacije koje zavise od informacija u objektu spoljne klase.
2. U jednoj spoljnoj klasi možete da imate više unutrašnjih klasa, od kojih svaka realizuje isti interfejs ili nasleđuje istu klasu na različit način. Ubrzo sledi primer.
3. Trenutak pravljenja objekta unutrašnje klase nije vezan za pravljenje objekta spoljašnje klase.
4. Pri radu sa unutrašnjom klasom nema potencijalno zbunjujuće relacije „je"; unutrašnja klasa je zaseban entitet.

Primera radi, da u programu **Sekvenca.java** nismo koristili unutrašnje klase, morali biste da kažete „**Sekvenca** *je* **Selektor**" i za određenu sekvencu bi mogao da postoji samo jedan **Selektor**. Mogli biste, takođe, da imate još jednu metodu, **obrnutiSelektor( )**; ona bi napravila **Selektor** koji se kreće unazad kroz sekvencu. Ova vrsta fleksibilnosti je moguća samo pri upotrebi unutrašnjih klasa.

Vežba 22: (2) U datoteci **Sekvenca.java** realizujte metodu **obrnutiSelektor( )**.

Vežba 23: (4) Napravite interfejs **U** koji ima tri metode. Napravite klasu **A** s metodom koja vraća referencu na **U** tako što pravi anonimnu unutrašnju klasu. Napravite drugu klasu, **B**, u kojoj se nalazi niz čiji su elementi reference na objekte klase **U**. Klasa **B** bi trebalo da ima jednu metodu koja prihvata i smešta u niz reference na **U**, drugu metodu koja postavlja na vrednost **null** određenu referencu u nizu (određenu argumentom metode) i treću metodu koja se kreće kroz niz i poziva metode iz **U**. U metodi **main( )** napravite grupu objekata klase **A** i jedan objekat klase **B**. Popunite **B** referencama na **U** koje su napravili objekti klase **A**. Upotrebite **B** za povratne pozive u sve objekte klase **A**. Izbacite neke reference na **U** iz objekta **B**.

Zaključci i povratni pozivi

Zaključak (engl. *closure*) jeste objekat kome možete da pristupate i koji čuva informacije iz oblasti važenja u kojoj je bio napravljen. Iz ove definicije vidite da je unutrašnja klasa objektno orijentisani zaključak, pošto ona sadrži delove informacija iz objekta spoljne klase („oblast važenja u kojoj je bio napravljen") i automatski čuva i referencu na ceo objekat spoljne klase u kome ima dozvolu da radi sa svim članovima, čak i privatnim.

Povratni pozivi (engl. *callbacks*) bili su jedan od najozbiljnijih argumenata u prilog uključivanja neke vrste mehanizma pokazivača u Javi. Pomoću povratnih poziva, nekom se objektu daje informacija koja mu omogućava da nešto kasnije pozove nazad objekat koji mu je uputio povratni poziv. Kao što ćete videti u nastavku knjige, to je veoma moćan koncept. Ako se povratni pozivi realizuju pomoću pokazivača, morate se osloniti na to da će se programer lepo ponašati i da neće zloupotrebiti pokazivač. Kao što ste do sada videli, u Javi se teži ka još većoj opreznosti, pa pokazivači nisu uključeni u jezik.

Zaključak koji obezbeđuje unutrašnja klasa savršeno je rešenje za povratni poziv; mnogo fleksibilnije i daleko sigurnije nego pokazivač. Evo jednostavnog primera:

```
//: unutrasnjeklase/PovratniPozivi.java
// Upotreba unutrašnjih klasa za povratne pozive
package unutrasnjeklase;
import static net.mindview.util.Print.*;

interface MozeDaSePoveca {
  void povecaj();
}

// Veoma jednostavna, samo realizuje interfejs:
class Pozvana1 implements MozeDaSePoveca {
  private int i = 0;
  public void povecaj() {
    i++;
    print(i);
  }
}

class MojaPovecaj {
  public void povecaj() { print("Druga operacija"); }
  static void f(MojaPovecaj mp) {  mp.povecaj(); }
}

// Ako vaša klasa mora da realizuje metodu povecaj()
// na neki drugi način, morate da upotrebite unutrašnju klasu:
class Pozvana2 extends MojaPovecaj {
  private int i = 0;
  private void povecaj() {
    super.povecaj();
    i++;
    print(i);
  }
  private class Zakljucak implements MozeDaSePoveca {
    public void povecaj() {
      // Specificirajte metodu iz spoljne klase, inače ćete dobiti
      // beskonačnu rekurziju:
      Pozvana2.this.povecaj();
    }
  }
```

```
  MozeDaSePoveca uzmiReferencuZaPovratniPoziv() {
    return new Zakljucak();
  }
}

class Poziva {
  private MozeDaSePoveca referencaZaPovratniPoziv;
  Poziva(MozeDaSePoveca cbh) { referencaZaPovratniPoziv = cbh; }
  void kreni() { referencaZaPovratniPoziv.povecaj(); }
}

public class PovratniPozivi {
  public static void main(String[] args) {
    Pozvana1 c1 = new Pozvana1();
    Pozvana2 c2 = new Pozvana2();
    MojaPovecaj.f(c2);
    Poziva poziva1 = new Poziva(c1);
    Poziva poziva2 = new Poziva(c2.uzmiReferencuZaPovratniPoziv());
    poziva1.kreni();
    poziva1.kreni();
    poziva2.kreni();
    poziva2.kreni();
  }
} /* Ispis:
Druga operacija
1
1
2
Druga operacija
2
Druga operacija
3
*///:~
```

Ovaj primer takođe pokazuje dodatne razlike između realizovanja interfejsa u spoljnoj i unutrašnjoj klasi. U pogledu pisanja programa, klasa **Pozvana1** je očigledno jednostavnije rešenje. Klasa **Pozvana2** nasleđuje klasu **MojaPovecaj** u okviru koje već postoji metoda **povecaj( )** čiji zadatak nema nikakve veze sa onim koji očekujemo od interfejsa **Povecaj**. Kada klasu **Pozvana2** nasledimo iz klase **MojaPovecaj**, njenu metodu **povecaj( )** ne možemo da redefinišemo tako da može da se koristi s interfejsom **MozeDaSePoveca**, pa morate da obezbedite posebnu realizaciju pomoću unutrašnje klase. Obratite pažnju na to da pravljenjem unutrašnje klase niti proširujete niti menjate interfejs spoljne klase.

U klasi **Pozvana2** sve je privatno osim metode **uzmiReferencuZaPovratniPoziv( )**. Da biste omogućili *bilo kakvu* vezu sa spoljašnjim svetom, neophodan vam je interfejs **MozeDaSePoveca**. U ovom primeru vidite kako vam Javini interfejsi pomažu da potpuno razdvojite interfejs i realizaciju.

Unutrašnja klasa **Zakljucak** samo realizuje interfejs **MozeDaSePoveca** da bi obezbedila povratnu vezu ka klasi **Pozvana2** – ali sigurnu povratnu vezu. Svako ko dobije referencu na

interfejs **MozeDaSePoveca** može, naravno, samo da pozove metodu **povecaj( )** i to je sve (za razliku od pokazivača koji bi omogućio potpunu slobodu).

Konstruktor klase **Poziva** dobija referencu na interfejs **MozeDaSePoveca** (premda referencu za povratni poziv možete da prihvatite u bilo kom trenutku) i potom, nešto kasnije, upotrebljava tu referencu za povratni poziv u klasu **Pozvana**.

Značaj povratnih poziva leži u njihovoj fleksibilnosti, odnosno u mogućnosti da prilikom izvršavanja dinamički odlučite koje ćete funkcije pozvati. Vrednost ove tehnike postaće jasnija u poglavlju *Grafička korisnička okruženja*, u kome se za pravljenje grafičkog korisničkog okruženja (GUI) svuda koriste povratni pozivi.

Unutrašnje klase i kosturi upravljanja

Kao konkretniji primer upotrebe unutrašnjih klasa može da se primeni postupak koji ću nazvati *kostur upravljanja* (engl. *control framework*).

Kostur programa (engl. *application framework*) jeste klasa ili skup klasa koje su projektovane da reše određeni problem. Kostur programa primenjujete tako što nasledite jednu ili više klasa i redefinišete neke metode. Kôd koji napišete u redefinisanim metodama prilagođava opšte rešenje koje pruža kostur aplikacije da bi rešio specifičan problem. To je primer projektnog obrasca *Template Method* (šablonska metoda) (videti *Thinking in Patterns (with Java)* na adresi *www.MindView.net*). Šablonska metoda sadrži osnovnu strukturu algoritma; da bi dovršila akciju tog algoritma, ona poziva jednu ili više metoda koje se mogu redefinisati. Projektni obrazac razdvaja ono što se menja od onoga što se ne menja; u ovom slučaju, ono što se ne menja je Šablonska metoda, a menjaju se metode koje se mogu redefinisati.

Kostur upravljanja je poseban tip kostura programa, a osnovna namena mu je da se odgovori na događaje; sistem čija je osnovna funkcija da reaguje na događaje naziva se *sistem kojim upravljaju događaji* (engl. *event-driven system*). Jedan od najvažnijih problema prilikom pravljenja programa je grafičko korisničko okruženje (GUI), kojim skoro potpuno upravljaju događaji. Kao što ćete videti u poglavlju *Grafička korisnička okruženja*, Javina biblioteka Swing je kostur upravljanja koji efikasno rešava problem grafičkog korisničkog okruženja, pri čemu često koristi unutrašnje klase.

Da biste razumeli na koji način unutrašnje klase omogućavaju jednostavno pravljenje i upotrebu kostura upravljanja, razmotrimo kostur upravljanja čiji je zadatak da izvrši događaje svaki put kada su „spremni". Premda termin „spreman" može da označava bilo šta, u našem slučaju on će biti zasnovan na časovniku. Naredni kostur upravljanja ne sadrži konkretne informacije o tome čime upravlja. Te informacije se daju tokom nasleđivanja, prilikom realizacije onog dela algoritma koji je nazvan **akcija( )**.

Prvo ćemo definisati interfejs koji opisuje bilo kakav upravljački događaj. On je napisan u obliku apstraktne klase, a ne kao pravi interfejs, jer je njegovo podrazumevano ponašanje da upravlja oslanjajući se na vreme, tako da u njemu mora da bude deo realizacije:

```
//: unutrasnjeklase/upravljac:Dogadjaj.java
// Zajedničke metode za bilo koji upravljački događaj.
package unutrasnjeklase.upravljac;
```

```
public abstract class Dogadjaj {
  private long dogVreme;
  protected final long vremeKasnjenja;
  public Dogadjaj(long vremeKasnjenja) {
    this.vremeKasnjenja = vremeKasnjenja;
    start();
  }
  public void start() { // Omogućava ponovno pokretanje
    dogVreme = System.nanoTime() + vremeKasnjenja;
  }
  public boolean spreman( ) {
    return System.nanoTime() >= dogVreme;
  }
  public abstract void akcija();
} ///:~
```

Konstruktor samo zapisuje trenutak (mereno od trenutka nastanka objekta) kada želite da bude pokrenut **Dogadjaj**, i zatim poziva metodu **start( )** koja tekućem vremenu dodaje vreme kašnjenja i tako se dobija vreme kada će se događaj odigrati. Umesto da bude uključena u konstruktor, **start( )** je zasebna metoda. Stoga merač vremena možete ponovo da pokrenete nakon odigravanja događaja, pa se objekat **Dogadjaj** može ponovo upotrebiti. Na primer, da biste dobili događaj koji se ponavlja, dovoljno je da iz metode **akcija( )** pozovete **start( )**.

Metoda **spreman( )** govori da li je došlo vreme da se pokrene **akcija( )**. Metoda **spreman( )**, naravno, može da bude redefinisana u izvedenoj klasi, kako bi izvršavanje događaja bilo zasnovano na nečem drugom.

Naredna datoteka sadrži stvarni kostur upravljanja koji upravlja događajima i poziva ih. Objekti klase **Dogadjaj** čuvaju se u kontejnerskom objektu tipa **List<Dogadjaj>** (čita se „lista događaja"), o kojoj ćete više saznati u poglavlju *Čuvanje objekata*. Za sada, dovoljno je znati da metoda **add( )** dodaje **Dogadjaj**e na kraj **List**e, **size( )** daje broj stavki u **List**i, foreach sintaksa pribavlja sukcesivne **Dogadjaj**e iz **List**e, a **remove( )** uklanja dati **Dogadjaj** iz **List**e.

```
//: unutrasnjeklase/upravljac/Upravljac.java
// Generički kostur za sve upravljačke sisteme.
package unutrasnjeklase.upravljac;
import java.util.*;

public class Upravljac {
  // Klasa iz paketa java.util za čuvanje objekata klase Dogadjaj:
  private List<Dogadjaj> listaDogadjaja = new ArrayList<Dogadjaj>();
  public void dodajDogadjaj(Dogadjaj c) { listaDogadjaja.add(c); }
  public void pokreni() {
    while(listaDogadjaja.size() > 0)
      // Napravite kopiju da ne biste modifikovali listu
      // dok birate elemente iz nje:
      for(Dogadjaj d : new ArrayList<Dogadjaj>(listaDogadjaja))
```

```
        if(d.spreman()) {
          System.out.println(d);
          d.akcija();
          listaDogadjaja.remove(d);
        }
    }
} ///:~
```

Metoda **pokreni( )** kruži kroz kopiju **listeDogadjaja**, tražeći **Dogadjaj** koji je **spreman( )** za pokretanje. Za svaki **Dogadjaj** koji je **spreman( )**, ispisuju se podaci objektovom metodom **toString( )** i poziva metoda **akcija( )**. Zatim se **Dogadjaj** uklanja iz liste.

Uviđate da za sada ne znate ništa o tome *šta tačno* klasa **Dogadjaj** radi. To i jeste srž ovog postupka; kako „razdvaja ono što se menja od onog što ostaje isto". Odnosno, da iskoristim svoj termin, „vektor promene" predstavlja različite akcije različitih vrsta objekata klase **Dogadjaj** koje definišete praveći različite potklase za događaje.

Na ovom mestu na scenu stupaju unutrašnje klase. One omogućavaju dve stvari:

1. Celu realizaciju programa koji upotrebljava kostur upravljanja možete da smestite u jednu klasu, čime kapsulirate sve što je za tu realizaciju potrebno. Unutrašnje klase se upotrebljavaju da iskažu više različitih akcija koje su potrebne za rešavanje problema.
2. Unutrašnje klase pojednostavljuju ovu realizaciju, a vi ste u mogućnosti da lako pristupite članovima spoljne klase. Bez ovih mogućnosti, program bi mogao da postane toliko komplikovan da biste morali da potražite alternativu.

Razmotrimo jednu realizaciju kostura upravljanja koja upravlja funkcijama staklene bašte.[4] Svaka akcija je potpuno različita: paljenje svetla, uključivanje vode i termostata, uključivanje zvonca i ponovno pokretanje sistema. Kostur upravljanja omogućuje da se taj različiti kôd lako razdvaja. Unutrašnje klase omogućavaju da pomoću samo jedne klase napravite više izvedenih verzija osnovne klase **Dogadjaj**. Za svaki tip akcije napravićete novu unutrašnju klasu izvedenu iz klase **Dogadjaj** i kôd za upravljanje napisati unutar metode **akcija( )**.

Kao što je za kostur upravljanja uobičajeno, klasa **UpravljanjeStaklenomBastom** je izvedena iz klase **Upravljac**:

```
//: unutrasnjeklase/UpravljanjeStaklenomBastom.java
// Ovim se pravi poseban program za upravljački sistem koji se ceo nalazi
// u jednoj klasi. Unutrašnje klase omogućavaju da kapsulirate različitu
// funkcionalnost za svaki poseban tip dogadjaja.
import unutrasnjeklase.upravljac.*;

public class UpravljanjeStaklenomBastom extends Upravljac {
  private boolean svetlo = false;
  private class UkljuciSvetlo extends Dogadjaj {
    public UkljuciSvetlo(long vremeKasnjenja) { super(vremeKasnjenja); }
    public void akcija() {
```

[4] Iz nekog razloga, ovaj problem sam uvek voleo da rešavam; on potiče iz moje ranije knjige *C++ Inside & Out*, međutim, Java omogućava mnogo bolje rešenje.

```
      // Ovde treba da bude kōd za upravljanje hardverom
      // kojim se fizički pali svetlo.
      svetlo = true;
    }
    public String toString()  { return "Svetlo je uključeno"; }
  }
  public class IskljuciSvetlo extends Dogadjaj {
    public IskljuciSvetlo(long vremeKasnjenja) { super(vremeKasnjenja); }
    public void akcija() {
      // Ovde treba da bude kōd za upravljanje hardverom.
      // kojim se fizički gasi svetlo.
      svetlo = false;
    }
    public String toString() { return "Svetlo je isključeno"; }
  }
  private boolean voda = false;
  public class UkljuciVodu extends Dogadjaj {
    public UkljuciVodu(long vremeKasnjenja) {super(vremeKasnjenja);}
    public void akcija() {
      // Ovde treba da bude kōd za upravljanje hardverom.
      voda = true;
    }
    public String toString() { return "Voda u stakleniku je puštena"; }
    }
  public class IskljuciVodu extends Dogadjaj {
    public IskljuciVodu(long vremeKasnjenja) { super(vremeKasnjenja); }
    public void akcija() {
      // Ovde treba da bude kōd za upravljanje hardverom.
      voda = false;
    }
    public String toString() { return "Voda u stakleniku je zatvorena"; }
    }
  }
  private String termostat = "Dan";
  public class TermostatNoc extends Dogadjaj {
    public TermostatNoc(long vremeKasnjenja) {
      super(vremeKasnjenja);
    }
    public void akcija() {
      // Ovde treba da bude kōd za upravljanje hardverom.
      termostat = "Noć";
    }
    public String toString() {
      return "Noćni režim termostata";
    }
  }
  public class TermostatDan extends Dogadjaj {
    public TermostatDan(long vremeKasnjenja) {
      super(vremeKasnjenja);
    }
```

```
    public void akcija() {
      // Ovde treba da bude kod za upravljanje hardverom.
      termostat = "Dan";
    }
    public String  toString() {
      return "Dnevni režim termostata";
    }
  }
  // Primer metode akcija() koja ubacuje
  // novu akciju u listu događaja.
  public class Zvono extends Dogadjaj {
    public Zvono(long vremeKasnjenja) { super(vremeKasnjenja); }
    public void akcija() {
      dodajDogadjaj(new Zvono(vremeKasnjenja));
    }
    public String toString() {  return "Zvonce!"; }
  }
  public class Restart extends Dogadjaj {
    private Dogadjaj[] listaDogadjaja;
    public Restart(long vremeKasnjenja) {
      super(vremeKasnjenja);
      this.listaDogadjaja = listaDogadjaja;
      for(Dogadjaj d : listaDogadjaja)  {
        dodajDogadjaj(d);
      }
    }
    public void akcija() {
      for(Dogadjaj d : listaDogadjaja)  {
        d.start(); // Ponovo pokreni svaki događaj
        dodajDogadjaj(d);
      }
      start(); // Ponovo pokreni ovaj događaj
      dodajDogadjaj(this);
    }
    public String toString() {
      return "Restartovanje sistema";
    }
  }
  public static class Zavrsi extends Dogadjaj {
    public Zavrsi(long vremeKasnjenja)  { super(vremeKasnjenja); }
    public void akcija() { System.exit(0); }
    public String toString() { return "Završavanje"; }
  }
} ///:~
```

Promenljive **svetlo**, **voda** i **termostat** pripadaju spoljnoj klasi **UpravljanjeStaklenomBastom**, a ipak unutrašnje klase mogu da im pristupe bez upotrebe kvalifikatora i bez posebnih dozvola. U stvarnom programu, većina metoda **akcija**( ) bi uključivala neku vrstu upravljanja hardverom.

Većina klasa izvedenih iz klase **Dogadjaj** slične su, ali se klase **Zvono** i **Restart** izdvajaju. Klasa **Zvono** pokreće zvonce i u listu događaja dodaje još jedan objekat klase **Zvono**, tako da će ponovo zazvoniti nešto kasnije. Obratite pažnju na to da unutrašnja klasa izgleda *gotovo* kao višestruko nasleđivanje: klasa **Zvono** ima sve metode kao i klasa **Dogadjaj**, a takođe izgleda kao da ima i sve metode spoljne klase **UpravljanjeStaklenomBastom.**

Klasa **Restart** dobija niz objekata **Dogadjaj** koje dodaje upravljaču. Pošto je **Restart** samo još jedna vrsta klase **Dogadjaj**, u metodi **Restart.akcija**( ) u listu takođe možete da dodate i objekat klase **Restart**, kako bi se sistem s vremena na vreme ponovo pokrenuo.

Naredna klasa konfiguriše sistem tako što pravi objekat klase **UpravljanjeStaklenomBastom** i dodajte razne vrste objekata **Dogadjaj**. Ovo je primer projektnog obrasca *Command* (naredba) – svaki objekat klase **listaDogadjaja** jeste zahtev kapsuliran kao objekat:

```
//: unutrasnjeklase/UpravljacStakleneBaste.java
// Konfiguriši i pokreni sistem staklene bašte.
// {Args: 5000}
import unutrasnjeklase.upravljac.*;

public class UpravljacStakleneBaste {
  public static void main(String[] args) {
    UpravljanjeStaklenomBastom gc = new UpravljanjeStaklenomBastom();
    // Umesto upisivanja direktno u program, na ovom mestu biste mogli
    // da učitate konfiguraciju iz tekstualne datoteke:
    gc.dodajDogadjaj(gc.new Zvono(900));
    Dogadjaj[] listaDogadjaja = {
      gc.new TermostatNoc(0),
      gc.new UkljuciSvetlo(200),
      gc.new IskljuciSvetlo(400),
      gc.new UkljuciVodu(600),
      gc.new IskljuciVodu(800),
      gc.new TermostatDan(1400)
    };
    gc.dodajDogadjaj(gc.new Restart(2000, listaDogadjaja));
    if(args.length == 1)
      gc.dodajDogadjaj(
        new UpravljanjeStaklenomBastom.Zavrsi(
          new Integer(args[0])));
    gc.pokreni();
  }
} /* Ispis:
Zvonce!
Noćni režim termostata
Svetlo je uključeno
Svetlo je isključeno
Voda u stakleniku je uključena
Voda u stakleniku je isključena
Dnevni režim termostata
Restartovanje sistema
Završavanje
*///:~
```

Ova klasa je odgovorna za inicijalizaciju sistema, pa ona uključuje sve potrebne događaje. Događaj **Restart** se pokreće više puta, a on svaki put učitava klasu **listaDogadjaja** u objekat **UpravljanjeStaklenomBastom**. Ukoliko zadate broj milisekundi kao argument na komandnoj liniji, **Restart** će okončati program posle toliko milisekundi. (To se upotrebljava priliom testiranja.)

Naravno, to se može postići i na fleksibilniji način – umesto da upišemo događaje direktno u program, pročitaćemo njihovu inicijalizacionu listu iz datoteke. U jednom vežbanju iz poglavlja *Javin ulazno-izlazni sistem* baš se to zahteva od vas.

Ovaj primer bi trebalo da vas uveri u vrednost unutrašnjih klasa, naročito kada se koriste unutar kostura upravljanja. U poglavlju *Grafička korisnička okruženja* videćete kako se unutrašnje klase jednostavno upotrebljavaju za opisivanje akcija grafičkog okruženja. Kada pročitate i to poglavlje, bićete uvereni u njihovu vrednost.

Vežba 24: (2) U datoteci **UpravljanjeStaklenomBastom.java** dodajte unutrašnje klase tipa **Dogadjaj** koje uključuju i isključuju ventilatore. Konfigurišite datoteku **UpravljacStakleneBaste.java** tako da upotrebljava te nove objekte tipa **Dogadjaj**.

Vežba 25: (3) U datoteci **UpravljanjeStaklenomBastom.java** nasledite klasu **UpravljanjeStaklenomBastom** da biste dodali unutrašnje klase tipa **Dogadjaj** koje uključuju i isključuju generatore vodenih kapljica. Napišite novu verziju programa **UpravljacStakleneBaste.java** tako da upotrebljava te nove objekte tipa **Dogadjaj**.

Nasleđivanje unutrašnjih klasa

Pošto konstruktor unutrašnje klase mora da bude povezan s referencom na objekat spoljne klase, pri njenom nasleđivanju stvari se malo komplikuju. Problem je što ta „tajna" referenca na spoljni objekat *mora* da bude inicijalizovana, a u izvedenoj klasi više ne postoji podrazumevani objekat za koji bi ona bila povezana. Rešenje je upotrebiti sintaksu koja izričito određuje povezivanje:

```
//: unutrasnjeklase/NasleduUnutrasnju.java
// Nasleđivanje unutrašnje klase.

class ImaUnutrasnju {
  class Unutrasnja {}
}

public class NasleduUnutrasnju extends ImaUnutrasnju.Unutrasnja {
  //! NasleduUnutrasnju() {} // Ne može da se prevede
  NasleduUnutrasnju(ImaUnutrasnju iu) {
    iu.super();
  }
  public static void main(String[] args) {
    ImaUnutrasnju iu = new ImaUnutrasnju ();
    NasleduUnutrasnju ii = new NasleduUnutrasnju(iu);
  }
} ///:~
```

Vidite da klasa **NasledиUnutrasnju** proširuje samo unutrašnju klasu, ali ne i spoljašnju. Kada dođe vreme da se napravi konstruktor, podrazumevani konstruktor vam ne odgovara, a ne možete samo da prosledite referencu spoljnom objektu. Pored toga, u konstruktoru morate da upotrebite sintaksu:

```
ReferencaNaSpoljnuKlasu.super();
```

Time obezbeđujete neophodnu referencu i moći ćete da prevedete program.

Vežba 26: (2) Napravite klasu koja sadrži unutrašnju klasu koja ima nepodrazumevani konstruktor (onaj koji prima argumente). Napravite drugu klasu koja sadrži unutrašnju klasu nasledenu iz prve unutrašnje klase.

Da li unutrašnja klasa može da se redefiniše?

Šta se dešava kada napravite unutrašnju klasu, nasledite spoljnu klasu i potom redefinišete unutrašnju? Odnosno, da li je moguće redefinisati unutrašnju klasu? Čini se da bi to bio moćan koncept, ali „redefinisanjem" unutrašnje klase kao da je ona metoda spoljne klase, u suštini se ništa ne postiže:

```
//: unutrasnjeklase/VelikoJaje.java
// Unutrašnja klasa ne može da bude redefinisana poput metode.
import static net.mindview.util.Print.*;

class Jaje {
  private Zumance z;
  protected class Zumance {
    public Zumance() { print("Jaje.Zumance()"); }
  }
  public Jaje() {
    print("Novo Jaje()");
    z = new Zumance();
  }
}

public class VelikoJaje extends Jaje {
  public class Zumance {
    public Zumance() { print("VelikoJaje.Zumance()"); }
  }
  public static void main(String[] args) {
    new VelikoJaje();
  }
} /* Ispis:
Novo Jaje()
Jaje.Zumance()
*///:~
```

Prevodilac automatski pravi podrazumevani konstruktor koji poziva podrazumevani konstruktor osnovne klase. S obzirom na to da se pravi klasa **VelikoJaje**, mogli biste očekivati da će biti pozvana „redefinisana" verzija klase **Zumance**, ali to se ne dešava, kao što vidite iz rezultata.

Ovaj primer samo pokazuje da se prilikom nasleđivanja spoljašnjih klasa ništa posebno ne dešava u unutrašnjim. Dve unutrašnje klase su potpuno odvojeni entiteti, svaka u svom imenskom prostoru. Ipak, moguće je izričito nasleđivanje unutrašnje klase:

```
//: unutrasnjeklase/VelikoJaje2.java
// Pravilno nasleđivanje unutrašnje klase
import static net.mindview.util.Print.*;

class Jaje2 {
  protected class Zumance {
    public Zumance() { print("Jaje2.Zumance()"); }
    public void f() { print("Jaje2.Zumance.f()"); }
  }
  private Zumance z = new Zumance();
  public Jaje2() { print("Novo Jaje2()"); }
  public void ubaciZumance(Zumance zz) { z = zz; }
  public void g() { z.f(); }
}

public class VelikoJaje2 extends Jaje2 {
  public class Zumance extends Jaje2.Zumance {
    public Zumance() { print("VelikoJaje2.Zumance()"); }
    public void f() { print("VelikoJaje2.Zumance.f()"); }
  }
  public VelikoJaje2() { ubaciZumance(new Zumance()); }
  public static void main(String[] args) {
    Jaje2 e2 = new VelikoJaje2();
    e2.g();
  }
} /* Ispis:
Jaje2.Zumance()
Novo Jaje2()
Jaje2.Zumance()
VelikoJaje2.Zumance()
VelikoJaje2.Zumance().f()
*///:~
```

Sada klasa **VelikoJaje2.Zumance** eksplicitno proširuje klasu **Jaje2.Zumance** i redefiniše njene metode. Metoda **ubaciZumance( )** omogućava da klasa **VelikoJaje2** jedan od svojih unutrašnjih objekata klase **Zumance** svede naviše ka referenci **z** u klasi **Jaje2**. Kada metoda **g( )** pozove metodu **z.f( )** biće upotrebljena redefinisana verzija metode **f( )**. Drugi poziv konstruktoru osnovne klase **Jaje2.Zumance( )** jeste poziv iz konstruktora **VelikoJaje2.Zumance**. I sami vidite da će prilikom poziva metode g( ) biti pozvana redefinisana verzija metode f( ).

Lokalne unutrašnje klase

Kao što je rečeno, unutrašnje klase se prave i unutar blokova koda, najčešće unutar tela metode. Lokalna unutrašnja klasa ne može imati specifikator pristupa, pošto ona nije deo spoljne klase, ali može pristupati finalnim promenljivama u tekućem bloku koda i svim članovima okružujuće klase. Evo primera gde se poredi pravljenje lokalne unutrašnje klase i anonimne unutrašnje klase:

```
//: unutrasnjeklase/LokalnaUnutrasnjaKlasa.java
// Čuva sekvencu Objekata.
import static net.mindview.util.Print.*;

interface Brojac {
  int next();
}

public class LokalnaUnutrasnjaKlasa {
  private int broj = 0;
  Counter uzmiBrojac(final String ime) {
    // Lokalna unutrasnja klasa:
    class LokalniBrojac implements Brojac {
      public LokalniBrojac() {
        // Lokalna unutrasnja klasa može imati konstruktor
        print("LokalniBrojac()");
      }
      public int next() {
        printnb(ime); // Pristupanje lokalnoj finalnoj
        return broj++;
      }
    }
    return new LokalniBrojac();
  }
  // Isto sa anonimnom unutrašnjom klasom:
  Brojac LokalniBrojac2(final String ime) {
    return new Brojac() {
      // Anonimna unutrašnja klasa ne može imati imenovani
      // konstruktor, nego samo inicijalizator instance:
      {
        print("Brojac()");
      }
      public int next() {
        printnb(ime); // Pristupanje lokalnoj finalnoj
        return broj++;
      }
    };
  }
  public static void main(String[] args) {
    LokalnaUnutrasnjaKlasa lic = new LokalnaUnutrasnjaKlasa();
```

```
    Brojac
      c1 = lic.uzmiBrojac("Lokalna unutrašnja "),
      c2 = lic.uzmiBrojac("Anonimna unutrašnja ");
    for(int i = 0; i < 5; i++)
      print(c1.next());
    for(int i = 0; i < 5; i++)
      print(c2.next());
  }
} /* Ispis:
LokalniBrojac()
Brojac()
Lokalna unutrašnja 0
Lokalna unutrašnja 1
Lokalna unutrašnja 2
Lokalna unutrašnja 3
Lokalna unutrašnja 4
Anonimna unutrašnja 5
Anonimna unutrašnja 6
Anonimna unutrašnja 7
Anonimna unutrašnja 8
Anonimna unutrašnja 9
*///:~
```

Brojac vraća sledeću vrednost u sekvenci. Realizovan je i kao lokalna klasa i kao anonimna unutrašnja klasa. Obe te klase jednako se ponašaju i imaju iste mogućnosti. Pošto ime lokalne unutrašnje klase nije dostupno izvan te metode, jedino opravdanje za upotrebu te klase umesto anonimne unutrašnje klase bila bi potreba za imenovanim konstruktorom i/ili preklopljenim konstruktorom, jer anonimna unutrašnja klasa može imati samo inicijalizaciju instance.

Drugi razlog za pravljenje lokalne unutrašnje klase umesto anonimne unutrašnje klase bila bi potreba da se napravi više objekata te klase.

Identifikatori unutrašnjih klasa

Pošto za svaku klasu postoji datoteka **.class** u kojoj se čuvaju sve informacije o pravljenju objekata tog tipa (pomoću tih informacija se dobija tzv. meta klasa – objekat tipa **Class**), verovatno pogađate da i za unutrašnje klase mora postojati *njihova* datoteka **.class**. Imena tih datoteka/klasa dobijaju se po preciznoj formuli: ime spoljne klase posle koga dolazi znak **$**, a zatim ime unutrašnje klase. Na primer, u programu **LokalnaUnutrasnjaKlasa.java** biće napravljene sledeće datoteke **.class**:

```
Brojac.class
LokalnaUnutrasnjaKlasa$1.class
LokalnaUnutrasnjaKlasa$1LokalniBrojac.class
LokalnaUnutrasnjaKlasa.klasa
```

Ukoliko su unutrašnje klase anonimne, prevodilac koristi brojeve kao njihove identifikatore. Ako su unutrašnje klase ugnežđene unutar drugih unutrašnjih klasa, dodaju se odgovarajuća imena nakon znaka $ i identifikatora spoljnih klasa.

Način generisanja internih imena je jednostavan, ali istovremeno robustan i sposoban da obradi većinu situacija.[5] Pošto je to standardna tehnika za modelu imena u Javi, generisane datoteke će automatski biti nezavisne od platforme. (Natera prevodilac na više načina menja vaše unutrašnje klase kako bi mogle da rade.)

Sažetak

Interfejsi i unutrašnje klase napredniji su koncepti od onih koji postoje u mnogim OOP jezicima. Na primer, u C++-u ne postoji ništa nalik njima. Oni zajedno rešavaju isti problem koji je i C++ pokušao da reši pomoću višestrukog nasleđivanja. Ispostavilo se da se u C++-u višestruko nasleđivanje prilično teško upotrebljava, dok su Javini interfejsi i unutrašnje klase mnogo pristupačniji.

Iako su te mogućnosti prilično jasne, njihova upotreba je pitanje projektovanja, poput polimorfizma. S vremenom ćete sve bolje prepoznavati situacije u kojima treba da koristite interfejs ili unutrašnju klasu, ili i jedno i drugo istovremeno. U ovom trenutku, bilo bi dobro da poznajete barem njihovu sintaksu i semantiku. Kako se budete sretali sa njima, sve više ćete ih prihvatati.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solutions Guide*, koji se može kupiti na lokaciji *www.MindView.net*.

[5] S druge strane, $ je metaznak u Unixovim komandnim okruženjima, pa ćete ponekad imati problema prilikom listanja datoteka **.class**. Ovo je pomalo čudno, jer kompanija Sun radi u Unixu. Pretpostavljam da to pitanje u Sunu nisu ni razmatrali, misleći da će se verovatno usredsrediti na datoteke sa izvornim kodom.

Čuvanje objekata

PRILIČNO JE JEDNOSTAVAN PROGRAM KOJI KORISTI TAČNO ODREĐEN BROJ OBJEKATA poznatog vremena trajanja. U programima po pravilu obično pravite nove objekte na osnovu nekog kriterijuma koji je poznat samo u vreme njegovog izvršavanja. Sve dok program ne počne da se izvršava, nećete znati koliko vam objekata treba, niti kog su oni tipa. Da biste rešili ovaj problem, morate biti u stanju da napravite proizvoljan broj objekata, u bilo kom trenutku, bilo gde. Dakle, ne možete se osloniti na to da ćete svakom objektu moći da pristupate preko neke reference kojoj je dodeljeno ime:

```
MojaKlasa referenca
```

pošto nikada nećete znati koliko će vam objekata zaista trebati.

Većina jezika ima neki način za rešavanje ovog temeljnog problema. Java nudi nekoliko načina za čuvanje objekata (odnosno, referenci na objekte). U sam jezik je ugrađen tip niza, o kome je već bilo reči. Niz je najefikasniji način čuvanja grupe objekata i preporučuje se za čuvanje grupe prostih tipova. Ali niz ima nepromenljivu veličinu, a u opštijem slučaju, u trenutku pisanja programa vi ne znate koliko objekata će vam biti potrebno, niti treba li vam neki napredniji način skladištenja objekata; zato je nepromenljiva veličina niza preteško ograničenje.

Za rešavanje ovog problema, biblioteka Javinih uslužnih klasa ima prilično zaokružen skup *kontejnerskih klasa*, čiji su osnovni tipovi **List**, **Set**, **Queue** (za redove za čekanje) i **Map**. Ovi tipovi objekata poznati su i pod imenom *klase kolekcija*, ali pošto se naziv **Collection** u Javinoj biblioteci koristi za označavanje određenog podskupa te biblioteke, koristiću precizniji pojam „kontejner". Kontejneri omogućavaju naprednije načine za smeštanje objekata i pomoću njih možete da rešite iznenađujuće veliki broj problema.

Pored ostalih svojih obeležja – na primer, **Set** (skup) ne može da sadrži ponovljene elemente, a **Map** (mapa) je asocijativan niz koji služi za čuvanje parova međusobno asociranih objekata – Javine kontejnerske klase automatski podešavaju svoju veličinu. Stoga, za razliku od nizova, u kontejner možete staviti proizvoljan broj proizvoljno velikih objekata; dok pišete program, ne morate znati koliki kontejner treba da napravite.

Iako u Javi nemaju direktnu podršku kroz rezervisane reči,[1] kontejnerske klase su jedna od najsnažnijih alatki za programiranje, jer primetno skraćuju vreme pisanja programa. U ovom poglavlju steći ćete osnovna znanja potrebna za rad s Javinom bibliotekom kontejnera u tipičnim primenama. Bavićemo se kontejnerima koje ćete upotrebljavati u svakodnevnom programiranju. Kasnije, u poglavlju *Detaljno razmatranje kontejnera*, upoznaćemo vas sa ostalim kontejnerima i s pojedinostima o njihovoj funkcionalnosti i načinima korišćenja.

---

[1] Ugrađenu podršku za kontejnere imaju brojni jezici, među kojima Perl, Python i Ruby.

Generičke klase i kontejneri za bezbedan rad s tipovima

Pre Jave SE5, jedan od problema prilikom korišćenja kontejnera bio je to što je prevodilac dopuštao umetanje pogrešnog tipa u kontejner. Na primer, recimo da nam treba kontejner sa objektima tipa **Jabuka** i da ćemo upotrebiti osnovni kontejner za sve namene, **ArrayList**. Za početak, smatrajte da je **ArrayList** „dinamički niz koji se automatski proširuje". Korišćenje klase **ArrayList** je jednostavno: treba da napravite kontejner, smestite u njega objekte metodom **add( )**, a kasnije da ih izvadite metodom **get( )** uz korišćenje indeksa, baš kao u slučaju niza, ali bez uglastih zagrada.[2] **ArrayList** ima i metodu **size( )** koja vraća trenutni broj elemenata i time vas sprečava da slučajno dođete do kraja i prouzrokujete „izuzetak tokom izvršavanja" (engl. *runtime exception*); izuzeci će biti predstavljeni u poglavlju *Obrada grešaka pomoću izuzetaka.*

U ovom primeru, u kontejner se stavljaju i iz njega vade objekti tipova **Jabuka** i **Narandza**. Pošto u primeru *nisu* upotrebljeni generički tipovi, Javin prevodilac će vas upozoriti na grešku. Da bismo sprečili ispisivanje tog upozorenja, ovde smo upotrebili posebnu *anotaciju* Jave SE5. Anotacije počinju znakom @ i primaju argument; ova glasi **@SuppressWarnings** („unchecked"), a njen argument pokazuje da ne želimo ispisivanje samo upozorenja „unchecked", koje kazuje da nije jasno koji se tip (Jabuka ili Narandza) dobija iz kontejnera:

```
//: cuvanje/JabukeINarandzeBezGenerickihTipova.java
// Jednostavan primer s kontejnerom (napisan tako da prevodilac
// daje upozorenja).
// {ThrowsException}
import java.util.*;

class Jabuka {
  private static long brojac;
  private final long id = brojac++;
  public long id() { return id; }
}

class Narandza {}

public class JabukeINarandzeBezGenerickihTipova {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    ArrayList jabuke = new ArrayList();
    for(int i = 0; i < 3; i++)
      jabuke.add(new Jabuka());
    // Nema prepreke da se jabukama dodaju Narandže:
    jabuke.add(new Narandza());
    for(int i = 0; i < jabuke.size(); i++)
```

[2] Ovde bi dobro došlo preklapanje operatora. Kontejnerske klase u jezicima C++ i C# proizvode čistiju sintaksu zato što se upotrebljava preklapanje operatora.

```
      ((Jabuka)jabuke.get(i)).id();
      // Narandza se otkriva tek prilikom izvršavanja
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Više o anotacijama u Javi SE5 saznaćete u poglavlju *Anotacije*.

Klase **Jabuka** i **Narandza** se razlikuju; nemaju ništa zajedničko, sem što potiču od natklase **Object**. (Ne zaboravite: ukoliko izričito ne navedete klasu od koje nasleđujete, automatski nasleđujete **Object**.) Pošto **ArrayList** sadrži klase **Object**, metodom **add()** kontejnera **ArrayList** u njega možete dodavati ne samo objekte tipa **Jabuka**, nego i objekte tipa **Narandza**, a da se ne pobuni ni prevodilac u vreme prevođenja, ni izvršno okruženje Jave u vreme izvršavanja. Kada metodom **get()** kontejnera **ArrayList** pokušate da pribavite ono što su po vašem mišljenju objekti tipa **Jabuka**, vratiće vam se referenca na **Object** koji morate svesti na tip **Jabuka**. Nadalje, ceo izraz morate zatvoriti u zagrade da bi se to svođenje obavilo pre pozivanja metode **id()** za klasu **Jabuka**; u protivnom, izazvaćete sintaksnu grešku.

U vreme izvršavanja, kada objekat tipa **Narandza** pokušate da svedete na tip **Jabuka**, javiće se greška u obliku prethodno spomenutog izuzetka.

U poglavlju *Generički tipovi* saznaćete da *pravljenje* klasa pomoću odgovarajućeg Javinog mehanizma ume da bude složeno, ali da je *primena* jednom definisanih generičkih klasa najčešće jednostavna. Primera radi, da biste definisali **ArrayList** predviđen za čuvanje objekata tipa **Jabuka**, pišete **ArrayList<Jabuka>** umesto samo **ArrayList**. Uglaste zagrade okružuju *parametre tipa* (može ih biti više) a oni zadaju tip(ove) koje taj primerak kontejnera može da primi.

Mehanizam generičkih tipova vas *već u vreme prevođenja* sprečava da u kontejner stavite pogrešan tip objekta.[3] Evo prethodnog primera napravljenog pomoću generičkih tipova:

```
//: cuvanje/JabukeINarandzePomocuGenerickihTipova.java
import java.util.*;

public class JabukeINarandzePomocuGenerickihTipova {
  public static void main(String[] args) {
    ArrayList<Jabuka> jabuke = new ArrayList<Jabuka>();
    for(int i = 0; i < 3; i++)
      jabuke.add(new Jabuka());
    // Greška koja se prijavljuje u vreme prevođenja:
    // jabuke.add(new Narandza());
    for(int i = 0; i < jabuke.size(); i++)
      System.out.println(jabuke.get(i).id());
    // Sintaksom foreach:
    for(Jabuka c : jabuke)
      System.out.println(c.id());
  }
```

[3] Na kraju poglavlja *Generički tipovi* razmotrićemo da li je to baš toliko težak problem. U istom poglavlju videćete i da su Javini generički tipovi podesni za mnogo više od kontejnera koji se staraju za bezbedan rad s tipovima.

```
} /* Ispis:
0
1
2
0
1
2
*///:~
```

Sada će prevodilac sprečiti stavljenje objekta tipa **Narandza** među **jabuke**, pa će se ta greška prijaviti u vreme prevođenja, a ne u vreme izvršavanja.

Obratite pažnju i na to da prilikom vađenja stavki iz **Liste** više nije potrebno svođenje tipova. Pošto **Lista** zna koji tip sadrži, ona obavlja svođenje kada vi pozovete **get()**. Dakle, ne samo da prevodilac zbog generičkih tipova proverava tip objekata koji stavljate u kontejner, nego je i sintaksa prilikom upotrebe objekata u njemu čistija.

Primer pokazuje i sledeće: ako vam ne treba indeks svakog elementa, za izbor svakog elementa u **Listi** možete upotrebiti sintaksu foreach.

Kada tip objekta u kontejneru zadate kao generički parametar, ne morate u kontejner stavljati samo taj tip, pošto svođenje naviše radi i s generičkim tipovima kao i sa ostalim tipovima:

```
//: cuvanje/GenerickiTipoviISvodjenjeNavise.java
import java.util.*;

class GreniSmit extends Jabuka {}
class Gala extends Jabuka {}
class Fudzi extends Jabuka {}
class Breburn extends Jabuka {}

public class GenerickiTipoviISvodjenjeNavise {
  public static void main(String[] args) {
    ArrayList<Jabuka> jabuke = new ArrayList<Jabuka>(
    jabuke.add(new GreniSmit());
    jabuke.add(new Gala());
    jabuke.add(new Fudzi());
    jabuke.add(new Breburn());
    for(Jabuka c : jabuke)
      System.out.println(c);
  }
} /* Ispis: (uzorak)
GreniSmit@7d772e
Gala@11b86e7
Fudzi@35ce36
Breburn@757aef
*///:~
```

Dakle, u kontejner određen da sadrži objekte tipa **Jabuka** možete dodavati i podtipove klase **Jabuka**.

Ispis proizvodi podrazumevana metoda **toString**() klase **Object**, koja ispisuje ime klase i heksadecimalni *heš kôd* objekta (koji generiše metoda **hashCode**()). Heš kodove ćemo detaljno razmotriti u poglavlju *Detaljno razmatranje kontejnera.*

Vežba 1: (2) Napravite novu klasu **MorskoPrase** sa članom **int brojPrasadi** koji se inicijalizuje u konstruktoru. Dodajte klasi metodu **skace**( ) koja ispisuje broj morskog praseta i da li ono skače. Napravite nov kontejner tipa **ArrayList** i dodajte nekoliko objekata klase **MorskoPrase** u listu. Sada upotrebite metodu **get**( ) da biste se kretali kroz listu i pozivali metodu **skace**( ) svakog objekta tipa **MorskoPrase**.

Osnovni pojmovi

Kontejnerska biblioteka Jave 2 obavlja zadatak „čuvanja objekata" i deli ga u dva zasebna pojma, izražena u obliku osnovnih interfejsa biblioteke:

1. **Collection (kolekcija):** grupa pojedinačnih elemenata na koje je često primenjeno jedno ili više pravila. **List** (lista) mora da čuva elemente u određenom redosledu, **Set** (skup) ne može da sadrži ponovljene elemente, a **Queue** (red) *daje* elemente redom koji zadaje *disciplina čekanja* (obično istim redom kojim su elementi umetani).
2. **Map (mapa):** grupa parova objekat–ključ, koja omogućava pronalaženje objekta pomoću njegovog ključa. **ArrayList** omogućava pronalaženje objekta pomoću njegovog indeksa, pa na neki način pridružuje brojeve objektima. *Mapa* omogućava pronalaženje objekta pomoću njemu pridruženog *drugog objekta.* Nju nazivaju i *asocijativan niz,* zato što objekte asocira (pridružuje) drugim objektima, ili *rečnik,* zato što objekat traži pomoću ključa, kao što se u rečniku definicija traži pomoću reči. **Map**e su moćne programske alatke.

Iako to nije uvek moguće, bilo bi idealno da veći deo vašeg koda radi s tim interfejsima, a da jedino mesto na kojem ćete spomenuti tačan tip koji upotrebljavate bude mesto pravljenja instance. Dakle, ovako možete napraviti **List**u:

```
List<Jabuka> jabuke = new ArrayList<Jabuka>();
```

Povedite računa o tome da je **ArrayList** sveden naviše na **List**, suprotno načinu na koji smo radili u prethodnim primerima. Interfejs koristimo da bismo realizaciju menjali samo na mestu pravljenja ukoliko naknadno odlučimo da je izmenimo. To se radi ovako:

```
List<Jabuka> jabuke = new LinkedList<Jabuka>();
```

Dakle, obično pravite objekat konkretne klase, svedete ga naviše na odgovarajući interfejs i zatim u ostatku koda upotrebljavate taj interfejs.

Ovaj pristup neće raditi uvek, zato što neke klase imaju dodatnu funkcionalnost. Na primer, **LinkedList** ima dodatne metode koje interfejs **List** ne sadrži, a **TreeMap** ima metode koje interfejs **Map** ne sadrži. Ako vam trebaju te metode, nećete moći da svedete naviše na opštiji interfejs.

Interfejs **Collection** poopštava ideju *sekvence* – načina čuvanja grupe objekata. Evo jednostavnog primera u kojem se **Collection** (ovde predstavljen **ArrayList**om) puni **Integer** objektima, a zatim se svaki element u rezultujućem kontejneru ispisuje:

```
//: cuvanje/JednostavnaKolekcija.java
import java.util.*;

public class JednostavnaKolekcija {
  public static void main(String[] args) {
    Collection<Integer> c = new ArrayList<Integer>();
    for(int i = 0; i < 10; i++)
      c.add(i); // Automatsko pakovanje
    for(Integer i : c)
      System.out.print(i + ", ");
  }
} /* Ispis:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*///:~
```

Pošto se u ovom primeru upotrebljavaju samo metode interfejsa **Collection**, odgovarao bi svaki objekat klase nasleđene od tog interfejsa, ali **ArrayList** predstavlja najosnovniji tip sekvence.

Ime metode **add**() navodi na pomisao da ona u kolekciju (**Collection**) dodaje nov element. Međutim, u dokumentaciji je precizno navedeno da **add**() „čini da ovaj primerak klase **Collection** sadrži dati element". To je urađeno zbog skupa (potklase **Set**), koji element dodaje kolekciji samo ukoliko se on tamo već ne nalazi. U slučaju klase **ArrayList** i svih vrsta **List**i, **add**() uvek znači „dodati", pošto **List**e dozvoljavaju postojanje duplikata.

Kroza sve kolekcije možete prolaziti sintaksom foreach, kao u prethodnom primeru. U nastavku poglavlja naučićete da upotrebljavate fleksibilniji *Iterator*.

Vežba 2: (1) Izmenite program **JednostavnaKolekcija.java** tako da se za **c** upotrebljava **Set**.

Vežba 3: (2) Izmenite program **unutrasnjeklase/Sekvenca.java** tako da mu možete dodati proizvoljan broj elemenata.

Dodavanje grupa elemenata

Metode za dodavanje grupa elemenata kontejneru (potklasi interfejsa **Collection**) imaju i klasa **Arrays** i klasa **Collections** paketa **java.util**. Metoda **Arrays.asList**() prima niz ili listu elemenata razdvojenih zarezima (koristeći argumente promenljive dužine) i pretvara ih u objekat tipa **List**. **Collections.addAll**() prima objekat tipa **Collection** i niz ili listu elemenata razdvojenih zarezima, čije elemente dodaje tom kontejneru. Evo primera u kojem su prikazane obe te metode, kao i konvencionalnija metoda **addAll**() koju sadrže svi tipovi kontejnera:

```
//: cuvanje/DodavanjeGrupa.java
// Dodavanje grupa elemenata objektima tipa Collection.
import java.util.*;

public class DodavanjeGrupa {
  public static void main(String[] args) {
    Collection<Integer> kolekcija =
      new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
```

```
    Integer[] josCelihBrojeva = { 6, 7, 8, 9, 10 };
    kolekcija.addAll(Arrays.asList(josCelihBrojeva));
    // Radi znatno brže, ali ovako ne možete
    // napraviti objekat tipa Collection:
    Collections.addAll(kolekcija, 11, 12, 13, 14, 15);
    Collections.addAll(kolekcija, josCelihBrojeva);
    // Pravi listu "podržanu" nizom:
    List<Integer> lista = Arrays.asList(16, 17, 18, 19, 20);
    lista.set(1, 99); // OK -- izmenjen jedan element
    // lista.add(21); // Greška pri izvršavanju zato što se
                      // pripadnom nizu ne može menjati veličina.
  }
} ///:~
```

Konstruktor za **Collection** može primiti drugi **Collection** koji upotrebljava za sopstvenu inicijalizaciju, pa **Arrays.asList()** možete upotrebiti za pravljenje ulaza za konstruktor. Međutim, **Collections.addAll()** radi mnogo brže, a jednako je lako napraviti **Collection** bez elemenata i zatim pozvati **Collections.addAll()**, pa tom pristupu treba dati prednost.

Metoda član **Collection.addAll()** kao argument može primiti samo drugi objekat tipa **Collection**, pa nije fleksibilna kao **Arrays.asList()** i **Collections.addAll()** koje primaju liste argumenata promenljive dužine.

Izlaz metode **Arrays.asList()** možete upotrebiti i neposredno, kao **List**u, ali je u tom slučaju on predstavljen nizom, a njegovu veličinu ne možete menjati. Ukoliko takvoj listi pokušate da dodate – **add()** – ili oduzmete – **delete()** – elemente, time biste pokušali da promenite veličinu niza, što u vreme izvršavanja izaziva grešku „Unsupported Operation".

Ograničenje metode **Arrays.asList()** jeste to što ona samo nagađa koji je rezultujući tip **List**e i ne obraća pažnju na to čemu je dodeljujete. Katkada to izaziva problem:

```
//: cuvanje/KaoNekakvaLista.java
// Arrays.asList() samo nagađa o kojem se tipu radi.
import java.util.*;

class Sneg {}
class Prsac extends Sneg {}
class Slab extends Prsac {}
class Krupan extends Prsac {}
class Krpe extends Sneg {}
class Slota extends Sneg {}

public class KaoNekakvaLista {
  public static void main(String[] args) {
    List<Sneg> sneg1 = Arrays.asList(
      new Krpe(), new Slota(), new Prsac());

    // Neće se prevesti:
    // List<Sneg> sneg2 = Arrays.asList(
    // new Slab(), new Krupan());
```

```
    // Prevodilac će saopštiti:
    // found   : java.util.List<Prsac>
    // required: java.util.List<Sneg>

    // Collections.addAll() neće se zbuniti:
    List<Sneg> sneg3 = new ArrayList<Sneg>();
    Collections.addAll(sneg3, new Slab(), new Krupan());

    // Daćemo smernicu eksplicitnim
    // zadavanjem tipa argumenta:
    List<Sneg> sneg4 = Arrays.<Sneg>asList(
       new Slab(), new Krupan());
  }
} ///:~
```

Kada pokušava da napravi **sneg2**, **Arrays.asList**() ima samo tipove klase **Prsac**, pa pravi **List<Prsac>**, a ne **List<Sneg>**, dok **Collections.addAll()** lepo radi jer iz prvog argumenta zna koji je ciljni tip.

Kao što vidite u primeru o pravljenju liste **sneg4**, u sredinu **Arrays.asList**() možete staviti „smernicu" koja će prevodiocu saopštiti stvarni ciljni tip rezultujućeg tipa **List** koji proizvodi **Arrays.asList**(). To se naziva eksplicitno *zadavanje tipa argumenta.*

Mape su komplikovanije (videćete), a Javina standardna biblioteka ne omogućava njihovu automatsku inicijalizaciju drugačije sem preko sadržaja neke druge **Map**e.

Ispisivanje sadržaja kontejnera

Sadržaj kontejnera se ispisuje lepo bez ikakve pomoći, za razliku od nizova gde morate upotrebiti **Arrays.toString**( ). Evo primera koji istovremeno ilustruje i najvažnije vrste kontejnera u Javi:

```
//: cuvanje/IspisivanjeKontejnera.java
// Sadržaj kontejnera se automatski ispisuje u čitljivom obliku.
import java.util.*;
import static net.mindview.util.Print.*;

public class IspisivanjeKontejnera {
  static Collection popuni(Collection<String> kolekcija) {
    kolekcija.add("pacov");
    kolekcija.add("mačka");
    kolekcija.add("kuče");
    kolekcija.add("kuče");
    return kolekcija;
  }
  static Map popuni(Map mapa) {
    mapa.put("pacov", "Zbunjeni");
    mapa.put("mačka", "Mica");
    mapa.put("kuče", "Lesi");
```

```
    mapa.put("kuče", "Bobi");
    return mapa;
  }
  public static void main(String[] args) {
    print(popuni(new ArrayList<String> ()));
    print(popuni(new LinkedList<String> ()));
    print(popuni(new HashSet<String> ()));
    print(popuni(new TreeSet<String> ()));
    print(popuni(new LinkedHashSet<String> ()));
    print(popuni(new HashMap<String, String> ()));
    print(popuni(new TreeMap<String, String> ()));
    print(popuni(new LinkedHashMap<String, String> ()));
  }
} /* Ispis:
[pacov, mačka, kuče, kuče]
[pacov, mačka, kuče, kuče]
[kuče, mačka, pacov]
[pacov, mačka, kuče]
{kuče=Bobi,  mačka= Mica, pacov=Zbunjeni}
{mačka= Mica, kuče=Bobi,  pacov=Zbunjeni}
{pacov=Zbunjeni, mačka= Mica, kuče=Bobi}
*///:~
```

Ovim su prikazane dve osnovne kategorije u Javinoj biblioteci kontejnera. Razlikuju se po broju elemenata koji se čuvaju u svakoj lokaciji kontejnera. Kategorija **Collection** čuva samo jedan element u svakoj lokaciji. Ova kategorija obuhvata interfejse: **List** (lista), koji čuva grupu elemenata u zadatom redosledu; **Set** (skup), koji dozvoljava dodavanje samo jednog elementa iste vrednosti i **Queue** (red), koji umetanje objekata dozvoljava samo na jednom „kraju" kontejnera, a vađenje objekata samo na drugom „kraju" (za potrebe ovog primera, to je samo drugi način posmatranja sekvence, pa zato nije prikazan). **Map**a na svakoj lokaciji sadrži dva objekta, ključ i njemu pridruženu *vrednost*.

Iz rezultata programa vidite da podrazumevano ponašanje što se tiče ispisivanja (izlaz metode **toString**() svakog kontejnera) daje prilično čitljive rezultate. Vrednosti objekata koji se čuvaju u nekom od kontejnera iz kategorije **Collection** ispisuju se unutar uglastih zagrada, pri čemu su elementi međusobno razdvojeni zarezom. Vrednosti objekata koji se čuvaju u **Map**i ispisuju se unutar vitičastih zagrada, pri čemu su ključevi i njima pridružene vrednosti povezane znakom jednakosti (ključevi nalevo, vrednosti nadesno od tog znaka).

Prva metoda **popuni**() radi sa svim tipovima kategorije **Collection**, od kojih svaki realizuje metodu **add**() za dodavanje novih elemenata.

ArrayList i **LinkedList** su (očigledno) tipovi kategorije **List**, a iz rezultata programa vidite da oba čuvaju elemente u poretku umetanja. Razlika između njih nije samo u performansama za određene vrste operacija, nego **LinkedList** sadrži više operacija od tipa **ArrayList**. O njima ćemo više govoriti u nastavku poglavlja.

HashSet, **TreeSet** i **LinkedHashSet** tipovi su kategorije **Set**. Rezultat programa pokazuje da svaki skup (**Set**) može sadržati samo po jedan primerak identičnih elemenata, ali i da različite realizacije skupova elemente skladište na različite načine. Tip **HashSet** skladišti elemente na prilično komplikovan način koji će biti detaljnije opisan u poglavlju *Detaljno razmatranje kontejnera* – zasad je dovoljno znati da ta tehnika omogućava naj-

brže vadenje elemenata, iako redosled njegovog skladištenja izgleda besmislen (često je važno samo da li je objekat član određenog skupa iz kategorije **Set**, dok redosled elemenata u tom skupu nema značaja). Ako je poredak skladištenja važan, možete upotrebiti **TreeSet** koji objekte čuva u rastućem poretku poređenja, ili **LinkedHashSet** koji ih čuva u redosledu kojim su bili umetani.

Mapa (koja se naziva i *asocijativan niz*) omogućava pronalaženje objekta pomoću njegovog ključa, kao u jednostavnoj bazi podataka. Objekat pridružen ključu jeste njegova *vrednost*. Ukoliko imate **Map**u u kojoj su imena država pridruženi imenima njihovih glavnih gradova i želite da saznate koji je glavni grad Gruzije, pretraživanje obavljate pomoću ključa „Gruzija" – gotovo kao da je u pitanju indeks niza. Zbog takvog ponašanja **Map**a prihvata samo po jedan primerak svakog ključa.

Map.put(ključ, vrednost) umeće u mapu vrednost (ono što hoćete da uskladištite) i pridružuje je određenom ključu (pomoću kojeg ćete tu vrednost pronaći). **Map.get(ključ)** daje vrednost pridruženu datom ključu. U gornjem primeru samo su dodavani parovi ključ–vrednost, jer pretraživanja nije bilo. To će biti pokazano kasnije.

Imajte u vidu da ne morate zadati veličinu **Map**e (niti se za to uopšte starajte), jer ona svoju veličinu podešava automatski. Pored toga, **Map**e same umeju da ispišu svoj sadržaj, pri čemu se prikazuju parovi ključ–vrednost. U **Map**ama se ključevi i vrednosti ne čuvaju u onom poretku kojim su umetani, zato što realizacija tipa **HashMap** upotrebljava veoma brz algoritam koji određuje taj poredak.

U primeru su upotrebljena tri osnovna tipa **Map**a: **HashMap**, **TreeMap** i **LinkedHashMap**. Kao i **HashSet**, **HashMap** obezbeđuje najbržu tehniku pretraživanja; takođe, ni ta mapa ne čuva svoje elemente u lako razumljivom poretku. U mapi tipa **TreeMap** ključevi su poređani u rastućem poretku, a u mapi tipa **LinkedHashMap** u poretku umetanja, pri čemu je zadržana brzina pretraživanja tipa **HashMap**.

Vežba 4: (3) Napravite *generatorsku* klasu koja vraća imena (kao objekte tipa **String**) likova vašeg omiljenog filma (ili *Snežane i sedam patuljaka* ili *Ratova zvezda*, nije važno) svaki put kada pozovete **next()**, i vraća se na početak liste sa imenima likova kada dođe do njenog kraja. Upotrebite taj generator za popunjavanje običnog niza i po jednog primerka tipova **ArrayList**, **LinkedList**, **HashSet**, **LinkedHashSet** i **TreeSet**. Zatim ispišite sadržaj svakog od tih kontejnera.

Liste

Liste čuvaju elemente u određenom poretku. Interfejs **List** kategoriji **Collection** dodaje više metoda za umetanje i uklanjanje elemenata iz sredine **List**e.

Postoje dva tipa **List**e:

- Osnovni tip, **ArrayList**, izvrstan je za pristupanje elementima nasumičnim redosledom, ali je sporiji prilikom umetanja i uklanjanja elemenata iz sredine **List**e.
- Tip **LinkedList** obezbeđuje optimalan sekvencijalni pristup, a ni umetanje i uklanjanje elemenata iz sredine **List**e ne košta mnogo. **LinkedList** je relativno spor prilikom pristupanja elementima nasumičnim redosledom, ali su njegove mogućnosti veće nego one tipa **ArrayList**.

U narednom primeru uvešćemo biblioteku **typeinfo.pets**, opisanu u poglavlju *Podaci o tipu* u nastavku knjige. Ta biblioteka sadrži hijerarhiju klasa **Pet** (kućni ljubimac) i neke alatke za nasumično generisanje **Pet** objekata. Zasad ne morate znati sve pojedinosti, nego samo da (1) postoji klasa **Pet** i razni njeni podtipovi, i da (2) statična metoda **Pets.arrayList()** vraća **ArrayList** popunjen nasumično izabranim **Pet** objektima:

```
//: cuvanje/ObelezjaLista.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ObelezjaLista {
  public static void main(String[] args) {
    Random rand = new Random(47);
    List<Pet> pets = Pets.arrayList(7);
    print("1: " + pets);
    Hamster h = new Hamster();
    pets.add(h); // Automatski podešava svoju veličinu
    print("2: " + pets);
    print("3: " + pets.contains(h));
    pets.remove(h); // Uklanjanje na osnovu objekta
    Pet p = pets.get(2);
    print("4: " +  p + " " + pets.indexOf(p));
    Pet cymric = new Cymric();
    print("5: " + pets.indexOf(cymric));
    print("6: " + pets.remove(cymric));
    // Mora biti tačno određeni objekat:
    print("7: " + pets.remove(p));
    print("8: " + pets);
    pets.add(3, new Mouse()); // Umeće tamo gde pokazuje indeks
    print("9: " + pets);
    List<Pet> sub = pets.subList(1, 4);
    print("subList: " + sub);
    print("10: " + pets.containsAll(sub));
    Collections.sort(sub); // Sortiranje u mestu
    print("sortirana subList: " + sub);
    // Za metodu containsAll() poredak nije važan:
    print("11: " + pets.containsAll(sub));
    Collections.shuffle(sub, rand); // Izmešaj
    print("izmešana subList: " + sub);
    print("12: " + pets.containsAll(sub));
    List<Pet> kopija = new ArrayList<Pet>(pets);
    sub = Arrays.asList(pets.get(1), pets.get(4));
    print("sub: " + sub);
    kopija.retainAll(sub);
    print("13: " + kopija);
    kopija = new ArrayList<Pet>(pets); // Napravi novu kopiju
    kopija.remove(2); // Uklanjanje na osnovu indeksa
    print("14: " + kopija);
```

```
    kopija.removeAll(sub); // Uklanja samo objekte koji tačno odgovaraju
    print("15: " + kopija);
    kopija.set(1, new Mouse()); // Ukloni jedan elemen
    print("16: " + kopija);
    kopija.addAll(2, sub); // Umeće listu u sredinu
    print("17: " + kopija);
    print("18: " + pets.isEmpty());
    pets.clear(); // Ukloni sve elemente
    print("19: " + pets);
    print("20: " + pets.isEmpty());
    pets.addAll(Pets.arrayList(4));
    print("21: " + pets);
    Object[] o = pets.toArray();
    print("22: " + o[3]);
    Pet[] pa = pets.toArray(new Pet[0]);
    print("23: " + pa[3].id());
  }
} /* Ispis:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sortirana subList: [Manx, Mouse, Mutt]
11: true
izmešana subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~
```

Da biste svaki red rezultata mogli da dovedete u vezu sa izvornim kodom, u programu su naredbe print numerisane. Prvi red rezultata pokazuje prvobitnu **Listu** kućnih ljubimaca, tj. objekata tipa **Pet**. Za razliku od niza, **Lista** dozvoljava dodavanje i uklanjanje elemenata

i nakon što je napravljena, pri čemu automatski podešava svoju veličinu. To je njena osnovna vrednost: sekvenca koja se može modifikovati. Rezultat dodavanja jednog objekta tipa **Hamster** vidite u redu 2 – objekat je dodat na kraj liste.

Metodom **contains**() možete saznati da li je određeni element deo **Liste**. Ukoliko objekat hoćete da uklonite, prosledite njegovu referencu metodi **remove**(). Takođe, ako imate referencu nekog objekta, metodom **indexOf**() možete saznati njegov indeks (redni broj) u **Listi**, kao što se vidi iz reda 4 rezultata.

Metoda **equals**() (deo korenske klase **Object**) upotrebljava se za ispitivanje da li je određeni element deo **Liste**, te za pronalaženje indeksa elementa i uklanjanje elementa iz **Liste** na osnovu reference. Svaki objekat tipa **Pet** po definiciji je jedinstven; čak i ako u listi već postoje dva objekta tipa **Cymric**, ako napravim nov takav objekat i prosledim ga metodi **indexOf**(), njen rezultat će biti **-1** (što znači da taj novi nije pronađen), pa će i pokušaji uklanjanja tog objekta metodom **remove**() vratiti **false** (što znači da uklanjanje nije uspelo). Za druge klase, **equals**() se može drugačije definisati. Primera radi, objekti tipa **String** (znakovni nizovi) jednaki su ukoliko je sadržaj oba znakovna niza identičan. Dakle, da ne bi bilo iznenađenja, povedite računa o tome da se ponašanje **Lista** menja u zavisnosti od metode **equals**().

Iz redova 7 i 8 rezultata vidi se da je uspelo uklanjanje objekta koji tačno odgovara jednom objektu u **Listi**.

Element je moguće umetnuti u sredinu **Liste**, kao što vidite iz reda 9 rezultata i koda koji mu prethodi, ali iz toga sledi zaključak: za tip **LinkedList**, umetanje i uklanjanje iz sredine liste jeftina je operacija (sem u ovom slučaju, kada se u sredinu liste pristupa nasumično), ali za **ArrayList** to je skupa operacija. Da li to znači da nikada ne treba umetati elemente u sredinu **ArrayListe** i da treba preći na **LinkedList** čim takvo nešto zatreba? Ne – o tome prosto treba da vodite računa, pa ako napravite mnogo umetanja u sredinu **ArrayListe** *i* ako program počne da usporava, možda je za to kriva vaša realizacija **Liste** (takva uska grla se najbolje pronalaze profajlerom, kao što ćete videti u dodatku na lokaciji *http://MindView.net/Books/BetterJava*). Optimizacija je složen posao i najbolje je ne baviti se njom dok vas program na to ne natera (mada nije loše znati o čemu se radi).

Metoda **subList**() služi za pravljenje isečka veće liste, pa je prirodno da se dobija rezultat **true** kada se isečak prosledi metodi **containsAll**() za tu veću listu. Zanimljivo je da pri tom poredak nije važan – u redovima 11 i 12 rezultata vidite da na rezultat metode **containsAll**() ne utiče pozivanje metoda **Collections.sort**() (za sortiranje) i **Collections.shuffle**() (za mešanje elemenata) na podlisti **sub**. Metoda **subList**() proizvodi listu povezanu sa originalnom listom. Zato se izmene u vraćenoj listi odražavaju na originalnoj, i obrnuto.

Metoda **retainAll**() zapravo sprovodi operaciju „preseka skupova". U ovom slučaju, ona u objektu **kopija** zadržava sve elemente koji su i u podlisti **sub**. I opet, rezultujuće ponašanje se menja u zavisnosti od metode **equals**().

Četrnaesti red ispisa prikazuje rezultat uklanjanja elementa na osnovu njegovog indeksa, što je jednostavnije nego uklanjanje na osnovu reference objekta, pošto vas indeksi oslobađaju brige o ponašanju metode **equals**().

I ponašanje metode **removeAll**() zavisi od metode **equals**(). Kao što joj i englesko ime kaže, ona iz **Liste** uklanja sve objekte koji su joj prosleđeni u argumentu **sub**.

Metoda **set()** je loše nazvana, jer ju je moguće pomešati s klasom **Set** – tu bi bolji naziv bio „replace" (zameni), pošto ona element datog indeksa (prvi argument) zamenjuje drugim argumentom.

Sedamnaesti red ispisa prikazuje da za **Liste** postoji preklopljena metoda **addAll()** koja služi za umetanje nove liste usred prvobitne, umesto običnog dodavanja na kraj prvobitne onom metodom **addAll()** koja je deo klase **Collection**.

Redovi ispisa 18–20 pokazuju rezultate metoda **isEmpty()** i **clear()**.

Redovi ispisa 22 i 23 pokazuju kako svaki kontejner (objekat tipa **Collection**) metodom **toArray()** možete pretvoriti u niz. Ta metoda je preklopljena; verzija koja ne prima argumente vraća niz elemenata tipa **Object**, ali ako preklopljenoj verziji prosledite niz ciljnog tipa, ona će napraviti niz tog tipa (ukoliko on zadovolji proveru tipova). Ako je niz prosleđen kao argument premali za skladištenje svih objekata u **List**i (kao što je ovde slučaj), metoda **toArray()** će napraviti nov niz odgovarajuće veličine. **Pet** objekti imaju metodu **id()** – vidite da je pozvana za jedan od objekata u rezultujućem nizu.

Vežba 5: (3) Prepravite program **ListFeatures.java** tako da umesto objekata tipa **Pet** upotrebljava cele brojeve (setite se automatskog pakovanja!) i objasnite sve razlike u rezultatima.

Vežba 6: (2) Prepravite program **ListFeatures.java** tako da umesto objekata tipa **Pet** upotrebljava znakovne nizove (objekt tipa **String**) i objasnite sve razlike u rezultatima.

Vežba 7: (3) Napravite klasu, a zatim inicijalizovan niz objekata te klase. Popunite tim nizom jednu **List**u. Napravite podskup liste metodom **subList( )**, a zatim ga uklonite iz nje.

Iteratori

U svakoj kontejnerskoj klasi mora postojati način da se elementi dodaju i izvade. Osnovni zadatak kontejnera i jeste da nešto sadrži. U klasi **List**, objekti se umeću metodom **add( )**, a metoda **get( )** je jedan od načina da se pročitaju.

Ako počnete da razmišljate na višem nivou o kontejnerskoj biblioteci, pojavljuje se jedan nedostatak: morate tačno znati tip kontejnera da biste ga koristili. To na prvi pogled ne izgleda loše, ali šta ako napišete kod za **List**u, a kasnije otkrijete da bi isti kôd trebalo da primenite na skup (**Set**)? Ili, primera radi, želite da napišete generički kôd koji ne zna (ili mu nije bitno) s kojim tipom kontejnera radi, kako biste mogli da ga koristite s različitim tipovima kontejnera a da ga pri tom ne pišete ponovo?

Ovakva apstrakcija se postiže korišćenjem *iteratora* (i to je projektni obrazac). Iterator je objekat čiji je zadatak da se pomera kroz niz objekata i da izabere svaki objekat u tom nizu, pri čemu programer klijent ne zna (ili mu nije važno) kakva je struktura tog niza. Pored toga, iterator se obično naziva lak objekat (engl. *light-weight*), jer koristi malo resursa. Zbog toga ćete za iteratore često vidati naizgled čudna ograničenja; na primer, Javin **Iterator** može da se kreće samo u jednom smeru. Sa iteratorom ne možete bogzna šta da radite, osim da:

1. Zatražite od kontejnera da vam metodom **iterator( )** prosledi **Iterator**. Dobijeni **Iterator** je spreman da vrati prvi element niza posle prvog poziva njegove metode **next( )**.
2. Dobijete sledeći objekat u nizu pomoću metode **next( )**.

3. Ustanovite da li ima *još* objekata u nizu pomoću metode **hasNext( )**.
4. Uklonite poslednji element koji je iterator vratio pomoću metode **remove( )**.

Da bismo videli kako on radi, opet ćemo upotrebiti klasu **Pet** i njene metode, opisane u poglavlju *Podaci o tipu*:

```
//: cuvanje/JednostavnaIteracija.java
import typeinfo.pets.*;
import java.util.*;

public class JednostavnaIteracija {
  public static void main(String[] args) {
    List<Pet> ljubimci = Pets.arrayList(12);
    Iterator<Pet> it = ljubimci.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // Jednostavniji pristup, kada je moguć:
    for(Pet p : ljubimci)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // Iterator može i da uklanja elemente:
    it = ljubimci.iterator();
    for(int i = 0; i < 6; i++) {
      it.next();
      it.remove();
    }
    System.out.println(ljubimci);
  }
} /* Ispis:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat
10:EgyptianMau 11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat
10:EgyptianMau 11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*///:~
```

Uz iterator ne morate da brinete o broju elemenata u kontejneru. Umesto vas, o tome vode računa metode **hasNext( )** i **next( )**.

Vidite da je foreach sintaksa prikladnija za jednostavno kretanje unapred kroz **List**u, ukoliko ne pokušavate da modifikujete sam **List** objekat.

Iterator takođe uklanja poslednji element proizveden metodom **next()**, što znači da **next()** morate pozvati pre nego što pozovete **remove()**.[4]

[4] **remove()** spada u takozvane opcione metode (ima ih još), što znači da je ne moraju realizovati sve realizacije **Iterator**a. O tome govorimo u poglavlju *Detaljno razmatranje kontejnera*. Kako svi standardni kontejneri Java biblioteke realizuju **remove()**, o tome ne morate da brinete (dok ne dođete do spomenutog poglavlja).

Korišćenje kontejnera i kretanje kroz njega da bi se izvršavale operacije nad svakim elementom veoma je moćna tehnika i zapazićete da se često primenjuje u ovoj knjizi.

Pogledajmo sada kako izgleda metoda **ispisiSve**() koja radi sa svim vrstama kontejnera:

```
//: cuvanje/IteracijaKrozSveVrsteKontejnera.java
import typeinfo.pets.*;
import java.util.*;

public class IteracijaKrozSveVrsteKontejnera {
  public static void ispisiSve(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    ArrayList<Pet> ljubimci = Pets.arrayList(8);
    LinkedList<Pet> ljubimciLL = new LinkedList<Pet>(ljubimci);
    HashSet<Pet> ljubimciHS = new HashSet<Pet>(ljubimci);
    TreeSet<Pet> ljubimciTS = new TreeSet<Pet>(ljubimci);
    ispisiSve(ljubimci.iterator());
    ispisiSve(ljubimciLL.iterator());
    ispisiSve(ljubimciHS.iterator());
    ispisiSve(ljubimciTS.iterator());
  }
} /* Ispis:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
*///:-
```

Uočili ste da metoda **ispisiSve**() ništa ne zna o vrsti sekvence kroz koju prolazi, a to pokazuje pravu snagu **Iterator**a: mogućnost da razdvoji operaciju prolaska kroz sekvencu od njene strukture. Zato se katkad kaže da iteratori *objedinjuju pristup kontejnerima*.

Vežba 8: (1) Prepravite vežbu 1 tako da se za kretanje kroz **List**u upotrebljava **Iterator** kada se pozove **skoci**().

Vežba 9: (4) Prepravite program **unutrasnjeklase/Sekvenca.java** tako da **Sekvenca** radi sa **Iterator**om umesto sa **Selector**om.

Vežba 10: (2) Prepravite vežbu 9 iz poglavlja *Polimorfizam* tako da se **Glodar**i čuvaju u **ArrayList**i i da se za kretanje kroz niz **Glodar**a upotrebljava **Iterator**.

Vežba 11: (2) Napišite metodu koja upotrebljava **Iterator** za prolazak kroz kontejner (objekat tipa **Collection**). Metodom **toString**() ispišite sadržaj svih objekata u kontejneru. Popunite objektima sve vrste kontejnera i na svaki od njih primenite svoju metodu.

ListIterator

Za liste postoji napredniji iterator, **ListIterator**. Dok se **Iterator** može pomerati samo unapred, **ListIterator** je dvosmeran. Takođe, ume da proizvede indekse sledećeg i prethodnog elementa u odnosu na mesto na koje iterator pokazuje u listi, kao i da metodom **set()** zameni poslednji element koji je posetio. Pozivom metode **listIterator()** napravićete **ListIterator** koji pokazuje na početak **Liste**, a zadavanjem argumenta **n** (kao u **listIterator(n)**) napravićete **ListIterator** koji počinje pokazujući na indeks **n** u listi. Evo primera u kojem su pokazane sve navedene mogućnosti:

```
//: cuvanje/IteracijaListe.java
import typeinfo.pets.*;
import java.util.*;

public class IteracijaListe {
  public static void main(String[] args) {
    List<Pet> ljubimci = Pets.arrayList(8);
    ListIterator<Pet> it = ljubimci.listIterator();
    while(it.hasNext())
      System.out.print(it.next() + ", " + it.nextIndex() +
        ", " + it.previousIndex() + "; ");
    System.out.println();
    // Unazad:
    while(it.hasPrevious())
      System.out.print(it.previous().id() + " ");
    System.out.println();
    System.out.println(ljubimci);
    it = ljubimci.listIterator(3);
    while(it.hasNext()) {
      it.next();
      it.set(Pets.randomPet());
    }
    System.out.println(ljubimci);
  }
} /* Ispis:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4; Cymric, 6,
5;
Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster, EgyptianMau]
*///:~
```

Metodom **Pets.randomPet()** zamenjeni su svi objekti tipa **Pet** u **Listi** od lokacije 3 unapred.

Vežba 12: (3) Napravite i popunite jednu celobrojnu listu (**List<Integer>**). Napravite drugu celobrojnu listu iste veličine i upotrebite **ListIteratore** za čitanje elemenata iz prve **Liste** i umetanje u drugu obrnutim redosledom. (Ovaj zadatak bi trebalo da rešite na više načina.)

Ulančana lista

I klasa **LinkedList** realizuje osnovni interfejs **List** kao **ArrayList**, ali određene operacije (umetanje u sredinu **Liste** i uklanjanje iz nje) obavlja efikasnije nego **ArrayList**. S druge strane, manje je efikasna za operacije pristupanja nesekvencijalnim redosledom.

LinkedList dodaje i metode koje omogućavaju da je upotrebimo kao stek, kao red za čekanje (**Queue**) ili dvostrani red za čekanje (engl. *double-ended queue, deque*).

Neke od ovih metoda su pseudonimi (alijasi) ili međusobne neznatne varijacije, da bi se dobila imena poznata u kontekstu određene upotrebe (naročito u redovima čekanja). Na primer, **getFirst**() i **element**() identične su – one vraćaju čelo (prvi element) liste, a da ga ne uklone, i generišu izuzetak **NoSuchElementException** ako je **Lista** prazna. Metoda **peek**() je njihova varijacija koja vraća **null** ako je lista prazna.

Identične su i metode **removeFirst**() i **remove**() – one uklanjaju i vraćaju kao rezultat čelo liste, i generišu izuzetak **NoSuchElementException** ako je lista prazna, dok njihova varijacija **poll**() u tom slučaju vraća **null**.

addFirst() umeće element na početak liste.

offer() je ista kao **add**() i **addLast**(). Sve one dodaju element na rep (kraj) liste.

removeLast() uklanja i vraća poslednji element liste.

Naredni primer pokazuje osnovne sličnosti spomenutih metoda i razlike između njih. U njemu se ne ponavlja ponašanje prikazano u programu **MogucnostiListe.java**:

```
//: cuvanje/MetodeKlaseLinkedList.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MetodeKlaseLinkedList {
  public static void main(String[] args) {
    LinkedList<Pet> ljubimci =
      new LinkedList<Pet>(Pets.arrayList(5));
    print(ljubimci);
    // Identične:
    print("ljubimci.getFirst(): " + ljubimci.getFirst());
    print("ljubimci.element(): " + ljubimci.element());
    // Razlike samo u reakciji na praznu listu:
    print("ljubimci.peek(): " + ljubimci.peek());
    // Identične; uklanjaju i vraćaju prvi element:
    print("ljubimci.remove(): " + ljubimci.remove());
    print("ljubimci.removeFirst(): " + ljubimci.removeFirst());
    // Razlike samo u reakciji na praznu listu:
    print("ljubimci.poll(): " + ljubimci.poll());
    print(ljubimci);
    ljubimci.addFirst(new Rat());
    print("Posle addFirst(): " + ljubimci);
    ljubimci.offer(Pets.randomPet());
    print("Posle offer(): " + ljubimci);
```

```
    ljubimci.add(Pets.randomPet());
    print("Posle add(): " + ljubimci);
    ljubimci.addLast(new Hamster());
    print("Posle addLast(): " + ljubimci);
    print("ljubimci.removeLast(): " + ljubimci.removeLast());
  }
} /* Ispis:
[Rat, Manx, Cymric, Mutt, Pug]
ljubimci.getFirst(): Rat
ljubimci.element(): Rat
ljubimci.peek(): Rat
ljubimci.remove(): Rat
ljubimci.removeFirst(): Manx
ljubimci.poll(): Cymric
[Mutt, Pug]
Posle addFirst(): [Rat, Mutt, Pug]
Posle offer(): [Rat, Mutt, Pug, Cymric]
Posle add(): [Rat, Mutt, Pug, Cymric, Pug]
Posle addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
ljubimci.removeLast(): Hamster
*///:~
```

Rezultat metode **Pets.arrayList()** prosleđujemo konstruktoru liste **LinkedList** da bismo je popunili. U dokumentaciji o interfejsu **Queue** videćete metode **element()**, **offer()**, **peek()**, **poll()** i **remove()** koje su klasi **LinkedList** dodate da bi ta klasa mogla da bude realizacija reda čekanja. Kompletni primeri redova čekanja dati su u nastavku poglavlja.

Vežba 13: (3) U primeru **unutrasnjeklase/UpravljacStaklenika.java**, klasa **Upravljac** upotrebljava listu **ArrayList**. Iskoristite umesto nje listu **LinkedList**, a za kruženje kroz skup događaja primenite **Iterator**.

Vežba 14: (3) Napravite praznu listu **LinkedList<Integer>**. U sredinu **List**e dodajte **Integer**e pomoću **ListIterator**a.

Pravljenje steka od ulančane liste

Stek se ponekad naziva LIFO kontejner (engl. *last-in, first-out*, poslednji koji ude, prvi izlazi). To znači da ono što poslednje stavite na stek (za to se tradicionalno koristi operacija *push*) prvo možete da skinete s njega (za to se tradicionalno koristi operacija *pop*). Stek se često poredi sa stogom – poslednji naviljak sena koji bacite na vrh stoga, prvi ćete skinuti s njega.

Klasa **LinkedList** ima metode koje direktno realizuju stek, pa umesto da pravite posebnu klasu steka, možete upotrebiti ulančanu listu. U nekim slučajevima ipak je bolje napraviti klasu steka:

```
//: net/mindview/util/Stack.java
// Pravljenje steka od objekta tipa LinkedList.
package net.mindview.util;
import java.util.LinkedList;
```

```
public class Stack<T> {
  private LinkedList<T> skladiste = new LinkedList<T>();
  public void push(T v) { skladiste.addFirst(v); }
  public T peek() { return skladiste.getFirst(); }
  public T pop() { return skladiste.removeFirst(); }
  public boolean empty() { return skladiste.isEmpty(); }
  public String toString() { return skladiste.toString(); }
} ///:~
```

Ovim smo dali najjednostavniji primer definicije generičke klase. Ono <T> nakon imena klase, kazuje prevodiocu da će ovo biti *parametrizovani tip*, i da je parametar tipa – koji će biti zamenjen stvarnim tipom kada klasa bude upotrebljena – ono **T**. U suštini, to kazuje „definišemo **Stack** koji čuva objekte tipa **T**". Taj **Stack** se realizuje ulančanom listom kojoj je takođe rečeno da čuva tip **T**. Obratite pažnju na to da **push( )** uzima objekat tipa **T** koji vraćaju **peek( )** i **pop( )**. Metoda **peek( )** vraća najviši element a da ga ne skida s vrha steka, dok ga **pop( )** uklanja i vraća.

Ako želite da se ograničite samo na funkcionalnost steka, nasleđivanje nije prikladno, zato što bi dalo klasu sa svim ostalim metodama ulančane liste (videćete u poglavlju *Detaljno razmatranje kontejnera* da su takvu grešku napravili projektanti biblioteke Java 1.0 u klasi **java.util.Stack**).

Evo jednostavnog primera ove nove klase **Stack**:

```
//: cuvanje/StekTest.java
import net.mindview.util.*;

public class StekTest {
  public static void main(String[] args) {
    Stack<String> stek = new Stack<String>();
    for(String s : "Moj pas ima buve".split(" "))
      stek.push(s);
    while(!stek.empty())
      System.out.print(stek.pop() + " ");
  }
} /* Ispis:
buve ima pas Moj
*///:~
```

Ukoliko u svom kodu hoćete da upotrebite ovu klasu **Stack**, moraćete potpuno da specifičirate njen paket – ili promenite ime klase – kada je budete pravili; u protivnom, verovatno ćete izazvati sukob s klasom **Stack** iz paketa **java.util**. Primera radi, ukoliko u gornji listing uvezemo **java.util.***, moraćemo da upotrebljavamo imena paketa kako bismo sprečili dvosmislenost imena i sukobe:

```
//: cuvanje/StekSukob.java
import net.mindview.util.*;

public class StekSukob {
  public static void main(String[] args) {
    net.mindview.util.Stack<String> stek =
```

```
      new net.mindview.util.Stack<String>();
    for(String s : "Moj pas ima buve".split(" "))
      stek.push(s);
    while(!stek.empty())
      System.out.print(stek.pop() + " ");
    System.out.println();
    java.util.Stack<String> stek2 =
      new java.util.Stack<String>();
    for(String s : "Moj pas ima buve".split(" "))
      stek2.push(s);
    while(!stek2.empty())
      System.out.print(stek2.pop() + " ");
  }
} /* Ispis:
buve ima pas Moj
buve ima pas Moj
*///:~
```

Obe klase **Stack** imaju isti interfejs, ali u paketu **java.util** ne postoji zajednički interfejs **Stack** – verovatno zato što je originalna, loše projektovana klasa **java.util.Stack** u Javi 1.0 usvojila to ime. Iako **java.util.Stack** postoji, **LinkedList** pravi bolji stek, pa prednost treba dati pristupu ostvarenom u paketu **net.mindview.util.Stack**.

Izbor realizacije klase **Stack** „kojoj treba dati prednost" možete kontrolisati eksplicitnim uvozom:

```
import net.mindview.util.Stack;
```

Sada će svako upućivanje na klasu **Stack** izabrati verziju **net.mindview.util**, a da biste izabrali **java.util.Stack**, morate upotrebiti potpuno ime koje obuhvata i ime paketa.

Vežba 15: (4) U programskim jezicima stekovi se često upotrebljavaju za izračunavanje vrednosti izraza. Izračunajte sledeći izraz koristeći paket **net.mindview.util.Stack**, gde + znači „stavite na stek sledeće slovo", a - znači „skinite vrh steka i ispišite ga":
„+U+n+c---+e+r+t---+a-+i-+n+t+y---+ -+r+u--+l+e+s---"

Funkcionalnost skupa

Skup (**Set**) ne može da sadrži više od jedne instance vrednosti objekta. Ukoliko pokušate da dodate više od jedne instance ekvivalentnog objekta, **Set** će sprečiti dupliranje. **Set** se najčešće upotrebljava za utvrđivanje pripadnosti, pa je lako saznati da li je određeni objekat u određenom skupu. Zato je za **Set** pretraživanje obično najvažnija operacija, pa se najčešće bira realizacija **HashSet**, optimizovana za brzo pretraživanje.

Set (skup) ima isti interfejs kao **Collection**, što znači da nema dodatnih funkcija kao što imaju dve različite liste. Skup je isti kao kolekcija, samo se različito ponaša. (Ovo je primer savršenog korišćenja nasleđivanja i polimorfizma: za izražavanje drugačijeg ponašanja.) **Set** utvrđuje pripadnost na osnovu „vrednosti" objekta. Objašnjenje od čega se sastoji vrednost objekta složeno je, kao što ćete videti u poglavlju *Detaljno razmatranje kontejnera.*

Sledeći primer pokazuje ponašanje skupa **HashSet** sa celobrojnim (**Integer**) objektima:

```
//: cuvanje/SkupCelihBrojeva.java
import java.util.*;

public class SkupCelihBrojeva {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    Set<Integer> skupcelihb = new HashSet<Integer>();
    for(int i = 0; i < 10000; i++)
       skupcelihb.add(slucajan.nextInt(30));
    System.out.println(skupcelihb);
  }
} /* Ispis:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26,
11, 18, 3, 12, 27, 17, 2, 13, 28, 20, 25, 10, 5, 0]
*///:~
```

U **skup** je dodato deset hiljada slučajnih brojeva između 0 i 29, pa je lako zamisliti kako svaki od tih brojeva ima mnogo duplikata. Pa ipak, prilikom ispisa se vidi da **skup** sadrži samo po jednu instancu određene vrednosti.

Primetili ste da brojevi u rezultatu nemaju smisleni poredak. Uzrok tome je transformisanje ključa (engl. *hashing*), koje **HashSet** upotrebljava radi veće brzine – ono je objašnjeno u poglavlju *Detaljno razmatranje kontejnera*. Redosled koji održava **HashSet** razlikuje se od redosleda u **TreeSet** ili **LinkedHashSet**, pošto svaka realizacija skladišti elemente na različit način. **TreeSet** ih čuva uređene u crveno-crnoj strukturi stabla, dok **HashSet** primenjuje transformisanje ključa, projektovano specijalno radi brzog pronalaženja. I **LinkedHashSet** upotrebljava je u crveno-crnoj strukturi stabla, dok **HashSet** primenjuje transformisanje ključa radi brzog pronalaženja, ali *naizgled* pomoću ulančane liste održava elemente u poretku umetanja.

Ako hoćete da rezultati budu uređeni, možete upotrebiti **TreeSet** umesto **HashSet**a:

```
//: cuvanje/UredjenSkupCelihBrojeva.java
import java.util.*;

public class UredjenSkupCelihBrojeva {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    SortedSet<Integer> skupcelihb = new TreeSet<Integer>();
    for(int i = 0; i < 10000; i++)
      skupcelihb.add(slucajan.nextInt(30));
    System.out.println(skupcelihb);
  }
} /* Ispis:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Među najčešćim operacijama koje ćete obavljati jeste utvrđivanje pripadnosti skupu pomoću metode **contains()**, ali postoje i operacije koje će vas podsetiti na Venove dijagrame koje ste možda učili u osnovnoj školi:

```
//: cuvanje/OperacijeSaSkupovima.java
import java.util.*;
import static net.mindview.util.Print.*;

public class OperacijeSaSkupovima {
  public static void main(String[] args) {
    Set<String> skup1 = new HashSet<String>();
    Collections.addAll(skup1,
      "A B C D E F G H I J K L".split(" "));
    skup1.add("M");
    print("H: " + skup1.contains("H"));
    print("N: " + skup1.contains("N"));
    Set<String> skup2 = new HashSet<String>();
    Collections.addAll(skup2, "H I J K L".split(" "));
    print("skup2 deo skupa1: " + skup1.containsAll(skup2));
    skup1.remove("H");
    print("skup1: " + skup1);
    print("skup2 deo skupa1: " + skup1.containsAll(skup2));
    skup1.removeAll(skup2);
    print("skup2 uklonjen iz skupa1: " + skup1);
    Collections.addAll(skup1, "X Y Z".split(" "));
    print("'X Y Z' dodati skupu1: " + skup1);
  }
} /* Ispis:
H: true
N: false
skup2 deo skupa1: true
skup1: [D, K, C, B, L, G, I, M, A, F, J, E]
skup2 deo skupa1: false
skup2 uklonjen iz skupa1: [D, C, B, G, M, A, F, E]
'X Y Z' dodati skupu1: [Z, D, C, B, G, M, A, F, Y, X, E]
*///:~
```

Imena metoda lako će razumeti oni koji znaju engleski. Ima ih još nekoliko i njih ćete naći u dokumentaciji JDK-a.

Lista jedinstvenih elemenata ume dobro da posluži. Na primer, recimo kako hoćete da ispišete sve reči datoteke **OperacijeSaSkupovima.java**. Za otvaranje i učitavanje datoteke u **skup** upotrebićemo uslužni program **net.mindview.TextFile** (predstavićemo ga u nastavku knjige):

```
//: cuvanje/JedinstveneReci.java
import java.util.*;
import net.mindview.util.*;
```

```
public class JedinstveneReci {
  public static void main(String[] args) {
    Set<String> reci = new TreeSet<String>(
      new TextFile("OperacijeSaSkupovima.java", "\W+"));
    System.out.println(reci);
  }
} /* Ispis:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L, M, N,
OperacijeSaSkupovima, Ispis, Print, Set, String, X, Y, Z, add, addAll,
args, class, contains, containsAll, cuvanje, deo, dodati, false, import,
iz, java, main, mindview, net, new, print, public, remove, removeAll,
skup1, skupa1, skupu1, skup2, split, static, true, uklonjen, util, void]
*///:~
```

TextFile je nasleđen iz klase **List<String>**. Konstruktor **TextFile** otvara datoteku i deli je na reči u skladu s *regularnim izrazom* \W+, koji znači „jedno ili više slova" (regularni izrazi su objašnjeni u poglavlju *Znakovni nizovi*). Rezultat se predaje konstruktoru skupa **TreeSet** koji sadržaj **Liste** dodaje sebi. Pošto je u pitanju **TreeSet**, rezultat je uređen. U ovom slučaju, uređivanje se obavlja *leksikografski*, pa su velika i mala slova u zasebnim grupama. Ukoliko biste hteli *abecedno* uređivanje, konstruktoru skupa **TreeSet** prosledite **String.CASE_INSENSITIVE_ORDER Comparator** (*comparator* je objekat koji uspostavlja poredak):

```
//: cuvanje/JedinstveneReciAbecedno.java
// Abecedno ispisivanje.
import java.util.*;
import net.mindview.util.*;

public class JedinstveneReciAbecedno {
  public static void main(String[] args) {
    Set<String> reci =
      new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
    reci.addAll(
      new TextFile("OperacijeSaSkupovima.java", "\W+"));
    System.out.println(reci);
  }
} /* Ispis:
[A, add, addAll, args, B, C, class, Collections, contains, containsAll,
cuvanje, D, dodati, E, F, false, G, H, HashSet, I, import, iz, J, java,
K, L, M, main, mindview, N, net, new, OperacijeSaSkupovima, Ispis,
Print, public, remove, removeAll, Set, skup1, skupa1, skupu1, skup2,
split, static, String, true, uklonjen, util, void, X, Y, Z]
*///:~
```

Comparatori će biti detaljno razmotreni u poglavlju *Nizovi*.

Vežba 16: (5) Napravite skup (**Set**) samoglasnika. Prepravite program **JedinstveneReci.java** tako da prebroji i ispiše broj samoglasnika u svakoj ulaznoj reči, kao i ukupan broj samoglasnika u ulaznoj datoteci.

Funkcionalnost mape

Mapiranje jednih objekata na druge može biti izuzetno moćan način rešavanja programerskih zadataka. Primera radi, uzmimo program za ispitivanje slučajnosti brojeva iz Javine klase **Random**. U idealnom slučaju, **Random** bi davala savršeno raspodeljene slučajne brojeve, ali da biste to ispitali, morate da generišete više slučajnih brojeva i da izbrojite one koji pripadaju različitim opsezima. **Map**a rešava taj zadatak s lakoćom; u ovom slučaju, ključ je broj koji proizvodi klasa **Random**, a vrednost je broj puta koliko se taj broj pojavio:

```
//: cuvanje/StatistickiPodaci.java
// Jednostavan prikaz rada HashMape.
import java.util.*;

public class StatistickiPodaci {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    Map<Integer,Integer> m =
      new HashMap<Integer,Integer>();
    for(int i = 0; i < 10000; i++) {
      // Pravljenje broja između 0 i 20:
      int s = slucajan.nextInt(20);
      Integer ucestanost = m.get(s);
      m.put(s, ucestanost == null ? 1 : ucestanost + 1);
    }
    System.out.println(m);
  }
} /* Ispis:
{15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478, 3=508, 7=471,
12=521, 17=509, 2=489, 13=506, 9=549, 6=519, 1=502, 14=477, 10=513,
5=503, 0=481}
*///:~
```

U metodi **main**(), automatsko pakovanje pretvara slučajno generisan **int** u referencu objekta tipa **Integer** koja se može staviti u mapu tipa **HashMap**. (U kontejner ne možete da stavite vrednost prostog tipa, već samo referencu na objekat.) Metoda **get**() vraća **null** ukoliko takvog ključa nije bilo u kontejneru (što znači da je taj broj prvi put pronađen). U protivnom, metoda vraća pridruženu **Integer** vrednost ključa koja se povećava za jedan (automatsko pakovanje ponovo pojednostavljuje izraz, ali se obavlja pretvaranje u **Integer** i iz njega).

Evo primera u kojem se objekti tipa **Pet** pronalaze pomoću tekstualnog opisa, tj. objekata tipa **String**. Pokazano je i kako se metodama **containsKey**() i **containsValue**() ispituje da li **Map**a sadrži određeni ključ, odnosno vrednost:

```
//: cuvanje/Mapaljubimaca.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class Mapaljubimaca {
  public static void main(String[] args) {
    Map<String,Pet> mapaljubimaca = new HashMap<String,Pet>();
    mapaljubimaca.put("Moja mačka", new Cat("Mica"));
    mapaljubimaca.put("Moj pas", new Dog("Bruno"));
    mapaljubimaca.put("Moj hrčak", new Hamster("Boško"));
    print(mapaljubimaca);
    Pet pas = mapaljubimaca.get("Moj pas");
    print(pas);
    print(mapaljubimaca.containsKey("Moj pas"));
    print(mapaljubimaca.containsValue(pas));
  }
} /* Ispis:
{Moja mačka=Cat Mica, Moj hrčak=Hamster Boško, Moj pas=Dog Bruno}
Dog Bruno
true
true
*///:~
```

Mape je lako proširiti na više dimenzija, kao i nizove i kolekcije; samo napravite **Map**u čije su vrednosti druge **Map**e (a vrednosti *tih* **Map**a mogu biti drugi kontejneri, čak i druge **Map**e). Dakle, moćne strukture podataka prave se veoma lako i brzo, pomoću kombinovanja kontejnera. Na primer, pretpostavimo da program treba da prati osobe koje imaju više kućnih ljubimaca – samo vam treba **Map<Person, List<Pet>>**:

```
//: cuvanje/MapaLista.java
package cuvanje;
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapaLista {
  public static Map<Person, List<? extends Pet>>
    ljudiSljubimcima = new HashMap<Person, List<? extends Pet>>();
  static {
    ljudiSljubimcima.put(new Person("Zora"),
      Arrays.asList(new Cymric("Moli"),new Mutt("Žarko")));
    ljudiSljubimcima.put(new Person("Kata"),
      Arrays.asList(new Cat("Željka"),
        new Cat("Elza"), new Dog("Margareta")));
    ljudiSljubimcima.put(new Person("Mirna"),
      Arrays.asList(
       new Pug("Lui ili Luis Disalić Meh"),
       new Cat("Stane ili Smradić el Negro"),
       new Cat("Pinkola")));
    ljudiSljubimcima.put(new Person("Luka"),
      Arrays.asList(new Rat("Fazi"), new Rat("Fizi")));
    ljudiSljubimcima.put(new Person("Isak"),
      Arrays.asList(new Rat("Pegavi")));
  }
```

```
  public static void main(String[] args) {
    print("ljudi: " + ljudiSljubimcima.keySet());
    print("ljubimci: " + ljudiSljubimcima.values());
    for(Person osoba : ljudiSljubimcima.keySet()) {
      print(osoba + " ima:");
      for(Pet ljubimac : ljudiSljubimcima.get(person))
        print("    " + ljubimac);
    }
  }
} /* Ispis:
ljudi: [Person Luka, Person Mirna, Person Isak, Person Zora, Person Kata]
ljubimci: [[Rat Fazi, Rat Fizi], [Pug Lui ili Luis Disalić Meh, Cat Stane
ili Smradić el Negro, Cat Pinkola], [Rat Pegavi], [Cymric Moli, Mutt Žarko],
[Cat Željka, Cat Elza, Dog Margareta]]
Person Luka ima:
    Rat Fazi
    Rat Fizi
Person Mirna ima:
    Pug Lui ili Luis Disalić Meh
    Cat Stane ili Smradić el Negro
    Cat Pinkola
Person Isak ima:
    Rat Pegavi
Person Zora ima:
    Cymric Moli
    Mutt Žarko
Person Kate ima:
    Cat Željka
    Cat Elza
    Dog Margareta
*///:-
```

Mapa može da vrati skup svojih ključeva, kolekciju svojih vrednosti ili skup svojih parova. Metoda **keySet**() pravi skup svih ključeva u mapi **ljudiSljubimcima**, koja se u foreach naredbi upotrebljava za iteriranje kroz **Map**u.

Vežba 17: (2) Klasu **MorskoPrase** iz vežbe 1 stavite u mapu tako da ime objekta klase **MorskoPrase** kao **String** bude ključ za objekat koji stavljate u tabelu. Uzmite **Iterator** za skup koji daje metoda **keySet**( ) i upotrebite ga za kretanje kroz mapu i pronalaženje objekta klase **MorskoPrase** za svaki ključ, potom ispišite svaki ključ i pozovite metodu **skace**( ) za svaki objekat.

Vežba 18: (3) Popunite mapu **HashMap** parovima ključ–vrednost. Ispišite rezultate tako da pokažete uređenje po heš kodu. Izvadite parove i uredite ih po ključu, a rezultat stavite u mapu **LinkedHashMap**. Pokažite da je redosled umetanja zadržan.

Vežba 19: (2) Ponovite prethodnu vežbu sa skupovima **HashSet** i **LinkedHashSet**.

Vežba 20: (3) Prepravite vežbu 16 tako da se broji učestalost pojave svakog samoglasnika.

Vežba 21: (3) Koristeći **Map<String,Integer>** i oblik programa **JedinstveneReci.java**, napišite program koji broji učestalost pojave reči u datoteci. Uredite rezultate metodom **Collections.sort()** kojoj je drugi argument **String.CASE_ INSENSITIVE_ORDER** (da bi se dobilo uređenje po abecedi) i prikažite rezultat.

Vežba 22: (5) Prepravite prethodnu vežbu tako da upotrebljava klasu koja sadrži **String** i polje brojača za skladištenje svih različitih reči, i **Set** tih objekata za održavanje liste reči.

Vežba 23: (4) Na osnovu programa **StatistickiPodaci.java** napišite program koji stalno izvršava test i proverava da li se u rezultatima neki broj pojavljuje češće od drugih.

Vežba 24: (2) Popunite mapu **LinkedHashMap** ključevima tipa **String** i objektima po svom izboru. Potom izdvojite parove, uredite ih po ključu i ponovo umetnite u **Map**u.

Vežba 25: (3) Napravite **Map<String,ArrayList<Integer>>**. Otvorite tekstualnu datoteku programom **net.mindview.TextFile** i učitajte je reč po reč (kao drugi argument konstruktora **TextFile** upotrebite \W+). Prebrojte reči tokom učitavanja i za svaku reč u datoteci, zapišite u listi **ArrayList<Integer>** koliko se puta pojavila – to je, efektivno, mesto u datoteci gde je ta reč pronađena.

Vežba 26: (4) Prepravite rezultujuću **Map**u iz prethodne vežbe tako da ponovo uspostavite poredak reči iz originalne datoteke.

Pravljenje reda za čekanje (od ulančane liste)

Red za čekanje (engl. *queue*) je FIFO kontejner (engl. *first-in, first-out* – prvi koji uđe, prvi izlazi). To znači da elemente stavljate s jednog kraja, a skidate s drugog, tj. oni izlaze redosledom kojim su stavljani. Redovi za čekanje se obično koriste za pouzdano prebacivanje objekata iz jednog dela programa u drugi. Kao što ćete videti u poglavlju *Paralelno izvršavanje*, redovi za čekanje su naročito važni u paralelnom programiranju, pošto bezbedno prebacuju objekte iz jednog posla (engl. *task*) u drugi.

LinkedList ima metode koje oponašaju red za čekanje i realizuje interfejs **Queue**, pa se može upotrebiti za realizovanje reda za čekanje. Pošto je **LinkedList**a svedena naviše na **Queue**, u ovom primeru upotrebljene su metode specifične za red za čekanje u interfejsu **Queue**:

```
//: cuvanje/QueueDemo.java
// Pravljenje reda za čekanje od klase LinkedList svođenjem naviše
// na Queue.
import java.util.*;

public class QueueDemo {
  public static void printQ(Queue redZaCekanje) {
    while(redZaCekanje.peek() != null)
      System.out.print(redZaCekanje.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> redZaCekanje = new LinkedList<Integer>();
```

```
    Random slucajan = new Random(47);
    for(int i = 0; i < 10; i++)
      redZaCekanje.offer(slucajan.nextInt(i + 10));
    printQ(redZaCekanje);
    Queue<Character> rcZnakova = new LinkedList<Character>;
    for(char znak : "Brontosaurus".toCharArray())
      rcZnakova.offer(znak);
    printQ(rcZnakova);
  }
} /* Ispis:
8 1 1 1 5 14 3 1 0 1
B r o n t o s a u r u s
*///:~
```

Metoda **offer( )** jedna je od onih specifičnih za klasu **Queue**; ona umeće element na rep reda za čekanje ako može, ili vraća **false**. Metode **peek( )** i **element( )** vraćaju čelo reda za čekanje, *a da ga ne uklone*, ali **peek( )** vraća **null** ako je red za čekanje prazan, dok **element( )** generiše izuzetak **NoSuchElementException**. Metode **poll( )** i **remove( )** uklanjaju i vraćaju čelo reda za čekanje, ali **poll( )** vraća **null** ako je red za čekanje prazan, dok **remove( )** u tom slučaju generiše izuzetak **NoSuchElementException**.

Automatsko pakovanje automatski pretvara **int** rezultat metode **nextInt( )** u objekat tipa **Integer** koji zahteva **redZaCekanje**, kao i **char znak** u **Character** objekat koji zahteva **rcZnakova**. Interfejs **Queue** sužava pristup metodama liste **LinkedList** tako da budu dostupne samo prikladne metode, pa baš i niste u iskušenju da upotrebljavate metode liste **LinkedList** (ovde biste **redZaCekanje** zaista mogli da svedete unazad na **LinkedList**, ali to vam je barem otežano).

Imajte u vidu da metode specifične za **Queue** pružaju potpunu i samostalnu funkcionalnost. Drugim rečima, možete napraviti upotrebljiv red za čekanje a da ne primenite nijednu metodu klase **Collection**, od koje je **Queue** nasleđen.

Vežba 27: (2) Napišite klasu **Naredba** koja sadrži jedan **String** i ima metodu **operacija( )** koja ga prikazuje. Napišite drugu klasu s metodom koja popunjava **Queue** objektima tipa **Naredba** i vraća taj red za čekanje. Popunjen **Queue** prosledite metodi u trećoj klasi koja uzima objekte reda za čekanje i poziva njihove metode **operacija( )**.

Prioritetni red čekanja (PriorityQueue)

„Prvi izlazi onaj koji je prvi ušao" (engl. *first-in first-out*, *FIFO*) opisuje najtipičniju *disciplinu čekanja*. Ako imamo grupu elemenata u redu za čekanje, disciplina čekanja određuje koji element sledeći izlazi iz reda. Po principu FIFO, sledeći element treba da bude onaj koji najduže čeka.

Prioritetni red čekanja kazuje da iz reda prvo izlazi element koji ima najveću potrebu (prioritet) da izađe. Na primer, na aerodromu se iz reda za čekanje izvlači putnik čiji se avion upravo sprema za poletanje. Ako napravite sistem za razmenu poruka, neke poruke su važnije od drugih i treba ih pre obraditi, bez obzira na to kada su stigle. Klasa **PriorityQueue** dodata je Javi SE5 da bi se napravila automatska realizacija takvog ponašanja.

Kada metodom **offer( )** ponudite (engl. *offer*) neki objekat klasi **PriorityQueue**, on će biti sortiran i ubačen na odgovarajuće mesto tog reda za čekanje.[5] Pri podrazumevanom sortiranju (uređivanju) upotrebljava se *prirodni redosled* objekata u redu za čekanje, ali poredak možete da izmenite svojim **Comparatorom**. **PriorityQueue** se stara da pri pozivanju metode **peek( )**, **poll( )** ili **remove( )**, od nje dobijete element s najvišim prioritetom.

Veoma je lako napraviti prioritetni red za čekanje koji radi sa ugrađenim tipovima kao što su **Integer**, **String** ili **Character**. U narednom primeru, prvi skup vrednosti čine identični slučajni brojevi iz prethodnog primera, pa se možete uveriti da oni iz prioritetnog rada za čekanje izlaze u drugom poretku:

```
//: cuvanje/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<Integer> prioritetniRed =
      new PriorityQueue<Integer>();
    Random slucajan = new Random(47);
    for(int i = 0; i < 10; i++)
      prioritetniRed.offer(slucajan.nextInt(i + 10));
    QueueDemo.printQ(prioritetniRed);

    List<Integer> celiBrojevi = Arrays.asList(25, 22, 20,
      18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
    prioritetniRed = new PriorityQueue<Integer>(celiBrojevi);
    QueueDemo.printQ(prioritetniRed);
    prioritetniRed = new PriorityQueue<Integer>(
        celiBrojevi.size(), Collections.reverseOrder());
    prioritetniRed.addAll(celiBrojevi);
    QueueDemo.printQ(prioritetniRed);

    String cinjenica = "EDUCATION SHOULD ESCHEW OBFUSCATION";
    List<String> strings = Arrays.asList(cinjenica.split(""));
    PriorityQueue<String> stringPQ =
      new PriorityQueue<String>(strings);
    QueueDemo.printQ(stringPQ);
    stringPQ = new PriorityQueue<String>(
      strings.size(), Collections.reverseOrder());
    stringPQ.addAll(strings);
    QueueDemo.printQ(stringPQ);

    Set<Character> skupZnakova = new HashSet<Character>();
    for(char c : cinjenica.toCharArray())
      skupZnakova.add(c); // Automatsko pakovanje
    PriorityQueue<Character> characterPQ =
```

[5] U stvari, ovo zavisi od realizacije. Algoritmi prioritetnih redova za čekanje obično sortiraju elemente odmah nakon umetanja (održavajući *gomilu*), ali izbor najvažnijeg elementa mogu obaviti i nakon uklanjanja. Koji se algoritam koristi, važno je ako se prioritet objekta može promeniti dok on čeka u redu.

```
      new PriorityQueue<Character>(skupZnakova);
    QueueDemo.printQ(characterPQ);
  }
} /* Ispis:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
      A A B C C C D D E E E F H H I I L N N O O O O S S S T T U U U W
W U U U T T S S S O O O O N N L I I H H F E E E D D C C C B A A
  A B C D E F H I L N O S T U W
*///:~
```

Vidite da su duplikati dozvoljeni i da najmanje vrednosti imaju najviši prioritet (u tipu **String**, i razmaci se broje kao vrednosti, i imaju viši prioritet od slova). Da biste videli kako svojim objektom tipa **Comparator** možete izmeniti redosled sortiranja, treći poziv konstruktora **PriorityQueue<Integer>** i drugi poziv konstruktora **PriorityQueue<String>** upotrebljavaju **Comparator** obrnutog redosleda koji daje metoda **Collections.reverseOrder( )** – dodata u Javu SE5.

U poslednjem odeljku dodaje se **HashSet** koji eliminiše duplikate znakova (**Character**), samo da bi bilo malo zanimljivije.

Integer, **String** i **Character** rade s klasom **PriorityQueue** zato što njihove klase već imaju ugrađen svoj prirodni poredak. Ukoliko u prioritetnom redu za čekanje hoćete da upotrebljavate sopstvene klase, morate dodati i funkcionalnost koja proizvodi prirodni poredak ili obezbediti svoj **Comparator**. U poglavlju *Detaljno razmatranje kontejnera* naći ćete i tananiji primer koji to pokazuje.

Vežba 28: (2) Pomoću metode **offer( )**, popunite primerak klase **PriorityQueue** brojevima tipa **Double** napravljenim pomoću klase **java.util.Random**. Zatim uklonite elemente metodom **poll( )** i prikažite ih.

Vežba 29: (2) Napravite jednostavnu klasu koja nasleđuje **Object** i nema članova, i pokažite da ne možete uspešno dodati više elemenata te klase prioritetnom redu za čekanje. To će biti potpuno objašnjeno u poglavlju *Detaljno razmatranje kontejnera.*

Poređenje kolekcija i Iteratora

Collection je korenski interfejs koji opisuje ono što je zajedničko svim kontejnerima sekvenci. Možete ga smatrati „slučajnim interfejsom" koji je nastao zbog međusobne sličnosti drugih interfejsa. Sem toga, klasa **java.util.AbstractCollection** ima podrazumevanu realizaciju klase **Collection**, pa nove podtipove od **AbstractCollection** možete praviti bez nepotrebnog dupliranja koda.

Jedan od argumenata za pravljenje interfejsa jeste to da on omogućava pisanje opštijeg koda. Ako je kôd napisan za interfejs a ne za realizaciju, primenljiv je na više tipova objekata.[6] Ukoliko napišem metodu koja uzima **Collection**, ta metoda se može primeniti na

[6] Ima ljudi koji zagovaraju automatsko pravljenje interfejsa za svaku moguću kombinaciju metoda u klasi – ponekad i za svaku klasu. Smatram kako interfejs treba da znači više od mehaničkog dupliranja kombinacija metoda, pa obično prvo sagledam vrednost koju će interfejs doneti i tek tad ga pravim.

svaki tip koji realizuje **Collection** – a to svakoj novoj klasi pruža mogućnost da realizuje **Collection** da bi mogla biti upotrebljena s mojom metodom. Zanimljivo je primetiti da standardna biblioteka C++-a za svoje kontejnere nema zajedničku osnovnu klasu – sve što im je zajedničko, postiže se iteratorima. Možda bi u Javi bilo pametno slediti primer iz C++-a i izraziti sličnost kontejnera iteratorom, a ne kolekcijom. Međutim, ta dva pristupa su povezana, pošto realizovanje klase **Collection** znači i obezbeđivanje metode **iterator( )**:

```
//: cuvanje/PoredjenjeInterfejsaIteratora.java
import typeinfo.pets.*;
import java.util.*;

public class PoredjenjeInterfejsaIteratora {
  public static void ispisi(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void ispisi(Collection<Pet> ljubimci)
    for(Pet p : ljubimci)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> listaljubimaca = Pets.arrayList(8);
    Set<Pet> skupljubimaca = new HashSet<Pet>( listaljubimaca);
    Map<String,Pet> mapaljubimaca =
      new LinkedHashMap<String,Pet>();
    String[] imena = ("Ralf, Erik, Robin, Lejsi, " +
      "Britni, Sima, Tufna, Paperjast").split(", ");
    for(int i = 0; i < imena.length; i++)
      mapaljubimaca.put(imena[i], listaljubimaca.get(i));
    ispisi(listaljubimaca);
    ispisi(skupljubimaca);
    ispisi(listaljubimaca.iterator());
    ispisi(skupljubimaca.iterator());
    System.out.println(mapaljubimaca);
    System.out.println(mapaljubimaca.keySet());
    ispisi(mapaljubimaca.values());
    ispisi(mapaljubimaca.values().iterator());
  }
} /* Ispis:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
{Ralf=Rat, Erik=Manx, Robin=Cymric, Lejsi=Mutt, Britni=Pug, Sima=Cymric,
Tufna=Pug, Paperjast=Manx}
```

```
[Ralf, Erik, Robin, Lejsi, Britni, Sima, Tufna, Paperjast]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Obe verzije metode **ispisi( )** rade i sa objektima tipa **Map** i s podtipovima od **Collection**. I interfejs **Collection** i **Iterator** odeljuju metode **ispisi( )** od poznavanja konkretne realizacije pripadnog kontejnera.

U ovom slučaju, oba pristupa su jednaka. U stvari, **Collection** je nešto bolji pošto se može iterirati (ima svojstvo **Iterable**), pa se u realizaciji **ispisi(Collection)** može upotrebiti foreach sintaksa, zbog čega je kôd nešto čistiji.

Upotreba **Iterator**a postaje obavezna kada realizujete stranu klasu (onu koja nije podtip od **Collection**), u kojoj bi bilo teško ili besmisleno realizovati interfejs **Collection**. Primera radi, ako **Collection** realizujemo nasleđivanjem klase koja sadrži **Pet** objekte, moramo realizovati sve metode interfejsa **Collection**, čak i ako nam ne trebaju u metodi **ispisi( )**. Iako je to lako napraviti pomoću nasleđivanja klase **AbstractCollection**, ipak morate da realizujete i **iterator( )** i **size( )**, da biste obezbedili metode koje **AbstractCollection** ne realizuje, ali ih upotrebljavaju druge metode u **AbstractCollection**:

```
//: cuvanje/SekvencaKolekcija.java
import typeinfo.pets.*;
import java.util.*;

public class SekvencaKolekcija
extends AbstractCollection<Pet> {
  private Pet[] ljubimci = Pets.createArray(8);
  public int size() { return ljubimci.length; }
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int indeks = 0;
      public boolean hasNext() {
        return indeks < ljubimci.length;
      }
      public Pet next() { return ljubimci[index++]; }
      public void remove() { // Nije realizovano
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    SekvencaKolekcija c = new SekvencaKolekcija();
    PoredjenjeInterfejsaiIteratora.ispisi(c);
    PoredjenjeInterfejsaiIteratora.ispisi(c.iterator());
  }
} /* Ispis:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Metoda **remove( )** jedna je od opcionih operacija, s kojima ćemo vas upoznati u poglavlju *Detaljno razmatranje kontejnera*. Ovde je nije neophodno realizovati, pa ćete izazvati izuzetak ako je pozovete.

Iz ovog primera možete videti sledeće: ako realizujete **Collection**, realizujete i **iterator( )**, a realizovanje samo metode **iterator( )** zahteva tek malo manje posla od nasleđivanja klase **AbstractCollection**. Međutim, ukoliko vaša klasa već nasleđuje drugu klasu, ne može naslediti i **AbstractCollection**. U tom slučaju, da biste realizovali **Collection**, morali biste da realizujete sve metode tog interfejsa. Tada bi bilo mnogo lakše naslediti **Collection** i dodati mogućnost pravljenja iteratora:

```
//: cuvanje/SekvencaBezKolekcije.java
import typeinfo.pets.*;
import java.util.*;

class Sekvencaljubimaca {
  protected Pet[] ljubimci = Pets.createArray(8);
}

public class SekvencaBezKolekcije extends Sekvencaljubimaca {
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int indeks = 0;
      public boolean hasNext() {
        return indeks < ljubimci.length;
      }
      public Pet next() { return ljubimci[indeks++]; }
      public void remove() { // Nije realizovano
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    SekvencaBezKolekcije nc = new SekvencaBezKolekcije();
    PoredjenjeInterfejsaIteratora.ispisi(nc.iterator());
  }
} /* Ispis:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Pravljenje **Iteratora** je najrazdvojeniji način povezivanja sekvence i metode koja prima tu sekvencu, a pri tom se klasa sekvence ograničava znatno manje nego kada se implementira klasa **Collection**.

Vežba 30: (5) Izmenite program **CollectionSequence.java** tako da ne nasleđuje **AbstractCollection**, nego da realizuje **Collection**.

Foreach i iteratori

Dosad smo foreach sintaksu uglavnom upotrebljavali s nizovima, ali ona radi i sa svim objektima **Collection**. Zapravo, već ste videli nekoliko primera za to u kojima se upotrebljavala **ArrayList**, ali ovo je opšti dokaz:

```
//: cuvanje/ForEachKolekcije.java
// Sve kolekcije rade s foreach sintaksom.
import java.util.*;

public class ForEachKolekcije {
  public static void main(String[] args) {
    Collection<String> cs = new LinkedList<String>();
    Collections.addAll(cs,
      "Take the long way home".split(" "));
    for(String s : cs)
      System.out.print("'" + s + "' ");
  }
} /* Ispis:
'Take' 'the' 'long' 'way' 'home'
*///:~
```

Pošto je **cs** tipa **Collection**, ovaj kod dokazuje da sve kolekcije mogu da rade s foreach sintaksom.

Ovo funkcioniše zato što je Java SE5 uvela nov interfejs nazvan **Iterable** koji sadrži metodu **iterator( )** za pravljenje **Iterator**a. Za kretanje kroz sekvencu, foreach upotrebljava interfejs **Iterable**. Dakle, ako napravite klasu koja realizuje **Iterable**, možete je upotrebiti u foreach naredbi:

```
//: cuvanje/IterabilnaKlasa.java
// Sve što je iterabilno funkcioniše s foreach sintaksom.
import java.util.*;

public class IterabilnaKlasa implements Iterable<String> {
  protected String[] rechi = ("And that is how " +
    "we know the Earth to be banana-shaped.").split(" ");
  public Iterator<String> iterator() {
    return new Iterator<String>() {
      private int indeks = 0;
      public boolean hasNext() {
        return indeks < rechi.length;
      }
      public String next() { return rechi[indeks++]; }
      public void remove() { // Nije realizovana
        throw new UnsupportedOperationException();
      }
    };
  }
```

```
  public static void main(String[] args) {
    for(String s : new IterabilnaKlasa())
      System.out.print(s + " ");
  }
} /* Ispis:
And that is how we know the Earth to be banana-shaped.
*///:~
```

Metoda **iterator( )** vraća instancu anonimne unutrašnje realizacije klase **Iterator<String>** koja daje svaku reč niza. U metodi **main( )** možete videti da **IterabilnaKlasa** zaista radi u foreach naredbi.

U Javi SE5 iterabilno je više klasa, prvenstveno sve klase **Collection** (ali ne **Map**e). Na primer, ovaj program prikazuje sve promenljive okruženja operativnog sistema:

```
//: cuvanje/PromenljiveOkruzenja.java
import java.util.*;

public class PromenljiveOkruzenja {
  public static void main(String[] args) {
    for(Map.Entry stavka: System.getenv().entrySet()) {
      System.out.println(stavka.getKey() + ": " +
        stavka.getValue());
    }
  }
} /* (Pokrenite da biste videli rezultate) *///:~
```

System.getenv( )[7] vraća **Map**u; **entrySet( )** pravi skup (**Set**) elemenata **Map.Entry**, a pošto je skup iterabilan, može se upotrebiti u foreach petlji.

Foreach naredba radi s nizovima i sa svim što je iterabilno, ali to ne znači da je svaki niz automatski iterabilan, niti da se obavlja ikakvo automatsko pakovanje:

```
//: cuvanje/NizNijeIterabilan.java
import java.util.*;

public class NizNijeIterabilan {
  static <T> void test(Iterable<T> ib) {
    for(T t : ib)
      System.out.print(t + " ");
  }
  public static void main(String[] args) {
    test(Arrays.asList(1, 2, 3));
    String[] znakovni_nizovi = { "A", "B", "C" };
    // Niz radi u foreach naredbi, ali nije iterabilan:
    //! test(znakovni_nizovi);
    // Morate ga izričito pretvoriti u nešto iterabilno:
    test(Arrays.asList(znakovni_nizovi));
```

[7] Ova metoda nije postojala pre Jave SE5, pošto se smatralo da bi bila previše usko povezana sa operativnim sistemom, pa bi se kršilo pravilo „napiši jednom, izvršavaj bilo gde". Činjenica da metoda sada postoji kazuje da su projektanti Jave nakon pojave .NET-a postali pragmatičniji.

```
  }
} /* Ispis:
1 2 3 A B C
*///:~
```

Pokušaj da se niz prosledi kao iterabilan argument, ne uspeva. Ne postoji automatska konverzija u **Iterable**; morate je obaviti ručno.

Vežba 31: (3) Izmenite **polimorfizam/oblik/GeneratorSlucajnihOblika.java** tako da postane iterabilan. Morate dodati konstruktor koji prima broj elemenata koji iterator treba da napravi pre zaustavljanja. Dokažite da to radi.

Adapterska metoda

Šta ako imate iterabilnu klasu i hteli biste da dodate jedan ili više novih načina upotrebe te klase u foreach naredbi? Primera radi, pretpostavimo da biste hteli da birate hoćete li kroz listu reči iterirati unapred ili unazad. Ukoliko samo nasledite tu klasu i redefinišete metodu **iterator( )**, zamenićete postojeću metodu i nećete moći da birate.

Jedno rešenje sam nazvao *Adapterska metoda*. „Adapterski" deo potiče iz projektnih obrazaca, pošto morate obezbediti određeni interfejs da biste zadovoljili foreach naredbu. Kada imate jedan interfejs a treba vam drugi, problem ćete rešiti ukoliko napišete adapter. Ovde hoću da *dodam* mogućnost pravljenja iteratora unazad podrazumevanom iteratoru unapred, pa ne mogu da ga redefinišem. Umesto toga, dodaću metodu koja proizvodi iterabilan objekat koji se potom može upotrebiti u foreach naredbi. Kao što ćete videti, time dobijamo više načina upotrebe foreach sintakse:

```
//: cuvanje/AdapterskaMetoda.java
// "Adapterska metoda" omogućava upotrebu foreach
// sintakse s dodatnim vrstama iterabilnih objekata.
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
  public ReversibleArrayList(Collection<T> c) { super(c); }
  public Iterable<T> obrnuto() {
    return new Iterable<T>() {
      public Iterator<T> iterator() {
        return new Iterator<T>() {
          int tekuci = size() - 1;
          public boolean hasNext() { return tekuci > -1; }
          public T next() { return get(tekuci--); }
          public void remove() { // Nije realizovano
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}
```

```
public class AdapterskaMetoda {
  public static void main(String[] args) {
    ReversibleArrayList<String> ral =
      new ReversibleArrayList<String>(
        Arrays.asList("To be or not to be".split(" ")));
    // Metodom iterator() grabi se običan iterator:
    for(String s : ral)
      System.out.print(s + " ");
    System.out.println();
    // Dajte ga iterabilnom objektu po svom izboru
    for(String s : ral.obrnuto())
      System.out.print(s + " ");
  }
} /* Ispis:
To be or not to be
be to not or be To
*///:~
```

Ako objekat **ral** samo stavite u foreach naredbu, dobićete (podrazumevani) iterator unapred. Ali ukoliko za taj objekat pozovete metodu **obrnuto( )**, ona će dati drugačije ponašanje.

Na taj način, klasi **IterabilnaKlasa.java** mogu se dodati dve adapterske metode:

```
//: cuvanje/VisestrukoIterabilnaKlasa.java
// Dodavanje više adapterskih metoda.
import java.util.*;

public class VisestrukoIterabilnaKlasa extends IterabilnaKlasa {
  public Iterable<String> obrnuto() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        return new Iterator<String>() {
          int tekuci = rechi.length - 1;
          public boolean hasNext() { return tekuci > -1; }
          public String next() { return rechi[tekuci--]; }
          public void remove() { // Nije realizovano
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
  public Iterable<String> slucajno() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        List<String> izmesana =
          new ArrayList<String>(Arrays.asList(words));
        Collections.shuffle(izmesana, new Random(47));
```

```
        return izmesana.iterator();
      }
    };
  }
  public static void main(String[] args) {
    VisestrukoIterabilnaKlasa mic = new VisestrukoIterabilnaKlasa();
    for(String s : mic.obrnuto())
      System.out.print(s + " ");
    System.out.println();
    for(String s : mic.slucajno())
      System.out.print(s + " ");
    System.out.println();
    for(String s : mic)
      System.out.print(s + " ");
  }
} /* Ispis:
banana-shaped. be to Earth the know we how is that And
is banana-shaped. Earth that how the be And we know to
And that is how we know the Earth to be banana-shaped.
*///:~
```

Obratite pažnju na to da druga metoda, **random( )**, ne pravi sopstveni **Iterator** nego vraća onaj iz izmešane **Liste**.

Iz rezultata vidite da metoda **Collections.shuffle( )** ne utiče na originalni niz, nego samo meša reference u objektu **izmesana**. Ovo je tačno samo zato što metoda **slucajno( )** omotava **ArrayList** oko rezultata metode **Arrays.asList( )**. Da je **Lista** napravljena metodom **Arrays.asList( )** izmešana direktno, ona bi modifikovala pripadni niz, kao što vidite:

```
//: cuvanje/ModifikovanjeNizovaKaoListi.java
import java.util.*;

public class ModifikovanjeNizovaKaoListi {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    Integer[] nizcb = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    List<Integer> lista1 =
      new ArrayList<Integer>(Arrays.asList(nizcb));
    System.out.println("Pre mešanja: " + lista1);
    Collections.shuffle(lista1, slucajan);
    System.out.println("Nakon mešanja: " + lista1);
    System.out.println("niz: " + Arrays.toString(nizcb));

    List<Integer> lista2 = Arrays.asList(nizcb);
    System.out.println("Pre mešanja: " + lista2);
    Collections.shuffle(lista2, slucajan);
    System.out.println("Nakon mešanja: " + lista2);
    System.out.println("niz: " + Arrays.toString(nizcb));
  }
```

```
} /* Ispis:
Pre mešanja: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Nakon mešanja: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
niz: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Pre mešanja: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Nakon mešanja: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
niz: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:~
```

U prvom slučaju, izlaz metode **Arrays.asList( )** predaje se konstruktoru **ArrayList( )**, a on pravi **ArrayList** koja referencira elemente niza **nizcb**. Mešanje tih referenci ne modifikuje niz. Međutim, ukoliko rezultat metode **Arrays.asList(nizcb)** upotrebite direktno, mešanje modifikuje poredak niza **nizcb**. Imajte u vidu da **Arrays.asList( )** pravi objekat tipa **List** koji upotrebljava pripadni niz kao svoju fizičku realizaciju. Ukoliko bilo kako izmenite tu **List**u, a ne želite da se modifikuje originalni niz, napravite njegovu kopiju u drugom kontejneru.

Vežba 32: (2) Na osnovu primera **VisestrukoIterabilnaKlasa** u programu **SekvencaBezKolekcije.java** dodajte metode **obrnuto( )** i **slucajno( )**, a u **SekvencaBezKolekcije** realizujte **Iterable**, i pokažite da u foreach naredbama funkcionišu sve te varijante.

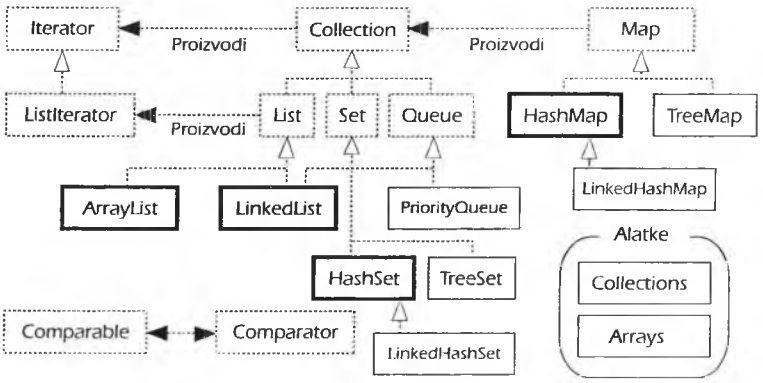
Sažetak

Java ima više načina za čuvanje objekata:

1. Niz povezuje numeričke indekse sa objektima. On sadrži objekte poznatog tipa, pa nakon uzimanja objekta iz niza ne morate da vršite konverziju tipova. Niz može da bude višedimenzionalan i da sadrži proste tipove. Međutim, ne možete da mu promenite veličinu.
2. Kolekcija sadrži pojedinačne elemente, dok mapa sadrži povezane parove. Pomoću Javinih generičkih tipova zadajete tip objekta koji će se čuvati u kontejneru, pa u njega ne možete staviti pogrešan tip i ne morate da proveravate tip elemenata kada ih izvadite iz njega. I kolekcije i **Map**e automatski prilagođavaju svoju veličinu nakon dodavanja i uklanjanja elemenata. Kontejneri ne mogu da sadrži proste tipove, ali se automatsko pakovanje stara za prevođenje primitivnih tipova u omotačke tipove koji se čuvaju u kontejneru, i obrnuto.
3. Poput niza, lista takođe povezuje numeričke indekse sa objektima; nizovi i liste mogu se smatrati uređenim kontejnerima.
4. Upotrebite listu **ArrayList** ako često koristite nasumični pristup, a **LinkedList** ako često umećete i uklanjate elemente u sredini liste.
5. Redovi za čekanje i stekovi prave se pomoću klase **LinkedList**.
6. **Map**a je način za povezivanje objekata sa drugim objektima. Klasa **HashMap** je projektovana za brz pristup, dok **TreeMap** čuva ključeve u uređenom redosledu i zato nije tako brza kao **HashMap**. Klasa **LinkedHashMap** čuva svoje elemente u uređenom redosledu, ali transformacija ključeva obezbeđuje brz pristup.

7. **Set** čuva samo objekte različitih vrednosti. Klasa **HashSet** obezbeduje pronalaženje maksimalnom brzinom, dok **TreeSet** čuva elemente u uređenom redosledu. Klasa **LinkedHashSet** čuva svoje elemente u uređenom redosledu.
8. Nema potrebe da se u novim programima koriste stare klase **Vector**, **Hashtable** i **Stack**.

Pogledajte pojednostavljen dijagram Javinih kontejnera (bez apstraktnih klasa i starih komponenata). U njemu su samo interfejsi i klase koje ćete redovno sretati.



Vidite da postoje samo četiri kontejnerske komponente: **Map**, **List**, **Set** i **Queue**, i samo dve ili tri realizacije svake od njih (realizacije klase **Queue** iz paketa **java.util.concurrent** nisu prikazane na dijagramu). Najčešće ćete upotrebljavati kontejnere oivičene debelom punom linijom.

Tačkastim okvirima obeleženi su interfejsi, a okvirima s punom linijom obične (konkretne) klase. Isprekidane prazne strelice ukazuju da određena klasa realizuje interfejs. Strelice s punom linijom ukazuju da klasa može da pravi objekte klase na koju pokazuje strelica. Na primer, bilo koja kolekcija može da pravi **Iterator**, dok **List** može da napravi **ListIterator** (kao i običan **Iterator**, pošto je **List** izvedena iz **Collection**).

Evo primera u kojem se vidi razlika između metoda raznih klasa. Kod je naveden u poglavlju *Generički tipovi*; ovde ga samo pozivam da bih dobio rezultate. U njima se vide i interfejsi koje realizuje pojedina klasa ili interfejs:

```
//: cuvanje/KontejnerskeMetode.java
import net.mindview.util.*;

public class KontejnerskeMetode {
  public static void main(String[] args) {
    RazlikeKontejnerskihMetoda.main(args);
  }
} /* Ispis: (primer)
Collection: [add, addAll, clear, contains, containsAll, equals, hashCode,
isEmpty, iterator, remove, removeAll, retainAll, size, toArray]
```

```
Interfejsi u klasi Collection: [Iterable]
Set extends Collection, dodaje: []
Interfejsi u klasi Set: [Collection]
HashSet extends Set, dodaje: []
Interfejsi u klasi HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, dodaje: []
Interfejsi u klasi LinkedHashSet: [Set, Cloneable, Serializable]
TreeSet extends Set, dodaje: [pollLast, navigableHeadSet,
descendingIterator, lower, headSet, ceiling, pollFirst, subSet,
navigableTailSet, comparator, first, floor, last, navigableSubSet,
higher, tailSet]
Interfejsi u klasi TreeSet: [NavigableSet, Cloneable, Serializable]
List extends Collection, dodaje: [listIterator, indexOf, get, subList,
set, lastIndexOf]
Interfejsi u klasi List: [Collection]
ArrayList extends List, dodaje: [ensureCapacity, trimToSize]
Interfejsi u klasi ArrayList: [List, RandomAccess, Cloneable,
Serializable]
LinkedList extends List, dodaje: [pollLast, offer, descendingIterator,
addFirst, peekLast, removeFirst, peekFirst, removeLast, getLast,
pollFirst, pop, poll, addLast, removeFirstOccurrence, getFirst,
element, peek, offerLast, push, offerFirst, removeLastOccurrence]
Interfejsi u klasi LinkedList: [List, Deque, Cloneable, Serializable]
Queue extends Collection, dodaje: [offer, element, peek, poll]
Interfejsi u klasi Queue: [Collection]
PriorityQueue extends Queue, dodaje: [comparator]
Interfejsi u klasi PriorityQueue: [Serializable]
Map: [clear, containsKey, containsValue, entrySet, equals, get, hashCode,
isEmpty, keySet, put, putAll, remove, size, values]
HashMap extends Map, dodaje: []
Interfejsi u klasi HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, dodaje: []
Interfejsi u klasi LinkedHashMap: [Map]
SortedMap extends Map, dodaje: [subMap, comparator, firstKey, lastKey,
headMap, tailMap]
Interfejsi u klasi SortedMap: [Map]
TreeMap extends Map, dodaje: [descendingEntrySet, subMap, pollLastEntry,
lastKey, floorEntry, lastEntry, lowerKey, navigableHeadMap,
navigableTailMap, descendingKeySet, tailMap, ceilingEntry, higherKey,
pollFirstEntry, comparator, firstKey, floorKey, higherEntry,
firstEntry, navigableSubMap, headMap, lowerEntry, ceilingKey]
Interfejsi u klasi TreeMap: [NavigableMap, Cloneable, Serializable]
*///:~
```

Vidite da svi skupovi (objekat tipa **Set**) sem tipa **TreeSet** imaju potpuno isti interfejs kao **Collection**. **List** i **Collection** se znatno razlikuju, iako **List** zahteva metode koje su u interfejsu **Collection**. S druge strane, metode interfejsa **Queue** su samostalne; za pravljenje funkcionalne realizacije reda za čekanje (**Queue**), metode interfejsa **Collection** nisu potrebne. Najzad, jedina dodirna tačka klasa **Map** i **Collection** jeste činjenica da **Map** može da pravi kolekcije pomoću metoda **entrySet( )** i **values( )**.

Obratite pažnju na interfejs **java.util.RandomAccess** koji je prikačen uz **ArrayList**, ali ne i uz **LinkedList**. Time se pruža informacija algoritmima čije će se ponašanje dinamički menjati u zavisnosti od upotrebe određene **List**e.

Ova organizacija je malo čudnija od objektno orijentisane hijerarhije. Međutim, kako budete više saznavali o kontejnerima paketa **java.util** (naročito u poglavlju *Detaljno razmatranje kontejnera*), videćete da nije problematična samo čudna struktura nasleđivanja. Oduvek je bilo teško projektovati biblioteke kontejnera, jer treba zadovoljiti međusobno suprotstavljene sile. Zato poneki kompromis ne bi trebalo da vas iznenadi.

Uza sve to, Javini kontejneri su osnovne alatke kojima svoje programe možete učiniti jednostavnijim, snažnijim i delotvornijim. Možda će vam trebati malo vremena da se naviknete na određene aspekte ove biblioteke, ali mislim da ćete njene klase brzo početi da nabavljate i upotrebljavate.

Rešenja odabranih vežbi su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.net*.

Obrada grešaka pomoću izuzetaka

OSNOVNA MUDROST JAVE GLASI DA KÔD KOJI JE LOŠE NAPISAN NEĆE NI BITI IZVRŠAVAN. Najbolje bi bilo kada biste sve greške mogli da otkrijete tokom prevođenja, pre nego što uopšte pokušate da pokrenete program. Međutim, ne mogu se sve greške otkriti pri prevođenju. Neki problemi se moraju rešavati za vreme rada, formalnim postupcima; oni omogućuju vinovniku greške da prosledi odgovarajuće informacije primaocu koji će znati kako da se na pravi način izbori s teškoćom.

Poboljšano oporavljanje od grešaka jedan je od najboljih načina za povećanje robusnosti koda. Oporavljanje od grešaka je najvažnije za svaki program koji pišete, a posebno je važno u Javi, gde je jedan od primarnih ciljeva pravljenje programskih komponenata za višekratnu upotrebu. *Da biste napravili robustan sistem, svaka njegova komponenta mora biti robusna.* Java omogućava da komponente pouzdano dojave svoje probleme klijentskom kodu tako što obezbeđuje dosledan model prijavljivanja grešaka pomoću izuzetaka.

Ciljevi obrade grešaka u Javi su pojednostavljivanje pravljenja velikih, pouzdanih programa pomoću manje koda nego što je trenutno moguće, i uz povećavanje verovatnoće da se u aplikaciji neće pojaviti greška koja nije obrađena. Izuzetke nije toliko teško savladati, a spadaju među elemente Jave od kojih projekti imaju trenutne i primetne koristi.

Pošto je obrada izuzetaka jedini zvanični način na koji Java prijavljuje greške, a sprovodi je prevodilac Jave, u ovoj knjizi ne može biti mnogo primera napisanih bez nje. U ovom poglavlju upoznajete se s pravilima rada sa izuzecima i načinom na koji možete da generišete sopstvene izuzetke kada se neka od vaših metoda splete.

Koncepti

C i drugi stariji jezici često su imali i po nekoliko šema za obradu grešaka koje su po pravilu bile utvrđene dogovorom, odnosno nisu bile deo programskog jezika. Uglavnom se vraćala posebna vrednost ili se postavljao indikator, a primalac bi posmatrao vrednost, odnosno indikator i odlučivao da li je sve u redu. Međutim, tokom godina otkriveno je da programeri koji koriste biblioteku obično sebe smatraju nepogrešivim i razmišljaju u stilu: „Da, greške se možda dešavaju drugima, ali u *mom* kodu ih nema". Zbog toga ne iznenađuje što nisu proveravali uslove pod kojima je nastajala greška (a ponekad su uzroci greške bili previše glupi da bi se proveravali).[1] *Kada* biste bili dovoljno pažljivi da tražite grešku posle svake metode, vaš kôd bi se pretvorio u nečitljivu noćnu moru. Pošto su programeri i dalje mogli da održavaju sisteme napisane opisanom tehnikom, odbijali su da priznaju istinu: ovakav pristup ispravljanju grešaka umnogome je ograničavao pisanje velikih, snažnih programa koji bi se lako održavali.

Rešenje je da se postupku obrade grešaka oduzme prirodnost i da se ojača formalnost. Ovaj način, zapravo, ima dugu istoriju, jer se *obrada izuzetaka* (engl. *exception handling*) sreće još u operativnim sistemima iz šezdesetih godina prošlog veka, čak i u naredbi **on error goto** iz programskog jezika BASIC. Tako se obrada grešaka u jeziku C++ zasnivala na jeziku Ada, a Javina pretežno na C++-u (iako više podseća na objektni Pascal).

---

[1] Programer na jeziku C, na primer, može da zatraži povratnu vrednost funkcije **printf**( ).

Reč „izuzetak" se koristi u smislu „ograđujem se od toga". Kada se pojavi problem, možda nećete znati kako da ga otklonite, ali ćete sigurno znati da program ne može tek tako da nastavi rad. Morate se zaustaviti i neko negde mora smisliti šta da se radi. Međutim, možda u tom trenutku nemate dovoljno informacija da biste rešili problem. Zbog toga ga prenosite na viši nivo i predajete nekome ko je dovoljno stručan da ga reši.

Još jedna prilično korisna osobina izuzetaka jeste da oni obično pojednostavljuju kôd za obradu grešaka. Da nema izuzetaka, morali biste da tražite određenu grešku i da se s njom borite na nekoliko mesta u programu. Sa izuzecima, više ne morate da proveravate greške prilikom poziva metoda, pošto izuzetak garantuje da će ih neko uhvatiti. Problem rešavate na jednom mestu, a to je tzv. *blok za obradu izuzetaka* (engl. *exception handler*). On skraćuje pisanje koda i odvaja kôd koji opisuje šta želite da uradite tokom normalnog izvršavanja, od koda koji se izvršava kada nešto krene naopako. Obično je kôd za upisivanje, čitanje i pronalaženje grešaka mnogo pregledniji kada se koriste izuzeci nego kada se primenjuje stari način rada s greškama.

Osnovni izuzeci

Vanredno stanje (engl. *exceptional condition*) jeste problem koji sprečava nastavak rada tekuće metode ili programskog bloka. Važno je razlikovati vanredno stanje od uobičajenog problema, s kojim možete da se izborite u tekućem okruženju. Kada nastupi vanredno stanje, ne možete nastaviti obradu zato što nemate neophodne informacije da biste se izborili s problemom u *tekućem okruženju*. Možete samo da iskočite iz tekućeg okruženja i da prenesete problem u više okruženje. To se dešava kada se pojavi izuzetak.

Jednostavan primer je deljenje. Da bi se izbeglo deljenje nulom, dobro je proveriti delilac unapred. Ipak, šta zaista znači kada je delilac nula? Možda znate, u kontekstu problema koji pokušavate da rešite određenom metodom, kako ćete se izboriti s deliocem koji je nula. Međutim, ako je to neočekivana vrednost, s njom ne možete da se izborite i zbog toga morate da generišete izuzetak umesto da nastavite s tom putanjom izvršavanja.

Kada nastane izuzetak, dešava se nekoliko stvari odjednom. Prvo, pravi se objekat izuzetka na isti način kao svi drugi Javini objekti: u dinamičkoj memoriji, operatorom **new**. Potom se aktuelna putanja izvršavanja (ona koja se ne može nastaviti) zaustavlja, a referenca na objekat izuzetka izbacuje se iz trenutnog okruženja. U tom trenutku, kontrolu izvršavanja preuzima mehanizam obrade izuzetaka i počinje da traži odgovarajuće mesto s kog bi se nastavilo izvršavanje programa. To odgovarajuće mesto je *blok za obradu izuzetaka* (engl. *exception handler*), čiji je zadatak da reši problem primenom drugačijeg pristupa problemu, ili da prosto nastavi izvršavanje od neke druge tačke.

Kao jednostavan primer generisanja izuzetka, posmatrajmo referencu na objekat koja se zove **t**. Može se desiti da vam bude prosleđena referenca koja nije inicijalizovana, što ćete hteti da proverite pre nego što preko te reference pokušate da pozovete neku metodu. Možete da pošaljete informacije o grešci u više okruženje ako napravite objekat koji predstavlja te informacije i zatim ga „izbacite" iz tekućeg konteksta. To se zove *generisanje izuzetka* (engl. *throwing an exception*). Evo kako izgleda:

```
if(t == null)
  throw new NullPointerException();
```

Ovim se generiše izuzetak koji u tekućem okruženju omogućuje odbacivanje odgovornosti za razmatranje ove teme. Problem se, nekim čudom, razmatra na nekom drugom mestu. Koje je to *mesto*, biće uskoro pokazano.

Izuzeci omogućavaju da sve što radite smatrate transakcijom, a izuzeci čuvaju te transakcije: „...osnovna premisa transakcija jeste da u distribuiranom izračunavanju moramo imati obradu izuzetaka. Transakcije su računarski ekvivalent ugovornog prava. Ako bilo šta pođe po zlu, prosto ćemo odbaciti sve to izračunavanje."[2] Izuzetke možete smatrati i ugrađenim sistemom za poništavanje, pošto (uz malo pažnje) u programu možete imati više mesta oporavka. Ako jedan deo programa zakaže, izuzetak će poništiti taj neuspeh i vratiti se na poznatu stabilnu tačku u programu.

Jedan od najvažnijih aspekata izuzetaka jeste sledeći: ukoliko se dogodi nešto loše, oni programu ne dozvoljavaju da nastavi izvršavanje svojom uobičajenom putanjom. U jezicima kao što su C i C++, to je zaista bio problem; naročito u C-u, gde se ne može primorati program da prekine sa izvršavanjem kada nastane problem, pa je probleme bilo moguće dugo ignorisati i tako zapasti u potpuno neodgovarajuće stanje. Ako ništa drugo, izuzeci omogućavaju da prisilite program da se zaustavi i saopšti šta ne valja, ili (u idealnom slučaju) da naterate program da reši problem i vrati se u određeno stabilno stanje.

Argumenti izuzetka

Poput bilo kog Javinog objekta, izuzeci se uvek prave u dinamičkoj memoriji pomoću operatora **new**. Taj operator zauzima prostor u memoriji i poziva konstruktor. U svim standardnim izuzecima postoje dva konstruktora: jedan je podrazumevani, a drugi ima argument tipa znakovnog niza u koji se mogu smestiti važne informacije o izuzetku:

```
throw new NullPointerException("t = null");
```

Znakovni niz se kasnije može izdvojiti pomoću raznih metoda, kao što će uskoro biti pokazano.

Rezervisana reč **throw** prouzrokuje brojna zanimljiva dešavanja. Nakon što operatorom **new** napravite objekat koji predstavlja izuzetak, dobijenu referencu ćete proslediti rezervisanoj reči **throw**. Time se postiže da se taj objekat vrati iz metode, iako ona obično nije projektovana da vraća taj tip objekta. Obradu izuzetaka pojednostavljeno možete shvatiti kao drugačiji mehanizam vraćanja rezultata, mada ovu analogiju ne bi trebalo da shvatate suviše doslovno. I obični programski blokovi mogu se prekinuti generisanjem izuzetka. U svakom slučaju, vraća se objekat izuzetka, a metoda ili programski blok se napuštaju.

Svaka sličnost sa običnim povratkom iz metode tu se i završava, pošto se mesto povratka potpuno razlikuje od mesta povratka iz uobičajenog poziva metode. (Obrešćete se u odgovarajućem bloku za obradu izuzetaka koji na steku s pozivima procedura može da bude udaljen više nivoa od mesta gde je izuzetak nastao.)

Pored toga, možete da generišete proizvoljan objekat tipa **Throwable**, što je korenska klasa svih izuzetaka. Obično ćete generisati drugačiju klasu izuzetaka za svaki tip greške.

---

[2] Izjavio Džim Grej, dobitnik Tjuringove nagrade za doprinos njegovog tima u oblasti transakcija, u razgovoru objavljenom na *www.acmqueue.org*.

Informacije o grešci su predstavljene i unutar objekta izuzetka i implicitno u imenu klase izuzetka, da bi neko u višem okruženju shvatio šta da radi s vašim izuzetkom. (Tip izuzetka je često jedina informacija, pošto se unutar objekta izuzetka retko čuva nešto korisno.)

Hvatanje izuzetka

Da biste videli kako se hvata izuzetak, prvo morate da razumete pojam *čuvane oblasti* (engl. *guarded region*). To je deo koda koji može da generiše izuzetke, iza koga sledi kôd koji obrađuje te izuzetke.

Blok **try**

Ako se nalazite unutar metode i generišete izuzetak (ili neka druga metoda koju pozivate iz te metode generiše izuzetak), izuzetak će prouzrokovati napuštanje metode. Ako ne želite da **throw** izazove izlazak iz metode, možete da uvedete specijalan blok unutar te metode koji će hvatati izuzetak. To se zove *ispitni blok* (engl. *try block*), zato što isprobavate razne pozive metoda. Ispitni blok je običan programski blok ispred koga se nalazi rezervisana reč **try**:

```
try {
  // Kôd koji može da generiše izuzetke
}
```

Kada ste pažljivo tražili greške u programskom jeziku koji ne podržava obradu izuzetaka, morali ste svaki poziv metode da okružite kodom za pripremu i ispitivanje grešaka, čak i ako ste istu metodu pozivali nekoliko puta. Uz postupak obrade grešaka koji se koristi u Javi, sve što treba ispitati smešta se u ispitni blok, a svi izuzeci se hvataju na istom mestu. To znači da se kôd mnogo lakše piše i čita, jer se njegova svrha ne prepliće s proverom grešaka.

Blokovi za obradu izuzetaka

Generisani izuzetak mora negde da se završi. Završiće se u *bloku za obradu izuzetaka* (engl. *exception handler*). Takav blok postoji za svaki tip izuzetka koji želite da obradite. Blokovi za obradu izuzetaka slede odmah nakon ispitnog bloka i označeni su rezervisanom reči **catch**:

```
try {
  // Kôd koji može da generiše izuzetke
} catch(Tip1 id1) {
  // Obrada izuzetaka tipa 1
} catch(Tip2 id2)
  // Obrada izuzetaka tipa 2
} catch(Tip3 id3) {
  // Obrada izuzetaka tipa 3
}
// itd...
```

Svaka odrednica **catch** (blok za obradu izuzetaka) nalik je maloj metodi koja prihvata samo jedan argument određenog tipa. Identifikator (**id1**, **id2** itd.) može se koristiti unutar procedure, poput argumenta metode. Identifikator se ponekad ne koristi jer tip izuzetka pruža dovoljno informacija za obradu, ali argument ipak mora da postoji.

Blokovi za obradu izuzetaka moraju da slede odmah posle ispitnog bloka. Ako nastane izuzetak, mehanizam njegove obrade traži prvi blok čiji argument odgovara tipu izuzetka. Potom se izvršava taj blok, nakon čega se smatra da je izuzetak obrađen. Traženje blokova za obradu izuzetaka zaustavlja se po izvršavanju tog bloka. Izvršava se samo prvi odgovarajući blok; postupak se razlikuje od naredbe **Switch** u kojoj je nakon svake odredbe **case** potrebna naredba **break** da bi se sprečilo izvršavanje ostalih grana.

Obratite pažnju na to da unutar ispitnog bloka nekoliko poziva različitih metoda može da generiše isti izuzetak, ali vam za obradu svih treba samo jedan blok.

Prekidanje ili nastavljanje?

Postoje dva osnovna modela u teoriji obrade izuzetaka. Java podržava model *prekidanja* (engl. *termination*),[3] gde se pretpostavlja da je greška toliko velika da nema načina vratiti se na mesto gde se desio izuzetak. Onaj koji je generisao izuzetak odlučio je da ne postoji način za izlazak iz te situacije, i da *neće* da se vraća nazad.

Alternativa se naziva *nastavljanje* (engl. *resumption*). To znači da se od bloka za obradu izuzetaka očekuje da uradi nešto kako bi rešio situaciju, nakon čega se ponovo pokušava s izvršavanjem neispravne metode pod pretpostavkom da će ponovljeno izvršavanje biti uspešno. Ako koristite ovaj model, znači da se i dalje nadate kako ćete nastaviti izvršavanje nakon obrade izuzetka.

Ukoliko hoćete da nastavite izvršavanje, nemojte generisati izuzetak kada naiđete na grešku, već pozovite metodu koja rešava problem. Drugi način je da se ispitni blok smesti unutar petlje **while** koja ulazi u taj blok sve dok rezultat ne postane zadovoljavajući.

Istorijski gledano, čak su i programeri koji su koristili model nastavljanja na kraju odustajali od toga i počeli da koriste isključivo model prekidanja. Iako nastavljanje na prvi pogled izgleda privlačno, nije toliko korisno u praksi. Najvažniji razlog verovatno je *povezanost* koja nastaje: nastavljački blok za obradu mora da zna gde je izuzetak nastao i mora sadržati negenerički kôd specifičan za mesto pojavljivanja izuzetka. To otežava pisanje i održavanje koda, naročito u velikim sistemima gde se izuzetak može pojaviti na mnogim mestima.

Pravljenje sopstvenih izuzetaka

Niste primorani da koristite samo postojeće Javine izuzetke. Projektanti hijerarhije izuzetaka u Javi nisu mogli da predvide sve greške koje biste mogli hteti da prijavite, pa možete da pravite sopstvene izuzetke koji će naznačiti posebne probleme na koje vaša biblioteka može da naiđe.

---

[3] Kao i većina jezika, među kojima su C++, C#, Python, D itd.

Vašu klasu izuzetka morate da izvedete iz neke od postojećih klasa izuzetaka, ako je moguće iz one čije je značenje blisko smislu novog izuzetka (mada to često nije moguće). Najprostiji način za pisanje novog tipa izuzetka jeste prepustiti prevodiocu da napravi podrazumevani konstruktor, što gotovo da ne zahteva pisanje koda:

```
//: izuzeci/NasleđivanjeIzuzetaka.java
// Pravljenje sopstvenih izuzetaka.

class JednostavanIzuzetak extends Exception {}

public class NasleđivanjeIzuzetaka {
  public void f() throws JednostavanIzuzetak {
    System.out.println("Bacam JednostavanIzuzetak iz f()");
    throw new JednostavanIzuzetak ();
  }
  public static void main(String[] args) {
    NasleđivanjeIzuzetaka sed = new NasleđivanjeIzuzetaka();
    try {
      sed.f();
    } catch(JednostavanIzuzetak e) {
      System.err.println("Uhvatio sam ga!");
    }
  }
} /* Ispis:
Bacam JednostavanIzuzetak iz f()
Uhvatio sam ga!
*///:~
```

Kada prevodilac napravi podrazumevani konstruktor, on će automatski (i nevidljivo) pozvati podrazumevani konstruktor osnovne klase. Naravno, u ovom slučaju nećete dobiti konstruktor **JednostavanIzuzetak(String)**, ali on se u praksi retko koristi. Kao što ćete videti, najvažnija osobina izuzetka je ime klase, pa je u najvećem broju slučajeva izuzetak poput prikazanog sasvim zadovoljavajući.

Ovde se rezultat štampa na konzoli, gde ga automatski hvata i ispituje sistem ove knjige za ispisivanje rezultata. Međutim, možda ćete grešku hteti da pošaljete u standardni izlazni tok za greške, tako što ćete je upisati u tok podataka **System.err**. To je, obično, bolje mesto za slanje informacija o grešci od toka **System.out** koji može biti preusmeren. Ako rezultate šaljete u **System.err**, oni neće biti preusmereni zajedno s podacima iz toka **System.out**, pa je veća verovatnoća da će ih korisnik primetiti.

Možete napraviti i klasu izuzetaka s konstruktorom koji ima argument tipa **String**:

```
//: izuzeci/PotpuniKonstruktori.java

class MojIzuzetak extends Exception {
  public MojIzuzetak() {}
  public MojIzuzetak(String poruka) {  super(poruka);  }
}
```

```
public class PotpuniKonstruktori {
  public static void f() throws MojIzuzetak {
    System.out.println("Izbacujem MojIzuzetak iz metode f()");
    throw new MojIzuzetak();
  }
  public static void g() throws MojIzuzetak {
    System.out.println("Izbacujem MojIzuzetak iz metode g()");
    throw new MojIzuzetak("Nastao u metodi g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MojIzuzetak e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MojIzuzetak e) {
      e.printStackTrace(System.out);
    }
  }
} /* Ispis:
Izbacujem MojIzuzetak iz metode f()
MojIzuzetak
        at PotpuniKonstruktori.f(PotpuniKonstruktori.java:11)
        at PotpuniKonstruktori.main(PotpuniKonstruktori.java:19)
Izbacujem MojIzuzetak iz metode g()
MojIzuzetak: Nastao u metodi g()
        at PotpuniKonstruktori.g(PotpuniKonstruktori.java:15)
        at PotpuniKonstruktori.main(PotpuniKonstruktori.java:24)
*///:~
```

Dodati kod je kratak i sadrži samo dva konstruktora koji definišu način pravljenja izuzetka tipa **MojIzuzetak**. U drugom konstruktoru se, pomoću rezervisane reči **super**, eksplicitno poziva konstruktor osnovne klase sa argumentom tipa **String**.

U blokovima za obradu izuzetaka poziva se jedna od **Throwable** metoda: **printStackTrace( )**. Kao što vidite iz rezultata, time se dobijaju informacije o nizu metoda koje su bile pozivane pre dolaska na mesto gde se desio izuzetak. Ovde se informacije o položaju na steku izvršavanja šalju u tok podataka **System.out** i automatski hvataju i ispisuju na izlazu.

Medutim, ako pozovete podrazumevanu verziju:

```
e.printStackTrace( );
```

informacije odlaze u standardni izlazni tok za greške.

Vežba 1: (2) Napravite klasu s metodom **main( )** koja generiše izuzetak klase **Exception** unutar bloka **try**. Dodelite konstruktoru za **Exception** argument tipa **String**. Uhvatite izuzetak u odredbi **catch** i odštampajte argument tipa **String**. Dodajte odredbu **finally** i odštampajte poruku da biste dokazali da ste bili gde treba.

Vežba 2: (1) Definišite referencu na objekat i inicijalizujte je vrednošću **null**. Pokušajte da pozovete metodu pomoću te reference. Sada smestite kôd u blok **try--catch** da biste uhvatili izuzetak.

Vežba 3: (1) Napišite kôd za generisanje i hvatanje izuzetka tipa **ArrayIndexOutOfBoundsException** (indeks niza izvan dozvoljenih granica).

Vežba 4: (2) Napravite sopstvenu klasu izuzetaka korišćenjem rezervisane reči **extends**. Napišite konstruktor za tu klasu koji prihvata argument tipa **String** i smešta ga u objekat s referencom tipa **String**. Napišite metodu koja štampa sačuvani znakovni niz. Napravite blok **try-catch** da biste isprobali nov izuzetak.

Vežba 5: (3) Napravite obradu izuzetka po modelu nastavljanja. Upotrebite petlju **while** koja se ponavlja sve dok ne prestane generisanje izuzetka.

Izuzeci i zapisivanje

Izlaz biste mogli i da zapišete (engl. *log*) pomoću metoda klase **java.util.logging**. Iako je zapisivanje detaljno objašnjeno u dodatku na lokaciji *http://MindView.net/Books/BetterJava*, elementarno zapisivanje je toliko jednostavno da ga već ovde možemo upotrebiti.

```
//: izuzeci/ZapisivanjeIzuzetaka.java
// Izuzetak koji se prijavljuje preko Zapisnika.
import java.util.logging.*;
import java.io.*;

class ZapisivanjeIzuzetaka extends Exception {
  private static Logger zapisnik =
    Logger.getLogger("ZapisivanjeIzuzetka");
  public ZapisivanjeIzuzetka() {
    StringWriter trag = new StringWriter();
    printStackTrace(new PrintWriter(trag));
    zapisnik.severe(trag.toString());
  }
}

public class ZapisivanjeIzuzetaka {
  public static void main(String[] args) {
    try {
      throw new ZapisivanjeIzuzetka();
    } catch(ZapisivanjeIzuzetka e) {
      System.err.println("Uhvatio " + e);
    }
    try {
      throw new ZapisivanjeIzuzetka();
    } catch(ZapisivanjeIzuzetka e) {
      System.err.println("Uhvatio " + e);
    }
  }
} /* Ispis: (85% podudarnih podataka)
Aug 30, 2005 4:02:31 PM ZapisivanjeIzuzetka <init>
```

```
SEVERE: ZapisivanjeIzuzetka
        at ZapisivanjeIzuzetaka.main(ZapisivanjeIzuzetaka.java:19)

Uhvatio LoggingException
Aug 30, 2005 4:02:31 PM ZapisivanjeIzuzetka <init>
SEVERE: ZapisivanjeIzuzetka
        at ZapisivanjeIzuzetaka.main(ZapisivanjeIzuzetaka.java:24)

Uhvatio ZapisivanjeIzuzetka
*///:~
```

Statična metoda **Logger.getLogger( )** pravi zapisnik (engl. *logger*), tj. objekat tipa **Logger** sa argumentom tipa **String** (obično imenom paketa i klase u kojima je nastala greška) koji svoj izlaz šalje u **System.err**. U zapisnik se najlakše upisuje tako da se pozove metoda pridružena nivou poruke koju treba zapisati; ovde je upotrebljena metoda **severe( )**. Za pravljenje znakovnog niza poruke za zapisnik, hteli bismo da upotrebimo ispis steka (engl. *stack trace*) na mestu gde je izuzetak bačen, ali metoda **printStackTrace( )** podrazumevano ne daje **String**. Da bismo dobili **String**, moramo upotrebiti preklopljenu **printStackTrace( )** koja kao argument prima objekat **java.io.PrintWriter**. (Sve će to biti detaljno objašnjeno u poglavlju *Javin ulazno-izlazni sistem.*) Ukoliko **PrintWriter** konstruktoru predamo objekat tipa **java.io.StringWriter**, izlaz se može pretvoriti u **String** metodom **toString( )**.

Iako je pristup upotrebljen u klasi **ZapisivanjeIzuzetka** veoma komforan, zato što svu infrastrukturu zapisivanja ugrađuje u sam izuzetak i stoga radi automatski bez ikakve intervencije programera klijenta, češće se dešava da hvatate i zapisujete tuđe izuzetke, pa poruku za zapisnik morate da generišete u bloku za obradu izuzetaka:

```
//: izuzeci/ZapisivanjeIzuzetaka2.java
// Zapisivanje uhvaćenih izuzetaka.
import java.util.logging.*;
import java.io.*;

public class ZapisivanjeIzuzetaka2 {
  private static Logger zapisnik =
    Logger.getLogger("ZapisivanjeIzuzetaka2");
  static void zapisiIzuzetak(Exception e) {
    StringWriter trag = new StringWriter();
    e.printStackTrace(new PrintWriter(trag));
    zapisnik.severe(trag.toString());
  }
  public static void main(String[] args) {
    try {
      throw new NullPointerException();
    } catch(NullPointerException e) {
      zapisiIzuzetak(e);
    }
  }
} /* Ispis: (90% podudarnih podataka)
Aug 30, 2005 4:07:54 PM ZapisivanjeIzuzetaka2 zapisiIzuzetak
```

```
SEVERE: java.lang.NullPointerException
        at ZapisivanjeIzuzetaka2.main(ZapisivanjeIzuzetaka2.java:16)
*///:~
```

Postupak pravljenja sopstvenog izuzetka može se proširiti. Možete da dodate sopstvene konstruktore i članove:

```
//: izuzeci/DodatniElementi.java
// Dodatno razvijanje klasa izuzetaka.
import static net.mindview.util.Print.*;

class MojIzuzetak2 extends Exception {
  private int x;
  public MojIzuzetak2() {}
  public MojIzuzetak2(String poruka) {  super(poruka);  }
  public MojIzuzetak2(String poruka, int x) {
    super(poruka);
    this.x = x;
  }
  public int vrednost() { return x; }
  public String getMessage()  {
     return "Detaljna poruka: " + x + " "+ super. getMessage();
  }
}

public class DodatniElementi {
  public static void f() throws MojIzuzetak2 {
    print("Izbacujem MojIzuzetak2 iz metode f()");
    throw new MojIzuzetak2();
  }
  public static void g() throws MojIzuzetak2 {
    print("Izbacujem MojIzuzetak2 iz metode g()");
    throw new MojIzuzetak2("Nastao u metodi g()");
  }
  public static void h() throws MojIzuzetak2 {
    print("Izbacujem MojIzuzetak2 iz metode h()");
    throw new MojIzuzetak2("Nastao u metodi h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MojIzuzetak2 e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MojIzuzetak2 e) {
      e.printStackTrace(System.out);
    }
    try {
      h();
```

```
      } catch(MojIzuzetak2 e) {
        e.printStackTrace(System.out);
        System.err.println("e.vrednost() = " + e.vrednost());
      }
    }
} /* Ispis:
Izbacujem MojIzuzetak2 iz metode f()
MojIzuzetak2: Detaljna poruka: 0 null
        at DodatniElementi.f(DodatniElementi.java:22)
        at DodatniElementi.main(DodatniElementi.java:34)
Izbacujem MojIzuzetak2 iz metode g()
MojIzuzetak2: Detaljna poruka: 0 Nastao u metodi g()
        at DodatniElementi.g(DodatniElementi.java:26)
        at DodatniElementi.main(DodatniElementi.java:39)
Izbacujem MojIzuzetak2 iz metode h()
MojIzuzetak2: Detaljna poruka: 47 Nastao u metodi h()
        at DodatniElementi.h(DodatniElementi.java:30)
        at DodatniElementi.main(DodatniElementi.java:44)
e.vrednost() = 47
*///:~
```

Klasi izuzetka dodato je polje **x**, uz metodu koja čita vrednost i dodatni konstruktor koji je postavlja. Pored toga, redefinisana je metoda **Throwable.getMessage( )** da bi se dobila zanimljivija detaljna poruka. Metoda **getMessage( )** liči na metodu **toString( )** za klase izuzetaka.

Pošto je izuzetak samo vrsta objekta, postupak proširenja klasa izuzetaka možete još da nastavite. Ipak, imajte u vidu da programeri klijenti koji koriste vaše pakete možda neće ni primetiti ulepšavanje, pošto će verovatno očekivati generisanje izuzetka i ništa više. (Tako se koristi većina izuzetaka u Javinim bibliotekama.)

Vežba 6: (1) Napravite dve nove klase izuzetaka koje same automatski zapisuju svoje izuzetke. Dokažite da one rade.

Vežba 7: (1) Izmenite vežbu 3 tako da odredba **catch** zapisuje izuzetke.

Specifikacija izuzetaka

U Javi bi trebalo da obavestite programera klijenta koji poziva vašu metodu o izuzecima koje ona može da generiše. To je učtivo, pošto onaj koji poziva metodu tada tačno zna kako da uhvati sve moguće izuzetke. Naravno, ako je dostupan izvorni kôd, programer klijent bi mogao da ga pregleda i pronađe sve naredbe **throw**, ali se biblioteke često ne isporučuju sa izvornim kodom. Da bi se sprečilo pojavljivanje problema, Java obezbeđuje sintaksu (i *primorava* vas da je koristite) koja omogućuje da učtivo kažete programeru klijentu koje izuzetke generiše određena metoda. To je *specifikacija izuzetaka* i pojavljuje se nakon spiska argumenata, u deklaraciji metode.

Specifikacija izuzetaka koristi dodatnu rezervisanu reč **throws**, iza koje sledi spisak svih mogućih tipova izuzetaka. Definicija vaše metode mogla bi da izgleda ovako:

```
void f() throws Preveliki, Premali, DeljenjeNulom { //...
```

Međutim, ako napišete:

```
void f() { //...
```

to znači da metoda ne generiše nikakve izuzetke (*sem* izuzetaka nasledenih od klase **RuntimeException** koji se i bez specifikacije izuzetka mogu generisati bilo gde, što će biti opisano kasnije).

Ne možete da lažete u specifikaciji izuzetka. Ukoliko kôd unutar vaše metode prouzrokuje izuzetke, a metoda ih ne obradi, prevodilac će to otkriti i saopštiće vam da morate obraditi izuzetak ili specifikacijom izuzetka naznačiti da ga vaša metoda može generisati. Podržavanjem specifikacija izuzetaka od dna do vrha, Java jemči da će određen nivo ispravnosti izuzetaka biti garantovan tokom *prevođenja*.

O jednoj stvari možete lagati: možete tvrditi da se izuzetak generiše, iako se to zapravo ne dešava. Prevodilac vam veruje i primorava korisnike da s vašom metodom rade kao da ona zaista generiše taj izuzetak. Time se postiže utisak da je metoda mesto nastanka izuzetka, pa kasnije zaista možete da generišete izuzetak ne menjajući postojeći kôd. To je važno i za pravljenje **apstraktnih** osnovnih klasa i **interfejsa** čije će izvedene klase ili realizacije možda generisati izuzetke.

Izuzeci koji se proveravaju i nameću u vreme prevođenja, nazivaju se *provereni izuzeci*.

Vežba 8: (1) Napišite klasu s metodom koja generiše izuzetak tipa definisanog u vežbi 4. Pokušajte da je prevedete bez specifikacije izuzetka i pogledajte šta će vam reći prevodilac. Dodajte odgovarajuću specifikaciju izuzetka. Isprobajte svoju klasu i njen izuzetak u odredbi **try-catch**.

Hvatanje bilo kog izuzetka

Moguće je napraviti blok koji hvata bilo koji tip izuzetka. To se radi hvatanjem osnovnog tipa izuzetka klase **Exception** (postoje i drugi tipovi osnovnih izuzetaka, ali je osnova **Exception** primenljiva na skoro sve programerske aktivnosti):

```
catch(Exception e) {
  System.out.println("Uhvaćen izuzetak");
}
```

Na ovaj način biće uhvaćen svaki izuzetak, pa bi ovakav blok trebalo staviti na *kraj* liste za obradu. Time se izbegava preuzimanje nadležnosti od blokova za obradu konkretnih izuzetaka, koji bi eventualno mogli da slede.

Pošto je klasa **Exception** osnova svih klasa izuzetaka koje su programeru važne, ovim postupkom ne dobijate mnogo određenijih informacija o izuzetku, ali možete da pozovete metode *njene* natklase **Throwable**:

String getMessage( )
String getLocalizedMessage( )

One daju detaljnu poruku ili poruku koja je prilagođena određenom lokalnom geografskom području.

String toString( )

Vraća kratak opis objekta, uključujući i detaljnu poruku ako postoji.

void printStackTrace( )
void printStackTrace(PrintStream)
void printStackTrace(java.io.PrintWriter)

Štampaju opis objekta i položaj nastanka izuzetka na steku izvršavanja. Stek izvršavanja prikazuje redosled poziva metoda koje su dovele do mesta gde je nastao izuzetak. Prva verzija koristi standardan izlazni tok za greške, a druga i treća omogućuju da izaberete tok podataka (u poglavlju *Javin ulazno-izlazni sistem* objasnićemo zašto su podržane dve vrste toka).

Throwable fillInStackTrace( )

Informacije o tekućem stanju slojeva steka beleži u objektu tipa **Throwable**. Ovo je korisna funkcija u slučajevima kada aplikacija ponovo izbacuje postojeću grešku ili izuzetak (uskoro će biti više reči o ovome).

Pored toga, iz tipa **Object** koji je osnovni tip svih objekata, pa i objekata tipa **Throwable**, dobijaju se i neke druge metode. Jedna koja bi mogla da bude korisna za izuzetke jeste **getClass( )**, čija je vrednost objekat koji predstavlja klasu datog objekta. Onda možete da ispitate objekat tipa **Class** i pomoću metode **getName( )** saznate ime klase i paketa, ili metodom **getSimpleName( )** saznate samo ime klase.

Evo primera koji ilustruje korišćenje osnovnih metoda klase **Exception**:

```
//: izuzeci/MetodeKlaseException.java
// Prikaz metoda klase MetodeKlaseException.
import static net.mindview.util.Print.*;

public class MetodeKlaseException {
  public static void main(String[] args) {
    try {
      throw new Exception("Evo izuzetka");
    } catch(Exception e) {
      print("Uhvatio sam izuzetak");
      print("getMessage(): " + e.getMessage());
      print("getLocalizedMessage(): " +
         e.getLocalizedMessage());
      print("toString(): " + e);
      print("printStackTrace():");
      e.printStackTrace(System.out);
    }
  }
}/* Ispis:
Uhvatio sam izuzetak
getMessage( ): Evo izuzetka
getLocalizedMessage(): Evo izuzetka
toString(): java.lang.Exception: Evo izuzetka
printStackTrace():
java.lang.Exception: Evo izuzetka
        at MetodeKlaseException.main(MetodeKlaseException.java:8)
* ///:~
```

Vidite da metode redom obezbeđuju sve više informacija, jer je svaka nadskup prethodne.

Vežba 9: (2) Napravite tri nova tipa izuzetaka. Napravite klasu s metodom koja generiše sva tri tipa izuzetaka. U metodi **main( )** pozovite metodu, ali upotrebite samo jednu odredbu **catch** koja će uhvatiti izuzetke sva tri tipa.

Položaj nastanka izuzetka na steku izvršavanja

Informacijama koje daje **printStackTrace( )** može se pristupiti i direktno, metodom **getStackTrace( )**. Ona vraća niz elemenata steka, od kojih svaki predstavlja po jedan njegov sloj. Element broj nula je na vrhu steka i predstavlja poslednji poziv metode u tom nizu (tačku u kojoj je ovaj objekat izuzetka napravljen i bačen). Poslednji element niza, onaj koji je na dnu steka, jeste prvi poziv metode u tom nizu. Naredni program daje prostu ilustraciju:

```
//: izuzeci/KoJeZvao.java
// Programski pristup podacima o položaju nastanka izuzetka
// na steku izvršavanja.

public class KoJeZvao {
  static void f() {
    // Generiši izuzetak da bi na stek izvršavanja upisao njegov
    // položaj nastanka
    try {
      throw new Exception();
    } catch (Exception e) {
   for(StackTraceElement ste : e.getStackTrace())
        System.out.println(ste.getMethodName());
    }
  }
  static void g() { f(); }
  static void h() { g(); }
  public static void main(String[] args) {
    f();
    System.out.println("--------------------------------");
    g();
    System.out.println("--------------------------------");
    h();
  }
} /* Ispis:
f
main
--------------------------------
f
g
main
--------------------------------
f
g
```

```
h
main
*///:~
```

Ovde smo ispisali samo ime metode, ali vi možete da ispišete ceo **StackTraceElement** koji sadrži dodatne podatke.

Ponovno generisanje izuzetka

Ponekad ćete poželeti da ponovo generišete izuzetak koji ste upravo uhvatili, naročito ako koristite klasu **Exception** za hvatanje svih izuzetaka. Pošto već imate referencu na aktuelni izuzetak, samo je ponovo izbacite:

```
catch (Exception e) {
  System.out.println("Generisan je izuzetak");
  throw e;
}
```

Ponovnim generisanjem, izuzetak se prosleduje blokovima za obradu izuzetaka u sledećem nivou okruženja. Sve kasnije naredbe **catch** za isti blok **try** i dalje se zanemaruju. Pored toga, čuvaju se svi podaci o izuzetku, pa blok u višem okruženju koji hvata određeni tip izuzetka može da ih izdvoji iz tog objekta.

Ako ponovo generišete tekući izuzetak, informacije koje štampate o njemu u metodi **printStackTrace**( ) pripadaće mestu porekla izuzetka, a ne mestu na kojem ga ponovo generišete. Nove informacije o stanju steka možete da upišete pozivanjem metode **fillInStackTrace**( ) koja vraća objekat izuzetka, napravljen tako što su informacije o tekućem stanju steka stavljene u stari objekat izuzetka. Evo kako to izgleda:

```
//: izuzeci/Ponovnogenerisanje.java
// Prikaz metode fillInStackTrace()

public class Ponovnogenerisanje {
  public static void f() throws Exception {
    System.out.println("Pravim izuzetak u metodi f()");
    throw new Exception("Izbacen iz f()");
  }
  public static void g() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.err.println("U metodi g(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw e;
    }
  }
public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
```

```
        System.err.println("U metodi h(), e.printStackTrace()");
        e.printStackTrace(System.out);
        throw (Exception)e.fillInStackTrace();
      }
    }
    public static void main(String[] args)  {
      try {
        g();
      } catch(Exception e) {
        System.out.println("Uhvatio izuzetak u metodi main:
           printStackTrace()");
        e.printStackTrace(System.out);
      }
      try {
        h();
      } catch(Exception e) {
        System.out.println("Uhvatio izuzetak u metodi main:
           printStackTrace()");
        e.printStackTrace(System.out);
      }
    }
} /* Ispis:
Pravim izuzetak u metodi f()
U metodi g(), e.printStackTrace()
java.lang.Exception: Izbacen iz f()
        at Ponovnogenerisanje.f(Ponovnogenerisanje.java:7)
        at Ponovnogenerisanje.g(Ponovnogenerisanje.java:11)
        at Ponovnogenerisanje.main(Ponovnogenerisanje.java:29)
Uhvatio izuzetak u metodi main: printStackTrace()
java.lang.Exception: izbacen iz f()
        at Ponovnogenerisanje.f(Ponovnogenerisanje.java:7)
        at Ponovnogenerisanje.g(Ponovnogenerisanje.java:11)
        at Ponovnogenerisanje.main(Ponovnogenerisanje.java:29)
Pravim izuzetak u metodi f()
U metodi h(), e.printStackTrace()
java.lang.Exception: Izbacen iz f()
        at Ponovnogenerisanje.f(Ponovnogenerisanje.java:7)
        at Ponovnogenerisanje.h(Ponovnogenerisanje.java:20)
        at Ponovnogenerisanje.main(Ponovnogenerisanje.java:35)
Uhvatio izuzetak u metodi main, printStackTrace()
java.lang.Exception: Izbacen iz f()
        at Ponovnogenerisanje.h(Ponovnogenerisanje.java:24)
        at Ponovnogenerisanje.main(Ponovnogenerisanje.java:35)
*///:~
```

Red gde se poziva metoda **fillInStackTrace**( ) postaje novo mesto porekla izuzetka.

Moguće je ponovo generisati izuzetak koji je različit od onog koji je uhvaćen. To prouzrokuje isti efekat kao korišćenje metode **fillInStackTrace**( ): gube se informacije o izvornom poreklu izuzetka, a preostaju informacije koje pripadaju novom generisanju:

```
//: izuzeci/PonovnogenerisanjeNovog.java
// Ponovno generisanje objekta koji se razlikuje od uhvaćenog.

class IzuzetakJedan extends Exception {
  public IzuzetakJedan(String s) { super(s); }
}

class IzuzetakDva extends Exception {
  public IzuzetakDva(String s) { super(s); }
}

public class PonovnogenerisanjeNovog {
  public static void f() throws IzuzetakJedan {
    System.out.println("Pravim izuzetak u metodi f()");
    throw new IzuzetakJedan("izbačen iz f()");
  }
  public static void main(String[] args)  {
    try {
      try {
        f();
      } catch(IzuzetakJedan e) {
         System.out.println(
            "Uhvaćen u unutrašnjem bloku try, e.printStackTrace()");
         e.printStackTrace(System.out);
         throw new IzuzetakDva("iz unutrašnjeg bloka try");
    }
  } catch(IzuzetakDva e) {
         System.out.println(
            "Uhvaćen u spoljnom bloku try, e.printStackTrace()");
         e.printStackTrace(System.out);
    }
  }
} /* Ispis:
Pravim izuzetak u metodi f()
Uhvaćen u unutrašnjem bloku try, e.printStackTrace()
IzuzetakJedan: izbačen iz f()
        at PonovnogenerisanjeNovog.f(PonovnogenerisanjeNovog.java:15)
        at PonovnogenerisanjeNovog.main(PonovnogenerisanjeNovog.java:20)
Uhvaćen u spoljnom bloku try, e.printStackTrace()
IzuzetakDva: iz unutrašnjeg bloka try
        at PonovnogenerisanjeNovog.main(PonovnogenerisanjeNovog.java:25)
*///:~
```

Konačan izuzetak zna samo da je stigao iz unutrašnjeg bloka **try**, a ne iz **f( )**.

Ne morate uopšte da brinete o brisanju prethodnog izuzetka ili bilo kog drugog izuzetka. Ti su objekti napravljeni operatorom **new** u dinamičkoj memoriji, pa ih sakupljač smeća automatski briše.

Ulančavanje izuzetaka

Često je potrebno uhvatiti jedan izuzetak i generisati drugi, ne izgubivši podatke o prvom – to se naziva *ulančavanje izuzetaka*. Pre objavljivanja JDK-a 1.4, programeri su sami morali da pišu kôd koji će očuvati podatke o prvobitnom izuzetku, ali sada sve potklase od **Throwable** imaju opciju da u konstruktoru prime objekat *uzrok* (engl. *cause*). Predviđeno je da *uzrok* bude prvobitni izuzetak, čijim prosleđivanjem zadržavate položaj prvog izuzetka na steku, iako pravite i generišete drugi izuzetak.

Zanimljivo je primetiti da jedine potklase od **Throwable** koje u konstruktoru primaju argument *uzrok* jesu tri osnovne klase izuzetaka: **Error** (pomoću koje JVM prijavljuje sistemske greške), **Exception** i **RuntimeException**. Ako hoćete da ulančite druge tipove izuzetaka, uradite to metodom **initCause( )**, a ne konstruktorom.

Evo primera koji omogućava dinamičko dodavanje polja objektu tipa **DinamickaPolja** u vreme izvršavanja:

```
//: izuzeci/DinamickaPolja.java
// Klasa koja sebi dinamički dodaje polja.
// Ilustruje ulančavanje izuzetaka.
import static net.mindview.util.Print.*;

class IzuzetakDinamickihPolja extends Exception {}

public class DinamickaPolja {
  private Object[][] polja;
  public DinamickaPolja(int pocetnaVelicina) {
    polja = new Object[pocetnaVelicina][2];
    for(int i = 0; i < pocetnaVelicina; i++)
      polja[i] = new Object[] { null, null };
  }
  public String toString() {
    StringBuilder rezultat = new StringBuilder();
    for(Object[] obj : polja) {
      rezultat.append(obj[0]);
      rezultat.append(": ");
      rezultat.append(obj[1]);
      rezultat.append("\n");
    }
    return rezultat.toString();
  }
  private int hasField(String id) {
    for(int i = 0; i < polja.length; i++)
      if(id.equals(polja[i][0]))
        return i;
    return -1;
  }
  private int
  getFieldNumber(String id) throws NoSuchFieldException {
    int brojPolja = hasField(id);
    if(brojPolja == -1)
```

```
      throw new NoSuchFieldException();
    return brojPolja;
  }
  private int makeField(String id) {
    for(int i = 0; i < polja.length; i++)
      if(polja[i][0] == null) {
        polja[i][0] = id;
        return i;
      }
    // Nema praznih polja. Dodaćemo jedno:
    Object[][] tmp = new Object[polja.length + 1][2];
    for(int i = 0; i < polja.length; i++)
      tmp[i] = polja[i];
    for(int i = polja.length; i < tmp.length; i++)
      tmp[i] = new Object[] { null, null };
    polja = tmp;
    // Rekurzivan poziv s proširenim poljima:
    return makeField(id);
  }
  public Object
  getField(String id) throws NoSuchFieldException {
    return polja[getFieldNumber(id)][1];
  }
  public Object setField(String id, Object vrednost)
  throws IzuzetakDinamickihPolja {
    if(vrednost == null) {
      // Većina izuzetaka nema konstruktor koji bi primio argument "cause".
      // U tim slučajevima morate upotrebiti initCause(),
      // dostupnu u svim potklasama od Throwable.
      IzuzetakDinamickihPolja idp =
        new IzuzetakDinamickihPolja();
      idp.initCause(new NullPointerException());
      throw idp;
    }
    int brojPolja = hasField(id);
    if(brojPolja == -1)
      brojPolja = makeField(id);
    Object rezultat = null;
    try {
      rezultat = getField(id); // Uzmi staru vrednost
    } catch(NoSuchFieldException e) {
      // Upotrebi konstruktor koji prima "cause":
      throw new RuntimeException(e);
    }
    polja[brojPolja][1] = vrednost;
    return rezultat;
  }
  public static void main(String[] args) {
    DinamickaPolja dp = new DinamickaPolja(3);
    print(dp);
```

```
      try {
        dp.setField("d", "Neka vrednost za d");
        dp.setField("broj", 47);
        dp.setField("broj2", 48);
        print(dp);
        dp.setField("d", "Nova vrednost za d");
        dp.setField("broj3", 11);
        print("dp: " + dp);
        print("dp.getField(\"d\") : " + dp.getField("d"));
        Object polje = dp.setField("d", null); // Izuzetak
      } catch(NoSuchFieldException e) {
        e.printStackTrace(System.out);
      } catch(IzuzetakDinamickihPolja e) {
        e.printStackTrace(System.out);
      }
    }
} /* Ispis:
null: null
null: null
null: null

d: Neka vrednost za d
broj: 47
broj2: 48

dp: d: Nova vrednost za d
broj: 47
broj2: 48
broj3: 11

dp.getField("d") : Nova vrednost za d
IzuzetakDinamickihPolja
        at DinamickaPolja.setField(DinamickaPolja.java:64)
        at DinamickaPolja.main(DinamickaPolja.java:94)
Caused by: java.lang.NullPointerException
        at DinamickaPolja.setField(DinamickaPolja.java:66)
        ... 1 more
*///:~
```

Svaki objekat tipa **DinamickaPolja** sadrži niz parova **Object-Object**. Prvi objekat je identifikator polja (tipa **String**), a drugi vrednost polja koja može biti bilo kojeg tipa sem prostog (bez omotača). Kada pravite taj objekat, pravite zasnovanu pretpostavku o tome koliko vam polja treba. Čim pozovete **setField( )**, ona će pronaći postojeće, tako nazvano polje ili će napraviti novo, i staviti u njega vašu vrednost. Ukoliko joj ponestane prostora, dodaće ga pravljenjem niza dužeg za jedan i kopiranjem starih elemenata u njega. Ako pokušate da unutra stavite vrednost **null**, generisaće **IzuzetakDinamickihPolja** tako što će ga napraviti i metodom **initCause( )** umetnuti **NullPointerException** kao uzrok.

Kao povratnu vrednost, metoda **setField( )** pribavlja i staru vrednost na mestu tog polja metodom **getField( )** koja može generisati izuzetak **NoSuchFieldException**. Ako

programer klijent pozove **getField( )**, on je odgovoran za obradu izuzetka **NoSuchFieldException**, ali ukoliko taj izuzetak bude generisan unutar metode **setField( )**, to je programska greška, pa se **NoSuchFieldException** pomoću konstruktora koji prima argument *cause* pretvara u **RuntimeException**.

Možda ste primetili da **toString( )** upotrebljava **StringBuilder** za pravljenje svog rezultata. O klasi **StringBuilder** saznaćete više u poglavlju *Znakovni nizovi*, ali zasad je dovoljno znati da je treba upotrebljavati kad god pišete u **toString( )** koja obuhvata kruženje u petlji, kao ovde.

Vežba 10: (2) Napravite klasu s dve metode, **f( )** i **g( )**. U metodi **g( )** generišite izuzetak novog tipa koji ćete definisati. U metodi **f( )** pozovite metodu **g( )**, uhvatite njen izuzetak i u odredbi **catch** generišite drugačiji izuzetak (drugog tipa koji ćete takođe sami definisati). Testirajte kôd u metodi **main( )**.

Vežba 11: (1) Ponovite prethodnu vežbu, ali unutar odredbe **catch** omotajte izuzetak metode **g( )** u **RuntimeException**.

Standardni izuzeci u Javi

Javina klasa **Throwable** opisuje sve što se može generisati kao izuzetak. Postoje dve opšte vrste objekata tipa **Throwable** (vrsta objekta = nasleđivanje iz). **Error** predstavlja sistemske greške i greške tokom prevođenja koje nije neophodno hvatati (osim u specijalnim slučajevima). **Exception** je prost tip izuzetka koji može nastati u bilo kojoj metodi klase standardne Javine biblioteke, kao i u metodama koje sami pravite i akcidentima prilikom izvršavanja. Prema tome, prost tip koji zanima programera u Javi jeste **Exception**.

Najbolji način za pregled izuzetaka jeste čitanje HTML dokumentacije Javinog razvojnog okruženja (JDK). To vredi uraditi samo jednom da bi se stekla predstava o različitim izuzecima, ali brzo ćete shvatiti da se izuzeci ne razlikuju ni po čemu bitnom, osim po imenu. Takođe, broj izuzetaka u Javi stalno raste i besmisleno je da se štampaju u knjizi. Svaka nova biblioteka koju nabavite od nezavisnog autora verovatno će sadržati i sopstvene izuzetke. Važno je razumeti šta su izuzeci i šta se s njima može uraditi.

Osnovna zamisao je da ime izuzetka predstavlja problem koji se desio, tj. da ime bude relativno očigledno. Nisu svi izuzeci definisani u biblioteci **java.lang**; neki su napravljeni radi podrške drugim bibliotekama kao što su **util**, **net** i **io**, što se može zaključiti iz punog imena njihovih klasa ili imena klase iz koje su izvedeni. Na primer, svi ulazno/izlazni izuzeci izvedeni su iz klase **java.io.IOException**.

Specijalan slučaj klase **RuntimeException**

Prvi primer u ovom poglavlju bio je:

```
if(t == null)
  throw new NullPointerException();
```

Bilo bi strašno da morate proveravati svaku referencu koja se prosleđuje metodi, kako bi se utvrdilo da li ima vrednost **null** (pošto ne znate da li je metodi prosleđena ispravna referenca). Srećom, to ne morate da radite, jer je ovaj postupak deo standardne provere

tokom izvršavanja koju Java sprovodi umesto vas. Ako je metoda pozvana preko reference **null**, Java će automatski generisati izuzetak tipa **NullPointerException**. Zbog toga su gornji redovi koda uvek suvišni, iako bi trebalo obaviti druge provere da se ne bi pojavio izuzetak **NullPointerException**.

Postoji čitava grupa tipova izuzetaka koji pripadaju ovoj kategoriji i njih Java uvek generiše automatski, pa ne morate da ih navodite u specifikaciji svojih izuzetaka. Zgodno je i to što su svi oni grupisani unutar iste osnovne klase **RuntimeException** koja predstavlja savršen primer nasleđivanja: uspostavlja porodicu tipova delimično zajedničkih osobina i ponašanja. Takođe, nikada nije potrebno da pišete specifikaciju izuzetka koja će saopštavati da bi metoda mogla generisati izuzetak tipa **RuntimeException** (ili bilo kojeg tipa nasleđenog od **RuntimeException**), pošto su to *neprovereni izuzeci* (engl. *unchecked exceptions*). Pošto **RuntimeException** ukazuje na greške u programiranju, ovakav izuzetak uopšte ne morate hvatati, jer se on obrađuje automatski. Kada biste morali da tražite izuzetke tipa **RuntimeException**, kôd bi postao prilično nepregledan. Čak i ako obično ne hvatate izuzetke tipa **RuntimeException**, možda ćete odlučiti da u svojim paketima generišete neku vrstu ovakvog izuzetka.

Šta se dešava ako ne uhvatite ovakav izuzetak? Pošto prevodilac ne zahteva specifikaciju takvog tipa izuzetaka, zvuči razumno da bi izuzetak **RuntimeException** mogao da se proširi sve do metode **main( )**, a da ne bude uhvaćen. Da bismo videli šta će se desiti u tom slučaju, pogledajmo sledeći primer:

```
//: izuzeci/NeHvataSe.java
// Zanemarivanje izuzetaka tipa RuntimeExceptions.
// {ThrowsException}

public class NeHvataSe {
  static void f() {
    throw new RuntimeException("Iz metode f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

Već vidite da je izuzetak tipa **RuntimeException** (i sve što je iz njega izvedeno) specijalan slučaj, pošto prevodilac ne zahteva specifikaciju ovakvih vrsta izuzetaka. Izlaz u sistemskom toku grešaka (**System.err**) izgleda ovako:

```
Exception in thread "main" java.lang.RuntimeException: Iz metode f()
        at NeHvataSe.f(NeHvataSe.java:7)
        at NeHvataSe.g(NeHvataSe.java:10)
        at NeHvataSe.main(NeHvataSe.java:13)
```

Dakle, odgovor glasi: ako izuzetak tipa **RuntimeException** stigne sve do metode **main( )** a da ne bude uhvaćen, prilikom izlaska iz programa poziva se metoda **printStackTrace( )** za taj izuzetak.

Imajte na umu da u svom kodu smete da zanemarujete samo izuzetke tipa **RuntimeException**, pošto za ostale izuzetke prevodilac zahteva obradu. Izuzetak tipa **RuntimeException** zanemarujete zato što predstavlja grešku u programiranju:

1. Greška koju ne možete da obradite (primera radi, primanje reference **null** koju metodi prosleđuje programer klijent).
2. Greška koju bi, kao programer, trebalo da proverite u svom kodu (npr. izuzetak tipa **ArrayIndexOutOfBoundsException** znači da je trebalo obratiti pažnju na dužinu niza). Izuzetak koji nastaje na mestu 1, često postaje problem na mestu 2.

Sada vam je jasno koliko je korisno postojanje izuzetaka u ovom slučaju – olakšavaju postupak pronalaženja i otklanjanja grešaka.

Zanimljivo je primetiti da se Javina tehnika obrade izuzetaka ne može klasifikovati kao alatka specifične namene. Ona je projektovana za rad s nezgodnim greškama tokom izvršavanja koje se dešavaju usled delovanja činilaca van domašaja vašeg koda, ali je od presudne važnosti i za određene vrste programerskih grešaka koje prevodilac ne može da otkrije.

Vežba 12: (3) Izmenite program **unutrasnjeklase/Sekvenca.java** tako da generiše odgovarajući izuzetak ako pokušate da umetnete previše elemenata.

Čišćenje odredbom **finally**

Često postoje delovi koda koje želite da izvršite bez obzira na to da li je unutar ispitnog bloka nastao izuzetak. To je obično slučaj u nekim operacijama koje ne predstavljaju oporavak memorije (pošto se o tome brine sakupljač smeća). Da bi se postigao ovaj efekat, koristi se odredba **finally**[4] posle svih blokova za obradu izuzetaka. Prema tome, kompletan odeljak za obradu izuzetaka izgleda ovako:

```
Try {
  // Čuvana oblast: Opasne aktivnosti
  // koje mogu da generišu A, B ili C
} catch(A a1) {
  // Procedura za obradu situacije A
} catch(B b1)
  // Procedura za obradu situacije B
} catch (C c1)
  // Procedura za obradu situacije C

} finally {
  // Aktivnosti koje se uvek dešavaju
}
```

[4] Obrada izuzetaka u jeziku C++ nema odredbu **finally**, pošto se oslanja na to da će ovakvu vrstu čišćenja obavljati destruktori.

Da biste se uverili da se odredba **finally** uvek izvršava, pokrenite sledeći program:

```
//: izuzeci/PrimerZaFinally.java
// Odredba finally se uvek izvršava.

class IzuzetakTri extends Exception {}

public class PrimerZaFinally {
  static int broj = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // Sufiksno uvećavanje prvi put daje vrednost nula:
        if(broj++ == 0)
          throw new IzuzetakTri();
        System.out.println("Bez izuzetaka");
      } catch(IzuzetakTri e) {
        System.out.println("IzuzetakTri");
      } finally {
        System.out.println("U bloku finally ");
        if(broj == 2) break; // zavrsi petlju while
      }
    }
  }
} /* Ispis:
IzuzetakTri
U bloku finally
Bez izuzetka
U bloku finally
*///:~
```

Iz rezultata vidite da se odredba **finally** uvek izvršava, bez obzira na to da li se izuzetak generiše ili ne.

Na osnovu ovog programa možete zaključiti kako da se izborite s činjenicom da izuzeci u Javi ne dozvoljavaju vraćanje na mesto nastanka izuzetka, kao što je ranije pomenuto. Ako blok **try** smestite unutar petlje, možete da uvedete uslov koji mora da bude zadovoljen pre nego što se nastavi izvršavanje programa. Možete da dodate i statički brojač ili neki drugi mehanizam koji će omogućiti petlji da isproba nekoliko pristupa pre nego što odustane. Na taj način vaši programi mogu da postanu robusniji.

Čemu služi odredba **finally**?

U jeziku koji nema sakupljač smeća *ni* automatske pozive destruktora,[5] odredba **finally** je važna zato što programeru omogućuje da garantuje oslobađanje memorije, bez obzira na to šta se dešava u bloku **try**. Međutim, Java ima sakupljač smeća, pa oslobađanje memorije

[5] Destruktor je funkcija koja se poziva kad god neki objekat prestane da se koristi. Uvek se zna tačno gde i kada destruktor biva pozvan. U jeziku C++ destruktor se poziva automatski, a i C# (koji je mnogo sličniji Javi) ume da automatski uništi nepotrebne objekte.

gotovo nikad nije problem. Uz to, nema destruktore koji bi bili pozivani. Dakle, kada u Javi treba koristiti odredbu **finally**?

Odredba **finally** je neophodna kada u prvobitno stanje morate da vratite nešto što *nije* memorija. To može biti zatvaranje otvorene datoteke ili raskid veze s mrežom, brisanje nečega što ste nacrtali na ekranu, ili čak pritiskanje nekog prekidača iz spoljnog sveta, kao što je modelovano u sledećem primeru:

```
//: izuzeci/PaliGasi.java
import static net.mindview.util.Print.*;

public class Prekidac {
  private boolean stanje = false;
  public boolean read() { return stanje; }
  public void ukljucen() { stanje = true; print(this);}
  public void iskljucen() { stanje = false; print(this); }
  public String toString()  { return stanje ? "ukljucen" : "iskljucen"; }
} ///:~

//: izuzeci/IzuzetakPaliGasi1.java
public class IzuzetakPaliGasi1 extends Exception {} ///:~

//: izuzeci/IzuzetakPaliGasi2.java
public class IzuzetakPaliGasi2 extends Exception {} ///:~

//: izuzeci/PrekidacPaliGasi.java
// Čemu služi odredba finally?

public class PrekidacPaliGasi {
  private static Prekidac pr = new Prekidac();
  public static void f()
  throws IzuzetakPaliGasi1, IzuzetakPaliGasi2 {}
  public static void main(String[] args) {
    try {
      pr.ukljucen();
      // Kod koji moze da generiše izuzetke...
      f();
      pr.iskljucen();
    } catch(IzuzetakPaliGasi1 e) {
      System.out.println("IzuzetakPaliGasi1");
      pr.iskljucen();
    } catch(IzuzetakPaliGasi2 e) {
      System.out.println("IzuzetakPaliGasi2");
      pr.iskljucen();
    }
  }
} /* Ispis:
ukljucen
iskljucen
*///:~
```

Cilj je osigurati da prekidač bude isključen kada se završi izvršavanje metode **main**( ), pa se **pr.iskljucen**( ) stavlja na kraj ispitnog bloka i na kraj svakog bloka za obradu izuzetaka. Međutim, moguće je da nastane izuzetak koji ovde nije uhvaćen, pa bi metoda **pr.iskljucen**( ) bila preskočena. Pomoću odredbe **finally**, kôd za čišćenje možete da smestite unutar ispitnog bloka na samo jednom mestu:

```
//: izuzeci/UzFinally.java
// Finally garantuje čišćenje.

public class UzFinally {
  static Prekidac pr = new Prekidac();
  public static void main(String[] args) {
    try {
      pr.ukljucen();
      // Kôd koji može da generiše izuzetke...
      PrekidacPaliGasi.f();
    } catch(IzuzetakPaliGasi1 e) {
      System.out.println("IzuzetakPaliGasi1");
    } catch(IzuzetakPaliGasi2 e) {
      System.out.println("IzuzetakPaliGasi2");
    } finally {
      pr.iskljucen();
    }
  }
} /* Ispis:
ukljucen
iskljucen
*///:~
```

Ovde je metoda **pr.iskljucen**( ) premeštena na mesto gde će se sigurno izvršavati bez obzira na to šta se desi.

Čak i u slučajevima kada se izuzetak ne uhvati u tekućem skupu blokova **catch**, **finally** će se izvršiti pre nego što mehanizam za obradu izuzetaka nastavi da traži blok za obradu na sledećem nivou:

```
//: izuzeci/UvekFinally.java
// Finally se uvek izvršava.
import static net.mindview.util.Print.*;

class IzuzetakCetiri extends Exception {}

public class UvekFinally {
  public static void main(String[] args) {
    System.out.println(
      "Ulazak u prvi ispitni blok");
    try {
      System.out.println(
        "Ulazak u drugi ispitni blok");
      try {
```

```
        throw new IzuzetakCetiri();
      } finally {
        System.out.println(
          "Finally u drugom ispitnom bloku");
      }
    } catch(IzuzetakCetiri e) {
      System.out.println(
        "Uhvaćen IzuzetakCetiri u prvom ispitnom bloku");
    } finally {
      System.out.println("Finally u prvom ispitnom bloku");
    }
  }
} /* Ispis:
Ulazak u prvi ispitni bolk
Ulazak u drugi ispitni bolk
Finally u drugom ispitnom bloku
Uhvaćen IzuzetakCetiri u prvom ispitnom bloku
Finally u prvom ispitnom bloku
} ///:~
```

Naredba **finally** će se izvršavati i u situacijama u kojima učestvuju naredbe **break** i **continue**. Obratite pažnju na sledeće: uz označen **break** i označen **continue**, **finally** ukida potrebu za naredbom **goto** u Javi.

Vežba 13: (2) Promenite vežbu 9 dodavanjem odredbe **finally**. Uverite se da se blok **finally** izvršava, čak i ako je generisan izuzetak **NullPointerException**.

Vežba 14: (2) Pokažite da prekidač **PrekidacPaliGasi.java** može da se pokvari tako što ćete generisati izuzetak **RuntimeException** unutar bloka **try**.

Vežba 15: (2) Pokažite da **UzFinally.java** ne zakazuje tako što ćete generisati izuzetak **RuntimeException** unutar bloka **try**.

Upotreba bloka **finally** tokom povratka iz metode pomoću rezervisane reči **return**

Pošto se blok **finally** uvek izvršava, moguć je povratak s više tačaka metode, a da se i dalje može jemčiti da će važno čišćenje biti obavljeno:

```
//: izuzeci/VisePovratakaIzMetode.java
import static net.mindview.util.Print.*;

public class VisePovratakaIzMetode {
  public static void f(int i) {
    print("Inicijalizacija koja zahteva čišćenje");
    try {
      print("Tačka 1");
      if(i == 1) return;
      print("Tačka 2");
      if(i == 2) return;
```

```
      print("Tačka 3");
      if(i == 3) return;
      print("Kraj");
      return;
    } finally {
      print("Obavljanje čišćenja");
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
      f(i);
  }
} /* Ispis:
Inicijalizacija koja zahteva čišćenje
Tačka 1
Obavljanje čišćenja
Inicijalizacija koja zahteva čišćenje
Tačka 1
Tačka 2
Obavljanje čišćenja
Inicijalizacija koja zahteva čišćenje
Tačka 1
Tačka 2
Tačka 3
Obavljanje čišćenja
Inicijalizacija koja zahteva čišćenje
Tačka 1
Tačka 2
Tačka 3
Kraj
Obavljanje čišćenja
*///:-
```

Iz rezultata vidite da nije važno gde se vraćate iz klase koja sadrži odredbu **finally**.

Vežba 16: (2) Prepravite program **ponovnaupotreba/CADSystem.java** tako da pokažete da se pravilno čišćenje jemči čak i kada se povratak rezervisanom rečju **return** obavi iz sredine bloka **try-finally**.

Vežba 17: (3) Prepravite program **polimorfizam/Zaba.java** tako da pravilno čišćenje jemči blokom **try-finally**, i pokažite da to radi čak i kada se rezervisanom rečju **return** vratite iz sredine bloka **try-finally**.

Mana: izgubljeni izuzetak

Realizacija izuzetaka u Javi je izvanredna, ali, nažalost, ima propusta. Iako izuzeci ukazuju na slabu tačku u programu i ne bi nikada smeli da se ignorišu, moguće je da se izuzetak zagubi. To može da se desi kada se koristi odredba **finally**:

```
//: izuzeci/ZagubljenaPoruka.java
// Kako se izuzetak može zagubiti.

class VeomaVazanIzuzetak extends Exception {
  public String toString() {
    return "Veoma važan izuzetak!";
  }
}
class TrivijalanIzuzetak extends Exception {
  public String toString() {
    return "Trivijalan izuzetak";
  }
}

public class ZagubljenaPoruka {
  void f() throws VeomaVazanIzuzetak {
    throw new VeomaVazanIzuzetak();
  }
  void pocisti() throws TrivijalanIzuzetak {
    throw new TrivijalanIzuzetak();
  }
  public static void main(String[] args)  {
      try {
        ZagubljenaPoruka lm = new ZagubljenaPoruka();
        try {
         lm.f();
        } finally {
         lm.pocisti();
        }
      }  catch(Exception e)  {
         System.out.println(e);
      }
  }
} /* Ispis:
Trivijalan izuzetak
* ///:~
```

Vidite da nema naznaka o prisustvu izuzetka tipa **VeomaVazanIzuzetak** koji je zamenjen izuzetkom tipa **TrivijalanIzuzetak** u odredbi **finally**. To je prilično ozbiljan nedostatak, jer znači da izuzetak može potpuno da se izgubi i to na način koji je mnogo teže otkriti nego u gornjem primeru. Nasuprot tome, u jeziku C++ se situacija, kada se drugi izuzetak generiše pre nego što je prvi obrađen, smatra velikom greškom u programiranju. Možda će u budućim verzijama Jave ovaj problem biti ispravljen (s druge strane, metodu koja generiše izuzetak, kao što je **pocisti( )** u gornjem primeru, obično ćete smestiti unutar odredbe **try-catch**).

Još jednostavniji način da izgubite izuzetak jeste običan povratak (pomoću rezervisane reči **return**) iz bloka **finally**:

```
//: izuzeci/PrigusivacIzuzetaka.java

public class PrigusivacIzuzetaka {
  public static void main(String[] args) {
    try {
      throw new RuntimeException();
    } finally {
      // Upotreba rezervisane reči 'return' u bloku finally
      // svaki bačeni izuzetak čini nemim.
      return;
    }
  }
} ///:~
```

Ako pokrenete ovaj program, videćete da ništa ne ispisuje na izlazu, iako se generiše jedan izuzetak.

Vežba 18: (3) Dodajte drugi nivo gubitka izuzetaka u **ZagubljenaPoruka.java** da bi **TrivijalanIzuzetak** sam sebe zamenio trećim izuzetkom.

Vežba 19: (2) Rešite problem u programu **ZagubljenaPoruka.java** tako što će se poziv čuvati u bloku **finally**.

Ograničenja izuzetaka

Kada redefinišete metodu, možete da generišete samo izuzetke koji su zadati u verziji metode iz osnovne klase. To je korisno ograničenje, pošto znači da će kôd koji radi sa osnovnom klasom automatski raditi i sa svakom klasom izvedenom iz nje (što je, naravno, osnovni koncept objektno orijentisanog programiranja), uključujući i izuzetke.

Ovaj primer ilustruje vrste ograničenja koja se u vreme prevođenja nameću izuzecima:

```
//: izuzeci/StormyInning.java
// Redefinisane metode mogu da generišu samo izuzetke navedene u njihovim
// osnovnim klasama, ili izuzetke izvedene iz izuzetaka osnovne klase.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  public Inning() throws BaseballException {}
  public void event () throws BaseballException {
    // Zapravo ne mora da generiše nikakav izuzetak
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Ne generiše izuzetke koje prevodilac proverava
}
```

```
class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  public void event() throws RainedOut;
  public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements Storm {
  // Dozvoljeno je dodavati nove izuzetke za konstruktore, ali morate da se
  // pozabavite izuzecima konstruktora osnovne klase:
  public StormyInning()
    throws RainedOut,
    BaseballException {}
  public StormyInning(String s)
    throws Foul,
    BaseballException {}
  // Standardne metode moraju da se prilagođavaju osnovnoj klasi:
  //! void walk() throws PopFoul {} //Greška tokom prevođenja
  // NIJE DOZVOljENO dodavanje izuzetaka
  // postojećim metodama iz osnovne klase:
  //! public void event() throws RainedOut {}
  // Ako metoda već ne postoji u osnovnoj klasi, izuzetak se može dodati:
  public void rainHard() throws RainedOut {}
  // Možete odlučiti da uopšte ne generišete izuzetke,
  // čak i ako to radi osnovna klasa:
  public void event() {}
  // Redefinisane metode mogu da generišu nasledene izuzetke:
  public void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.out.println("Pop foul");
    } catch(RainedOut e) {
      System.out.println("Rained out");
    } catch(BaseballException e) {
      System.out.println("Generic baseball exception");
    }
    // Situacija koja nije generisala izuzetak u izvedenoj verziji.
    try {
      // Šta se dešava pri konverziji tipa u opštiji?
      Inning i = new StormyInning();
      i.atBat();
      // Morate da uhvatite izuzetke
      // verzije metoda u osnovnoj klasi:
    } catch(Strike e) {
```

```
        System.out.println("Strike");
      } catch(Foul e) {
        System.out.println("Foul");
      } catch(RainedOut e) {
        System.out.println("Rained out");
      } catch(BaseballException e) {
        System.out.println(
          "Generic baseball exception");
      }
    }
} ///:~
```

Vidite kako u klasi **Inning** i konstruktor i metoda **event( )** tvrde da će generisati izuzetak, ali to ne rade. To je ispravno zato što omogućuje da primorate korisnika da uhvati sve izuzetke koji mogu da nastanu u redefinisanim verzijama metode **event( )**. Isto važi i za apstraktne metode, kao što se vidi u **atBat( )**.

Interfejs **Storm** je zanimljiv jer sadrži jednu metodu – **event( )** – koja je definisana u **Inning**, i jednu metodu koja nije definisana. Obe metode generišu nov tip izuzetka, **RainedOut**. Kada klasa **StormyInning** nasledi **Inning** i realizuje **Storm**, videćete da metoda **event( )** interfejsa **Storm** *ne može* da promeni specifikaciju izuzetka metode **event( )** u klasi **Inning**. I ovde to ima smisla, inače nikada ne biste znali da li ste uhvatili pravi izuzetak kada radite sa osnovnom klasom. Naravno, ukoliko se metoda opisana u interfejsu ne nalazi u osnovnoj klasi, poput **rainHard( )**, ona može da generiše sopstvene izuzetke.

Ograničenja za izuzetke ne primenjuju se na konstruktore. U **StormyInning** vidite da konstruktor može da generiše šta god želi, bez obzira na to šta generiše konstruktor osnovne klase. Međutim, pošto konstruktor osnovne klase uvek mora da se nekako pozove (ovde se podrazumevani konstruktor poziva automatski), konstruktor izvedene klase mora da deklariše sve izuzetke konstruktora osnovne klase u svojoj specifikaciji izuzetaka.

Konstruktor izvedene klase ne može da uhvati izuzetke koje je generisao konstruktor osnovne klase.

StormyInning.walk( ) neće biti preveden zato što generiše izuzetak, što nije slučaj s metodom **Inning.walk( )**. Kada bi to bilo dozvoljeno, mogli biste da napišete kôd koji bi pozivao **Inning.walk( )** i ne bi uopšte morao da obrađuje izuzetke, ali bi se tada, nakon zamene objekta klase koja je izvedena iz klase **Inning**, generisali izuzeci i kôd bi zapao u probleme. Primoravanjem metoda izvedene klase da poštuju specifikacije izuzetaka metoda osnovne klase, omogućuje se zamenjivost objekata.

Redefinisana metoda **event( )** pokazuje da verzija metoda izvedene klase može odlučiti da ne generiše nikakve izuzetke, čak i ako to čini verzija osnovne klase. I ovo je u redu zato što ne prouzrokuje probleme u kodu koji je već napisan, pod pretpostavkom da verzija osnovne klase generiše izuzetke. Slična logika se primenjuje i na metodu **atBat( )** koja generiše izuzetak tipa **PopFoul** izveden iz izuzetka tipa **Foul** koji generiše verzija metode **atBat( )** osnovne klase. Ako neko napiše kôd koji radi s klasom **Inning** i poziva metodu **atBat( )**, moraće da hvata izuzetak tipa **Foul**. Pošto je izuzetak tipa **PopFoul** izveden iz **Foul**, blok za obradu izuzetaka će hvatati i **PopFoul**.

Poslednje što nas zanima je metoda **main( )**. U njoj možete da vidite sledeće: ako radite sa objektom tipa **StormyInning**, prevodilac vas primorava da hvatate samo izuzetke koji su specifični za tu klasu, ali ako konvertujete objekat u prost tip, prevodilac vas s pravom primorava da hvatate izuzetke osnovnog tipa. Sva ova ograničenja čine tehniku za obradu izuzetaka mnogo snažnijom.[6]

Korisno je uvideti sledeće: iako prevodilac tokom nasleđivanja podržava specifikacije izuzetaka, one nisu deo potpisa metode koji se sastoji samo od imena metode i tipova argumenata. Stoga ne možete da preklopite metode prema specifikacijama izuzetaka. Pored toga, samo zato što specifikacija izuzetka postoji u verziji metode iz osnovne klase, ne znači da ona mora da postoji i u verziji metode u izvedenoj klasi. Ovo se prilično razlikuje od pravila nasleđivanja, kada metoda osnovne klase mora da postoji i u izvedenoj klasi. Drugim rečima, „interfejs specifikacije izuzetka" za odredenu metodu može da se suzi tokom nasleđivanja i redefinisanja, ali se neće proširiti, što je upravo suprotno pravilu za interfejs klase tokom nasleđivanja.

Vežba 20: (3) Promenite **StormyInning.java** dodavanjem izuzetka tipa **UmpireArgument** i metoda koji ga generišu. Testirajte izmenjenu hijerarhiju.

Konstruktori

Prilikom pisanja koda sa izuzecima, posebno je važno da se uvek pitate: „Ako nastane izuzetak, hoće li objekat biti pravilno počišćen?" U većini slučajeva sve je u redu, ali se u konstruktorima pojavljuje problem. Konstruktor stavlja objekat u bezbedno početno stanje, ali može da izvodi neke operacije, npr. otvaranje datoteka, koje se ne brišu sve dok korisnik ne završi rad sa objektom i pozove specijalnu metodu za čišćenje. Ako generišete izuzetak unutar konstruktora, čišćenje se možda neće pravilno izvršiti. To znači da morate biti posebno obazrivi dok sastavljate konstruktor.

Pošto ste upravo savladali odredbu **finally**, možda mislite da je ona pravo rešenje. Ipak, nije toliko jednostavno, pošto **finally** *svaki put* izvršava kôd za čišćenje. Ukoliko konstruktor bude prekinut tokom svog izvršavanja, to može značiti da nije uspeo da napravi deo objekta koji će biti počišćen u bloku **finally**.

U sledećem primeru pravi se klasa **UlaznaDatoteka** koja otvara datoteku i omogućuje čitanje po jednog reda. Ona koristi klase **FileReader** i **BufferedReader** iz standardne Javine U/I biblioteke. O njima će biti reči u poglavlju *Javin ulazno-izlazni sistem*, ali one su toliko jednostavne da ćete bez problema razumeti njihovo osnovno korišćenje:

```
//: izuzeci/UlaznaDatoteka.java
// Obratite pažnju na izuzetke u konstruktorima.
import java.io.*;

public class UlaznaDatoteka {
  private BufferedReader in;
```

[6] U ISO standard za C++ dodata su slična ograničenja kojima se zahteva da izuzeci izvedenih metoda budu isti kao izuzeci koje generiše metoda osnovne klase, ili izvedeni iz njih. To je jedini slučaj kada je C++ zaista u stanju da proverava specifikacije izuzetaka tokom prevođenja.

```
    public UlaznaDatoteka(String imeDatoteke) throws Exception {
      try {
        in = new BufferedReader(new FileReader(imeDatoteke));
        // Još koda koji bi mogao da generiše izuzetke
      } catch(FileNotFoundException e) {
        System.out.println("Nisam našao " + imeDatoteke);
        // Nije otvorena, pa nemojte ni da je zatvarate
        throw e;
      } catch(Exception e) {
        // Svi ostali izuzeci moraju da je zatvore
        try {
          in.close();
        } catch(IOException e2) {
          System.out.println("in.close() neuspešan");
        }
        throw e; // Generiši ponovo
      } finally {
        // Nemojte je zatvarati ovde!!!
      }
    }
    String citajRed() {
      String s;
      try {
        s = in.readLine();
     } catch(IOException e) {
        throw new RuntimeException("citajRed() neuspešna");
    }
    return s;
  }
  public void ciscenje() {
      try {
        in.close();
        System.out.println("ciscenje() uspešno");
     } catch(IOException e2) {
        throw new RuntimeException("in.close() neuspešna");
      }
    }
  } ///:~
```

Konstruktor klase **UlaznaDatoteka** prima argument tipa **String**, a to je ime datoteke koju želite da otvorite. Unutar bloka **try** pravi se objekat klase **FileReader** pomoću imena te datoteke. Ovakav objekat nije posebno koristan sve dok ga ne upotrebite za pravljenje objekta tipa **BufferedReader** s kojim možete i da komunicirate. Jedna od prednosti klase **UlaznaDatoteka** jeste to što objedinjuje ove dve operacije.

Ako konstruktor klase **FileReader** ne uspe da obavi zadatak, generisaće izuzetak **FileNotFoundException**. Taj izuzetak se mora uhvatiti zasebno, jer je to jedini slučaj kada ne želite da zatvorite datoteku, pošto ona nije ni otvorena. *Svi drugi* blokovi **catch** moraju da zatvore datoteku, pošto je *bila* otvorena u vreme nastanka izuzetka. (Naravno, to je komplikovanije ako nekoliko metoda može da generiše izuzetak **FileNotFoundException**.

U tom slučaju, možda ćete podeliti kôd u nekoliko blokova **try**.) Metoda **close**( ) mogla bi da generiše izuzetak, pa se on ispituje i hvata čak i ako je generisan unutar bloka druge odredbe **catch** – za Javinog prevodioca, to je još samo jedan par vitičastih zagrada. Nakon izvođenja lokalnih operacija, ponovo se generiše izuzetak. To je u redu, pošto konstruktor nije bio uspešan, a metoda koja poziva ne bi trebalo da pretpostavi kako je objekat pravilno napravljen i ispravan.

U ovom primeru, u kome se ne koristi prethodno pomenuta tehnika indikatora, odredba **finally** sasvim sigurno *nije* mesto za zatvaranje datoteke metodom **close**( ), pošto bi se u tom slučaju zatvarala kad god se izvrši konstruktor. Želimo da datoteka bude otvorena tokom korisnog životnog veka objekta klase **UlaznaDatoteka**.

Metoda **citajRed**( ) vraća objekat klase **String** koji sadrži sledeći red datoteke. Ona poziva metodu **readLine**( ) koja može da generiše izuzetak, ali se taj izuzetak hvata, pa **citajRed**( ) ne generiše nikakav izuzetak. Pri projektovanju izuzetaka, problem je to što se ne zna da li izuzetak treba potpuno obraditi na ovom nivou, da li ga treba obraditi samo delimično i isti (ili drugačiji) izuzetak proslediti dalje, ili ga valja proslediti bez ikakve obrade. Prosleđivanje, kada je odgovarajuće, sigurno može da pojednostavi pisanje koda. U ovoj situaciji, metoda **citajRed**( ) *pretvara* izuzetak u **RuntimeException** da bi ukazala na programsku grešku.

Korisnik mora da pozove metodu **ciscenje**( ) kada završi s korišćenjem objekta klase **UlaznaDatoteka**. Na taj način se mogu osloboditi sistemski resursi (npr. identifikatori datoteka) koje koriste objekti tipa **BufferedReader** i/ili **FileReader**. To ne bi trebalo da radite dok ne završite s korišćenjem objekta klase **UlaznaDatoteka**. Možda nameravate da stavite tu funkciju u metodu **finalize**( ), ali kao što je pomenuto u poglavlju *Inicijalizacija i čišćenje*, ne možete uvek biti sigurni da će metoda **finalize**( ) biti pozvana (čak i ako ste sigurni da će biti pozvana, ne znate *kada* će se to desiti). To je jedan od nedostataka Jave: nijedno čišćenje, osim brisanja memorije, ne izvršava se automatski. Zato programera klijenta morate obavestiti da je on odgovoran za čišćenje.

Najbezbedniji način korišćenja klase koja može generisati izuzetak tokom konstrukcije i koja zahteva čišćenje jeste upotreba ugneždenih blokova **try**:

```
//: izuzeci/Ciscenje.java
// Zajemčeno pravilno čišćenje resursa.

public class Ciscenje {
  public static void main(String[] args) {
    try {
      UlaznaDatoteka in = new UlaznaDatoteka("Ciscenje.java");
      try {
        String s;
        int i = 1;
        while((s = in.getLine()) != null)
          ; // Ovde obradi red po red...
      } catch(Exception e) {
        System.out.println("Uhvatio izuzetak u metodi main");
        e.printStackTrace(System.out);
```

```
      } finally {
        in.ciscenje();
      }
    } catch(Exception e) {
      System.out.println("Nije uspela konstrukcija UlazneDatoteke");
    }
  }
} /* Ispis:
ciscenje() uspešno
*///:~
```

Pažljivo razmotrite upotrebljenu logiku: konstrukcija objekta tipa **UlaznaDatoteka** suštinski je u sopstvenom bloku **try**. Ako ta konstrukcija ne uspe, ulazi se u spoljni blok **catch**, a **ciscenje( )** se ne poziva. Međutim, ukoliko konstrukcija uspe, morate se postarati da objekat bude počišćen, pa neposredno nakon konstrukcije pravite nov blok **try**. Odredba **finally** koja obavlja čišćenje pridružena je unutrašnjem bloku **try**; zato se blok **finally** neće izvršiti ako konstrukcija ne uspe, a izvršava se uvek kada konstrukcija uspe.

Ovaj opšti način čišćenja treba primenjivati i kada konstruktor uopšte ne generiše izuzetke. Osnovno pravilo glasi: neposredno nakon pravljenja objekta koji zahteva čišćenje, otvorite blok **try-finally**:

```
//: izuzeci/NacinCiscenja.java
// Iza svakog objekta koji treba počistiti mora slediti blok try-finally

class TrebaJePocistiti { // Konstrukcija ne može da ne uspe
  private static long brojac = 1;
  private final long id = brojac++;
  public void ciscenje() {
    System.out.println("TrebaJePocistiti " + id + " počišćena");
  }
}

class IzuzetakTokomKonstrukcije extends Exception {}

class TrebaJePocistiti2 extends TrebaJePocistiti {
  // Konstrukcija može da ne uspe:
  public TrebaJePocistiti2() throws IzuzetakTokomKonstrukcije {}
}

public class NacinCiscenja {
  public static void main(String[] args) {
    // Deo 1:
    TrebaJePocistiti nc1 = new TrebaJePocistiti();
    try {
      // ...
    } finally {
      nc1.ciscenje();
    }
```

```
    // Deo 2:
    // Ako konstrukcija ne može da ne uspe, objekte možete da grupišete:
    TrebaJePocistiti nc2 = new TrebaJePocistiti();
    TrebaJePocistiti nc3 = new TrebaJePocistiti();
    try {
      // ...
    } finally {
      nc3.ciscenje(); // Obrnut redosled u odnosu na konstrukciju
      nc2.ciscenje();
    }

    // Deo 3:
    // Ako konstrukcija može da ne uspe, morate čuvati svaku od njih:
    try {
      TrebaJePocistiti2 nc4 = new TrebaJePocistiti2();
      try {
        TrebaJePocistiti2 nc5 = new TrebaJePocistiti2();
        try {
          // ...
        } finally {
          nc5.ciscenje();
        }
      } catch(IzuzetakTokomKonstrukcije e) { // konstruktor nc5
        System.out.println(e);
      } finally {
        nc4.ciscenje();
      }
    } catch(IzuzetakTokomKonstrukcije e) { // konstruktor nc4
      System.out.println(e);
    }
  }
} /* Ispis:
TrebaJePocistiti 1 počišćena
TrebaJePocistiti 3 počišćena
TrebaJePocistiti 2 počišćena
TrebaJePocistiti 5 počišćena
TrebaJePocistiti 4 počišćena
*///:~
```

U metodi **main**( ), deo 1 je lako razumeti: iza objekta koji treba počistiti sledi blok **try-finally**. Ukoliko konstrukcija objekta ne može da ne uspe, blok **catch** nije potreban. U delu 2, vidite da objekti čija konstrukcija ne može da ne uspe mogu biti grupisani i za konstrukciju i za čišćenje.

Deo 3 pokazuje kako se radi sa objektima čija konstrukcija može da ne uspe *i* koji zahtevaju čišćenje. Pravilna obrada ove situacije znači pravljenje krša, jer svaku konstrukciju morate da obuhvatite sopstvenim blokom **try-catch**, a nakon njega mora da sledi blok **try-finally** za zajemčeno čišćenje.

Ružnoća koda za obradu izuzetaka u ovom slučaju, predstavlja jak argument za pravljenje konstruktora koji ne mogu da ne uspeju, mada to nije uvek moguće.

Povedite računa o sledećem: ako **ciscenje( )** može da generiše izuzetak, možda će vam biti potrebni dodatni blokovi **try**. Dakle, morate pažljivo da razmotrite sve mogućnosti i da obezbedite svaku od njih.

Vežba 21: (2) Pokažite da konstruktor izvedene klase ne može da hvata izuzetke koje generiše konstruktor osnovne klase.

Vežba 22: (2) Napravite klasu **NeuspesanKonstruktor** čiji konstruktor može da ne uspe da završi konstrukciju i tada generiše izuzetak. U metodi **main( )** napišite kôd koji pravilno štiti od takvog neuspeha.

Vežba 23: (4) U prethodnoj vežbi, dodajte klasu s nekom metodom **ciscenje( )**. Izmenite **NeuspesanKonstruktor** tako da konstruktor pravi jedan od objekata koji zahtevaju čišćenje kao objekat član, nakon čega konstruktor generiše izuzetak, a onda pravi drugi objekat član koji zahteva čišćenje. Napišite kôd koji pravilno štiti od neuspeha, a u metodi **main( )** dokažite da ste se obezbedili od svih mogućih neuspeha.

Vežba 24: (3) Klasi **NeuspesanKonstruktor** dodajte neku metodu **ciscenje( )**. Napišite kôd koji pravilno upotrebljava tu klasu.

Pronalaženje sličnih izuzetaka

Kada se generiše izuzetak, sistem za obradu izuzetaka traži „najbliže" blokove po redosledu kojim su pisani. Kada pronađe odgovarajući, smatra se da je izuzetak obrađen, a traženje prestaje.

Pronalaženje sličnih izuzetaka ne zahteva da izuzetak i blok za obradu savršeno odgovaraju jedan drugom. Objektu izvedene klase odgovaraće blok za osnovnu klasu, kao što je prikazano u ovom primeru:

```
//: izuzeci/Covek.java
// Hvatanje hijerarhija izuzetaka.

class Dosada extends Exception {}
class Kijanje extends Dosada {}

public class Covek {
  public static void main(String[] args) {
    // Uhvati tačan tip
    try {
      throw new Kijanje();
    } catch(Kijanje s) {
      System.out.println("Uhvatio Kijanje");
    } catch(Dosada a) {
      System.out.println("Uhvatio Dosadu");
    }
  }
} /* Ispis:
Caught Sneeze
Caught Annoyance
} ///:~
```

Izuzetak tipa **Kijanje** uhvatiće prva odredba **catch** kojoj odgovara, a to je, naravno, prva. Međutim, ako uklonite prvu odredbu **catch** i ostavite samo odredbu **catch** za klasu **Dosada**, kôd će i dalje raditi, jer blok hvata osnovnu klasu klase **Kijanje**. Drugim rečima, **catch (Dosada e)** uhvatiće izuzetak tipa **Dosada** *i sve klase koje su izvedene iz tog tipa.* To je korisno: ako odlučite da metodi dodate više izvedenih izuzetaka, kôd programera klijenta neće morati da se menja sve dok klijent obrađuje izuzetke osnovne klase.

Ako pokušate da „maskirate" izuzetke izvedene klase tako što na prvo mesto stavite odredbu **catch** za hvatanje osnovne klase, kao ovde:

```
try {
  throw new Kijanje();
} catch(Dosada a) {
  // ...
} catch(Kijanje s) {
  // ...
}
```

prevodilac će prikazati grešku, pošto vidi da odredba za hvatanje **Kijanje** nije dostupna.

Vežba 25: (2) Napravite hijerarhiju izuzetaka u tri nivoa. Potom napravite osnovnu klasu **A** s metodom koja generiše izuzetak u osnovi hijerarhije. Izvedite klasu **B** iz **A** i redefinišite metodu tako da generiše izuzetak u drugom nivou hijerarhije. Ponovite postupak i izvedite klasu **C** iz klase **B**. U metodi **main( )** napravite objekat klase **C** i svedite ga naviše na tip **A**, a zatim pozovite metodu.

Drugi pristupi

Sistem za obradu izuzetaka predstavlja skrivena vrata koja programu omogućavaju da napusti izvršavanje normalne sekvence naredbe. Skrivena vrata se upotrebljavaju kada nastupi „izuzetan uslov", poput onoga kada normalno izvršavanje više nije moguće ili poželjno. Izuzeci predstavljaju uslove koje tekuća metoda ne može da obradi. Sistemi za obradu izuzetaka su razvijeni zato što je pristup koji je podrazumevao obradu svih mogućih greška izazvanih svakim pozivom svake funkcije bio previše tegoban, pa ga programeri nisu sledili. Stoga su greške ignorisali. Vredi primetiti da je olakšavanje programiranja obrade grešaka bilo glavna motivacija za izum izuzetaka.

Jedna od važnih smernica u obradi izuzetaka glasi: „Ne hvataj izuzetak s kojim ne znaš šta da uradiš". U stvari, jedan od važnih ciljeva obrade izuzetaka bio je da se kôd za obradu grešaka odmakne od tačke gde je greška nastala. Time dobijate mogućnost da se u jednom delu svog koda usredsredite na ono što hoćete da uradite, a da se u drugom, zasebnom delu bavite problemima. Stoga glavni deo vašeg koda neće biti zatrpan logikom za obradu grešaka, pa će biti mnogo razumljiviji i lakše će se održavati. Obrada izuzetaka obično smanjuje obim koda za obradu grešaka, jer jedan blok za obradu može da obradi greške nastale na mnogo mesta.

Provereni izuzeci donekle komplikuju ovaj scenario, pošto vas primoravaju da stavljate odredbe **catch** i na mesta na kojima niste spremni da obradite određenu grešku. To prouzrokuje problem „izaziva oštećenja ako se proguta“:

```
try {
  // ... radi se nešto korisno
} catch(ObavezanIzuzetak e) {} // Progutano!
```

Programeri (među kojima i ja, u prvom izdanju ove knjige) obično urade ono najjednostavnije i samo „progutaju“ izuzetak – često nenamerno. Kada to urade, prevodilac je zadovoljan, pa ukoliko se ne sete da ponovo prođu kroz kôd i isprave ga na tom mestu, taj izuzetak će biti izgubljen. Izuzetak se pojavljuje, ali i nestaje, jer odmah biva progutan. Pošto vas prevodilac prisiljava da odmah pišete kôd za obradu izuzetka, izgleda kao da je ovo najlakše rešenje, iako je verovatno najgore od svega što možete da uradite.

Kada sam shvatio da sam i ja to uradio, zgrozio sam se pa sam u drugom izdanju „rešio“ problem tako što sam ispisao slojeve steka u bloku za obradu (što se još vidi – kako i treba – u više primera u ovom poglavlju). Mada je to korisno za praćenje ponašanja izuzetaka, i dalje se vidi da na tom mestu zapravo ne znate šta da radite s tim izuzetkom. U ovom odeljku ću vas upoznati s nekim elementima i komplikacijama koje nastaju zbog proverenih izuzetaka, i s mogućnostima da ih rešite.

Tema izgleda jednostavna, ali ne samo da je komplikovana, nego i sporna. Ima ljudi koji kruto zastupaju suprotstavljena stanovišta i smatraju da pravi odgovor (njihov) naprosto bode oči. Mislim da je razlog za jedno od tih mišljenja jasna prednost koja se vidi prilikom prelaska iz jezika u kojem se tipovi ne proveravaju, kakav je bio C pre ANSI standarda, u jezik u kojem se tipovi obavezno proveravaju i statički zadaju (tj. proveravaju već tokom prevođenja), kao što su C++ i Java. Kada obavite taj prelazak (kao ja), prednosti su toliko dramatične pa izgleda da je statička provera tipova uvek najbolji odgovor na većinu problema. Pokušaću da vam prenesem mali deo evolucije mog stanovišta. Posumnjao sam u *apsolutnu* vrednost statičke provere tipova; jasno je da je ona najčešće veoma korisna, ali postoji nejasna granica nakon koje ona postaje smetnja. (Jedan od mojih omiljenih citata je: „Svi modeli su netačni. Neki su korisni.“).

Istorija

Obrada izuzetaka je nastala u sistemima PL/1 i Mesa. Kasnije se pojavljivala u jezicima CLU, Smalltalk, Modula-3, Ada, Eiffel, C++, Python, Java, kao i u jezicima Ruby i C# nastalim nakon Jave. U tom pogledu Java je slična C++-u, sem na mestima gde su projektanti Jave smatrali da pristup iz C++-a prouzrokuje probleme.

Da bi se programerima pružio sistem za obradu i oporavak od grešaka koji će imati više izgleda da bude upotrebljavan, jeziku C++ je obrada izuzetaka dodata relativno kasno u procesu standardizacije. Nju je zagovarao Bjarne Stroustrup, autor jezika. Model izuzetaka u jeziku C++ prvenstveno potiče iz CLU-a. Međutim, tada su postojali i drugi jezici koji su podržavali obradu izuzetaka: Ada, Smalltalk (imali su izuzetke, ali ne i specifikacije izuzetaka) i Modula-3 (koja je imala i izuzetke i specifikacije).

U svom plodonosnom članku[7] o toj temi, Liskova i Snyder primećuju da je glavni nedostatak jezika poput C-a koji greške prijavljuju u prolazu sledeći:

> *„...iza svakog poziva mora slediti uslovno ispitivanje da bi se utvrdio ishod poziva. Ovaj zahtev vodi ka programima koji se teško čitaju, a verovatno su i neefikasni, pa obeshrabruju programere da se bave prijavljivanjem i obradom izuzetaka."*

Dakle, jedan od prvobitnih motiva za obradu izuzetaka bio je da se spreči ovaj zahtev, ali uz proverene izuzetke u Javi često dobijamo upravo tu vrstu koda. U nastavku, autori kažu:

> *„...zahtev da tekst bloka za obradu bude priključen pozivu koji je izazvao izuzetak vodi ka nečitljivim programima u kojima se izrazi prekidaju blokovima za obradu."*

Sledeći pristup CLU-a prilikom projektovanja C++ izuzetaka, Stroustrup kaže da je cilj bio smanjenje količine koda potrebnog za oporavak od grešaka. Mislim da je opazio kako programeri na C-u obično ne pišu kôd za obradu grešaka, pošto su količina i položaj tog koda bili zastrašujući i zbunjujući. Zato su se navikli da to rade kao u C-u, tj. da ignorišu greške u kodu i da za otkrivanje i otklanjanje problema upotrebljavaju „dibagere". Da bi ti C programeri upotrebljavali izuzetke, trebalo ih je ubediti da pišu „dodatni" kôd koji inače nisu pisali. Dakle, da bi se programeri zainteresovali za bolju obradu grešaka, količina koda koju je trebalo da „dodaju" nije smela da bude prevelika. Ne zaboravite na taj cilj kada budemo razmatrali posledice proverenih izuzetaka u Javi.

C++ je od CLU-a preuzeo još jednu ideju: specifikacije izuzetaka pomoći kojih se u potpisu metode programski navode izuzeci koji mogu nastati zbog poziva te metode. Specifikacija izuzetaka zapravo ima dve svrhe. Ona može reći: „Ovaj izuzetak je nastao u mom kodu; ti ga obradi". Ali može reći i: „Ja ću ignorisati ovaj izuzetak koji može nastati kao posledica mog koda, a ti ga obradi". Dok smo razmatrali mehanizme i sintaksu izuzetaka, bili smo usredsređeni na deo „ti ga obradi", ali ovde me posebno zanima činjenica da izuzetke često ignorišemo, a specifikacija izuzetka to može da utvrdi.

U C++-u, specifikacija izuzetka nije deo podataka o tipu funkcije. Tokom prevođenja proverava se samo to da li su specifikacije izuzetaka dosledne; na primer, ako funkcija ili metoda generišu izuzetke, onda i njihove preklopljene ili izvedene verzije moraju generisati iste izuzetke. Za razliku od Jave, tokom prevođenja se ne proverava da li funkcija/metoda zaista generiše taj izuzetak, niti da li je specifikacija izuzetaka potpuna (tj. da li tačno opisuje sve izuzetke koji mogu biti generisani). Ta provera se obavlja, ali samo u vreme izvršavanja. Ukoliko se generiše izuzetak koji krši specifikaciju izuzetaka, C++ program će pozvati funkciju **unexpected( )** (neočekivan) iz svoje standardne biblioteke.

Zanimljivo je primetiti da se specifikacije izuzetaka uopšte ne upotrebljavaju u standardnoj biblioteci C++-a, zato što se koriste šabloni. U Javi postoje ograničenja načina upotrebe generičkih tipova u specifikacijama izuzetaka.

---

[7] Barbara Liskov i Alan Snyder, *Exception Handling in CLU*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, November 1979. Ovog članka nema na Internetu. Pošto je dostupan samo u odštampanom obliku, potražite ga u nekoj biblioteci.

Perspektive

Prvo, vredi primetiti da su tvorci Jave zapravo izumeli proverene izuzetke (očigledno inspirisani specifikacijama izuzetaka u C++-u i činjenicom da se C++ programeri obično ne bakću s njima). Međutim, taj eksperiment nije ponovljen ni u jednom kasnijem jeziku.

Drugo, provereni izuzeci izgledaju kao očito korisni u uvodnim primerima i malim programima. Rečeno je da se suptilne razlike pojavljuju kada programi postanu veliki. Naravno, do tolike veličine obično ne dolazi preko noći; ona nastaje neosetno. Jezici koji nisu podesni za velike projekte, upotrebljavaju se za male. Ti mali rastu i od nekog trenutka shvatamo da su od „upravljivih" postali „teško upravljivi". Smatram da to može biti posledica i preteranog proveravanja tipova; konkretno, provere izuzetaka.

Izgleda da je veličina programa značajna. To je problem, pošto se u većini rasprava za ilustraciju upotrebljavaju mali programi. Jedan od projektanata C#-a kazao je sledeće:

> *„Ispitivanje malih programa navodi na zaključak da zahtevanje specifikacija izuzetaka može povećati produktivnost programera i kvalitet koda, ali iskustvo s velikim softverskim projektima nameće drugačiji zaključak – smanjuje se produktivnost a kvalitet koda se povećava malo ili nimalo."*[8]

Autori CLU-a su kazali sledeće o neuhvaćenim izuzecima:

> *„Smatrali smo da nije realno zahtevati od programera da piše blok za obradu u situacijama kada se ne može preduzeti ništa smisleno."*[9]

Kada objašnjava zašto deklaracija funkcije bez specifikacije znači da ona može generisati *bilo koji* izuzetak, a ne da *uopšte ne može* generisati izuzetak, Stroustrup kaže:

> *„To bi zahtevalo specifikacije izuzetaka za gotovo sve funkcije, predstavljalo bi značajan uzrok ponovnog prevođenja i sprečilo bi saradnju sa softverom napisanim na drugim jezicima. To bi navelo programere da sabotiraju mehanizme za obradu izuzetaka i da pišu maskirni kôd koji sakriva izuzetke. Stoga bi ljudi koji nisu uočili izuzetak stekli lažno osećanje bezbednosti."*[10]

Vidimo da se baš takvo ponašanje – sabotiranje izuzetaka – dešava s proverenim izuzecima u Javi.

Martin Fowler (autor knjiga *UML Distilled*, *Refactoring* i *Analysis Patterns*) napisao mi je sledeće:

> *„...sve u svemu, smatram da su izuzeci korisni, ali Javini provereni izuzeci više koštaju nego što vrede."*

Sada smatram da je važan doprinos Jave bio to što je objedinila model prijavljivanja grešaka, tako da se sve greške *prijavljuju* pomoću izuzetaka. Toga nije bilo u C++-u, zato što je, zbog kompatibilnosti sa C-om, još uvek bio na raspolaganju stari model prostog ignorisanja grešaka. Ali ako imate dosledno prijavljivanje pomoću izuzetaka, možete ih koristiti ako hoćete, a ako ne, oni će se popeti na najviši nivo (konzolu ili drugi kontejnerski

---

[8] http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820

[9] *Exception Handling in CLU*, Liskov & Snyder.

[10] Bjarne Stroustrup, *The C++ Programming Language, treće izdanje* (Addison-Wesley, 1997), s. 376.

program). Kada je Java modifikovala C++-ov model tako da su izuzeci postali jedini način prijavljivanja grešaka, dodatno pojačanje u vidu proverenih izuzetaka možda je postalo manje potrebno.

Ranije sam čvrsto verovao da su i provereni izuzeci i statička provera tipova neophodni za razvoj robusnih programa. Međutim, i anegdotsko i neposredno iskustvo[11] s jezicima koji su više dinamički nego statički, dovelo me je do stanovišta da velike prednosti zapravo potiču od:

1. Objedinjenog modela prijavljivanja grešaka preko izuzetaka, bez obzira na to da li prevodilac primorava programera da ih obradi.
2. Provere tipova, bez obzira na to kada se odigravala. Drugim rečima, dokle god se sprovodi propisna upotreba tipova, često nije važno da li se ona odigrava u vreme prevođenja ili u vreme izvršavanja.

Povrh svega toga, smanjenje ograničenja kojima je programer izvrgnut tokom prevođenja donosi veoma značajno povećanje produktivnosti. Štaviše, za kompenzaciju previše ograničavajuće statičke dodele tipova, potrebni su *refleksija* i *generički tipovi*, kao što ćete videti u brojnim primerima u celoj knjizi.

Već su mi rekli da je to što sam upravo napisao svetogrđe i da ću time uništiti svoju reputaciju i prouzrokovati propast civilizacija i neuspeh većeg procenta programskih projekata. Veruje se da prevodilac može spasiti projekat ukoliko ukaže na greške tokom prevođenja, ali je još važnije shvatiti ograničenja toga što prevodilac može da uradi; u dodatku koji ćete naći na *http://MindView.net/Books/BetterJava*, naglašavam vrednost automatskog postupka prevođenja i pakovanja programa, i testiranja sa @Unit, koji vam mnogo više pomažu nego kada sve pretvorite u sintaksnu pogrešku. Ne zaboravite:

> *„Dobar programski jezik je onaj koji pomaže programerima da pišu dobre programe. Nijedan programski jezik ne može sprečiti da se na njemu pišu loši programi."*[12]

U svakom slučaju, neznatna je verovatnoća da će provereni izuzeci ikada biti izbačeni iz Jave. To bi bila previše radikalna promena jezika, a u Sunu ima mnogo zagovornika proverenih izuzetaka. Sun je oduvek sprovodio politiku apsolutne vertikalne kompatibilnosti – da biste stekli osećaj za razmere toga, gotovo sav Sunov softver može da se izvršava na svom Sunovom hardveru, koliko god bio star. Međutim, ukoliko shvatite da vam neki provereni izuzeci smetaju ili naročito ako osećate da vas primoravaju da hvatate izuzetke s kojima ne znate šta da radite, može se i drugačije.

Prosleđivanje izuzetaka na konzolu

U jednostavnim programima, kakvi su mnogi u ovoj knjizi, najlakši način da sačuvate izuzetke, a da ne pišete gomilu koda, jeste prosleđivanje izuzetaka iz metode **main( )** na konzolu. Na primer, ako hoćete da otvorite datoteku za čitanje (o čemu ćete sve pojedinosti

---

[11] Indirektno, iskustvo s jezikom Smalltalk kroz razgovore s mnogim iskusnim programerima; direktno iskustvo s jezikom Python (*www.Python.org*).

[12] Kees Koster, projektant jezika CDL, čije reči navodi Bertrand Meyer, projektant jezika Eiffel, *www.elj.com/elj/v1/n1/bm/right/*.

saznati u poglavlju *Javin ulazno-izlazni sistem*), morate otvoriti i zatvoriti **FileInputStream** koji generiše izuzetke. U jednostavnom programu možete uraditi ovo (isti pristup je upotrebljen u mnogim primerima u ovoj knjizi):

```
//: izuzeci/MainIzuzetak.java
import java.io.*;

public class MainIzuzetak {
  // Prosledićemo sve izuzetke na konzolu:
  public static void main(String[] args) throws Exception {
    // Otvaranje datoteke:
    FileInputStream datoteka =
      new FileInputStream("MainIzuzetak.java");
    // Upotreba datoteke ...
    // Zatvori datoteku:
    datoteka.close();
  }
} ///:~
```

Imajte u vidu da je i **main( )** metoda koja može imati svoju specifikaciju izuzetaka, a ovde je tip izuzetka **Exception**, korenska klasa svih proverenih izuzetaka. Ako ih prosledite na konzolu, pošteđeni ste pisanja blokova **try-catch** unutar metode **main( )**. (Nažalost, U/I datoteka je znatno složenija nego što izgleda na osnovu ovog primera, pa nemojte previše da se uzbuđujete dok ne pročitate poglavlje *Javin ulazno-izlazni sistem*).

Vežba 26: (1) Izmenite znakovni niz imena datoteke u programu **MainIzuzetak.java** tako da glasi na ime nepostojeće datoteke. Pokrenite program i pogledajte šta će se desiti.

Pretvaranje proverenih izuzetaka u neproverene

Generisanje izuzetka iz metode **main( )** podesno je kada pišete jednostavne programe za vlastitu upotrebu, ali u opštem slučaju nije korisno. Pravi problem nastaje kada pišete telo obične metode pa pozivate drugu metodu i shvatite sledeće: „Nemam pojma šta da radim sa ovim izuzetkom ovde, a neću ni da ga progutam niti da ispisujem neku banalnu poruku." Uz ulančane izuzetke, nudi se jedno novo i jednostavno rešenje. Prosto „omotajte" proveren izuzetak u **RuntimeException** tako što ga prosledite konstruktoru klase **RuntimeException**. To se radi ovako:

```
try {
  // ... radite korisne stvari
} catch(NeZnamStaDaRadimSaOvimproverenim Izuzetkom e) {
  throw new RuntimeException(e);
}
```

Izgleda da je ovo idealno rešenje kada hoćete da „isključite" proveren izuzetak – nije progutan, ne morate da ga stavljate u specifikaciju izuzetaka svoje metode, a zbog ulančavanja izuzetaka ne gubite informacije iz originalnog izuzetka.

Ova tehnika omogućava ignorisanje izuzetka; on će se sam popeti po steku poziva metoda, a da ne morate da pišete blokove **try-catch** niti specifikacije izuzetaka. Međutim, i dalje možete metodom **getCause( )** uhvatiti i obraditi pojedine izuzetke, kao ovde:

```
//: izuzeci/IskljucivanjeProvere.java
// "Isključivanje" provere izuzetaka.
import java.io.*;
import static net.mindview.util.Print.*;

class OmotajproverenIzuzetak {
  void throwRuntimeException(int tip) {
    try {
      switch(tip) {
        case 0: throw new FileNotFoundException();
        case 1: throw new IOException();
        case 2: throw new RuntimeException("Gde sam?");
        default: return;
      }
    } catch(Exception e) { // Prilagođenje za neproverene:
      throw new RuntimeException(e);
    }
  }
}

class NekiDrugiIzuzetak extends Exception {}

public class IskljucivanjeProvere {
  public static void main(String[] args) {
    OmotajproverenIzuzetak opi = new OmotajproverenIzuzetak();
    // Možete pozvati throwRuntimeException() bez bloka try
    // i ostaviti RuntimeException da napusti metodu:
    opi.throwRuntimeException(3);
    // A možete odlučiti i da hvatate izuzetke:
    for(int i = 0; i < 4; i++)
      try {
        if(i < 3)
          opi.throwRuntimeException(i);
        else
          throw new NekiDrugiIzuzetak();
      } catch(NekiDrugiIzuzetak e) {
          print("NekiDrugiIzuzetak: " + e);
      } catch(RuntimeException re) {
        try {
          throw re.getCause();
        } catch(FileNotFoundException e) {
          print("FileNotFoundException: " + e);
        } catch(IOException e) {
          print("IOException: " + e);
```

```
        } catch(Throwable e) {
          print("Throwable: " + e);
        }
      }
    }
  } /* Ispis:
  FileNotFoundException: java.io.FileNotFoundException
  IOException: java.io.IOException
  Throwable: java.lang.RuntimeException: Gde sam?
  NekiDrugiIzuzetak: NekiDrugiIzuzetak
  *///:~
```

OmotajproverenIzuzetak.throwRuntimeException( ) sadrži kôd koji generiše izuzetke različitih tipova. Oni se hvataju i omotavaju u **RuntimeException** objekte, pa postaju „uzrok" (engl. *cause*) tih izuzetaka.

U programu **IskljucivanjeProvere** vidite da je moguće pozvati **throwRuntimeException( )** bez bloka **try**, pošto ta metoda ne generiše proverene izuzetke. Međutim, kada ste spremni za hvatanje izuzetaka, i dalje možete da uhvatite svaki izuzetak koji hoćete – samo napišite kôd unutar bloka **try**. Počinjete od svih izuzetaka za koje izričito znate da se mogu pojaviti iz koda u vašem bloku **try** – u ovom slučaju, prvo se hvata **NekiDrugiIzuzetak**. Na kraju, hvatate **RuntimeException** i generišete (rezervisana reč **throw**) rezultat metode **getCause( )** (omotani izuzetak). Time se izdvajaju prvobitni izuzeci koji se potom mogu obrađivati u sopstvenim blokovima **catch**.

Tehniku omotavanja proverenog izuzetka u **RuntimeException** upotrebljavaćemo u ostatku knjige kad god bude umesno. Drugo rešenje je da napravite sopstvenu potklasu od **RuntimeException**. Na taj način ne morate vi da hvatate izuzetak, a može da ga uhvati ko bude hteo.

Vežba 27: (1) Prepravite vežbu 3 tako da se izuzetak pretvara u **RuntimeException**.

Vežba 28: (1) Prepravite vežbu 4 tako da namenska klasa izuzetaka nasledi **RuntimeException** i pokažite da tada prevodilac dozvoljava da izostavite blok **try**.

Vežba 29: (1) Prepravite sve tipove izuzetaka u programu **StormyInning.java** tako da prošire (rezervisana reč **extend**) **RuntimeException**, i pokažite da tada nisu potrebne ni specifikacije izuzetaka niti blokovi **try**. Uklonite komentare **//!** i pokažite kako se metode mogu prevesti bez specifikacija.

Vežba 30: (2) Prepravite program **Covek.java** tako da izuzeci naslede **RuntimeException**. Izmenite **main( )** tako da se tehnika iz programa **IskljucivanjeProvere.java** upotrebljava za obradu različitih tipova izuzetaka.

Uputstva za izuzetke

Koristite izuzetke da:

1. Obradite probleme na odgovarajućem nivou. (Izbegavajte hvatanje izuzetaka s kojima ne znate šta da radite.)
2. Rešite problem i ponovo pozovete metodu koja je prouzrokovala izuzetak.

3. Zakrpite grešku i nastavite rad bez ponovnog isprobavanja metode.
4. Izračunate alternativan rezultat umesto onoga koji je metoda trebalo da vrati.
5. Uradite sve što možete u tekućem okruženju i ponovo generišete *isti* izuzetak ka višem okruženju.
6. Uradite sve što možete u tekućem okruženju i ponovo generišete *drugačiji* izuzetak ka višem okruženju.
7. Završite program.
8. Pojednostavite kôd. (Ako način na koji generišete izuzetke još više komplikuje kôd, onda ga nije lako i poželjno koristiti.)
9. Povećate pouzdanost biblioteke i programa. (Ovo je kratkoročno ulaganje kad je u pitanju uklanjanje grešaka, a dugoročno ulaganje u snagu aplikacije.)

Sažetak

Izuzeci su nezaobilazan deo programiranja na Javi; ako ne znate da radite s njima, nećete baš mnogo postići. Zato smo izuzetke predstavili na ovom mestu – ima mnogo biblioteka (kao ranije spomenuta biblioteka U/I) koje uopšte ne možete da upotrebljavate bez poznavanja izuzetaka.

Jedna od prednosti obrade izuzetaka jeste sledeća: omogućava da se na jednom mestu usredsredite na problem koji pokušavate da rešite, a da se na drugom mestu bavite greškama u tom kodu. I mada se izuzeci po pravilu smatraju alatkama koje omogućavaju *prijavljivanje* i *oporavak od* grešaka u vreme izvršavanja, pitam se koliko često se aspekat „oporavka" realizuje, pa čak i koliko često je to uopšte moguće. Po mom osećaju, to je manje od 10 posto vremena, pa čak i tada se verovatno svodi na otpetljavanje steka do poznatog stabilnog stanja, a ne do stvarnog nastavljanja rada. Bila to istina ili ne, smatram da osnovna vrednost izuzetaka leži u njihovoj funkciji „prijavljivanja". Činjenica da u Javi sve greške valja prijavljivati u obliku izuzetaka daje joj veliku prednost nad jezicima poput C++-a, koji dozvoljavaju prijavljivanje grešaka na više različitih načina, a možete i da ih uopšte ne prijavite. Dosledan sistem za prijavljivanje grešaka znači da se više ne morate pitati: „Da li su mi negde procurile greške?" nakon pisanja svakog parčeta koda (ukoliko ne „progutate" izuzetke, naravno!).

Kao što ćete videti u sledećim poglavljima, ukoliko rešite ovo pitanje – čak i ako to uradite generišući **RuntimeException** – svoj trud pri projektovanju i realizaciji možete usredsrediti na zanimljivije i teže probleme.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.BruceEckel.com*.

Znakovni nizovi

TVRDI SE DA OBRADA ZNAKOVNIH NIZOVA SPADA MEĐU NAJČEŠĆE AKTIVNOSTI U programiranju. To je pogotovo tačno u Web sistemima, gde se Java mnogo koristi. U ovom poglavlju detaljnije ćemo razmotriti najčešće korišćenu klasu tog jezika, **String**, kao i neke njene prateće i pomoćne metode.

Nepromenljivi znakovni nizovi

Objekti klase **String** su nepromenljivi. U JDK dokumentaciji klase **String**, videćete da svaka njena metoda koja naizgled menja znakovni niz, zapravo pravi i vraća potpuno nov objekat tipa **String** koji sadrži modifikaciju. Originalni **String** se ostavlja neizmenjen.

Pogledajte sledeći kôd:

```
//: znakovninizovi/Nepromenljiv.java
import static net.mindview.util.Print.*;

public class Nepromenljiv {
  public static String velslova(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = "zdravo";
    print(q); // zdravo
    String qq = upcase(q);
    print(qq); // ZDRAVO
    print(q); // zdravo
  }
} /* Ispis:
zdravo
ZDRAVO
zdravo
*///:-
```

Kada se **q** prosledi metodi **velslova**( ), zapravo se prosleđuje kopija reference od **q**. Objekat s kojim je ta referenca povezana ostaje na istom fizičkom mestu. Reference se kopiraju prilikom prosleđivanja.

Pogledamo li definiciju metode **velslova**( ), videćemo da je prosleđena referenca nazvana **s** i da postoji samo dok se izvršava telo metode **velslova**( ). Kada se izvršavanje metode **velslova**( ) završi, lokalna referenca **s** iščezava. Metoda **velslova**( ) vraća rezultat, a to je originalni znakovni niz u kojem su sva slova pretvorena u velika. Naravno, ona zapravo vraća referencu tog rezultata. Ali ispostavlja se da je referenca koju vraća povezana s novim objektom, a da je originalni **q** ostavljen na miru.

Takvo ponašanje obično hoćemo. Pretpostavimo da kažete:

```
String s = "asdf";
String x = Nepromenljiv.velslova(s);
```

Da li zaista želite da metoda **velslova( )** *izmeni* svoj argument? Onome ko čita kôd, argument obično izgleda kao podatak dat metodi na znanje, a ne nešto što će biti izmenjeno. Ovo je važna garancija, jer se zbog nje kôd lakše čita i razume.

Poređenje preklapanja operatora + i StringBuildera

Pošto su objekti tipa **String** nepromenljivi, možete napraviti proizvoljan broj pseudonima (alijasa) određenog znakovnog niza. S obzirom na to da je svaki znakovni niz samo za čitanje, ne postoji mogućnost da će jedna referenca izmeniti nešto što će uticati na druge reference.

Nepromenljivost može da utiče na efikasnost. Primer za to je operator +, koji je preklopljen za objekte tipa **String**. Preklapanje znači da je operaciji dato drugo značenje kada se primeni na određenu klasu. (+ i += za znakovne nizove jedini su preklopljeni operatori u Javi, koja ne dozvoljava preklapanje drugih.)[1]

Operator + omogućava nadovezivanje znakovnih nizova:

```
//: znakovninizovi/Nadovezivanje.java

public class Nadovezivanje {
  public static void main(String[] args) {
    String mango = "mango";
    String s = "abc" + mango + "def" + 47;
    System.out.println(s);
  }
} /* Ispis:
abcmangodef47
*///:~
```

Pokušajte da zamislite kako bi to *moglo* da radi. **String** objekat „abc" bi mogao da ima metodu **append( )** koja pravi nov objekat tipa **String** koji sadrži „abc" nadovezan na sadržaj objekta **mango**. Novi objekat tipa **String** bi zatim napravio drugi novi **String** koji bi dodao „def" itd.

To bi svakako radilo, ali bi zahtevalo pravljenje mnogo **String** objekata samo da bi se sastavio nov znakovni niz, i ostalo bi mnogo pomoćnih znakovnih nizova koje treba sakupiti kao smeće. Pretpostavljam da su projektanti Jave najpre isprobali taj pristup (što je lekcija iz projektovanja softvera – o sistemu zapravo ne znate ništa dok ga ne isprobate u kodu i ne naterate da radi). Pretpostavljam i da su videli kako su performanse takvog pristupa neprihvatljive.

Da biste videli šta se zaista dešava, gornji kôd možete prevesti unazad alatkom **javap** koja se isporučuje kao deo razvojnog okruženja JDK. Ovako treba da izgleda komandna linija:

```
javap -c Nadovezivanje
```

[1] C++ dozvoljava programerima da preklapaju operatore kako god hoće. Pošto je to često komplikovan postupak (videti poglavlje 10 knjige *Misliti na jeziku C++*, Mikro knjiga, 2003), projektanti Jave su ga proglasili „lošim" i zabranili u Javi. Toliko loše baš i nije, pošto su to na kraju i sami uradili, a da ironija bude veća, preklapanje operatora bi u Javi bilo mnogo lakše nego u C++-u. To se može videti u Pythonu (videti *www.Python.org*) i C#-u, koji imaju skupljanje i uklanjanje smeća te jednostavno preklapanje operatora.

Indikator **-c** daje bajtkodove JVM-a. Kada odbacimo delove koji nas ne zanimaju i malo uredimo ostatak, evo kako glase relevantni bajtkodovi:

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=3, Args_size=1
   0:   ldc #2; //String mango
   2:   astore_1
   3:   new #3; //class StringBuilder
   6:   dup
   7:   invokespecial #4; //StringBuilder."<init>":()
   10:  ldc #5; //String abc
   12:  invokevirtual #6; //StringBuilder.append:(String)
   15:  aload_1
   16:  invokevirtual #6; //StringBuilder.append:(String)
   19:  ldc #7; //String def
   21:  invokevirtual #6; //StringBuilder.append:(String)
   24:  bipush 47
   26:  invokevirtual #8; //StringBuilder.append:(I)
   29:  invokevirtual #9; //StringBuilder.toString:()
   32:  astore_2
   33:  getstatic #10; //Field System.out:PrintStream;
   36:  aload_2
   37:  invokevirtual #11; // PrintStream.println:(String)
   40:  return
```

Ukoliko ste radili sa asemblerom, ovo bi trebalo da vam izgleda poznato – naredbe poput **dup** i **invokevirtual** pripadaju asembleru Javine virtuelne mašine (JVM). Ako prvi put u životu vidite asembler, ne upuštajte se – važno je uočiti da prevodilac uvodi klasu **java.lang.StringBuilder**. U izvornom kodu se **StringBuilder** nije spominjao, ali je prevodilac odlučio da ga ipak upotrebi, zato što je mnogo efikasniji.

U ovom slučaju, prevodilac pravi objekat tipa **StringBuilder** da bi napravio znakovni niz **s** i četiri puta poziva **append( )**, po jednom za svaki od delova. Najzad, poziva **toString( )** da bi napravio rezultat, koji sprema (asemblerskom naredbom **astore_2**) kao **s**.

Pre nego što zaključite da je dovoljno svuda upotrebljavati objekte tipa **String** i prepustiti prevodiocu da se stara za efikasnost, pogledajmo šta radi prevodilac. Evo primera koji pravi **String** na dva načina: koristeći objekte tipa **String**, odnosno ručnim programiranjem pomoću klase **StringBuilder**:

```
//: znakovninizovi/KudaStringBuilder.java

public class KudaStringBuilder {
  public String implicitno(String[] polja) {
    String rezultat = "";
    for(int i = 0; i < polja.length; i++)
      rezultat += polja[i];
    return rezultat;
  }
```

```
  public String eksplicitno(String[] polja) {
    StringBuilder rezultat = new StringBuilder();
    for(int i = 0; i < polja.length; i++)
      rezultat.append(polja[i]);
    return rezultat.toString();
  }
} ///:~
```

Ako sada pokrenete **javap -c WitherStringBuilder**, videćete (pojednostavljen) kôd te dve različite metode. Prvo, **implicitno( )**:

```
public java.lang.String implicitno(java.lang.String[]);
  Code:
   0:   ldc #2; //String
   2:   astore_2
   3:   iconst_0
   4:   istore_3
   5:   iload_3
   6:   aload_1
   7:   arraylength
   8:   if_icmpge 38
   11:  new #3; //class StringBuilder
   14:  dup
   15:  invokespecial #4; // StringBuilder."<init>":()
   18:  aload_2
   19:  invokevirtual #5; // StringBuilder.append:()
   22:  aload_1
   23:  iload_3
   24:  aaload
   25:  invokevirtual #5; // StringBuilder.append:()
   28:  invokevirtual #6; // StringBuilder.toString:()
   31:  astore_2
   32:  iinc 3, 1
   35:  goto 5
   38:  aload_2
   39:  areturn
```

Obratite pažnju na redove **8:** i **35:** koji zajedno čine petlju. Red **8:** sadrži naredbu „integer compare greater than or equal to" (celobrojnog poređenja veće ili jednako) operanada na steku i skače na **38:** kada petlja završi rad. Red **35:** (naredba goto) je vraćanje na početak petlje, red **5:**. Važno je primetiti da se konstrukcija **StringBuilder**a odigrava *unutar* te petlje, što znači da ćete dobiti nov objekat tipa **StringBuilder** svaki put kada prođete kroz petlju.

Ovo su bajtkod metode **eksplicitno( )**:

```
public java.lang.String eksplicitno(java.lang.String[]);
  Code:
   0:   new #3; //class StringBuilder
   3:   dup
   4:   invokespecial #4; // StringBuilder."<init>":()
   7:   astore_2
```

```
8:   iconst_0
9:   istore_3
10:  iload_3
11:  aload_1
12:  arraylength
13:  if_icmpge 30
16:  aload_2
17:  aload_1
18:  iload_3
19:  aaload
20:  invokevirtual #5; // StringBuilder.append:()
23:  pop
24:  iinc 3, 1
27:  goto 10
30:  aload_2
31:  invokevirtual #6; // StringBuilder.toString:()
34:  areturn
```

Ne samo da je petlja kraća i jednostavnija, nego metoda pravi samo jedan objekat tipa **StringBuilder**. Eksplicitno pravljenje **StringBuilder**a omogućava da unapred zadate njegovu veličinu ukoliko znate koliki treba da bude, tako da ne mora uvek iznova da pravi sebi bafer.

Dakle, kada pravite metodu **toString( )**, ako su operacije jednostavne pa prevodilac može sam da ih shvati, po pravilu prepustite njemu da rezultat izračuna na razborit način. Ali ako program obuhvata kruženje kroz petlju, u metodi **toString( )** treba eksplicitno da navedete **StringBuilder**, ovako:

```
//: znakovninizovi/UzStringBuilder.java
import java.util.*;

public class UzStringBuilder {
  public static Random slucajan = new Random(47);
  public String toString() {
    StringBuilder rezultat = new StringBuilder("[");
    for(int i = 0; i < 25; i++) {
      rezultat.append(slucajan.nextInt(100));
      rezultat.append(", ");
    }
    rezultat.delete(rezultat.length()-2, rezultat.length());
    rezultat.append("]");
    return rezultat.toString();
  }
  public static void main(String[] args) {
    UzStringBuilder usb = new UzStringBuilder();
    System.out.println(usb);
  }
} /* Ispis:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9, 78, 98, 61, 20,
58, 16, 40, 11, 22, 4]
*///:~
```

Obratite pažnju na to da se svaki deo rezultata dodaje po jednom naredbom **append( )**. Ako pokušate prečicom i napišete nešto kao **append(a + ": " + c)**, umešaće se prevodilac i ponovo početi da pravi nove objekte tipa **StringBuilder**.

Ukoliko niste sigurni koji pristup da odaberete, uvek možete da pokrenete **javap** i tako saznate.

Iako **StringBuilder** ima sve potrebne metode, među kojima su **insert( )**, **replace( )**, **substring( )**, pa čak i **reverse( )**, najčešće ćete upotrebljavati **append( )** i **toString( )**. Obratite pažnju na upotrebu metode **delete( )** za uklanjanje poslednjeg zareza i razmaka pre dodavanja završne uglaste zagrade.

Klasa **StringBuilder** je uvedena u Javi SE5. Pre nje, Java je imala klasu **StringBuffer**, koja se starala za bezbedno paralelno izvršavanje (videti poglavlje *Paralelno izvršavanje*) i stoga je bila znatno skuplja. Zato bi operacije sa znakovnim nizovima u Javi SE5/6 trebalo da budu znatno brže.

Vežba 1: (2) Analizirajte metodu **Prskalica.toString( )** u programu **ponovnaupotreba/Prskalica.java** i ispitajte da li bi se metodom **toString( )** sa eksplicitnim pravljenjem klase **StringBuilder** uštedelo na pravljenju objekata tipa **StringBuilder**.

Nenamerna rekurzija

Pošto su Javini standardni kontejneri, kao i sve druge klase, nasleđeni iz klase **Object**, oni sadrže metodu **toString( )**. Ona je redefinisana da bi kontejneri, kao i objekti koje sadrže, mogli da se predstave u obliku znakovnog niza, tj. tipa **String**. Unutar klase **ArrayList**, na primer, metoda **toString( )** prolazi kroz elemente kontejnera i poziva metodu **toString( )** svakog elementa.

```
//: znakovni_nizovi/IspisivanjeArrayListe.java
import generics.coffee.*;
import java.util.*;

public class IspisivanjeArrayListe {
  public static void main(String[] args) {
    ArrayList<Coffee> kafe = new ArrayList<Coffee>();
    for(Coffee c : new CoffeeGenerator(10))
      kafe.add(c);
    System.out.println(kafe);
  }
} /* Ispis:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve 5,
Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
*///:~
```

Zamislimo da želite da vaša metoda **toString( )** ispiše adresu ove klase. Čini se da ima smisla pozvati **this**:

```
//: znakovni_nizovi/BeskonacnaRekurzija.java
// Slučajna rekurzija.
// {RunByHand}
```

```
import java.util.*;

public class BeskonacnaRekurzija {
  public String toString() {
    return " Adresa objekta klase BeskonacnaRekurzija: "
      + this + "\n";
  }
  public static void main(String[] args) {
    List<BeskonacnaRekurzija> v = new ArrayList<BeskonacnaRekurzija>();
    for(int i = 0; i < 10; i++)
      v.add(new BeskonacnaRekurzija());
    System.out.println(v);
  }
} ///:~
```

Ako napravite objekat klase **BeskonacnaRekurzija** i zatim ga ispišete, dobićete beskrajan niz izuzetaka. To će se desiti i ako stavite objekte klase **BeskonacnaRekurzija** u neki kontejner klase **ArrayList** i ispišete taj kontejner kao što je ovde prikazano. Zapravo se dešava automatska konverzija tipa u **String**. Kada kažete:

```
"Adresa objekta klase BeskonacnaRekurzija: " + this
```

prevodilac vidi **String** za kojim sledi + i nešto što nije **String**, pa pokušava da konvertuje **this** u **String**. Konverzija se obavlja pozivom metode **toString( )** koja daje rekurzivan poziv.

Ako zaista želite da ispišete adresu objekta u ovom slučaju, rešenje je da se pozove metoda **Object.toString( )**, koja to radi. Dakle, umesto **this**, treba napisati **super.toString( )**.

Vežba 2: (1) Popravite program **BeskonacnaRekurzija.java**.

Operacije sa znakovnim nizovima

Ovo su neke od osnovnih metoda za **String** objekte. Preklopljene metode su opisane u istom redu tabele:

Metoda	Argumenti, preklapanje	Upotreba
Konstruktor	**Preklopljene**. podrazumevani, **String**, **StringBuilder**, **StringBuffer**, **char** nizovi, **byte** nizovi.	Pravljenje **String** objekata.
length( )		Broj znakova u objektu tipa **String**
charAt( )	**int** indeks	Znak (**char**) na zadatom mestu znakovnog niza.
getChars( ), getBytes( )	Početak i kraj podniza koji treba kopirati, niz u koji treba kopirati, indeks mesta u odredišnom nizu.	Kopiranje objekata tipa **char** ili **byte** u spoljni niz.
toCharArray( )		Pravi **char[]** sa znakovima koje sadrži **String**
equals( ), equalsIgnoreCase( )	**String** za poređenje.	Ispitivanje jednakosti sadržaja dva znakovna niza.

Metoda	Argumenti, preklapanje	Upotreba
compareTo( )	**String** za poređenje.	Rezultat je negativan, nula, ili pozitivan, u zavisnosti od leksikografskog uređenja **String**a i argumenta. Velika i mala slova nisu jednaka!
contains( )	Objekat tipa **CharSequence** koji se traži.	Rezultat je **true** ako **String** sadrži argument.
contentEquals( )	Objekat tipa **CharSequence** ili **StringBuffer** za poređenje.	Rezultat je **true** ako se argument tačno podudara.
equalsIgnoreCase( )	**String** za poređenje	Rezultat je **true** ako su sadržaji jednaki; zanemaruje se veličina slova.
regionMatches( )	Pomeraj od početka ovog **String**a, drugi **String** i pomeraj od njegovog početka, i dužina koja se poredi. Preklapanjem se dodaje zanemarenje veličine slova.	Rezultat tipa **boolean** pokazuje da li se dato područje podudara.
startsWith( )	**String** s kojim eventualno počinje. Preklapanje dodaje pomeraj (ofset) u argument.	Rezultat tipa **boolean** pokazuje da li **String** počinje argumentom.
endsWith( )	**String** koji bi mogao biti sufiks od **String**	Rezultat tipa **boolean** pokazuje da li je argument neki sufiks.
indexOf( ), lastIndexOf( )	Preklopljen: **char**, **char** i početni indeks, **String**, **String** i početni indeks.	Vraća -1 ako se argument ne pronađe u ovom **String**u; u protivnom, vraća indeks početka argumenta. **lastIndexOf( )** pretražuje unazad, od kraja.
substring( ) (takođe i subSequence( ))	Preklopljen: početni indeks; početni indeks + završni indeks.	Vraća nov objekat tipa **String** koji sadrži dati skup znakova.
concat( )	**String** koji treba nadovezati.	Vraća nov objekat tipa **String** koji sadrži znakove originalnog **String**a i zatim znakove argumenta
replace( )	Stari znak koji se traži, novi znak s kojim ga treba zameniti. Može zameniti i jedan **CharSequence** drugim	Vraća nov objekat tipa **String** sa sprovedenim zamenama. Vraća stari **String** ako ne nađe ono što treba da zameni.
toLowerCase( ) toUpperCase( )		Vraća nov objekat tipa **String** sa svim malim, odnosno velikim slovima. Vraća stari **String** ako nema šta da izmeni.
trim( )		Vraća nov objekat tipa **String** bez belina na oba kraja. Vraća stari **String** ako nema šta da izmeni.
valueOf( )	Preklopljen: **Object**, **char[]**, **char[]** i pomeraj od početka i broj znakova, **boolean**, **char**, **int**, **long**, **float**, **double**	Vraća **String** koji sadrži tekstualni prikaz argumenta.
intern( )		Pravi tačno jednu **String** referencu za svaku jedinstvenu sekvencu znakova.

Vidite da svaka **String** metoda vraća nov **String** objekat kada treba da izmeni sadržaj znakovnog niza. Imajte u vidu i sledeće: ukoliko sadržaj ne treba menjati, metoda prosto vraća referencu na originalni **String**. Time se štedi memorija i drugi resursi.

U nastavku poglavlja objasnićemo metode klase **String** koje rade s *regularnim izrazima*.

Formatiranje izlaza

Jedna od dugo priželjkivanih mogućnosti koja se najzad pojavila u Javi SE5 jeste formatiranje izlaza u stilu C-ove naredbe **printf( )**. Ne samo da to pojednostavljuje izlazni kôd, nego programerima u Javi daje moćnu alatku za određivanje formatiranja i poravnavanja izlaza.[2]

Metoda printf()

C-ova funkcija **printf( )** ne sastavlja nizove kao Java, nego prima *znakovni niz za formatiranje* i u njega umeće vrednosti, formatirajući u hodu. Umesto da za nadovezivanje teksta i promenljivih unutar navodnika upotrebljava preklopljen operator + (koji u C-u nije preklopljen), **printf( )** upotrebljava posebne oznake koje pokazuju gde treba umetnuti podatke. Iza njih sledi lista argumenata koji se umeću u znakovni niz za formatiranje, međusobno odvojenih zarezima.

Na primer:

```
printf("Red 1: [%d %f]\n", x, y);
```

Prilikom izvršavanja, vrednost od **x** se umeće u %d, a vrednost od y u **%f**. Te oznake se nazivaju *specifikatori formata* i, sem što kazuju gde treba umetnuti vrednost, one saopštavaju i vrstu promenljive koju treba umetnuti i kako je treba formatirati. Na primer, gornje **%d** kazuje da je **x** celi broj, a **%f** kazuje da je **y** neki broj zapisan u formatu pokretnog zareza (**float** ili **double**).

System.out.format()

U Javu SE5 je uvedena metoda **format( )**, dostupna objektima tipa **PrintStream** ili **PrintWriter** (o kojima ćete više saznati u poglavlju *Javin ulazno-izlazni sistem*), koja obuhvata standardni izlazni tok **System.out**. Metoda **format( )** je napravljena po uzoru na C-ovu funkciju **printf( )**. Za nostalgičare su napravili čak i metodu **printf( )** koja samo poziva metodu **format( )**. Evo jednostavnog primera:

```
//: znakovninizovi/JednostavnoFormatiranje.java

public class JednostavnoFormatiranje {
  public static void main(String[] args) {
    int x = 5;
    double y = 5.332542;
```

[2] U pisanju ovog odeljka, kao i odeljka „Skeniranje ulaza", pomogao mi je Mark Welsh.

```
    // Stari način:
    System.out.println("Red 1: [" + x + " " + y + "]");
    // Novi način:
    System.out.format("Red 1: [%d %f]\n", x, y);
    // ili
    System.out.printf("Red 1: [%d %f]\n", x, y);
  }
} /* Ispis:
Red 1: [5 5.332542]
Red 1: [5 5.332542]
Red 1: [5 5.332542]
*///:~
```

Vidite da su **format( )** i **printf( )** ekvivalentne. U oba slučaja postoji samo jedan znakovni niz za formatiranje, iza kojega sledi po jedan argument za svaki specifikator formata.

Klasa **Formatter**

Svu novu Javinu funkcionalnost za formatiranje sadrži klasa **Formatter** iz paketa **java.util**. Klasu **Formatter** možete smatrati prevodiocem koji znakovni niz za formatiranje i podatke pretvara u željeni rezultat. Kada pravite objekat tipa **Formatter**, njegovom konstruktoru prosledite podatak gde da pošalje rezultat:

```
//: znakovninizovi/Kornjaca.java
import java.io.*;
import java.util.*;

public class Kornjaca {
  private String ime;
  private Formatter f;
  public Kornjaca(String ime, Formatter f) {
    this.ime = ime;
    this.f = f;
  }
  public void pomeriNa(int x, int y) {
    f.format("Kornjača %s je na (%d,%d)\n", ime, x, y);
  }
  public static void main(String[] args) {
    PrintStream nadimakZaIzlaz = System.out;
    Kornjaca tom = new Kornjaca("Tom",
      new Formatter(System.out));
    Kornjaca teo = new Kornjaca("Teo",
      new Formatter(nadimakZaIzlaz));
    tom.pomeriNa(0,0);
    teo.pomeriNa(4,8);
    tom.pomeriNa(3,4);
    teo.pomeriNa(2,5);
    tom.pomeriNa(3,3);
    teo.pomeriNa(3,3);
  }
```

```
} /* Ispis:
Kornjača Tom je na (0,0)
Kornjača Teo je na (4,8)
Kornjača Tom je na (3,4)
Kornjača Teo je na (2,5)
Kornjača Tom je na (3,3)
Kornjača Teo je na (3,3)
*///:~
```

Sav izlaz za kornjaču **tom** ide na **System.out**, a onaj za kornjaču **teo** na nadimak za **System.out**. Konstruktor je preklopljen tako da prima više izlaznih lokacija, ali najkorisnije su **PrintStream** (kao gore), **OutputStream** i **File**. O njima ćete saznati više u poglavlju *Javin ulazno-izlazni sistem*.

Vežba 3: (1) Prepravite program **Kornjaca.java** tako da sav izlaz šalje na **System.err**.

U prethodnom primeru upotrebljen je nov specifikator formata, **%s**. On ukazuje na argument tipa **String** i primer je najjednostavnije vrste specifikatora formata – onoga koji navodi samo tip konverzije.

Specifikatori formata

Za određivanje razmaka i poravnavanja prilikom umetanja podataka potrebni su složeniji specifikatori formata. Ovo je njihova opšta sintaksa:

```
%[indeks_argumenta$][indikatori][širina][.preciznost]konverzija
```

Najmanju veličinu polja zadajete parametrom širina. **Formatter** jemči da će polje biti široko najmanje određen broj znakova tako što po potrebi dodaje razmake. Podaci se podrazumevano poravnavaju udesno, ali to možete redefinisati stavljanjem – u odeljak indikatora.

Parametar *preciznost* je suprotan *širini*, jer zadaje maksimum. Za razliku od *širine* koja je primenljiva na sve tipove konverzije podataka i jednako radi za sve, *preciznost* ima razna značenja za različite tipove. Za znakovne nizove, specifikator *preciznost* zadaje najveći broj znakova objekta tipa **String** koje treba ispisati. Za brojeve s pokretnim zarezom, *preciznost* zadaje broj decimalnih mesta koje treba ispisati (podrazumeva se 6); ako decimala ima previše, Java će rezultat zaokružiti, a ako ih ima premalo, dodaće prateće nule. Pošto celi brojevi nemaju razlomljen deo, *preciznost* se na njih ne može primeniti, i dobićete izuzetak ukoliko upotrebite preciznost s konverzijom celobrojnog tipa.

U narednom primeru, specifikatori formata su upotrebljeni za ispisivanje računa za kupovinu:

```
//: znakovninizovi/Racun.java
import java.util.*;

public class Racun {
  private double ukupno = 0;
  private Formatter f = new Formatter(System.out);
  public void printTitle() {
```

```
    f.format("%-15s %5s %10s\n", "Artikal", "Kol", "Cena");
    f.format("%-15s %5s %10s\n", "----", "---", "-----");
  }
  public void print(String ime, int kol, double cena) {
    f.format("%-15.15s %5d %10.2f\n", ime, kol, cena);
    ukupno += cena;
  }
  public void printTotal() {
    f.format("%-15s %5s %10.2f\n", "Porez", "", ukupno*0.06);
    f.format("%-15s %5s %10s\n", "", "", "-----");
    f.format("%-15s %5s %10.2f\n", "Ukupno", "",
      ukupno * 1.06);
  }
  public static void main(String[] args) {
    Racun racun = new Racun();
    racun.printTitle();
    racun.print("Džekov čarobni pasulj", 4, 4.25);
    racun.print("Princezin grašak", 3, 5.1);
    racun.print("Kaša tri medveda", 1, 14.29);
    racun.printTotal();
  }
} /* Ispis:
Artikal          Kol       Cena
----             ---       -----
Džekov čarobni     4       4.25
Princezin graša    3       5.10
Kaša tri medved    1      14.29
Porez                      1.42
                          -----
Ukupno                    25.06
*///:~
```

Kao što vidite, **Formatter** je moćna alatka za određivanje razmaka i poravnavanja, a ima prilično sažetu notaciju. Ovde su znakovni nizovi za formatiranje prosto kopirani da bi se dobili odgovarajući razmaci.

Vežba 4: (3) Prepravite program **Racun.java** tako da sve širine određuje isti skup konstantnih vrednosti. Cilj je omogućiti laku promenu širine tako što će se izmeniti samo jedan broj na jednom mestu.

Konverzije klase **Formatter**

Najčešće ćete sretati sledeće konverzije:

Znakovi konverzije	
d	Celi broj (napisan decimalno)
c	Unicode znak
b	Logička vrednost (tipa Boolean)

Znakovi konverzije	
s	Znakovni niz
f	Broj s pokretnim zarezom (napisan decimalno)
e	Broj s pokretnim zarezom (napisan u naučnoj notaciji)
x	Celi broj (napisan heksadecimalno)
h	Ključ za heširanje (napisan heksadecimalno)
%	Literal %

Evo kako rade te konverzije:

```
//: znakovninizovi/Konverzija.java
import java.math.*;
import java.util.*;

public class Konverzija {
  public static void main(String[] args) {
    Formatter f = new Formatter(System.out);

    char u = 'a';
    System.out.println("u = 'a'");
    f.format("s: %s\n", u);
    // f.format("d: %d\n", u);
    f.format("c: %c\n", u);
    f.format("b: %b\n", u);
    // f.format("f: %f\n", u);
    // f.format("e: %e\n", u);
    // f.format("x: %x\n", u);
    f.format("h: %h\n", u);

    int v = 121;
    System.out.println("v = 121");
    f.format("d: %d\n", v);
    f.format("c: %c\n", v);
    f.format("b: %b\n", v);
    f.format("s: %s\n", v);
    // f.format("f: %f\n", v);
    // f.format("e: %e\n", v);
    f.format("x: %x\n", v);
    f.format("h: %h\n", v);

    BigInteger w = new BigInteger("50000000000000");
    System.out.println(
      "w = new BigInteger(\"50000000000000\")");
    f.format("d: %d\n", w);
    // f.format("c: %c\n", w);
    f.format("b: %b\n", w);
    f.format("s: %s\n", w);
    // f.format("f: %f\n", w);
```

```
    // f.format("e: %e\n", w);
    f.format("x: %x\n", w);
    f.format("h: %h\n", w);

    double x = 179.543;
    System.out.println("x = 179.543");
    // f.format("d: %d\n", x);
    // f.format("c: %c\n", x);
    f.format("b: %b\n", x);
    f.format("s: %s\n", x);
    f.format("f: %f\n", x);
    f.format("e: %e\n", x);
    // f.format("x: %x\n", x);
    f.format("h: %h\n", x);

    Konverzija y = new Konverzija();
    System.out.println("y = nova Konverzija()");
    // f.format("d: %d\n", y);
    // f.format("c: %c\n", y);
    f.format("b: %b\n", y);
    f.format("s: %s\n", y);
    // f.format("f: %f\n", y);
    // f.format("e: %e\n", y);
    // f.format("x: %x\n", y);
    f.format("h: %h\n", y);
    boolean z = false;
    System.out.println("z = false");
    // f.format("d: %d\n", z);
    // f.format("c: %c\n", z);
    f.format("b: %b\n", z);
    f.format("s: %s\n", z);
    // f.format("f: %f\n", z);
    // f.format("e: %e\n", z);
    // f.format("x: %x\n", z);
    f.format("h: %h\n", z);
  }
} /* Ispis: (primer)
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("50000000000000")
```

```
d: 50000000000000
b: true
s: 50000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = nova Konverzija()
b: true
s: Konverzija@9cab16
h: 9cab16
z = false
b: false
s: false
h: 4d5
*///:~
```

Redovi pretvoreni u komentare nisu validni za taj tip promenljive; njihovim izvršavanjem biste izazvali izuzetak.

Obratite pažnju na konverziju **b**, koja radi za sve gornje promenljive. Iako je validna za sve tipove argumenata, ne ponaša se kao što možda pretpostavljate. Za proste logičke tipove (**boolean**) i objekat tipa **Boolean**, njen rezultat je **true** ili **false**, u zavisnosti od vrednosti. Međutim, za sve drugačije argumente, rezultat je uvek **true** ukoliko tip argumenta nije **null**. Čak i numerička vrednost nula, koja je u mnogim jezicima (i C-u) sinonim za **false**, daje **true**, pa morate biti oprezni kada tu konverziju upotrebljavate s tipovima koji nisu logički.

Ima još tipova konverzija i drugih opcija specifikatora formata. To je opisano u dokumentaciji razvojnog okruženja JDK za klasu **Formatter**.

Vežba 5: (5) Za sve osnovne tipove konverzija iz gornje tabele, napišite najsloženiji mogući izraz za formatiranje, tj. upotrebite sve moguće specifikatore formata dostupne za taj tip konverzije.

String.format()

Java SE5 se ugledala i na C-ovu funkciju **sprintf( )** kojom se prave znakovni nizovi. **String.format( )** je statična metoda koja prima iste argumente kao **Formatterova** metoda **format( )**, ali vraća objekat tipa **String**. Podesna je kada metodu **format( )** treba da zovete samo jedanput:

```
//: znakovninizovi/IzuzetakBazePodataka.java

public class IzuzetakBazePodataka extends Exception {
  public IzuzetakBazePodataka (int IDtransakcije, int IDupita,
    String poruka) {
```

```
    super(String.format("(t%d, u%d) %s", IDtransakcije,
        IDupita, poruka));
  }
  public static void main(String[] args) {
    try {
      throw new IzuzetakBazePodataka(3, 7, "Upisivanje neuspešno");
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Ispis:
IzuzetakBazePodataka: (t3, u7) Upisivanje neuspešno
*///:~
```

String.format( ) samo napravi objekat tipa **Formatter** i prosledi mu vaše argumente, ali koristeći ovu metodu dobijate jasniji i razumljiviji kôd nego da sve to radite ručno.

Alatka za ispisivanje u heksadecimalnom formatu

Kao drugi primer, često se javlja potreba da čovek pogleda bajtove binarne datoteke u heksadecimalnom formatu. Evo malog uslužnog programa koji, koristeći **String.format( )**, ispisuje binarni niz bajtova u čitljivom heksadecimalnom formatu:

```
//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;

public class Hex {
  public static String format(byte[] podaci) {
    StringBuilder rezultat = new StringBuilder();
    int n = 0;
    for(byte b : podaci) {
      if(n % 16 == 0)
        rezultat.append(String.format("%05X: ", n));
      rezultat.append(String.format("%02X ", b));
      n++;
      if(n % 16 == 0) rezultat.append("\n");
    }
    rezultat.append("\n");
    return rezultat.toString();
  }
  public static void main(String[] args) throws Exception {
    if(args.length == 0)
      // Testiranje ispisivanjem datoteke ove klase:
      System.out.println(
        format(BinaryFile.read("Hex.class")));
    else
      System.out.println(
        format(BinaryFile.read(new File(args[0]))));
  }
```

```
} /* Ispis: (primer)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*///:~
```

Za otvaranje i učitavanje binarne datoteke u ovom programu upotrebljena je uslužna klasa **net.mindview.util.BinaryFile** koja će biti predstavljena u poglavlju *Javin ulazno-izlazni sistem*. Metoda **read**( ) vraća celu datoteku u obliku niza bajtova (**byte**).

Vežba 6: (2) Napravite klasu koja sadrži polja tipova **int**, **long**, **float** i **double**. Za tu klasu napravite metodu **toString**( ) koja upotrebljava **String.format**( ) i pokažite da klasa radi ispravno.

Regularni izrazi

Regularni izrazi su već dugo sastavni deo standardnih Unixovih alatki sed i awk, i jezika Python i Perl (neki tvrde da su oni glavni razlog što je Perl tako uspešan). U Javi su alatke za obradu znakovnih nizova ranije bile delegirane u klase **String**, **StringBuffer** i **StringTokenizer**. U poređenju s regularnim izrazima, te alatke su imale relativno skromne mogućnosti.

Regularni izrazi su moćne i fleksibilne alatke za obradu teksta. One omogućavaju programsko zadavanje složenih uzoraka teksta koji mogu biti pronađeni u ulaznom znakovnom nizu. Kada te uzorke pronađete, možete da reagujete kako god hoćete. Iako je sintaksa regularnih izraza isprva teška, njihov sažet dinamički jezik možete upotrebiti za rešavanje svih vrsta zadataka u obradi, pronalaženju, izboru, uređivanju i proveri znakovnih nizova, na potpuno uopšten način.

Osnove

Regularan izraz je način opisivanja znakovnih nizova na uopšten način, tako da možete reći: „Ako znakovni niz ima ovo u sebi, onda odgovara onome što tražim". Na primer, kako biste saopštili da ispred broja može, ali ne mora biti znak minus, pišete minus i iza njega znak pitanja:

```
-?
```

Da biste opisali celi broj, kažete da je to jedna ili više cifara. U regularnim izrazima, cifra (engl. *digit*) opisuje se sa **\d**. Ukoliko ste imali posla s regularnim izrazima u drugim jezicima, odmah ćete uočiti razliku u obradi obrnutih kosih crta. U drugim jezicima \\ znači: „Hoću da umetnem običnu (doslovnu, literalnu) kosu crtu u regularan izraz. Ne treba joj pridavati specijalno značenje." U Javi, \\ znači: „Umećem kosu crtu u regularan izraz, pa sledeći znak ima specijalno značenje". Na primer, regularan izraz za cifru je **\\d**. Ako hoćete da umetnete stvarnu kosu crtu, pišete \\\\. Međutim, za prelazak u novi red i tabulator piše se samo jedna kosa crta: \n\t.

+ označava „jedan ili više prethodnih izraza". Dakle, da biste rekli „možda znak minus, a iza njega jedna ili više cifara", pišete:

```
-?\\d+
```

Regularni izrazi se najjednostavnije koriste pomoću funkcionalnosti ugrađene u klasu **String**. Na primer, utvrdićemo da li se nekoliko znakovnih nizova podudara (engl. *match*) s gornjim regularnim izrazom:

```
//: znakovninizovi/PodudaranjeCelihBrojeva.java

public class PodudaranjeCelihBrojeva {
  public static void main(String[] args) {
    System.out.println("-1234".matches("-?\\d+"));
    System.out.println("5678".matches("-?\\d+"));
    System.out.println("+911".matches("-?\\d+"));
    System.out.println("+911".matches("(-|\\+)?\\d+"));
  }
} /* Ispis:
true
true
false
true
*///:~
```

Prva dva izraza odgovaraju, ali treći počinje znakom +, koji je sam po sebi legitiman, ali ne odgovara našem regularnom izrazu. Zato nam treba način da kažemo: „može početi znakom + ili -". U regularnim izrazima, zagrade imaju efekat grupisanja izraza, a vertikalna crta | znači logičko ili (OR). Dakle izraz

```
(-|\\+)?
```

znači da ovaj deo znakovnog niza može biti – ili + ili (zbog znaka **?**) ni jedno ni drugo. Pošto u regularnim izrazima znak + ima specijalno značenje, mora se pomoću \\ pretvoriti u izlaznu sekvencu, da bi se u izrazu pojavio kao običan znak.

U klasu **String** ugrađena je metoda **split( )**, korisna alatka za regularne izraze koja znači: „Iscepaj ovaj znakovni niz na delove koji se podudaraju s datim regularnim izrazom."

```
//: znakovninizovi/Cepanje.java
import java.util.*;

public class Cepanje {
  public static String vitezovi =
    "Then, when you have found the shrubbery, you must " +
    "cut down the mightiest tree in the forest... " +
    "with... a herring!";
  public static void split(String regiz) {
    System.out.println(
      Arrays.toString(vitezovi.split(regiz)));
  }
```

```
  public static void main(String[] args) {
    split(" "); // Ne mora da sadrži specijalne znakove
    split("\\W+"); // Znakovi koji ne mogu biti deo reči
    split("n\\W+"); // 'n' i znakovi koji ne mogu biti deo reči
  }
} /* Ispis:
[Then,, when, you, have, found, the, shrubbery,, you, must, cut, down, the,
mightiest, tree, in, the, forest..., with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you, must, cut, down, the,
mightiest, tree, in, the, forest, with, a, herring]
[The, whe, you have found the shrubbery, you must cut dow, the mightiest tree
i, the forest... with... a herring!]
*///:~
```

Prvo, imajte u vidu da obične znakove možete upotrebljavati kao regularne izraze – regularan izraz ne mora sadržati specijalne znakove, kao što vidite u prvom pozivu metode **split( )**, koji tekst cepa na svakom razmaku.

Drugi i treći poziv metode **split( )** sadrže **\W**, izraz za znak koji ne može biti deo reči (verzija napisana malim slovom, **\w**, obeležava znak koji može biti deo reči). Uočite da su u drugom slučaju uklonjeni znakovi interpunkcije. Treći poziv metode **split( )** kazuje: „slovo **n** i jedan ili više znakova koji ne mogu biti deo reči". Iz rezultata vidite da se u njemu ne pojavljuju uzorci definisani u regularnom izrazu.

Preklopljena verzija metode **String.split( )** omogućava zadavanje najvećeg dozvoljenog broja cepanja.

Poslednja alatka regularnih izraza ugrađena u klasu **String** jeste zamena. Možete zameniti prvi podudarni deo ili sve njih:

```
//: znakovninizovi/Zamena.java
import static net.mindview.util.Print.*;

public class Zamena {
  static String s = Splitting.vitezovi;
  public static void main(String[] args) {
    print(s.replaceFirst("f\\w+", "located"));
    print(s.replaceAll("shrubbery|tree|herring","banana"));
  }
} /* Ispis:
Then, when you have located the shrubbery, you must cut down the
mightiest tree in the forest... with... a herring!
Then, when you have found the banana, you must cut down the mightiest banana
in the forest... with... a banana!
*///:~
```

Prvom izrazu odgovara slovo **f** i jedan ili više znakova koji mogu biti deo reči (povedite računa o tome da je ovoga puta **w** malo slovo). Zamenjuje se samo prvi pronađeni podudarni deo, pa reč „found" biva zamenjena rečju „located."

Drugom izrazu odgovara bilo koja od tri reči koje su razdvojene vertikalnim crtama (logičkim OR), i zamenjuju se svi pronađeni podudarni delovi.

Videćete da još moćnije alatke za zamenu imaju regularni izrazi koji se ne upotrebljavaju za znakovne nizove – na primer, možete pozivati metode da one obavljaju zamene. U slučajevima kada se regularni izraz upotrebljava više puta, znatno su efikasniji regularni izrazi koji se ne upotrebljavaju za znakovne nizove.

Vežba 7: (5) Koristeći dokumentaciju klase **java.util.regex.Pattern** kao resurs, napišite i testirajte regularan izraz koji proverava da li rečenica počinje velikim slovom i završava se tačkom.

Vežba 8: (2) Pocepajte znakovni niz **Cepanje.vitezovi** tamo gde se nađu reči „the" ili „you."

Vežba 9: (4) Koristeći dokumentaciju klase **java.util.regex.Pattern** kao resurs, sve samoglasnike u **Cepanje.vitezovi** zamenite podcrtama.

Pravljenje regularnih izraza

Kad učite o regularnim izrazima, možete početi od podskupa svih mogućih tvorbi. Potpuna lista tvorbi za pravljenje regularnih izraza nalazi se u dokumentaciji razvojnog okruženja JDK za klasu **Pattern** iz paketa **java.util.regex**.

Znakovi	
B	Znak **B**
\xhh	Znak čiji je heksadecimalni kôd **oxhh**
\uhhhh	Unicode znak čiji je heksadecimalni kôd **oxhhhh**
\t	Tabulator
\n	Prelazak u novi red
\r	Vraćanje na početak tekućeg reda
\f	Prelazak na novi list
\e	Znak Esc (izlaz)

Snaga regularnih izraza najbolje se vidi u definisanju klasa znakova. Evo nekih tipičnih načina pravljenja klasa znakova i nekih unapred definisanih klasa:

Znakovi	
.	Bilo koji znak
[abc]	Bilo koji od znakova **a**, **b** ili **c** (isto što i **a|b|c**)
[^abc]	Bilo koji znak osim **a**, **b** ili **c** (negacija)
[a-zA-Z]	Bilo koji znak od **a** do **z** ili od **A** do **Z** (opseg)
[abc[hij]]	Bilo koji od znakova **a,b,c,h,i,j** (isto što i **a|b|c|h|i|j**) (unija)
[a-z&&[hij]]	Bilo koji od znakova **h**, **i** ili **j** (presek)
\s	Bilo koji od znakova za belinu (razmak, tabulator, prelazak u novi red, prelazak na novi list, vraćanje na početak tekućeg reda)
\S	Bilo koji znak sem znakova za belinu (**[^\s]**)

Znakovi	
\d	Numerička cifra [0-9]
\D	Bilo koji znak sem znakova za cifre [^0-9]
\w	Bilo koji znak koji može biti deo reči [a-zA-Z_0-9]
\W	Bilo koji znak sem znakova koji mogu biti deo reči [^\w]

Ovde smo pokazali samo mali deo mogućih regularnih izraza; da biste lako pristupali svim mogućim uzorcima regularnih izraza, stranicu JDK dokumentacije za **java.util.regex.Pattern** pretvorite u obeleživač.

Logički operatori	
XY	X, a potom Y
X|Y	X ili Y
(X)	Grupa koju treba pronaći. U nastavku izraza, itu grupu za hvatanje označavate sa \i.

Izrazi za granice	
^	Početak reda
$	Kraj reda
\b	Granica reči
\B	Granica koja ne može biti deo reči
\G	Kraj prethodno pronađene podudarnosti

Kao primer, svi sledeći izrazi se podudaraju s nizom znakova „Rudolph":

```
//: znakovninizovi/Rudolph.java

public class Rudolph {
  public static void main(String[] args) {
    for(String uzorak : new String[]{ "Rudolph",
      "[rR]udolph", "[rR][aeiou][a-z]ol.*", "R.*" })
      System.out.println("Rudolph".matches(uzorak));
  }
} /* Ispis:
true
true
true
true
*///:~
```

Naravno, ne bi trebalo pisati najteže shvatljiv regularan izraz, nego najjednostavniji koji „završava posao". Kada počnete da pišete nove regularne izraze, verovatno ćete često upotrebljavati svoj kôd kao referencu.

Kvantifikatori

Kvantifikator opisuje način na koji uzorak apsorbuje ulazni tekst:

- *Pohlepno*: „ukoliko se drugačije ne naredi", kvantifikatori su pohlepni (engl. *greedy*). Pohlepan izraz pronalazi sve moguće podudarnosti sa uzorkom. Tipičan uzrok problema jeste pretpostavka da će se izraz podudariti samo s prvom podudarnom grupom znakova, a on je zapravo pohlepan i nastaviće da traži sve dok ne pronađe najveći mogući podudarni znakovni niz.
- *Rezervisano*: Ovaj kvantifikator se obeležava znakom pitanja, a pronalazi minimalan broj znakova potreban za podudaranje sa uzorkom. Naziva se i *lenj, minimalno podudarajući, nepohlepan.*
- *Posesivno*: trenutno dostupan samo u Javi (u drugim jezicima ne) i napredniji, pa ga verovatno nećete odmah upotrebljavati. Kada se regularan izraz primeni na znakovni niz, on generiše mnogo stanja da bi mogao da se vrati ako ne uspe da nađe podudarnost. Posesivni kvantifikatori ne čuvaju ta međustanja i time sprečavaju vraćanje. Možete ih upotrebiti kako biste sprečili regularan izraz da podivlja, a i da bi radio efikasnije.

Pohlepni	Rezervisani	Posesivni	Podudara se sa
X?	X??	X?+	X, jednom ili nula puta
X*	X*?	X*+	X, nula ili više puta
X+	X+?	X++	X, jednom ili više puta
X{n}	X{n}?	X{n}+	X, tačno n puta
X{n,}	X{n,}?	X{n,}+	X, najmanje n puta
X{n,m}	X{n,m}?	X{n,m}+	X, najmanje n puta, ali ne više od m puta

Imajte u vidu da izraz X često treba da zatvorite u zagrade da bi radio onako kako hoćete. Na primer:

```
abc+
```

možda izgleda kao da se podudara s nizom abc jednom ili više puta, i čini se da ćete dobiti tri podudaranja kada ga primenite na ulazni znakovni niz abcabcabc. Međutim, izraz *zapravo* kaže: „Pronađi ab i iza njega jedno ili više slova c." Da biste pronašli ceo znakovni niz abc jednom ili više puta, morate napisati:

```
(abc)+
```

Regularni izrazi će vas lako prevariti; u poređenju s Javom, to je sasvim drugačiji jezik.

CharSequence

Interfejs **CharSequence** utvrđuje opštu definiciju niza znakova apstrahovanu iz klasa **CharBuffer, String, StringBuffer** ili **StringBuilder**:

```
interface CharSequence {
  charAt(int i);
  length();
  subSequence(int pocetak, int kraj);
  toString();
}
```

Ovaj interfejs realizuju prethodno spomenute klase. Mnoge operacije s regularnim izrazima primaju argumente tipa **CharSequence**.

Klase Pattern i Matcher

Po pravilu, nećete se služiti relativno ograničenim uslužnim metodama klase **String**, nego ćete prevoditi objekte regularnih izraza. Za to treba da uvezete **java.util.regex** i zatim prevedete regularan izraz statičnom metodom **Pattern.compile( )**. Tako dobijate objekat tipa **Pattern**, napravljen na osnovu njegovog argumenta tipa **String. Pattern** upotrebljavate pozivanjem metode **matcher( )** a njoj prosleđujete znakovni niz koji treba pretražiti. Metoda **matcher( )** pravi objekat tipa **Matcher** kojem je na raspolaganju skup operacija (možete ih videti u JDK dokumentaciji klase **java.util.regex.Matcher**). Na primer, metoda **replaceAll( )** svojim argumentom zamenjuje sve podudarne delove znakovnog niza.

Kao prvi primer, sledeća klasa se može upotrebiti za testiranje regularnih izraza na ulazni znakovni niz. Prvi argument na komandnoj liniji je ulazni znakovni niz s kojim se regularni izraz poredi. Iza njega slede jedan ili više regularnih izraza koje treba primeniti na ulaz. U Unixu/Linuxu, regularni izrazi na komandnoj liniji moraju biti zatvoreni u navodnike. Ovaj program možete upotrebljavati za testiranje tokom konstruisanja regularnih izraza, kako biste videli da li proizvode željeno ponašanje u pogledu podudaranja.

```
//: znakovninizovi/TestiranjeRegularnihIzraza.java
// Služi za isprobavanje regularnih izraza.
// {Args: abcabcabcdefabc "abc+" "(abc)+" "(abc){2,}" }
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestiranjeRegularnihIzraza {
  public static void main(String[] args) {
    if(args.length < 2) {
      print("Upotreba:\njava TestiranjeRegularnihIzraza " +
        "nizZnakova regularanIzraz+");
      System.exit(0);
    }
    print("Ulaz: \"" + args[0] + "\"");
    for(String arg : args) {
      print("Regularan izraz: \"" + arg + "\"");
```

```
      Pattern p = Pattern.compile(arg);
      Matcher m = p.matcher(args[0]);
      while(m.find()) {
        print("Podudara se \"" + m.group() + "\" na mestima " +
          m.start() + "-" + (m.end() - 1));
      }
    }
  }
} /* Ispis:
Ulaz: "abcabcabcdefabc"
Regularan izraz: "abcabcabcdefabc"
Podudara se "abcabcabcdefabc" na mestima 0-14
Regularan izraz: "abc+"
Podudara se "abc" na mestima 0-2
Podudara se "abc" na mestima 3-5
Podudara se "abc" na mestima 6-8
Podudara se "abc" na mestima 12-14
Regularan izraz: "(abc)+"
Podudara se "abcabcabc" na mestima 0-8
Podudara se "abc" na mestima 12-14
Regularan izraz: "(abc){2,}"
Podudara se "abcabcabc" na mestima 0-8
*///:~
```

Objekat tipa **Pattern** predstavlja prevedenu verziju regularnog izraza. Kao što vidite u prethodnom primeru, za pravljenje objekta tipa **Matcher** od prevedenog objekta tipa **Pattern** možete upotrebiti metodu **matcher( )** i ulazni znakovni niz. I klasa **Pattern** ima statičnu metodu:

```
static boolean matches(String regiz, CharSequence ulaz)
```

koja ispituje da li **regiz** odgovara celom ulaznom nizu znakova (objektu tipa **CharSequence**), i metodu **split( )** koja proizvodi niz objekata tipa **String** koji se podudaraju s regularnim izrazom **regiz**.

Objekat tipa **Matcher** generišete pozivom metode **Pattern.matcher( )** uz ulazni znakovni niz kao argument. Zatim se objekat tipa **Matcher** upotrebljava za pristupanje rezultatima, jer se za utvrđivanje uspešnosti ili neupešnosti različitih vrsta podudaranja upotrebljavaju njegove metode:

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

Metoda **matches( )** je uspešna ukoliko se uzorak podudara s celokupnim ulaznim znakovnim nizom, dok je metoda **lookingAt( )** uspešna ako se ulazni znakovni niz podudara sa uzorkom počev od njegovog početka.

Vežba 10: (2) Za rečenicu „Java now has regular expressions" odredite da li će sledeći izrazi pronaći neko podudaranje:

```
^Java
\Breg.*
n.w\s+h(a|i)s
s?
s*
s+
s{4}
s{1}.
s{0,3}
```

Vežba 11: (2) Primenite regularan izraz

```
(?i)((^[aeiou])|(\s+[aeiou]))\w+?[aeiou]\b
```

na

```
"Arline ate eight apples and one orange while Anita hadn't any"
```

Metoda **find()**

Metoda **Matcher.find()** može se upotrebiti za pronalaženje podudaranja više uzoraka u znakovnom nizu na koji je primenjena. Na primer:

```
//: znakovninizovi/Pronalazenje.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Pronalazenje {
  public static void main(String[] args) {
    Matcher m = Pattern.compile("\\w+")
      .matcher("Evening is full of the linnet's wings");
    while(m.find())
      printnb(m.group() + " ");
    print();
    int i = 0;
    while(m.find(i)) {
      printnb(m.group() + " ");
      i++;
    }
  }
} /* Ispis:
Evening is full of the linnet s wings
Evening vening ening ning ing ng g is is s full full ull ll l of of f the the
he e linnet linnet innet nnet net et t s s wings wings ings ngs gs s
*///:~
```

Uzorak \\w+ cepa ulaz na reči. Metoda **find( )** radi kao iterator, pošto se kroz ulazni znakovni niz kreće unapred. Međutim, drugoj verziji metode **find( )** možete dati celobrojni argument koji joj kazuje redni broj znaka za početak pretraživanja – ta verzija zadaje vrednost argumenta kao mesto pretraživanja, kao što se vidi iz rezultata.

Grupe

Grupe su regularni izrazi navedeni u zagradama koji se naknadno mogu pozivati preko broja grupe. Grupa 0 je ceo izraz, grupa 1 je prva grupa u zagradi itd. Stoga u

```
A(B(C))D
```

postoje tri grupe: grupa 0 je **ABCD**, grupa 1 je **BC**, a grupa 2 je **C**.

Objekat tipa **Matcher** ima metode koje daju podatke o grupama:

public int groupCount( ) vraća broj grupa u uzorku objekta. Grupa 0 se ovde ne broji.

public String group( ) vraća grupu 0 (ceo podudarni deo) iz prethodne operacije pronalaženja podudarnosti (metode **find( )**, na primer).

public String group(int i) vraća grupu datog rednog broja iz prethodne operacije pronalaženja podudarnosti. Ako je pronalaženje podudarnosti uspešno obavljeno, ali se data grupa nije podudarila ni s jednim delom ulaznog znakovnog niza, vraća se **null**.

public int start(int group) vraća indeks početka grupe pronađene u prethodnoj operaciji pronalaženja podudarnosti.

public int end(int group) vraća indeks poslednjeg znaka, plus jedan, grupe pronađene u prethodnoj operaciji pronalaženja podudarnosti.

Evo primera:

```
//: znakovninizovi/Grupe.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Grupe {
  static public final String PESMA =
    "Twas brillig, and the slithy toves\n" +
    "Did gyre and gimble in the wabe.\n" +
    "All mimsy were the borogoves,\n" +
    "And the mome raths outgrabe.\n\n" +
    "Beware the Jabberwock, my son,\n" +
    "The jaws that bite, the claws that catch.\n" +
    "Beware the Jubjub bird, and shun\n" +
    "The frumious Bandersnatch.";
  public static void main(String[] args) {
    Matcher m =
      Pattern.compile("(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$")
        .matcher(PESMA);
```

```
    while(m.find()) {
      for(int j = 0; j <= m.groupCount(); j++)
        printnb("[" + m.group(j) + "]");
      print();
    }
  }
} /* Ispis:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun] [The frumious
Bandersnatch.][The][frumious Bandersnatch.][frumious][Bandersnatch.]
*///:~
```

Ovo je početak pesme „Jabberwocky" Luisa Kerola, iz knjige *Alisa iza ogledala*. Uzorak u regularnom izrazu ima više grupa u zagradama koje se sastoje od proizvoljnog broja znakova koji nisu beline (\S+) praćenog proizvoljnim brojem znakova koji jesu beline (\s+). Cilj je uhvatiti tri poslednje reči u svakom redu; kraj reda označava **$**. Međutim, uobičajeno ponašanje je da se **$** poredi s krajem celog ulaznog niza, pa regularnom izrazu morate eksplicitno reći da obrati pažnju na znakove za prelazak u novi red u ulaznom nizu. To se postiže indikatorom uzorka **(?m)** na početku niza (ubrzo ćemo razmotriti indikatore uzoraka).

Vežba 12: (5) Prepravite program **Grupe.java** tako da broji sve različite reči koje ne počinju velikim slovom.

Metode **start()** i **end()**

Nakon uspešno obavljene operacije traženja podudaranja, metoda **start( )** vraća indeks početka prethodno podudarnog dela, a **end( )** vraća indeks poslednjeg podudarnog znaka plus jedan. Pozivanjem metoda **start( )** ili **end( )** nakon neuspešno obavljene operacije traženja podudaranja (ili pre nego što je takva operacija započeta) prouzrokuje se izuzetak **IllegalStateException**. Naredni program prikazuje i rad metoda **matches( )** i **lookingAt( )**:

```
//: znakovninizovi/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class StartEnd {
  public static String ulaz =
    "As long as there is injustice, whenever a\n" +
    "Targathian baby cries out, wherever a distress\n" +
    "signal sounds among the stars ... We'll be there.\n" +
    "This fine ship, and this fine crew ...\n" +
    "Never give up! Never surrender!";
```

[1] Citat iz govora komandanta Tagarta u seriji *Galaksija*.

```
  private static class Ispisi {
    private boolean regizOdstampan = false;
    private String regiz;
    Ispisi(String regiz) { this.regiz = regiz; }
    void Ispisi(String poruka) {
      if(!regizOdstampan) {
        print(regiz);
        regizOdstampan = true;
      }
      print(poruka);
    }
  }
  static void ispitaj(String s, String regiz) {
    Ispisi d = new Ispisi(regiz);
    Pattern p = Pattern.compile(regiz);
    Matcher m = p.matcher(s);
    while(m.find())
      d.ispisi("find() '" + m.group() +
        "' početak = "+ m.start() + " kraj = " + m.end());
    if(m.lookingAt()) // reset() nije potrebna
      d.ispisi("lookingAt() početak = "
        + m.start() + " kraj = " + m.end());
    if(m.matches()) // reset() nije potrebna
      d.ispisi("matches() početak = "
        + m.start() + " kraj = " + m.end());
  }
  public static void main(String[] args) {
    for(String in : ulaz.split("\n")) {
      print("ulaz : " + in);
      for(String regiz : new String[]{"\\w*ere\\w*",
        "\\w*ever", "T\\w+", "Never.*?!"})
        ispitaj(in, regiz);
    }
  }
} /* Ispis:
ulaz : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' početak = 11 kraj = 16
\w*ever
find() 'whenever' početak = 31 kraj = 39
ulaz : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' početak = 27 kraj = 35
\w*ever
find() 'wherever' početak = 27 kraj = 35
T\w+
find() 'Targathian' početak = 0 kraj = 10
lookingAt() početak = 0 kraj = 10
ulaz : signal sounds among the stars ... We'll be there.
\w*ere\w*
```

```
find() 'there' početak = 43 kraj = 48
ulaz : This fine ship, and this fine crew ...
T\w+
find() 'This' početak = 0 kraj = 4
lookingAt() početak = 0 kraj = 4
ulaz : Never give up! Never surrender!
\w*ever
find() 'Never' početak = 0 kraj = 5
find() 'Never' početak = 15 kraj = 20
lookingAt() početak = 0 kraj = 5
Never.*?!
find() 'Never give up!' početak = 0 kraj = 14
find() 'Never surrender!' početak = 15 kraj = 31
lookingAt() početak = 0 kraj = 14
matches() početak = 0 kraj = 31
*///:~
```

Vodite računa o tome da će metoda **find( )** pronaći regularan izraz bilo gde u ulazu, ali **lookingAt( )** i **matches( )** uspevaju samo ako se regularan izraz podudara od samog početka ulaza. Dok metoda **matches( )** uspeva samo ukoliko se *ceo* ulaz podudara s regularnim izrazom, **lookingAt( )**[4] uspeva ako se podudara samo prvi deo ulaza.

Vežba 13: (2) Prepravite program **StartEnd.java** tako da upotrebljava **Grupe.PESMA** kao ulaz, ali da ipak proizvodi pozitivne rezultate za metode **find( ), lookingAt( )** i **matches( )**.

Indikatori klase **Pattern**

Alternativna metoda **compile( )** prihvata indikatore koji utiču na ponašanje prilikom traženja podudarnosti:

```
Pattern Pattern.compile(String regiz, int indikator)
```

gde je **indikator** jedna od sledećih konstanti klase **Pattern**:

Indikator prevođenja	Uticaj
Pattern.CANON_EQ	Smatra se da se dva znaka podudaraju ako i samo ako se podudaraju njihove pune kanonske dekompozicije. Na primer, kada je ovaj indikator zadat, izraz \u003F se podudara sa znakovnim nizom "?". Kanonska ekvivalencija se podrazumevano ne uzima u obzir.
Pattern.CASE_INSENSITIVE (?i)	Podrazumevano ponašanje poređenja bez uzimanja u obzir veličine slova (engl. *case-insensitive matching*) jeste da se porede samo znakovi iz skupa US-ASCII. Ovaj indikator služi da se ispita podudaranje uzorka bez uzimanja u obzir da li su slova velika ili mala. Ako uz ovaj indikator zadate i indikator **UNICODE_CASE**, možete porediti i znakove iz skupa Unicode.

[4] Nemam pojma odakle im ovakvo ime metode, niti na šta bi ono trebalo da upućuje, ali umiruju činjenicu da osoba koja izmišlja neintuitivna imena metoda još uvek radi za Sun, i da je njihova očigledna politika nerevidiranja projekata koda još uvek na snazi. Izvinite zbog sarkazma, ali tako nešto počinje da zamara nakon nekoliko godina.

Indikator prevođenja	Uticaj
Pattern.COMMENTS **(?x)**	U ovom režimu zanemaruju se bjeline. Do kraja reda zanemaruju se ugrađeni komentari koji počinju znakom #. Ugrađivanjem indikatora **(?x)** u izraz može se uključiti režim Unix lines.
Pattern.DOTALL **(?s)**	U režimu dotall, izraz **.** se podudara s bilo kojim znakom, čak i s graničnikom reda. Izraz **.** se podrazumevano ne podudara s graničnicima redova.
Pattern.MULTILINE **(?m)**	U režimu multiline, izrazi **^** i **$** podudaraju se s početkom odnosno krajem reda. **^** se podudara s početkom ulaznog znakovnog niza, a **$** s krajem ulaznog znakovnog niza. Ovi izrazi se podrazumevano podudaraju samo s početkom i krajem celog ulaznog znakovnog niza.
Pattern.UNICODE_CASE **(?u)**	Poređenje bez uzimanja u obzir veličine slova, kada je uključeno indikatorom **CASE_INSENSITIVE**, sprovodi se u skladu sa standardom Unicode. Podrazumevano ponašanje poređenja bez uzimanja u obzir veličine slova jeste da se porede samo znakovi iz skupa US-ASCII.
Pattern.UNIX_LINES **(?d)**	U ovom režimu se u ponašanju izraza **.**, **^** i **$** kao graničnik reda priznaje samo **\n**

Od ovih indikatora naročito su korisni **Pattern.CASE_INSENSITIVE, Pattern.MULTILINE** i **Pattern.COMMENTS** (koji povećava jasnoću i/ili pruža **dokumentaciju**). Imajte u vidu da se ponašanje većine indikatora može postići postavljanjem znakova u zagradama, napisanih ispod indikatora u tabeli, u regularan izraz ispred mesta na kojem hoćete da aktivirate taj režim.

Uticaj navedenih i drugih indikatora možete da kombinujete operacijom OR (|):

```
//: znakovninizovi/OIndikatorima.java
import java.util.regex.*;

public class OIndikatorima {
  public static void main(String[] args) {
    Pattern p =  Pattern.compile("^java",
      Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
    Matcher m = p.matcher(
      "java ima regiz\njava ima regiz\n" +
      "JAVA ima prilično dobre regularne izraze\n" +
      "U jeziku Java postoje regularni izrazi");
    while(m.find())
      System.out.println(m.group());
  }
} /* Ispis:
java
Java
JAVA
*///:~
```

Napravljen je uzorak koji se podudara s redovima koji počinju rečima java, Java, JAVA itd. Podudaranje se ispituje za svaki red višerednog skupa (podudaranje delova koji počinju na početku niza znakova i slede iza graničnika svakog reda u nizu znakova). Obratite pažnju na to da metoda **group( )** daje samo deo koji se podudario.

Metoda **split()**

Metoda **split( )** cepa ulazni znakovni niz na niz **String** objekata, razgraničenih datim regularnim izrazom.

```
String[] split(CharSequence ulaz)
String[] split(CharSequence ulaz, int najvišePodnizova)
```

Ovo je podesno za cepanje ulaznog teksta na delove koji imaju zajednički graničnik:

```
//: znakovninizovi/PrimerZaSplit.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PrimerZaSplit {
  public static void main(String[] args) {
    String ulaz =
      "This!!unusual use!!of exclamation!!points";
    print(Arrays.toString(
      Pattern.compile("!!").split(ulaz)));
    // Uradi samo prva tri:
    print(Arrays.toString(
      Pattern.compile("!!").split(ulaz, 3)));
  }
} /* Ispis:
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*///:~
```

Drugi oblik metode **split( )** ograničava broj podnizova u rezultatu.

Vežba 14: (1) Ponovno napišite program **PrimerZaSplit** uz korišćenje metode **String.split( )**.

Operacije zamene

Regularni izrazi su posebno podesni za zamenu teksta. Na raspolaganju su ove metode:

replaceFirst(String zamena) zamenjuje prvi podudarni deo ulaznog znakovnog niza **zamenom**.

replaceAll(String zamena) zamenjuje svaki podudarni deo ulaznog znakovnog niza **zamenom**.

appendReplacement(StringBuffer sbaf, String zamena) u baferu **sbaf** obavlja zamene korak po korak, umesto da zameni samo prvi podudarni deo ili sve njih, kao što rade **replaceFirst( )** odnosno **replaceAll( )**. Ova metoda je *veoma* važna, pošto omogućava pozivanje metoda i obavljanje druge obrade potrebne za proizvodnju znakovnog niza **zamena** (**replaceFirst( )** i **replaceAll( )** mogu da umetnu samo nepromenljive znakovne nizove). Pomoću ove metode možete programski da raščlanite grupe i napravite moćne zamene.

appendTail(StringBuffer sbaf, String zamena) poziva se nakon jednog ili više poziva metode **appendReplacement( )** da bi se kopirao ostatak ulaznog znakovnog niza.

Evo primera u kojem je pokazana upotreba svih operacija zamene. Blok teksta u komentaru na početku programa biva izvađen i obrađen regularnim izrazima da bi se upotrebio kao ulaz u ostatku primera:

```
//: znakovninizovi/Zamene.java
import java.util.regex.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

/*! Here's a block of text to use as input to the regular
    expression matcher. Note that we'll first extract
    the block of text by looking for the special delimiters,
    then process the extracted block. !*/

public class Zamene {
  public static void main(String[] args) throws Exception {
    String s = TextFile.read("Zamene.java");
    // Pronađi gornji blok teksta naročito pretvoren u komentar:
    Matcher ulazUM =
      Pattern.compile("/\\*!(.*)!\\*/", Pattern.DOTALL)
        .matcher(s);
    if(ulazUM.find())
      s = ulazUM.group(1); // Ono što hvataju zagrade
    // Dva i više razmaka zameni jednim:
    s = s.replaceAll(" {2,}", " ");
    // Jedan ili više razmaka na početku svakog reda
    // zameni s nula razmaka. Mora se uključiti režim MULTILINE:
    s = s.replaceAll("(?m)^ +", "");
    print(s);
    s = s.replaceFirst("[aeiou]", "(VOWEL1)");
    StringBuffer sbaf = new StringBuffer();
    Pattern p = Pattern.compile("[aeiou]");
    Matcher m = p.matcher(s);
    // Obradi podatke metode find() dok
    // obavljaš zamene:
    while(m.find())
      m.appendReplacement(sbaf, m.group().toUpperCase());
    // Umetni ostatak teksta:
    m.appendTail(sbaf);
    print(sbaf);
  }
```

```
} /* Ispis:
Here's a block of text to use as input to the regular
expression matcher. Note that we'll first extract
the block of text by looking for the special delimiters,
then process the extracted block.
H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO
thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE'll
fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr
thE spEcIAl dElImItErs, thEn prOcEss thE
ExtrActEd blOck.
*///:~
```

Datoteka se otvara i učitava pomoću klase **TextFile** iz biblioteke **net.mindview.util** (njen kôd će biti prikazan u poglavlju *Javin ulazno-izlazni sistem*). Statična metoda **read**( ) čita celu datoteku i vraća je kao **String**. **ulazUM** se pravi tako da se podudara s celim tekstom (obratite pažnju na zagrade za grupisanje) između **/*!** i **!*/**. Zatim se dva i više razmaka svode na jedan razmak i uklanjaju se svi razmaci s početka svakog reda (mora se uključiti režim multiline da bi se to uradilo u svim redovima, a ne samo na početku ulaza). Te dve zamene se obavljaju ekvivalentom (u ovom slučaju, podesnijim) metode **replaceAll**( ) koji je deo klase **String**. Pošto se svaka zamena u programu upotrebljava samo jedanput, to ne košta ništa više nego da se obavljalo prethodno prevođenje u objekat tipa **Pattern**.

replaceFirst( ) obavlja samo prvu zamenu koju pronađe. Sem toga, zamenski znakovni nizovi u metodama **replaceFirst**( ) i **replaceAll**( ) mogu biti samo literali, pa ako svaku zamenu hoćete još da obradite, od tih metoda nećete imati koristi. U tom slučaju upotrebite metodu **appendReplacement**( ) koja omogućava da tokom obavljanja postupka zamene upišete proizvoljnu količinu koda. U prethodnom primeru, bira se i obrađuje jedna grupa **group**( ) – u ovom slučaju, regularni izraz pronalazi samoglasnik (engl. *vowel*) koji biva napisan velikim slovom – dok se pravi rezultujući **sbaf**. Obično se korak po korak obave sve zamene i zatim pozove **appendTail**( ), ali ako hoćete da simulirate **replaceFirst**( ) (ili „replace n"), zamenu obavite samo jednom i zatim pozovete **appendTail**( ) da ostatak spremi u **sbaf**.

Metoda **appendReplacement**( ) omogućava i da uhvaćenu grupu naznačimo direktno u zamenskom znakovnom nizu izrazom „$g", gde g označava broj grupe. Međutim, to je podesno za jednostavniju obradu i u prethodnom programu ne bi dalo željeni rezultat.

Metoda **reset()**

Postojeći objekat tipa **Matcher** metodama **reset**( ) možete primeniti na nov niz znakova:

```
//: znakovninizovi/Resetovanje.java
import java.util.regex.*;

public class Resetovanje {
  public static void main(String[] args) throws Exception {
    Matcher m = Pattern.compile("[frb][aiu][gx]")
      .matcher("fix the rug with bags");
```

```
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    m.reset("fix the rig with rags");
    while(m.find())
      System.out.print(m.group() + " ");
  }
} /* Ispis:
fix rug bag
fix rig rag
*///:~
```

reset( ) bez argumenata primenjuje objekat tipa **Matcher** na početak tekućeg niza.

Regularni izrazi i Javin ulazno-izlazni sistem

U većini dosadašnjih primera regularne izraze smo primenjivali na statične znakovne nizove. U narednom primeru prikazan je jedan način primene regularnih izraza za pronalaženje delova datoteke koji se s njim podudaraju. Napravljen na osnovu Unixove alatke *grep*, **JGrep.java** prima dva argumenta: ime datoteke i regularan izraz koji treba pronaći. Iz rezultata se vidi broj reda gde je pronađena podudarnost i položaj(i) podudarajućih delova reda.

```
//: znakovninizovi/JGrep.java
// Veoma jednostavna verzija programa "grep".
// {Args: JGrep.java "\\b[Ssct]\\w+"}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println("Upotreba: java JGrep regiz datoteke");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    // Iteracija kroz redove ulazne datoteke:
    int indeks = 0;
    Matcher m = p.matcher("");
    for(String red : new TextFile(args[0])) {
      m.reset(red);
      while(m.find())
        System.out.println(indeks++ + ": " +
          m.group() + ": " + m.start());
    }
  }
} /* Ispis: (primer)
0: strings: 4
1: simple: 10
2: the: 28
```

```
3: Ssct: 26
4: class: 7
5: static: 9
6: String: 26
7: throws: 41
8: System: 6
9: System: 6
10: compile: 24
11: through: 15
12: the: 23
13: the: 36
14: String: 8
15: System: 8
16: start: 31
*///:~
```

Datoteka se otvara kao objekat tipa **net.mindview.util.TextFile** (što će biti objašnjeno u poglavlju *Javin ulazno-izlazni sistem*), koji redove datoteke učitava u **ArrayList**. To znači da iteraciju kroz redove **TextFile** objekta možemo obaviti foreach sintaksom.

Iako smo mogli da pravimo nov objekat tipa **Matcher** unutar petlje **for**, nešto je bolje napraviti prazan objekat tipa **Matcher** izvan petlje i dodeljivati mu red po red ulaza metodom **reset( )**. Rezultat toga se pretražuje metodom **find( )**.

Argumenti ispitivanja (treći red programa) znače da se datoteka **JGrep.java** otvara za učitavanje kao ulaz i da se u njoj traže reči koje počinju sa [**Ssct**].

O regularnim izrazima možete naučiti mnogo više iz knjige *Mastering Regular Expressions, 2nd Edition*, koju je napisao Jeffrey E. F. Friedl (O'Reilly, 2002). I na Internetu ima mnogo uvoda u regularne izraze, a i u dokumentaciji jezika Perl i Python često se mogu naći korisni podaci.

Vežba 15: (5) Prepravite **JGrep.java** tako da prima indikatore kao argumente (npr. **Pattern.CASE_INSENSITIVE**, **Pattern.MULTILINE**).

Vežba 16: (5) Prepravite **JGrep.java** tako da kao argument prima ime datoteke ili direktorijuma (ako se zada direktorijum, pretraživanje treba da obuhvati sve datoteke u njemu). Uputstvo: generišite listu imena datoteka naredbom:

```
File[ ] datoteke = new File(".").listFiles();
```

Vežba 17: (8) Napišite program koji učitava datoteku Javinog izvornog koda (čije ime zadajete na komandnoj liniji) i prikazuje sve njene komentare.

Vežba 18: (8) Napišite program koji učitava datoteku Javinog izvornog koda (čije ime zadajete na komandnoj liniji) i prikazuje sve doslovno (literalno) navedene znakovne nizove u kodu.

Vežba 19: (8) Na osnovu prethodne dve vežbe, napišite program koji pretražuje Javin izvorni kôd i ispisuje imena svih klasa koje se spominju u tom programu.

Leksičko analiziranje ulaza

Dosad je učitavanje podataka iz datoteke čitljive ljudima ili sa standardnog ulaza bilo relativno teško. Uobičajeno rešenje bilo je da se učita red teksta, raščlani na lekseme (engl. *tokens*) i potom jezički analizira (engl. *parse*) raznim metodama za tipove **Integer, Double** itd.:

```
//: znakovninizovi/ProstoCitanje.java
import java.io.*;

public class ProstoCitanje {
  public static BufferedReader ulaz = new BufferedReader(
    new StringReader("Hajduk Veljko\n22 1.61803"));
  public static void main(String[] args) {
    try {
      System.out.println("Kako se zoveš?");
      String ime = ulaz.readLine();
      System.out.println(ime);
      System.out.println("Koliko godina imaš?
                         Koji je tvoj omiljeni broj tipa double?");
      System.out.println("(ulaz: <starost> <double>)");
      String brojevi = ulaz.readLine();
      System.out.println(brojevi);
      String[] numNiz = brojevi.split(" ");
      int starost = Integer.parseInt(numNiz[0]);
      double omiljeni = Double.parseDouble(numNiz[1]);
      System.out.format("Zdravo %s.\n", ime);
      System.out.format("Za 5 godina imaćeš %d.\n",
        starost + 5);
      System.out.format("Moj omiljeni broj tipa double je %f.",
        omiljeni / 2);
    } catch(IOException e) {
      System.err.println("Izuzetak ulazno-izlaznog sistema");
    }
  }
} /* Ispis:
Kako se zoveš?
Hajduk Veljko
Koliko godina imaš? Koji je tvoj omiljeni broj tipa double?
(ulaz: <starost> <double>)
22 1.61803
Zdravo Hajduk Veljko.
Za 5 godina imaćeš 27.
Moj omiljeni broj tipa double je 0.809015.
*///:~
```

U polju **ulaz** koriste se klase iz paketa **java.io** koji neće biti zvanično predstavljen do poglavlja *Javin ulazno-izlazni sistem*. Objekat tipa **StringReader** pretvara znakovni niz u tok podataka (engl. *stream*) koji se može čitati, i taj objekat se upotrebljava za pravljenje

objekta tipa **BufferedReader**, pošto **BufferedReader** ima metodu **readLine( )**. Zato se objekat **ulaz** može čitati red po red, kao da je standardni ulaz s konzole.

readLine( ) pribavlja **String** za svaki red ulaza. Jednostavno je ako svaki red podataka hoćete da pretvorite u jedan ulaz, ali ukoliko su dve ulazne vrednosti **u** istom redu, stvari se komplikuju – da bismo svaki ulaz mogli da analiziramo zasebno, red se mora podeliti na dva dela. Ovde se podela obavlja kada se pravi **numNiz**, ali pošto je metoda **split( )** uvedena tek u J2SE1.4, pre se to moralo raditi drugačije.

Klasa **Scanner** se koristi od Jave SE5. Ona obavlja glavni deo posla oko leksičkog analiziranja ulaza:

```
//: znakovninizovi/BoljeCitanje.java
import java.util.*;

public class BoljeCitanje {
  public static void main(String[] args) {
    Scanner stdulaz = new Scanner(SimpleRead.ulaz);
    System.out.println("Kako se zoveš?");
    String ime = stdulaz.nextLine();
    System.out.println(ime);
    System.out.println("Koliko godina imaš?
                       Koji je tvoj omiljeni broj tipa double?");
    System.out.println("(ulaz: <starost> <double>)");
    int starost = stdulaz.nextInt();
    double omiljeni = stdulaz.nextDouble();
    System.out.println(starost);
    System.out.println(omiljeni);
    System.out.format("Zdravo %s.\n", ime);
    System.out.format("Za 5 godina imaćeš %d.\n",
      starost + 5);
    System.out.format("Moj omiljeni broj tipa double je %f.",
      omiljeni / 2);
  }
} /* Ispis:
Kako se zoveš?
Hajduk Veljko
Koliko godina imaš? Koji je tvoj omiljeni broj tipa double?
(ulaz: <starost> <double>)
22
1.61803
Zdravo Hajduk Veljko.
Za 5 godina imaćeš 27.
Moj omiljeni broj tipa double je 0.809015.
*///:-
```

Konstruktor klase **Scanner** prima gotovo sve vrste ulaznih objekata – među njima su objekat tipa **File** (koji će takođe biti objašnjen u poglavlju *Javin ulazno-izlazni sistem*), objekat tipa **InputStream**, objekat tipa **String** ili u ovom slučaju objekat tipa **Readable**,

što je interfejs koji je Java SE5 uvela za opisivanje „nečega što ima metodu **read( )**". U tu kategoriju spada **BufferedReader** iz prethodnog primera.

U klasi **Scanner** se metode za učitavanje ulaza, podelu na lekseme i jezičku analizu kriju u raznim vrstama metoda „next". Obična metoda **next( )** vraća sledeću leksemu znakovnog niza, a postoje i „next" metode za sve proste tipove (sem **char**), kao i za **BigDecimal** i **BigInteger**. Sve „next" metode *blokiraju*, što znači da vraćaju rezultat tek kada celokupna leksema postane dostupna na ulazu. Postoje i odgovarajuće metode „hasNext", koje vraćaju **true** ukoliko je sledeća ulazna leksema ispravnog tipa.

Zanimljiva razlika između dva prethodna primera: u programu **BoljeCitanje.java** ne postoji blok **try** za izuzetke tipa **IOException**. Jedna od pretpostavki klase **Scanner** jeste da **IOException** signalizira kraj ulaza, pa te izuzetke **Scanner** guta. Međutim, najnoviji izuzetak je dostupan preko metode **ioException( )**, pa možete da ga ispitate ako treba.

Vežba 18: (2) Napravite klasu koja sadrži polja tipova **int**, **long**, **float**, **double** i **String**. Napravite konstruktor za tu klasu; on treba da prima jedan argument tipa **String**, da analizira taj znakovni niz i da ga raščlani na spomenuta polja. Dodajte metodu **toString( )** i pokažite da klasa ispravno radi.

Graničnici klase **Scanner**

Klasa **Scanner** podrazumevano cepa ulazne lekseme na mestima razmaka, ali možete da zadate i sopstveni uzorak graničnika u obliku regularnog izraza:

```
//: znakovninizovi/GranicnikKlaseScanner.java
import java.util.*;

public class GranicnikKlaseScanner {
  public static void main(String[] args) {
    Scanner skener = new Scanner("12, 42, 78, 99, 42");
    skener.useDelimiter("\\s*,\\s*");
    while(skener.hasNextInt())
      System.out.println(skener.nextInt());
  }
} /* Ispis:
12
42
78
99
42
*///:~
```

U ovom primeru, kao graničnici prilikom čitanja datog znakovnog niza upotrebljeni su zarezi (oko kojih može biti proizvoljna količina belina). Ista tehnika se može upotrebiti za učitavanje datoteka u kojima su podaci razgraničeni zarezima. Pored metode **useDelimiter( )** za zadavanje uzorka graničnika, postoji i **delimiter( )** – on vraća tekući objekat tipa **Pattern** koji se upotrebljava kao graničnik.

Leksička analiza pomoću regularnih izraza

Pored traženja unapred definisanih prostih tipova, možete tražiti i sopstvene (korisnički definisane) uzorke, što je podesno za analizu složenijih podataka. U narednom primeru, tražimo preteće podatke u zapisniku kakav vodi, recimo, zaštitna barijera (engl. *firewall*):

```
//: znakovninizovi/AnalizatorPretnji.java
import java.util.regex.*;
import java.util.*;

public class AnalizatorPretnji {
  static String preteciPodaci =
    "58.27.82.161@02/10/2005\n" +
    "204.45.234.40@02/11/2005\n" +
    "58.27.82.161@02/11/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "[Sledeći odeljak zapisnika s drugačijim formatom podataka]";
  public static void main(String[] args) {
    Scanner skener = new Scanner(preteciPodaci);
    String uzorak = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@" +
      "(\\d{2}/\\d{2}/\\d{4})";
    while(skener.hasNext(uzorak)) {
      skener.next(uzorak);
      MatchResult pronasao = skener.match();
      String ip = pronasao.group(1);
      String datum = pronasao.group(2);
      System.out.format("Pretnja dana %s od %s\n", datum,ip);
    }
  }
} /* Ispis:
Pretnja dana 02/10/2005 od 58.27.82.161
Pretnja dana 02/11/2005 od 204.45.234.40
Pretnja dana 02/11/2005 od 58.27.82.161
Pretnja dana 02/12/2005 od 58.27.82.161
Pretnja dana 02/12/2005 od 58.27.82.161
*///:~
```

Kada metodu **next( )** upotrebljavate s konkretnim uzorkom, ona ga traži u sledećoj ulaznoj leksemi. Rezultat pravi metoda **match( )**, i kao što vidite u gornjem primeru, ona radi baš kao regularni izrazi koje ste već upoznali.

Kada leksičku analizu obavljate pomoću regularnih izraza, vodite računa o tome da se uzorak traži samo u sledećoj ulaznoj leksemi, pa ako uzorak sadrži i graničnik, neće nikada biti pronađen.

Klasa **StringTokenizer**

Pre regularnih izraza (koji su uvedeni u J2SE1.4) ili klase **Scanner** (koja se koristi od Jave SE5), znakovni niz se cepao na lekseme pomoću klase **StringTokenizer**. Ali sada se to mnogo lakše i sažetije obavlja pomoću regularnog izraza ili klase **Scanner**. Pogledajte jednostavno poređenje klase **StringTokenizer** i druge dve tehnike:

```
//: znakovninizovi/UmestoStringTokenizera.java
import java.util.*;

public class UmestoStringTokenizera {
  public static void main(String[] args) {
    String input = "But I'm not dead yet! I feel happy!";
    StringTokenizer stoke = new StringTokenizer(ulaz);
    while(stoke.hasMoreElements())
      System.out.print(stoke.nextToken() + " ");
    System.out.println();
    System.out.println(Arrays.toString(ulaz.split(" ")));
    Scanner skener = new Scanner(ulaz);
    while(skener.hasNext())
      System.out.print(skener.next() + " ");
  }
} /* Ispis:
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*///:~
```

Pomoću regularnih izraza ili objekata tipa **Scanner**, znakovni niz možete podeliti na delove i korišćenjem složenijih uzoraka – što bi bilo teško pomoću klase **StringTokenizer**. Izgleda da možemo bezbedno reći kako je **StringTokenizer** zastarela.

Sažetak

Ranije je Javina podrška za obradu znakovnih nizova bila rudimentarna, ali u novijim izdanjima jezika postoji mnogo sofisticiranija podrška usvojena iz drugih jezika. Sada je podrška za znakovne nizove prilično potpuna, iako ponekad morate paziti na efikasnost, recimo prilikom upotrebe klase **StringBuilder**.

Rešenja odabranih vežbi data su u elektronskom dokumentu *Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.BruceEckel.com*.

Podaci o tipu

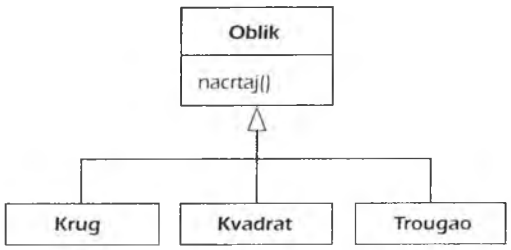
Prepoznavanje tipa u vreme izvršavanja (engl. run-time type identification, RTTI*) omogućuje ustanovljavanje i upotrebu tipa objekta dok se program izvršava.*

RTTI VAS OSLOBAĐA OGRANIČENJA DA TIPOVE MORATE ZNATI VEĆ U VREME PREVOĐENJA, te stoga omogućuje pisanje veoma moćnih programa. Potreba za prepoznavanjem tipa u vreme izvršavanja pokreće niz zanimljivih, a često i složenih problema u objektno orijentisanom projektovanju, a otvara i osnovno pitanje: kako bi trebalo da izgleda struktura programa.

Ovo poglavlje obrađuje metode za otkrivanje informacija o objektima i klasama u vreme izvršavanja. To se obavlja na dva načina: tradicionalnim otkrivanjem tipa tokom izvršavanja, kada se pretpostavlja da su svi tipovi dostupni već tokom prevođenja koda, i *refleksivnim* mehanizmom, koji omogućuje otkrivanje informacija o klasama isključivo tokom izvršavanja.

Potreba za prepoznavanjem tipa tokom izvršavanja

Posmatrajmo sada već poznat primer hijerarhije klasa u kojoj se koristi polimorfizam. Generički tip je osnovna klasa **Oblik**, a specifični izvedeni tipovi su **Krug**, **Kvadrat** i **Trougao**:



Ovo je tipičan dijagram hijerarhije klasa u kome se osnovna klasa nalazi na vrhu, a izvedene klase se račvaju niže. Uobičajeni cilj u objektno orijentisanom programiranju jeste da kôd radi s referencama na osnovni tip (u ovom slučaju **Oblik**), jer ako kasnije odlučite da proširite program dodavanjem nove klase (kao što je **Romboid**, izveden iz klase **Oblik**), to neće uticati na najveći deo koda. U ovom primeru, metoda **crtaj**() u interfejsu **Oblik** dinamički je povezana, pa programer klijent treba da poziva metodu **crtaj**() preko generičke reference na **Oblik**. Metoda **crtaj**() se redefiniše u svim izvedenim klasama, a pošto je reč o dinamički povezanoj metodi, ona će se ponašati na odgovarajući način čak i ako se poziva preko generičke reference na objekat klase **Oblik**. To je polimorfizam.

Stoga obično pravite određeni objekat (**Krug**, **Kvadrat** ili **Trougao**), pretvarate ga u opštiji tip **Oblik** (zanemarujete specifični tip objekta) i koristite tu referencu tipa **Oblik** u ostatku programa.

Hijerarhija **Oblik**a se može programirati ovako:

```
//: podaciotipu/Oblici.java
import java.util.*;

abstract class Oblik {
  void crtaj() {  System.out.println(this + ".crtaj()");  }
  abstract public String toString();
}

class Krug extends Oblik {
  public String toString() { return "Krug"; }
}

class Kvadrat extends Oblik {
  public String toString() { return "Kvadrat"; }
}

class Trougao extends Oblik {
  public String toString() { return "Trougao"; }
}

public class Oblici {
  public static void main(String[] args) {
    List<Oblik> listaOblika = Arrays.asList(
       new Krug(), new Kvadrat(), new Trougao()
    );
    for(Oblik oblik : listaOblika)
      oblik.crtaj();
  }
} /* Ispis:
Krug.crtaj()
Kvadrat.crtaj()
Trougao.crtaj()
*///:~
```

Osnovna klasa sadrži metodu **crtaj()** koja indirektno koristi metodu **toString()** za štampanje identifikatora klase pomoću prosleđivanja reference **this** metodi **System.out.println()** (obratite pažnju na to da je metoda **toString()** deklarisana kao apstraktna, da bi se njeni naslednici primorali da je redefinišu i da bi se sprečilo pravljenje objekta tipa **Oblik**). Ukoliko se u izrazu za nadovezivanje znakovnih nizova (u kome postoje znak + i objekti tipa **String**) pojavi neki objekat, automatski se poziva metoda **toString()** prosleđenog objekta da bi predstavila njegov sadržaj u objektu tipa **String**. Svaka izvedena klasa redefiniše metodu **toString()** klase **Object**, pa **crtaj()** na kraju u svakom pojedinačnom slučaju (polimorfno) štampa nešto različito.

U ovom primeru, do svođenja naviše dolazi kada se oblik smesti u **List<Oblik>**. Prilikom svođenja naviše gubi se podatak o tome da su objekti *specifični tipovi* klase **Oblik**. Što se tiče niza, svi njegovi elementi su tipa **Oblik**.

U trenutku kada se element čita iz niza, kontejner – za koji su svi njegovi elementi tipa **Object** – automatski pretvara svoj rezultat ponovo u **Oblik**. To je najosnovniji oblik prepoznavanja tipa u vreme izvršavanja, pošto se u Javi u vreme izvršavanja proverava pravilnost svih konverzija tipova. Upravo to i jeste smisao prepoznavanja tipa u vreme izvršavanja: tokom izvršavanja, prepoznaje se tip objekta.

U ovom slučaju, konverzija je samo delimična: **Object** se konvertuje u tip **Oblik**, a ne do kraja u tipove **Krug**, **Kvadrat** ili **Trougao**. U tom trenutku znate *samo* da je **List<Oblik>** pun objekata tipa **Oblik**. Tokom prevođenja to osigurava kontejner i Javin sistem generičkih tipova, ali tokom izvršavanja obezbeđuje ih konverzija tipa.

Sada na scenu stupa polimorfizam, pa se kôd koji se izvršava za **Oblik** bira prema tome da li se radi o referenci na **Krug**, **Kvadrat** ili **Trougao**. Tako bi uglavnom trebalo da bude, jer želite da najveći deo programa zna što manje o *specifičnim* tipovima objekata, odnosno da radi sa opštom predstavom o porodici objekata (u ovom slučaju, **Oblik**). Takav kôd će se lakše pisati, čitati i održavati, a projekti će se lakše primenjivati, razumeti i menjati. Zbog toga je polimorfizam jedan od opštih principa u objektno orijentisanom programiranju.

Međutim, šta će se desiti ako pri programiranju imate poseban problem koji se najlakše rešava ako znate tačan tip generičke reference? Na primer, pretpostavimo da želite omogućiti korisnicima da posebnom bojom istaknu sve figure određenog oblika? Tako bi mogli da pronađu sve trouglove na ekranu, jer su posebno obojeni. Ili, recimo, metoda je dobila zadatak da rotira sve oblike navedene u nekoj listi, ali krugove nema smisla rotirati, pa biste njih hteli da preskočite. To omogućuje prepoznavanje tipa u vreme izvršavanja: možete da zatražite od reference na **Oblik** da vrati tačan tip na koji ukazuje i tako izabere i izdvoji specijalne slučajeve.

Objekat tipa **Class**

Da biste razumeli kako se u Javi odvija prepoznavanje tipa tokom izvršavanja, prvo morate da shvatite kako se u vreme izvršavanja predstavljaju informacije o tipu. To se postiže pomoću specijalne vrste objekta koji pripada klasi *Class*, a sadrži informacije o klasi. Zapravo, za pravljenje svih objekata „običnih" klasa koristi se upravo objekat klase **Class**. Prepoznavanje tipa u vreme izvršavanja, Java obavlja pomoću **Class** objekta, čak i kada radite nešto poput svođenja naviše. Klasa **Class** obezbeđuje još načina za upotrebljavanje prepoznavanja tipa u vreme izvršavanja.

Za svaku klasu koja je deo programa postoji po jedan objekat tipa **Class**. Kad god napišete i prevedete kôd za neku novu klasu, pravi se i jedan objekat tipa **Class** (i čuva u datoteci **.class** istovetnog imena). Za pravljenje objekta te klase, Javina virtuelna mašina (JVM) koja izvršava program upotrebljava podsistem nazvan *učitavač klasa* (engl. *class loader*).

Taj podsistem može da obuhvati ceo lanac učitavača klasa, ali postoji samo jedan *primordijalni učitavač klasa* koji je deo realizacije JVM-a. Primordijalni učitavač klasa, obično s lokalnog diska, učitava takozvane *klase od poverenja*, među kojima su i klase Java API-ja. U lancu najčešće nije potrebno imati dodatne učitavače klasa, ali ako imate posebne potrebe (kao što su učitavanje klasa na poseban način za podršku aplikacija Web servera ili učitavanje klasa preko mreže), onda postoji način da te klase angažujete.

U JVM-u se sve klase učitavaju dinamički, prilikom prve upotrebe. To se dešava kada program napravi prvu referencu nekog statičnog člana te klase. Ispostavlja se da je i konstruktor statična metoda klase, iako se za njega ne piše rezervisana reč **static**. Zato se i pravljenje novog objekta te klase operatorom **new** smatra za referenciranje statičnog člana te klase.

Dakle, program napisan na Javi ne učitava se potpuno pre nego što počne izvršavanje, nego se delovi programa učitavaju po potrebi. Po tome se Java razlikuje od mnogih tradicionalnih jezika. Dinamičko učitavanje omogućuje ponašanje koje je teško ili nemoguće postići u jezicima sa statičkim učitavanjem, kao što je C++.

Učitavač klasa prvo proverava da li je učitan objekat tipa **Class** za taj tip. Ako nije, podrazumevani učitavač klasa učitava ga pronalaženjem datoteke **.class** s tim imenom. (Ili učitavač klasa, instaliran kao softverski dodatak, traži odgovarajuće bajtkodove u bazi podataka, na primer.) Tokom učitavanja bajtkodova klase, JVM *proverava* da li su bajtovi oštećeni i da li sačinjavaju loš kôd. (To je jedan od načina na koji Java postiže bezbedan rad.)

Kada se objekat **Class** za taj tip učita u memoriju, koristi se za pravljenje svih objekata tog tipa. Evo jednog programa koji će vas u to ubediti:

```
//: podaciotipu/ProdavnicaSlatkisa.java
// Ispitivanje načina rada učitavača klase.

class Bombona {
  static {  print("Ucitavam Bombonu");  }
}

class Zvaka {
  static {  print("Ucitavam Zvaku");  }
}

class Kolacic {
  static {  print("Ucitavam Kolacic");  }
}

public class ProdavnicaSlatkisa {
  public static void main(String[ ] args) {
    print("u metodi main");
    new Bombona();
    print("Posle pravljenja Bombone");
    try {
      Class.forName("Zvaka");
    } catch(ClassNotFoundException e) {
       print("Nisam pronašao klasu Zvaka");
    }
    print("Nakon Class.forName(\"Zvaka\")");
    new Kolacic();
    print("Posle pravljenja Kolacica");
  }
} /* Ispis:
u metodi main
Ucitavam Bombonu
Posle pravljenja Bombone
```

```
Ucitavam Zvaku
Nakon Class.forName("Zvaka")
Ucitavam Kolacic
Posle pravljenja Kolacica
*///:~
```

Klase **Bombona**, **Zvaka** i **Kolacic** sadrže blok **static** koji se izvršava pri njihovom prvom učitavanju. Informacije se ispisuju da biste znali kada se učitava određena klasa. Objekti se u metodi **main**() prave između naredaba za štampanje, da bi se lakše odredio trenutak učitavanja.

Vidite da se svaki objekat klase **Class** učitava tek kada zatreba, a inicijalizacija bloka **static** obavlja se po učitavanju klase.

Posebno je zanimljiv red:

```
Class.forName("Zvaka");
```

Svi objekti tipa **Class** pripadaju klasi **Class**. Objekat tipa **Class** je isti kao i svi drugi objekti, pa možete da dobijete referencu na njega i da radite s njom (to radi učitavač klasa). Jedan od načina za dobijanje reference na objekat tipa **Class** jeste statična metoda **forName()** čiji argument tipa **String** sadrži tekstualno ime (pazite na pravopis, mala i velika slova!) određene klase čiju referencu hoćete. Ova metoda vraća referencu na objekat tipa **Class**, koja je ovde bila zanemarena; metodu **forName**() pozivamo radi njenog sporednog dejstva, a to je učitavanje klase **Zvaka** ako nije već učitana. U postupku učitavanja izvršava se statični blok klase **Zvaka**.

U prethodnom primeru, ako metoda **Class.forName**( ) zakaže zato što ne može da pronađe klasu koju treba da učita, generisaće izuzetak **ClassNotFoundException**. Ovde samo prijavljujemo problem i idemo dalje, ali u sofisticiranijim programima mogli biste pokušati da rešite problem u bloku za obradu izuzetaka.

Kad god budete hteli da upotrebite podatke o tipu prepoznate u vreme izvršavanja, najpre ćete morati da pribavite referencu odgovarajućeg objekta **Class**. Jedan od podesnih načina za to pruža metoda **Class.forName**( ), zato što vam nije potreban objekat tog tipa da biste dobili referencu klase **Class**. Međutim, ukoliko već imate objekat tipa koji vas zanima, referencu klase **Class** možete pribaviti pozivanjem metode **getClass**( ) koja je deo korenske klase **Object**. Ona vraća referencu klase **Class** koja predstavlja stvarni tip objekta. **Class** ima mnogo zanimljivih metoda; evo nekih od njih:

```
//: podaciotipu/igracke/IspitivanjeIgracaka.java
// Testiranje klase Class.
package podaciotipu.igracke;
import static net.mindview.util.Print.*;

interface ImaBaterije {}
interface OtpornaNaVodu {}
interface Puca {}

class Igracka {
  // Sledeći podrazumevani konstruktor pretvorite u komentar
  // da biste videli izuzetak NoSuchMethodError od (*1*)
```

```
  Igracka() {}
  Igracka(int i) {}
}

class LepaIgracka extends Igracka
implements ImaBaterije, OtpornaNaVodu, Puca {
  LepaIgracka() { super(1); }
}

public class IspitivanjeIgracaka {
  static void printInfo(Class cc) {
    print("Ime klase: " + cc.getName() +
      " je interfejs? [" + cc.isInterface() + "]");
    print("Prosto ime: " + cc.getSimpleName());
    print("Kanonsko ime : " + cc.getCanonicalName());
  }
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("typeinfo.igracke.LepaIgracka");
    } catch(ClassNotFoundException e) {
      print("Ne mogu da pronađem klasu LepaIgracka");
      System.exit(1);
    }
    printInfo(c);
    for(Class fejs : c.getInterfaces())
      printInfo(fejs);
    Class nad = c.getSuperclass();
    Object obj = null;
    try {
      // Zahteva podrazumevani konstruktor:
      obj = nad.newInstance();
    } catch(InstantiationException e) {
      print("Ne mogu da napravim primerak");

      System.exit(1);
       } catch(IllegalAccessException e) {
         print("Ne mogu da pristupim");
      System.exit(1);
    }
       printInfo(obj.getClass());
  }
} /* Ispis:
Ime klase: typeinfo.igracke.LepaIgracka je interfejs? [false]
Prosto ime: LepaIgracka
Kanonsko ime : typeinfo.igracke.LepaIgracka
Ime klase: typeinfo.igracke.ImaBaterije je interfejs? [true]
Prosto ime: ImaBaterije
Kanonsko ime : typeinfo.igracke.ImaBaterije
Ime klase: typeinfo.igracke.OtpornaNaVodu je interfejs? [true]
Prosto ime: OtpornaNaVodu
```

```
Kanonsko ime : typeinfo.igracke.OtpornaNaVodu
Ime klase: typeinfo.igracke.Puca je interfejs? [true]
Prosto ime: Puca
Kanonsko ime : typeinfo.igracke.Puca
Ime klase: typeinfo.igracke.Toy je interfejs? [false]
Prosto ime: Toy
Kanonsko ime : typeinfo.igracke.Toy
*///:~
```

LepaIgracka nasleđuje klasu **Igracka** i realizuje interfejse **ImaBaterije**, **OtpornaNaVodu** i **Puca**. U metodi **main( )**, pravi se referenca klase **Class** i inicijalizuje na objekat **LepaIgracka** tipa **Class** pomoću metode **forName( )** u odgovarajućem bloku **try**. Vodite računa o tome da morate navesti puno ime (uključujući i ime paketa) u znakovnom nizu koji prosleđujete metodi **forName( )**.

Metoda **printInfo( )** upotrebljava **getName( )** za dobijanje punog imena klase, a **getSimpleName( )** i **getCanonicalName( )** – uvedene u Javi SE5 – za dobijanje imena klase bez imena paketa, odnosno punog imena. Kao što joj ime govori, metoda **isInterface( )** kazuje da li dati objekat tipa **Class** predstavlja interfejs. Dakle, pomoću objekta tipa **Class** možete saznati zaista sve o određenom tipu.

U metodi **main( )** poziva se metoda **Class.getInterfaces( )**; ona vraća niz **Class** objekata koji predstavlja interfejse obuhvaćene u **Class** objektu o kojem je reč.

Ukoliko imate objekat tipa **Class**, metodom **getSuperclass( )** možete ga pitati koja je njegova neposredna natklasa. Metoda vraća referencu objekta tipa **Class** koju možete ispitivati dalje. Dakle, u vreme izvršavanja možete saznati celokupnu hijerarhiju klasa svakog objekta.

Metoda **newInstance( )** klase **Class** jeste jedan od načina da se realizuje „virtuelni konstruktor", koji omogućava da kažete: „Ne znam tačno tvoj tip, ali svejedno je – napravi se kako treba". U prethodnom primeru, **nad** je samo **Class** referenca bez ikakvih daljih podataka o tipu koji bi bili poznati u vreme prevođenja. I kada napravite nov primerak neke klase, vraća vam se referenca tipa **Object**. Ali ta referenca pokazuje na objekat tipa **Igracka**. Naravno, da biste mogli slati i druge poruke sem onih koje prihvata klasa **Object**, morate da saznate pravi tip i izvršite konverziju tipa. Sem toga, klasa koja se pravi metodom **newInstance( )** mora imati podrazumevani konstruktor. U nastavku poglavlja videćete kako se objekti klasa prave dinamički, pomoću bilo kog konstruktora, u Javinom *refleksivnom* interfejsu za programiranje.

Vežba 1: (1) U programu **IspitivanjeIgracaka.java**, pretvorite podrazumevani konstruktor klase **Igracka** u komentar i objasnite šta se zbog toga dešava.

Vežba 2: (2) U program **IspitivanjeIgracaka.java** dodajte novu vrstu interfejsa. Dokažite da se on ispravno tretira i prikazuje.

Vežba 3: (2) Dodajte **Romboid** u program **Oblici.java**. Napravite jedan **Romboid**, svedite ga naviše na **Oblik**, zatim ponovo naniže na **Romboid**. Pokušajte da svedete naniže na **Krug** i vidite šta se dešava.

Vežba 4: (2) Prepravite prethodnu vežbu tako da se naredbom **instanceof** tip utvrdi pre svođenja naniže.

Vežba 5: (3) U programu **Oblici.java** realizujte metodu **rotiraj(Oblik)** tako da proverava da li joj je zadato da rotira **Krug** (i neka ga u tom slučaju ne rotira).

Vežba 6: (4) Prepravite program **Oblici.java** tako da ističe sve oblike određenog tipa, odnosno postavlja indikator u njima. Metoda **toString( )** svake izvedene potklase klase **Oblik** treba da pokaže da li je taj **Oblik** „istaknut".

Vežba 7: (3) Prepravite **ProdavnicaSlatkisa.java** tako da svaku vrstu pravljenja objekta određuje argument koji se zadaje na komandnoj liniji. Dakle, ako na komandnoj liniji piše **„java ProdavnicaSlatkisa Bombon"**, onda se pravi samo objekat tipa **Bombon**. Obratite pažnju na to kako argumentom s komandne linije određujete **Class** objekte koji će biti učitani.

Vežba 8: (5) Napišite metodu koja prima objekat i rekurzivno ispisuje sve klase u njegovoj hijerarhiji.

Vežba 9: (5) Prepravite prethodnu vežbu tako da se pomoću metode **Class.getDeclaredFields( )** prikazuju i podaci o poljima klase.

Vežba 10: (3) Napišite program koji utvrđuje da li je niz elemenata tipa **char** prost tip ili pravi objekat.

Literali klase

Java nudi još jedan način za dobijanje reference na objekat klase **Class**, a to je korišćenje *literala klase* (engl. *class literal*). U prethodno navedenom programu to bi izgledalo ovako:

```
LepaIgracka.class;
```

što ne samo da je jednostavnije, već je i bezbednije jer se proverava već tokom prevođenja (te stoga ne mora biti smešteno u blok **try**). Pošto se ne poziva metoda **forName()**, istovremeno je i efikasnije.

Literali klase rade sa običnim klasama, ali i sa interfejsima, nizovima i prostim tipovima. Pored toga, sve omotačke klase prostih tipova sadrže standardno polje **TYPE**. Ovo polje daje referencu na objekat tipa **Class** za sve odgovarajuće proste tipove, kao što su:

...ekvivalentno je...	
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

Preporučujem da koristite verzije **class** ako je to moguće, pošto se tako pristupa i običnim klasama.

Zanimljivo je primetiti da se pravljenjem reference na objekat tipa **Class** literalom **class**, taj objekat ne inicijalizuje automatski. Priprema klase za upotrebu zapravo obuhvata tri koraka:

1. *Učitavanje*, koje obavlja učitavač klasa. Ona pronalazi bajtkod (obično na disku navedenom u sistemskoj promenljivoj **classpath**, ali to ne mora biti slučaj) i od njih pravi objekat tipa **Class**.
2. *Povezivanje*. U fazi povezivanja proverava se bajtkod klase, dodeljuje memorija statičnim poljima, i ako treba, razrešavaju sve reference na druge klase koje ova klasa pravi.
3. *Inicijalizaciju*. Ako postoji natklasa, ona se inicijalizuje. Izvršavaju se statični inicijalizatori i statični blokovi inicijalizatora.

Inicijalizacija se odgađa do prve reference na neku statičnu metodu (konstruktor je implicitno statičan) ili na neko nekonstantno statično polje:

```
//: podaciotipu/InicijalizacijaObjektaTipaClass.java
import java.util.*;

class Initabela {
  static final int staticFinal = 47;
  static final int staticFinal2 =
    ClassInitialization.slucajan.nextInt(1000);
  static {
    System.out.println("Inicijalizacija Initabele");
  }
}

class Initabela2 {
  static int staticnaNeFinalna = 147;
  static {
    System.out.println("Inicijalizacija Initabele2");
  }
}

class Initabela3 {
  static int staticnaNeFinalna = 74;
  static {
    System.out.println("Inicijalizacija Initabele3");
  }
}

public class InicijalizacijaObjektaTipaClass {
  public static Random slucajan = new Random(47);
  public static void main(String[] args)
throws Exception {
```

```
    Class Initabela = Initabela.class;
    System.out.println("Nakon pravljenja ref Initabele");
    // Ne pokreće inicijalizaciju:
    System.out.println(Initabela.staticFinal);
    // Pokreće inicijalizaciju:
    System.out.println(Initabela.staticFinal2);
    // Pokreće inicijalizaciju:
    System.out.println(Initabela2.staticnaNeFinalna);
    Class Initabela3 = Class.forName("Initabela3");
    System.out.println("Nakon pravljenja ref Initabele3");
    System.out.println(Initabela3.staticnaNeFinalna);
  }
} /* Ispis:
Nakon pravljenja ref Initabele
47
Inicijalizacija Initabele
258
Inicijalizacija Initabele2
147
Inicijalizacija Initabele3
Nakon pravljenja ref Initabele3
74
*///:~
```

Zapravo, inicijalizacija je „lenja koliko god je to moguće". Nakon pravljenja reference objekta **Initabela**, vidite da upotreba sintakse **.class** za dobijanje reference na klasu ne prouzrokuje inicijalizaciju. Međutim, **Class.forName( )** odmah inicijalizuje klasu da bi proizvela **Class** referencu, kao što vidite iz pravljenja objekta **Initabela3**.

Ukoliko je neka statična finalna vrednost „konstanta u vreme prevođenja", kao što je **Initabela.staticFinal**, ta vrednost se može pročitati a da klasa **Initabela** ne bude inicijalizovana. Međutim, ako je neko polje statično i finalno, takvo ponašanje nije zajemčeno: pristupanje polju **Initabela.staticFinal2** nameće inicijalizaciju klase, pošto ono ne može biti konstanta u vreme prevođenja.

Ako neko statično polje nije finalno, pristupanje njemu uvek zahteva povezivanje (da bi se polju dodelila memorija) i inicijalizaciju (da bi se inicijalizovala ta memorija) pre nego što se polje počne čitati, kao što vidite iz pristupanja polju **Initabela2.staticnaNeFinalna**.

Generičke reference klasa

Class referenca upućuje na neki **Class** objekat koji proizvodi instance klasa i sadrži kôd svih metoda za te instance. Sadrži i statične delove te klase. Stoga **Class** referenca zapravo pokazuje tačan tip onoga na šta upućuje: određenog objekta klase **Class**.

Međutim, projektanti Jave SE5 ugrabili su priliku da to naprave malo specifičnijim, tako što su omogućili da generičkom sintaksom ograničite tip **Class** objekta na koji upućuje **Class** referenca. U narednom primeru, obe sintakse su ispravne:

```
//: podaciotipu/GenerickeClassReference.java

public class GenerickeClassReference {
  public static void main(String[ ] args) {
    Class intClass = int.class;
    Class<Integer> genericIntClass = int.class;
    genericIntClass = Integer.class; // Ista stvar
    intClass = double.class;
    // genericIntClass = double.class; // Nije dozvoljeno
  }
} ///:~
```

Obična referenca klase ne prouzrokuje upozorenje. Međutim, vidite da se obična referenca klase može dodeliti bilo kojem drugom objektu tipa **Class**, dok se generička referenca klase može dodeliti samo svom deklarisanom tipu. Korišćenjem generičke sintakse omogućavate prevodiocu da sprovede još jednu proveru tipova.

Kako biste to ograničenje malo ublažili? Na prvi pogled, izgleda da biste mogli uraditi nešto poput:

```
Class<Number> genericNumberClass = int.class;
```

Ovo naizgled ima smisla jer se **Integer** nasleđuje od klase **Number**. Ali to nije ispravno, zato što **Integer Class** objekat nije potklasa **Number Class** objekta. (Razlika vam je možda previše tanana; razmotrićemo je pobliže u poglavlju *Generički tipovi*).

Da bih ublažio ograničenja prilikom korišćenja generičkih **Class** referenci, upotrebljavam *džokera* koji je deo Javinih generičkih tipova. Džokerski znak je **?** i on naznačuje „bilo šta". Stoga običnoj **Class** referenci u gornjem primeru možemo dodati džokere i dobićemo iste rezultate:

```
//: podaciotipu/DzokerskeClassReference.java

public class DzokerskeClassReference {
  public static void main(String[ ] args) {
    Class<?> intClass = int.class;
    intClass = double.class;
  }
} ///:~
```

U Javi SE5, **Class<?>** treba da ima prednost nad golim **Class** iako su ekvivalentni, a golo **Class**, kao što ste videli, ne prouzrokuje upozorenje prevodioca. Prednost reference **Class<?>** jeste to što pokazuje da nespecifičnu referencu klase ne upotrebljavate slučajno ili zbog neznanja, nego *namerno*.

Za pravljenje **Class** reference ograničene na određeni tip ili *bilo koji njegov podtip*, kombinujte džokera i rezervisanu reč **extends**; time pravite *ograničenje*. Dakle, umesto da kažete samo **Class<Number>**, recite:

```
//: podaciotipu/OgraniceneClassReference.java

public class OgraniceneClassReference {
  public static void main(String[ ] args) {
    Class<? extends Number> ogranicen = int.class;
    ogranicen = double.class;
    ogranicen = Number.class;
    // Ili bilo šta drugo izvedeno iz klase Number.
  }
} ///:~
```

Generička sintaksa se dodaje **Class** referencama samo da bi se proverili tipovi u vreme prevođenja, pa ako negde pogrešite, saznaćete to nešto ranije. Ni sa običnim **Class** referencama ne možete zastraniti, ali ako negde pogrešite, saznaćete to tek nakon izvršavanja programa, što ume da bude nezgodno.

Evo primera upotrebe generičke sintakse. U programu se skladišti referenca klase i kasnije pravi **Lista** popunjena objektima generisanim metodom **newInstance( )**:

```
//: podaciotipu/PopunjenaLista.java
import java.util.*;

class PrebrojaniCeoBroj {
  private static long brojac;
  private final long id = brojac++;
  public String toString() { return Long.toString(id); }
}

public class PopunjenaLista<T> {
  private Class<T> tip;
  public PopunjenaLista(Class<T> tip) { this.tip = tip; }
  public List<T> create(int nElemenata) {
    List<T> rezultat = new ArrayList<T>();
    try {
         for(int i = 0; i < nElemenata; i++)
        rezultat.add(tip.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return rezultat;
  }
  public static void main(String[ ] args) {
    PopunjenaLista<PrebrojaniCeoBroj> pl =
      new PopunjenaLista<PrebrojaniCeoBroj>(PrebrojaniCeoBroj.class);
    System.out.println(pl.create(15));
  }
} /* Ispis:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Vodite računa o tome da ova klasa mora pretpostaviti kako svaki tip s kojim radi ima podrazumevani konstruktor (onaj bez argumenata), inače ćete dobiti izuzetak. Za ovaj program prevodilac ne generiše nikakvo upozorenje.

Zanimljiva se stvar dešava kada za objekte tipa **Class** upotrebite generičku sintaksu: metoda **newInstance( )** vratiće tačan tip objekta, a ne samo tip **Object** kako ste videli u programu **IspitivanjeIgracaka.java**. To je donekle ograničeno:

```
//: podaciotipu/igracke/GenerickoIspitivanjeIgracaka.java
// Ispitivanje klase Class.
package podaciotipu.igracke;

public class GenerickoIspitivanjeIgracaka {
  public static void main(String[ ] args) throws Exception {
    Class<LepaIgracka> liClass = LepaIgracka.class;
    // Daje tačan tip:
    LepaIgracka LepaIgracka = liClass.newInstance();
    Class<? super LepaIgracka> nad = liClass.getSuperclass();
    // Ovo se ne bi prevelo:
    // Class<Igracka> nad2 = liClass.getSuperclass();
    // Daje običan tip Object:
       Object obj = nad.newInstance();
  }
} ///:~
```

Ako dobijete natklasu, prevodilac će dozvoliti samo da kažete kako je referenca natklase „neka klasa koja je natklasa klase **LepaIgracka**", kao što se vidi iz izraza **Class<? super LepaIgracka>**. Neće prihvatiti deklaraciju **Class<Igracka>**. To izgleda pomalo čudno, zato što **getSuperclass( )** vraća osnovnu *klasu* (ne interfejs), a prevodilac u vreme prevođenja zna koja je to klasa – u ovom slučaju, **Igracka.class**, a ne samo „neka natklasa od **LepaIgracka**". U svakom slučaju, zbog te neodređenosti, povratna vrednost metode **nad.newInstance( )** nije tačan tip, nego samo **Object**.

Nova sintaksa za konverziju tipova

U Javu SE5 dodata je i posebna sintaksa za konverziju tipova koja se upotrebljava za **Class** reference; to je metoda **cast( )**:

```
//: podaciotipu/KonverzijaClassTipova.java

class Zgrada {}
class Kuca extends Zgrada {}
public class KonverzijaClassTipova {
  public static void main(String[] args) {
    Zgrada z = new Kuca();
    Class<Kuca> tipKuce = Kuca.class;
    Kuca k = tipKuce.cast(z);
    k = (Kuca)z; // ... ili uradite samo ovo.
  }
} ///:~
```

Metoda **cast( )** prima objekat kao argument i pretvara ga u tip **Class** reference. Naravno, nakon što pogledate gornji kôd, to izgleda kao mnogo više posla nego u poslednjem redu metode **main( )** koji radi istu stvar.

Nova sintaksa za konverziju tipova je podesna za situacije u kojima *ne možete* da upotrebite običnu konverziju tipa. To se najčešće događa kada pišete generički kôd (što ćete naučiti u poglavlju *Generički tipovi*) i uskladištili ste **Class** referencu koju kasnije nameravate da upotrebite za konverziju. Ispostavlja se da je to retko potrebno – u celoj biblioteci Jave SE5 našao sam samo jedan slučaj gde je metoda **cast( )** bila korišćena (u **com.sun.mirror.util.DeclarationFilter**).

Jedna nova mogućnost uopšte *nije* našla primenu u biblioteci Jave SE5: **Class.asSubclass( )**. Ona služi za konverziju objekta tipa **Class** u specifičniji tip.

Provera pre konverzije tipa

Dosad ste videli sledeće načine za prepoznavanje tipa tokom izvršavanja:

1. Klasična konverzija tipa, npr. **(Oblik)**, koja tokom izvršavanja prepoznaje tip da bi proverila ispravnost konverzije tipa ili generisala izuzetak tipa **ClassCastException** ako je konverzija loša.
2. Objekat tipa **Class** koji predstavlja tip objekta. Objekat tipa **Class** se može ispitivati tokom izvršavanja da bi se saznale korisne informacije.

U jeziku C++, klasična konverzija tipa **(Oblik)** *ne prepoznaje* stvarni tip tokom izvršavanja, već samo saopštava prevodiocu da objekat posmatra kao novi tip. U Javi, koja proverava tip, ovakva konverzija tipa često se naziva „svođenje naniže koje čuva tip" (engl. *type-safe downcast*). Ime „svođenje naniže" (engl. *downcast*) nastalo je zbog uobičajenog izgleda dijagrama hijerarhije klasa. Ako je konverzija iz klase **Krug** u **Oblik** svođenje naviše (engl. *upcast*), onda je svođenje iz **Oblik** u **Krug** svođenje naniže. Međutim, pošto znamo da je **Krug** istovremeno i **Oblik**, prevodilac dozvoljava dodelu uz svođenje naviše bez ikakvih ograničenja. Prevodilac *ne može* znati, za dati **Oblik**, šta je taj **Oblik** zapravo – možda je baš **Oblik**, ili podtip klase **Oblik**, kao što su **Krug**, **Kvadrat**, **Trougao** ili drugi oblik. U vreme prevođenja, prevodilac vidi samo **Oblik**. Stoga prevodilac ne dozvoljava dodelu uz svođenje naniže bez eksplicitnog korišćenja operatora konverzije tipa, da bi mu saopštio kako vi imate dodatne podatke na osnovu kojih znate tačan tip (prevodilac će proveriti da li je to svođenje naniže razborito i neće dozvoliti svođenje naniže na tip koji nije potklasa).

Postoji i treći oblik prepoznavanja tipa tokom izvršavanja u Javi. To je rezervisana reč **instanceof** koja proverava da li je objekat instanca određenog tipa. Ona vraća rezultat tipa **boolean**, pa je na sledeći način možete koristiti u logičkim izrazima:

```
if(x instanceof Pas)
 ((Pas)x).laj();
```

Gornja naredba **if** proverava da li objekat **x** pripada klasi **Pas** *pre* nego što izvrši konverziju u tip **Pas**. Kada nema drugih informacija o tipu objekta, važno je da se **instanceof** upotrebi pre svođenja naniže; u suprotnom, nastaće izuzetak tipa **ClassCastException**.

Obično ćete biti usredsređeni na jedan tip (npr. da pretvorite sve trouglove u ružičaste), ali pomoću rezervisane reči **instanceof** možete lako proveriti *sve* objekte. Pretpostavimo da imamo porodicu klasa za opisivanje kućnih **Ljubimac**a (i njihovih vlasnika, što će nam zatrebati kasnije). Svaka **Jedinka** u hijerarhiji ima **id** i opciono ime. Iako sledeće klase nasleđuju klasu **Jedinka**, ona je donekle složena, pa ćemo njen kôd prikazati i objasniti u poglavlju *Detaljno razmatranje kontejnera*. Kao što zapažate, u ovom trenutku i nije neophodno da vidite kôd klase **Jedinka** – treba samo da znate sledeće: možete je napraviti sa imenom ili bez imena i svaka **Jedinka** ima metodu **id( )** koja vraća jedinstven identifikator (napravljen brojanjem svakog objekta). Tu je i metoda **toString( )**; ako napravite primerak klase **Jedinka**, a ne date joj ime, metoda **toString( )** vratiće samo prosto ime tipa.

Ova hijerarhija klasa nasleđuje klasu **Jedinka**:

```
//: podaciotipu/ljubimci/Osoba.java
package podaciotipu.ljubimci;

public class Osoba extends Jedinka {
  public Osoba(String ime) { super(ime); }
} ///:~

//: podaciotipu/ljubimci/Ljubimac.java
package podaciotipu.ljubimci;

public class Ljubimac extends Jedinka {
  public Ljubimac(String ime) { super(ime); }
  public Ljubimac() { super(); }
} ///:~
//: podaciotipu/ljubimci/Pas.java
package podaciotipu.ljubimci;

public class Pas extends Ljubimac {
  public Pas(String ime) { super(ime); }
  public Pas() { super(); }
} ///:~

//: podaciotipu/ljubimci/Mesanac.java
package podaciotipu.ljubimci;

public class Mesanac extends Pas {
  public Mesanac(String ime) { super(ime); }
  public Mesanac() { super(); }
} ///:~

//: podaciotipu/ljubimci/Mops.java
package podaciotipu.ljubimci;

public class Mops extends Pas {
  public Mops(String ime) { super(ime); }
```

```
  public Mops() { super(); }
} ///:~

//: podaciotipu/ljubimci/Macka.java
package podaciotipu.ljubimci;

public class Macka extends Ljubimac {
  public Macka(String ime) { super(ime); }
  public Macka() { super(); }
} ///:~

//: podaciotipu/ljubimci/Egipatska.java
package podaciotipu.ljubimci;

public class Egipatska extends Macka {
  public Egipatska(String ime) { super(ime); }
  public Egipatska() { super(); }
} ///:~
//: podaciotipu/ljubimci/Manska.java
package podaciotipu.ljubimci;

public class Manska extends Macka {
  public Manska(String ime) { super(ime); }
  public Manska() { super(); }
} ///:~

//: podaciotipu/ljubimci/Velska.java
package podaciotipu.ljubimci;

public class Velska extends Manska {
  public Velska(String ime) { super(ime); }
  public Velska() { super(); }
} ///:~

//: podaciotipu/ljubimci/Glodar.java
package podaciotipu.ljubimci;

public class Glodar extends Ljubimac {
  public Glodar(String ime) { super(ime); }
  public Glodar() { super(); }
} ///:~

//: podaciotipu/ljubimci/Pacov.java
package podaciotipu.ljubimci;

public class Pacov extends Glodar {
  public Pacov(String ime) { super(ime); }
  public Pacov() { super(); }
} ///:~
```

```
//: podaciotipu/ljubimci/Mis.java
package podaciotipu.ljubimci;

public class Mis extends Glodar {
  public Mis(String ime) { super(ime); }
  public Mis() { super(); }
} ///:~
//: podaciotipu/ljubimci/Hrcak.java
package podaciotipu.ljubimci;

public class Hrcak extends Glodar {
  public Hrcak(String ime) { super(ime); }
  public Hrcak() { super(); }
} ///:~
```

Sada nam treba način za slučajno generisanje različitih vrsta ljubimaca, i da bi nam bilo lakše, za pravljenje nizova i **Lista** ljubimaca. Da bi ova alatka mogla da se razvija kroz nekoliko različitih realizacija, definisaćemo je kao apstraktnu klasu:

```
//: podaciotipu/ljubimci/PravljenjeLjubimaca.java
// Pravi slučajne sekvence Ljubimaca.
package podaciotipu.ljubimci;
   import java.util.*;

public abstract class PravljenjeLjubimaca {
  private Random slucajan = new Random(47);
  // Lista različitih vrsta Ljubimaca koje treba napraviti:
  public abstract List<Class<? extends Ljubimac>> tipovi();
  public Ljubimac nekiLjubimac() { // Napravi jednog nasumičnog Ljubimca
    int n = slucajan.nextInt(tipovi().size());
    try {
      return tipovi().get(n).newInstance();
    } catch(InstantiationException e) {
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
  public Ljubimac[ ] createArray(int velicina) {
    Ljubimac[ ] result = new Ljubimac[velicina];
    for(int i = 0; i < velicina; i++)
      rezultat[i] = nekiLjubimac();
    return rezultat;
  }
  public ArrayList<Ljubimac> arrayList(int velicina) {
    ArrayList<Ljubimac> rezultat = new ArrayList<Ljubimac>();
    Collections.addAll(rezultat, createArray(velicina));
    return rezultat;
  }
} ///:~
```

Da bi dobila **Listu** objekata tipa **Class**, apstraktna metoda **tipovi( )** obraća se nekoj izvedenoj klasi – ovo je varijanta projektnog obrasca *Template Method* (Šablonska metoda). Pošto je zadato da tip klase bude „nešto izvedeno iz klase **Ljubimac**", metoda **newInstance( )** pravi objekat tipa **Ljubimac** bez konverzije tipa. **nekiLjubimac( )** nasumično indeksira **Listu** i izabrane objekte tipa **Class** upotrebljava za generisanje nove instance te klase pomoću metode **Class.newInstance( )**. Metoda **createArray( )** upotrebljava metodu **nekiLjubimac( )** za popunjavanje niza, a **arrayList( )** upotrebljava **createArray( )**.

Kada pozovete metodu **newInstance( )**, možete da prouzrokujete dve vrste izuzetaka koje su obrađene u odredbama **catch** iza bloka **try**. Podsetimo se, imena izuzetaka su relativno jasna objašnjenja onoga što je zakazalo (**IllegalAccessException** označava kršenje pravila Javinog mehanizma bezbednosti, u ovom slučaju podrazumevani konstruktor je privatan).

Kada izvodite potklasu klase **PravljenjeLjubimaca**, jedino morate da navedete **Listu** tipova ljubimaca koju hoćete da napravite metodom **nekiLjubimac( )** i drugim metodama. Metoda **getTypes( )** obično vraća samo referencu neke statične **Liste**. Evo jedne realizacije metodom **forName( )**:

```
//: podaciotipu/ljubimci/ForNamePravljenje.java
package podaciotipu.ljubimci;
import java.util.*;

public class ForNamePravljenje extends PravljenjeLjubimaca {
  private static List<Class<? extends Ljubimac>> tipovi =
    new ArrayList<Class<? extends Ljubimac>>();
  // Tipovi koje hoćete da napravite nasumično:
  private static String[ ] imenaTipova = {
    "podaciotipu.ljubimci.Mesanac",
    "podaciotipu.ljubimci.Mops",
    "podaciotipu.ljubimci.Egipatska",
    "podaciotipu.ljubimci.Manska",
    "podaciotipu.ljubimci.Velska",
    "podaciotipu.ljubimci.Pacov",
    "podaciotipu.ljubimci.Mis",
    "podaciotipu.ljubimci.Hrcak"
  };
  @SuppressWarnings("unchecked")
  private static void loader() {
    try {
      for(String ime : imenaTipova)
        tipovi.add(
          (Class<? Extends Ljubimac>)Class.forName(ime));
    } catch(ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
  static { loader(); }
  public List<Class<? extends Ljubimac>> tipovi() {return tipovi;}
} ///:~
```

Metoda **loader( )** pravi **Listu** objekata tipa **Class** pomoću metode **Class.forName( )**. To može da prouzrokuje izuzetak **ClassNotFoundException**, što ima smisla jer joj prosleđujete znakovni niz čija se validnost ne može proveriti u vreme prevođenja. Pošto su objekti tipa **Ljubimac** u paketu **podaciotipu**, prilikom navođenja imena klase mora se navesti i ime paketa.

Da bi se napravila **List**a objekata tipa **Class** za različite tipove, potrebna je konverzija tipova koja prouzrokuje upozorenje u vreme prevođenja. Metodu **loader( )** definišemo zasebno i zatim smeštamo unutar statičnog bloka za inicijalizaciju, zato što se anotacija **@SuppressWarnings** ne može napisati direktno unutar statičnog bloka za inicijalizaciju.

Za brojanje primeraka klase **Ljubimci** treba nam alatka koja prati broj različitih tipova primeraka klase **Ljubimci**. Za to je **Map**a savršena; ključevi su imena tipova **Ljubimac**a, a vrednosti **Integer**i koji odražavaju broj **Ljubimac**a. Na taj način možete reći: „Koliko ima objekata tipa **Hrcak**?" Za brojanje primeraka klase **Ljubimci** možemo upotrebiti rezervisanu reč **instanceof**:

```
//: podaciotipu/PrebrojavanjeLjubimaca.java
// Upotreba rezervisane reči instanceof.
import podaciotipu.ljubimci.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PrebrojavanjeLjubimaca {
  static class BrojacLjubimaca extends HashMap<String,Integer> {
    public void prebroj(String tip) {
      Integer kolicina = get(tip);
      if(kolicina == null)
        put(tip, 1);
        else
        put(tip, kolicina + 1);
    }
  }
  public static void
  prebrojLjubimce(PravljenjeLjubimaca pravljenje) {
    BrojacLjubimaca brojac= new BrojacLjubimaca();
    for(Ljubimac ljubimac : pravljenje.createArray(20)) {
      // Ispiši svakog pojedinog ljubimca:
      printnb(ljubimac.getClass().getSimpleName() + " ");
      if(ljubimac instanceof Ljubimac)
        brojac.prebroj("Ljubimac");
      if(ljubimac instanceof Pas)
        brojac.prebroj("Pas");
      if(ljubimac instanceof Mesanac)
        brojac.prebroj("Mesanac");
      if(ljubimac instanceof Mops)
        brojac.prebroj("Mops");
      if(ljubimac instanceof Macka)
        brojac.prebroj("Macka");
```

```
      if(ljubimac instanceof Manska)
        brojac.prebroj("Egipatska");
      if(ljubimac instanceof Manska)
        brojac.prebroj("Manska");
      if(ljubimac instanceof Manska)
        brojac.prebroj("Velska");
      if(ljubimac instanceof Glodar)
        brojac.prebroj("Glodar");
      if(ljubimac instanceof Pacov)
        brojac.prebroj("Pacov");
      if(ljubimac instanceof Mis)
        brojac.prebroj("Mis");
      if(ljubimac instanceof Hrcak)
        brojac.prebroj("Hrcak");
    }
    // Ispiši broj ljubimaca:
    print();
    print(brojac);
  }
  public static void main(String[] args) {
    prebrojLjubimce(new ForNamePravljenje());
  }
} /* Ispis:
Pacov Manska Velska Mesanac Mops Velska Mops Manska Velska Pacov Egipatska
Hrcak Egipatska Mesanac Mesana Velska Mis Mops Mis Velska
{Mops=3, Macka=9, Hrcak=1, Velska=7, Mis=2, Mesanac=3, Glodar=5, Ljubimac=20,
Manska=7, Egipatska=7, Pas=6, Pacov=2}
*///:~
```

U metodi **prebrojLjubimce( )**, **PravljenjeLjubimaca** nasumično popunjava niz primercima klase **Ljubimci**. Zatim se svaki **Ljubimac** u nizu ispituje i broji pomoću rezervisane reči **instanceof**.

Rezervisana reč **instanceof** ima prilično blago ograničenje: omogućuje poređenja samo sa imenovanim tipom, a ne sa objektom tipa **Class**. Gledajući gornji primer, možda ste pomislili kako je dosadno pisati sve te izraze sa **instanceof**, i u pravu ste. Međutim, nema načina za pametno automatizovanje poređenja pomoću **instanceof** tako što će se napraviti **niz** objekata tipa **Class** i porediti s njima (ne očajavajte, ipak postoji alternativno rešenje). To nije veliko ograničenje kao što možda mislite, pošto vam projekat i nije baš najbolji ako pišete mnogo izraza **instanceof**.

Korišćenje literala klase

Ako prepravimo primer **PravljenjeLjubimaca.java** tako da koristi literale klase, dobićemo mnogo čitljiviji kôd:

```
//: podaciotipu/ljubimci/RucnoPravljenjeLjubimaca.java
// Korišćenje literala klase.
package podaciotipu.ljubimci;
import java.util.*;
```

```
public class RucnoPravljenjeLjubimaca extends PravljenjeLjubimaca {
  // Blok try nije potreban.
  @SuppressWarnings("unchecked")
  public static final List<Class<? extends Ljubimac>> sviTipovi =
    Collections.unmodifiableList(Arrays.asList(
      Ljubimac.class, Macka.class, Glodar.class,
      Mesanac.class, Mops.class, Egipatska.class, Manska.class,
      Velska.class, Pacov.class, Mis.class,Hrcak.class));
  // Tipovi za nasumično pravljenje:
  private static final List<Class<? extends Ljubimac>> tipovi =
    sviTipovi.subList(sviTipovi.indexOf(Mesanac.class),
      sviTipovi.size());
  public List<Class<? extends Ljubimac>> tipovi() {
    return tipovi;
  }
     public static void main(String[] args) {
       System.out.println(tipovi);
  }
} /* Ispis:
[class podaciotipu.ljubimci.Mesanac, class podaciotipu.ljubimci.Mops,
class podaciotipu.ljubimci.Egipatska, class podaciotipu.ljubimci.Manska,
class podaciotipu.ljubimci.Velska, class podaciotipu.ljubimci.Pacov,
class podaciotipu.ljubimci.Mis, class podaciotipu.ljubimci.Hrcak]
*///:~
```

U primeru **PrebrojavanjeLjubimaca3.java** koji ćemo prikazati u narednom odeljku, moramo unapred da popunimo **Map**u svim tipovima klase **Ljubimac** (ne samo onima koji će biti nasumično generisani), pa nam je neophodna **List**a **sviTipovi**. Lista **tipovi** je deo liste **sviTipovi** – napravljene metodom **List.subList( )**; ona obuhvata tačne tipove ljubimaca, pa se upotrebljava za nasumično pravljenje **Ljubimac**a.

Ovoga puta lista **tipovi** ne mora da se pravi u bloku **try**, pošto se proverava tokom prevođenja i stoga neće generisati nikakve izuzetke, za razliku od metode **Class.forName( )**.

U biblioteci **podaciotipu.ljubimci** sada imamo dve realizacije klase **PravljenjeLjubimaca**. Da bismo drugu od njih proglasili podrazumevanom, možemo napraviti projektni obrazac *Façade* (Fasada) koji upotrebljava program **RucnoPravljenjeLjubimaca**:

```
//: podaciotipu/ljubimci/Ljubimci.java
// Fasada za pravljenje podrazumevane realizacije klase
// PravljenjeLjubimaca.
package podaciotipu.ljubimci;
import java.util.*;

public class Ljubimci {
  public static final PravljenjeLjubimaca pravljenje =
    new RucnoPravljenjeLjubimaca();
  public static Ljubimac nekiLjubimac() {
    return pravljenje.nekiLjubimac();
  }
```

```
  public static Ljubimac[ ] createArray(int velicina) {
    return pravljenje.createArray(velicina);
  }
  public static ArrayList<Ljubimac> arrayList(int velicina) {
    return pravljenje.arrayList(velicina);
  }
} ///:~
```

Time je napravljena i indirekcija za metode **nekiLjubimac( )**, **createArray( )** i **arrayList( )**.

Pošto metoda **PrebrojavanjeLjubimaca.prebrojLjubimce( )** prima argument **PravljenjeLjubimaca**, lako nam je da ispitamo program **RucnoPravljenjeLjubimaca** (preko prethodne Fasade):

```
//: podaciotipu/PrebrojavanjeLjubimaca2.java
import podaciotipu.ljubimci.*;

public class PrebrojavanjeLjubimaca2 {
  public static void main(String[ ] args) {
    PrebrojavanjeLjubimaca.prebrojLjubimce(Ljubimci.pravljenje);
  }
} /* (Pokrenite da biste videli rezultate) *///:~
```

Rezultati su isti kao oni programa **PrebrojavanjeLjubimaca.java**.

Dinamički **instanceof**

Metoda **Class.isInstance** nudi način za dinamičko ispitivanje tipa objekta. Tako se sve dosadne naredbe **instanceof** mogu ukloniti iz primera **PrebrojavanjeLjubimaca**:

```
//: podaciotipu/PrebrojavanjeLjubimaca3.java
// Korišćenje metode isInstance().
import podaciotipu.ljubimci.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PrebrojavanjeLjubimaca3 {
  static class BrojacLjubimaca
  extends LinkedHashMap<Class<? extends Ljubimac>,Integer> {
    public BrojacLjubimaca() {
      super(MapData.map(RucnoPravljenjeLjubimaca.sviTipovi, 0));
    }
    public void prebroj(Ljubimac ljubimac) {
      // Class.isInstance() uklanja naredbe instanceof:
      for(Map.Entry<Class<? extends Ljubimac>,Integer> par
          : entrySet())
        if(par.getKey().isInstance(ljubimac))
          put(par.getKey(), par.getValue() + 1);
    }
```

```
    public String toString() {
      StringBuilder rezultat = new StringBuilder("{");
      for(Map.Entry<Class<? extends Ljubimac>,Integer> par
          : entrySet()) {
        rezultat.append(par.getKey().getSimpleName());
        rezultat.append("=");
        rezultat.append(par.getValue());
        rezultat.append(", ");
      }
      rezultat.delete(rezultat.length()-2, rezultat.length());
      rezultat.append("}");
      return rezultat.toString();
    }
  }
  public static void main(String[] args) {
    BrojacLjubimaca brojLjubimaca = new BrojacLjubimaca();
    for(Ljubimac ljubimac : Ljubimci.createArray(20)) {
      printnb(ljubimac.getClass().getSimpleName() + " ");
      brojLjubimaca.prebroj(ljubimac);
    }
    print();
    print(brojLjubimaca);
  }
} /* Ispis:
Pacov Manska Velska Mesanac Mops Velska Mops Manska Velska Pacov Egipatska
Hrcak Egipatska Mesanac Mesanac Velska Mis Mops Mis Velska
{Ljubimac=20, Pas=6, Macka=9, Glodar=5, Mesanac=3, Mops=3, Egipatska=2, Man-
ska=7, Velska=5, Pacov=2, Mis=2, Hrcak=1}
*///:~
```

Da bi se prebrojali svi različiti tipovi primeraka klase **Ljubimac**, **Map**a **BrojacLjubimaca** se unapred popunjava tipovima iz liste **RucnoPravljenjeLjubimaca.sviTipovi**. Ta lista upotrebljava klasu **net.mindview.util.MapData** koja prima objekat tipa **Iterable** (listu **sviTipovi**) i konstantu (u ovom slučaju, nulu), i popunjava **Map**u ključevima uzetim iz liste **sviTipovi** i vrednošću nula. Da **Map**u nismo popunili unapred, prebrojali bismo samo nasumično generisane tipove, ali ne i osnovne tipove kao što su **Ljubimac** i **Macka**.

Vidite da zbog korišćenja metode **isInstance**() nisu potrebni izrazi **instanceof**. Osim toga, to znači i da nove tipove kućnih ljubimaca možete da dodajete jednostavnim menjanjem niza **tipovi**; ostatak programa ne mora da se menja (što nije bio slučaj pri korišćenju izraza **instanceof**).

Metodu **toString**( ) preklopili smo da bi davala čitljiviji izlaz koji i dalje odgovara tipičnom izlazu koji se dobija ispisivanjem **Map**e.

Rekurzivno brojanje

Mapa u **PrebrojavanjeLjubimaca3.BrojacLjubimaca** bila je unapred popunjena svim različitim klasama **Ljubimac**. Umesto da mapu popunjavamo unapred, možemo upotrebiti metodu **Class.isAssignableFrom**( ) i napraviti alatku opšte namene koja ume da broji sve, a ne samo primerke klase **Ljubimci**:

```
//: net/mindview/util/BrojacTipova.java
// Broji instance date porodice tipova.
package net.mindview.util;
import java.util.*;

public class BrojacTipova extends HashMap<Class<?>,Integer>{
  private Class<?> osnovniTip;
  public BrojacTipova(Class<?> osnovniTip) {
    this.osnovniTip = osnovniTip;
  }
  public void prebroj(Object obj) {
    Class<?> tip = obj.getClass();
    if(!osnovniTip.isAssignableFrom(tip))
      throw new RuntimeException(obj + " netačan tip: "
        + tip + ", trebalo bi da bude tip ili podtip od "
        + osnovniTip);
    countClass(tip);
  }
  private void countClass(Class<?> tip) {
    Integer kolicina = get(tip);
    put(tip, kolicina == null ? 1 : kolicina + 1);
    Class<?> natKlasa = tip.getSuperclass();
    if(nadKlasa != null &&
       osnovniTip.isAssignableFrom(nadKlasa))
      countClass(natKlasa);
  }
  public String toString() {
    StringBuilder rezultat = new StringBuilder("{");
    for(Map.Entry<Class<?>,Integer> par : entrySet()) {
      rezultat.append(par.getKey().getSimpleName());
      rezultat.append("=");
      rezultat.append(par.getValue());
      rezultat.append(", ");
    }
    rezultat.delete(rezultat.length()-2, rezultat.length());
    rezultat.append("}");
    return rezultat.toString();
  }
} ///:~
```

Metoda **prebroj( )** od svog argumenta dobija objekat tipa **Class**, i upotrebljava metodu **isAssignableFrom( )** za proveru tokom izvršavanja da li objekat koji ste prosledili zaista pripada hijerarhiji o kojoj je reč. Metoda **countClass( )** najpre broji tačan tip te klase. Zatim, ako je **osnovniTip** dodeljiv (engl. *assignable*) iz natklase, **countClass( )** se poziva rekurzivno na natklasu.

```
//: podaciotipu/PrebrojavanjeLjubimaca4.java
import podaciotipu.ljubimci.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class PrebrojavanjeLjubimaca4 {
  public static void main(String[] args) {
    BrojacTipova brojac = new BrojacTipova(Ljubimac.class);
    for(Ljubimac ljubimac : Ljubimci.createArray(20)) {
      printnb(ljubimac.getClass().getSimpleName() + " ");
      brojac.prebroj(ljubimac);
    }
    print();
    print(brojac);
  }
} /* Ispis: (Sample)
Pacov Manska Velska Mesanac Mops Velska Mops Manska Velska Pacov Egipatska
Hrcak Egipatska Mesanac Mesanac Velska Mis Mops Mis Velska
{Mis=2, Pas=6, Manska=7, Egipatska=2, Glodar=5, Mops=3, Mesanac=3, Velska=5,
Macka=9, Hrcak=1, Ljubimac=20, Pacov=2}
*///:~
```

Kao što vidite iz rezultata, broje se i osnovni tipovi i tačni tipovi.

Vežba 11: (2) Biblioteci **podaciotipu.ljubimci** dodajte klasu **MorskoPrase** i prepravite sve primere u ovom poglavlju tako da se prilagode novoj klasi.

Vežba 12: (3) Upotrebite **BrojacTipova** s klasom **GeneratorKafe.java** iz poglavlja *Generički tipovi.*

Vežba 13: (3) Upotrebite **BrojacTipova** iz primera **RegisteredFactories.java** u ovom poglavlju.

Registrovane proizvodne metode

Ako generišete objekte hijerarhije **Ljubimci**, morate se setiti da sve buduće tipove te hijerarhije dodate i u klasu **rucnoPravljenjeLjubimaca.java**. Ukoliko često pravite nove klase, to može predstavljati problem.

Možda ste pomislili da potklasama treba dodati statični inicijalizator koji upisuje klasu u neku listu. Nažalost, statični inicijalizatori se pozivaju prilikom učitavanja klase, pa se javlja problem kokoške i jajeta: kada klasa nije u listi poznatih tipova, rucnoPravljenjeLjubimaca ne može da pravi objekte te klase, a zbog toga klasa neće ni biti učitana ni stavljena na listu.

U suštini, takvu listu morate sami da napravite, i to ručno (sem ukoliko napišete alatku koja analizira izvorni kod). Stoga je korisno da lista bude na centralnom, vidljivom mestu. Najbolje je da tu listu stavite u osnovnu klasu hijerarhije.

Napravićemo još jednu izmenu – prebacićemo pravljenje objekta u klase, za šta ćemo upotrebiti projektni obrazac *Factory Method* (Proizvodna metoda). Proizvodna metoda pravi objekat odgovarajućeg tipa, a može se pozivati polimorfno. Evo jednostavnog primera u kome se koristi proizvodna metoda **create( )** interfejsa **Proizvodjac**:

```
//: podaciotipu/proizvodjac/Proizvodjac.java
package podaciotipu.proizvodjac;
public interface Proizvodjac<T> { T create(); } ///:~
```

Generički parametar T znači da metoda **create( )** može da vraća različit tip rezultata za razne realizacije interfejsa **Proizvodjac**.

U ovom primeru, osnovna klasa **Deo** sadrži listu objekata tipa Proizvodjac. Dodavanjem u listu **proizvodjaciDelova**, u osnovnoj klasi se „registruju" proizvodni interfejsi za tipove koje će praviti metoda **createRandom( )**:

```
//: podaciotipu/RegistrovaniProizvodjaci.java
// Registrovanje proizvodnih klasa u osnovnoj klasi.
import podaciotipu.proizvodjac.*;
import java.util.*;

class Deo {
  public String toString() {
    return getClass().getSimpleName();
  }
  static List<Proizvodjac<? extends Deo>> proizvodjaciDelova =
    new ArrayList<Proizvodjac<? extends Deo>>();
  static {
    // Collections.addAll() daje upozorenje "unchecked generic
    // array creation ... for varargs parameter"
    // ("neprovereno pravljenje generičkog niza ...
    // za parametar promenljive dužine").
    proizvodjaciDelova.add(new PrecistacGoriva.Proizvodjac());
    proizvodjaciDelova.add(new PrecistacVazduha.Proizvodjac());
    proizvodjaciDelova.add(new PrecistacVazduhaZaKabinu.Proizvodjac());
    proizvodjaciDelova.add(new PrecistacUlja.Proizvodjac());
    proizvodjaciDelova.add(new KaisVentilatora.Proizvodjac());
    proizvodjaciDelova.add(new KaisServoVolana.Proizvodjac());
    proizvodjaciDelova.add(new KaisGeneratora.Proizvodjac());
  }
  private static Random slucajan = new Random(47);
  public static Deo createRandom() {
    int n = slucajan.nextInt(proizvodjaciDelova.size());
    return proizvodjaciDelova.get(n).create();
  }
}

class Filtar extends Deo {}

class PrecistacGoriva extends Filtar {
  // Pravljenje proizvodne klase za svaki konkretni tip:
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<PrecistacGoriva> {
    public PrecistacGoriva create() { return new PrecistacGoriva(); }
  }
}

class PrecistacVazduha extends Filtar {
  public static class Proizvodjac
```

```
    implements podaciotipu.proizvodjac.Proizvodjac<PrecistacVazduha> {
      public PrecistacVazduha create() { return new PrecistacVazduha(); }
    }
}

class PrecistacVazduhaZaKabinu extends Filtar {
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<PrecistacVazduhaZa
     Kabinu> {
    public PrecistacVazduhaZaKabinu create() {
      return new PrecistacVazduhaZaKabinu();
    }
  }
}

class PrecistacUlja extends Filtar {
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<PrecistacUlja> {
    public PrecistacUlja create() { return new PrecistacUlja(); }
  }
}

class Kais extends Deo {}

class KaisVentilatora extends Kais {
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<KaisVentilatora> {
    public KaisVentilatora create() { return new KaisVentilatora(); }
  }
}

class KaisGeneratora extends Kais {
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<KaisGeneratora> {
    public KaisGeneratora create() {
      return new KaisGeneratora();
    }
  }
}

class KaisServoVolana extends Kais {
  public static class Proizvodjac
  implements podaciotipu.proizvodjac.Proizvodjac<KaisServoVolana> {
    public KaisServoVolana create() {
      return new KaisServoVolana();
    }
  }
}
public class RegistrovaniProizvodjaci{
  public static void main(String[] args) {
```

```
    for(int i = 0; i < 10; i++)
      System.out.println(Deo.createRandom());
  }
} /* Ispis:
KaisGeneratora
PrecistacVazduhaZaKabinu
KaisGeneratora
PrecistacVazduha
KaisServoVolana
PrecistacVazduhaZaKabinu
PrecistacGoriva
KaisServoVolana
KaisServoVolana
PrecistacGoriva
*///:~
```

Neke klase iz hijerarhije nisu u listi: **Filtar** i **Kais** su samo klasifikatori, pa nikad ne pravimo objekte tih klasa, nego samo njihovih potklasa. Klase koje metoda **createRandom( )** *treba* da uzme u obzir sadrže unutrašnju klasu **Proizvodjac**. Kao što vidite, osnovni interfejs **Proizvodjac** koristi se tako što se navede potpuno ime **podaciotipu.proizvodjac.Proizvodjac** i tako se izbegava dvosmislenost.

Prilikom pravljenja liste objekata nismo koristili metodu **Collections.addAll( )**, zato što bi to izazvalo grešku „generic array creation" (pravljenje generičkog niza, što će biti objašnjeno u poglavlju *Generički tipovi*). Umesto nje, koristio sam metodu **add( )**. Metoda **createRandom( )** nasumično bira objekat iz liste **proizvodjaciDelova** i poziva njegovu metodu **create( )** da napravi nov **Deo**.

Vežba 14: (4) I konstruktor je svojevrsna proizvodna metoda. Prepravite program **RegistrovaniProizvodjaci.java** tako da se umesto eksplicitnog navođenja imena proizvođača, objekat klase smešta u listu, i svaki se objekat pravi metodom **newInstance( )**.

Vežba 15: (4) Realizujte novu klasu **PravljenjeLjubimaca** koristeći Registrovane Proizvodjace i prepravite Fasadu **Ljubimci** tako da upotrebljava nju, a ne druge dve. Postarajte se da ostali primeri u kojima se upotrebljava program **Ljubimci.java** i dalje rade ispravno.

Vežba 16: (4) Izmenite hijerarhiju **Kafa** u poglavlju *Generički tipovi* tako da se koriste Registrovani Proizvodjaci.

Poređenje **instanceof** sa ekvivalencijama klase

Kada tražite informacije o tipu, postoji važna razlika između oba oblika naredbe **instanceof** (tj. **instanceof** ili **isInstance**(), koje daju istovetne rezultate) i direktnog poređenja objekata tipa **Class**. Evo primera koji ilustruje razliku:

```
//: podaciotipu/SrodnostProtivTacnogTipa.java
// Razlika između instanceof i ekvivalencije
package podaciotipu;
import static net.mindview.util.Print.*;
```

```
class Osnovna {}
class Izvedena extends Osnovna {}

public class SrodnostProtivTacnogTipa {
  static void test(Object x) {
   print("Testiranje x tipa " +  x.getClass());
    print("x instanceof Osnovna " +  (x instanceof Osnovna));
    print("x instanceof Izvedena " +  (x instanceof Izvedena));
    print("Osnovna.isInstance(x) " +  Osnovna.class.isInstance(x));
    print("Izvedena.isInstance(x) " +  Izvedena.class.isInstance(x));
    print("x.getClass() == Osnovna.class " +
      (x.getClass() == Osnovna.class));
    print("x.getClass() == Izvedena.class " +
      (x.getClass() == Izvedena.class));
    print("x.getClass().equals(Osnovna.class)) " +
      (x.getClass().equals(Osnovna.class)));
    print("x.getClass().equals(Izvedena.class)) " +
      (x.getClass().equals(Izvedena.class)));
  }
  public static void main(String[ ] args) {
    test(new Osnovna());
    test(new Izvedena());
  }
} /* Ispis:
Testiranje x tipa class Osnovna
x instanceof Osnovna true
x instanceof Izvedena false
Osnovna.isInstance(x) true
Izvedena.isInstance(x) false
x.getClass() == Osnovna.class true
x.getClass() == Izvedena.class false
x.getClass().equals(Osnovna.class)) true
x.getClass().equals(Izvedena.class)) false
Testiranje x tipa class Izvedena
x instanceof Osnovna true
x instanceof Izvedena true
Osnovna.isInstance(x) true
Izvedena.isInstance(x) true
x.getClass() == Osnovna.class false
x.getClass() == Izvedena.class true
x.getClass().equals(Osnovna.class)) false
x.getClass().equals(Izvedena.class) true
*///:~
```

Metoda **test**() proverava tip, pri čemu njen argument koristi oba oblika **instanceof**. Potom čita referencu na **Class** i koristi = = i **equals**() za ispitivanje jednakosti objekata tipa **Class**. Kao što je i očekivano, **instanceof** i **isInstance**() daju istovetne rezultate, isto kao i **equals**() i ==. Međutim, sami testovi dovode do različitih zaključaka. Saglasno s pojmom tipa, **instanceof** pita: „Da li je ovo tvoja klasa, ili klasa izvedena iz nje?" S druge strane, ako stvarne objekte tipa **Class** uporedite pomoću operatora = =, tu nasleđivanje ne igra nikakvu ulogu: objekti su ili istovetnog tipa ili nisu.

Refleksija: informacije o klasi u vreme izvršavanja

Ako ne znate kog je tipa objekat, saznaćete iz prepoznavanja tipa tokom izvršavanja. Međutim, postoji jedno ograničenje: da biste tip mogli da otkrijete tokom izvršavanja i da biste uradili nešto korisno s tom informacijom, on mora biti poznat u vreme prevođenja. Drugim rečima, prevodilac mora da zna za sve klase s kojima radite kako bi prepoznao tip tokom izvršavanja.

Na prvi pogled to ne deluje kao veliko ograničenje, ali pretpostavimo da ste dobili referencu na objekat koji ne pripada prostoru vašeg programa. Zapravo, klasa objekta čak nije ni dostupna programu tokom prevođenja. Na primer, zamislimo da ste dobili gomilu bajtova iz datoteke s diska ili ste ih preuzeli iz mreže, a rečeno vam je da oni predstavljaju klasu. Pošto prevodilac ne može da zna za tu klasu dok prevodi kôd, kako uopšte možete da je koristite?

U tradicionalnom programerskom okruženju ovo izgleda kao malo verovatno. Međutim, kako širimo vidike programiranja, srećemo važne slučajeve u kojima se to dešava. Prvi slučaj je programiranje komponenata, gde se projekti prave u okruženju za brzo razvijanje aplikacija (engl. *Rapid Application Development, RAD*), pomoću alatke za pravljenje aplikacija u *integrisanom razvojnom okruženju* (engl. *Integrated Development Environment, IDE*). To je vizuelni pristup pravljenju programa (koji se na ekranu prikazuje kao „obrazac"), a sastoji se od premeštanja po obrascu ikonica koje predstavljaju komponente. Te komponente se zatim konfigurišu podešavanjem nekih parametara u vreme programiranja. Pri konfigurisanju tokom projektovanja, mora biti omogućeno da se svim komponentama dodeljuju instance, da svaka može prikazivati svoje delove i da dozvoljava čitanje i postavljanje svojih vrednosti. Osim toga, komponente koje obrađuju grafičke događaje moraju da prikazuju informacije o odgovarajućim metodama kako bi razvojno okruženje pomoglo programeru u redefinisanju metoda za obradu događaja. Refleksija obezbeđuje mehanizam za otkrivanje postojećih metoda u klasi i daje imena tih metoda. Java obezbeđuje strukturu za programiranje u komponentama pomoću tehnologije zrna Jave (engl. *Java Beans*) koja je opisana u poglavlju *Grafička korisnička okruženja*.

Još jedan značajan motiv za otkrivanje informacija o klasi u vreme izvršavanja jeste omogućavanje pravljenja i pokretanja objekata na udaljenim platformama u mreži. To se zove *daljinsko pozivanje metoda* (engl. *Remote Method Invocation, RMI*) a omogućuje Javinom programu da koristi objekte koji su raštrkani na raznim računarima. Raštrkanost objekata je opravdana: na primer, možda izvršavate zadatak s mnogo izračunavanja, pa želite da ga podelite i pošaljete delove besposlenim računarima kako biste ubrzali izvršavanje. Ponekad ćete hteti da određenom računaru pošaljete kôd koji obrađuje određene vrste zadataka (npr. „poslovna pravila" u višeslojnoj kljent/server arhitekturi), pa taj računar postaje zajednička ostava koja opisuje akcije i može se lako prilagoditi tako da utiče na sve u sistemu. (To je zanimljiv napredak, pošto je svrha računara isključivo da olakšava promene softvera!) Uzgred, distribuirana obrada podržava i specijalizovan hardver koji možda odgovara određenom zadatku, npr. inverziji matrice, ali nije pogodan ili je preskup za programiranje opšte namene.

Klasa **Class**, opisana ranije u poglavlju, podržava pojam *refleksije*, a postoji i dodatna biblioteka **java.lang.reflect** s klasama **Field**, **Method** i **Constructor** (pri čemu svaka realizuje interfejs **Member**). Objekte tih tipova pravi Javina virtuelna mašina tokom izvršavanja, da bi predstavila odgovarajućeg člana nepoznate klase. Zatim možete koristiti konstruktore da biste pravili nove objekte, metode **get**() i **set**() za čitanje i menjanje polja povezanih sa objektima tipa **Field** i metodu **invoke**() za pozivanje metode povezane s objektom tipa **Method**. Pored toga, možete da pozovete metode **getFields**(), **getMethods**(), **getConstructors**() itd. da biste dobili nizove objekata koji predstavljaju polja, metode i konstruktore. (Više o ovome naći ćete u dokumentaciji o klasi **Class** u Javinom razvojnom okruženju, JDK.) Znači, informacije o klasi anonimnih objekata mogu se u potpunosti otkriti tokom izvršavanja, a ništa ne mora da se zna tokom prevođenja.

Važno je shvatiti da refleksija nije ništa posebno. Kada koristite refleksiju za rad sa objektima nepoznatog tipa, Javina virtuelna mašina će proveriti da li neki objekat pripada određenoj klasi (kao pri običnom postupku prepoznavanja tipa u vreme izvršavanja), ali zatim, pre nego što uradi bilo šta drugo, mora da učita objekat tipa **Class**. Stoga datoteka **.class** za taj tip i dalje mora da bude dostupna Javinoj virtuelnoj mašini, bilo na lokalnom sistemu ili u mreži. Prema tome, prava razlika između prepoznavanja tipa u vreme izvršavanja i refleksije leži u tome što u prvom načinu prevodilac otvara i ispituje datoteku **.class** tokom prevođenja. Drugim rečima, možete da pozovete sve metode objekta na uobičajen način. Pri refleksiji, datoteka **.class** nije dostupna tokom prevođenja; nju otvara i ispituje izvršno okruženje.

Čitanje metoda klase

Retko će biti potrebno da direktno koristite refleksivne alatke; one postoje u jeziku da bi podržavale pravljenje dinamičnijeg koda. Refleksija je stavljena u jezik da bi podržala druge Javine mogućnosti kao što su serijalizacija objekata i zrna Jave (obrađena u nastavku knjige). Međutim, postoje prilike kada je mogućnost dinamičkog izdvajanja informacija o klasi korisna.

Jedna izuzetno korisna alatka je čitač metoda klase. Izvorni kôd s definicijom klase ili dokumentacija na Webu prikazuju samo metode koje su definisane ili redefinisane *unutar definicije te klase*. Ko zna koliko još dostupnih metoda postoji u osnovnim klasama. Njihovo pronalaženje je zamorno i oduzima mnogo vremena.[1] Srećom, refleksija obezbeđuje način za pravljenje jednostavne alatke koja automatski prikazuje ceo interfejs. Evo kako ona radi:

```
//: podaciotipu/PrikaziMetode.java
// Primena refleksije za prikazivanje svih metoda klase,
// čak i kada su definisane u osnovnoj klasi.
// {Args: PrikaziMetode}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;
```

[1] Naročito ranije. Međutim, kompanija Sun je prilično poboljšala HTML dokumentaciju za Javu, pa se metode osnovne klase lakše pronalaze.

```
public class PrikaziMetode{
  private static String upotreba =
    "upotreba: \n" +
    "PrikaziMetode puno.ime.klase\n" +
    "Da bi se prikazale sve metode klase ili: \n" +
    "PrikaziMetode puno.ime.klase reč\n" +
    "Da bi se pronasle metode na osnovu 'reci'";
  private static Pattern uzorak = Pattern.compile("\\w+\\.");
  public static void main(String[ ] args) {
    if(args.length < 1) {
      print(upotreba);
      System.exit(0);
    }
     int redovi = 0;
    try {
      Class<?> c = Class.forName(args[0]);
      Method[ ] metode = c.getMethods();
      Constructor[ ] ctors = c.getConstructors();
      if(args.length == 1) {
        for (Method metoda : metode)
          print(
            p.matcher(metoda.toString()).replaceAll(""));
        for(Constructor ctor : ctors)
          print(p.matcher(ctor.toString()).replaceAll(""));
        redovi = metode.length + ctors.length;
      } else {
        for(Method metoda : methoda)
          if(metoda.toString().indexOf(args[1]) != -1) {
            print(
              p.matcher(metoda.toString()).replaceAll(""));
            redovi++;
          }
        for(Constructor ctor : ctors)
          if(ctor.toString().indexOf(args[1]) != -1) {
            print(p.matcher(
              ctor.toString()).replaceAll(""));
            redovi++;
          }
      }
    } catch(ClassNotFoundException e) {
      print("Ne postoji takva klasa: " + e);
    }
  }
} /* Ispis:
public static void main(String[])
public native int hashCode()
public final native Class getClass()
public final void wait(long,int) throws InterruptedException
public final void wait() throws InterruptedException
public final native void wait(long) throws InterruptedException
```

```
public boolean equals(Object)
public String toString()
public final native void notify()
public final native void notifyAll()
public PrikaziMetode()
*///:~
```

Metode **getMethods()** i **getConstructors()** klase **Class** vraćaju niz objekata tipa **Method** odnosno **Constructor**. Svaka od ovih klasa ima dalje metode za izdvajanje imena, argumenata i povratnih vrednosti metoda koje predstavljaju. Međutim, za dobijanje znakovnog niza s potpunim potpisom metode možete da koristite samo metodu **toString()**, kao što je ovde učinjeno. Ostatak koda služi samo za izdvajanje informacija s komandne linije, za utvrđivanje da li određeni potpis odgovara ciljnom znakovnom nizu (korišćenjem funkcije **indexOf()**) i za odbacivanje kvalifikatora imena pomoću regularnih izraza (koji su predstavljeni u poglavlju *Znakovni nizovi*).

Pošto rezultat koji daje **Class.forName()** ne može da bude poznat u vreme prevođenja, sve informacije o potpisu metode izdvajaju se tokom izvršavanja. Ako potražite objašnjenje refleksije u dokumentaciji na Webu, videćete da ima dovoljno podrške za stvarno podešavanje i pozivanje metode za objekat koji je potpuno nepoznat tokom prevođenja (primera za ovo biće u nastavku knjige). Iako ovo možda nikada nećete morati da radite sami, vrednost pune refleksije ume da iznenadi.

Gornje rezultate je proizvela komandna linija:

```
java PrikaziMetode PrikaziMetode
```

Ovo daje listing koji sadrži podrazumevani javni konstruktor, mada se u kodu vidi da nije definisan nikakav konstruktor. Konstruktor koji vidite automatski pravi prevodilac. Ako potom klasu **PrikaziMetode** pretvorite u nejavnu (dakle, s paketnim pristupom), napravljen podrazumevani konstruktor se više neće pojavljivati u rezultatu programa. Napravljenom podrazumevanom konstruktoru se automatski dodeljuje isti nivo pristupa kao klasi.

Bilo bi zanimljivo pozvati i **java PrikaziMetode java.lang.String** uz dodatni argument tipa **char**, **int**, **String** itd.

Ova alatka zaista može da uštedi vreme prilikom programiranja, kada ne možete da se setite da li klasa ima neku metodu, a ne želite da pregledate hijerarhiju klasa u dokumentaciji na Webu, ili ne znate da li ta klasa, primera radi, može da radi nešto sa objektima tipa **Color**.

Poglavlje *Grafička korisnička okruženja* sadrži grafičku verziju ovog programa (prilagođenu za izdvajanje informacija za Swing komponente), pa možete da je izvršavate dok pišete kôd da biste brzo pronašli ono što vam treba.

Vežba 17: (2) Izmenite regularan izraz u programu **PokaziMetode.java** tako da se uklanjaju i rezervisane reči **native** i **final**. Uputstvo: upotrebite operator | (OR).

Vežba 18: (1) Izmenite klasu **PokaziMetode** tako da ne bude javna i dokažite da se napravljen podrazumevani konstruktor više ne pojavljuje u rezultatu programa.

Vežba 19: (4) U programu **IspitivanjeIgracaka.java**, upotrebite refleksiju za pravljenje objekta tipa **Igracka** pomoću nepodrazumevanog konstruktora.

Vežba 20: (5) U dokumentaciji JDK na adresi *http://java.sun.com* potražite interfejs za **java.lang.Class**. Napišite program koji ime klase prima kao argument komandne linije, zatim **Class** metodama ispisuje sve informacije dostupne za tu klasu. Ispitajte program pomoću neke klase iz standardne biblioteke i neke klase koju ste sami napravili.

Dinamički posrednici

Projektni obrazac *Proxy* (Posrednik) jedan je od osnovnih projektnih obrazaca. To je objekat koji umećete umesto „pravog" objekta da biste obavili neke dopunske ili različite operacije – među njima je obično i komunikacija sa određenim „pravim" objektom. Strukturu projektnog obrasca *Proxy* pokazaćemo na jednom trivijalnom primeru:

```
//: podaciotipu/ProstPrimerZaPosrednika.java
import static net.mindview.util.Print.*;

interface Interfejs {
  void uradiNesto();
  void nestoDrugo(String arg);
}

class PraviObjekat implements Interfejs {
  public void uradiNesto() { print("uradiNesto"); }
  public void nestoDrugo(String arg) {
    print("nestoDrugo " + arg);
  }
}

class JednostavanPosrednik implements Interfejs {
  private Interfejs imaposrednika;
  public JednostavanPosrednik(Interfejs imaposrednika) {
    this.imaposrednika = imaposrednika;
  }
  public void uradiNesto() {
    print("JednostavanPosrednik uradiNesto");
    imaposrednika.uradiNesto();
  }
  public void nestoDrugo(String arg) {
    print("JednostavanPosrednik nestoDrugo " + arg);
    imaposrednika.nestoDrugo(arg);
  }
}

class ProstPrimerZaPosrednika {
  public static void potrosac(Interfejs ifejs) {
    ifejs.uradiNesto();
    ifejs.nestoDrugo("bonobo");
  }
```

```
  public static void main(String[] args) {
    potrosac(new PraviObjekat());
    potrosac(new JednostavanPosrednik(new PraviObjekat()));
  }
} /* Ispis:
uradiNesto
nestoDrugo bonobo
JednostavanPosrednik uradiNesto
uradiNesto
JednostavanPosrednik nestoDrugo bonobo
nestoDrugo bonobo
*///:~
```

Pošto **potrosac( )** prima jedan **Interfejs**, on ne može znati da li je dobio **PraviObjekat** ili objekat tipa **JednostavanPosrednik**, jer oba realizuju **Interfejs**. Ali **JednostavanPosrednik**, koji je umetnut između klijenta i objekta **PraviObjekat** obavlja operacije i zatim poziva identičnu metodu za **PraviObjekat**.

Projektni obrazac *Proxy* je podesan kad god neke dodatne operacije treba izdvojiti od „pravog objekta", i naročito kada hoćete da olakšate prelaz od nekorišćenja tih dodatnih operacija na korišćenje i obrnuto (svrha projektnog obrasca jeste kapsuliranje promena – da biste opravdali upotrebu obrasca, nešto morate da menjate). Na primer, kako biste pratili pozive metoda u objektu **PraviObjekat** ili merili režijske troškove tih poziva? Takvom kodu nije mesto u gotovoj aplikaciji, pa *Proxy* omogućuje da ga lako dodate i uklonite.

Javin *dinamički posrednik* (engl. *dynamic proxy*) podiže ideju upotrebe posrednika na viši nivo, pošto objekat posrednika pravi dinamički i isto tako obrađuje pozive posredovanih metoda. Svi pozivi dinamičkog posrednika bivaju preusmereni na jedini *blok za pozive* (engl. *invocation handler*) koji prepoznaje poziv i odlučuje šta s njim da radi. Preradio sam **ProstPrimerZaPosrednika.java** tako da posrednik bude dinamički:

```
//: podaciotipu/ProstDinamickiPosrednik.java
import java.lang.reflect.*;

class BlokDinamickogPosrednika implements InvocationHandler {
  private Object imaposrednika;
  public BlokDinamickogPosrednika(Object imaposrednika) {
    this.imaposrednika = imaposrednika;
  }
  public Object
  invoke(Object posrednik, Method metoda, Object[] argumenti)
  throws Throwable {
    System.out.println("**** posrednik: " + posrednik.getClass() +
      ", metoda: " + metoda + ", argumenti: " + argumenti);
    if(argumenti != null)
      for(Object arg : argumenti)
        System.out.println("  " + arg);
    return metoda.invoke(imaposrednika, argumenti);
  }
}
```

```
class ProstDinamickiPosrednik {
  public static void potrosac(Interfejs ifejs) {
    ifejs.uradiNesto();
    ifejs.nestoDrugo("bonobo");
  }
  public static void main(String[] args) {
    PraviObjekat pravi = new PraviObjekat();
    potrosac(pravi);
    // Umetni posrednika i pozovi ponovo:
    Interfejs posrednik = (Interfejs)Proxy.newProxyInstance(
      Interfejs.class.getClassLoader(),
      new Class[]{ Interfejs.class },
      new BlokDinamickogPosrednika(pravi));
    potrosac(posrednik);
  }
} /* Ispis: (95% podudaranja)
uradiNesto
nestoDrugo bonobo
**** posrednik: class $Proxy0, metoda: public abstract void
Interfejs.uradiNesto(), argumenti: null
uradiNesto
**** posrednik: class $Proxy0, metoda: public abstract void
Interfejs.nestoDrugo(java.lang.String), argumenti:
[ Ljava.lang.Object;@42e816
  bonobo
nestoDrugo bonobo
*///:~
```

Dinamičkog posrednika pravite pozivom statične metode **Proxy.newProxy-Instance( )** koja zahteva učitavač klase (po pravilu, možete joj dati učitavač klase nekog već učitanog objekta), listu interfejsa (ne klasa niti apstraktnih klasa) koje posrednik treba da realizuje i realizaciju interfejsa **InvocationHandler**. Dinamički posrednik preusmerava sve pozive bloku za pozive, pa konstruktor bloka za pozive obično dobija referencu na „pravi" objekat, da bi mogao da prosleđuje zahteve nakon što obavi svoj posrednički posao.

Metoda **invoke( )** dobija objekat posrednika, za slučaj da hoćete da saznate odakle je poziv došao – što je retko potrebno. Međutim, budite oprezni kada iz metode **invoke( )** pozivate metode posrednika, pošto se pozivi kroz interfejs preusmeravaju kroz posrednik.

Po pravilu, obavićete posredovanu operaciju i zatim upotrebiti **Method.invoke( )** za prosleđivanje zahteva objektu koji ima posrednika, prosleđujući mu potrebne argumente. Možda će vam to isprva izgledati ograničavajuće – činiće se da možete obavljati samo generičke operacije. Međutim, pozive određenih metoda možete izdvojiti filtrom, dok ćete druge metode samo proslediti:

```
//: podaciotipu/IzborMetoda.java
// Traženje određenih metoda u dinamičkom posredniku.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;
```

```
class SelektorMetoda implements InvocationHandler {
  private Object imaposrednika;
  public SelektorMetoda(Object imaposrednika) {
    this.imaposrednika = imaposrednika;
  }
  public Object
  invoke(Object posrednik, Method metoda, Object[] argumenti)
  throws Throwable {
    if(metoda.getName().equals("zanimljivo"))
      print("Posrednik je otkrio zanimljivu metodu");
    return metoda.invoke(imaposrednika, argumenti);
  }
}

interface NekeMetode {
  void dosadna1();
  void dosadna2();
  void zanimljiva(String arg);
  void zanimljiva3();
}

class Realizacija implements NekeMetode {
  public void dosadna1() { print("dosadna1"); }
  public void dosadna2() { print("dosadna2"); }
  public void zanimljiva(String arg) {
    print("zanimljiva " + arg);
  }
  public void dosadna3() { print("dosadna3"); }
}

class IzborMetoda {
  public static void main(String[] args) {
    NekeMetode posrednik= (NekeMetode)Proxy.newProxyInstance(
      NekeMetode.class.getClassLoader(),
      new Class[]{ NekeMetode.class },
      new IzborMetoda(new Realizacija()));
    posrednik.dosadna1();
    posrednik.dosadna2();
    posrednik.zanimljiva("bonobo");
    posrednik.dosadna3();
  }
} /* Ispis:
dosadna1
dosadna2
Posrednik je otkrio zanimljivu metodu
zanimljiva bonobo
dosadna3
*///:~
```

Ovde su nas zanimala samo imena metoda, ali vi možete tražiti i druge delove potpisa metoda, pa čak i vrednosti određenih argumenata.

Dinamički posrednik nije alatka koju ćete upotrebljavati svakog dana, ali neke vrste problema njome možete veoma lepo da rešite. O projektnom obrascu *Proxy* i drugim projektnim obrascima više ćete saznati u knjigama *Thinking in Patterns* (posetite *www.MindView.net*) i *Design Patterns*, autor je Erich Gamma i dr. (Addison-Wesley, 1995).

Vežba 21: (3) Prepravite program **ProstPrimerZaPosrednik.java** tako da meri vremena poziva metoda.

Vežba 22: (3) Prepravite program **ProstDinamickiPosrednik.java** tako da meri vremena poziva metoda.

Vežba 23: (3) Unutar metode **invoke( )** u programu **ProstDinamickiPosrednik.java**, pokušajte da ispišete argument **posrednik** i objasnite šta se dešava.

Projekat:[2] Napišite sistem u kojem dinamički posrednici realizuju *transakcije*, gde posrednik obavlja *potvrđivanje* ako je posredovani poziv bio uspešan (nije generisao izuzetke), odnosno *poništavanje* ako je poziv zakazao. Potvrđivanje i poništavanje se obavljaju na spoljnoj tekstualnoj datoteci koja je izvan kontrole Javinih izuzetaka. Moraćete da pazite na *atomizovanost* (usitnjenost) operacija.

Null objekti

Kada ugrađenu **null** referencu upotrebljavate da biste naznačili kako određeni objekat ne postoji, svaki put kada dobijete neku referencu, morate proveriti da li je jednaka **null**. To ume da postane veoma zamorno i da proizvede zamoran kod. Problem je to što **null** referenca nema svoje ponašanje, sem što proizvodi **NullPointerException** ako išta pokušate s njom da uradite. Katkada je korisno uvesti *Null objekat*[3] koji prima poruke za objekat koji „zamenjuje", ali vraća vrednosti koje pokazuju da nema „pravog" objekta. Onda biste smeli pretpostaviti da su svi objekti validni pa ne biste gubili programsko vreme proveravajući reference na **null** (i na čitanje rezultujućeg koda).

Iako bi bilo zabavno zamišljati programski jezik koji automatski pravi Null objekte za vas, u praksi nema smisla upotrebljavati ih posvuda – ponekad baš treba proveriti da li referenca ukazuje na **null**, katkada se može razborito pretpostaviti da **null** referenca neće naići, dok ima i slučajeva kada je prihvatljivo otkrivanje grešaka preko izuzetka **NullPointerException**. Izgleda da su Null objekti najkorisniji kada su „bliže podacima", sa objektima koji predstavljaju entitete u prostoru problema. Jednostavan primer je to što mnogi sistemi imaju klasu **Osoba**, a u kodu ima situacija kada stvarna osoba ne postoji (ili postoji, ali vi još niste dobili sve podatke o njoj), pa bi se tada tradicionalno koristila **null** referenca i odgovarajuća provera. Umesto toga, možemo napraviti Null objekat. Iako

---

[2] Projekti su predlozi za semestralne radove (na primer). *Solution guide* ne sadrži rešenja projekata.

[3] Izumili su ga Bobby Woolf i Bruce Anderson. To se može smatrati specijalnim slučajem projektnog obrasca *Strategy* (Strategija). Varijanta *Null objekta* je obrazac *Null iterator* koji iteriranje čvorova u kompozitnoj hijerarhiji čini nevidljivim za klijenta (klijent istu logiku može da upotrebi za iteriranje kompozitnih i završnih čvorova).

Null objekat odgovara na sve poruke na koje bi odgovorio i pravi objekat, i đalje je potreban način ispitivanja jednakosti s **null**. Najjednostavniji način da se to uradi je istoimeni interfejs:

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} ///:~
```

Time bi se omogućilo da **instanceof** otkrije Null objekat, i što je još važnije, ne biste morali da dodajete metodu **isNull( )** svim svojim klasama (što bi, na kraju krajeva, samo predstavljalo drugačiji način prepoznavanja informacija tokom izvršavanja – pa zašto onda ne bismo koristili ugrađeni RTTI?).

```
//: podaciotipu/Osoba.java
// Klasa koja ima Null objekat.
import net.mindview.util.*;

class Osoba {
  public final String ime;
  public final String prezime;
  public final String adresa;
  // etc.
  public Osoba(String ime, String prezime, String adresa){
    this.ime = ime;
    this.prezime = prezime;
    this.adresa = adresa;
  }
  public String toString() {
    return "Osoba: " + ime + " " + prezime + " " + adresa;
  }
  public static class NullOsoba
  extends Osoba implements Null {
    private NullOsoba() { super("Nema", "Nema", "Nema"); }
    public String toString() { return "NullOsoba"; }
  }
  public static final Osoba NULL = new NullOsoba();
} ///:~
```

Po pravilu, Null objekat je projektni obrazac *Singleton* (Singlton), pa je ovde napravljen kao statična finalna instanca. To funkcioniše zato što je **Osoba** *nepromenljiva* – možete postavljati samo vrednosti u konstruktoru, i zatim ih čitati, ali ih ne možete menjati (pošto su objekti tipa **String** po svojoj prirodi nepromenljivi). Ukoliko hoćete da promenite objekat tipa **NullOsoba**, jedino možete da ga zamenite novim objektom tipa **Osoba**. Imajte u vidu da pomoću rezervisane reči **instanceof** i dalje možete otkrivati generički objekat **Null** ili specifičniji objekat tipa **NullOsoba**, ali singltonskim pristupom možete se ograničiti na korišćenje metode **equals( )** ili čak operatora == za poređenje sa **Osoba.NULL**.

Pretpostavimo da živite u vreme nastanka Interneta i da ste dobili mnogo kapitala za realizaciju svoje Izvanredne ideje. Spremni ste da primite osoblje, ali dok čekate da se radna mesta popune, možete upotrebiti Null objekte tipa **Osoba** da čuvaju mesto za svako **RadnoMesto**:

```
//: podaciotipu/RadnoMesto.java

class RadnoMesto {
  private String zvanje;
  private Osoba osoba;
  public RadnoMesto(String Funkcija, Osoba zaposleni) {
    zvanje = Funkcija;
    osoba = zaposleni;
    if(osoba == null)
      osoba = Osoba.NULL;
  }
  public RadnoMesto(String Funkcija) {
    zvanje = Funkcija;
    osoba = Osoba.NULL;
  }
  public String dajZvanje() { return zvanje; }
  public void zadajZvanje(String novoZvanje) {
    zvanje = novoZvanje
  }
  public Osoba dajOsobu() { return osoba; }
  public void zadajOsobu(Osoba novaOsoba) {
    osoba = novaOsoba;
    if(osoba == null)
      osoba = Osoba.NULL;
  }
  public String toString() {
    return "RadnoMesto: " + zvanje + " " + osoba;
  }
} ///:~
```

Uz klasu **RadnoMesto** ne moramo da pravimo Null objekat, pošto postojanje objekta tipa **Osoba.NULL** implicira postojanje praznog objekta tipa **RadnoMesto**. Moguće je da se kasnije mora eksplicitno dodati Null objekat za **RadnoMesto**, ali YAGNI[4] (*You Aren't Going to Need It*, Neće biti potrebno) uči da u prvoj verziji programa pokušate s „najjednostavnijim što bi moglo da funkcioniše" i čekate dok vas neki aspekt programa ne natera da dodate nešto novo; to je bolje nego da odmah pretpostavljate da će to biti neophodno.

Kada budete popunjavali radna mesta, klasa **Osoblje** će moći da traži Null objekte:

```
//: podaciotipu/Osoblje.java
import java.util.*;

public class Osoblje extends ArrayList<RadnoMesto> {
  public void add(String zvanje, Osoba osoba) {
```

[4] Princip *Ekstremnog programiranja* (XP), kao što je i „Napravi najjednostavniju moguću stvar koja radi."

```
    add(new RadnoMesto(zvanje, osoba));
  }
  public void add(String... zvanja) {
    for(String zvanje : zvanja)
      add(new RadnoMesto(zvanje));
  }
  public Osoblje(String... zvanja) { add(zvanja); }
  public boolean slobodnoRadnoMesto(String zvanje) {
    for(RadnoMesto radnomesto : this)
      if(radnomesto.dajZvanje().equals(zvanje) &&
         radnomesto.dajOsobu() == Osoba.NULL)
        return true;
    return false;
  }
  public void popuniRadnoMesto(String zvanje, Osoba zaposli) {
    for(RadnoMesto radnomesto : this)
      if(radnomesto.dajZvanje().equals(zvanje) &&
         radnomesto.dajOsobu() == Osoba.NULL) {
        radnomesto.zadajOsobu(zaposli);
        return;
      }
    throw new RuntimeException(
      "RadnoMesto " + zvanje + " nije slobodno");
  }
  public static void main(String[] args) {
    Osoblje osoblje = new Osoblje("Predsednik", "Tehnički direktor",
      "Direktor marketinga", "Direktor proizvodnje",
      "Vođa projekta", "Softveraš",
      "Softveraš", "Softveraš",
      "Softveraš", "Inženjer ispitivanja",
      "Pisac tehničkih tekstova");
    osoblje.popuniRadnoMesto("Predsednik",
      new Osoba("Ja", "Prezime", "Vrh"));
    osoblje.popuniRadnoMesto("Vođa projekta",
      new Osoba("Dženet", "Planer", "Spravice"));
    if(osoblje.slobodnoRadnoMesto("Softveraš"))
      osoblje.popuniRadnoMesto("Softveraš",
        new Osoba("Bob", "Koder", "Svetlograd"));
    System.out.println(osoblje);
  }
} /* Ispis:
[RadnoMesto: Predsednik Osoba: Ja Prezime Vrh, RadnoMesto: Tehnički direktor
NullOsoba, RadnoMesto: Direktor marketinga NullOsoba, RadnoMesto: Direktor
proizvodnje NullOsoba, RadnoMesto: Vođa projekta Osoba: Dženet Planer
Spravice, RadnoMesto: Softveraš Osoba: Bob Koder Svetlograd, RadnoMesto:
Softveraš NullOsoba, RadnoMesto: Softveraš NullOsoba, RadnoMesto: Softveraš
NullOsoba, RadnoMesto: Inženjer ispitivanja NullOsoba, RadnoMesto: Pisac
tehničkih tekstova NullOsoba]
*///:~
```

Vodite računa o tome da na nekim mestima ipak morate proveravati postoje li Null objekti, što se ne razlikuje baš mnogo od provere **null** vrednosti. Na drugim mestima – kao što su **toString( )** konverzije, u ovom slučaju – ne morate da obavljate dodatne provere; smete da pretpostavite kako su sve reference objekata validne.

Ako umesto s konkretnim klasama radite sa interfejsima, možete upotrebiti **DynamicProxy** za automatsko pravljenje Null objekata. Pretpostavimo da imamo interfejs **Robot** koji definiše ime, model i **List<Operacija>** koji opisuju šta **Robot** ume da radi. **Operacija** sadrži opis i komandu – to je jedna vrsta obrasca *Command* (Komanda):

```
//: podaciotipu/Operacija.java

public interface Operacija {
  String opis();
  void komanda();
} ///:~
```

Uslugama **Robot**a pristupate pozivanjem metode **operacije( )**:

```
//: podaciotipu/Robot.java
import java.util.*;
import net.mindview.util.*;

public interface Robot {
  String ime();
  String model();
  List<Operacija> operacije();
  class Test {
    public static void test(Robot r) {
      if(r instanceof Null)
        System.out.println("[Null Robot]");
      System.out.println("Robot ime: " + r.ime());
      System.out.println("Robot model: " + r.model());
      for(Operacija operacija : r.operacije()) {
        System.out.println(operacija.opis());
        operacija.komanda();
      }
    }
  }
} ///:~
```

U programu je i ugneždena klasa za testiranje.
Sada možemo da napravimo **Robot**a koji čisti sneg:

```
//: podaciotipu/RobotKojiCistiSneg.java
import java.util.*;

public class RobotKojiCistiSneg implements Robot {
  private String ime;
  public RobotKojiCistiSneg(String ime) { this.ime = ime;}
```

```
    public String ime() { return ime; }
    public String model() { return "SnegoBot Serija 11"; }
    public List<Operacija> operacije() {
      return Arrays.asList(
        new Operacija() {
          public String opis() {
            return ime + " može da čisti sneg";
          }
          public void komanda() {
            System.out.println(ime + " čisti sneg");
          }
        },
        new Operacija() {
          public String opis() {
            return ime + " može da čisti led";
          }
          public void komanda() {
            System.out.println(ime + " čisti led");
          }
        },
        new Operacija() {
          public String opis() {
            return ime + " može da počisti krov";
          }
          public void komanda() {
            System.out.println(ime + " počisti krov");
          }
        }
      );
    }
    public static void main(String[] args) {
      Robot.Test.test(new RobotKojiCistiSneg("SamoRadi"));
    }
  } /* Ispis:
  Robot ime: SamoRadi
  Robot model: SnegoBot Serija 11
  SamoRadi može da čisti sneg
  SamoRadi čisti sneg
  SamoRadi može da čisti led
  SamoRadi čisti led
  SamoRadi može da počisti krov
  SamoRadi počisti krov
  *///:~
```

Pretpostavljamo da će biti mnogo različitih tipova **Robot**a i hteli bismo da svaki Null objekat radi nešto posebno za svaki tip **Robot**a – u ovom slučaju, sadrži podatke o tačnom tipu **Robot**a koji Null objekat zamenjuje. Te podatke će hvatati dinamički posrednik:

```
//: podaciotipu/NullRobot.java
// Pravljenje Null objekta pomoću dinamičkog posrednika.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class BlokZaObraduNullRobota implements InvocationHandler {
  private String nullIme;
  private Robot imaposrednika = new NRobot();
  BlokZaObraduNullRobota(Class<? extends Robot> tip) {
    nullIme = tip.getSimpleName() + " NullRobot";
  }
  private class NRobot implements Null, Robot {
    public String ime() { return nullIme; }
    public String model() { return nullIme; }
    public List<Operacija> operacije() {
      return Collections.emptyList();
    }
  }
  public Object
  invoke(Object posrednik, Method metoda, Object[] argumenti)
  throws Throwable {
    return method.invoke(imaposrednika, argumenti);
  }
}

public class NullRobot {
  public static Robot
  novNullRobot(Class<? extends Robot> tip) {
    return (Robot)Proxy.newProxyInstance(
      NullRobot.class.getClassLoader(),
      new Class[]{ Null.class, Robot.class },
      new BlokZaObraduNullRobota(tip));
  }
  public static void main(String[] args) {
    Robot[] botovi = {
      new RobotKojiCistiSneg("Snezana"),
      novNullRobot(RobotKojiCistiSneg.class)
    };
    for(Robot bot : botovi)
      Robot.Test.test(bot);
  }
} /* Ispis:
Robot ime: Snezana
Robot model: SnegoBot Serija 11
Snezana može da čisti sneg
Snezana čisti sneg
Snezana može da čisti led
Snezana čisti led
Snezana može da počisti krov
Snezana počisti krov
```

```
[Null Robot]
Robot ime: RobotKojiCistiSneg NullRobot
Robot model: RobotKojiCistiSneg NullRobot
*///:~
```

Kad god vam zatreba prazan (null) objekat tipa **Robot**, samo pozovete metodu **novNullRobot( )** prosleđujući joj tip **Robot**a za koji hoćete posrednika. Posrednik ispunjava zahteve interfejsa **Robot** i **Null** i daje specifično ime tipa za koji posreduje.

Lažni objekti i vezivne funkcije

Lažni objekat (engl. *Mock Object*) i *vezivna funkcija* (engl. *Stub*) logičke su varijacije Null objekta. Kao Null objekat, oni su zamene za pravi objekat koji će biti upotrebljen u završenom programu. Međutim, i lažni objekat i vezivna funkcija glume da su živi objekti koji isporučuju prave podatke, umesto što su inteligentne zamene za **null**, kao Null objekat.

Lažni objekat i vezivna funkcija razlikuju se u stepenu. Lažni objekti su uglavnom laki, sami sebe testiraju, i obično se pravi mnogo njih za razne situacije tokom testiranja. Vezivne funkcije vraćaju samo začetke podataka, obično su teški objekti i često se više puta upotrebljavaju u raznim testiranjima. Vezivne funkcije se mogu konfigurisati tako da se izmene u zavisnosti od načina na koji su pozvane. Dakle, vezivna funkcija je sofisticirani objekat koji radi svašta, dok se za razne poslove pravi mnoštvo različitih malih lažnih objekata.

Vežba 24: (4) Dodajte Null objekte u program **RegistrovaniProizvodjaci.java**.

Interfejsi i podaci o tipu

Važna svrha rezervisane reči **interface** jeste da programeru omogući da izoluje komponente i tako smanji njihov međusobni uticaj. To se postiže pisanjem u interfejse, ali što se tiče podataka o tipu, interfejse je moguće zaobići – oni nisu savršeno jemstvo razdvajanja realizacije klase od načina pristupa klasi. Za početak, evo jednog interfejsa:

```
//: podaciotipu/interfejsa/A.java
package podaciotipu.interfejsa;

public interface A {
  void f();
} ///:~
```

Kada se ovaj interfejs realizuje, videćete kako se ipak može saznati stvarni realizovani tip:

```
//: podaciotipu/KrsenjeInterfejsa.java
// Zaobilaženje interfejsa.
import podaciotipu.interfejsa.*;

class B implements A {
  public void f() {}
  public void g() {}
}
```

```
public class KrsenjeInterfejsa {
  public static void main(String[] args) {
    A a = new B();
    a.f();
    // a.g(); // Ovo bi izazvalo grešku u prevođenju
    System.out.println(a.getClass().getName());
    if(a instanceof B) {
      B b = (B)a;
      b.g();
    }
  }
} /* Ispis:
B
*///:~
```

Pomoću prepoznavanja tipa tokom izvršavanja saznajemo da je **a** realizovano kao **B**. Svođenjem tipa na **B**, možemo pozvati metodu koja nije u **A**.

Ovo je sasvim dozvoljeno i prihvatljivo, ali vama možda ne odgovara da programeri klijenti to rade, pošto im to daje priliku da se tešnje povežu s vašim kodom nego što biste vi hteli. Dakle, vi možda mislite da vas štiti rezervisana reč **interface**, ali to nije istina, a činjenica da upotrebljavate **B** da biste realizovali **A**, u ovom slučaju je zapravo pitanje javne tajne.[5]

Možete reći da su programeri sami krivi ako su odlučili da upotrebe klasu umesto interfejsa. To je verovatno tačno u većini slučajeva, ali ako vam „verovatno" nije dovoljno, možete upotrebiti strože kontrole.

Najlakše rešenje je upotrebiti paketni pristup za realizaciju tako da je klijenti izvan paketa ne vide:

```
//: podaciotipu/paketnipristup/SkrivenaC.java
package podaciotipu.paketnipristup;
import podaciotipu.interfejsa.*;
import static net.mindview.util.Print.*;

class C implements A {
  public void f() { print("javna C.f()"); }
  public void g() { print("javna C.g()"); }
  void u() { print("paketni C.u()"); }
  protected void v() { print("zaštićena C.v()"); }
  private void w() { print("privatna C.w()"); }
}

public class SkrivenaC {
  public static A napraviA() { return new C(); }
} ///:~
```

[5] Najčuveniji primer ovoga je operativni sistem Windows; on je imao zvanični API koji je trebalo da bude korišćen za sve, i nezvaničan ali vidljiv skup funkcija koje su korisnici mogli otkriti i pozivati. Da bi rešavali probleme, programeri su upotrebljavali skrivene API funkcije i time primorali Microsoft da ih održava kao da su deo zvaničnog API-ja. To je postalo izvor velikog troška i truda za tu kompaniju.

Jedini javni deo ovog paketa, **SkrivenaC**, proizvodi interfejs **A** kada je pozovete. Čak i kada biste iz metode **napraviA( )** vraćali **C**, zanimljivo je da izvan paketa i dalje ne biste mogli da koristite ništa osim **A**, pošto ime klase **C** izvan paketa ne možete ni da upotrebite.

Ako sada pokušate da svedete naniže na **C**, to nećete moći da uradite, pošto izvan paketa nema dostupnog tipa **C**:

```
//: podaciotipu/SkrivenaRealizacija.java
// Zaobilaženje paketnog pristupa.
import podaciotipu.interfejsa.*;
import podaciotipu.paketnipristup.*;
import java.lang.reflect.*;

public class SkrivenaRealizacija {
  public static void main(String[] args) throws Exception {
    A a = SkrivenaC.napraviA();
    a.f();
    System.out.println(a.getClass().getName());
    // Greška tokom prevođenja: cannot find symbol 'C':
    /* if(a instanceof C) {
      C c = (C)a;
      c.g();
    } */
    // Pazi sad! Refleksija ipak omogućava da pozovemo g():
    pozivSkriveneMetode(a, "g");
    // Pa čak i metode koje su još manje dostupne!
    pozivSkriveneMetode(a, "u");
    pozivSkriveneMetode(a, "v");
    pozivSkriveneMetode(a, "w");
  }
  static void pozivSkriveneMetode(Object a, String imeMetode)
  throws Exception {
    Method g = a.getClass().getDeclaredMethod(imeMetode);
    g.setAccessible(true);
    g.invoke(a);
  }
} /* Ispis:
javna C.f()
podaciotipu.paketnipristup.C
javna C.g()
paketni C.u()
zaštićena C.v()
privatna C.w()
*///:~
```

Kao što vidite, i dalje je zbog refleksije moguće pozivati *sve* metode, čak i privatne! Ako znate ime metode koju hoćete da pozovete, samo pozovite **setAccessible(true)** za taj objekat tipa **Method**, kako piše u definiciji metode **pozivSkriveneMetode( )**.

Možda mislite da ćete to sprečiti tako što ćete distribuirati samo već preveden kôd, ali varate se. Dovoljno je takav kôd propustiti kroz **javap**, prevodilac unazad koji se isporučuje uz JDK. Evo kako izgleda odgovarajuća komandna linija:

```
javap -private C
```

Indikator **-private** kazuje da treba prikazati sve članove, čak i one privatne. Evo rezultata:

```
class podaciotipu.paketnipristup.C extends
java.lang.Object implements podaciotipu.interfejsa.A {
    podaciotipu.paketnipristup.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}
```

Na taj način, svi mogu saznati imena i potpise svih vaših (i najprivatnijih) metoda i pozvati ih.

Šta će biti ako interfejs realizujete kao privatnu unutrašnju klasu? Evo kako to izgleda:

```
//: podaciotipu/UnutrašnjaRealizacija.java
// Privatne unutrašnje klase ne mogu se sakriti od refleksije.
import podaciotipu.interfejsa.*;
import static net.mindview.util.Print.*;

class UnutrasnjaA {
  private static class C implements A {
    public void f() { print("javna C.f()"); }
    public void g() { print("javna C.g()"); }
    void u() { print("paketna C.u()"); }
    protected void v() { print("zaštićena C.v()"); }
    private void w() { print("privatna C.w()"); }
  }
  public static A napraviA() { return new C(); }
}

public class UnutrašnjaRealizacija {
  public static void main(String[] args) throws Exception {
    A a = UnutrasnjaA.napraviA();
    a.f();
    System.out.println(a.getClass().getName());
    // Refleksija i dalje hvata privatne klase:
    SkrivenaRealizacija.pozivSkriveneMetode(a, "g");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "u");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "v");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "w");
  }
```

```
} /* Ispis:
javna C.f()
UnutrasnjaA$C
javna C.g()
paketna C.u()
zaštićena C.v()
privatna C.w()
*///:~
```

Time se ništa nije sakrilo od refleksije. Da li će to uspeti anonimnoj klasi?

```
//: podaciotipu/AnonimnaRealizacija.java
// Anonimne unutrašnje klase ne mogu se sakriti od refleksije.
import podaciotipu.interfejsa.*;
import static net.mindview.util.Print.*;

class AnonimnaA {
  public static A napraviA() {
    return new A() {
      public void f() { print("javna C.f()"); }
      public void g() { print("javna C.g()"); }
      void u() { print("paketna C.u()"); }
      protected void v() { print("zaštićena C.v()"); }
      private void w() { print("privatna C.w()"); }
    };
  }
}
public class AnonimnaRealizacija {
  public static void main(String[] args) throws Exception {
    A a = AnonimnaA.napraviA();
    a.f();
    System.out.println(a.getClass().getName());
    // Refleksija i dalje hvata anonimne klase:
    SkrivenaRealizacija.pozivSkriveneMetode(a, "g");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "u");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "v");
    SkrivenaRealizacija.pozivSkriveneMetode(a, "w");
  }
} /* Ispis:
javna C.f()
AnonimnaA$1
javna C.g()
paketna C.u()
zaštićena C.v()
privatna C.w()
*///:~
```

Kao da ne postoji način da se spreči da refleksija pronađe i pozove metode koje imaju nejavni pristup. Isto važi i za polja, čak i ona privatna:

```
//: podaciotipu/ModifikovanjePrivatnihPolja.java
import java.lang.reflect.*;

class UzPrivatnoFinalnoPolje {
  private int i = 1;
  private final String s = "Potpuno sam bezbedan";
  private String s2 = "Jesam li bezbedan?";
  public String toString() {
    return "i = " + i + ", " + s + ", " + s2;
  }
}

public class ModifikovanjePrivatnihPolja {
  public static void main(String[] args) throws Exception {
    UzPrivatnoFinalnoPolje pf = new UzPrivatnoFinalnoPolje();
    System.out.println(pf);
    Field p = pf.getClass().getDeclaredField("i");
    p.setAccessible(true);
    System.out.println("p.getInt(pf): " + p.getInt(pf));
    p.setInt(pf, 47);
    System.out.println(pf);
    p = pf.getClass().getDeclaredField("s");
    p.setAccessible(true);
    System.out.println("p.get(pf): " + p.get(pf));
    p.set(pf, "Ne, nisi!");
    System.out.println(pf);
    p = pf.getClass().getDeclaredField("s2");
    p.setAccessible(true);
    System.out.println("p.get(pf): " + p.get(pf));
    p.set(pf, "Ne, nisi!");
    System.out.println(pf);
  }
} /* Ispis:
i = 1, Potpuno sam bezbedan, Jesam li bezbedan?
p.getInt(pf): 1
i = 47, Potpuno sam bezbedan, Jesam li bezbedan?
p.get(pf): Potpuno sam bezbedan
i = 47, Potpuno sam bezbedan, Jesam li bezbedan?
p.get(pf): Jesam li bezbedan?
i = 47, Potpuno sam bezbedan, Ne, nisi!
*///:~
```

Međutim, finalna polja se zapravo ne mogu menjati. Izvršni sistem se neće žaliti na pokušaje takvih izmena, ali ih neće ni sprovesti.

Po pravilu, sva ta kršenja prava pristupa nisu najgora stvar na svetu. Ukoliko neko upotrebi takvu tehniku za pozivanje metoda koje ste označili kao **private** ili kao metode s paketnim pristupom (čime ste svima jasno stavili do znanja kako ne bi trebalo da ih pozivaju), onda teško može da se žali kada izmenite neki aspekt tih metoda. S druge strane, činjenica da uvek imate mala vrata u klasu, može omogućiti da rešite određene vrste problema koji bi inače bili teški ili nemogući, a prednosti refleksije su, uopšteno govoreći, neosporne.

Vežba 25: (2) Napravite klasu koja sadrži privatne metode, zaštićene metode i metode s paketnim pristupom. Napišite kôd za pristupanje tim metodama spolja, tj. izvan paketa te klase.

Sažetak

Prepoznavanje tipa tokom izvršavanja omogućuje otkrivanje informacija o tipu pomoću anonimne reference na klasu. Početnici ga često zloupotrebljavaju jer im izgleda pogodnije od poziva polimorfnih metoda. Mnogi ljudi, navikli na proceduralno programiranje, teško se odvikavaju od organizovanja programa u obliku skupa naredaba **switch**. Takva struktura bi se mogla postići određivanjem tipa u vreme izvršavanja, ali bi se izgubila važna osobina polimorfizma tokom razvoja i održavanja koda. Suština objektno orijentisanog programiranja je da se u kodu koriste pozivi polimorfnih metoda, a da se prepoznavanje tipa u vreme izvršavanja koristi samo kada je neophodno.

Da bi se pozivi polimorfnih metoda koristili na propisani način, mora se kontrolisati definicija osnovne klase, zato što bi se moglo desiti da u određenom trenutku, kada proširujete program, ustanovite kako osnovna klasa ne sadrži potrebnu metodu. Ako osnovna klasa potiče iz biblioteke ili je kontroliše neko drugi, rešenje problema leži u prepoznavanju tipa tokom izvršavanja, jer tako možete izvesti nov tip i dodati željenu metodu. Na nekom drugom mestu u kodu možete da otkrijete pravi tip i pozovete specijalnu metodu. Polimorfizam i mogućnost proširivanja programa neće biti upropašćeni, jer se pri dodavanju novog tipa ne mora tragati za naredbama **switch** u programu. Međutim, u kodu za koji je neophodna nova metoda, morate primeniti prepoznavanje tipa u vreme izvršavanja da biste otkrili određeni tip.

Zbog smeštanja nove funkcije u osnovnu klasu moglo bi se desiti da zarad jedne klase svim ostalim klasama izvedenim iz iste osnovne klase treba dodati neku besmislenu metodu. Time se gubi preglednost interfejsa i nerviraju se programeri, koji moraju da redefinišu apstraktne metode kada izvode druge klase iz te osnovne klase. Na primer, posmatrajmo hijerarhiju klasa koja predstavlja muzičke instrumente. Pretpostavimo da želite da pročistite piskove svih duvačkih instrumenata u orkestru. Jedna mogućnost je da stavite metodu **procistiPisak**() u osnovnu klasu **Instrument**, ali to zbunjuje zato što podrazumeva da i udarački i žičani instrumenti imaju pisak. Prepoznavanje tipa u vreme izvršavanja obezbeđuje mnogo pogodnije rešenje u ovom slučaju zato što metodu možete staviti u određenu klasu (u ovom slučaju, u klasu duvačkih instrumenata) gde joj je i mesto. Međutim, mnogo bolje rešenje je da se metoda **pripremiInstrument**() stavi u osnovnu klasu. Ipak, to možda nećete predvideti kada se prvi put sretnete s problemom, pa ćete pogrešno zaključiti da morate koristiti prepoznavanje tipa u vreme izvršavanja.

Na kraju, primenom prepoznavanja tipa tokom izvršavanja ponekad se rešavaju problemi sa efikasnošću. Ako se u kodu dobro koristi polimorfizam, ali objekat reaguje na kôd opšte namene izuzetno neefikasno, možete da izolujete njegov tip zahvaljujući prepoznavanju u vreme izvršavanja i da napišete kôd za taj poseban slučaj da biste povećali efikasnost. Ipak, nemojte da srljate u programiranje radi povećanja efikasnosti bez prethodne provere, jer je to zamka. Najbolje je prvo nekako naterati program da *radi*, potom odlučiti da li radi dovoljno brzo, i tek nakon toga razmotriti efikasnost.(To se radi profajlerom – videti dodatak na adresi *http://MindView.net/Books/BetterJava*.)

Videli smo da refleksija otvara nov svet mogućnosti u programiranju, tako što omogućuje mnogo dinamičniji stil programiranja. Ima ljudi koje onespokojava dinamična priroda refleksije. Činjenica da možete raditi nešto što se može proveriti tek u vreme izvršavanja i prijaviti pomoću izuzetaka, izgleda onima koji su navikli na bezbednost statičke provere tipova kao pogrešan smer. Neki preteruju, pa kažu kako je uvođenje mogućnosti pojave izuzetka u vreme izvršavanja jasan pokazatelj da takav kôd treba izbegavati. Smatram da je to osećanje bezbednosti iluzorno – u vreme izvršavanja uvek se može desiti nešto što će izazvati izuzetak, čak i u programu koji ne sadrži ni blokove **try** niti specifikacije izuzetaka. Mislim da dosledni model prijavljivanja grešaka *omogućuje* da pišemo dinamički kôd i upotrebljavamo refleksiju. Naravno, vredi pokušati napisati kôd koji se može statički proveriti ... kada je to moguće. Ali smatram da je dinamički kôd jedna od važnih mogućnosti koje odvajaju Javu od jezika kao što je C++.

Vežba 26: (3) Realizujte metodu **procistiPisak()** kako je opisano u sažetku.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Generički tipovi

OBIČNE KLASE I METODE RADE S POTPUNO ODREĐENIM TIPOVIMA: BILO PROSTIM, BILO tipovima klasa. Ako pišete kôd koji se može upotrebiti s više tipova, ta krutost ume previše da ograničava.[1]

Jedan od načina na koji objektno orijentisani jezici omogućuju uopštavanje jeste polimorfizam. Primera radi, možete napisati metodu koja kao argument prima objekat osnovne klase, i zatim upotrebiti tu metodu s bilo kojom klasom izvedenom iz te osnovne klase. Tako vaša metoda postaje nešto opštija i može se upotrebljavati na više mesta. Isto važi i unutar klasa – gde god upotrebljavate određeni tip, osnovni tip pruža veću fleksibilnost. Naravno, može se proširiti (naslediti) sve sem finalne klase,[2] pa se ta fleksibilnost većinom dobija automatski.

Ponekad ograničenost na jednu hijerarhiju previše sputava. Ukoliko je argument metode interfejs a ne klasa, ograničenja se ublažavaju jer se obuhvata sve što realizuje taj interfejs – čak i klase koje još nisu ni napravljene. Time programer klijent dobija mogućnost da realizuje interfejs kako bi se prilagodio vašoj klasi ili metodi. Tako interfejsi omogućuju premošćavanje hijerarhije klasa, ukoliko imate mogućnost da napravite novu klasu za to.

Katkada i interfejs previše ograničava. Svaki interfejs zahteva da kôd radi baš s tim konkretnim interfejsom. Kôd bi bio opštiji kada biste mogli da zadate da radi „s nekim nespecificiranim tipom", a ne s bilo kojim konkretnim interfejsom ili klasom.

To je zamisao u pozadini generičkih tipova – oni predstavljaju jedno od značajnijih poboljšanja koje je donela Java SE5. Generički tipovi realizuju koncept *parametrizovanih tipova* – oni omogućuju pisanje komponenata (obično kontejnera) koje se lako upotrebljavaju s više tipova. Termin „generički" znači „koji odgovara velikim grupama klasa ili važi za njih". Generički tipovi su uvedeni u programske jezike da bi omogućili programerima najveću moguću izražajnost kada pišu klase ili metode, tako što se ublažavaju ograničenja na tipove s kojima te klase ili metode rade. Kao što ćete videti u ovom poglavlju, Javini generički tipovi nisu toliko moćni – štaviše, mogli biste se zapitati da li reč „generički" uopšte prikladno opisuje te mogućnosti.

Ako nikada niste radili s mehanizmom parametrizovanih tipova, Javini generički tipovi verovatno će vam izgledati kao podesan dodatak jeziku. Kada napravite primerak parametrizovanog tipa, konverzije tipova se obavljaju automatski, a ispravnost tipova proverava u vreme prevođenja. To liči na poboljšanje.

Međutim, ukoliko ste prethodno imali posla s nekim mehanizmom parametrizovanih tipova, recimo u jeziku C++, videćete da s Javinim generičkim tipovima ne možete da uradite baš sve što biste od njih očekivali. Korišćenje tuđih generičkih tipova je prilično lako, ali pravljenje sopstvenih je puno iznenađenja. Između ostaloga, pokušaću da vam objasnim kako je došlo do takve realizacije generičkih tipova u Javi.

---

[1] Angelika Langer piše *Java Generics FAQ* (Odgovori na najčešća pitanja o Javinim generičkim tipovima, videti *www.langer.camelot.de*). Ti i drugi njeni tekstovi (napisani zajedno s Klausom Kreftom) bili su mi od neprocenjive vrednosti tokom pripreme ovog poglavlja.

[2] Ili klasa koja ima samo privatne konstruktore.

Ne kažem da su Javini generički tipovi beskorisni. U mnogo slučajeva, oni kôd čine jednostavnijim, pa čak i elegantnim. Ali ako ste koristili jezik u kojem je realizovana čistija verzija generičkih tipova, mogli biste da se razočarate. U ovom poglavlju istražićemo i jake i slabe strane Javinih generičkih tipova, da biste delotvornije mogli da ih koristite.

Poređenje sa C++-om

Projektanti Jave kažu da je dobar deo tog jezika bio reakcija na C++. Uprkos tome, Javu je moguće predavati uglavnom bez spominjanja C++-a, i ja sam se trudio da radim tako, sem kada poređenje daje dublji uvid.

Generički tipovi zahtevaju više poređenja s C++-om iz dva razloga. Prvo, kada shvatite određene aspekte C++-ovih *šablona* (engl. *templates*) – koji su bili glavno nadahnuće za generičke tipove, čak i za njihovu osnovnu sintaksu – bolje ćete shvatiti temelje tog koncepta, kao i – a to je veoma važno – ograničenja prilikom upotrebe Javinih generičkih tipova i razloge za njih. Krajnji cilj je da steknete jasnu predstavu o tome gde su granice, pošto znam iz sopstvenog iskustva: kada shvatite gde su granice, postajete bolji programer. Kada znate šta ne može da se uradi, bolje ćete iskoristiti ono što može (delom i zato što ne gubite vreme sudarajući se sa zidovima).

Drugi razlog je značajan nesporazum koji unutar Javine zajednice vlada kada je reč o C++ šablonima. Taj nesporazum može još više da vas zbuni u pogledu predviđene upotrebe generičkih tipova.

Stoga ću u ovom poglavlju dati tek minimalan broj primera s C++ šablonima.

Jednostavni generički tipovi

Jedna od najvažnijih početnih motivacija za uvođenje generičkih tipova bila je pravljenje *kontejnerskih klasa*, koje ste upoznali u poglavlju *Čuvanje objekata* (a naučićete još u poglavlju *Detaljno razmatranje kontejnera*). Kontejner je mesto za skladištenje objekata dok radite s njima. Isto važi i za nizove, ali kontejneri su fleksibilniji i imaju drugačija obeležja od običnih nizova. Gotovo svi programi zahtevaju da negde čuvate grupu objekata dok radite s njima, pa kontejneri spadaju među najčešće upotrebljavane biblioteke klasa.

Pogledajmo klasu koja čuva jedan objekat. Naravno, klasa bi mogla da zada tačan tip tog objekta, ovako:

```
//: generici/Skladiste1.java

class Automobil {}

public class Skladiste1 {
  private Automobil a;
  public Skladiste1(Automobil a) { this.a = a; }
  Automobil get() { return a; }
} ///:~
```

Ali tu alatku nije lako ponovno upotrebiti, pošto se ne može koristiti za skladištenje ičega drugoga. Bilo bi bolje da ne moramo da pišemo novu za svaki tip na koji naiđemo.

Pre Jave SE5, jednostavno bismo joj dali da čuva jedan **Object**:

```
//: genericki/Skladiste2.java

public class Skladiste2 {
  private Object a;
  public Skladiste2(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
  public static void main(String[] args) {
    Skladiste2 h2 = new Skladiste2(new Automobil());
    Automobil a = (Automobil)h2.get();
    h2.set("Nije Automobil");
    String s = (String)h2.get();
    h2.set(1); // Automatskim pakovanjem postaje Integer
    Integer x = (Integer)h2.get();
  }
} ///:~
```

Sada **Skladiste2** može da primi bilo šta – a u ovom primeru isto **Skladiste2** čuva tri različita tipa objekata.

Ima slučajeva kada hoćete da kontejner skladišti više tipova objekata, ali se u kontejner obično stavlja jedan tip objekata. Jedna od osnovnih motivacija za generičke tipove bilo je zadavanje tipa objekta koji kontejner sadrži, i da tu specifikaciju podrži i proveri prevodilac.

Zato bismo umesto tipa **Object** hteli da upotrebimo nespecificiran tip, koji može biti određen kasnije. To se radi tako što nakon imena klase unutar znakova manje od i veće od napišete *parametar tipa*, i zamenite ga stvarnim tipom kada tu klasu upotrebite. Za klasu **Skladiste** to bi izgledalo ovako (**T** je parametar tipa):

```
//: genericki/Skladiste3.java

public class Skladiste3<T> {
  private T a;
  public Skladiste3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    Skladiste3<Automobil> h3 =
      new Skladiste3<Automobil>(new Automobil());
    Automobil a = h3.get(); // Konverzija tipa nije potrebna
    // h3.set("Nije Automobil"); // Greška
    // h3.set(1); // Greška
  }
} ///:~
```

Kada sada napravite neko **Skladiste3**, istom sintaksom sa znakovima manje od i veće od morate zadati tip koji hoćete da čuvate u njemu, kao što vidite u metodi **main**( ). U Skladiste je dozvoljeno stavljati samo objekte tog tipa (ili podtipa, pošto princip zamene

funkcioniše i u generičkim tipovima). A kada iz njega izvadite nešto, ono je automatski ispravnog tipa.

To je osnovna ideja Javinih generičkih tipova: vi zadate tip koji hoćete da upotrebljavate, a Java se dalje stara za detalje.

Po pravilu, generičke tipove možete tretirati kao da su bilo koji drugi tip – samo što imaju i parametar tipa. Ali kao što ćete videti, generički tip upotrebljavate navođenjem njegovog imena i liste njegovih argumenata tipova.

Vežba 1: (1) Upotrebite **Skladiste3** s bibliotekom **podaciotipu.ljubimci** da biste pokazali da **Skladiste3**, specificirano da čuva osnovni tip, može da čuva i njegov izvedeni tip.

Vežba 2: (1) Napravite klasu skladista koje čuva tri objekta istog tipa, metode za skladištenje i vađenje tih objekata, i konstruktor za inicijalizaciju sva tri objekta.

Biblioteka n-torki

Često bi bilo poželjno vratiti više objekata iz jednog poziva metode. Naredba **return** dozvoljava zadavanje samo jednog objekta, pa je rešenje napraviti objekat koji sadrži više objekata koje hoćete da vratite. Naravno, možete da pišete posebnu klasu svaki put kada naiđete na takvu situaciju, ali pomoću generičkih tipova problem možete rešiti jednom zauvek i tako sebi uštedeti trud ubuduće. Istovremeno, u vreme prevođenja svi generički tipovi će biti provereni.

Ovaj koncept se naziva *n-torka* (engl. *tuple*), a radi se o grupi objekata omotanih zajedno u jedan objekat. Primalac objekta može da čita njegove elemente, ali ne i da stavlja nove u njega. (Ovaj koncept nazivaju i *Data Transfer Object* ili *Messenger*, Objekat za prenos podataka ili Dostavljač poruka.)

n-torke mogu biti proizvoljne dužine i svaki njihov objekat može biti različitog tipa. Međutim, mi hoćemo da zadamo tip svakog objekta i obezbedimo da primalac – kada pročita njegovu vrednost-dobije ispravan tip. Problem zbog različitih dužina rešićemo pravljenjem različitih n-torki. Evo jedne koja čuva dva objekta:

```
//: net/mindview/util/Dvojka.java
package net.mindview.util;

public class Dvojka<A,B> {
  public final A prvi;
  public final B drugi;
  public Dvojka(A a, B b) { prvi = a; drugi = b; }
  public String toString() {
    return "(" + prvi + ", " + drugi + ")";
  }
} ///:~
```

Konstruktor hvata objekat koji treba sačuvati, a **toString( )** je prigodna funkcija za prikazivanje vrednosti iz liste. Obratite pažnju na to da n-torka svoje elemente implicitno čuva u poretku.

Nakon prvog čitanja ovog primera, možda ste pomislili da se njima krše principi bezbednosti uobičajeni u Java programiranju. Zar ne bi **prvi** i **drugi** trebalo da budu privatni i da im pristupaju samo metode nazvane **dajPrvi( )** i **dajDrugi( )**? Razmotrimo bezbednost koju biste postigli u tom slučaju: klijenti bi i dalje mogli da čitaju objekte **prvi** i **drugi** i da rade s njima šta god hoće, ali ne bi mogli da ih dodele ničemu drugom. Istu bezbednost daje deklaracija **final**, ali je gornji oblik kraći i jednostavniji.

Druga projektna opaska jeste da biste možda *hteli* da dopustite programeru klijentu da objekat **prvi** ili **drugi** usmeri ka drugom objektu. Međutim, bezbednije je da ga ostavite u gornjem obliku i samo primorate korisnika da napravi novu klasu **Dvojka** ukoliko želi neku s drugačijim elementima.

Duže n-torke se prave nasleđivanjem. Videćete da je dodavanje većeg broja parametara tipa jednostavno:

```
//: net/mindview/util/Trojka.java
package net.mindview.util;

public class Trojka<A,B,C> extends Dvojka<A,B> {
  public final C treci;
  public Trojka(A a, B b, C c) {
    super(a, b);
    treci = c;
  }
  public String toString() {
    return "(" + prvi + ", " + drugi + ", " + treci +")";
  }
} ///:~

//: net/mindview/util/Cetvorka.java
package net.mindview.util;

public class Cetvorka<A,B,C,D> extends Trojka<A,B,C> {
  public final D cetvrti;
  public Cetvorka(A a, B b, C c, D d) {
    super(a, b, c);
    cetvrti = d;
  }
  public String toString() {
    return "(" + prvi + ", " + drugi + ", " +
      treci + ", " + cetvrti + ")";
  }
} ///:~

//: net/mindview/util/Petorka.java
package net.mindview.util;

public class Petorka<A,B,C,D,E>
extends Cetvorka<A,B,C,D> {
```

```
  public final E peti;
  public Petorka(A a, B b, C c, D d, E e) {
    super(a, b, c, d);
    peti = e;
  }
  public String toString() {
    return "(" + prvi + ", " + drugi + ", " +
      treci + ", " + cetvrti + ", " + peti + ")";
  }
} ///:~
```

n-torku koristite tako što n-torku željene dužine definišete kao povratnu vrednost svoje funkcije i zatim je napravite i vratite naredbom **return**:

```
//: genericki/IspitivanjeEntorki.java
import net.mindview.util.*;

class Amfibija {}
class Vozilo {}

public class IspitivanjeEntorki {
  static Dvojka<String,Integer> f() {
    // Automatsko pakovanje pretvara int u Integer:
    return new Dvojka<String,Integer>("zdravo", 47);
  }
  static Trojka<Amfibija,String,Integer> g() {
    return new Trojka<Amfibija, String, Integer>(
      new Amfibija(), "zdravo", 47);
  }
  static
  Cetvorka<Vozilo,Amfibija,String,Integer> h() {
    return
      new Cetvorka<Vozilo,Amfibija,String,Integer>(
        new Vozilo(), new Amfibija(), "zdravo", 47);
  }
  static
  Petorka<Vozilo,Amfibija,String,Integer,Double> k() {
    return new
      Petorka<Vozilo,Amfibija,String,Integer,Double>(
        new Vozilo(), new Amfibija(), "zdravo", 47, 11.1);
  }
  public static void main(String[] args) {
    Dvojka<String,Integer> iesi = f();
    System.out.println(iesi);
    // iesi.prvi = "tamo"; // Greška u prevođenju: final
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
```

```
} /* Ispis: (80% podudaranja)
(zdravo, 47)
(Amfibija@1f6a7b9, zdravo, 47)
(Vozilo@35ce36, Amfibija@757aef, zdravo, 47)
(Vozilo@9cab16, Amfibija@1a46e30, zdravo, 47, 11.1)
*///:~
```

Pomoću generičkih tipova lako možete postići da svaka n-torka vraća proizvoljnu grupu tipova; dovoljno je da napišete odgovarajući izraz.

Možete videti kako odredba **final** za javna polja sprečava da ona budu ponovno dodeljena nakon konstrukcije, jer se naredba **iesi.prvi = "tamo"** ne može prevesti.

Treba prilično mnogo pisati u **new** izrazima. U nastavku poglavlja videćete kako da ih pojednostavite *generičkim metodama.*

Vežba 3: (1) Napravite i isprobajte generičku **Sestorku**.

Vežba 4: (3) Uopštite program **unutrasnjeklase/Sekvenca.java** pomoću generičkih tipova.

Klasa steka

Pogledajmo nešto što je malo komplikovanije: klasični stek s kojeg se prvo uzima ono što je na njega poslednje stavljeno. U poglavlju *Čuvanje objekata* realizovali smo stek pomoću ulančane liste (**LinkedList**) kao klasu **net.mindview.util.Stack** (na stranama 320-323). U tom primeru vidite da ulančana lista već ima metode potrebne za pravljenje steka. Klasu **Stack** napravili smo slaganjem jedne generičke klase (**Stack<T>**) s drugom generičkom klasom (**LinkedList<T>**). Na tom primeru vidite da je generički tip kao i svaki drugi, (uz nekoliko izuzetaka koje ćemo razmotriti kasnije).

Umesto da upotrebimo **LinkedList**, možemo da realizujemo sopstveni unutrašnji ulančani mehanizam za skladištenje.

```
//: generics/UlancaniStek.java
// Stek realizovan pomoću unutrašnje ulančane strukture.

public class UlancaniStek<T> {
  private static class Cvor<U> {
    U stavka;
    Cvor<U> sledeca;
    Cvor() { stavka = null; sledeca = null; }
    Cvor(U stavka, Cvor<U> sledeca) {
      this.stavka = stavka;
      this.sledeca = sledeca;
    }
    boolean kraj() { return stavka == null && sledeca == null; }
  }
  private Cvor<T> vrh = new Cvor<T>(); // Oznaka kraja
  public void stavina(T stavka) {
    vrh = new Cvor<T>(stavka, vrh);
  }
```

```
  public T skinisa() {
    T rezultat = vrh.stavka;
    if(!vrh.kraj())
      vrh = vrh.sledeca;
    return rezultat;
  }
  public static void main(String[] args) {
    UlancaniStek<String> uss = new UlancaniStek<String>();
    for(String s : "Phasers on stun!".split(" "))
      uss.stavina(s);
    String s;
    while((s = uss.skinisa()) != null)
      System.out.println(s);
  }
} /* Ispis:
stun!
on
Phasers
*///:~
```

I unutrašnja klasa **Cvor** je generička i ima vlastiti parametar tipa.

U ovom primeru, „oznaka kraja" (engl. *end sentinel*) utvrđuje kada je stek prazan. Oznaka se pravi prilikom konstruisanja objekta tipa **UlancaniStek**, i svaki put kada pozovete **stavina( )**, pravi se nov **Cvor**<T> i ulančava s prethodnim objektom tipa **Cvor**<T>. Kada pozovete **skinisa( )**, uvek vraćate **vrh.stavka** i potom odbacujete tekući **Cvor**<T> i prelazite na sledeći – sem kada naiđete na oznaku kraja, jer se u tom slučaju ne pomerate. Stoga će klijent koji stalno poziva **skinisa( )** stalno dobijati **null**, što pokazuje da je stek prazan.

Vežba 5: (2) Uklonite parametar tipa iz klase **Cvor** i izmenite ostatak koda u programu **UlancaniStek.java** tako da pokažete da unutrašnja klasa ima pristup generičkim parametrima tipa svoje spoljne klase.

NasumicnaLista

Kao drugi primer za skladiste, pretpostavimo da hoćete posebnu vrstu liste koja nasumično bira jedan od svojih elemenata kada se pozove **izaberi( )**. Pošto hoćete alatku koja radi sa svim objektima, koristite generičke tipove:

```
//: genericki/NasumicnaLista.java
import java.util.*;

public class NasumicnaLista<T> {
  private ArrayList<T> skladiste = new ArrayList<T>();
  private Random slucajan = new Random(47);
  public void add(T stavka) { skladiste.add(stavka); }
  public T select() {
    return skladiste.daj(slucajan.nextInt(skladiste.size()));
  }
```

```
  public static void main(String[] args) {
    NasumicnaLista<String> nls = new NasumicnaLista<String>();
    for(String s: ("The quick brown fox jumped over " +
        "the lazy brown dog").split(" "))
      nls.add(s);
    for(int i = 0; i < 11; i++)
      System.out.print(nls.izaberi() + " ");
  }
} /* Ispis:
brown over fox quick quick dog brown The brown lazy brown
*///:~
```

Vežba 6: (1) Upotrebite klasu **NasumicnaLista** s dva tipa više nego što je pokazano u metodi **main( )**.

Generički interfejsi

Generički tipovi su prikladni i za interfejse. Na primer, *generator* je klasa koja pravi objekte. Zapravo, radi se o specijalizaciji projektnog obrasca *Factory Method* (Proizvodna metoda), ali kada od generatora zatražite nov objekat, ne prosleđujete mu argumente, dok ih Proizvodna metoda obično prosleđujete. Generator zna kako da pravi nove objekte bez ikakvih dodatnih informacija.

Generator obično definiše samo jednu metodu, onu koja pravi nove objekte. Ovde ćemo je nazvati **sledeci( )** i uključiti među standardne uslužne klase:

```
//: net/mindview/util/Generator.java
// Generički interfejs.
package net.mindview.util;
public interface Generator<T> { T sledeci(); } ///:~
```

Povratni tip metode **sledeci( )** parametrizovan je u T. Kao što vidite, generički tipovi se u interfejsima upotrebljavaju kao i u klasama.

Realizaciju interfejsa **Generator** pokazaćemo na klasama u hijerarhiji kafe:

```
//: generic ki/kafa/Kafa.java
package generic ki.kafa;

public class Kafa {
  private static long brojac = 0;
  private final long id = brojac++;
  public String toString() {
    return getClass().getSimpleName() + " " + id;
  }
} ///:~

//: generic ki/kafa/SMlekom.java
package generic ki.kafa;
public class SMlekom extends Kafa {} ///:~
```

```
//: generici/kafa/Moka.java
package generici.kafa;
public class Moka extends Kafa {} ///:~

//: generici/kafa/Cappuccino.java
package generici.kafa;
public class Cappuccino extends Kafa {} ///:~

//: generici/kafa/Americano.java
package generici.kafa;
public class Americano extends Kafa {} ///:~

//: generici/kafa/Kratka.java
package generici.kafa;
public class Kratka extends Kafa {} ///:~
```

Sada možemo da realizujemo **Generator<Kafa>** koji pravi različite tipove **Kafa** objekata:

```
//: generici/kafa/GeneratorKafe.java
// Generiše različite tipove Kafa:
package generici.kafa;
import java.util.*;
import net.mindview.util.*;

public class GeneratorKafe
implements Generator<Kafa>, Iterable<Kafa> {
  private Class[] tipovi = { SMlekom.class, Moka.class,
    Cappuccino.class, Americano.class, Kratka.class, };
  private static Random slucajan = new Random(47);
  public GeneratorKafe() {}
  // Za iteraciju:
  private int velicina = 0;
  public GeneratorKafe(int vel) { velicina = vel; }
  public Kafa sledeci() {
    try {
      return (Kafa)
        tipovi[slucajan.nextInt(tipovi.length)].newInstance();
      // U vreme izvršavanja, prijavi greške programera:
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  class IteratorKafa implements Iterator<Kafa> {
    int broj = velicina;
    public boolean hasNext() { return broj > 0; }
    public Kafa sledeci() {
      broj--;
      return GeneratorKafe.this.sledeci();
    }
```

```
    public void ukloni() { // Nije realizovano
      throw new UnsupportedOperationException();
    }
  };
  public Iterator<Kafa> iterator() {
    return new IteratorKafa();
  }
  public static void main(String[] args) {
    GeneratorKafe gen = new GeneratorKafe();
    for(int i = 0; i < 5; i++)
      System.out.println(gen.sledeci());
    for(Kafa c : new GeneratorKafe(5))
      System.out.println(c);
  }
} /* Ispis:
Americano 0
SMlekom 1
Americano 2
Moka 3
Moka 4
Kratka 5
Americano 6
SMlekom 7
Cappuccino 8
Cappuccino 9
*///:~
```

Parametrizovan interfejs **Generator** stara se da metoda **sledeci( )** vraća parametar tipa. I **GeneratorKafe** realizuje interfejs **Iterable**, pa ga možemo upotrebiti u foreach naredbi. Međutim, da bi znao kada da se zaustavi, treba mu „oznaka kraja", a njega pravi drugi konstruktor.

Evo druge realizacije interfejsa **Generator<T>** koji ovoga puta pravi brojeve Fibonaccijevog niza:

```
//: genericki/Fibonacci.java
// Pravljenje Fibonaccijevog niza.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
  private int broj = 0;
  public Integer sledeci() { return fib(broj++); }
  private int fib(int n) {
    if(n < 2) return 1;
    return fib(n-2) + fib(n-1);
  }
  public static void main(String[] args) {
    Fibonacci gen = new Fibonacci();
    for(int i = 0; i < 18; i++)
      System.out.print(gen.sledeci() + " ");
  }
```

```
} /* Ispis:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

Iako s prostim tipom **int** radimo i unutar i izvan klase, parametar tipa je **Integer**. To je jedno od ograničenja Javinih generičkih tipova: prosti tipovi ne mogu biti parametri tipa. Međutim, Java SE5 je veoma podesno uvela automatsko pakovanje i raspakivanje, za konverziju prostih tipova u omotačke i nazad. Rezultat toga vidite upravo ovde, jer klasa bez problema upotrebljava i pravi brojeve tipa **int**.

Možemo otići korak dalje i napraviti Fibonaccijev generator koji realizuje interfejs **Iterable**. Mogli bismo da ponovo realizujemo klasu i dodamo joj interfejs **Iterable**, ali nemate uvek kontrolu nad izvornim kodom, niti hoćete da ponovo pišete ono što ne morate. Umesto toga, napravićemo *adapter* koji će dati željeni interfejs. (Taj projektni obrazac predstavili smo u prethodnom delu knjige).

Adapteri se mogu realizovati na više načina. Na primer, adaptiranu klasu možete da generišete nasleđivanjem:

```
//: genericki/IterabilniFibonacci.java
// Adaptacija Fibonaccijeve klase koja je čini iterabilnom.
import java.util.*;

public class IterabilniFibonacci
extends Fibonacci implements Iterable<Integer> {
  private int n;
  public IterabilniFibonacci(int broj) { n = broj; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      public boolean hasNext() { return n > 0; }
      public Integer sledeci() {
        n--;
        return IterabilniFibonacci.this.sledeci();
      }
      public void ukloni() { // Nije realizovano
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(int i : new IterabilniFibonacci(18))
      System.out.print(i + " ");
  }
} /* Ispis:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

U foreach naredbi, klasu **IterabilniFibonacci** upotrebljavate tako što njenom konstruktoru zadajete granicu, da bi metoda **hasNext( )** znala kada treba da vrati **false**.

Vežba 7: (2) Napravite klasu **Fibonacci** iterabilnom pomoću kompozicije, a ne nasleđivanja.

Vežba 8: (2) Po uzoru na primer **Kafa**, napravite hijerarhiju objekata tipa **Likovi** iz vašeg omiljenog filma; podelite ih na **Pozitivce** i **Negativce**. Napravite generator za objekte tipa **Likovi**, po uzoru na **GeneratorKafe**.

Generičke metode

Dosad smo parametrizovali cele klase. Možete parametrizovati i pojedine metode unutar klase. Sama klasa može, ali ne mora biti generička – to ne zavisi od generičnosti metode.

Generičnost metode znači da ona može da se menja nezavisno od klase. Po pravilu, generičke metode bi trebalo da upotrebljavate „kad god možete". Drugim rečima, ako se umesto cele klase samo metoda može napraviti da bude generička, verovatno će tako i kôd biti jasniji. Pored toga, ako je metoda statična, ona nema pristupa generičkim parametrima tipa (svoje generičke) klase, pa ukoliko treba da koristi generičnost, mora sama da bude generička.

Generička metoda se definiše stavljanjem liste generičkih parametara ispred povratne vrednosti, ovako:

```
//: generici/GenerickeMetode.java

public class GenerickeMetode {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenerickeMetode gm = new GenerickeMetode();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
} /* Ispis:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenerickeMetode
*///:~
```

Klasa **GenerickeMetode** nije parametrizovana, iako i klasa i njene metode mogu biti parametrizovane istovremeno. Ali u ovom slučaju, parametar tipa ima samo metoda **f( )**, što se vidi iz liste parametara ispred povratnog tipa metode.

Vodite računa o tome da parametre tipa generičke klase morate da zadate prilikom pravljenja njenog primerka. Ali tipove parametara generičke metode obično ne morate da zadajete, pošto prevodilac to može da shvati i uradi umesto vas. To se naziva *zaključivanje o tipu argumenta* (engl. *type argument inference*). Stoga pozivi metode **f( )** izgledaju kao pozivi obične metode; izgleda kao da je **f( )** preklopljena beskonačan broj puta. Ona prima čak i argument tipa **GenerickeMetode**.

Pri pozivima metode **f( )** u kojima su upotrebljeni prosti tipovi, na scenu stupa automatsko pakovanje koje automatski omotava proste tipove u odgovarajuće objekte. U stvari, generičke metode i automatsko pakovanje mogu da zamene deo koda koji je prethodno zahtevao ručnu konverziju tipova.

Vežba 9: (1) Izmenite **GenerickeMetode.java** tako da **f( )** prima tri argumenta različitih parametrizovanih tipova.

Vežba 10: (1) Izmenite prethodnu vežbu tako da jedan od argumenata metode **f( )** ne bude parametrizovan.

Korišćenje zaključivanja o tipu argumenta

Jedna od primedaba na račun generičkih tipova glasi: zbog njih kôd postaje još opširniji. Pogledajmo program **cuvanje/MapaLista.java** iz poglavlja *Čuvanje objekata*. Ovako izgleda pravljenje **Mape Lista**:

```
Map<Osoba, List<? extends Ljubimci>> ljudiSLjubimcima =
  new HashMap<Osoba, List<? extends Ljubimci>>();
```

(Ovu upotrebu rezervisane reči **extends** i znaka pitanja objasnićemo u nastavku poglavlja.) Izgleda da se ponavljate i da bi prevodilac trebalo da iz jedne liste generičkih argumenata shvati šta da stavi u drugu. Nažalost, on to ne može, ali zaključivanje o tipu argumenta u generičkoj metodi može da donese neka pojednostavljenja. Primera radi, napravićemo uslužnu klasu s raznim statičnim metodama koja pravi najčešće upotrebljavane realizacije raznih kontejnera:

```
//: net/mindview/util/Nova.java
// Uslužne metode koje pravljenje generičkih kontejnera
// pojednostavljuju tako što same zaključuju o tipu argumenta.
package net.mindview.util;
import java.util.*;

public class Nova {
  public static <K,V> Map<K,V> map() {
    return new HashMap<K,V>();
  }
  public static <T> List<T> list() {
    return new ArrayList<T>();
  }
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
  }
```

```
  public static <T> Set<T> set() {
    return new HashSet<T>();
  }
  public static <T> Queue<T> queue() {
    return new LinkedList<T>();
  }
  // Primeri:
  public static void main(String[] args) {
    Map<String, List<String>> sls = Nova.map();
    List<String> ls = Nova.list();
    LinkedList<String> lls = Nova.lList();
    Set<String> ss = Nova.set();
    Queue<String> qs = Nova.queue();
  }
} ///:~
```

U metodi **main( )** možete videti primere korišćenja ove klase – zaključivanje o tipu argumenta uklanja potrebu za ponavljanjem liste generičkih parametara. To se može primeniti na **cuvanje/MapaLista.java**:

```
//: genericki/JednostavnijiLjubimci.java
import podaciotipu.ljubimci.*;
import java.util.*;
import net.mindview.util.*;

public class JednostavnijiLjubimci {
  public static void main(String[] args) {
    Map<Osoba, List<? extends Ljubimci>> ljudiSLjubimcima = Nova.map();
    // Ostatak koda je isti...
  }
} ///:~
```

Mada je ovo zanimljiv primer zaključivanja o tipu argumenta, teško je reći koliko je to zapravo korisno. Osoba koja čita kôd mora da analizira i shvati ovu dodatnu biblioteku i njene posledice, pa je možda jednako produktivno ostaviti prvobitnu definiciju (mada ima mnogo ponavljanja) – da ironija bude veća, jednostavnosti radi. Međutim, kada bi u standardnu Javinu biblioteku bila dodata uslužna klasa poput gornje **Nova.java**, bilo bi pametno koristiti je.

Zaključivanje o tipu argumenta funkcioniše isključivo za dodeljivanje. Ukoliko rezultat poziva metode kao što je **Nova.map( )** prosledite kao argument drugoj metodi, prevodilac *neće* pokušati da zaključi koji je tip argumenta. Poziv te metode on će tretirati kao da je njena povratna vrednost dodeljena promenljivoj tipa **Object**. Evo jednog takvog (neuspešnog) primera:

```
//: genericki/GraniceZakljucivanja.java
import podaciotipu.ljubimci.*;
import java.util.*;
public class GraniceZakljucivanja {
```

```
  static void
  f(Map<Osoba, List<? extends Ljubimci>> ljudiSLjubimcima) {}
  public static void main(String[] args) {
    // f(Nova.map()); // Neće biti prevedeno
  }
} ///:~
```

Vežba 11: (1) Ispitajte klasu **Nova.java** pravljenjem sopstvenih klasa. Dokažite da **Nova** s njima radi ispravno.

Eksplicitno zadavanje tipa

Tip generičke metode možete zadati eksplicitno, iako se to retko koristi. To se radi tako što se tip upiše unutar znakova manje od i veće od, iza tačke i neposredno ispred imena metode. Kada iz iste klase pozivate metodu, morate pisati **this** ispred tačke, a kada radite sa statičnim metodama, ispred tačke morate pisati ime klase. Problem prikazan u programu **GraniceZakljucivanja.java** možemo rešiti takvom sintaksom:

```
//: generacki/IzricitoZadavanjeTipa.java
import podaciotipu.ljubimci.*;
import java.util.*;
import net.mindview.util.*;

public class IzricitoZadavanjeTipa {
  static void f(Map<Osoba, List<Ljubimci>> ljudiSLjubimcima) {}
  public static void main(String[] args) {
    f(Nova.<Osoba, List<Ljubimci>>map());
  }
} ///:~
```

Naravno, time se gubi prednost korišćenja klase **Nova** koja smanjuje količinu pisanja, ali je dodatna sintaksa potrebna samo kada ne pišete naredbe dodeljivanja.

Vežba 12: (1) Ponovite prethodnu vežbu uz eksplicitno zadavanje tipova.

Argumenti promenljive dužine i generičke metode

Generičke metode i promenljive liste argumenata lepo rade zajedno:

```
//: generacki/GenerickiArgumentiPromenljiveDuzine.java
import java.util.*;

public class GenerickiArgumentiPromenljiveDuzine {
  public static <T> List<T> napraviListu(T... args) {
    List<T> rezultat = new ArrayList<T>();
    for(T stavka : args)
      rezultat.add(stavka);
    return rezultat;
  }
```

```
  public static void main(String[] args) {
    List<String> ls = napraviListu("A");
    System.out.println(ls);
    ls = napraviListu("A", "B", "C");
    System.out.println(ls);
    ls = napraviListu("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
} /* Ispis:
[A]
[A, B, C]
[, A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y,
Z]
*///:~
```

Ovde prikazana metoda **napraviListu( )** ima istu funkcionalnost kao metoda **java.util.Arrays.asList( )** iz standardne biblioteke.

Generička metoda za upotrebu s **Generator**ima

Za popunjavanje kontejnera (podtipova klase **Collection**) podesno je upotrebiti generator. Ovu operaciju je umesno uopštiti:

```
//: generici/Generatori.java
// Uslužna metoda za korišćenje s Generatorima.
import generici.kafa.*;
import java.util.*;
import net.mindview.util.*;

public class Generatori {
  public static <T> Collection<T>
  popuni(Collection<T> kntnr, Generator<T> gen, int n) {
    for(int i = 0; i < n; i++)
      kntnr.add(gen.sledeci());
    return kntnr;
  }
  public static void main(String[] args) {
    Collection<Kafa> kafa = popuni(
      new ArrayList<Kafa>(), new GeneratorKafe(), 4);
    for(Kafa c : kafa)
      System.out.println(c);
    Collection<Integer> fbrojevi = popuni(
      new ArrayList<Integer>(), new Fibonacci(), 12);
    for(int i : fbrojevi)
      System.out.print(i + ", ");
  }
} /* Ispis:
Americano 0
SMlekom 1
```

```
Americano 2
Moka 3
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
*///:~
```

Obratite pažnju na to da se metoda **popuni( )** može transparentno primeniti i na kontejner **Kafa** i na generator **Integer**.

Vežba 13: (4) Preklopite metodu **popuni( )** tako da argumenti i povratni tipovi redom budu **List**, **Queue** i **Set**, podtipovi klase **Collection**. Na taj način nećete izgubiti tip kontejnera. Možete li preklapanjem učiniti da se **List** i **LinkedList** razlikuju?

Generator opšte namene

Ova klasa pravi **Generator** za svaku klasu koja ima podrazumevani konstruktor. Da bi se pisalo manje koda, klasa obuhvata i generičku metodu koja pravi **ElementarniGenerator**:

```
//: net/mindview/util/ElementarniGenerator.java
// Automatski pravi Generator date klase
// koja ima podrazumevani konstruktor (bez argumenata).
package net.mindview.util;

public class ElementarniGenerator<T> implements Generator<T> {
  private Class<T> tip;
  public ElementarniGenerator(Class<T> tip){ this.tip = tip; }
  public T sledeci() {
    try {
      // Pretpostavlja da je tip javna klasa:
      return tip.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  // Pravi podrazumevani generator kada joj se da leksema tipa:
  public static <T> Generator<T> napravi(Class<T> tip) {
    return new ElementarniGenerator<T>(tip);
  }
} ///:~
```

Ova klasa predstavlja elementarnu realizaciju koja pravi objekte klase koja je (1) javna (pošto je **ElementarniGenerator** u zasebnom paketu, klasa o kojoj se radi mora imati javni, a ne samo paketni pristup), i (2) ima podrazumevani konstruktor (onaj koji ne prima argumente). Da biste napravili jedan od objekata tipa **ElementarniGenerator**, pozivate metodu **napravi( )** i prosleđujete joj leksemu (engl. *token*) tipa koji hoćete da generišete. Generička metoda **napravi( )** omogućuje da kažete **ElementarniGenerator.napravi(MojTip.class)** umesto **new ElementarniGenerator<MojTip>(MojTip.class)**, što bi bilo ružnije.

Na primer, ova jednostavna klasa ima podrazumevani konstruktor:

```
//: genericki/PrebrojaniObjekat.java

public class PrebrojaniObjekat {
  private static long brojac = 0;
  private final long id = brojac++;
  public long id() { return id; }
  public String toString() { return "PrebrojaniObjekat " + id;}
} ///:~
```

Klasa **PrebrojaniObjekat** prati koliko je sopstvenih primeraka napravila i to prijavljuje svojom metodom **toString( )**.

Pomoću klase **ElementarniGenerator** lako je napraviti **Generator** za **PrebrojaniObjekat**:

```
//: genericki/PrimerElementarnogGeneratora.java
import net.mindview.util.*;

public class PrimerElementarnogGeneratora {
  public static void main(String[] args) {
    Generator<PrebrojaniObjekat> gen =
      ElementarniGenerator.napravi(PrebrojaniObjekat.class);
    for(int i = 0; i < 5; i++)
      System.out.println(gen.sledeci());
  }
} /* Ispis:
PrebrojaniObjekat 0
PrebrojaniObjekat 1
PrebrojaniObjekat 2
PrebrojaniObjekat 3
PrebrojaniObjekat 4
*///:~
```

Iz ovoga vidite koliko generička metoda smanjuje količinu pisanja kada se pravi objekat **Generator**. Javin generički mehanizam vas primorava da ipak prosledite objekat tipa **Class**, pa ga onda možete upotrebiti i za zaključivanje o tipu argumenta u metodi **napravi( )**.

Vežba 14: (1) Izmenite **PrimerElementarnogGeneratora.java** tako da se **Generator** pravi eksplicitno (tj. umesto generičke metode **napravi( )** upotrebite eksplicitan konstruktor).

Pojednostavljenje upotrebe n-torki

Zaključivanje o tipu argumenta zajedno sa uvozom statičnih metoda omogućuje da se prethodno prikazane n-torke prerade u biblioteku opštije namene. Ovde ćemo n-torke praviti preklopljenom statičnom metodom:

```
//: net/mindview/util/N_torka.java
// Biblioteka n-torki u kojoj se upotrebljava
// zaključivanje o tipu argumenta.
package net.mindview.util;
```

```
public class N_torka {
  public static <A,B> Dvojka<A,B> n_torka(A a, B b) {
    return new Dvojka<A,B>(a, b);
  }
  public static <A,B,C> Trojka<A,B,C>
  n_torka(A a, B b, C c) {
    return new Trojka<A,B,C>(a, b, c);
  }
  public static <A,B,C,D> Cetvorka<A,B,C,D>
  n_torka(A a, B b, C c, D d) {
    return new Cetvorka<A,B,C,D>(a, b, c, d);
  }
  public static <A,B,C,D,E>
  Petorka<A,B,C,D,E> n_torka(A a, B b, C c, D d, E e) {
    return new Petorka<A,B,C,D,E>(a, b, c, d, e);
  }
} ///:~
```

Da bismo ispitali program **N_torka.java**, prilagodićemo program **IspitivanjeEntorki.java**:

```
//: genericki/IspitivanjeEntorki2.java
import net.mindview.util.*;
import static net.mindview.util.N_torka.*;

public class IspitivanjeEntorki2 {
  static Dvojka<String,Integer> f() {
    return n_torka("zdravo", 47);
  }
  static Dvojka f2() { return n_torka("zdravo", 47); }
  static Trojka<Amfibija,String,Integer> g() {
    return n_torka(new Amfibija(), "zdravo", 47);
  }
  static
  Cetvorka<Vozilo,Amfibija,String,Integer> h() {
    return n_torka(new Vozilo(), new Amfibija(), "zdravo", 47);
  }
  static
  Petorka<Vozilo,Amfibija,String,Integer,Double> k() {
    return n_torka(new Vozilo(), new Amfibija(),
      "zdravo", 47, 11.1);
  }
  public static void main(String[] args) {
    Dvojka<String,Integer> iesi = f();
    System.out.println(iesi);
    System.out.println(f2());
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
```

```
} /* Ispis: (80% podudaranja)
(zdravo, 47)
(zdravo, 47)
(Amfibija@7d772e, zdravo, 47)
(Vozilo@757aef, Amfibija@d9f9c3, zdravo, 47)
(Vozilo@1a46e30, Amfibija@3e25a5, zdravo, 47, 11.1)
*///:-
```

Vodite računa o tome da **f( )** vraća parametrizovan objekat tipa **Dvojka**, dok **f2( )** vraća neparametrizovan objekat tipa **Dvojka**. U ovom slučaju prevodilac ne upozorava na **f2( )**, zato što se parametrizovana vrednost ne upotrebljava na parametrizovan način; na neki način, ona se „svodi naviše" na neparametrizovan tip **Dvojka**. Međutim, kada biste pokušali da rezultat **f2( )** uhvatite u parametrizovan tip **Dvojka**, prevodilac bi dao upozorenje.

Vežba 15: (1) Proverite prethodnu tvrdnju.

Vežba 16: (2) Dodajte **Sestorku** programu **N_torka.java** i ispitajte ga programom **IspitivanjeEntorki2.java**.

Uslužna metoda za **Set**

Kao još jedan primer upotrebe generičkih metoda, razmotrićemo matematičke odnose koji se mogu izraziti skupovima (objektima tipa **Set**). Njih je podesno definisati generičkim metodama koje se mogu upotrebljavati sa svim različitim tipovima:

```
//: net/mindview/util/Skupovi.java
package net.mindview.util;
import java.util.*;

public class Skupovi {
  public static <T> Set<T> unija(Set<T> a, Set<T> b) {
    Set<T> rezultat = new HashSet<T>(a);
    rezultat.addAll(b);
    return rezultat;
  }
  public static <T>
  Set<T> presek(Set<T> a, Set<T> b) {
    Set<T> rezultat = new HashSet<T>(a);
    rezultat.retainAll(b);
    return rezultat;
  }
  // Oduzmi podskup od nadskupa:
  public static <T> Set<T>
  razlika(Set<T> nadskup, Set<T> podskup) {
    Set<T> rezultat = new HashSet<T>(nadskup);
    rezultat.removeAll(podskup);
    return rezultat;
  }
```

```
  // Refleksivno--nije sve u preseku:
  public static <T> Set<T> komplement(Set<T> a, Set<T> b) {
    return razlika(unija(a, b), presek(a, b));
  }
} ///:~
```

Prve tri metode dupliraju prvi argument kopiranjem njegovih referenci u nov objekat tipa **HashSet**, pa se argumenti **Skupovi** ne menjaju direktno. Povratna vrednost je nov objekat tipa **Set**.

Ove četiri metode predstavljaju matematičke operacije sa skupovima: **unija( )** vraća **Set** koji sadrži kombinaciju dva argumenta, **presek( )** vraća **Set** koji sadrži zajedničke elemente dvaju argumenata, **razlika( )** oduzima elemente skupa **podskup** od elemenata skupa **nadskup**, a **komplement( )** vraća **Set** svih elemenata koji nisu u preseku. Kao jednostavan primer upotrebe ovih metoda, sledeći **enum** (nabrojani tip) sadrži imena raznih vodenih boja:

```
//: genericki/vodeneboje/Vodeneboje.java
package genericki.vodeneboje;

public enum Vodeneboje {
  ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
  BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
  CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
  COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
  SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
  BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} ///:~
```

Odgovara nam (kako ne bismo morali da navodimo puna imena) da prethodnu listu nabrajanja statično uvezemo u sledeći primer. U njemu se upotrebljava **EnumSet**, alatka Jave SE5 za lako pravljenje skupova (objekata tipa **Set**) od nabrojanih tipova (**enum**). (O klasi **EnumSet** više pričamo u poglavlju *Nabrojani tipovi*.) Ovde se statičnoj metodi **EnumSet.range( )** daju prvi i poslednji element opsega (engl. *range*) koji treba napraviti u rezultujućem skupu:

```
//: genericki/SkupoviVodenihBoja.java
import genericki.vodeneboje.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Skupovi.*;
import static genericki.vodeneboje.Vodeneboje.*;

public class SkupoviVodenihBoja {
  public static void main(String[] args) {
    Set<Vodeneboje> skup1 =
      EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
    Set<Vodeneboje> skup2 =
      EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
```

```
    print("skup1: " + skup1);
    print("skup2: " + skup2);
    print("unija(skup1, skup2): " + unija(skup1, skup2));
    Set<Vodeneboje> podskup = presek(skup1, skup2);
    print("presek(skup1, skup2): " + podskup);
    print("razlika(skup1, podskup): " +
      razlika(skup1, podskup));
    print("razlika(skup2, podskup): " +
      razlika(skup2, podskup));
    print("komplement(skup1, skup2): " +
      komplement(skup1, skup2));
  }
} /* Ispis: (primer)
skup1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
PERMANENT_GREEN, VIRIDIAN_HUE]
skup2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA,
RAW_UMBER, BURNT_UMBER]
unija(skup1, skup2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE,
PERMANENT_GREEN, BURNT_UMBER, COBALT_BLUE_HUE, VIOLET, BRILLIANT_RED,
RAW_UMBER, ULTRAMARINE, BURNT_SIENNA, CRIMSON, CERULEAN_BLUE_HUE,
PHTHALO_BLUE, MAGENTA, VIRIDIAN_HUE]
presek(skup1, skup2): [ULTRAMARINE, PERMANENT_GREEN, COBALT_BLUE_HUE,
PHTHALO_BLUE, CERULEAN_BLUE_HUE, VIRIDIAN_HUE]
razlika(skup1, podskup): [ROSE_MADDER, CRIMSON, VIOLET, MAGENTA,
BRILLIANT_RED]
razlika(skup2, podskup): [RAW_UMBER, SAP_GREEN, YELLOW_OCHRE,
BURNT_SIENNA, BURNT_UMBER]
komplement(skup1, skup2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE,
BURNT_UMBER, VIOLET, BRILLIANT_RED, RAW_UMBER, BURNT_SIENNA, CRIMSON,
MAGENTA]
*///:~
```

Rezultate svake operacije vidite u rezultatu programa.

U narednom primeru upotrebljena je metoda **Skupovi.razlika**( ) da bi se pokazale razlike između raznih **Collection** i **Map** klasa u paketu **java.util**:

```
//: net/mindview/util/RazlikeKontejnerskihMetoda.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class RazlikeKontejnerskihMetoda {
  static Set<String> skupMetoda(Class<?> tip) {
    Set<String> rezultat = new TreeSet<String>();
    for(Method m : tip.getMethods())
      rezultat.add(m.getName());
    return rezultat;
  }
```

```
  static void interfejsi(Class<?> tip) {
    System.out.print("Interfejsi u " +
      tip.getSimpleName() + ": ");
    List<String> rezultat = new ArrayList<String>();
    for(Class<?> c : tip.getInterfaces())
      rezultat.add(c.getSimpleName());
    System.out.println(rezultat);
  }
  static Set<String> objekat = skupMetoda(Object.class);
  static { objekat.add("klon"); }
  static void
  razlika(Class<?> nadskup, Class<?> podskup) {
    System.out.print(nadskup.getSimpleName() +
      " extends " + podskup.getSimpleName() + ", dodaje: ");
    Set<String> komp = Skupovi.razlika(
      skupMetoda(nadskup), skupMetoda(podskup));
    komp.removeAll(objekat); // Ne prikazuj metode klase 'Object'
    System.out.println(komp);
    interfejsi(nadskup);
  }
  public static void main(String[] args) {
    System.out.println("Collection: " +
      skupMetoda(Collection.class));
    interfaces(Collection.class);
    razlika(Set.class, Collection.class);
    razlika(HashSet.class, Set.class);
    razlika(LinkedHashSet.class, HashSet.class);
    razlika(TreeSet.class, Set.class);
    razlika(List.class, Collection.class);
    razlika(ArrayList.class, List.class);
    razlika(LinkedList.class, List.class);
    razlika(Queue.class, Collection.class);
    razlika(PriorityQueue.class, Queue.class);
    System.out.println("Map: " + skupMetoda(Map.class));
    razlika(HashMap.class, Map.class);
    razlika(LinkedHashMap.class, HashMap.class);
    razlika(SortedMap.class, Map.class);
    razlika(TreeMap.class, Map.class);
  }
} ///:~
```

Rezultat ovog programa bio je upotrebljen u „Sažetku" poglavlja *Čuvanje objekata.*

Vežba 17: (4) Proučite JDK dokumentaciju za **EnumSet**. Videćete da je definisana i metoda **clone**( ). Međutim, njome ne možete klonirati referencu interfejsa **Set** prosleđenu u programu **Skupovi.java**. Umete li da izmenite program **Skupovi.java** tako da radi i opšti slučaj interfejsa **Set** (kao što je prikazano) i specijalni slučaj interfejsa **EnumSet** (upotrebite **clone**( ) umesto da pravite nov **HashSet**)?

Anonimne unutrašnje klase

Generički tipovi se mogu koristiti i sa unutrašnjim klasama i sa anonimnim unutrašnjim klasama. Evo primera koji realizuje interfejs **Generator** pomoću anonimnih unutrašnjih klasa:

```
//: generics/SluzbenikuBanci.java
// Pojednostavljena simulacija službenika u banci.
import java.util.*;
import net.mindview.util.*;

class Klijent {
  private static long brojac = 1;
  private final long id = brojac++;
  private Klijent() {}
  public String toString() { return "Klijent " + id; }
  // Metoda za pravljenje objekata tipa Generator:
  public static Generator<Klijent> generator() {
    return new Generator<Klijent>() {
      public Klijent sledeci() { return new Klijent(); }
    };
  }
}

class Sluzbenik {
  private static long brojac = 1;
  private final long id = brojac++;
  private Sluzbenik() {}
  public String toString() { return "Sluzbenik " + id; }
  // Samo jedan objekat tipa Generator:
  public static Generator<Sluzbenik> generator =
    new Generator<Sluzbenik>() {
      public Sluzbenik sledeci() { return new Sluzbenik(); }
    };
}

public class SluzbenikuBanci {
  public static void usluzi(Sluzbenik t, Klijent c) {
    System.out.println(t + " uslužuje klijenta " + c);
  }
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    Queue<Klijent> redZaCekanje = new LinkedList<Klijent>();
    Generatori.popuni(redZaCekanje, Klijent.generator(), 15);
    List<Sluzbenik> sluzbenici = new ArrayList<Sluzbenik>();
    Generatori.popuni(sluzbenici, Sluzbenik.generator, 4);
    for(Klijent c : redZaCekanje)
      usluzi(sluzbenici.get(slucajan.nextInt(sluzbenici.size())), c);
  }
```

```
} /* Ispis:
Sluzbenik 3 uslužuje klijenta Klijent 1
Sluzbenik 2 uslužuje klijenta Klijent 2
Sluzbenik 3 uslužuje klijenta Klijent 3
Sluzbenik 1 uslužuje klijenta Klijent 4
Sluzbenik 1 uslužuje klijenta Klijent 5
Sluzbenik 3 uslužuje klijenta Klijent 6
Sluzbenik 1 uslužuje klijenta Klijent 7
Sluzbenik 2 uslužuje klijenta Klijent 8
Sluzbenik 3 uslužuje klijenta Klijent 9
Sluzbenik 3 uslužuje klijenta Klijent 10
Sluzbenik 2 uslužuje klijenta Klijent 11
Sluzbenik 4 uslužuje klijenta Klijent 12
Sluzbenik 2 uslužuje klijenta Klijent 13
Sluzbenik 1 uslužuje klijenta Klijent 14
Sluzbenik 1 uslužuje klijenta Klijent 15
*///:~
```

I **Klijent** i **Sluzbenik** imaju privatne konstruktore, čime vas primoravaju da koristite objekte tipa **Generator**. **Klijent** ima metodu **generator( )** koja pravi nov objekat tipa **Generator<Klijent>** kad god je pozovete. Možda vam neće biti potrebni svi ti **Generator**i, a **Sluzbenik** pravi samo jedan javni **generator**. Oba pristupa možete videti na delu u metodama **popuni( )** unutar metode **main( )**.

Pošto su statični i metoda **generator( )** objekta **Klijent** i objekat tipa **Generator** u klasi **Sluzbenik**, oni ne mogu biti deo interfejsa, pa ovaj idiom ne možete da uopštite pomoću generičkih tipova. Uprkos tome, on radi prilično dobro s metodom **popuni( )**.

Ostale verzije problema čekanja u redu razmotrićemo u poglavlju *Paralelno izvršavanje*.

Vežba 18: (3) Na osnovu programa **SluzbenikuBanci.java**, napravite **Okean** gde postoje **VelikaRiba** i **MalaRiba**, i prva jede drugu.

Pravljenje složenih modela

Generički tipovi omogućuju jednostavno i bezbedno pravljenje složenih modela. Na primer, lako možemo da napravimo **List**u n-torki:

```
//: generics/ListaN_torki.java
// Kombinovanje generičkih tipova da bi se dobili
// složeni generički tipovi.
import java.util.*;
import net.mindview.util.*;

public class ListaN_torki<A,B,C,D>
extends ArrayList<Cetvorka<A,B,C,D>> {
  public static void main(String[] args) {
    ListaN_torki<Vozilo, Amfibija, String, Integer> tl =
      new ListaN_torki<Vozilo, Amfibija, String, Integer>();
    tl.add(IspitivanjeEntorki.h());
    tl.add(IspitivanjeEntorki.h());
```

```
    for(Cetvorka<Vozilo,Amfibija,String,Integer> i: tl)
      System.out.println(i);
  }
} /* Ispis: (75% podudaranja)
(Vozilo@11b86e7, Amfibija@35ce36, zdravo, 47)
(Vozilo@757aef, Amfibija@d9f9c3, zdravo, 47)
*///:~
```

Jeste da se moralo prilično pisati (naročito u pravljenju iteratora), ali smo dobili moćnu strukturu podataka u kratkom programu.

Evo još jednog primera koji pokazuje koliko je jednostavno pravljenje složenih modela pomoću generičkih tipova. Iako je svaka klasa napravljena kao zaseban blok, celina ima mnogo delova. U ovom slučaju, model je prodavnica s pasažima, policama i proizvodima:

```
//: generici/Prodavnica.java
// Pravljenje složenog modela pomoću generičkih kontejnera.
import java.util.*;
import net.mindview.util.*;

class Proizvod {
  private final int id;
  private String opis;
  private double cena;
  public Proizvod(int IDbroj, String ops, double cena){
    id = IDbroj;
    opis = ops;
    this.cena = cena;
    System.out.println(toString());
  }
  public String toString() {
    return id + ": " + opis + ", cena: $" + cena;
  }
  public void promenaCene(double change) {
    cena += change;
  }
  public static Generator<Proizvod> generator =
    new Generator<Proizvod>() {
      private Random slucajan = new Random(47);
      public Proizvod sledeci() {
        return new Proizvod(slucajan.nextInt(1000), "Test",
          Math.round(slucajan.nextDouble() * 1000.0) + 0.99);
      }
    };
}

class Polica extends ArrayList<Proizvod> {
  public Polica(int nProizvoda) {
    Generatori.popuni(this, Proizvod.generator, nProizvoda);
  }
}
```

```
class Pasaz extends ArrayList<Polica> {
  public Pasaz(int nPolica, int nProizvoda) {
    for(int i = 0; i < nPolica; i++)
      add(new Polica(nProizvoda));
  }
}

class Blagajna {}
class Kancelarija {}

public class Store extends ArrayList<Pasaz> {
  private ArrayList<Blagajna> blagajne =
    new ArrayList<Blagajna>();
  private Kancelarija kancelarija = new Kancelarija();
  public Store(int nPasaza, int nPolica, int nProizvoda) {
    for(int i = 0; i < nPasaza; i++)
      add(new Pasaz(nPolica, nProizvoda));
  }
  public String toString() {
    StringBuilder rezultat = new StringBuilder();
    for(Pasaz a : this)
      for(Polica s : a)
        for(Proizvod p : s) {
          rezultat.append("\n");
          rezultat.append(p);
        }
    return rezultat.toString();
  }
  public static void main(String[] args) {
    System.out.println(new Prodavnica(14, 5, 10));
  }
} /* Ispis:
258: Test, cena: $400.99
861: Test, cena: $160.99
868: Test, cena: $417.99
207: Test, cena: $268.99
551: Test, cena: $114.99
278: Test, cena: $804.99
520: Test, cena: $554.99
140: Test, cena: $530.99
...
*///:-
```

Kao što vidite u metodi **Prodavnica.toString( )**, dobili smo mnogo slojeva kontejnera koji su uprkos tome upravljivi i omogućuju bezbedan rad s tipovima. Impresivno je to što sastavljanje takvog modela nije teško.

Vežba 19: (2) Na osnovu programa **Prodavnica.java**, napravite model teretnog broda podeljenog na skladišta.

Tajanstveno brisanje

Kako budete sve temeljnije upoznavali generičke tipove, nailazićete na stvari koje vam spočetka neće biti logične. Na primer, iako se može reći **ArrayList.class**, ne može se reći **ArrayList<Integer>.class**. I pogledajte ovo:

```
//: generics/EkvivalentnostTipovaZbogBrisanja.java
import java.util.*;

public class EkvivalentnostTipovaZbogBrisanja {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
  }
} /* Ispis:
true
*///:~
```

Lako je dokazati da su **ArrayList<String>** i **ArrayList<Integer>** različiti tipovi. Različiti tipovi se ponašaju različito, i ako pokušate, primera radi, da stavite **Integer** u **ArrayList<String>**, dobićete različito ponašanje (to neće uspeti) nego kada **Integer** stavite u **ArrayList<Integer>** (to će uspeti). Pa ipak, gornji program kaže da su oba isti tip.

Evo primera koji će vas zbuniti još više:

```
//: generics/IzgubljeneInformacije.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Kvark<Q> {}
class Cestica<POLOZAJ,MOMENAT> {}

public class IzgubljeneInformacije {
  public static void main(String[] args) {
    List<Frob> lista = new ArrayList<Frob>();
    Map<Frob,Fnorkle> mapa = new HashMap<Frob,Fnorkle>();
    Kvark<Fnorkle> kvark = new Kvark<Fnorkle>();
    Cestica<Long,Double> p = new Cestica<Long,Double>();
    System.out.println(Arrays.toString(
      lista.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      mapa.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      kvark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }
```

```
} /* Ispis:
[E]
[K, V]
[Q]
[POLOZAJ, MOMENAT]
*///:~
```

U JDK dokumentaciji piše da metoda **Class.getTypeParameters( )** „vraća niz objekata tipa **TypeVariable** koji predstavljaju promenljive tipa deklarisane u generičkoj deklaraciji ...". Iz toga bi sledilo da je moguće otkriti parametarske tipove. Međutim, kao što vidite iz rezultata programa, otkrili smo samo identifikatore koji čuvaju mesta parametara, što i nije preterano zanimljivo.

Prava istina glasi:

U generičkom kodu uopšte nisu dostupne informacije o tipovima generičkih parametara.

Dakle, možete saznati stvari kao što su identifikator parametra tipa i ograničenja generičkog tipa – ali ne možete otkriti stvarne parametre tipova upotrebljene za pravljenje određene instance. Ta činjenica naročito smeta onima koji su koristili C++ i predstavlja osnovni problem pri radu s Javinim generičkim tipovima.

Javini generički tipovi realizuju se uz *brisanje* (engl. *erasure*). To znači sledeće: kada upotrebite generički tip, brišu se sve specifične informacije o tipu. Unutar generičkog tipa možete znati samo da upotrebljavate objekat. Stoga **List<String>** i **List<Integer>** u vreme izvršavanja i *jesu*, zapravo, isti tip. Oba oblika bivaju „obrisana" do njihovog *sirovog tipa*, a to je **List**. Tokom učenja Javinih generičkih tipova, gotovo da će najteže biti shvatiti brisanje i kako se s njim ophoditi. Time ćemo se baviti u narednom odeljku.

Pristup u C++-u

Pogledajmo primer iz C++-a u kojem su upotrebljeni *šabloni* (engl. *templates*). Videćete da je sintaksa za parametrizovane tipove potpuno jednaka, pošto je Java napravljena na osnovu C++-a:

```
//: generici/Sabloni.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
  T obj;
public:
  Manipulator(T x) { obj = x; }
  void manipulisi() { obj.f(); }
};

class ImaF {
public:
  void f() { cout << "ImaF::f()" << endl; }
};
```

```
int main() {
  ImaF imaf;
  Manipulator<ImaF> manipulator(imaf);
  manipulator.manipulisi();
} /* Ispis:
ImaF::f()
///:~
```

Klasa **Manipulator** sadrži objekat tipa **T**. Zanimljiva je metoda **manipulisi( )** koja poziva metodu **f( )** za **obj**. Kako ona može znati da metoda **f( )** postoji za parametar tipa **T**? Prevodilac C++-a proverava tipove kada pravite primerak šablona, pa u trenutku pravljenja klase **Manipulator<ImaF>** vidi da **ImaF** ima metodu **f( )**. Da je nema, dobili biste grešku u vreme prevođenja, i bezbednost tipova bi bila očuvana.

Pisanje ovakvog koda u C++-u je jednostavno, zato što kôd šablona zna tip parametara svog šablona u trenutku kada treba da ga napravi. Javini generički tipovi su drugačiji. Napisali smo **ImaF** u Javi:

```
//: generic/ImaF.java

public class ImaF {
  public void f() { System.out.println("ImaF.f()"); }
} ///:~
```

Ako i ostatak primera napišemo na Javi, on neće moći da se prevede:

```
//: generic/Manipulacija.java
// {CompileTimeError} (Ne može da se prevede)

class Manipulator<T> {
  private T obj;
  public Manipulator(T x) { obj = x; }
  // Greška: cannot find symbol: method f():
  public void manipulisi() { obj.f(); }
}

public class Manipulacija {
  public static void main(String[] args) {
    ImaF imaf = new ImaF();
    Manipulator<ImaF> manipulator =
      new Manipulator<ImaF>(imaf);
    manipulator.manipulisi();
  }
} ///:~
```

Zahtev da metoda **manipulisi( )** mora moći da poziva **f( )** za **obj**, Javin prevodilac zbog brisanja ne može da dovede u vezu sa činjenicom da **ImaF** ima metodu **f( )**. Da bismo pozvali **f( )**, moramo pomoći generičkoj klasi dajući joj *granicu* (engl. *bound*) koja prevodiocu kazuje da prihvata samo tipove usaglašene s tom granicom. Za to ćemo ponovo upotrebiti rezervisanu reč **extends**. Sledeći program će se uspešno prevesti, upravo zbog granice:

```
//: generics/Manipulator2.java

class Manipulator2<T extends ImaF> {
  private T obj;
  public Manipulator2(T x) { obj = x; }
  public void manipulisi() { obj.f(); }
} ///:~
```

Granica <**T extends ImaF**> kazuje da **T** mora biti tipa **ImaF** ili njegovih podtipova. Ako je to istina, onda je bezbedno pozvati **f( )** za **obj**.

Kažemo da se parametar generičkog tipa *briše do svoje prve granice* (granica može biti više, što ćete videti u nastavku) ili da se *briše parametar tipa*. Prevodilac zapravo zamenjuje parametar tipa onim što je obrisano, pa u gornjem slučaju **T** brisanjem postaje **ImaF**, što je isto kao da se **T** u telu klase zameni sa **ImaF**.

Možda ćete ispravno primetiti da u programu **Manipulator2.java**, generički tipovi ne rade ništa. Mogli ste i sami da obavite brisanje i da napravite klasu bez generičkih tipova:

```
//: generics/Manipulator3.java

class Manipulator3 {
  private ImaF obj;
  public Manipulator3(ImaF x) { obj = x; }
  public void manipulisi() { obj.f(); }
} ///:~
```

Time smo došli do važnog zaključka: generički tipovi su korisni samo kada hoćete da upotrebite parametre tipa koji su opštiji od određenog konkretnog tipa (i njegovih podtipova) – dakle, kada hoćete da kôd radi s različitim klasama. Zato su, u korisnom generičkom kodu, parametri tipa i njihova primena obično složeniji od proste zamene klasa. Međutim, to ne znači da je neispravno sve što je oblika <**T extends ImaF**>. Na primer, ako klasa ima metodu koja vraća **T**, onda je generički kôd koristan jer će vratiti tačan tip:

```
//: generics/VracanjeGenerickogTipa.java

class VracanjeGenerickogTipa<T extends ImaF> {
  private T obj;
  public VracanjeGenerickogTipa(T x) { obj = x; }
  public T get() { return obj; }
} ///:~
```

Da biste prosudili da li je korišćenje generičkog koda opravdano, morate pročitati sav kôd i proceniti da li je dovoljno složen.

Granice ćemo detaljnije razmotriti u nastavku poglavlja.

Vežba 20: (1) Napravite interfejs s dve metode i klasu koja ga realizuje i dodaje još jednu metodu. U drugoj klasi, napravite generičku metodu čiji je argument tipa ograničen interfejsom i pokažite da se metode interfejsa mogu pozivati iz te generičke metode. U metodi **main( )**, generičkoj metodi prosledite primerak klase koja je obavila realizaciju.

Migracijska kompatibilnost

Da bismo sprečili moguće zabune u vezi s brisanjem, morate nedvosmisleno shvatiti da brisanje *nije* obeležje jezika. Radi se o kompromisu u realizaciji Javinih generičkih tipova, potrebnom zato što je jezik na početku bio napravljen bez njih. Taj kompromis će vam smetati, pa se morate navići na njega i shvatiti zašto je učinjen.

Da su generički tipovi bili deo Jave od verzije 1.0, ne bi bili realizovani pomoću brisanja – koristila bi se *konkretizacija* za pamćenje parametara tipa kao prvorazrednih entiteta, pa biste s njima mogli da obavljate jezičke i refleksivne operacije zasnovane na tipovima. Videćete u nastavku poglavlja da brisanje smanjuje „opštost" generičkih tipova. Oni u Javi jesu korisni, ali ne toliko koliko bi mogli biti, a razlog je upravo brisanje.

U realizaciji zasnovanoj na brisanju, generički tipovi se tretiraju kao drugorazredni tipovi koji se ne mogu upotrebljavati u nekim važnim kontekstima. Generički tipovi postoje samo tokom statičke provere tipova. Nakon toga, svi generički tipovi u programu bivaju obrisani i zamenjeni nekom negeneričkom gornjom granicom. Na primer, anotacije tipa kao što je **List<T>** bivaju brisanjem svedene na **List**, a promenljive običnog tipa svedene na **Object**, ukoliko neka granica nije zadata.

Osnovna motivacija za brisanje jeste to što omogućuje korišćenje generičkih klijenata s negeneričkim bibliotekama i obrnuto. To se često naziva *migracijska kompatibilnost*. U idealnom svetu, sav kôd bi u istom trenutku postao generički. U stvarnosti, čak i da programeri pišu samo generički kôd, morali bi da vode računa o negeneričkim bibliotekama napisanim pre Jave SE5. Autori tih biblioteka možda nikada neće prepraviti svoj kôd tako da postane generički, ili će to možda uraditi polako.

Zato Javina realizacija generičkih tipova mora da podržava ne samo *vertikalnu kompatibilnost* – postojeći kôd i datoteke klasa moraju ostati legalne i značiti isto što su značile pre – nego i migracijsku kompatibilnost, da bi biblioteke mogle postati generičke kada to njima bude odgovaralo, i da tada ne bi srušile kôd i aplikacije koje od njih zavise. Kada su to sebi zadali kao cilj, projektanti Jave i razne grupe koje su radile na tom problemu odlučili su da je brisanje jedino ostvarivo rešenje. Brisanje omogućuje migraciju ka generičkim tipovima tako što dozvoljava postojanje negeneričkog koda zajedno s generičkim.

Na primer, recimo da određena aplikacija koristi dve biblioteke, **X** i **Y**, i da **Y** koristi biblioteku **Z**. Pošto je u međuvremenu objavljena Java SE5, tvorci te aplikacije i tih biblioteka verovatno će, jednog dana, odlučiti da pređu na generički kôd. Međutim, svaki od njih će imati drugačiju motivaciju i ograničenja u pogledu vremena tog prelaska. Da bi se postigla migracijska kompatibilnost, svaka biblioteka i aplikacija mora biti nezavisna od svih ostalih u odnosu na to da li koristi generički kôd. Stoga ne smeju imati mogućnost da otkriju da li druge biblioteke koriste ili ne koriste generički kôd. Zato dokaz da određena biblioteka koristi generički kôd mora biti „obrisan".

Bez neke vrste putanje za migraciju, sve biblioteke napisane tokom vremena bile su u opasnosti da budu odsečene od programera koji su odlučili da pređu na Javin generički kôd. Tvrdi se da su biblioteke deo jezika koji najviše utiče na produktivnost, pa toliki rizik nije bio prihvatljiv. Da li je brisanje bilo jedini ili najbolji put za migraciju, pokazaće vreme.

Problem s brisanjem

Dakle, osnovni razlog za brisanje jeste prelazak s negeneričkog na generički kôd i namera da se generički kôd uključi u jezik bez rušenja postojećih biblioteka. Brisanje omogućuje postojećem negeneričkom klijentskom kodu da nastavi da radi bez ikakve izmene, sve dok klijenti ne budu spremni da prerade kôd na generički. Ta motivacija je plemenita, zato što ne ruši odjednom sav postojeći kôd.

Cena brisanja je znatna. Generički tipovi se ne mogu upotrebljavati u operacijama koje u vreme izvršavanja izričito upućuju na tipove, kao što su eksplicitne konverzije tipova, operacije **instanceof** i izrazi **new**. Pošto se gube sve informacije o tipu parametara, kada pišete generički kôd morate se stalno podsećati da samo *izgleda* kao da imate informacije o tipu parametara. Dakle, kada pišete ovakvo parče koda:

```
class Nesto<T> {
  T promenljiva;
}
```

i uz to napravite primerak klase **Nesto**:

```
Nesto<Macka> f = new Nesto<Macka>();
```

čini se da kôd u klasi **Nesto** treba da zna kako sada radi sa objektom tipa **Macka**. Sintaksa vas jako navodi na pomisao da tip **T** biva zamenjen u celoj klasi. Ali to nije istina, i kad god pišete kôd za tu klasu, morate sebi reći: „Ne, to je samo **objekat**".

Pored toga, brisanje i migracijska kompatibilnost znače da se generički kôd ne koristi ni tamo gde biste to možda želeli:

```
//: generics/BrisanjeINasledjivanje.java

class GenerickaOsnovna<T> {
  private T element;
  public void set(T arg) { arg = element; }
  public T get() { return element; }
}

class Izvedena1<T> extends GenerickaOsnovna<T> {}

class Izvedena2 extends GenerickaOsnovna {} // Nema upozorenja

// class Izvedena3 extends GenerickaOsnovna<?> {}
// Čudna greška:
//   unexpected type found : ? (pronađen neočekivan tip)
//   required: class or interface without bounds
// (zahteva se: klasa ili interfejs bez granica)

public class BrisanjeINasledjivanje {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Izvedena2 d2 = new Izvedena2();
```

```
    Object obj = d2.get();
    d2.set(obj); // Ovde biste dobili upozorenje!
  }
} ///:~
```

Izvedena2 nasleđuje klasu **GenerickaOsnovna** bez generičkih parametara i prevodilac ne daje upozorenje. Upozorenja nema do poziva metode **set( )**.

Za isključivanje upozorenja Java ima anotaciju, onu koju vidite u listingu (ta anotacija nije bila podržana u ranijim verzijama Jave SE5):

```
@SuppressWarnings("unchecked")
```

Vodite računa o tome da smo ovu naredbu stavili neposredno pre metode koja generiše upozorenje, a ne pre cele klase. Kada isključujete upozorenje, bolje je da se usredsredite na što „uže" područje, da ne biste preširokim isključivanjem upozorenja slučajno sakrili neki pravi problem.

Greška koju proizvodi **Izvedena3** valjda znači da je prevodilac očekivao sirovu osnovnu klasu.

Dodajte ovome trud zbog rada s granicama u slučaju kada parametar tipa hoćete da tretirate kao nešto više od običnog objekta, i dobili ste mnogo više posla za mnogo manje rezultata nego što dobijate od parametrizovanih tipova u jezicima kao što su C++, Ada ili Eiffel. To ne znači da su ti jezici bolji od Jave za većinu programerskih zadataka, nego da su njihovi mehanizmi parametrizovanih tipova fleksibilniji i moćniji od Javinog.

Šta se zbiva na granicama

Zbog brisanja, smatram da aspekt generičkog koda koji najviše zbunjuje jeste činjenica da možete napisati ono što nema smisla. Na primer:

```
//: generics/TvoracNizova.java
import java.lang.reflect.*;
import java.util.*;

public class TvoracNizova<T> {
  private Class<T> vrsta;
  public TvoracNizova(Class<T> vrsta) { this.vrsta = vrsta; }
  @SuppressWarnings("unchecked")
  T[] napravi(int velicina) {
    return (T[])Array.newInstance(vrsta, velicina);
  }
  public static void main(String[] args) {
    TvoracNizova<String> tvoracStringova =
      new TvoracNizova<String>(String.class);
    String[] nizStringova = tvoracStringova.napravi(9);
    System.out.println(Arrays.toString(nizStringova));
  }
} /* Ispis:
[null, null, null, null, null, null, null, null, null]
*///:~
```

Iako je klasa **vrsta** napisana kao **Class<T>**, brisanje znači da će ona biti uskladištena samo kao **Class**, bez parametra. Zato, kada s njom nešto napravite, recimo neki niz, metoda **Array.newInstance( )** ne dobija informacije o tipu koje **vrsta** implicira; zbog toga ta metoda ne može dati rezultat specifičnog podtipa, te ga morate eksplicitno konvertovati, što proizvodi upozorenje koje ne možete da otklonite.

Imajte u vidu da je metoda **Array.newInstance( )** preporučena za pravljenje nizova u generičkom kodu.

Ukoliko umesto niza napravimo kontejner, stvari izgledaju drugačije:

```
//: generics/TvoracListi.java
import java.util.*;

public class TvoracListi<T> {
  List<T> napravi() { return new ArrayList<T>(); }
  public static void main(String[] args) {
    TvoracListi<String> tvoracStringova= new TvoracListi<String>();
    List<String> listaStringova = tvoracStringova.napravi();
  }
} ///:~
```

Prevodilac ne daje nikakvo upozorenje, iako (od brisanja) znamo da <T> u **new ArrayList<T>( )** unutar metode **napravi( )** biva uklonjen – u vreme izvršavanja unutar klase nema nikakvog <T>, pa on kao da ne znači ništa. Ukoliko dosledno takvom shvatanju, promenite izraz u **new ArrayList( )**, prevodilac će dati upozorenje.

Da li u ovom slučaju parametar tipa zaista ne znači ništa? Šta bi se desilo da na sledeći način u **list**u stavite neke objekte pre nego što je vratite:

```
//: generics/TvoracPopunjeneListe.java
import java.util.*;

public class TvoracPopunjeneListe<T> {
  List<T> napravi(T t, int n) {
    List<T> rezultat = new ArrayList<T>();
    for(int i = 0; i < n; i++)
      rezultat.add(t);
    return rezultat;
  }
  public static void main(String[] args) {
    TvoracPopunjeneListe<String> tvoracStringova =
      new TvoracPopunjeneListe<String>();
    List<String> lista = tvoracStringova.napravi("Zdravo", 4);
    System.out.println(lista);
  }
} /* Ispis:
[Zdravo, Zdravo, Zdravo, Zdravo]
*///:~
```

Iako prevodilac ne može ništa da zna o tipu **T** unutar metode **napravi( )**, ipak može da proveri – u vreme prevođenja – da li je ono što ste stavili u **rezultat** tipa T pa može biti

usklađeno sa **ArrayList<T>**. Dakle, iako brisanje uklanja informacije o stvarnom tipu unutar metode ili klase, prevodilac ipak može da obezbedi unutrašnju doslednost u načinu na koji je taj tip upotrebljen unutar metode, odnosno klase.

Pošto brisanje uklanja informacije o tipu iz tela metode, u vreme izvršavanja važne su *granice*: tačke gde objekti ulaze u metodu i izlaze iz nje. U tim tačkama prevodilac u vreme prevođenja proverava tipove i umeće kôd za njihove konverzije. Pogledajmo sledeći negenerički primer:

```
//: generics/JednostavnoSkladiste.java

public class JednostavnoSkladiste {
  private Object obj;
  public void set(Object obj) { this.obj = obj; }
  public Object get() { return obj; }
  public static void main(String[] args) {
    JednostavnoSkladiste Skladiste = new JednostavnoSkladiste();
    Skladiste.set("Stavka");
    String s = (String)Skladiste.get();
  }
} ///:~
```

Ako rezultat prevedemo unazad komandom **javap -c JednostavnoSkladiste**, dobićemo (nakon uređivanja):

```
public void set(java.lang.Object);
   0:   aload_0
   1:   aload_1
   2:   putfield #2; //Field obj:Object;
   5:   return

public java.lang.Object get();
   0:   aload_0
   1:   getfield #2; //Field obj:Object;
   4:   areturn

public static void main(java.lang.String[]);
   0:   new #3; //class JednostavnoSkladiste
   3:   dup
   4:   invokespecial #4; //Method "<init>":()V
   7:   astore_1
   8:   aload_1
   9:   ldc #5; //String Stavka
   11:  invokevirtual #6; //Method set:(Object;)V
   14:  aload_1
   15:  invokevirtual #7; //Method get:()Object;
   18:  checkcast #8; //class java/lang/String
   21:  astore_2
   22:  return
```

Metode **set( )** i **get( )** samo zadaju, odnosno čitaju vrednost, a konverzija tipa se proverava na mestu poziva metode **get( )**.

Ubacićemo sada generičke tipove u gornji kod:

```
//: genericki/GenerickoSkladiste.java

public class GenerickoSkladiste<T> {
  private T obj;
  public void set(T obj) { this.obj = obj; }
  public T get() { return obj; }
  public static void main(String[] args) {
    GenerickoSkladiste<String> Skladiste =
      new GenerickoSkladiste<String>();
    Skladiste.set("Stavka");
    String s = Skladiste.get();
  }
} ///:~
```

Više nema potrebe da rezultat metode **get( )** eksplicitno konvertujemo. Uz to takođe znamo da se tip vrednosti prosleđene metodi **set( )** proverava u vreme prevođenja. Ovo je relevantni bajtkod:

```
public void set(java.lang.Object);
   0:   aload_0
   1:   aload_1
   2:   putfield #2; //Field obj:Object;
   5:   return

public java.lang.Object get();
   0:   aload_0
   1:   getfield #2; //Field obj:Object;
   4:   areturn

public static void main(java.lang.String[]);
   0:   new #3; //class GenerickoSkladiste
   3:   dup
   4:   invokespecial #4; //Method "<init>":()V
   7:   astore_1
   8:   aload_1
   9:   ldc #5; //String Stavka
   11:  invokevirtual #6; //Method set:(Object;)V
   14:  aload_1
   15:  invokevirtual #7; //Method get:()Object;
   18:  checkcast #8; //class java/lang/String
   21:  astore_2
   22:  return
```

Dobija se identičan kôd. Dodatna provera ulaznog tipa u metodi **set( )** ne košta nas ništa, pošto je obavlja prevodilac. Tu je i konverzija tipa izlazne vrednosti metode **get( )**, ali toliko biste i sami morali da uradite – a ovu automatski umeće prevodilac, pa je kôd koji morate da napišete (i pročitate) čistiji.

Pošto metode **get( )** i **set( )** proizvode isti bajtkod, u generičkom kodu se sve što je važno odigrava na granicama – dodatna provera ulaznih vrednosti u vreme prevođenja i umetanje konverzije izlaznih vrednosti. Smanjićete zbrku oko brisanja ako ne zaboravite da se „na granicama odigrava sve što je važno".

Kompenzacija za brisanje

Kao što smo videli, brisanjem se gubi mogućnost obavljanja određenih operacija u generičkom kodu. Sve ono za šta je potrebno poznavanje stvarnih tipova u vreme izvršavanja, neće raditi:

```
//: generics/Obrisana.java
// {CompileTimeError} (Neće biti prevedena)

public class Obrisana<T> {
  private final int VELICINA = 100;
  public static void f(Object arg) {
    if(arg instanceof T) {}         // Greška
    T var = new T();                // Greška
    T[] niz = new T[VELICINA];      // Greška
    T[] niz = (T)new Object[VELICINA];
                                    // Upozorenje koje nije zaustavljeno
  }
} ///:~
```

Dešava se da programiranjem zaobiđete ove probleme, ali ponekad morate da kompenzujete brisanje uvođenjem *oznake tipa* (engl. *type tag*). To znači da eksplicitno prosleđujete **Class** objekat za svoj tip, da biste mogli da ga upotrebljavate u izrazima s tipom.

Primera radi, u prethodnom programu nije uspeo pokušaj da se upotrebi naredba **instanceof**, zato što su informacije o tipu obrisane. Ako uvedete oznaku tipa, umesto te naredbe moći ćete da upotrebite dinamičku metodu **isInstance( )**:

```
//: generics/HvatanjeTipaKlase.java

class Zgrada {}
class Kuca extends Zgrada {}

public class HvatanjeTipaKlase<T> {
  Class<T> vrsta;
  public HvatanjeTipaKlase(Class<T> vrsta) {
    this.vrsta = vrsta;
  }
```

```
  public boolean f(Object arg) {
    return vrsta.isInstance(arg);
  }
  public static void main(String[] args) {
    HvatanjeTipaKlase<Zgrada> ctt1 =
      new HvatanjeTipaKlase<Zgrada>(Zgrada.class);
    System.out.println(ctt1.f(new Zgrada()));
    System.out.println(ctt1.f(new Kuca()));
    HvatanjeTipaKlase<Kuca> ctt2 =
      new HvatanjeTipaKlase<Kuca>(Kuca.class);
    System.out.println(ctt2.f(new Zgrada()));
    System.out.println(ctt2.f(new Kuca()));
  }
} /* Ispis:
true
true
false
true
*///:~
```

Prevodilac proverava da li oznaka tipa odgovara generičkom argumentu.

Vežba 21: (4) Izmenite program **HvatanjeTipaKlase.java** tako što ćete dodati mapu **Map<String,Class<?>>**, metodu **dodajTip(String imetipa, Class<?> vrsta)** i metodu **napraviNovu(String imetipa)**. Metoda **napraviNovu( )** proizvodi novu instancu klase pridružene znakovnom nizu svog argumenta ili poruku o grešci.

Pravljenje instanci tipova

Pokušaj pravljenja nove generičke metode naredbom **new T( )** u programu **Obrisana.java** ne uspeva, delom zbog brisanja, a delom zbog toga što prevodilac ne može da proveri da li **T** ima podrazumevani konstruktor (bez argumenata). U C++-u je ova operacija prirodna, jednostavna i bezbedna (proverava se u vreme prevođenja):

```
//: generics/PravljenjeInstanceGenerickogTipa.cpp
// C++, ne Java!

template<class T> class Nesto {
  T x; // Pravljenje polja tipa T
  T* y; // Pokazivač na T
public:
  // Inicijalizacija pokazivača:
  Nesto() { y = new T(); }
};

class Bar {};

int main() {
  Nesto<Bar> nb;
  Nesto<int> ni; // ... radi i s prostim tipovima
} ///:~
```

U Javi je rešenje proslediti proizvodni objekat i pomoću njega napraviti novu instancu. Podesan proizvodni objekat je upravo objekat tipa **Class**, pa ako koristite oznaku tipa, za pravljenje novog objekta tog tipa možete upotrebiti metodu **newInstance( )**:

```
//: generics/PravljenjeInstanceGenerickogTipa.java
import static net.mindview.util.Print.*;

class KlasaKaoProizvodjac<T> {
  T x;
  public KlasaKaoProizvodjac(Class<T> vrsta) {
    try {
      x = vrsta.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class Zaposleni {}

public class PravljenjeInstanceGenerickogTipa {
  public static void main(String[] args) {
    KlasaKaoProizvodjac<Zaposleni> fz =
      new KlasaKaoProizvodjac<Zaposleni>(Zaposleni.class);
    print("KlasaKaoProizvodjac<Zaposleni> uspela");
    try {
      KlasaKaoProizvodjac<Integer> fi =
        new KlasaKaoProizvodjac<Integer>(Integer.class);
    } catch(Exception e) {
      print("KlasaKaoProizvodjac<Integer> zakazala");
    }
  }
} /* Ispis:
KlasaKaoProizvodjac<Zaposleni> uspela
KlasaKaoProizvodjac<Integer> zakazala
*///:~
```

Ovo će se prevesti, ali će **KlasaKaoProizvodjac<Integer>** zakazati zato što **Integer** nema podrazumevani konstruktor. Pošto se greška ne hvata u vreme prevođenja, ovaj pristup se ne sviđa ekipi u kompaniji Sun. Oni predlažu da upotrebite eksplicitnog proizvođača i ograničite tip, tako da prima samo klasu koja realizuje tog proizvođača:

```
//: generics/OgranicenjeProizvodjaca.java

interface ProizvodjacI<T> {
  T napravi();
}

class Nesto2<T> {
  private T x;
```

```
  public <F extends ProizvodjacI<T>> Nesto2(F proizvodjac) {
    x = proizvodjac.napravi();
  }
  // ...
}

class IntegerProizvodjac implements ProizvodjacI<Integer> {
  public Integer napravi() {
    return new Integer(0);
  }
}

class Spravica {
  public static class Proizvodjac implements ProizvodjacI<Spravica> {
    public Spravica napravi() {
      return new Spravica();
    }
  }
}

public class OgranicenjeProizvodjaca {
  public static void main(String[] args) {
    new Nesto2<Integer>(new IntegerProizvodjac());
    new Nesto2<Spravica>(new Spravica.Proizvodjac());
  }
} ///:~
```

Imajte u vidu da je ovo samo varijanta prosleđivanja **Class<T>**. Na oba načina prosleđuju se proizvodni objekti; **Class<T>** je igrom slučaja ugrađen proizvodni objekat, dok se u gornjem pristupu pravi eksplicitan proizvodni objekat. Uz to se obavlja i provera u vreme prevođenja.

Drugi pristup je projektni obrazac *Template Method* (Šablonska metoda). U narednom primeru šablonska metoda je **get( )**, a metoda **napravi( )** definiše se u potklasi tako da proizvodi objekat tog tipa:

```
//: generici/TvoracGenerickih.java

abstract class GenerickiUzNapravi<T> {
  final T element;
  GenerickiUzNapravi() { element = napravi(); }
  abstract T napravi();
}

class X {}

class Tvorac extends GenerickiUzNapravi<X> {
  X napravi() { return new X(); }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
}
```

```
public class TvoracGenerickih {
  public static void main(String[] args) {
    Tvorac c = new Tvorac();
    c.f();
  }
} /* Ispis:
X
*///:~
```

Vežba 22: (6) Upotrebite oznaku tipa i refleksiju za pravljenje metode koja koristi verziju sa argumentom metode **newInstance( )** koja treba da pravi objekte klase čiji konstruktor prima argumente.

Vežba 23: (1) Prepravite program **OgranicenjeProizvodjaca.java** tako da metoda **napravi( )** prima jedan argument.

Vežba 24: (3) Izmenite vežbu 21 tako da se proizvodni objekti drže u **Map**i umesto u **Class<?>**.

Nizovi generičkih tipova

Kao što ste videli u programu **Obrisana.java**, ne možete praviti nizove generičkih tipova. Opšte rešenje je upotrebiti **ArrayList** gde god ste u iskušenju da napravite niz generičkih tipova:

```
//: generics/ListaGenerickih.java
import java.util.*;

public class ListaGenerickih<T> {
  private List<T> niz = new ArrayList<T>();
  public void add(T stavka) { niz.add(stavka); }
  public T get(int indeks) { return niz.get(indeks); }
} ///:~
```

Ovako ste dobili ponašanje niza, ali i proveru tipova u vreme prevođenja koju donosi generički kôd.

Katkada će vam ipak zatrebati niz generičkih tipova (na primer, **ArrayList** upotrebljava nizove interno). Da stvar bude zanimljivija, možete definisati *referencu* tako da prevodilac bude zadovoljan. Na primer:

```
//: generics/NizGenerickihReferenci.java

class Generic<T> {}

public class NizGenerickihReferenci {
  static Generic<Integer>[] gia;
} ///:~
```

Prevodilac će to prihvatiti bez ikakvog upozorenja. Ali pošto nećete moći da napravite niz tačno tog tipa (uključujući tu i parametre tipa), stvari su pomalo zbunjujuće. Pošto svi nizovi imaju istu strukturu (veličinu svakog elementa i raspored elemenata) bez obzira na tip koji sadrže, izgleda kao da bi se mogao napraviti niz tipa **Object** i konvertovati u niz željenog tipa. To će se doduše prevesti, ali neće moći da se izvršava, jer proizvodi **ClassCastException**:

```
//: genericki/NizGenerickogTipa.java

public class NizGenerickogTipa {
  static final int VELICINA = 100;
  static Generic<Integer>[] gia;
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    // Prevešće se, ali pravi ClassCastException:
    //! gia = (Generic<Integer>[])new Object[VELICINA];
    // U vreme izvršavanja, tip je sirov (obrisan):
    gia = (Generic<Integer>[])new Generic[VELICINA];
    System.out.println(gia.getClass().getSimpleName());
    gia[0] = new Generic<Integer>();
    //! gia[1] = new Object(); // Greška u vreme prevođenja
    // Otkriva neslaganje tipova u vreme prevođenja:
    //! gia[2] = new Generic<Double>();
  }
} /* Ispis:
Generic[]
*///:~
```

Problem je u tome što nizovi pamte svoj stvarni tip, a on biva utvrđen u trenutku pravljenja niza. Dakle, iako je **gia** konvertovana u **Generic<Integer>[]**, ta informacija postoji samo u vreme prevođenja (a da nema anotacije **@SuppressWarnings**, dobili biste upozorenje za tu konverziju). U vreme izvršavanja, to je samo niz tipa **Object**, i to stvara probleme. Jedini način da uspešno napravite niz generičkog tipa jeste da napravite nov niz obrisanog tipa i da njega konvertujete.

Pogledajmo nešto teži primer. Razmotrićemo jednostavan generički omotač oko niza:

```
//: genericki/GenerickiNiz.java

public class GenerickiNiz<T> {
  private T[] niz;
  @SuppressWarnings("unchecked")
  public GenerickiNiz(int vel) {
    niz = (T[])new Object[vel];
  }
  public void put(int indeks, T stavka) {
    niz[indeks] = stavka;
  }
```

```
  public T get(int indeks) { return niz[indeks]; }
  // Metoda koja eksponira kako je niz predstavljen:
  public T[] rep() { return niz; }
  public static void main(String[] args) {
    GenerickiNiz<Integer> gai =
      new GenerickiNiz<Integer>(10);
    // Ovo prouzrokuje ClassCastException:
    //! Integer[] ia = gai.rep();
    // Ovo je u redu:
    Object[] oa = gai.rep();
  }
} ///:~
```

Kao pre, ne možemo reći **T[] niz = new T[vel]**, pa pravimo niz objekata i njega konvertujemo.

Metoda **rep( )** vraća **T[]**, koji bi u metodi **main( )** trebalo da bude **Integer[]** za **gai**, ali ako ga pozovete i rezultat pokušate da uhvatite kao referencu na **Integer[]**, dobićete **ClassCastException**, i opet zato što je stvarni tip u vreme izvršavanja **Object[]**.

Ako program **GenerickiNiz.java** prevedete nakon što ste komentarisali anotaciju **@SuppressWarnings**, prevodilac će dati upozorenje:

```
Note: GenerickiNiz.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

U ovom slučaju, dobili smo samo jedno upozorenje i mislimo da se ono odnosi na konverziju tipa. Ali ako zaista hoćete da budete sigurni, prevedite sa **-Xlint:unchecked**:

```
GenerickiNiz.java:7: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
    niz = (T[])new Object[vel];
                ^
1 warning
```

Upozorenje se stvarno odnosi na konverziju. Pošto upozorenja smetaju, najbolje je da utvrdimo kako smo određeno upozorenje očekivali pa da ga isključimo pomoću **@SuppressWarnings**. Tako ćemo zaista ispitati upozorenje kada se pojavi.

Zbog brisanja, tip niza u vreme izvršavanja može biti samo **Object[]**. Ukoliko ga odmah konvertujemo u **T[]**, onda se stvarni tip niza gubi u vreme prevođenja, pa ga ni prevodilac neće proveravati i time sprečiti potencijalne greške. Zato je bolje upotrebiti **Object[]** unutar kontejnera i dodati konverziju u tip **T** na mestu gde upotrebljavate element niza. Pogledajmo kako bi to izgledalo u primeru **GenerickiNiz.java**:

```
//: generics/GenerickiNiz2.java

public class GenerickiNiz2<T> {
  private Object[] niz;
```

```
  public GenerickiNiz2(int vel) {
    niz = new Object[vel];
  }
  public void put(int indeks, T stavka) {
    niz[indeks] = stavka;
  }
  @SuppressWarnings("unchecked")
  public T get(int indeks) { return (T)niz[indeks]; }
  @SuppressWarnings("unchecked")
  public T[] rep() {
    return (T[])niz; // Upozorenje: neproverena konverzija
  }
  public static void main(String[] args) {
    GenerickiNiz2<Integer> gai =
      new GenerickiNiz2<Integer>(10);
    for(int i = 0; i < 10; i ++)
      gai.put(i, i);
    for(int i = 0; i < 10; i ++)
      System.out.print(gai.get(i) + " ");
    System.out.println();
    try {
      Integer[] ia = gai.rep();
    } catch(Exception e) { System.out.println(e); }
  }
} /* Ispis: (primer)
0 1 2 3 4 5 6 7 8 9
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to
[Ljava.lang.Integer;
*///:~
```

Isprva ovo ne izgleda bitno drugačije, samo što je konverzija premeštena. Bez anotacija **@SuppressWarnings**, dobijaćete upozorenja „unchecked" (neprovereno). Međutim, interno je niz sada predstavljen kao **Object**[], a ne kao **T**[]. Kada se pozove metoda **get**( ), ona konvertuje objekat u tip T, što u stvari i jeste tačan tip, pa je to bezbedno. Međutim, ako pozovete **rep**( ), ona će ponovo pokušati da **Object**[] konvertuje u **T**[], što je i dalje netačno, i proizvodi upozorenje u vreme prevođenja i izuzetak u vreme izvršavanja. Dakle, nema načina da se obori tip pripadnog internog niza koji može biti samo **Object**[]. Prednost internog tretiranja **niz**a kao tipa **Object**[] umesto **T**[] jeste u tome što je manje verovatno da ćete u vreme izvršavanja zaboraviti koji je tip niza i zato slučajno napraviti grešku (iako se većina takvih grešaka, a možda i sve, brzo otkrije u vreme izvršavanja).

U novom kodu, trebalo bi da prosledite leksemu tipa. U tom slučaju, **GenerickiNiz** bi izgledao ovako:

```
//: generics/GenerickiNizSLeksemomTipa.java
import java.lang.reflect.*;

public class GenerickiNizSLeksemomTipa<T> {
  private T[] niz;
  @SuppressWarnings("unchecked")
```

```
  public GenerickiNizSLeksemomTipa(Class<T> tip, int vel) {
    niz = (T[])Array.newInstance(tip, vel);
  }
  public void put(int indeks, T stavka) {
    niz[indeks] = stavka;
  }
  public T get(int indeks) { return niz[indeks]; }
  // Eksponiraj kako je niz interno predstavljen:
  public T[] rep() { return niz; }
  public static void main(String[] args) {
    GenerickiNizSLeksemomTipa<Integer> gai =
      new GenerickiNizSLeksemomTipa<Integer>(
        Integer.class, 10);
    // Ovo sada radi:
    Integer[] ia = gai.rep();
  }
} ///:~
```

Leksemu niza **Class<T>** prosleđujemo konstruktoru kako bismo se oporavili od brisanja i mogli da napravimo niz pravog tipa koji nam je potreban, iako upozorenje zbog konverzije moramo da potisnemo sa **@SuppressWarnings**. Kada dobijemo pravi tip, možemo ga vratiti i dobiti željene rezultate, kao što vidite u metodi **main( )**. U vreme izvršavanja, tip niza je tačan tip **T[]**.

Nažalost, u izvornom kodu standardnih biblioteka Jave SE5, svuda ćete naći konverzije **Object** nizova u parametrizovane tipove. Primera radi, ovako izgleda (nakon čišćenja i pojednostavljenja) konstruktor za kopiraj-**ArrayList**-iz-kolekcije:

```
public ArrayList(Collection c) {
  velicina = c.size();
  elementPodataka = (E[])new Object[velicina];
  c.toArray(elementPodataka);
}
```

Ako pregledate klasu **ArrayList.java**, naći ćete mnoštvo takvih eksplicitnih konverzija. I šta se dešava kada klasu prevedemo?

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Kao što smo i mislili, standardne biblioteke prouzrokuju mnoštvo upozorenja. Ako ste pisali na C-u, pogotovo onome pre izdavanja standarda ANSI C, sećate se jedne posledice upozorenja: kada shvatite da možete da ih zanemarite, tako i radite. Zato je najbolje da prevodilac ne daje nikakve poruke ukoliko programer na njih ne mora da reaguje.

U svom weblogu,[3] Neal Gafter (jedan od vodećih programera Jave SE5) ističe da je bio lenj tokom prerade Javinih biblioteka i da mi ostali ne bi trebalo da sledimo njegov primer. Neal kaže i da nije mogao da popravi deo koda Javine biblioteke, a da pri tom ne

[3] *http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html*

sruši postojeći interfejs. Dakle, čak i ako se u izvornom kodu Javinih biblioteka pojavljuju neki idiomi projektovanja, ne znači da je to pravi način rada. Dok čitate kôd biblioteka, ne smete uzimati zdravo za gotovo da je to uzor koji treba da sledite u svom kodu.

Granice

Granice su ukratko predstavljene u prethodnom delu poglavlja (videti stranice 514 i 515). Granice omogućuju da namećete ograničenja parametarskim tipovima koje možete upotrebljavati u generičkom kodu. Iako time dobijate mogućnost da namećete pravila o tipovima na koje možete primeniti svoj generički kôd, potencijalno važniji efekat je da možete pozivati metode definisane u vašim ograničenim tipovima.

Pošto brisanje uklanja informacije o tipovima, jedine metode koje možete pozivati za neograničen generički parametar jesu one dostupne za **Object**. Međutim, ukoliko taj parametar možete ograničiti tako da bude podskup tipova, onda možete pozivati metode tog podskupa. Da bi sproveo to ograničenje, Javin generički kôd ponovo upotrebljava rezervisanu reč **extends**. Ne smete smetnuti s uma da u kontekstu generičkih granica, **extends** ima sasvim drugačije značenje nego obično. U narednom primeru prikazaćemo osnovno o granicama:

```
//: generics/OsnovnoOGranicama.java

interface ImaBoju { java.awt.Color getColor(); }

class Obojen<T extends ImaBoju> {
  T stavka;
  Obojen(T stavka) { this.stavka = stavka; }
  T getItem() { return stavka; }
  // Granica omogućuje da pozovete metodu:
  java.awt.Color boja() { return stavka.getColor(); }
}

class Dimenzija { public int x, y, z; }

// Ovo neće raditi -- prve moraju biti klase, a tek onda interfejsi:
// class ObojenaDimenzija<T extends ImaBoju & Dimenzija> {

// Više granica:
class ObojenaDimenzija<T extends Dimenzija & ImaBoju> {
  T stavka;
  ObojenaDimenzija(T stavka) { this.stavka = stavka; }
  T getItem() { return stavka; }
  java.awt.Color boja() { return stavka.getColor(); }
  int getX() { return stavka.x; }
  int getY() { return stavka.y; }
  int getZ() { return stavka.z; }
}
```

```
interface Tezinu { int tezina(); }

// Kao pri nasleđivanju, možete imati samo jednu
// konkretnu klasu, ali više interfejsa:
class Predmet<T extends Dimenzija & ImaBoju & Tezinu> {
  T stavka;
  Predmet(T stavka) { this.stavka = stavka; }
  T getItem() { return stavka; }
  java.awt.Color boja() { return stavka.getColor(); }
  int getX() { return stavka.x; }
  int getY() { return stavka.y; }
  int getZ() { return stavka.z; }
  int tezina() { return stavka.tezina(); }
}

class Ogranicen
extends Dimenzija implements ImaBoju, Tezinu {
  public java.awt.Color getColor() { return null; }
  public int tezina() { return 0; }
}

public class OsnovnoOGranicama {
  public static void main(String[] args) {
    Predmet<Ogranicen> predmet =
      new Predmet<Ogranicen>(new Ogranicen());
    predmet.boja();
    predmet.getY();
    predmet.tezina();
  }
} ///:~
```

Možda ste primetili da **OsnovnoOGranicama.java** sadrži ponavljanja koja se mogu izbeći nasleđivanjem. Sada ćete videti kako svaki nivo nasleđivanja dodaje svoja ograničenja:

```
//: genericki/NasledjivanjeGranica.java

class DrziStavku<T> {
  T stavka;
  DrziStavku(T stavka) { this.stavka = stavka; }
  T getItem() { return stavka; }
}

class Obojena2<T extends ImaBoju> extends DrziStavku<T> {
  Obojena2(T stavka) { super(stavka); }
  java.awt.Color boja() { return stavka.getColor(); }
}
class ObojenaDimenzija2<T extends Dimenzija & ImaBoju>
extends Obojena2<T> {
  ObojenaDimenzija2(T stavka) {  super(stavka); }
  int getX() { return stavka.x; }
```

```
  int getY() { return stavka.y; }
  int getZ() { return stavka.z; }
}

class Predmet2<T extends Dimenzija & ImaBoju & Tezinu>
extends ObojenaDimenzija2<T> {
  Predmet2(T stavka) {  super(stavka); }
  int tezina() { return stavka.tezina(); }
}

public class NasledjivanjeGranica {
  public static void main(String[] args) {
    Predmet2<Ogranicen> predmet2 =
      new Predmet2<Ogranicen>(new Ogranicen());
    predmet2.boja();
    predmet2.getY();
    predmet2.tezina();
  }
} ///:~
```

Klasa **DrziStavku** sadrži neki objekat, pa se to ponašanje nasleđuje u klasi **Obojena2** koja zahteva i da njen parametar bude usaglašen sa interfejsom **ImaBoju**. **ObojenaDimenzija2** i **Predmet2** dalje proširuju hijerarhiju i dodaju granice na svakom nivou. Sada se metode nasleđuju, pa se ne moraju ponavljati u svakoj klasi.

Evo primera s više slojeva:

```
//: generics/EpskaBitka.java
// Primer granica u Javinom generičkom kodu.
import java.util.*;

interface SuperSila {}
interface RentgenskiVid extends SuperSila {
  void vidiKrozZidove();
}
interface SuperSluh extends SuperSila {
  void cujeTiheZvukove();
}
interface SuperNjuh extends SuperSila {
  void pratiPoMirisu();
}

class SuperJunak<SNAGA extends SuperSila> {
  SNAGA snaga;
  SuperJunak(SNAGA snaga) { this.snaga = snaga; }
  SNAGA getPower() { return snaga; }
}

class SuperUhoda<SNAGA extends RentgenskiVid>
extends SuperJunak<SNAGA> {
```

```
  SuperUhoda(SNAGA snaga) { super(snaga); }
  void vidi() { snaga.vidiKrozZidove(); }
}

class PasJunak<SNAGA extends SuperSluh & SuperNjuh>
extends SuperJunak<SNAGA> {
  PasJunak(SNAGA snaga) { super(snaga); }
  void cuje() { snaga.cujeTiheZvukove(); }
  void njusi() { snaga.pratiPoMirisu(); }
}

class SuperCujeNjusi implements SuperSluh, SuperNjuh {
  public void cujeTiheZvukove() {}
  public void pratiPoMirisu() {}
}

class Ker extends PasJunak<SuperCujeNjusi> {
  Ker() { super(new SuperCujeNJusi()); }
}

public class EpskaBitka {
  // Granice u generičkim metodama:
  static <SNAGA extends SuperSluh>
  void koristiSuperSluh(SuperJunak<SNAGA> junak) {
    junak.getPower().cujeTiheZvukove();
  }
  static <SNAGA extends SuperSluh & SuperNjuh>
  void superTragac(SuperJunak<SNAGA> junak) {
    junak.getPower().cujeTiheZvukove();
    junak.getPower().pratiPoMirisu();
  }
  public static void main(String[] args) {
    Ker ker = new Ker();
    koristiSuperSluh(ker);
    superTragac(ker);
    // Ovo možete da uradite:
    List<? extends SuperSluh> audioFili;
    // Ali ovo ne možete:
    // List<? extends SuperSluh & SuperNjuh> kerovi;
  }
} ///:~
```

Vodite računa o tome da se džokerski argumenti (koje ćemo sledeće proučiti) mogu upotrebljavati samo za jednu granicu.

Vežba 25: (2) Napravite dva interfejsa i klasu koja ih realizuje. Napravite dve generičke metode, jednu čiji je argument parametar ograničen prvim interfejsom i drugu čiji je argument parametar ograničen drugim interfejsom. Napravite primerak klase koja realizuje oba interfejsa i pokažite da se ona može upotrebljavati sa oba interfejsa.

Džokerski argumenti

Već ste videli neke jednostavne upotrebe *džokerskih argumenata* – znakova pitanja u izrazima generičkih argumenata – u poglavlju *Čuvanje objekata*, a još više u poglavlju *Podaci o tipu*. U ovom odeljku razmotrićemo ih detaljnije.

Počećemo od primera koji pokazuje određeno ponašanje nizova: niz izvedenog tipa možete da dodelite referenci niza osnovnog tipa:

```
//: generics/KovarijantniNizovi.java

class Voce {}
class Jabuka extends Voce {}
class Jonathan extends Jabuka {}
class Narandža extends Voce {}

public class KovarijantniNizovi {
  public static void main(String[] args) {
    Voce[] voce = new Jabuka[10];
    voce[0] = new Jabuka(); // OK
    voce[1] = new Jonathan(); // OK
    // U vreme izvršavanja tip je Jabuka[], ne Voce[] niti Narandža[]:
    try {
      // Prevodilac dozvoljava da dodate Voce:
      voce[0] = new Voce(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
    try {
      // Prevodilac dozvoljava da dodate Narandže:
      voce[0] = new Narandža(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
  }
} /* Ispis:
java.lang.ArrayStoreException: Voce
java.lang.ArrayStoreException: Narandža
*///:~
```

Prvi red u metodi **main**( ) pravi niz **Jabuka** i dodeljuje ga referenci na niz tipa **Voce**. To ima smisla – **Jabuka** je vrsta voća, pa niz **Jabuka** treba istovremeno da bude i niz tipa **Voce**.

Međutim, da je pravi tip niza **Jabuka**[], u njega biste mogli da stavite samo objekat tipa **Jabuka** ili podtipa od **Jabuka**, što bi zaista delovalo i u vreme prevođenja i u vreme izvršavanja. Ali obratite pažnju na to da prevodilac dozvoljava stavljanje objekta tipa **Voce** u niz. Prevodiocu to ima smisla, pošto on ima referencu na **Voce**[] – zašto da ne dozvoli da se u niz stavi objekat tipa **Voce** ili bilo šta izvedeno iz tipa **Voce**, kao što je **Narandža**? Stoga je u vreme prevođenja to dozvoljeno. Međutim, u vreme izvršavanja, mehanizam nizova zna da radi s nizom **Jabuka**[] i generiše izuzetak kada se strani tip stavi u taj niz.

„Svođenje naviše" ovde nije prava reč. Vi zapravo jedan niz dodeljujete drugom. Svaki niz se ponaša kao da sadrži druge objekte, ali pošto možemo da svedemo naviše, jasno je da objekti niza mogu da poštuju pravila o tipu objekata koje niz sadrži. To je kao da nizovi znaju šta sadrže, pa ne možete da ih prevarite zbog provera u vreme prevođenja i u vreme izvršavanja.

Takvo ponašanje nizova nije toliko strašno, pošto u vreme izvršavanja vi ipak *saznajete* da ste umetnuli pogrešan tip. Ali jedan od osnovnih ciljeva generičkog koda jeste otkrivanje takvih grešaka već u vreme prevođenja. Šta će se desiti ako umesto nizova pokušamo da upotrebimo kontejnere?

```
//: generics/NeKovarijantniGenerickiTipovi.java
// {CompileTimeError} (Ne može se prevesti)
import java.util.*;

public class NeKovarijantniGenerickiTipovi {
  // Greška u vreme prevođenja: neusklađeni tipovi:
  List<Voce> listav = new ArrayList<Jabuka>();
} ///:~
```

Iako ćete isprva ovo možda protumačiti kao „Kontejner **Jabuka** se ne može dodeliti kontejneru za **Voce**", ne zaboravite da generički kôd nisu samo kontejneri. Greška nam zapravo kaže: „Generički tip *zadat tipom* **Jabuka** ne možete dodeliti generičkom tipu koji je *zadat tipom* **Voce**". Kada bi, kao u slučaju nizova, prevodilac o kodu znao dovoljno da može utvrditi da se radi o kontejnerima, možda bi bio malo popustljiviji. Ali on to ne zna, te odbija da dozvoli „svođenje naviše". To zapravo i nije „svođenje naviše" – **Lista Jabuka** nije **Lista** tipa **Voce**. **Lista Jabuka** može da sadrži objekte tipa **Jabuka** i podtipove od **Jabuka**, a **Lista Voce** može da sadrži sve podtipove tipa **Voce**. Da, uključujući i objekte tipa **Jabuka**, ali ona time ne postaje **Lista Jabuka** – i dalje je **Lista** tipa **Voce**. **Lista Jabuka** nije istog tipa kao **Lista** tipa **Voce**, iako je **Jabuka** podtip tipa **Voce**.

Pravi problem je to što govorimo o tipu kontejnera, a ne o tipu onoga što kontejner sadrži. Za razliku od nizova, generički kôd nema ugrađenu kovarijansu. Razlog je to što su nizovi potpuno definisani u jeziku i mogu im se ugraditi provere i u vreme prevođenja i u vreme izvršavanja; s druge strane, u generičkom kodu, prevodilac i izvršno okruženje ne mogu znati šta hoćete da uradite sa svojim tipovima i kakva bi trebalo da budu pravila.

Međutim, ponekad bismo hteli da između kontejnera i onoga što kontejneri sadrže uspostavimo neku vrstu svođenja naviše. Tome služe džokerski argumenti.

```
//: generics/GenerickiTipoviIKovarijansa.java
import java.util.*;

public class GenerickiTipoviIKovarijansa {
  public static void main(String[] args) {
    // Džokerski argumenti omogućuju kovarijansu:
    List<? extends Voce> listav = new ArrayList<Jabuka>();
    // Greška u vreme prevođenja: nije moguće
    // dodavanje objekta bilo kojeg tipa:
    // listav.add(new Jabuka());
    // listav.add(new Voce());
    // listav.add(new Object());
    listav.add(null); // Dozvoljeno, ali nekorisno
    // Znamo da vraća barem Voce:
    Voce f = listav.get(0);
  }
} ///:~
```

Tip objekta **listav** sada je **List<? extends Voce>**, što možete čitati kao „lista bilo kojeg tipa koji nasleđuje **Voce**". Međutim, to ne znači da ova **List**a prima sve podtipove klase **Voce**. Džokerski argument označava određen tip, pa gornji izraz znači „određen tip koji referenca **listav** ne specificira". Stoga dodeljena **List**a mora da čuva specificiran tip kao što su **Voce** ili **Jabuka**. Da bi svođenje naviše na **listav** bilo moguće, taj tip je „nije važno koji".

Ukoliko je jedino ograničenje da ta **List**a sadrži određeno **Voce** ili podtip klase **Voce**, ali zapravo vam nije važno koji, šta možete da uradite s takvom **List**om? Ako ne znate koji tip ta **List**a sadrži, kako da joj bezbedno dodate neki objekat? Ne možete, kao što ni u programu **KovarijantniNizovi.java** niste mogli da niz „svedete naviše", sem što u ovom slučaju to sprečava prevodilac, a tamo sistem za izvršavanje. Sada problem otkrivate ranije.

Možda ste pomislili da je ovo otišlo predaleko, jer sada **List**i za koju ste upravo rekli da sadrži objekte tipa **Jabuka**, ne možete da dodate objekat tipa **Jabuka**. U pravu ste, ali prevodilac to ne zna. Objekat tipa **List<? extends Voce>** može legalno da ukazuje na objekat tipa **List<Narandža>**. Nakon što uradite ovu vrstu „svođenja naviše", gubite mogućnost da bilo šta prosledite u listu, čak i **Object**.

S druge strane, ako pozovete metodu koja vraća **Voce**, to je bezbedno jer znate da sadržaj **List**e mora u najmanju ruku biti tipa **Voce**, pa će prevodilac to dozvoliti.

Vežba 26: (2) Pokažite kovarijansu nizova pomoću klasa **Number** i **Integer**.

Vežba 27: (2) Pokažite da kovarijansa ne radi s **List**ama i klasama **Number** i **Integer**, a zatim uvedite džokerske argumente.

Koliko je prevodilac pametan?

Ako sada mislite da ne možete pozivati metode koje primaju argumente, pogledajte ovo:

```
//: genericki/PametPrevodioca.java
import java.util.*;

public class PametPrevodioca {
  public static void main(String[] args) {
    List<? extends Voce> listav =
      Arrays.asList(new Jabuka());
    Jabuka a = (Jabuka)listav.get(0); // Nema upozorenja
    listav.contains(new Jabuka()); // Argument je 'Object'
    listav.indexOf(new Jabuka()); // Argument je 'Object'
  }
} ///:~
```

Pozvali smo metode **contains( )** i **indexOf( )** koje primaju objekte tipa **Jabuka** kao argumente, i sve dobro radi. Znači li to da prevodilac ispituje kod i proverava da li određena metoda modifikuje svoj objekat?

Iz dokumentacije za **ArrayList** vidimo da prevodilac nije toliko pametan. Dok metoda **add( )** prima argument tipa generičkog parametra, metode **contains( )** i **indexOf( )** primaju argumente tipa **Object**. Stoga kada zadate **ArrayList<? extends Voce>**, argument za **add( )** postaje **? extends Voce**. Iz tog opisa prevodilac ne može znati koji bi podtip klase **Voce** trebalo da dođe na to mesto, te stoga ne prihvata nijedan od tih podtipova. Nije

važno da li ste najpre tip **Jabuka** sveli naviše na **Voce** – prevodilac prosto odbija da pozove metodu (kao što je **add( )**) ukoliko je u listi argumenata džokerski argument.

Argumenti metoda **contains( )** i **indexOf( )** tipa jesu **Object**, nema nikakvih džokerskih argumenata, i prevodilac dozvoljava poziv. To znači da je na projektantu generičke klase da odluči koji pozivi su bezbedni i da za njihove argumente upotrebi tip **Object**. Ukoliko hoćete da onemogućite određeni poziv kada se u tipu upotrebljava džokerski argument, u listu argumenata stavite parametar tipa.

To možete videti u ovoj veoma jednostavnoj klasi **Skladiste**:

```
//: generics/Skladiste.java

public class Skladiste<T> {
  private T vrednost;
  public Skladiste() {}
  public Skladiste(T vre) { vrednost = vre; }
  public void set(T vre) { vrednost = vre; }
  public T get() { return vrednost; }
  public boolean equals(Object obj) {
    return vrednost.equals(obj);
  }
  public static void main(String[] args) {
    Skladiste<Jabuka> Jabuka = new Skladiste<Jabuka>(new Jabuka());
    Jabuka d = Jabuka.get();
    Jabuka.set(d);
    // Skladiste<Voce> Voce = Jabuka; // Svođenje naviše nije moguće
    Skladiste<? extends Voce> voce = Jabuka; // OK
    Voce p = voce.get();
    d = (Jabuka)voce.get(); // Vraća 'Object'
    try {
      Narandža c = (Narandža)voce.get(); // Nema upozorenja
    } catch(Exception e) { System.out.println(e); }
    // voce.set(new Jabuka()); // Ne možete pozvati set()
    // voce.set(new Voce()); // Ne možete pozvati set()
    System.out.println(voce.equals(d)); // OK
  }
} /* Ispis: (primer)
java.lang.ClassCastException: Jabuka cannot be cast to Narandža
true
*///:~
```

Skladiste ima metodu **set( )** koja prima objekat tipa **T**, metodu **get( )** koja vraća objekat tipa **T**, i metodu **equals( )** koja prima **Object**. Kao što ste već videli, ako napravite objekat tipa **Skladiste<Jabuka>**, ne možete ga svesti naviše na **Skladiste<Voce>**, ali ga možete svesti naviše na **Skladiste<? extends Voce>**. Ako pozovete **get( )**, ona vam vraća samo objekat tipa **Voce** – toliko zna uz datu granicu „bilo šta što nasleđuje **Voce**". Ukoliko znate stvarni tip, možete ga svesti naviše na određen podtip klase **Voce** i za to nećete dobijati upozorenja, ali rizikujete da se pojavi **ClassCastException**. Metoda **set( )** ne radi ni

sa objektom tipa **Jabuka** niti sa objektom tipa **Voce**, pošto je njen argument takođe **? Extends Voce**, što znači da može biti bilo šta, a prevodilac ne može da proveri bezbednost tipa „bilo šta".

Međutim, metoda **equals( )** radi dobro, zato što kao argument prima **Object**, a ne T. Dakle, prevodilac pazi samo na tipove objekata koji bivaju prosleđeni ili vraćeni. On ne analizira kôd kako bi video da li zaista upisujete ili učitavate.

Kontravarijansa

Može se otići i suprotnim putem i upotrebiti *džokerski argument nadtipa*. Tada kažete da je džokerski argument ograničen bilo kojom natklasom određene klase, tako što zadate **<? super MojaKlasa>** ili čak upotrebite parametar tipa: **<? super T>** (iako generički parametar ne možete zadati kao granicu nadtipa, tj. ne možete reći **<T super MojaKlasa>**). Ovo omogućuje da bezbedno prosledite objekat nekog tipa u generički tip. Dakle, sa džokerskim argumentima nadtipova možete upisivati u kontejnere (objekte tipa **Collection**):

```
//: generics/DzokeriNadtipova.java
import java.util.*;

public class DzokeriNadtipova {
  static void upisiU(List<? super Jabuka> jabuke) {
    jabuke.add(new Jabuka());
    jabuke.add(new Jonathan());
    // jabuke.add(new Voce()); // Greška
  }
} ///:~
```

Argument **jabuke** je **List**a nekog tipa koji je nadtip klase **Jabuka**; zato znate da je toj listi bezbedno dodati objekat tipa **Jabuka** ili podtipa klase **Jabuka**. Međutim, pošto je **Jabuka** *donja granica*, ne znate da li je takvoj **List**i bezbedno dodati **Voce**, pošto biste na taj način dozvolili da se **List**a otvara za dodavanje tipova koji nisu **Jabuka**, što bi ugrozilo bezbednost statičnih tipova.

Dakle, granice podtipova i nadtipova možete smatrati načinima „upisivanja" (prosleđivanja u metodu) u generički tip, odnosno „učitavanja" (vraćanja iz metode) iz generičkog tipa.

Granice nadtipova ublažuju ograničenja onoga što možete da prosledite u metodu:

```
//: generics/GenerickoUpisivanje.java
import java.util.*;

public class GenerickoUpisivanje {
  static <T> void tacnoUpisi(List<T> lista, T stavka) {
      lista.add(stavka);
  }
  static List<Jabuka> jabuke = new ArrayList<Jabuka>();
  static List<Voce> voce = new ArrayList<Voce>();
  static void f1() {
    tacnoUpisi(jabuke, new Jabuka());
```

```
    // tacnoUpisi(voce, new Jabuka()); // Greška:
    // Incompatible types: found Voce, required Jabuka
    // (Neusklađeni tipovi: našao Voce, očekivao Jabuka)
  }
  static <T> void
  upisivanjeSDzokerom(List<? super T> lista, T stavka) {
    lista.add(stavka);
  }
  static void f2() {
    upisivanjeSDzokerom(jabuke, new Jabuka());
    upisivanjeSDzokerom(voce, new Jabuka());
  }
  public static void main(String[] args) { f1(); f2(); }
} ///:~
```

Metoda **tacnoUpisi( )** upotrebljava tačan parametar tipa (nema džokerskog argumenta). U **f1( )** možete videti da to lepo radi – dokle god u listu **List<Jabuka>** stavljate samo objekte tipa **Jabuka**. Međutim, **tacnoUpisi( )** ne dozvoljava da u listu **List<Voce>** stavite objekat tipa **Jabuka**, mada vi znate da bi to trebalo da bude moguće.

U metodi **upisivanjeSDzokerom( )** argument je lista **List<? super T>**, pa ona sadrži određeni tip izveden iz **T**; stoga je njenim metodama kao argument bezbedno proslediti **T** ili nešto izvedeno iz **T**. To možete videti u **f2( )**, gde je i dalje moguće staviti objekat tipa **Jabuka** u listu **List<Jabuka>**, kao pre, ali sada je objekat tipa **Jabuka** moguće staviti i u listu **List<Voce>**, kao što smo očekivali.

Istu vrstu analize možemo sprovesti kao pregled kovarijanse i džokerskih argumenata:

```
//: genericki/GenerickoCitanje.java
import java.util.*;

public class GenerickoCitanje {
  static <T> T tacnoProcitaj(List<T> lista) {
    return lista.get(0);
  }
  static List<Jabuka> jabuke = Arrays.asList(new Jabuka());
  static List<Voce> voce = Arrays.asList(new Voce());
  // Statična metoda se prilagođava svakom pozivu:
  static void f1() {
    Jabuka a = tacnoProcitaj(jabuke);
    Voce f = tacnoProcitaj(voce);
    f = tacnoProcitaj(jabuke);
  }
  // Međutim, ako imate klasu, njen tip se utvrđuje
  // prilikom pravljenja njene instance:
  static class Citac<T> {
    T tacnoProcitaj(List<T> lista) { return lista.get(0); }
  }
  static void f2() {
    Citac<Voce> citacVoca = new Citac<Voce>();
    Voce f = citacVoca.tacnoProcitaj(voce);
```

```
    // Voce a = citacVoca.tacnoProcitaj(jabuke); // Greška:
    // tacnoProcitaj(List<Voce>) ne može biti
    // primenjeno na (List<Jabuka>).
  }
  static class KovarijantniCitac<T> {
    T citajKovarijantno(List<? extends T> lista) {
      return lista.get(0);
    }
  }
  static void f3() {
    KovarijantniCitac<Voce> citacVoca =
      new KovarijantniCitac<Voce>();
    Voce f = citacVoca.citajKovarijantno(voce);
    Voce a = citacVoca.citajKovarijantno(jabuke);
  }
  public static void main(String[] args) {
    f1(); f2(); f3();
  }
} ///:~
```

Kao pre, prva metoda **tacnoProcitaj( )** upotrebljava precizan tip. Taj precizan tip bez džokerskog argumenta možete i upisivati u listu i čitati iz nje. Pored toga, za povratnu vrednost, statična generička metoda **tacnoProcitaj( )** zapravo se prilagođava svakom pozivu i vraća objekat tipa **Jabuka** iz liste **List<Jabuka>** i objekat tipa **Voce** iz liste **List<Voce>**, kao što možete videti u **f1( )**. Dakle, ako možete da napravite statičnu generičku metodu, samo za čitanje nije vam potrebna kovarijansa.

Međutim, ako imate generičku klasu, parametar tipa te klase utvrđuje se prilikom pravljenja njene instance. Kao što vidite u **f2( )**, instanca klase **citacVoca** može da pročita objekat tipa **Voce** iz liste **List<Voce>**, pošto je to njen tačan tip. Ali bi i lista **List<Jabuka>** trebalo da proizvede objekte tipa **Voce**, a **citacVoca** to ne dozvoljava.

Da bi se rešio problem, metoda **KovarijantniCitac.citajKovarijantno( )** prima **List<? extends T>**, pa je bezbedno pročitati **T** iz te liste (znate da je u njoj sve barem tipa **T** ili izvedeno iz tipa **T**). U **f3( )** vidite da je sada bezbedno pročitati objekat tipa **Voce** iz liste **List<Jabuka>**.

Vežba 28: (4) Napravite generičku klasu **Genericka1<T>** čija jedina metoda prima argument tipa **T**. Napravite drugu generičku klasu **Genericka2<T>** čija jedina metoda vraća argument tipa **T**. Napišite generičku metodu s kontravarijantnim argumentom prve generičke klase koja poziva svoju metodu. Napišite drugu generičku metodu s kovarijantnim argumentom druge generičke klase koja poziva svoju metodu. Testirajte kôd pomoću biblioteke **podaciotipu.ljubimci**.

Neograničeni džokerski argumenti

Neograničeni džokerski argument <?> prividno označava „bilo šta", pa izgleda kao da je korišćenje neograničenog džokerskog argumenta ekvivalentno korišćenju sirovog tipa. Zaista, prevodilac se na prvi pogled slaže s tom tvrdnjom:

```
//: genericki/NeograniceniDzokeri1.java
import java.util.*;

public class NeograniceniDzokeri1 {
  static List lista1;
  static List<?> lista2;
  static List<? extends Object> lista3;
  static void assign1(List lista) {
    lista1 = lista;
    lista2 = lista;
    // lista3 = lista; // Upozorenje: unchecked conversion (neproverena
                       // konverzija)
    // Found: List, Required: List<? extends Object>
  }
  static void assign2(List<?> lista) {
    lista1 = lista;
    lista2 = lista;
    lista3 = lista;
  }
  static void assign3(List<? extends Object> lista) {
    lista1 = lista;
    lista2 = lista;
    lista3 = lista;
  }
  public static void main(String[] args) {
    assign1(new ArrayList());
    assign2(new ArrayList());
    // assign3(new ArrayList()); // Upozorenje:
    // unchecked conversion (neproverena konverzija).
    // Found: ArrayList, Required: List<? extends Object>
    assign1(new ArrayList<String>());
    assign2(new ArrayList<String>());
    assign3(new ArrayList<String>());
    // Oba oblika su prihvatljiva kao List<?>:
    List<?> divljaLista = new ArrayList();
    divljaLista = new ArrayList<String>();
    assign1(divljaLista);
    assign2(divljaLista);
    assign3(divljaLista);
  }
} ///:~
```

Ima mnogo ovakvih slučajeva gde bi prevodilac mogao da se malo manje brine o tome da li upotrebljavate neki sirov tip ili <?>. U tim slučajevima, možete smatrati da je <?> dekoracija; pa ipak, taj džokerski argument je vredan, zato što u stvari kazuje: „Ovaj kod sam napisao imajući u vidu Javin generički mehanizam, i ovde ne označava da upotrebljavam neki sirov tip, nego da u ovom slučaju generički parametar može da sadrži bilo koji tip".

U drugom primeru pokazana je jedna važna primena neograničenih džokerskih argumenata. Kada radite s više generičkih parametara, ponekad treba dozvoliti da jedan parametar bude proizvoljnog tipa, dok za drugi morate zadati određeni tip:

```
//: generics/NeograniceniDzokeri2.java
import java.util.*;

public class NeograniceniDzokeri2 {
  static Map mapa1;
  static Map<?,?> mapa2;
  static Map<String,?> mapa3;
  static void dodeli1(Map mapa) { mapa1 = mapa; }
  static void dodeli2(Map<?,?> mapa) { mapa2 = mapa; }
  static void dodeli3(Map<String,?> mapa) { mapa3 = mapa; }
  public static void main(String[] args) {
    dodeli1(new HashMap());
    dodeli2(new HashMap());
    // dodeli3(new HashMap()); // Upozorenje:
    // Unchecked conversion (neproverena konverzija).
    // Found: HashMap, Required: Map<String,?>
    // (Našao: HashMap, Očekivao: Map<String,?>)
    dodeli1(new HashMap<String,Integer>());
    dodeli2(new HashMap<String,Integer>());
    dodeli3(new HashMap<String,Integer>());
  }
} ///:~
```

I ponovo, kada su svi džokerski argumenti neograničeni, kao u **Map<?,?>**, izgleda da prevodilac to ne razlikuje od sirove **Mape**. Pored toga, program **NeograniceniDzokeri1.java** pokazuje da prevodilac različito tretira liste **List<?>** i **List<? extends Object>**.

Zbunjuje to što prevodilac ne pravi uvek razliku između, na primer, lista **List** i **List<?>**, pa biste mogli pomisliti da su one jednake. Zaista, pošto se generički argument briše do svoje prve granice, izgledalo bi kao da je **List<?>** ekvivalentna sa **List<Object>** i da je **List** isto što i **List<Object>** – sem što nijedna od tih tvrdnji nije potpuno tačna. **List** zapravo znači „sirova **Lista** koja sadrži bilo koji objekat tipa **Object**", dok **List<?>** znači „nesirova **Lista** *određenog tipa*, samo što mi ne znamo kojega".

Kada prevodilac pravi razliku između sirovih tipova i tipova koji su zadati neograničenim džokerskim argumentima? U sledećem primeru upotrebljena je prethodno definisana klasa **Skladiste<T>**. U njemu su metode koje **Skladiste** primaju kao argument, ali na različite načine: kao sirov tip, sa određenim parametrom tipa i sa neograničenim džokerskim argumentom kao parametrom:

```
//: generics/Dzokeri.java
// Istraživanje značenja džokerskih argumenata.

public class Dzokeri {
   // Sirov argument:
  static void siroviArgumenti(Skladiste skladiste, Object arg) {
```

```
    // skladiste.set(arg); // Upozorenje:
    //   Unchecked call to set(T) as a
    //   member of the raw type Skladiste
    //   (Neproveren poziv metode set(T) kao
    //   člana sirovog tipa Skladiste)
    // skladiste.set(new Dzokeri()); // Isto upozorenje

    // Ovo nije dozvoljeno; nema tipa 'T':
    // T t = skladiste.get();

    // OK, ali izgubljena je informacija o tipu:
    Object obj = skladiste.get();
  }
  // Slična kao siroviArgumenti(), ali prouzrokuje
  // greške umesto upozorenja:
  static void neogArg(Skladiste<?> skladiste, Object arg) {
    // skladiste.set(arg); // Greška:
    //   Metoda set((rezultat od) ?) u Skladiste<(rezultat od) ?>
    //   ne može biti primenjena na (Object)
    // skladiste.set(new Dzokeri()); // Ista greška

    // Ovo nije dozvoljeno; nema tipa 'T':
    // T t = skladiste.get();

    // OK, ali je izgubljena informacija o tipu:
    Object obj = skladiste.get();
  }
  static <T> T odredjena1(Skladiste<T> skladiste) {
    T t = skladiste.get();
    return t;
  }
  static <T> T odredjena2(Skladiste<T> skladiste, T arg) {
    Skladiste.set(arg);
    T t = skladiste.get();
    return t;
  }
  static <T>
  T divljiPodtip(Skladiste<? extends T> skladiste, T arg) {
    // skladiste.set(arg); // Greska:
    //   Metoda set((rezultat od) ? extends T) u
    //   Skladiste<(rezultat od) ? extends T>
    //   ne može da se primeni na (T)
    T t = skladiste.get();
    return t;
  }
  static <T>
  void divljiNadtip(Skladiste<? super T> skladiste, T arg) {
    skladiste.set(arg);
    // T t = skladiste.get();  // Greska:
    //   Neusklađeni tipovi: našao Object, očekivao T
```

```
    // OK, ali je izgubljena informacija o tipu:
    Object obj = skladiste.get();
  }
  public static void main(String[] args) {
    Skladiste sirovo = new Skladiste<Long>();
    // Ili:
    sirovo = new Skladiste();
    Skladiste<Long> potpunoZadato = new Skladiste<Long>();
    Skladiste<?> neograniceno = new Skladiste<Long>();
    Skladiste<? extends Long> ograniceno = new Skladiste<Long>();
    Long lng = 1L;

    siroviArgumenti(sirovo, lng);
    siroviArgumenti(potpunoZadato, lng);
    siroviArgumenti(neograniceno, lng);
    siroviArgumenti(ograniceno, lng);

    neogArg(sirovo, lng);
    neogArg(potpunoZadato, lng);
    neogArg(neograniceno, lng);
    neogArg(ograniceno, lng);

    // Object r1 = odredjena1(sirovo); // Upozorenja:
    //   Unchecked conversion from Skladiste to Skladiste<T>
    //   (Neproverena konverzija tipa Skladiste u Skladiste<T>)
    //   Unchecked method invocation: odredjena1(Skladiste<T>)
    //   is applied to (Skladiste)
    //   (Neproveren poziv metode: odredjena1(Skladiste<T>)
    //    primenjena na (Skladiste))
    Long r2 = odredjena1(potpunoZadato);
    Object r3 = odredjena1(neograniceno); // Mora vratiti Object
    Long r4 = odredjena1(ograniceno);

    // Long r5 = odredjena2(sirovo, lng); // Upozorenja:
    //   Unchecked conversion from Skladiste to Skladiste<Long>
    //   (Neproverena konverzija tipa Skladiste u Skladiste<Long>)
    //   Unchecked method invocation: odredjena2(Skladiste<T>,T)
    //   is applied to (Skladiste,Long)
    //   (Neproveren poziv metode: odredjena2(Skladiste<T>,T)
    //    primenjena na (Skladiste,Long))
    Long r6 = odredjena2(potpunoZadato, lng);
    // Long r7 = odredjena2(neograniceno, lng); // Greska:
    //   odredjena2(Skladiste<T>,T) ne može biti primenjena
    //   na (Skladiste<(rezultat od) ?>,Long)
    // Long r8 = odredjena2(ograniceno, lng); // Greska:
    //   odredjena2(Skladiste<T>,T) ne može biti primenjena
    //   na (Skladiste<(rezultat od) ? extends Long>,Long)

    // Long r9 = divljiPodtip(sirovo, lng); // Upozorenja:
    //   Unchecked conversion from Skladiste
```

```
    //   to Skladiste<? extends Long>
    //   (Neproverena konverzija tipa Skladiste
    //   u Skladiste<? extends Long >)
    //   Unchecked method invocation:
    //   divljiPodtip(Skladiste<? extends T>,T) is
    //   applied to (Skladiste,Long)
    //   (Neproveren poziv metode:
    //    divljiPodtip(Skladiste<? extends T>,T)
    //    primenjena na (Skladiste,Long))

    Long r10 = divljiPodtip(potpunoZadato, lng);
    // OK, ali može da vrati samo Object:
    Object r11 = divljiPodtip(neograniceno, lng);
    Long r12 = divljiPodtip(ograniceno, lng);

    // divljiNadtip(sirovo, lng); // Upozorenja:
    //   Unchecked conversion from Skladiste
    //   to Skladiste<? super Long>
    //   (Neproverena konverzija tipa Skladiste
    //   u Skladiste<? super Long >)
    //   Unchecked method invocation:
    //   divljiNadtip(Skladiste<? super T>,T)
    //   is applied to (Skladiste,Long)
    //   (Neproveren poziv metode:
    //    divljiNadtip(Skladiste<? super T>,T)
    //    primenjena na (Skladiste,Long))

    divljiNadtip(potpunoZadato, lng);
    // divljiNadtip(neograniceno, lng); // Greska:
    //   divljiNadtip(Skladiste<? super T>,T) ne može biti
    //   primenjena na (Skladiste<(rezultat od) ?>,Long)
    // divljiNadtip(ograniceno, lng); // Greska:
    //   divljiNadtip(Skladiste<? super T>,T) ne može biti
    //  primenjena na (Skladiste<(rezultat od) ? extends Long>,Long)
  }
} ///:~
```

U metodi **siroviArgumenti( )**, prevodilac zna da je **Skladiste** generički tip, pa iako je ovde izražen kao sirov tip, prevodilac zna da prosleđivanje objekta tipa **Object** metodi **set( )** nije bezbedno. Pošto je to sirov tip, metodi **set( )** možete proslediti objekat bilo kojeg tipa koji će biti sveden naviše na **Object**. Dakle, kadgod imate sirov tip, nemate proveru u vreme prevođenja. Isto pokazuje poziv metode **get( )**: nema tipa **T**, pa rezultat može biti samo **Object**.

Lako je pomisliti da su sirovo **Skladiste** i **Skladiste**<?> približno ista stvar. Metoda **neogArg( )** ističe njihovu različitost – ona otkriva istu vrstu problema, ali ih prijavljuje kao greške, a ne kao upozorenja, pošto sirovo **Skladiste** prima sve kombinacije svih tipova, dok **Skladiste**<?> prima homogenu kolekciju *određenog tipa*, i ne možete mu proslediti **Object**.

U metodama **odredjena1( )** i **odredjena2( )**, upotrebljeni su tačni generički parametri – nema džokerskih argumenata. Videćete da **odredjena2( )** zbog dodatnog argumenta ima drugačija ograničenja nego metoda **odredjena1( )**.

U metodi **divljiPodtip( )**, ograničenja tipa **Skladiste** ublažena su na **Skladiste** koje prima sve što je izvedeno iz **T**. Opet, to znači da **T** može biti **Voce**, dok bi **Skladiste** moglo biti **Skladiste<Jabuka>**. Da bi se sprečilo stavljanje objekta tipa **Narandža** u **Skladiste<Jabuka>**, zabranjen je poziv metode **set( )** (i svih drugih metoda koje kao argument primaju taj parametar tipa). Međutim, i dalje znate da sve što izađe iz objekta tipa **Skladiste<? extends Voce>** mora u najmanju ruku biti tipa **Voce**, pa je dozvoljena metoda **get( )** (i sve druge metode čija je povratna vrednost tog parametarskog tipa).

Džokerski argumenti nadtipova prikazani su u metodi **divljiNadtip( )**, koja se ponaša suprotno od metode **divljiPodtip( )**: **skladiste** može biti kontejner koji prima svaki tip koji je nadtip od **T**. Otuda metoda **set( )** prima objekat tipa **T**, pošto sve što radi sa osnovnim tipom, zbog polimorfizma radi i s njegovim izvedenim tipom (dakle i sa **T**). Međutim, poziv metode **get( )** ne pomaže, pošto tip koji sadrži **Skladiste** može biti bilo koji nadtip, pa je jedini bezbedan nadtip **Object**.

U metodi **neograniceno( )** iz ovog primera vide se i ograničenja onoga što se može i ne može uraditi s neograničenim parametrom: objekat tipa **T** ne možete ni dobiti od metode **get( )** niti proslediti metodi **set( )**, pošto tip **T** nemate.

U metodi **main( )** vidite koja od ovih metoda prima koje tipove argumenata bez grešaka i upozorenja. Radi migracijske kompatibilnosti, metoda **siroviArgumenti( )** bez ikakvih upozorenja prima sve varijante objekta **Skladiste**. I metoda **neogArg( )** jednako prihvata sve tipove, mada ih, kao što je već bilo rečeno, u telu metode obrađuje na različite načine.

Ako referencu sirovog tipa **Skladiste** prosledite metodi koja prima određen generički tip (bez džokerskih argumenata), dobićete upozorenje, zato što određeni argument očekuje informacije koje u sirovom tipu ne postoje. A ukoliko metodi **odredjena1( )** prosledite neograničenu referencu, nema informacija o tipu pomoću kojih bi se utvrdio povratni tip.

Vidite da **odredjena2( )** ima najviše ograničenja, pošto zahteva upravo **Skladiste<T>** i argument tipa T, i zato generiše greške i upozorenja ukoliko joj ne date tačne argumente. Ponekad je takvo ponašanje dobro, ali ako vas previše sputava, možete upotrebiti različite džokerske argumente, u zavisnosti od toga da li hoćete povratne vrednosti tipova određenih vašim generičkim argumentom – kao što je urađeno u metodi **divljiPodtip( )** – ili svom generičkom argumentu hoćete da prosleđujete argumente određenih tipova – kao što je urađeno u metodi **divljiNadtip( )**.

Dakle, prednost korišćenja određenih (tačnih) tipova umesto džokerskih argumenata jeste to što s generičkim parametrima možete više da uradite. Ali korišćenje džokerskih argumenata omogućuje prihvatanje šireg opsega parametrizovanih tipova kao argumenata. U svakom pojedinom slučaju sami odlučujete koji kompromis vam više odgovara.

Konverzija hvatanjem

Jedna konkretna situacija *zahteva* da upotrebite džokerski argument <?>, a ne sirov tip. Ako sirov tip prosledite metodi koja upotrebljava <?>, prevodilac može da zaključi koji je stvarni parametar tipa, tako da i ta metoda može da pozove drugu metodu koja prima

tačno taj tip. Ta tehnika se naziva *konverzija hvatanjem* (engl. *capture conversion*), jer se nespecificiran džokerski tip hvata i konvertuje u određen tip. Pokazaćemo konverziju hvatanjem u narednom primeru, gde komentari o upozorenjima važe tek kada se ukloni anotacija **@SuppressWarnings**:

```
//: genericki/KonverzijaHvatanjem.java

public class KonverzijaHvatanjem {
  static <T> void f1(Skladiste<T> skladiste) {
    T t = skladiste.get();
    System.out.println(t.getClass().getSimpleName());
  }
  static void f2(Skladiste<?> skladiste) {
    f1(skladiste); // Poziv sa uhvaćenim tipom
  }
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Skladiste sirovo = new Skladiste<Integer>(1);
    // f1(sirovo); // Prouzrokuje upozorenja
    f2(sirovo); // Nema upozorenja
    Skladiste sirovNadtip = new Skladiste();
    sirovNadtip.set(new Object()); // Upozorenje
    f2(sirovNadtip); // Nema upozorenja
    // Svođenje naviše na Skladiste<?>, koje prevodilac ipak shvata:
    Skladiste<?> sa dzokerima = new Skladiste<Double>(1.0);
    f2(sa dzokerima);
  }
} /* Ispis:
Integer
Object
Double
*///:~
```

Svi parametri tipa u **f1**( ) određeni su, bez džokerskih argumenata i granica. U **f2**( ), parametar **Skladiste** je neograničeni džokerski argument, pa se čini da zapravo nije poznato šta predstavlja. Međutim, unutar **f2**( ) poziva se **f1**( ), a **f1**( ) zahteva poznat parametar. Da bi se mogao upotrebiti u pozivu metode **f1**( ), taj parametar tipa se hvata u toku pozivanja **f2**( ).

Možda se pitate bi li se ova tehnika mogla upotrebiti za upisivanje? Ali to bi zahtevalo da zajedno s tipom **Skladiste**<?> prosledite i određen tip. Konverzija hvatanjem radi samo u situacijama gde unutar metode morate da radite s određenim tipom. Imajte u vidu da iz **f2**( ) ne možete da vratite T, jer je T za f2( ) nepoznat. Konverzija hvatanjem je zanimljiva, ali veoma ograničena.

Vežba 29: (5) Napravite generičku metodu koja kao argument prima **Skladiste**<**List**<?>>. Utvrdite koje metode možete, a koje ne možete pozivati za **Skladiste** i za **List**u. Ponovite za argument **List**<**Skladiste**<?>>.

Nedostaci

U ovom odeljku govorimo o nekim nedostacima korišćenja generičkih tipova u Javi.

Prosti tipovi ne mogu biti parametri tipa

Kao što je u prethodnom delu poglavlja bilo spomenuto, jedno od ograničenja Javinih generičkih tipova jeste to što proste tipove ne možete upotrebiti kao parametre tipa. Primera radi, ne možete napraviti **ArrayList<int>**.

Zato treba upotrebljavati omotačke klase za proste tipove zajedno sa automatskim pakovanjem Jave SE5. Ako napravite **ArrayList<Integer>** i s tim kontejnerom upotrebite proste cele brojeve (**int**), videćete da ih automatsko pakovanje automatski konvertuje u **Integer** i nazad – gotovo kao da imate **ArrayList<int>**:

```
//: generici/ListaInt.java
// Automatsko pakovanje nadoknađuje nemogućnost
// upotrebe prostih tipova u generičkom kodu.
import java.util.*;

public class ListaInt {
  public static void main(String[] args) {
    List<Integer> li = new ArrayList<Integer>();
    for(int i = 0; i < 5; i++)
      li.add(i);
    for(int i : li)
      System.out.print(i + " ");
  }
} /* Ispis:
0 1 2 3 4
*///:-
```

Obratite pažnju na to da automatsko pakovanje omogućuje čak i da foreach sintaksom pravite cele brojeve (**int**).

Po pravilu, ovo rešenje radi dobro – cele brojeve (**int**) možete da skladištite i da ih učitavate. Sve to prate automatske konverzije, ali korisnik ih ne vidi. Međutim, ako performanse programa ne zadovoljavaju, možete upotrebiti specijalizovanu verziju kontejnera prilagođenu prostim tipovima; jedna verzija njihovog otvorenog koda nalazi se na adresi **org.apache.commons.collections.primitives**.

Ovo je drugi pristup, koji pravi skup bajtova (**Set** objekata tipa **Byte**):

```
//: generici/SkupBajtova.java
import java.util.*;

public class SkupBajtova {
  Byte[] moguci = { 1,2,3,4,5,6,7,8,9 };
  Set<Byte> mojSkup =
    new HashSet<Byte>(Arrays.asList(moguci));
```

```
  // Ali ovo ne možete:
  // Set<Byte> mojSkup2 = new HashSet<Byte>(
  //   Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
} ///:~
```

Vodite računa o tome da automatsko pakovanje rešava neke probleme, ali ne sve. U sledećem primeru imamo generički interfejs **Generator** koji specificira metodu **sledeci( )** koja vraća objekat parametarskog tipa. Klasa **PNiz** sadrži generičku metodu koja pomoću generatora popunjava niz objektima (pošto je metoda statična, u ovom slučaju ne bi odgovaralo da *klasu* napravimo kao generičku). U poglavlju *Nizovi* naći ćete više realizacija interfejsa **Generator**, a u metodi **main( )** vidite kako **PNiz.popuni( )** popunjava nizove objektima:

```
//: genericki/TestProstihGenerickih.java
import net.mindview.util.*;

// Popuni niz generatorom:
class PNiz {
  public static <T> T[] popuni(T[] n, Generator<T> gen) {
    for(int i = 0; i < n.length; i++)
      n[i] = gen.sledeci();
    return n;
  }
}

public class TestProstihGenerickih {
  public static void main(String[] args) {
    String[] znakovni_nizovi = PNiz.popuni(
      new String[7], new RandomGenerator.String(10));
    for(String s : znakovni_nizovi)
      System.out.println(s);
    Integer[] celi_brojevi = PNiz.popuni(
      new Integer[7], new RandomGenerator.Integer());
    for(int i: celi_brojevi)
      System.out.println(i);
    // Ovde vam automatsko pakovanje ne može pomoći. Ovo se neće prevesti:
    // int[] b =
    //   PNiz.popuni(new int[7], new RandIntGenerator());
  }
} /* Ispis:
YNzbrnyGcF
OWZnTcQrGs
eGZMmJMRoE
suEcUOneOE
dLsmwHLGEa
hKcxrEqUCB
bkInaMesbt
7052
6665
2654
```

```
3909
5202
2209
5458
*///:~
```

Pošto **RandomGenerator.Integer** realizuje interfejs **Generator<Integer>**, nadao sam se da će automatsko pakovanje automatski konvertovati vrednost rezultata metode **sledeci( )** iz **Integer** u **int**. Međutim, automatsko pakovanje ne važi za nizove, pa to ne radi.

Vežba 30: (2) Napravite **Skladiste** za sve omotače prostih tipova i pokažite da automatsko pakovanje i raspakivanje funkcioniše za metode **set( )** i **get( )** svake instance.

Realizacija parametrizovanih interfejsa

Klasa ne može da realizuje dve varijante istog generičkog interfejsa. (Zbog brisanja, obe postaju isti interfejs.) To se događa u sledećoj situaciji:

```
//: generics/ViseVarijanataInterfejsa.java
// {CompileTimeError} (Ne može se prevesti)

interface PlacaSe<T> {}

class Zaposleni implements PlacaSe<Zaposleni> {}
class PoSatu extends Zaposleni
  implements PlacaSe<PoSatu> {} ///:~
```

PoSatu se neće prevesti zato što se klase **PlacaSe<Zaposleni>** i **PlacaSe<PoSatu>** brisanjem svode na istu klasu **PlacaSe**, pa bi gornji kôd značio da isti interfejs realizujete dvaput. Što je još zanimljivije, ako iz obe upotrebe interfejsa **PlacaSe** uklonite generičke parametre – a prevodilac baš to radi brisanjem – kôd će biti preveden.

Ovaj nedostatak ume da smeta kada radite s nekim od osnovnih Javinih interfejsa, kao što je **Comparable<T>**, što ćete videti u nastavku odeljka.

Vežba 31: (1) Uklonite svu generičnost iz programa **ViseVarijanataInterfejsa.java** i prepravite kôd tako da se prevede.

Eksplicitna konverzija tipova i upozorenja

Konvertovanje tipova i naredba **instanceof** ne utiču na parametre generičkog tipa. Sledeći kontejner interno skladišti vrednosti kao objekte tipa **Object** i konvertuje ih nazad u **T** kada ih izvadite:

```
//: generics/KonverzijaGenerickih.java

class StekNepromenljiveVelicine<T> {
  private int indeks = 0;
  private Object[] skladiste;
  public StekNepromenljiveVelicine(int velicina) {
    skladiste = new Object[velicina];
  }
```

```
  public void stavina(T stavka) { skladiste[indeks++] = stavka; }
  @SuppressWarnings("unchecked")
  public T skinisa() { return (T)skladiste[--indeks]; }
}

public class KonverzijaGenerickih {
  public static final int VELICINA = 10;
  public static void main(String[] args) {
    StekNepromenljiveVelicine<String> znakovni_nizovi =
      new StekNepromenljiveVelicine<String>(VELICINA);
    for(String s : "A B C D E F G H I J".split(" "))
      znakovni_nizovi.stavina(s);
    for(int i = 0; i < VELICINA; i++) {
      String s = znakovni_nizovi.skinisa();
      System.out.print(s + " ");
    }
  }
} /* Ispis:
J I H G F E D C B A
*///:~
```

Da nema anotacije **@SuppressWarnings**, prevodilac bi za metodu **skinisa**( ) dao upozorenje „unchecked cast" (neproverena konverzija). On zbog brisanja ne može znati da li je konverzija bezbedna, iako metoda **skinisa**( ) zapravo ne obavlja nikakvu konverziju. **T** se briše do svoje prve granice, što je podrazumevano **Object**, pa **skinisa**( ) samo konvertuje **Object** u **Object**.

Ima slučajeva kada generički kôd ne uklanja potrebu za eksplicitnom konverzijom. Tada prevodilac izdaje upozorenje, što je neumesno. Na primer:

```
//: generics/PotrebnaKonverzija.java
import java.io.*;
import java.util.*;

public class PotrebnaKonverzija {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream ulaz = new ObjectInputStream(
      new FileInputStream(args[0]));
    List<Spravica> oblici = (List<Spravica>)ulaz.readObject();
  }
} ///:~
```

Kao što ćete videti u sledećem poglavlju, metoda **readObject**( ) ne može znati šta čita, pa vraća objekat koji se mora konvertovati. Ali kada komentarišete anotaciju **@Suppress-Warnings** i prevedete program, dobićete upozorenje:

```
Note: PotrebnaKonverzija.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

A ako poslušate ta uputstva i ponovo prevedete kôd uz **-Xlint:unchecked**:

```
PotrebnaKonverzija.java:12: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.List<Spravica>
    List<Shape> oblici = (List<Spravica>)ulaz.readObject();
```

morate da konvertujete, a ipak vam kažu da ne bi trebalo. Da biste rešili ovaj problem, morate upotrebiti nov oblik konverzije uveden u Javi SE5 – konverziju preko generičke klase:

```
//: genericki/ClassKonverzija.java
import java.io.*;
import java.util.*;

public class ClassKonverzija {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream ulaz = new ObjectInputStream(
      new FileInputStream(args[0]));
      // Neće se prevesti:
//    List<Spravica> lz1 =
//    List<Spravica>.class.cast(ulaz.readObject());
    List<Spravica> lz2 = List.class.cast(ulaz.readObject());
  }
} ///:~
```

Međutim, ne možete konvertovati objekat u stvarni tip (**List<Spravica>**). Drugim rečima, ne možete reći:

```
List<Spravica>.class.cast(ulaz.readObject())
```

Pa čak i kada dodate još jednu konverziju:

```
(List<Spravica>)List.class.cast(ulaz.readObject())
```

ipak ćete dobiti upozorenje.

Vežba 32: (1) Uverite se da **StekNepromenljiveVelicine** u programu **KonverzijaGenerickih.java** generiše izuzetke kada pokušate da izađete izvan njegovih granica. Da li to znači da ne morate pisati kôd za proveru granica?

Vežba 33: (3) Popravite program **KonverzijaGenerickih.java** pomoću klase **ArrayList**.

Preklapanje

Ovo se neće prevesti, iako bi se reklo da valja pokušati:

```
//: genericki/ListaZaUpotrebu.java
// {CompileTimeError} (Neće se prevesti)
import java.util.*;
```

```
public class ListaZaUpotrebu<W,T> {
  void f(List<T> v) {}
  void f(List<W> v) {}
} ///:~
```

Brisanje prouzrokuje da obe varijante preklopljene metode imaju isti potpis.

Umesto gornjeg pristupa, metodama morate dati različita imena kada obrisani argumenti ne proizvode jedinstvene liste argumenata:

```
//: genericki/ListaZaUpotrebu2.java
import java.util.*;

public class ListaZaUpotrebu2<W,T> {
  void f1(List<T> v) {}
  void f2(List<W> v) {}
} ///:~
```

Srećom, ovu vrstu problema otkriva prevodilac.

Osnovna klasa otima interfejs

Pretpostavimo da imate klasu **Ljubimac** koja se može porediti (engl. *compare to*) sa drugim objektima tipa **Ljubimac**, zato što realizuje interfejs **Comparable**:

```
//: genericki/UporedivLjubimac.java

public class UporedivLjubimac
implements Comparable<UporedivLjubimac> {
  public int compareTo(UporedivLjubimac arg) { return 0; }
} ///:~
```

Vredelo bi suziti tip s kojim se potklasa od **UporedivLjubimac** može porediti. Primera radi, **Macka** bi trebalo da je uporediva (engl. *comparable*) samo s drugim objektima tipa **Macka**:

```
//: genericki/OtetiInterfejs.java
// {CompileTimeError} (Ne može se prevesti)

class Macka extends UporedivLjubimac implements Comparable<Macka>{
  // Greška: interfejs Comparable ne može se nasleđivati
  // s različitim argumentima: <Macka> i <Ljubimac>
  public int compareTo(Macka arg) { return 0; }
} ///:~
```

Nažalost, ovo neće raditi. Kada se interfejsu **Comparable** jednom utvrdi argument **UporedivLjubimac**, nijedna druga klasa koja ga realizuje neće moći da se poredi ni sa čim drugim sem sa objektima klase **UporedivLjubimac**:

```
//: generics/OgraniceniUporediviLjubimci.java

class Hrcak extends UporedivLjubimac
implements Comparable<UporedivLjubimac> {
  public int compareTo(UporedivLjubimac arg) { return 0; }
}

// Ili samo:

class Guster extends UporedivLjubimac {
  public int compareTo(UporedivLjubimac arg) { return 0; }
} ///:~
```

Hrcak pokazuje da je moguće ponovo realizovati isti interfejs koji je u klasi **UporedivLjubimac**, ukoliko je identičan, uključujući tu i parametarske tipove. Međutim, to je isto kao da su samo preklopljene metode u osnovnoj klasi, kao što se vidi u klasi **Guster**.

Samoograničeni tipovi

U Javinom generičkom kodu periodično se ponavlja jedan zbunjujući idiom projektovanja. Evo kako izgleda:

```
class SamoOgraniceni<T extends SamoOgraniceni<T>> { // ...
```

Ovo izaziva vrtoglavicu kao dva ogledala uperena jedno u drugo – to je neka vrsta beskonačne refleksije. Klasa **SamoOgraniceni** prima generički argument **T**, **T** je ograničen granicom, a ta granica je **SamoOgraniceni** sa argumentom **T**.

Ovaj primer je teško analizirati kada ga prvi put vidite; njime se ističe da je rezervisana reč **extends**, kada je upotrebljena s granicama, potpuno drugačija nego kada se koristi za pravljenje potklasa.

Generički kod koji se neobično ponavlja

Da biste shvatili šta znači samoograničen tip, počnimo od jednostavnije verzije ovog idioma projektovanja, one bez samoograničenja.

Generički parametar ne možete naslediti direktno. Međutim, *možete* da nasledite klasu koja taj generički parametar upotrebljava u sopstvenoj definiciji. Drugim rečima, možete reći:

```
//: generics/GenerickiTipKojiSamSebePonavlja.java

class GenerickiTip<T> {}

public class GenerickiTipKojiSamSebePonavlja
  extends GenerickiTip<GenerickiTipKojiSamSebePonavlja> {} ///:~
```

Ovo bismo mogli nazvati *generički tip koji se neobično ponavlja* (*Curiously Recurring Generics, CRG*), po Coplienovom *Šablonskom obrascu koji se neobično ponavlja* u C++-u. Ono „neobično ponavlja" odnosi se na činjenicu da se klasa, prilično neobično, pojavljuje u sopstvenoj osnovnoj klasi.

Da biste shvatili šta to znači, naglas izgovorite: „Pravim novu klasu koja nasleđuje generički tip koji ime moje klase uzima za svoj parametar". Šta generički osnovni tip može da uradi s datim imenom izvedene klase? Pa, u Javi se generički mehanizam odnosi na argumente i povratne tipove; on može da napravi osnovnu klasu koja za svoje argumente i povratne tipove upotrebljava izvedenu klasu. Isto tako, izvedenu klasu on može da upotrebi za tipove polja, iako će oni brisanjem biti svedeni na **Object**. To izražava sledeća generička klasa:

```
//: genericki/OsnovnoSkladiste.java

public class OsnovnoSkladiste<T> {
  T element;
  void set(T arg) { element = arg; }
  T get() { return element; }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
} ///:~
```

Dobili smo običan generički tip čije metode i primaju i daju objekte istog parametarskog tipa, kao i metodu koja obrađuje to uskladišteno polje (iako nad njim obavlja samo operacije tipa **Object**).

OsnovnoSkladiste možemo da upotrebimo u generičkom tipu koji se neobično ponavlja (CRG):

```
//: genericki/CRGsaOsnovnimSkladistem.java

class Podtip extends OsnovnoSkladiste<Podtip> {}

public class CRGsaOsnovnimSkladistem {
  public static void main(String[] args) {
    Podtip st1 = new Podtip(), st2 = new Podtip();
    st1.set(st2);
    Podtip st3 = st1.get();
    st1.f();
  }
} /* Ispis:
Podtip
*///:~
```

Vodite računa o nečemu važnom: nova klasa **Podtip** uzima argumente i povratne vrednosti tipa **Podtip**, a ne samo tipa osnovne klase **OsnovnoSkladiste**. To je suština CRG-a: *Osnovna klasa za svoje parametre upotrebljava izvedenu klasu.* To znači da generička osnovna klasa postaje neka vrsta šablona za zajedničku funkcionalnost svih svojih izvedenih

klasa, ali se za sve argumente i povratne vrednosti te funkcionalnosti, upotrebljava izvedeni tip. Drugim rečima, u rezultujućoj klasi se upotrebljava tačan, a ne osnovni tip. Zato su u klasi **Podtip** i argument metode **set( )** i povratni tip metode **get( )** tačno tipa **Podtip**.

Samoograničenje

OsnovnoSkladiste može da upotrebi bilo koji tip kao svoj generički parametar, kao ovde:

```
//: genericki/Neogranicena.java

class Drugo {}
class DrugoOsnovno extends OsnovnoSkladiste<Drugo> {}

public class Neogranicena {
  public static void main(String[] args) {
    DrugoOsnovno b = new DrugoOsnovno(), b2 = new DrugoOsnovno();
    b.set(new Drugo());
    Drugo drugo = b.get();
    b.f();
  }
} /* Ispis:
Drugo
*///:~
```

Samoograničenje obuhvata i dodatni korak kojim se *nameće* da generički tip bude upotrebljen kao sopstveni argument ograničenja. Pogledajte kako se rezultujuća klasa može i kako se ne može upotrebljavati:

```
//: genericki/SamoOgranicavajuci.java

class SamoOgraniceni<T extends SamoOgraniceni<T>> {
  T element;
  SamoOgraniceni<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A extends SamoOgraniceni<A> {}
class B extends SamoOgraniceni<A> {} // Takođe OK

class C extends SamoOgraniceni<C> {
  C setIGet(C arg) { set(arg); return get(); }
}

class D {}
// Ovo ne možete:
// class E extends SamoOgraniceni<D> {}
// Greška u vreme prevođenja: parametar tipa D nije unutar svojih granica
```

```
// Ovo nažalost možete da uradite, što znači da
// ovaj idiom ne možete da nametnete:
class F extends SamoOgraniceni {}

public class SamoOgranicavajuci {
  public static void main(String[] args) {
    A a = new A();
    a.set(new A());
    a = a.set(new A()).get();
    a = a.get();
    C c = new C();
    c = c.setIGet(new C());
  }
} ///:~
```

Samoograničenje zahteva da se klasa upotrebi u ovakvom odnosu nasleđivanja:

```
class A extends SamoOgraniceni<A> {}
```

Time vas prisiljava da osnovnoj klasi kao parametar prosledite klasu koju definišete.

Šta dobijate od samoograničenja parametra? Parametar tipa mora biti isti kao klasa koja se definiše. Kao što vidite u definiciji klase **B**, podtip možete izvesti i od tipa **SamoOgraniceni** čiji je parametar drugi **SamoOgraniceni**, mada izgleda da je najčešća ona upotreba koju vidite za klasu **A**. Pokušaj definisanja klase **E** pokazuje da ne možete upotrebiti parametar tipa koji nije **SamoOgraniceni**.

Nažalost, **F** će se prevesti bez upozorenja, pa se idiom samoograničenja ne može nametnuti. Ako je to zaista važno, možete upotrebiti spoljnu alatku koja će obezbediti da sirovi tipovi ne budu upotrebljeni umesto parametrizovanih.

Imajte u vidu da ograničenje možete ukloniti i sve će se klase prevesti, ali isto važi i za klasu **E**:

```
//: generics/NijeSamoOgranicena.java

public class NijeSamoOgranicena<T> {
  T element;
  NijeSamoOgranicena<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A2 extends NijeSamoOgranicena<A2> {}
class B2 extends NijeSamoOgranicena<A2> {}

class C2 extends NijeSamoOgranicena<C2> {
  C2 setIGet(C2 arg) { set(arg); return get(); }
}
```

```
class D2 {}
// Ovo je sada ispravno:
class E2 extends NijeSamoOgranicena<D2> {} ///:~
```

Dakle, ograničenje samoograničenjem služi samo da nametne odnos nasleđivanja. Ako ste upotrebili samoograničenje, znate da će parametar tipa upotrebljen u klasi biti isti osnovni tip kao klasa koja taj parametar koristi. Ono primorava sve koji upotrebljavaju tu klasu da slede taj oblik.

Samoograničenje se može upotrebiti i za generičke metode:

```
//: generics/SamoOgranicavajuceMetode.java

public class SamoOgranicavajuceMetode {
  static <T extends SamoOgraniceni<T>> T f(T arg) {
    return arg.set(arg).get();
  }
  public static void main(String[] args) {
    A a = f(new A());
  }
} ///:~
```

Time sprečavate da metoda bude primenjena na išta drugo do na samoograničeni argument prikazanog oblika.

Kovarijansa argumenata

Vrednost samoograničavajućih tipova jeste to što proizvode *kovarijantne tipove argumenata* – tipovi argumenata metoda variraju kao njihove potklase.

Iako samoograničavajući tipovi proizvode i povratne tipove koji su isti kao tip potklase, to nije toliko važno, pošto je Java SE5 uvela *kovarijantne povratne tipove*:

```
//: generics/KovarijantniPovratniTipovi.java

class Osnovna {}
class Izvedena extends Osnovna {}

interface ObicnaDajOb {
  Osnovna dajob();
}

interface IzvedenaDajOb extends ObicnaDajOb {
  // Povratni tip redefinisane metode sme da se menja:
  Izvedena dajob();
}

public class KovarijantniPovratniTipovi {
  void test(IzvedenaDajOb d) {
    Izvedena d2 = d.dajob();
  }
} ///:~
```

Metoda **dajob( )** interfejsa **IzvedenaDajOb** redefiniše metodu **dajob( )** interfejsa **ObicnaDajOb** *i* vraća tip izveden iz tipa koji vraća **ObicnaDajOb.dajob( )**. Iako je to sasvim logično – trebalo bi da metoda izvedenog tipa može da vraća specifičniji tip od metode osnovnog tipa koju redefiniše – nije bilo dozvoljeno u prethodnim verzijama Jave.

Samoograničena metoda zaista daje tačan izvedeni tip kao povratnu vrednost, što vidite ovde, u metodi **dajob( )**:

```
//: generici/GenerickeMetodeIPovratniTipovi.java

interface GenerickaDajOb<T extends GenerickaDajOb<T>> {
  T dajob();
}

interface DajOb extends GenerickaDajOb<DajOb> {}

public class GenerickeMetodeIPovratniTipovi {
  void test(DajOb g) {
    DajOb rezultat = g.dajob();
    GenerickaDajOb gg = g.dajob(); // Takođe osnovni tip
  }
} ///:~
```

Obratite pažnju na to kako se ovaj kôd ne bi preveo da u Javu SE5 nisu bili uvedeni kovarijantni povratni tipovi.

Međutim, u negeneričkom kodu, tipove *argumenata* ne možete naterati da se menjaju s podtipovima:

```
//: generici/ObicniArgumenti.java

class ObicnaZadajOb {
  void set(Osnovna osnovna) {
    System.out.println("ObicnaZadajOb.zadajOb(Osnovna)");
  }
}

class IzvedenaZadajOb extends ObicnaZadajOb {
  void zadajOb(Izvedena izvedena) {
    System.out.println("IzvedenaZadajOb.zadajOb(Izvedena)");
  }
}

public class ObicniArgumenti {
  public static void main(String[] args) {
    Osnovna osnovna = new Osnovna();
    Izvedena izvedena = new Izvedena();
    IzvedenaZadajOb ds = new IzvedenaZadajOb();
    ds.zadajOb(izvedena);
    ds.zadajOb(osnovna); // Prevešće se: preklopljena je,
    // nije redefinisana!
```

```
  }
} /* Ispis:
IzvedenaZadajOb.zadajOb(Izvedena)
ObicnaZadajOb.zadajOb(Osnovna)
*///:~
```

Dozvoljene su i **zadajOb(izvedena)** i **zadajOb(osnovna)**, pa **IzvedenaZadajOb.zadajOb( )** ne redefiniše metodu **ObicnaZadajOb.zadajOb( )**, nego je *preklapa*. Iz rezultata vidite da u klasi **IzvedenaZadajOb** postoje dve metode, pa je verzija osnovne klase i dalje dostupna, što dokazuje da je bila preklopljena.

Međutim, uz samoograničavajuće tipove postoji samo jedna metoda u izvedenoj klasi, i ona kao svoj argument uzima izvedeni, a ne osnovni tip:

```
//: generici/SamoOgranicavanjeIKovarijantniArgumenti.java

interface SamoOgranicenaZadajOb<T extends SamoOgranicenaZadajOb<T>> {
  void zadajOb(T arg);
}

interface ZadajOb extends SamoOgranicenaZadajOb<ZadajOb> {}

public class SamoOgranicavanjeIKovarijantniArgumenti {
  void testA(ZadajOb s1, ZadajOb s2, SamoOgranicenaZadajOb soz) {
    s1.zadajOb(s2);
    // s1.zadajOb(soz); // Pogrešno:
    // zadajOb(ZadajOb) u SamoOgranicenaZadajOb<ZadajOb>
    // ne može biti primenjena na (SamoOgranicenaZadajOb)
  }
} ///:~
```

Prevodilac ne priznaje pokušaj da se kao argument metode **zadajOb( )** prosledi osnovni tip, pošto ne postoji metoda s takvim potpisom. U stvari, argument je redefinisan.

Ako nema samoograničavanja, uskače uobičajeni mehanizam nasleđivanja i dobijate preklapanje, kao u negeneričkom slučaju:

```
//: generici/ObicnoGenerickoNasledjivanje.java

class GenerickaZadajOb<T> { // Nije samoograničena
  void zadajOb(T arg){
    System.out.println("GenerickaZadajOb.zadajOb(Osnovna)");
  }
}

class IzvedenaZO extends GenerickaZadajOb<Osnovna> {
  void zadajOb(Izvedena izvedena){
    System.out.println("IzvedenaZO.zadajOb(Izvedena)");
  }
}
```

```
public class ObicnoGenerickoNasledjivanje {
  public static void main(String[] args) {
    Osnovna osnovna = new Osnovna();
    Izvedena izvedena = new Izvedena();
    IzvedenaZO dgs = new IzvedenaZO();
    dgs.zadajOb(izvedena);
    dgs.zadajOb(osnovna); // Prevešće se: preklopljena je,
    // nije redefinisana!
  }
} /* Ispis:
IzvedenaZO.zadajOb(Izvedena)
GenerickaZadajOb.zadajOb(Osnovna)
*///:~
```

Ovaj kôd podražava program **ObicniArgumenti.java**; u tom primeru, **IzvedenaZadajOb** nasleduje klasu **ObicnaZadajOb** koja sadrži svoju metodu **zadajOb(Osnovna)**. Ovde, **IzvedenaZO** nasleđuje klasu **GenerickaZadajOb<Osnovna>** koja takođe sadrži svoju metodu **zadajOb(Osnovna)** a nju je napravila spomenuta generička klasa. Kao u programu **ObicniArgumenti.java**, iz rezultata vidite da **IzvedenaZO** sadrži dve preklopljene verzije metode **zadajOb( )**. Bez samoograničavanja, tipove argumenata preklapate. Ako upotrebite samoograničavanje, dobijate samo jednu verziju metode koja prima argument tačnog (određenog) tipa.

Vežba 34: (4) Napravite samoograničen generički tip koji sadrži apstraktnu metodu koja prima argument tipa generičkog parametra i proizvodi povratnu vrednost istog tipa generičkog parametra. U neapstraktnoj metodi klase pozovite apstraktnu metodu i vratite njen rezultat. Nasledite samoograničeni tip i testirajte rezultujuću klasu.

Dinamička bezbednost tipova

Pošto generičke kontejnere možete prosleđivati kodu napisanom pre pojave Jave SE5, postoji mogućnost da stari kôd pokvari vaše kontejnere. Java SE5 u paketu **java.util.Collections** ima skup uslužnih metoda za rešavanje problema s proverom tipova u toj situaciji: statične metode **checkedCollection( )**, **checkedList( )**, **checkedMap( )**, **checkedSet( )**, **checkedSortedMap( )** i **checkedSortedSet( )**. Svaka od njih prima kontejner koji hoćete da dinamički ispitate kao svoj prvi argument i tip koji hoćete da nametnete kao svoj drugi argument.

Proveravani kontejner će generisati izuzetak **ClassCastException** na mestu gde pokušate da *umetnete* neodgovarajući objekat, za razliku od pre-generičkog (sirovog) kontejnera koji bi vas obavestio da postoji problem prilikom *vađenja* objekta. U tom slučaju znate da postoji problem, ali ne znate ko ga je prouzrokovao. Proveravani kontejneri omogućuju da saznate ko je pokušao da umetne neodgovarajući objekat.

Razmotrićemo problem „stavljanja mačke u listu pasa" koji realizuje proveravani kontejner. Ovde zastareli kôd predstavlja **metodaUStaromStilu( )** zato što uzima sirovu **Listu**, a anotacija **@SuppressWarnings("unchecked")** potrebna je da bi se sprečilo rezultujuće upozorenje:

```
//: generics/ProveravanaLista.java
// Koristimo metodu Collection.checkedList().
import podaciotipu.ljubimci.*;
import java.util.*;

public class ProveravanaLista {
  @SuppressWarnings("unchecked")
  static void metodaUStaromStilu(List verovatnoPsi) {
    verovatnoPsi.add(new Macka());
  }
  public static void main(String[] args) {
    List<Pas> psi1 = new ArrayList<Pas>();
    metodaUStaromStilu(psi1); // Ćutke prihvata Macka
    List<Pas> psi2 = Collections.checkedList(
      new ArrayList<Pas>(), Pas.class);
    try {
      metodaUStaromStilu(psi2); // Generiše izuzetak
    } catch(Exception e) {
      System.out.println(e);
    }
    // Izvedeni tipovi dobro rade:
    List<Ljubimac> ljubimci = Collections.checkedList(
      new ArrayList<Ljubimac>(), Ljubimac.class);
    ljubimci.add(new Pas());
    ljubimci.add(new Macka());
  }
} /* Ispis:
java.lang.ClassCastException: Attempt to insert class
podaciotipu.ljubimci.Macka element into collection with element type class
podaciotipu.ljubimci.Pas
(java.lang.ClassCastException: Pokušaj umetanja klase
podaciotipu.ljubimci.Macka u kontejner čiji tip određuje klasa
podaciotipu.ljubimci.Pas)
*///:~
```

Kada pokrenete program, videćete da lista **psi1** ćutke podnosi umetanje objekta tipa **Macka**, dok lista **psi2** odmah generiše izuzetak kada se umetne neodgovarajući tip. Vidite i to da se u kontejner koji proverava na osnovni tip mogu stavljati objekti izvedenog tipa.

Vežba 35: (1) Prepravite program **ProveravanaLista.java** tako da upotrebljava klase **Kafa** definisane u ovom poglavlju.

Izuzeci

U generičkom kodu se, zbog brisanja, izuzeci veoma malo upotrebljavaju. Blok **catch** ne može da hvata izuzetke generičkog tipa, zato što tačan tip izuzetka mora biti poznat i u vreme prevođenja i u vreme izvršavanja. Takođe, generička klasa ne može ni posredno ni neposredno da nasledi klasu **Throwable** (i time opet sprečava da definišete generički izuzetak koji se ne može uhvatiti).

Međutim, u bloku **throws** unutar deklaracije metode mogu se upotrebljavati parametri tipa. To omogućuje da napišite generički kôd koji se menja u zavisnosti od tipa proveravanog izuzetka:

```
//: generics/GenerisanjeGenerickog1zuzetka.java
import java.util.*;

interface Preradjivac<T,E extends Exception> {
  void obrada(List<T> kontejnerRezultata) throws E;
}

class PokretacObrada<T,E extends Exception>
extends ArrayList<Preradjivac<T,E>> {
  List<T> obradiSve() throws E {
    List<T> kontejnerRezultata = new ArrayList<T>();
    for(Preradjivac<T,E> preradjivac : this)
      preradjivac.obrada(kontejnerRezultata);
    return kontejnerRezultata;
  }
}

class Greska1 extends Exception {}

class Preradjivac1 implements Preradjivac<String,Greska1> {
  static int broj = 3;
  public void
  obrada(List<String> kontejnerRezultata) throws Greska1 {
    if(broj-- > 1)
      kontejnerRezultata.add("Hep!");
    else
      kontejnerRezultata.add("Ho!");
    if(broj < 0)
      throw new Greska1();
  }
}

class Greska2 extends Exception {}

class Preradjivac2 implements Preradjivac<Integer,Greska2> {
  static int broj = 2;
  public void
  obrada(List<Integer> kontejnerRezultata) throws Greska2 {
    if(broj-- == 0)
      kontejnerRezultata.add(47);
    else {
      kontejnerRezultata.add(11);
    }
    if(broj < 0)
      throw new Greska2();
```

```
  }
}

public class GenerisanjeGenerickogIzuzetka {
  public static void main(String[] args) {
    PokretacObrada<String,Greska1> pokretac =
      new PokretacObrada<String,Greska1>();
    for(int i = 0; i < 3; i++)
      pokretac.add(new Preradjivac1());
    try {
      System.out.println(pokretac.obradiSve());
    } catch(Greska1 e) {
      System.out.println(e);
    }

    PokretacObrada<Integer,Greska2> pokretac2 =
      new PokretacObrada<Integer,Greska2>();
    for(int i = 0; i < 3; i++)
      pokretac2.add(new Preradjivac2());
    try {
      System.out.println(pokretac2.obradiSve());
    } catch(Greska2 e) {
      System.out.println(e);
    }
  }
} ///:~
```

Preradjivac poziva metodu **obrada( )** i može generisati izuzetak tipa **E**. Rezultat metode **obrada( )** smešta se u **List<T> kontejnerRezultata** (to nazivamo *parametar kolekcije*). Klasa **PokretacObrada** ima metodu **obradiSve( )** koja izvršava sve objekte tipa **Obrada** koje klasa sadrži i vraća **kontejnerRezultata**.

Da ne možete parametrizovati generisane izuzetke, ne biste mogli napisati ovaj kôd generički, i to zbog proveravanih izuzetaka.

Vežba 36: (2) Dodajte drugi parametrizovani izuzetak klasi **Preradjivac** i pokažite da izuzeci mogu da se menjaju nezavisno.

Miksini

Termin *miksin* (engl. *mixin*) s vremenom je dobio razna značenja, ali osnovno je: mešanje mogućnosti više klasa da bi se dobila rezultujuća klasa koja predstavlja sve umešane tipove, tj. ona se naziva mešavina ili miksin. To se obično radi u zadnji čas, što je čini prigodnom alatkom za lako sastavljanje klasa.

Jedna od prednosti miksina jeste to što oni karakteristike i ponašanja dosledno primenjuju na razne klase. Uz to, kada nešto promenite u miksin klasi, te promene se prenose na sve klase na koje se miksin primeni. Zato su miksini značajan deo *aspektno orijentisanog programiranja* (AOP), a za rešavanje problema s mešavinama često se predlažu razni aspekti.

Miksini u jeziku C++

Jedan od najjačih razloga za uvođenje višestrukog nasleđivanja u C++-u jeste njegova upotreba za miksine. Međutim, zanimljiviji i elegantniji pristup miksinima koristi parametrizovane tipove, gde je miksin klasa koja nasleđuje svoj parametar tipa. U C++-u se miksini lako prave jer C++ pamti tip parametara svojih šablona.

Evo primera s dva tipa miksina iz jezika C++: jedan služi za dodavanje svojstva posedovanja vremenske oznake (engl. *time stamp*) a drugi dodaje serijski broj svake instance objekta:

```
//: genericki/Miksini.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class SVremenskomOznakom : public T {
  long vremenskaOznaka;
public:
  SVremenskomOznakom() { vremenskaOznaka = time(0); }
  long dajOznaku() { return vremenskaOznaka; }
};

template<class T> class SaSerijskimBrojem : public T {
  long serijskiBroj;
  static long brojac;
public:
  SaSerijskimBrojem() { serijskiBroj = brojac++; }
  long dajSerijskiBroj() { return serijskiBroj; }
};

// Definicija i inicijalizacija statične memorije:
template<class T> long SaSerijskimBrojem<T>::brojac = 1;

class Osnovni {
  string vrednost;
public:
  void postavi(string vre) { vrednost = vre; }
  string daj() { return vrednost; }
};

int main() {
  SVremenskomOznakom<SaSerijskimBrojem<Osnovni> > miksin1, miksin2;
  miksin1.postavi("ispitni znakovni niz 1");
  miksin2.postavi("ispitni znakovni niz 2");
  cout << miksin1.daj() << " " << miksin1.dajOznaku() <<
    " " << miksin1.dajSerijskiBroj() << endl;
  cout << miksin2.daj() << " " << miksin2.dajOznaku() <<
    " " << miksin2.dajSerijskiBroj() << endl;
```

```
} /* Ispis: (primer)
ispitni znakovni niz 1 1129840250 1
ispitni znakovni niz 2 1129840250 2
*///:~
```

U metodi **main( )**, rezultujući tip objekata **miksin1** i **miksin2** ima sve metode umešanih tipova. Smatrajte miksin funkcijom koja preslikava postojeće klase na nove potklase. Obratite pažnju na to kako se pomoću ove tehnike lako prave miksini; u suštini, vi samo kažete: „Evo šta hoću", i to se izvrši:

```
SVremenskomOznakom<SaSerijskimBrojem<Osnovni> > miksin1, miksin2;
```

Nažalost, Javin generički mehanizam to ne dozvoljava. Brisanje uklanja tip osnovne klase, pa generička klasa ne može direktno da nasledi generički parametar.

Mešanje pomoću interfejsa

Često se predlaže da se efekat miksina postigne pomoću interfejsa, i to na ovaj način:

```
//: generici/Miksini.java
import java.util.*;

interface SVremenskomOznakom { long dajOznaku(); }

class RzcSVremenskomOznakom implements SVremenskomOznakom {
  private final long vremenskaOznaka;
  public RzcSVremenskomOznakom() {
    vremenskaOznaka = new Date().getTime();
  }
  public long dajOznaku() { return vremenskaOznaka; }
}

interface SaSerijskimBrojem { long dajSerijskiBroj(); }

class RzcSaSerijskimBrojem implements SaSerijskimBrojem {
  private static long brojac = 1;
  private final long serijskiBroj = brojac++;
  public long dajSerijskiBroj() { return serijskiBroj; }
}

interface Osnovni {
  public void postavi(String vre);
  public String daj();
}

class RzcOsnovnog implements Osnovni {
  private String vrednost;
  public void postavi(String vre) { vrednost = vre; }
  public String daj() { return vrednost; }
}
```

```
class Miksin extends RzcOsnovnog
implements SVremenskomOznakom, SaSerijskimBrojem {
  private SVremenskomOznakom vremenskaOznaka
  = new RzcSVremenskomOznakom();
  private SaSerijskimBrojem serijskiBroj =
    new RzcSaSerijskimBrojem();
  public long dajOznaku() { return vremenskaOznaka.dajOznaku(); }
  public long dajSerijskiBroj() {
    return serijskiBroj.dajSerijskiBroj();
  }
}

public class Miksini {
  public static void main(String[] args) {
    Miksin miksin1 = new Miksin(), miksin2 = new Miksin();
    miksin1.postavi("ispitni znakovni niz 1");
    miksin2.postavi("ispitni znakovni niz 2");
    System.out.println(miksin1.daj() + " " +
      miksin1.dajOznaku() +  " " + miksin1.dajSerijskiBroj());
    System.out.println(miksin2.daj() + " " +
      miksin2.dajOznaku() +  " " + miksin2.dajSerijskiBroj());
  }
} /* Ispis: (primer)
ispitni znakovni niz 1 1132437151359 1
ispitni znakovni niz 2 1132437151359 2
*///:~
```

Klasa **Miksin** u osnovi koristi *delegiranje*, pa je za svaki umešani tip neophodno polje u objektu tipa **Miksin**. Da biste odgovarajućem objektu prosledili pozive, u klasi **Miksin** morate da napišete sve potrebne metode. U primeru su upotrebljene trivijalne klase, ali sa složenijim miksinom, kôd veoma brzo raste.[4]

Vežba 37: (2) U program **Miksini.java** dodajte novu miksin klasu **Obojen**, ubacite je u **Miksin** i pokažite da radi.

Korišćenje obrasca Decorator

Kada pogledate način na koji se koristi, koncept miksina izgleda tesno povezan sa projektnim obrascem *Decorator* (Dekorator).[5] Kada uobičajeno pravljenje potklasa daje toliko klasa da postizanje svih mogućih kombinacija postaje nepraktično, često se koriste dekoratori.

Obrazac *Decorator* opisuje korišćenje slojevitih objekata za dinamičko i nevidljivo dodavanje dogovornosti objektima pojedinačno. *Decorator* specificira da svi objekti koji se omotavaju oko vašeg početnog objekta imaju isti osnovni interfejs. Objekat može imati

---

[4] Imajte u vidu da neka razvojna okruženja, kao što su Eclipse i IntelliJ Idea, automatski generišu kôd za delegiranje.

[5] Uzorci su tema knjige *Thinking in Patterns (with Java)* koju možete naći na adresi *www.MindView.net*. Pogledajte i knjigu *Design Patterns*, autori su Erich Gamma i dr. (Addison-Wesley, 1995).

svojstvo „može se dekorisati" (*dekorabilnost*), a slojeve funkcionalnosti dodajete omotavanjem drugih klasa oko dekorabilnog objekta. Tako korišćenje dekoratora postaje nevidljivo – postoji skup zajedničkih poruka koje možete slati objektu, bez obzira na to da li je on bio dekorisan ili nije. Klasa za dekorisanje može da dodaje i metode, ali kao što ćete videti, u ograničenom obimu.

Dekoratori se realizuju pomoću kompozicije i formalnih struktura (hijerarhije dekorabilan/dekorator), dok se miksini realizuju nasleđivanjem. Stoga miksine, zasnovane na parametrizovanim tipovima, možete smatrati generičkim mehanizmom dekoratora koji ne zahtevaju strukturu nasleđivanja projektnog obrasca *Decorator*.

Napisaćemo prethodni primer pomoću projektnog obrasca *Decorator*:

```
//: generici/dekorator/Dekorisanje.java
package generici.dekorator;
import java.util.*;

class Osnovni {
  private String vrednost;
  public void set(String vre) { vrednost = vre; }
  public String daj() { return vrednost; }
}

class Dekorator extends Osnovni {
  protected Osnovni osnovni;
  public Dekorator(Osnovni osnovni) { this.osnovni = osnovni; }
  public void set(String vre) { osnovni.set(vre); }
  public String daj() { return osnovni.daj(); }
}

class SVremenskomOznakom extends Dekorator {
  private final long vremenskaOznaka;
  public SVremenskomOznakom(Osnovni osnovni) {
    super(osnovni);
    vremenskaOznaka = new Date().getTime();
  }
  public long dajOznaku() { return vremenskaOznaka; }
}

class SaSerijskimBrojem extends Dekorator {
  private static long brojac = 1;
  private final long serijskiBroj = brojac++;
  public SaSerijskimBrojem(Osnovni osnovni) { super(osnovni); }
  public long dajSerijskiBroj() { return serijskiBroj; }
}

public class Dekorisanje {
  public static void main(String[] args) {
    SVremenskomOznakom t = new SVremenskomOznakom(new Osnovni());
    SVremenskomOznakom t2 = new SVremenskomOznakom(
```

```
      new SaSerijskimBrojem(new Osnovni()));
    //! t2.dajSerijskiBroj(); // Nije dostupno
    SaSerijskimBrojem s = new SaSerijskimBrojem(new Osnovni());
    SaSerijskimBrojem s2 = new SaSerijskimBrojem(
      new SVremenskomOznakom(new Osnovni()));
    //! s2.dajOznaku(); // Nije dostupno
  }
} ///:~
```

Klasa koja se dobija korišćenjem miksina sadrži sve potrebne metode, ali je tip objekta dobijenog pomoću dekoratora jednak poslednjem tipu kojim je objekat bio dekorisan. Drugim rečima, iako je *moguće* dodati više slojeva, stvarni tip zadaje samo poslednji sloj i vidljive su samo njegove metode, dok je tip miksina jednak *svim* umešanim tipovima. Zato je značajan nedostatak Decoratora to što on efektivno radi samo s jednim (poslednjim) slojem dekoracije, a i korišćenje miksina je verovatno prirodnije. Dakle, Decorator tek ograničeno rešava problem koji potpuno rešavaju miksini.

Vežba 38: (4) Napravite jednostavan sistem Decoratora koji počinje od obične kafe, na koju možete primeniti dekoratore mleka, pene, čokolade, karamela i šlaga.

Miksini s dinamičkim posrednicima

Pomoću dinamičkog posrednika može se napraviti mehanizam koji bolje modeluje miksine od Decoratora (objašnjenje načina rada Javinih dinamičkih posrednika pročitajte u poglavlju *Podaci o tipu*). Uz dinamički posrednik, *dinamički* tip rezultujuće klase jednak je kombinaciji umešanih tipova.

Zbog ograničenja kojima podležu dinamički posrednici, svaka umešana klasa mora biti realizacija nekog interfejsa:

```
//: generics/MiksinDinamickimPosrednikom.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Entorka.*;

class PosrednikZaMiksin implements InvocationHandler {
  Map<String,Object> delegiranoMetodom;
  public PosrednikZaMiksin(Dvojka<Object,Class<?>>... parovi) {
    delegiranoMetodom = new HashMap<String,Object>();
    for(Dvojka<Object,Class<?>> par : parovi) {
      for(Method metoda : par.drugi.getMethods()) {
        String imeMetode = metoda.getName();
        // Metodu realizuje prvi interfejs u mapi
        if (!delegiranoMetodom.containsKey(imeMetode))
          delegiranoMetodom.put(imeMetode, par.prvi);
      }
    }
  }
```

```
  public Object invoke(Object posrednik, Method metoda,
    Object[] args) throws Throwable {
    String imeMetode = metoda.getName();
    Object delegiraj = delegiranoMetodom.daj(imeMetode);
    return metoda.invoke(delegiraj, argumenti);
  }
  @SuppressWarnings("unchecked")
  public static Object newInstance(Dvojka... parovi) {
    Class[] interfejsi = new Class[parovi.length];
    for(int i = 0; i < parovi.length; i++) {
      interfejsi[i] = (Class)parovi[i].drugi;
    }
    ClassLoader cl =
      parovi[0].prvi.getClass().getClassLoader();
    return Proxy.newProxyInstance(
      cl, interfejsi, new PosrednikZaMiksin(parovi));
  }
}

public class MiksinDinamickimPosrednikom {
  public static void main(String[] args) {
    Object miksin = PosrednikZaMiksin.newInstance(
      n_torka(new RzcOsnovnog(), Osnovni.class),
      n_torka(new RzcSVremenskomOznakom(), SVremenskomOznakom.class),
      n_torka(new RzcSaSerijskimBrojem(),SaSerijskimBrojem.class));
    Osnovni b = (Osnovni)miksin;
    SVremenskomOznakom t = (SVremenskomOznakom)miksin;
    SaSerijskimBrojem s = (SaSerijskimBrojem)miksin;
    b.set("Zdravo");
    System.out.println(b.daj());
    System.out.println(t.dajOznaku());
    System.out.println(s.dajSerijskiBroj());
  }
} /* Ispis: (primer)
Zdravo
1132519137015
1
*///:~
```

Za razliku od statičnog tipa, samo dinamički tip sadrži sve umešane tipove, pa ovo i dalje nije onako lepo kao C++-ov pristup, jer morate da svedete naniže na odgovarajući tip pre nego što pozovete metode za njega. Međutim, znatno je bliže pravom miksinu.

U podršku miksina u Javi uloženo je prilično mnogo rada. Eksplicitno za tu svrhu napravljen je i najmanje jedan softverski dodatak, jezik Jam.

Vežba 39: (1) U program **MiksinDinamickimPosrednikom.java** dodajte novu miksin klasu **Obojen**, umešajte je u **miksin** i dokažite da radi.

Latentni tipovi

Na početku ovog poglavlja predstavili smo ideju pisanja koda koji se može primeniti uopšteno u najvećoj mogućoj meri. Da bismo to postigli, moramo ublažiti ograničenja tipova s kojima naš kôd radi, a da ne izgubimo prednosti statične provere (bezbednosti) tipova. Tada ćemo moći da pišemo kôd koji se bez izmena može upotrebljavati u više situacija – tj. opštiji kôd.

Izgleda da Javin generički kôd pravi još jedan korak u tom smeru. Kada pišete ili koristite generički kôd koji samo čuva objekte, on radi sa svim tipovima (sem prostih, iako smo videli da automatsko pakovanje to izglađuje). Ili, drugim rečima, generičko skladište može da kaže: „Meni je svejedno koji si tip". Kôd kojem nije važno s kojim tipovima radi, zaista se može primeniti svuda i stoga je potpuno opšti (generički).

Kao što ste takođe videli, problem nastaje kada hoćete da obrađujete generičke tipove (ukoliko se to ne može obaviti pozivanjem metoda klase **Object**), jer brisanje zahteva da zadate granice generičkih tipova koji se mogu upotrebiti, da bi bilo moguće bezbedno pozivati konkretne metode za generičke objekte u vašem kodu. To je značajno ograničenje pojma opštosti, pošto svoje generičke tipove morate ograničiti tako da nasleđuju određene klase ili realizuju određene interfejse. U nekim slučajevima, možda ćete umesto generičkih klasa i interfejsa upotrebiti obične klase ili interfejse, pošto se ograničeni generički tip ne mora razlikovati od specificiranja klase ili interfejsa.

Neki programski jezici to rešavaju tako što koriste *latentne* ili *strukturirane tipove*. Malo neobavezniji termin je *duck typing*, kao u izreci „If it walks like a duck and talks like a duck, you might as well treat it like a duck". (Ako hoda kao patka i zvuči kao patka, onda smatrajte da je to patka.) Duck typing je postao prilično popularan termin, možda zato što za sobom ne vuče istorijski prtljag kao prethodna dva termina.

Generički kôd obično poziva tek nekoliko metoda za određen generički tip, a jezik s latentnim tipovima ublažuje to ograničenje (i daje opštiji kôd) tako što zahteva realizovanje samo nekog podskupa metoda, a *ne* određene klase ili interfejsa. Latentni tipovi omogućuju rad s raznim hijerarhijama klasa jer se pozivaju metode koje nisu deo zajedničkog interfejsa. Stoga parče koda može zapravo reći: „Ako možeš **govoriti**( ) i **sedeti**( ), meni je svejedno koji si tip". Ukoliko ne zahteva određen tip, kôd je opštiji.

Latentni tipovi su mehanizam za raspoređivanje i ponovno korišćenje koda. Višekratno upotrebljiv kôd je lakše pisati s njima nego bez njih. Raspoređivanje i ponovno korišćenje koda jesu osnovni principi sveg programiranja: napiši kôd jedanput, koristi ga više puta i čuvaj na jednom mestu. Pošto ne moram da navedem određeni interfejs koji moj kôd obrađuje, latentni tipovi mi omogućuju da pišem manje koda i da ga lakše primenjujem na više mesta.

Od jezika koji podržavaju latentne tipove, dva su Python (možete ga besplatno preuzeti na adresi *www.Python.org*) i C++.[6] Python tipove proverava dinamički (gotovo sve proveravanje tipova obavlja se u vreme izvršavanja), a C++ statički (u vreme prevođenja). Dakle, latentni tipovi ne zahtevaju ni statičko ni dinamičko proveravanje tipova.

[6] Latentne tipove podržavaju i jezici Ruby i Smalltalk.

Ako uzmemo gornji opis i izrazimo ga na Pythonu, dobićemo ovo:

```
#: genericki/PsiIRoboti.py

class Pas:
    def govoriti(self):
        print "Av!"
    def sedeti(self):
        print "Sedim"
    def reprodukovatise(self):
        pass

class Robot:
    def govoriti(self):
        print "Klik!"
    def sedeti(self):
        print "Klank!"
    def promenaUlja(self):
        pass

def obavi(bilosta):
    bilosta.govoriti()
    bilosta.sedeti()

a = Pas()
b = Robot()
obavi(a)
obavi(b)
#:~
```

Python određuje oblast važenja na osnovu uvučenosti od leve margine (pa vitičaste zagrade nisu potrebne), a dvotačkom označava početak nove oblasti važenja. Znakom # označava se komentar do kraja reda, kao // u Javi. Metode klasa svojim prvim argumentom eksplicitno zadaju ekvivalent reference **this** koji se po konvenciji naziva **self**. Pozivi konstruktora ne zahtevaju bilo koju vrstu rezervisane reči **new**. U jeziku Python mogu se koristiti i obične funkcije (koje nisu članice), kao što možete videti u funkciji **obavi( )**.

U **obavi(bilosta)**, obratite pažnju na to da **bilosta** nema tip i da je samo identifikator. Pošto **bilosta** mora biti u stanju da obavi operacije koje od njega zahteva funkcija **obavi( )**, impliciran je neki interfejs. Ali taj interfejs ne morate eksplicitno da napišete – on je *latentan*. Funkciji **obavi( )** nije važno koji je tip njenog argumenta, pa joj se može proslediti svaki objekat koji podržava metode **govoriti( )** i **sedeti( )**. Ako funkciji **obavi( )** prosledite objekat koji ne podržava te operacije, dobićete izuzetak u vreme izvršavanja.

Isti učinak možemo da postignemo na C++-u:

```
//: genericki/PsiIRoboti.cpp

class Pas {
public:
  void govoriti() {}
```

```
  void sedeti() {}
  void reprodukovatise() {}
};

class Robot {
public:
  void govoriti() {}
  void sedeti() {}
  void promenaUlja() {
};

template<class T> void obavi(T bilosta) {
  bilosta.govoriti();
  bilosta.sedeti();
}

int main() {
  Pas d;
  Robot r;
  obavi(d);
  obavi(r);
} ///:~
```

I u Pythonu i u C++-u, **Pas** i **Robot** nemaju ništa zajedničko, sem što imaju dve metode sa identičnim potpisima. Sa stanovišta tipova, to su sasvim različiti tipovi. Međutim, funkciji **obavi**( ) nije važno koji je tip njenog argumenta, a latentnost tipova joj omogućuje da prihvati objekte oba tipa.

C++ proverava može li zaista da pošalje te poruke. Ako pokušate da prosledite pogrešan tip, prevodilac će vam ispisati poruku o grešci (te poruke o greškama su oduvek bile užasne i opširne, i osnovni su razlog što C++-ovi šabloni imaju lošu reputaciju). Iako to rade u različita vremena – C++ u vreme prevođenja, a Python u vreme izvršavanja – oba jezika proveravaju upotrebu tipova, pa ih smatraju *strogo tipiziranim* jezicima.[7] Latentni tipovi ne ugrožavaju strogu tipiziranost.

Pošto su generički tipovi u Javu dodati naknadno, nije bilo šanse da se realizuje bilo koja vrsta latentnih tipova, pa Java nema tu mogućnost. Zato isprva izgleda kao da je Javin generički mehanizam „manje generički" od jezika koji podržavaju latentne tipove.[8] Na primer, ako gornji primer pokušamo da realizujemo u Javi, moraćemo da upotrebimo klasu ili interfejs koje ćemo specificirati u izrazu za granice:

```
//: generics/Obavlja.java

public interface Obavlja {
  void govoriti();
  void sedeti();
} ///:~
```

[7] Pošto je u njemu dozvoljena eksplicitna konverzija tipova koja u suštini onemogućuje sistem tipova, neki tvrde da je C++ slabo tipiziran jezik, ali su to ekstremna stanovišta. Verovatno je tačnije reći da je C++ „strogo tipiziran jezik sa skrivenim vratima".

[8] Javinu realizaciju generičkih tipova brisanjem, katkada nazivaju *drugorazredni* generički tipovi.

```
//: generics/PsiIRoboti.java
// U Javi nema latentnih tipova
import podaciotipu.ljubimci.*;
import static net.mindview.util.Print.*;

class CirkuskiPas extends Pas implements Obavlja {
  public void govoriti() { print("Vau!"); }
  public void sedeti() { print("Sedim"); }
  public void reprodukovatise() {}
}

class Robot implements Obavlja {
  public void govoriti() { print("Klik!"); }
  public void sedeti() { print("Klank!"); }
  public void promenaUlja() {}
}

class Komuniciraj {
  public static <T extends Obavlja>
  void obavi(T izvodjac) {
    izvodjac.govoriti();
    izvodjac.sedeti();
  }
}

public class PsiIRoboti {
  public static void main(String[] args) {
    CirkuskiPas d = new CirkuskiPas();
    Robot r = new Robot();
    Komuniciraj.obavi(d);
    Komuniciraj.obavi(r);
  }
} /* Ispis:
Vau!
Sedim
Klik!
Klank!
*///:~
```

Međutim, imajte u vidu da metodi **obavi( )** generički tipovi nisu neophodni. Možemo jednostavno specificirati da ona prihvata objekat koji realizuje interfejs **Obavlja**:

```
//: generics/ProstiPsiIRoboti.java
// Uklanjamo generički kod; program i dalje radi.

class KomunicirajJednostavno {
  static void obavi(Obavlja izvodjac) {
    izvodjac.govoriti();
    izvodjac.sedeti();
```

```
  }
}

public class ProstiPsiIRoboti {
  public static void main(String[] args) {
    KomunicirajJednostavno.obavi(new CirkuskiPas());
    KomunicirajJednostavno.obavi(new Robot());
  }
} /* Ispis:
Vau!
Sedim
Klik!
Klank!
*///:~
```

U ovom slučaju, generički kôd nije bio neophodan, pošto su klase ionako morale da realizuju interfejs **Obavlja**.

Kompenzacija za nepostojanje latentnih tipova

Iako Java ne podržava latentne tipove, ispostavlja se da to ne znači da se generički kôd sa ograničenjima ne može upotrebljavati u raznim hijerarhijama tipova. Drugim rečima, ipak je moguće pisati pravi generički kôd, ali je to nešto teže.

Refleksija

Jedan pristup je upotreba refleksije. Evo metode **obavi**( ) koja upotrebljava latentne tipove:

```
//: generics/LatentnaRefleksija.java
// Pravljenje latentnih tipova pomoću refleksije.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// Ne realizuje interfejs Obavlja:
class Mimika {
  public void hodajNasuprotVetru() {}
  public void sedeti() { print("Pravim se da sedim"); }
  public void gurajNevidljiveZidove() {}
  public String toString() { return "Mimika"; }
}

// Ne realizuje interfejs Obavlja:
class PametanPas {
  public void govoriti() { print("Vau!"); }
  public void sedeti() { print("Sedim"); }
  public void reprodukovatise() {}
}
```

```
class KomunicirajRefleksivno
  public static void obavi(Object zvucnik) {
    Class<?> zvck = zvucnik.getClass();
    try {
      try {
        Method govoriti = zvck.getMethod("govoriti");
        govoriti.invoke(zvucnik);
      } catch(NoSuchMethodException e) {
        print(zvucnik + " ne može govoriti");
      }
      try {
        Method sedeti = zvck.getMethod("sedeti");
        sedeti.invoke(zvucnik);
      } catch(NoSuchMethodException e) {
        print(zvucnik + " ne može sedeti");
      }
    } catch(Exception e) {
      throw new RuntimeException(zvucnik.toString(), e);
    }
  }
}

public class LatentnaRefleksija {
  public static void main(String[] args) {
    KomunicirajRefleksivno.obavi(new PametanPas());
    KomunicirajRefleksivno.obavi(new Robot());
    KomunicirajRefleksivno.obavi(new Mimika());
  }
} /* Ispis:
Vau!
Sedim
Klik!
Klank!
Mimika ne može govoriti
Pravim se da sedim
*///:~
```

Ovde su klase sasvim različite i nemaju zajedničkih osnovnih klasa (sem klase **Object**) niti interfejsa. Pomoću refleksije, metoda **KomunicirajRefleksivno.obavi( )** može dinamički da utvrdi da li su željene metode dostupne i da ih pozove. Može da se izbori čak i sa činjenicom da **Mimika** ima samo jednu od potrebnih metoda i da svoj cilj ispunjava delimično.

Primena metode na sekvencu

Refleksija pruža zanimljive mogućnosti, ali svu proveru tipova odlaže za trenutak izvršavanja i zato je nepoželjna u mnogim situacijama. Provera tipova u vreme prevođenja obično je mnogo poželjnija. Ali, da li je moguće imati proveru tipova u vreme prevođenja *i* latentne tipove?

Pogledajmo primer koji istražuje taj problem. Pretpostavimo da hoćete da napravite metodu **primeni**( ) koja bilo koju metodu primenjuje na sve objekte u nekoj sekvenci. U ovoj situaciji interfejsi kao da ne odgovaraju. Hoćete da primenite bilo koju metodu na kolekciju objekata, a interfejsi nameću ograničenje koje ne dozvoljava da opišete „bilo koju metodu". Kako da to uradite u Javi?

Problem možemo najpre da rešimo refleksijom, i to ispada prilično elegantno zbog argumenata promenljive dužine koji se koriste od Jave SE5:

```
//: genericki/Primeni.java
// {main: TestZaPrimeni}
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Primeni {
  public static <T, S extends Iterable<? extends T>>
  void primeni(S sekv, Method f, Object... argumenti) {
    try {
      for(T t: sekv)
        f.invoke(t, argumenti);
    } catch(Exception e) {
      // Greške su programerski propusti
      throw new RuntimeException(e);
    }
  }
}

class Oblik {
  public void rotirati() { print(this + " rotirati"); }
  public void promvelicine(int novaVelicina) {
    print(this + " promvelicine " + novaVelicina);
  }
}

class Kvadrat extends Oblik {}

class PopunjenaLista<T> extends ArrayList<T> {
  public PopunjenaLista(Class<? extends T> tip, int velicina) {
    try {
      for(int i = 0; i < velicina; i++)
        // Pretpostavlja da postoji podrazumevani konstruktor:
        add(tip.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
class TestZaPrimeni {
  public static void main(String[] args) throws Exception {
    List<Oblik> oblici = new ArrayList<Oblik>();
    for(int i = 0; i < 10; i++)
      oblici.add(new Oblik());
    Primeni.primeni(oblici, Oblik.class.getMethod("rotirati"));
    Primeni.primeni(oblici,
      Oblik.class.getMethod("promvelicine", int.class), 5);
    List<Kvadrat> kvadrati = new ArrayList<Kvadrat>();
    for(int i = 0; i < 10; i++)
      kvadrati.add(new Kvadrat());
    Primeni.primeni(kvadrati, Oblik.class.getMethod("rotirati"));
    Primeni.primeni(kvadrati,
      Oblik.class.getMethod("promvelicine", int.class), 5);

    Primeni.primeni(new PopunjenaLista<Oblik>(Oblik.class, 10),
      Oblik.class.getMethod("rotirati"));
    Primeni.primeni(new PopunjenaLista<Oblik>(Kvadrat.class, 10),
      Oblik.class.getMethod("rotirati"));

    JednostavanRedZaCekanje<Oblik> oblikR
    = new JednostavanRedZaCekanje<Oblik>();
    for(int i = 0; i < 5; i++) {
      oblikR.add(new Oblik());
      oblikR.add(new Kvadrat());
    }
    Primeni.primeni(oblikR, Oblik.class.getMethod("rotirati"));
  }
} /* (Pokrenite da biste videli rezultate) *///:~
```

U programu **Primeni** imamo sreće, zato što je u Javu ugrađen interfejs **Iterable** koji se upotrebljava u Javinoj biblioteci kontejnera. Zato metoda **primeni( )** može da prihvati sve što realizuje interfejs **Iterable**, a to su sve potklase od **Collection** kao što je **List**. Ali ona može da prihvati i sve drugo u čemu ste vi realizovali **Iterable** – na primer, ovde definisanu klasu **JednostavanRedZaCekanje** koja se gore upotrebljava u metodi **main( )**:

```
//: generics/JednostavanRedZaCekanje.java
// Drugačija vrsta kontejnera koji je iterabilan
import java.util.*;

public class JednostavanRedZaCekanje<T> implements Iterable<T> {
  private LinkedList<T> skladiste = new LinkedList<T>();
  public void add(T t) { skladiste.offer(t); }
  public T get() { return skladiste.poll(); }
  public Iterator<T> iterator() {
    return skladiste.iterator();
  }
} ///:~
```

U programu **Primeni.java**, izuzeci su pretvoreni u izuzetke **RuntimeException** zato što nema pravog načina da se od njih oporavimo – u ovom slučaju oni zaista predstavljaju programerske propuste.

Vodite računa o tome da sam morao da stavim granice i džokerske argumente da bi klase **Primeni** i **PopunjenaLista** mogle da se koriste u svim željenim situacijama. Ako ih izvadite, videćete da **Primeni** i **PopunjenaLista** neće raditi u nekim primenama.

PopunjenaLista nas stavlja u pomalo nezgodan položaj. Da bi se u njoj neki tip mogao upotrebiti, on mora imati podrazumevani konstruktor (bez argumenata). Java to ne može da nametne u vreme prevođenja, pa se cela priča prebacuje na vreme izvršavanja. Obično se predlaže da se uspešna provera u vreme prevođenja postigne definisanjem proizvodnog interfejsa koji ima metodu za generisanje objekata; onda bi **PopunjenaLista** prihvatala taj interfejs, a ne „sirovog proizvođača" lekseme tipa (engl. *type token*). Ali onda bi sve klase upotrebljene u objektu tipa **PopunjenaLista** morale realizovati taj proizvodni interfejs. Nažalost, većina napisanih klasa ne zna za vaš interfejs, te ga i ne realizuje. Kasnije ću vam pokazati jedno rešenje ovog problema pomoću adaptera.

No, možda je prikazani pristup korišćenja lekseme tipa razborit kompromis (barem kao prvo rešenje). Uz takav pristup, korišćenje nečega kao što je objekat tipa **PopunjenaLista** dovoljno je lako da on biva korišćen, a ne zaobiđen. Naravno, greške se prijavljuju tek u vreme izvršavanja, pa se morate nadati da će one izroniti rano u procesu razvoja koda.

Znajte da se tehnika lekseme tipa preporučuje u literaturi o Javi, npr. u članku *Generički tipovi u programskom jeziku Java*,[9] gde autor, Gilad Bracha, kaže: „Taj idiom se mnogo koristi u novim interfejsima za programiranje aplikacija, recimo za obradu anotacija". Međutim, nisu baš svi srećni zbog toga; ima ljudi koji mnogo radije koriste proizvodni pristup, predstavljen u prethodnom delu poglavlja.

Sem toga, koliko god da je Javino rešenje ispalo elegantno, moramo primetiti da zbog upotrebe refleksije ono može biti sporije (iako je to u novijim verzijama Jave znatno poboljšano) od realizacije bez refleksije, pošto se u vreme izvršavanja dešava mnogo toga. To ne bi trebalo da vas spreči da koristite ovo rešenje, barem za početak (ukoliko ne padnete u iskušenje da prerano optimizujete kôd), ali svakako imajte na umu tu osnovnu razliku između navedena dva pristupa.

Vežba 40: (3) Svim ljubimcima u programu **podaciotipu.ljubimci** dodajte metodu **govoriti( )**. Preradite program **Primeni.java** tako da poziva metodu **govoriti( )** za raznorodne kolekcije objekata tipa **Ljubimac**.

Kada slučajno nemate odgovarajući interfejs

Imali smo sreće u gornjem primeru, zato što je interfejs **Iterable** već ugrađen, a radi tačno ono što nam treba. A šta da radimo u opštem slučaju, kada nema već ugrađenog interfejsa koji slučajno baš zadovoljava naše potrebe?

Primera radi, hajde da uopštimo ideju klase **PopunjenaLista** i napravimo parametrizovanu metodu **popuni( )** koja prima neku sekvencu i popunjava je pomoću nekog **Generatora**. Pokušajte da napišete to u Javi i naletećete na problem, jer nema podesnog

[9] Pogledajte poslednji odeljak u ovom poglavlju.

interfejsa **ImaAdd**, poput interfejsa **Iterable** nađenog u prethodnom primeru. Stoga umesto da kažete: „Sve za šta se može pozvati metoda **add( )**", morate reći „podtip od **Collection**". Rezultujući kôd nije naročito opšti, pošto mora biti ograničen na rad samo s realizacijama klase **Collection**. Ako pokušam da upotrebim klasu koja ne realizuje **Collection**, moj generički kôd neće raditi. Evo kako to izgleda:

```
//: generics/Popuni.java
// Uopštavanje ideje programa PopunjenaLista
// {main: PopuniTest}
import java.util.*;

// Ne radi sa "svime što ima metodu add()." Ne postoji
// interfejs "ImaAdd", pa moramo da upotrebimo
// neki kontejner (Collection). U ovom slučaju, ne možemo da
// uopštimo pomoću generičkog koda.

public class Popuni {
  public static <T> void popuni(Collection<T>kolekcija,
  Class<? extends T> classLeksema, int velicina) {
    for(int i = 0; i < velicina; i++)
      // Pretpostavlja da postoji podrazumevani konstruktor:
      try {
        kolekcija.add(classLeksema.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
  }      }
}

class Ugovor {
  private static long brojac = 0;
  private final long id = brojac++;
  public String toString() {
    return getClass().getName() + " " + id;
  }
}
class PrenosVlasnistva extends Ugovor {}

class PopuniTest {
  public static void main(String[] args) {
    List<Ugovor> ugovori = new ArrayList<Ugovor>();
    Popuni.popuni(ugovori, Ugovor.class, 3);
    Popuni.popuni(ugovori, PrenosVlasnistva.class, 2);
    for(Ugovor c: ugovori)
      System.out.println(c);
    JednostavanRedZaCekanje<Ugovor> redUgovora =
      new JednostavanRedZaCekanje<Ugovor>();
    // Neće raditi. popuni() nije dovoljno opšta:
    // Popuni.popuni(redUgovora, Ugovor.class, 3);
  }
```

```
} /* Ispis:
Ugovor 0
Ugovor 1
Ugovor 2
PrenosVlasnistva 3
PrenosVlasnistva 4
*///:~
```

Ovde bi mehanizam parametrizovanih tipova s latentnim tipovima dobro poslužio, pošto ne biste bili na milosti projektantskih odluka tvorca neke biblioteke i ne biste morali da prerađujete svoj kôd kad god naiđete na neku biblioteku koja nije predvidela vašu situaciju (pa bi vaš kôd zaista bio opšti). U gornjem slučaju, pošto projektanti Jave (razumljivo) nisu videli potrebu za interfejsom **ImaAdd**, ograničeni smo na hijerarhiju klase **Collection**, pa **JednostavanRedZaCekanje** ne radi mada ima metodu **add**( ). Pošto je ograničen na rad samo s kontejnerima klase **Collection**, prethodni program nije naročito opšti. To ne bi bilo tako da imamo latentne tipove.

Simuliranje latentnih tipova pomoću adaptera

Dakle, Javin generički kôd nema latentne tipove, a oni nam trebaju da bismo mogli da pišemo programe koji se mogu primeniti na najrazličitije klase (tj. opšte (generičke) programe). Možemo li nekako da zaobiđemo to ograničenje?

Šta bismo ovde postigli pomoću latentnih tipova? Mogli bismo da napišemo kôd koji kaže: „Svejedno mi je koji tip ovde koristim, samo ako ima ove metode". U stvari, latentni tipovi prave *implicitan interfejs* koji sadrži željene metode. Iz toga sledi da će problem biti rešen ako sami napišemo potreban interfejs (pošto Java to nije uradila umesto nas).

Pisanje koda koji od interfejsa koji imamo pravi željeni interfejs, primer je projektnog obrasca *Adapter* (Adapter). Adaptere možemo upotrebiti za prilagođavanje postojećih klasa tako da naprave željeni interfejs, uz relativno malu količinu koda. Rešenje u kojem smo upotrebili prethodno definisanu hijerarhiju klasa **Kafa**, pokazuje različite načine pisanja adaptera:

```
//: genericki/Popuni2.java
// Upotreba adaptera za simuliranje latentnih tipova.
// {main: Popuni2Test}
import genericki.kafa.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface ImaAdd<T> { void add(T t); }

public class Popuni2 {
  // Verzija s Class leksemom:
  public static <T> void popuni(ImaAdd<T> imajuciadd,
  Class<? extends T> classLeksema, int velicina) {
    for(int i = 0; i < velicina; i++)
      try {
```

```
        imaadd.add(classLeksema.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
  // Generatorska verzija:
  public static <T> void popuni(ImaAdd<T> imaadd,
  Generator<T> generator, int velicina) {
    for(int i = 0; i < velicina; i++)
      imaadd.add(generator.sledeci());
  }
}

// Za prilagođavanje osnovnog tipa morate upotrebiti kompoziciju.
// Pomoću kompozicije, napravićemo da svi podtipovi
// klase Collection imaju metodu add():
class ImaAddCollectionAdapter<T> implements ImaAdd<T> {
  private Collection<T> c;
  public ImaAddCollectionAdapter(Collection<T> c) {
    this.c = c;
  }
  public void add(T stavka) { c.add(stavka); }
}

// Pomagač za automatsko hvatanje tipa:
class Adapter {
  public static <T>
  ImaAdd<T> adapterKolekcije(Collection<T> c) {
    return new ImaAddCollectionAdapter<T>(c);
  }
}

// Za prilagođavanje određenog tipa možete upotrebiti nasleđivanje.
// Pomoću nasleđivanja, napravićemo da
// JednostavanRedZaCekanje ima metodu add():
class ImaAddJednostavanRedZaCekanje<T>
extends JednostavanRedZaCekanje<T> implements ImaAdd<T> {
  public void add(T stavka) { super.add(stavka); }
}

class Popuni2Test {
  public static void main(String[] args) {
    // Prilagodi kontejner (objekat tipa Collection):
    List<Kafa> nosilac = new ArrayList<Kafa>();
    Popuni2.popuni(
      new ImaAddCollectionAdapter<Kafa>(nosilac),
      Kafa.class, 3);
    // Pomagačka metoda tip hvata:
    Popuni2.popuni(Adapter.adapterKolekcije(nosilac),
      SMlekom.class, 2);
```

```
    for(Kafa c: nosilac)
      print(c);
    print("----------------------");
    // Upotreba prilagođene klase:
    ImaAddJednostavanRedZaCekanje<Kafa> redKafa =
      new ImaAddJednostavanRedZaCekanje<Kafa>();
    Popuni2.popuni(redKafa, Moka.class, 4);
    Popuni2.popuni(redKafa, SMlekom.class, 1);
    for(Kafa c: redKafa)
      print(c);
  }
} /* Ispis:
Kafa 0
Kafa 1
Kafa 2
SMlekom 3
SMlekom 4
----------------------
Moka 5
Moka 6
Moka 7
Moka 8
SMlekom 9
*///:~
```

Popuni2 ne zahteva kontejner tipa **Collection**, kao što je zahtevao program **Popuni**. Umesto toga, on zahteva nešto što realizuje interfejs **ImaAdd**, a **ImaAdd** je napisan upravo za program **Popuni** – on je pojava latentnog tipa koju sam hteo da prevodilac napravi umesto mene.

U ovoj verziji, dodao sam i preklopljenu metodu **popuni( )** koja umesto lekseme tipa prima **Generator**. Tip **Generator**a se proverava u vreme prevođenja: prevodilac vas neće pustiti da mu prosledite neispravan tip **Generator**a, pa u vreme izvršavanja neće biti izuzetaka.

Prvi adapter, **ImaAddCollectionAdapter**, radi sa osnovnim tipom **Collection**, što znači da se može upotrebiti svaka realizacija klase **Collection**. Ova verzija jednostavno skladišti referencu objekta tipa **Collection** i koristi je za realizaciju metode **add( )**.

Ako imate određeni tip, a ne osnovnu klasu hijerarhije, pravljenje adaptera nasleđivanjem zahtevaće nešto manje koda, kao što možete videti u **ImaAddJednostavanRedZaCekanje**.

U metodi **Popuni2Test.main( )**, možete videti razne vrste adaptera na delu. Prvo, neki podtip klase **Collection** biva prilagođen pomoću klase **ImaAddCollectionAdapter**. Druga verzija adaptera koristi generičku pomagačku metodu, i možete videti kako ta generička metoda hvata tip, tako da on ne mora biti izričito napisan – to je podesan trik koji daje elegantniji kôd.

Zatim se upotrebljava prethodno prilagođena klasa **ImaAddJednostavanRedZaCekanje**. Vodite računa o tome da u oba slučaja adapteri omogućuju da se klase, koje prethodno nisu realizovale interfejs **ImaAdd** mogu popunjavati metodom **Popuni2.popuni( )**.

Izgleda kao da ovakvo korišćenje adaptera kompenzuje nepostojanje latentnih tipova i time omogućuje pisanje zaista opšteg koda. Međutim, to je dodatni korak koji moraju razumeti i tvorac i korisnik biblioteke, a manje iskusni programeri možda neće tako brzo shvatiti njegov koncept. Latentni tipovi uklanjaju taj dodatni korak, čime olakšavaju korišćenje generičkog koda, i u tome je njihova vrednost.

Vežba 41: (1) Prepravite program **Popuni2.java** tako da umesto klasa **Kafa** upotrebljava klase iz **podaciotipu.ljubimci**.

Upotreba funkcijskih objekata kao strategija

U ovom završnom primeru napravićemo pravi generički kôd pomoću adaptera, kao što je opisano u prethodnom odeljku. Najpre je u primeru trebalo da se napravi zbir sekvence elemenata (bilo kojeg tipa koji se može sabirati), ali se razvio do izvođenja opštih operacija korišćenjem *funkcionalnog* stila programiranja.

Ako samo pogledate proces pokušavanja da se saberu objekti, videćete da u tom slučaju imamo zajedničke operacije raznih klasa, ali te operacije nisu predstavljene u nekoj osnovnoj klasi koju bismo mogli da specificiramo – ponekad čak možete da upotrebite operator +, a u drugim slučajevima može postojati neka vrsta metode add. To je uobičajena situacija koja se sreće kada pokušavate da pišete generički kôd, zato što hoćete da kôd bude primenljiv na različite klase – naročito, kao u ovom slučaju, na više postojećih klasa koje ne možete da prepravite. Čak i kada biste suzili ovaj izbor na potklase od **Number**, ta natklasa ništa ne kaže o nekoj sabirljivosti.

Rešenje je primeniti projektni obrazac *Strategy* (Strategija) koji proizvodi elegantniji kôd tako što „ono što se menja" potpuno izoluje unutar određenog *funkcijskog objekta.*[10] Funkcijski objekat je onaj koji se na neki način ponaša kao funkcija – obično kada je u pitanju jedna metoda (u jezicima koji podržavaju preklapanje operatora, možete napraviti da poziv te metode *izgleda* kao običan poziv metode). Vrednost funkcijskih objekata je to što, za razliku od običnih metoda, njih možete prosleđivati, a one takođe mogu imati i stanje koje se ne gubi između poziva. Naravno, nešto slično možete postići s bilo kojom metodom određene klase, ali (kao što je slučaj sa svim projektnim obrascima) funkcijski objekat se prvenstveno odlikuje svojom svrhom. Ovde je svrha napraviti nešto što se ponaša kao metoda koja se može prosleđivati; stoga je funkcijski objekat tesno povezan sa projektnim obrascem *Strategy* (a ponekad je isto što i on).

Kao što mi se desilo s više projektnih obrazaca, ovde mi reči postaju nekako nejasne: pravimo funkcijske objekte koji obavljaju prilagođavanje, a njih prosleđujemo metodama koje ih upotrebljavaju kao strategije.

Sledeći taj pristup, dodao sam razne vrste generičkih metoda koje sam prvobitno nameravao da napravim, i još neke. Rezultat je ovo:

```
//: generics/Funkcional.java
import java.math.*;
import java.util.concurrent.atomic.*;
```

[10] Za njih se upotrebljava i ime *funktori*. Ja ću koristiti termin *funkcijski objekat*, pošto „funktor" ima specifično i drugo značenje u matematici.

```
import java.util.*;
import static net.mindview.util.Print.*;

// Razni tipovi funkcijskih objekata:
interface Kombinator<T> { T objedini(T x, T y); }
interface UnarnaFunkcija<R,T> { R funkcija(T x); }
interface Kolektor<T> extends UnarnaFunkcija<T,T> {
  T rezultat(); // Izdvoji rezultat parametra kolekcije
}
interface UnarniPredikat<T> { boolean test(T x); }

public class Funkcional {
  // Poziva objekat Kombinator za svaki element da bi ga objedinio
  // s tekućim rezultatom, koji zatim vraća:
  public static <T> T
  redukuj(Iterable<T> sekv, Kombinator<T> kombinator) {
    Iterator<T> it = sekv.iterator();
    if(it.hasNext()) {
      T rezultat = it.sledeci();
      while(it.hasNext())
        rezultat = kombinator.objedini(rezultat, it.sledeci());
      return rezultat;
    }
    // Ako je sekv prazna lista:
    return null; // Ili generiši izuzetak
  }
  // Uzmi funkcijski objekat i pozovi ga za svaki objekat u
  // listi; povratnu vrednost zanemari. Funkcijski objekat
  // može delovati kao parametar kolekcije, pa se
  // na kraju on vraća kao rezultat.
  public static <T> Kolektor<T>
  forEach(Iterable<T> sekv, Kolektor<T> funk) {
    for(T t : sekv)
      funk.funkcija(t);
    return funk;
  }
  // Pravi listu rezultata pozivanjem funkcijskog
  // objekta za svaki objekat u listi:
  public static <R,T> List<R>
  transformisi(Iterable<T> sekv, UnarnaFunkcija<R,T> funk) {
    List<R> rezultat = new ArrayList<R>();
    for(T t : sekv)
      rezultat.add(funk.funkcija(t));
    return rezultat;
  }
  // Primenjuje unarni predikat na svaku stavku sekvence,
  // i vraća listu stavki koje su dale "true":
  public static <T> List<T>
  filtar(Iterable<T> sekv, UnarniPredikat<T> pred) {
    List<T> rezultat = new ArrayList<T>();
```

```
    for(T t : sekv)
      if(pred.test(t))
        rezultat.add(t);
    return rezultat;
  }
  // Da bismo mogli da koristimo gornje generičke metode,
  // moramo napraviti funkcijske objekte za prilagođavanje metoda
  // našim specifičnim potrebama:
  static class SabiracCelih implements Kombinator<Integer> {
    public Integer objedini(Integer x, Integer y) {
      return x + y;
    }
  }
  static class
  OduzimacCelih implements Kombinator<Integer> {
    public Integer objedini(Integer x, Integer y) {
      return x - y;
    }
  }
  static class
  SabiracVelikihDecimalnih implements Kombinator<BigDecimal> {
    public BigDecimal objedini(BigDecimal x, BigDecimal y) {
      return x.add(y);
    }
  }
  static class
  SabiracVelikihCelih implements Kombinator<BigInteger> {
    public BigInteger objedini(BigInteger x, BigInteger y) {
      return x.add(y);
    }
  }
  static class
  SabiracAtomicLong implements Kombinator<AtomicLong> {
    public AtomicLong objedini(AtomicLong x, AtomicLong y) {
      // Nisam siguran da ovo ima smisla:
      return new AtomicLong(x.addAndGet(y.get()));
    }
  }
  // Možemo napraviti čak i UnarnuFunkciju s metodom "ulp"
  // (Units in the last place, jedinica na poslednjem mestu):
  static class VelikiDecimalniUlp
  implements UnarnaFunkcija<BigDecimal,BigDecimal> {
    public BigDecimal funkcija(BigDecimal x) {
      return x.ulp();
    }
  }
  static class VeceOd<T extends Comparable<T>>
  implements UnarniPredikat<T> {
    private T granica;
    public VeceOd(T granica) { this.granica = granica; }
```

```
    public boolean test(T x) {
      return x.compareTo(granica) > 0;
    }
  }
  static class KolektorMnoziCele
  implements Kolektor<Integer> {
    private Integer vre = 1;
    public Integer funkcija(Integer x) {
      vre *= x;
      return vre;
    }
    public Integer rezultat() { return vre; }
  }
  public static void main(String[] args) {
    // Generički tipovi, argumenti promenljive dužine
    // i pakovanje u zajedničkom radu:
    List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
    Integer rezultat = redukuj(li, new SabiracCelih());
    print(rezultat);

    rezultat = redukuj(li, new OduzimacCelih());
    print(rezultat);

    print(filtar(li, new VeceOd<Integer>(4)));

    print(forEach(li,
      new KolektorMnoziCele()).rezultat());

    print(forEach(filtar(li, new VeceOd<Integer>(4)),
      new KolektorMnoziCele()).rezultat());

    MathContext mk = new MathContext(7);
    List<BigDecimal> lvd = Arrays.asList(
      new BigDecimal(1.1, mk), new BigDecimal(2.2, mk),
      new BigDecimal(3.3, mk), new BigDecimal(4.4, mk));
    BigDecimal rvd = redukuj(lvd, new SabiracVelikihDecimalnih());
    print(rvd);

    print(filtar(lvd,
      new VeceOd<BigDecimal>(new BigDecimal(3))));

    // Koristi ugrađeno generisanje prostih faktora tipa BigInteger:
    List<BigInteger> lbi = new ArrayList<BigInteger>();
    BigInteger bi = BigInteger.valueOf(11);
    for(int i = 0; i < 11; i++) {
      lbi.add(bi);
      bi = bi.nextProbablePrime();
    }
    print(lbi);
```

```
    BigInteger rbi = redukuj(lbi, new SabiracVelikihCelih());
    print(rbi);
    // Zbir stavki sa ove liste prostih faktora i sam je prost broj:
    print(rbi.isProbablePrime(5));

    List<AtomicLong> lal = Arrays.asList(
      new AtomicLong(11), new AtomicLong(47),
      new AtomicLong(74), new AtomicLong(133));
    AtomicLong ral = redukuj(lal, new SabiracAtomicLong());
    print(ral);

    print(transformisi(lvd,new VelikiDecimalniUlp()));
  }
} /* Ispis:
28
-26
[5, 6, 7]
5040
210
11.000000
[3.300000, 4.400000]
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
311
true
265
[0.000001, 0.000001, 0.000001, 0.000001]
*///:~
```

Počinjem definisanjem interfejsa za različite tipove funkcijskih objekata. Njih sam pravio po potrebi, kako sam razvijao različite metode i uviđao potrebu za svakim. Klasu **Kombinator** je predložio anonimni komentator jednog od članaka objavljenih na mojoj Web lokaciji. **Kombinator** apstrahuje konkretne detalje sabiranja dva objekta i kazuje samo da oni bivaju nekako objedinjeni. Kao rezultat, možete videti da **SabiracCelih** i **OduzimacCelih** mogu biti tipovi nadtipa **Kombinator**.

UnarnaFunkcija prima samo jedan argument i daje rezultat; argument i rezultat ne moraju biti istog tipa. **Kolektor** se upotrebljava kao „parametar kolekcije", a rezultat možete da izdvojite kada završite. **UnarniPredikat** daje rezultat tipa **boolean**. Mogu se definisati funkcijski objekti i drugih tipova, ali ovo je bilo dovoljno za pouku.

Klasa **Funkcional** sadrži više generičkih metoda koje funkcijske objekte primenjuju na sekvence. **redukuj**( ) primenjuje funkciju u objektu tipa **Kombinator** na svaki element sekvence da bi proizvela samo jedan rezultat.

forEach( ) prima **Kolektor** i primenjuje svoju funkciju na svaki element, zanemarujući rezultat svakog poziva funkcije. Nju možete pozivati samo zbog sporednog efekta (to ne bi bio funkcionalni stil programiranja, ali ipak može da posluži), ili **Kolektor** može da održava interno stanje da bi postao parametar kolekcije, kao što je slučaj u ovom primeru.

transformisi( ) pravi listu pozivanjem funkcijskog objekta **UnarnaFunkcija** za svaki objekat u sekvenci i hvatanjem rezultata.

Najzad, **filtar( )** primenjuje funkcijski objekat **UnarniPredikat** na svaki objekat u sekvenci i one koji vraćaju **true** smešta u **List**u koju vraća.

Možete definisati i sopstvene generičke funkcije. Primera radi, C++-ova standardna biblioteka šablona STL ima ih na gomile. Ovaj problem je bio rešen i u nekim bibliotekama otvorenog izvornog koda; jedna od njih je JGA (Generic Algorithms for Java, Generički algoritmi za Javu).

C++-ovi latentni tipovi se staraju za pronalaženje odgovarajućih operacija prilikom poziva funkcija, ali u Javi moramo da pišemo funkcijske objekte za prilagođavanje generičkih metoda našim potrebama. Zato sledeći deo klase sadrži različite realizacije funkcijskih objekata. Obratite pažnju na to, recimo, da **SabiracCelih** i **SabiracVelikihDecimalnih** rešavaju isti problem – sabiranje dva objekta – pozivanjem odgovarajućih operacija za svoj tip. Dakle, to je kombinacija obrazaca *Adapter* i *Strategy*.

U metodi **main( )**, vidite da se u svakom pozivu metode uz odgovarajući funkcijski objekat prosleđuje sekvenca. Takođe, više izraza je prilično složeno, kao:

```
forEach(filtar(li, new VeceOd(4)),
        new KolektorMnoziCele()).rezultat()
```

Prethodni izraz pravi listu biranjem svih elemenata u **li** većih od 4, primenjuje na nju **KolektorMnoziCele( )** i izdvaja **rezultat( )**. Detalje ostatka programa neću da objašnjavam – verovatno ćete ih i sami shvatiti kada ih proučite.

Vežba 42: (5) Napravite dve zasebne klase koje nemaju ništa zajedničko. Svaka klasa treba da sadrži neku vrednost i da ima barem one metode koje proizvode tu vrednost i menjaju je. Izmenite **Funkcional.java** tako da obavlja funkcionalne operacije na kolekcijama vaših klasa (te operacije ne moraju biti aritmetičke kao što su u programu **Funkcional.java**).

Sažetak: da li je eksplicitna konverzija tipova zaista tako loša?

Pošto C++ šablone objašnjavam od njihovog nastanka, sledeće razmatranje sam navodio verovatno češće od ikoga. Tek nedavno sam prestao da se pitam koliko često je takvo razmatranje umesno – koliko puta se problem koji ću opisati zaista prikrade i pomoli?

Razmatranje ide ovako. Kontejnerske klase **List**, **Set**, **Map** itd. spadaju među najprikladnija mesta za upotrebu mehanizma generičkih tipova. Upoznali ste ih u poglavlju *Čuvanje objekata* i još ćete čitati o njima u poglavlju *Detaljno razmatranje kontejnera*. Pre Jave SE5, tip objekta stavljenog u kontejner bio je sveden naviše na **Object**, pa se gubila informacija o pravom tipu. Kada je trebalo nešto uraditi sa objektom i izvaditi ga iz kontejnera, morao se eksplicitno konvertovati nazad u pravi tip. Moj primer bila je lista objekata tipa **Macka** (na početku poglavlja *Čuvanje objekata* data je varijanta s jabukama i narandžama). Dok nije bilo generičke verzije kontejnera koju je donela Java SE5, unutra smo stavljali **Objecte**, iz kontejnera smo vadili **Objecte**, pa je bilo veoma lako staviti objekat tipa **Pas** u listu objekata tipa **Macka**.

Međutim, predgenerička Java ne bi dopustila da *zloupotrebite* objekte stavljene u kontejner. Da ste stavili objekat tipa **Pas** u kontejner **Macka** i pokušali da sve u kontejneru tretirate kao da je tipa **Macka**, dobili biste **RuntimeException** – da ste izvadili referencu objekta tipa **Pas** iz kontejnera **Macka** i pokušali da ga ekplicitno konvertujete u tip **Macka**, dobili biste izuzetak. Problem nije ostajao sakriven, ali ste ga otkrivali u vreme izvršavanja, a ne u vreme prevođenja.

U prethodnim izdanjima ove knjige, dalje sam pisao:

> *Ovo je više od neprijatnosti. Zbog toga mogu nastati greške koje je teško otkriti. Ako deo (ili više delova) programa umeće objekte u kontejner, a vi tek preko izuzetka u nekom zasebnom delu programa otkrijete da je u kontejner stavljen loš objekat, onda morate da tražite gde je loš objekat umetnut.*

Nakon daljnjeg razmišljanja o ovome, počeo sam da sumnjam. Prvo, koliko često se to događa? Ne sećam se da mi se ikada desilo nešto takvo, a i na konferencijama nisam čuo da se to nekome dogodilo. U jednoj knjizi navodio se primer liste nazvane **datoteke** koja sadrži **String** objekte – u tom primeru izgledalo je sasvim prirodno dodati objekat tipa **Datoteka** u **datoteke**, pa je objekat trebalo nazvati recimo **imenaDatoteka**. Koliko god da Java proverava tipove, i dalje je moguće pisati nejasne programe, a loše napisan program koji se ipak prevede i dalje ostaje loš. Možda većina programera kontejnerima daje prikladna imena, recimo **macke** kao vizuelno upozorenje da se unutra ne dodaju objekti koji nisu tipa **Macka**. A da se to i desi, koliko dugo bi ostalo neotkriveno? Čini mi se da bi se pojavio izuzetak čim bi počelo testiranje s pravim podacima.

Jedan autor je čak tvrdio da bi takva greška mogla ostati „sakrivena godinama". Ali nešto ne pamtim poplavu izveštaja o ljudima koji su veoma teško pronalazili greške tipa „pas u listi mačaka", pa čak se ne sećam ni da su ih često pravili. S druge strane, u poglavlju *Paralelno izvršavanje* videćete da se s nitima veoma lako i često prave greške koje se pojavljuju izuzetno retko, a daju tek neodređenu predstavu o tome šta ne valja. Dakle, da li je greška „pas u listi mačaka" pravi razlog što je ova veoma značajna i prilično složena mogućnost dodata Javi?

Smatram da je *svrha* jezičke mogućnosti opšte namene nazvane „generički tipovi" (ne nužno i njene konkretne realizacije u Javi) *izražajnost*, a ne samo pravljenje kontejnera čiji se tipovi proveravaju. Kontejneri bezbednih tipova su sporedan efekat sposobnosti pravljenja koda opštije namene.

Zato, iako se greška tipa „pas u listi mačaka" često navodi kao opravdanje za uvođenje generičkih tipova, pitanje je da li to stoji. I kao što sam tvrdio na početku ovog poglavlja, ne verujem da je to zapravo *svrha* generičkih tipova. Kao što im i ime govori, generički tipovi omogućuju pisanje opštijeg koda koji manje ograničava tipove s kojima može da radi, pa se isto parče koda može primeniti na više tipova. U ovom poglavlju videli ste da je prilično lako napisati zaista opšte klase „skladište" (a Javini kontejneri to jesu), ali napisati generički kôd koji obrađuje svoje generičke tipove zahteva dodatni trud tvorca klase i njenog korisnika, koji mora da shvati koncept i realizaciju projektnog obrasca *Adapter*. Taj dodatni trud otežava korišćenje generičkih tipova i čini ih manje primenljivim u slučajevima gde bi inače dobro poslužili.

Takođe, imajte u vidu sledeće: zbog toga što su generički tipovi bili naknadno ubačeni u Javu, umesto da su od početka projektovani kao njen sastavni deo, neki kontejneri nisu toliko robusni koliko bi trebalo da budu. Na primer, pogledajte **Map**, konkretno metode **containsKey(Object ključ)** i **get(Object ključ)**. Da su te klase projektovane s postojećim generičkim tipovima, te metode bi upotrebljavale parametrizovane tipove, a ne **Object**, i tako obezbedile proveru tipova u vreme prevođenja koju bi generički tipovi trebalo da obezbeđuju. Primera radi, u C++ **map**ama tip ključa se uvek proverava u vreme prevođenja.

Jedno je sasvim jasno: uvođenje bilo koje vrste generičkog mehanizma u kasnoj verziji jezika, nakon što je on ušao u opštu upotrebu, predstavlja veoma, veoma zapetljan poduhvat koji se ne može ostvariti bez žrtava. U C++ su šabloni uvedeni u početnoj ISO verziji jezika (iako je i to prouzrokovalo poteškoće, pošto se pre pojave prvog standardnog C++-a koristila prethodna verzija bez šablona), pa su šabloni u stvari *oduvek* bili deo tog jezika. U Javu su generički tipovi uvedeni tek 10 godina nakon njenog izlaska u svet, pa su problemi s prelaskom na generičke tipove bili veoma veliki i znatno su uticali na njihov dizajn. Posledica toga je da vi, programer, trpite zato što projektanti Jave nisu imali viziju kada su pisali verziju 1.0. Kada je Java pravljena, njeni projektanti su naravno znali za C++-ove šablone, i čak su razmatrali da li da ih uključe u jezik, ali su iz nekog razloga (izgleda da su žurili) odlučili da ih izostave. Zato trpe i jezik i programeri koji ga koriste. Samo vreme će pokazati sve posledice koje će Javina realizacija generičkih tipova imati na taj jezik.

Neki jezici, među kojima su *Nice* (videti *http://nice.sourceforge.net*; ovaj jezik generiše Javin bajtkod i radi s postojećim Javinim bibliotekama) i NextGen (videti *http://japan.cs.rice.edu/nextgen*) imaju čistiji i lakši pristup parametrizovanim tipovima. Nije nemoguće da neki takav jezik postane Javin naslednik, pošto oni rade tačno ono što su autori C++-a uradili s C-om: uzeli postojeće i to poboljšali.

Pročitajte i ovo

Uvodni dokument za generičke tipove je *Generics in the Java Programming Language*, čiji je autor Gilad Bracha, a nalazi se na lokaciji *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*.

Java Generics FAQs Angelike Langer veoma je koristan resurs, na lokaciji *www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html*.

Više o džokerskim argumentima saznaćete iz teksta *Adding Wildcards to the Java Programming Language*, čiji su autori Torgerson, Ernst, Hansen, Von der Ahe, Bracha i Gafter. Članak je dostupan na lokaciji *www.jot.fm/issues/issue_2004_12/article5*.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide* koji se može kupiti na lokaciji *www.MindView.com*.

Nizovi

Na kraju poglavlja Inicijalizacija i čišćenje *objašnjeno je kako se definiše i inicijalizuje niz.*

POJEDNOSTAVLJENO VIĐENJE NIZA JESTE: NAPRAVITE GA I POPUNITE, BIRATE ELEMENTE IZ njega pomoću celobrojnog indeksa, i on ne menja svoju veličinu. To je uglavnom sve što morate da znate, ali ponekad s nizovima morate da obavite sofisticiranije operacije, a katkada i da procenite da li je bolje upotrebiti niz ili neki fleksibilniji kontejner. U ovom poglavlju bolje ćete upoznati nizove.

Šta to nizove čini posebnim

Ima više drugih načina za čuvanje objekata, pa šta to niz čini posebnim?

Nizovi se od drugih vrsta kontejnera razlikuju po trima osobinama: efikasnosti, tipu i sposobnosti čuvanja prostih tipova. Niz je Javin najefikasniji način za čuvanje i nasumičan pristup grupi referenci na objekte. Niz je jednostavna linearna sekvenca čijim elementima se pristupa brzo, ali se ta brzina plaća: kada napravite niz, njegova veličina je tačno određena i ne može se menjati tokom celog njegovog životnog veka. Kao rešenje ovog problema može vam pasti na pamet klasa **ArrayList** (koju ste upoznali u poglavlju *Čuvanje objekata*): ako ponestane prostora u nizu, ona automatski pravi nov niz i premešta sve reference iz starog niza u novi. Iako bi klasi **ArrayList** po pravilu trebalo da dajete prednost nad nizom, ona je zbog svoje prilagodljive veličine osetno manje efikasna od običnog niza.

I nizovi i kontejneri jemče da ih ne možete zloupotrebiti. Bez obzira na to da li koristite niz ili kontejner, pojaviće se izuzetak tipa **RuntimeException** ako im prekoračite granice, što ukazuje na grešku programera.

Pre generičkih tipova, druge kontejnerske klase su radile sa objektima kao da nisu određenog tipa, tj. odnosile su se prema njima kao da su tipa **Object** (**Object** je korenska klasa svih Javinih klasa.) Zato je niz superiorniji od *pred*generičkih kontejnera: kada pravite niz, znate da će on sadržati određeni tip. To znači da će se proveravanje tipa tokom prevođenja onemogućiti stavljanje pogrešnog tipa u niz, ili čitanje pogrešnog tipa iz niza. Naravno, Java će sprečiti i slanje neodgovarajuće poruke objektu, i to u vreme prevođenja ili izvršavanja. Dakle, nijedan od dva načina nije rizičniji. Lepše je da vam prevodilac upozori na grešku u programu, pa da krajnji korisnik ne bude iznenađen nekog izuzetka.

Niz može da čuva proste tipove, dok predgenerički kontejneri to nisu mogli. Međutim, generički kontejneri *mogu* da zadaju i proveravaju tip objekata koje sadrže, a sa automatskim pakovanjem deluju kao da mogu da primaju i proste tipove, pošto je konverzija automatska. Evo primera u kojem se porede nizovi i generički kontejneri:

```
//: nizovi/PoredjenjeSKontejnerima.java
import java.util.*;
import static net.mindview.util.Print.*;
```

```
class SferaOdBerilijuma {
  private static long brojac;
  private final long id = brojac++;
  public String toString() { return "Sfera " + id; }
}

public class PoredjenjeSKontejnerima {
  public static void main(String[] args) {
    SferaOdBerilijuma[] sfere = new SferaOdBerilijuma[10];
    for(int i = 0; i < 5; i++)
      sfere[i] = new SferaOdBerilijuma();
    print(Arrays.toString(sfere));
    print(sfere[4]);

    List<SferaOdBerilijuma> listaSfera =
      new ArrayList<SferaOdBerilijuma>();
    for(int i = 0; i < 5; i++)
      listaSfera.add(new SferaOdBerilijuma());
    print(listaSfera);
    print(listaSfera.get(4));

    int[] celiBrojevi = { 0, 1, 2, 3, 4, 5 };
    print(Arrays.toString(celiBrojevi));
    print(celiBrojevi[4]);

    List<Integer> listaCelihBrojeva = new ArrayList<Integer>(
      Arrays.asList(0, 1, 2, 3, 4, 5));
    listaCelihBrojeva.add(97);
    print(listaCelihBrojeva);
    print(listaCelihBrojeva.get(4));
  }
} /* Ispis:
[Sfera 0, Sfera 1, Sfera 2, Sfera 3, Sfera 4, null, null, null, null,
null]
Sfera 4
[Sfera 5, Sfera 6, Sfera 7, Sfera 8, Sfera 9]
Sfera 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*///:~
```

U Javi se granice proveravaju bez obzira na to da li koristite niz ili kontejner; prividno je jedina razlika među njima to što nizovi imaju operator [ ] za pristupanje elementima, a liste imaju metode kao što su **add**() i **get**(). Sličnost između nizova i klase **ArrayList** namerna je, da bi bilo lako koristiti i jedne i druge. Ali kao što ste videli u poglavlju *Čuvanje objekata*, kontejneri su mnogo funkcionalniji od nizova.

Od uvođenja automatskog pakovanja, proste tipove je gotovo jednako lako čuvati u kontejnerima kao u nizovima. Jedina preostala prednost nizova je efikasnost. Medutim, kada rešavate opštiji problem, nizovi mogu da nametnu prestroga ograničenja, i u tim slučajevima upotrebljavajte kontejnere.

Nizovi su prvorazredni objekti

Bez obzira na to s kojom vrstom niza radite, identifikator niza je referenca na stvarni objekat koji je napravljen u dinamičkoj memoriji. To je objekat koji sadrži reference na druge objekte, a može biti napravljen bilo implicitno, kao deo sintakse za inicijalizaciju niza, bilo eksplicitno, operatorom **new**. Deo objekta – niza (zapravo, jedino polje kome možete da pristupite) jeste član **length** koji se može samo čitati, a označava broj elemenata niza. Pored toga, sintaksna oznaka [] jedini je način za pristup objektu – nizu.

Sledeći primer prikazuje različite načine za inicijalizaciju niza i dodeljivanje referenci objektima u nizu. Primer pokazuje i da se nizovi objekata i nizovi prostih tipova koriste gotovo istovetno. Jedina razlika je to što nizovi objekata sadrže reference, dok nizovi prostih tipova sadrže vrednosti prostih tipova.

```
//: nizovi/MogucnostiNiza.java
// Inicijalizacija i ponovna dodela vrednosti nizu.
import java.util.*;
import static net.mindview.util.Print.*;

public class MogucnostiNiza {
  public static void main(String[] args) {
    // Nizovi objekata:
    SferaOdBerilijuma[] a; // Neinicijalizovana lokalna promenljiva
    SferaOdBerilijuma[] b = new SferaOdBerilijuma[5];
    // Reference unutar niza se
    // automatski inicijalizuju na null:
    print("b: " + Arrays.toString(b));
    SferaOdBerilijuma[] c = new SferaOdBerilijuma[4];
    for(int i = 0; i < c.length; i++)
      if(c[i] == null) // Moguće testiranje na null referencu
        c[i] = new SferaOdBerilijuma();
// Agregatna inicijalizacija:
    SferaOdBerilijuma[] d = { new SferaOdBerilijuma(),
      new SferaOdBerilijuma(), new SferaOdBerilijuma()
    };
// Dinamička agregatna inicijalizacija:
    a = new SferaOdBerilijuma[]{
      new SferaOdBerilijuma(), new SferaOdBerilijuma(),
    };
    // (Zarez na poslednjem mestu je neobavezan u oba slučaja)
    print("a.length = " + a.length);
    print("b.length = " + b.length);
```

```
    print("c.length = " + c.length);
    print("d.length = " + d.length);
    a = d;
    print("a.length = " + a.length);

    // Nizovi prostih tipova:
    int[] e; // Null referenca
    int[] f = new int[5];
    // Prosti tipovi unutar niza se
    // automatski inicijalizuju na nulu:
    print("f: " + Arrays.toString(f));
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Greška u prevođenju: promenljiva e nije inicijalizovana:
    //!print("e.length = " + e.length);
    print("f.length = " + f.length);
    print("g.length = " + g.length);
    print("h.length = " + h.length);
    e = h;
    print("e.length = " + e.length);
    e = new int[]{ 1, 2 };
    print("e.length = " + e.length);
  }
} /* Ispis:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*///:~
```

Niz **a** je neinicijalizovana lokalna promenljiva, a prevodilac sprečava da se s tom referencom uradi bilo šta sve dok se ona pravilno ne inicijalizuje. Niz **b** je inicijalizovan tako da ukazuje na niz referenci tipa **SferaOdBerilijuma**, ali se u taj niz nikada ne smeštaju objekti klase **SferaOdBerilijuma**. Međutim, uvek postoji mogućnost da pročitate dužinu niza, pošto **b** ukazuje na postojeći objekat. Ovde se pojavljuje mali nedostatak: ne možete da saznate koliko se stvarno elemenata nalazi *u* nizu, pošto polje **length** saopštava samo koliko elemenata *može da stane* u niz, tj. kolika je dužina niza, a ne koliko elemenata sadrži. Međutim, kada se napravi niz, njegove reference se automatski inicijalizuju vrednošću **null**,

pa možete da utvrdite da li određeni element niza sadrži objekat tako što proverite da li je njegova vrednost **null**. Slično tome, niz prostih tipova se automatski inicijalizuje vrednošću nula za numeričke tipove, **(char)0** za tip **char**, odnosno **false** za tip **boolean**.

Niz **c** pokazuje kako se pravi objekat niza, nakon čega sledi dodela objekata klase **SferaOdBerilijuma** svim njegovim elementima. Niz **d** pokazuje sintaksu agregatne inicijalizacije koja omogućuje pravljenje objekta niza (implicitno, pomoću operatora **new**, kao za niz **c**) *i* njegovo inicijalizovanje objektima klase **SferaOdBerilijuma**, sve u jednoj naredbi.

Inicijalizacija sledećeg niza se može označiti kao „dinamička agregatna inicijalizacija". Agregatna inicijalizacija koja je upotrebljena za niz **d** mora se koristiti prilikom definicije niza, ali pomoću druge vrste zapisa moguće je napraviti i inicijalizovati niz bilo gde. Na primer, pretpostavimo da metoda **sakrij()** koristi niz objekata klase **SferaOdBerilijuma**. Mogli biste je pozvati na sledeći način:

```
sakrij(d);
```

ali možete i dinamički da napravite niz koji želite da prosledite kao argument:

```
sakrij(new SferaOdBerilijuma[] { new SferaOdBerilijuma(),
new SferaOdBerilijuma() });
```

U mnogim situacijama ovakav način pisanja je pogodniji.
Izraz:

```
a=d;
```

pokazuje kako se referenca koja je povezana s jednim nizom može dodeliti drugom nizu, baš kao što se može uraditi s bilo kojim drugim tipom reference na objekat. Sada i **a** i **d** ukazuju na isti niz.

Drugi deo programa **MogućnostiNiza.java** pokazuje da se nizovi prostih tipova ponašaju isto kao i nizovi objekata, *osim* što nizovi prostih tipova direktno sadrže vrednosti prostih tipova.

Vežba 1: (2) Napravite metodu koja niz **SferaOdBerilijuma** prima kao argument. Pozovite tu metodu tako što ćete njen argument napraviti dinamički. Pokažite da u tom slučaju obična agregatna inicijalizacija nizova ne radi. Pronađite jedine situacije u kojima obična agregatna inicijalizacija nizova radi, i one gde je redundantna.

Vraćanje niza vrednosti

Pretpostavimo da pišete metodu koja ne treba da vrati jednu vrednost, već više njih. U jezicima kao što su C i C++ to nije lako, zato što ne možete da vratite ceo niz već samo pokazivač na niz. To stvara probleme jer postaje teško kontrolisati životni vek niza, što prouzrokuje neefikasno korišćenje memorije.

U Javi možete da vratite niz i nikada ne morate da brinete o njemu, jer će on postojati sve dok vam je potreban, a kada završite, obrisaće ga sakupljač smeća.

Kao primer, pogledajmo kako se vraća niz elemenata tipa **String**:

```
//: nizovi/Sladoled.java
// Vraćanje nizova iz metoda.
import java.util.*;

public class Sladoled {
  private static Random slucajan = new Random(47);
  static String[] UKUSI = {
    "čokolada", "jagoda",
    "vanila", "mentol",
    "moka", "rum",
    "praline", "voćna pita"
  };
  public static String[] skupUkusa(int n) {
    if (n > UKUSI.length);
       throw new IllegalArgumentException("Zadat prevelik");
    String[] rezultati = new String[n];
    boolean[] izabran = new boolean[UKUSI.length];
    for (int i = 0; i < n; i++) {
      int t;
      do
        t = slucajan.nextInt(UKUSI.length);
      while (izabran [t]);
      rezultati[i] = UKUSI[t];
      izabran[t] = true;
    }
    return rezultati;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++) {
      System.out.println(Arrays.toString(skupUkusa(3)));
  }
} /* Ispis:
[rum, mentol, moka]
[čokolada, jagoda, moka]
[jagoda, mentol, moka]
[rum, vanila, voćna pita]
[vanila, čokolada, moka]
[praline, jagoda, moka]
[moka, jagoda, mentol]
*///:~
```

Metoda **skupUkusa()** pravi niz elemenata tipa **String** nazvan **rezultati**. Dužina niza je određena argumentom koji se prosleđuje metodi. Nakon toga, nasumično se biraju ukusi iz niza **UKUSI** i stavljaju u niz **rezultati** koji se vraća kao rezultat metode. Vraćanje niza je isto kao vraćanje bilo kog objekta – radi se o referenci. Nije važno da li je niz napravljen unutar metode **skupUkusa()** ili na bilo kom drugom mestu. Skupljač smeća se brine o brisanju niza kada se završi rad s njim, tj. niz postoji sve dok vam je potreban.

Uzgred, dok **skupUkusa()** nasumično bira ukuse, obezbeđuje da isti element ne bude izabran dvaput. To se radi u petlji **do** koja bira nasumično sve dok ne pronađe neku vrednost što se još ne nalazi u nizu **izabran**. (Naravno, moglo je da se obavi i poređenje vrednosti objekata klase **String** da bi se ustanovilo da li se izabrani element već nalazi u nizu **rezultati**.) Ako pronađe odgovarajući element, dodaje nov ukus i traži sledeći (**i** se povećava za jedan).

Iz rezultata vidite da **skupUkusa()** svaki put bira ukuse slučajnim redosledom.

Vežba 2: (1) Napišite metodu koja prima **int** argument i vraća niz te veličine, popunjen objektima **SferaOdBerilijuma**.

Višedimenzionalni nizovi

Lako je praviti višedimenzionalne nizove. Za višedimenzionalni niz prostih tipova, svaki vektor niza pišete unutar vitičastih zagrada:

```
//: nizovi/VisedimenzionalniNizoviProstih.java
// Pravljenje višedimenzionalnih nizova.
import java.util.*;

public class VisedimenzionalniNizoviProstih {
  public static void main(String[] args) {
    int[][] a = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Ispis:
[[1, 2, 3], [4, 5, 6]]
*///:~
```

Sa svakim ugnežđenim skupom vitičastih zagrada prelazite na sledeći nivo niza.

U ovom primeru upotrebljena je metoda **Arrays.deepToString()** Jave SE5, koja višedimenzionalne nizove pretvara u znakovne nizove (objekte tipa **String**), kao što vidite iz rezultata.

Niz možete napraviti i pomoću rezervisane reči **new**. Evo jednog trodimenzionalnog niza napravljenog u izrazu **new**:

```
//: nizovi/TriDPomocuNew.java
import java.util.*;

public class TriDPomocuNew {
  public static void main(String[] args) {
    // 3-D niz zadate dužine:
    int[][][] a = new int[2][2][4];
    System.out.println(Arrays.deepToString(a));
  }
} /* Ispis:
```

```
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
*///:~
```

Vidite da se nizovi prostih tipova inicijalizuju automatski, ukoliko im inicijalizacijsku vrednost ne zadate eksplicitno. Nizovi objekata se inicijalizuju na **null**.

Svaki vektor nizova koji sačinjavaju matricu može biti proizvoljne dužine (naziva se: nepravilan, neregularan ili nepotpun niz):

```
//: nizovi/NepravilanNiz.java
import java.util.*;

public class NepravilanNiz {
  public static void main(String[] args) {
    Random slucajan = new Random(47);
    // 3-D niz s vektorima različite dužine:
    int[][][] a = new int[slucajan.nextInt(7)][][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new int[slucajan.nextInt(5)][];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = new int[slucajan.nextInt(5)];
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Ispis:
[[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]], [[0, 0, 0], [0],
[0, 0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0], []], [[0], [], [0]]]
*///:~
```

Prvi **new** pravi niz čiji je prvi element nasumično odabrane dužine, a ostali elementi neodređene dužine. Drugi **new** unutrašnje petlje **for** popunjava elemente, ali treći indeks ostavlja kao neodređen do trećeg **new**.

Na isti način možete praviti nizove neprostih objekata. Ovde ćete videti kako smo više **new** izraza okupili unutar vitičastih zagrada:

```
//: nizovi/ VisedimenzionalniNizoviObjekata.java
import java.util.*;

public class VisedimenzionalniNizoviObjekata {
  public static void main(String[] args) {
    SferaOdBerilijuma[][] sfere = {
      { new SferaOdBerilijuma(), new SferaOdBerilijuma() },
      { new SferaOdBerilijuma(), new SferaOdBerilijuma(),
        new SferaOdBerilijuma(), new SferaOdBerilijuma() },
      { new SferaOdBerilijuma(), new SferaOdBerilijuma(),
        new SferaOdBerilijuma(), new SferaOdBerilijuma(),
        new SferaOdBerilijuma(), new SferaOdBerilijuma(),
        new SferaOdBerilijuma(), new SferaOdBerilijuma() },
    };
    System.out.println(Arrays.deepToString(sfere));
  }
```

```
} /* Ispis:
[[Sfera 0, Sfera 1], [Sfera 2, Sfera 3, Sfera 4, Sfera 5], [Sfera 6,
Sfera 7, Sfera 8, Sfera 9, Sfera 10, Sfera 11, Sfera 12, Sfera 13]]
*///:~
```

Vidite da su i **sfere** nepravilan niz, u kojem su dužine svih lista objekata različite. Automatsko pakovanje radi i sa inicijalizatorima nizova:

```
//: nizovi/AutomatskoPakovanjeNizova.java
import java.util.*;

public class AutomatskoPakovanjeNizova {
  public static void main(String[] args) {
    Integer[][] a = { // Automatsko pakovanje:
      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
      { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
      { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
      { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Ispis:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26,
27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*///:~
```

Evo kako se parče po parče pravi niz neprostih objekata:

```
//: nizovi/SastavljanjeVisedimenzionalnihNizova.java
// Pravljenje višedimenzionalnih nizova.
import java.util.*;

public class SastavljanjeVisedimenzionalnihNizova {
  public static void main(String[] args) {
    Integer[][] a;
    a = new Integer[3][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer[3];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = i * j; // Automatsko pakovanje
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Ispis:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*///:~
```

Ono **i*j** služi samo za to da se nešto zanimljivo upiše u taj **Integer**.

Metoda **Arrays.deepToString()** radi i sa nizovima prostih i sa nizovima objekata:

```
//: nizovi/VisedimNizOmotaca.java
// Višedimenzionalni nizovi "omotačkih" objekata.
import java.util.*;

public class VisedimNizOmotaca {
  public static void main(String[] args) {
    Integer[][] a1 = { // Automatsko pakovanje
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    Double[][][] a2 = { // Automatsko pakovanje
      { { 1.1, 2.2 }, { 3.3, 4.4 } },
      { { 5.5, 6.6 }, { 7.7, 8.8 } },
      { { 9.9, 1.2 }, { 2.3, 3.4 } },
    };
    String[][] a3 = {
      { "The", "Quick", "Sly", "Fox" },
      { "Jumped", "Over" },
      { "The", "Lazy", "Brown", "Dog", "and", "friend" },
    };
    System.out.println("a1: " + Arrays.deepToString(a1));
    System.out.println("a2: " + Arrays.deepToString(a2));
    System.out.println("a3: " + Arrays.deepToString(a3));
  }
} /* Ispis:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]], [[9.9, 1.2],
[2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The, Lazy, Brown, Dog,
and, friend]]
*///:~
```

I opet, u **Integer** i **Double** nizovima, automatsko pakovanje Jave SE5 pravi omotačke objekte umesto nas.

Vežba 3: (4) Napišite metodu koja pravi i inicijalizuje dvodimenzionalni niz tipa **double**. Veličinu niza određuju argumenti metode, a inicijalizacione vrednosti su date opsegom određenim početnom i završnom vrednošću koje su i argumenti te metode. Napravite drugu metodu koja ispisuje niz generisan prvom metodom. U metodi **main**() testirajte metode pravljenjem i ispisivanjem nekoliko nizova različitih veličina.

Vežba 4: (2) Ponovite prethodnu vežbu za trodimenzionalni niz.

Vežba 5: (1) Pokažite da se višedimenzionalni nizovi neprostih objekata automatski inicijalizuju na **null**.

Vežba 6: (1) Napišite metodu koja prima dva **int** argumenta koji pokazuju odgovarajuće dimenzije 2-D niza. Metoda treba da napravi i popuni 2-D niz **SferaOdBerilijuma** u skladu sa argumentima dimenzija.

Vežba 7: (1) Ponovite prethodnu vežbu za 3-D niz.

Nizovi i generički tipovi

Po pravilu, nizovi i generički tipovi ne idu zajedno. Nije moguće napraviti instancu niza parametrizovanih tipova:

```
ljusti<Banana>[] ljusti = new ljusti<Banana>[10]; // Nedozvoljeno
```

Brisanjem se uklanja podatak o parametru tipa, a niz mora da zna tačan tip koji sadrži, da bi rad s tipovima proveravanjem učinio bezbednim.

Međutim, možete parametrizovati tip samog niza:

```
//: nizovi/ParametrizovanTipNiza.java

class ClassParametar<T> {
  public T[] f(T[] arg) { return arg; }
}

class ParametarMetode {
  public static <T> T[] f(T[] arg) { return arg; }
}

public class ParametrizovanTipNiza {
  public static void main(String[] args) {
    Integer[] celib = { 1, 2, 3, 4, 5 };
    Double[] bdouble = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Integer[] celib2 =
      new ClassParameter<Integer>().f(celib);
    Double[] bdouble2 =
      new ClassParameter<Double>().f(bdouble);
    celib2 = MethodParameter.f(celib);
    bdouble2 = MethodParameter.f(bdouble);
  }
} ///:~
```

Vodite računa o pogodnosti upotrebe parametrizovane metode umesto parametrizovane klase: ne morate da pravite instancu klase s parametrom za sve različite tipove na koje ćete je primeniti, i možete da je napravite statičnom. Naravno, ne možete uvek da upotrebite parametrizovanu metodu umesto parametrizovane klase, ali to bi moglo da bude bolje rešenje.

Nije najtačnije reći da je nemoguće napraviti niz generičkog tipa. Istina je da prevodilac neće dozvoliti pravljenje *instance* niza generičkog tipa. Međutim, dopustiće da napravite referencu na taj niz. Na primer:

```
List<String>[] ls;
```

Ovo će proći kroz prevodilac bez problema. I mada ne možete napraviti objekat niza koji sadrži generički tip, možete napraviti niz negeneričkog tipa i eksplicitno ga konvertovati:

```
//: nizovi/NizGenerickihTipova.java
// Moguće je napraviti niz generičkog tipa.
import java.util.*;

public class NizGenerickihTipova {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    List<String>[] ls;
    List[] la = new List[10];
    ls = (List<String>[])la; // Upozorenje "Unchecked"
    ls[0] = new ArrayList<String>();
    // Provera u vreme prevođenja daje grešku:
    //! ls[1] = new ArrayList<Integer>();

    // Problem: List<String> je podtip klase Object
    Object[] objekti = ls; // Dakle, dodeljivanje je OK
    // Prevodi se i izvršava bez pritužbe:
    objekti[1] = new ArrayList<Integer>();

    // Međutim, za jednostavne potrebe moguće je
    // napraviti niz generičkih tipova, iako
    // će se javiti upozorenje "unchecked":
    List<SferaOdBerilijuma>[] sfere =
      (List<SferaOdBerilijuma>[])new List[10];
    for(int i = 0; i < sfere.length; i++)
      sfere[i] = new ArrayList<SferaOdBerilijuma>();
  }
} ///:~
```

Čim imate referencu na **List<String>**[], neke provere će se obaviti u vreme prevođenja. Problem je to što su nizovi kovarijantni, pa je niz **List<String>**[] istovremeno i niz **Object**[]. To možete upotrebiti za dodeljivanje liste **ArrayList<Integer>** svom nizu, a da ne izazovete grešku ni u vreme prevođenja ni u vreme izvršavanja.

Ukoliko znate da nećete svoditi naviše i potrebe su vam relativno jednostavne, lako ćete napraviti niz generičkih tipova koji će biti elementarno proveravan u vreme prevođenja. Međutim gotovo uvek je bolje izabrati generički kontejner nego niz generičkih tipova.

Po pravilu, videćete da su generički tipovi delotvorni na *granicama* klase ili metode. U njenoj unutrašnjosti, brisanje čini generički tip neupotrebljivim. Stoga ne možete, na primer, napraviti niz generičkog tipa:

```
//: nizovi/NizGenerickogTipa.java
// Java neće da prevede nizove generičkih tipova.

public class NizGenerickogTipa<T> {
  T[] niz; // OK
  @SuppressWarnings("unchecked")
  public ArrayOfGenericType(int velicina) {
```

```
    //! niz = new T[velicina]; // Nedozvoljeno
    niz = (T[])new Object[velicina]; // Upozorenje "unchecked"
  }
  // Nedozvoljeno:
  //! public <U> U[] napraviNiz() { return new U[10]; }
} ///:~
```

Brisanje ponovo smeta – u ovom primeru pokušali smo da napravimo nizove tipova koji su bili obrisani i stoga su nepoznati. Imajte u vidu da niz tipa **Object** *možete* napraviti i eksplicitno konvertovati, ali bez anotacije **@SuppressWarnings** dobili biste upozorenje „unchecked" u vreme prevođenja, zato što niz niti sadrži tip **T** niti se dinamički proverava na tip **T**. Drugim rečima, ako napravite **String**[], Java će se i u vreme prevođenja i u vreme izvršavanja postarati da u taj niz možete smeštati samo objekte tipa **String**. Međutim, ukoliko napravite niz **Object**[], u njega ćete moći da smeštate sve sem prostih tipova.

Vežba 8: (1) Dokažite tvrdnje iz prethodnog pasusa.

Vežba 9: (3) Napravite klase potrebne za primer **ljusti<Banana>** i pokažite da ih prevodilac ne prihvata. Rešite problem tako što ćete napraviti objekat tipa **ArrayList**.

Vežba 10: (2) Izmenite **NizGenerickihTipova.java** tako da se umesto nizova koriste kontejneri. Pokažite da se tako gube i upozorenja u vreme prevođenja.

Pravljenje podataka za testiranje

Prilikom eksperimentisanja s nizovima i programima uopšte, odesno je imati mogućnost lakog generisanja nizova popunjenih podacima za testiranje. Alatke opisane u ovom odeljku popunjavaju niz vrednostima ili objektima.

Arrays.fill()

Standardna klasa **Arrays** u Javi ima svoju metodu **fill**(), ali je ona prilično prosta – samo kopira jednu vrednost u svaki element niza, a ako se radi o objektima, kopira istu referencu u svaku lokaciju. Evo primera:

```
//: nizovi/PopunjavanjeNizova.java
// Upotreba metode Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class PopunjavanjeNizova {
  public static void main(String[] args) {
    int velicina = 6;
    boolean[] a1 = new boolean[velicina];
    byte[] a2 = new byte[velicina];
    char[] a3 = new char[velicina];
    short[] a4 = new short[velicina];
    int[] a5 = new int[velicina];
    long[] a6 = new long[velicina];
```

```
    float[] a7 = new float[velicina];
    double[] a8 = new double[velicina];
    String[] a9 = new String[velicina];
    Arrays.fill(a1, true);
    print("a1 = " + Arrays.toString(a1));
    Arrays.fill(a2, (byte)11);
    print("a2 = " + Arrays.toString(a2));
    Arrays.fill(a3, 'x');
    print("a3 = " + Arrays.toString(a3));
    Arrays.fill(a4, (short)17);
    print("a4 = " + Arrays.toString(a4));
    Arrays.fill(a5, 19);
       print("a5 = " + Arrays.toString(a5));
       Arrays.fill(a6, 23);
    print("a6 = " + Arrays.toString(a6));
    Arrays.fill(a7, 29);
       print("a7 = " + Arrays.toString(a7));
    Arrays.fill(a8, 47);
       print("a8 = " + Arrays.toString(a8));
    Arrays.fill(a9, "Zdravo");
    print("a9 = " + Arrays.toString(a9));
    // Manipulisanje opsezima:
    Arrays.fill(a9, 3, 5, "svima");
    print("a9 = " + Arrays.toString(a9));
  }
} /* Ispis:
a1 = [true, true, true, true, true, true]
a2 = [11, 11, 11, 11, 11, 11]
a3 = [x, x, x, x, x, x]
a4 = [17, 17, 17, 17, 17, 17]
a5 = [19, 19, 19, 19, 19, 19]
a6 = [23, 23, 23, 23, 23, 23]
a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9 = [Zdravo, Zdravo, Zdravo, Zdravo, Zdravo, Zdravo]
a9 = [Zdravo, Zdravo, Zdravo, svima, svima, Zdravo]
*///:~
```

Možete da popunite ceo niz ili, kao što pokazuju poslednje naredbe, nekoliko elemenata niza. Medutim, pošto metodom **Arrays.fill**() možete da popunite niz samo jednom vrednošću, rezultati nisu naročito upotrebljivi.

Generatori podataka

Za pravljenje zanimljivijih nizova podataka, ali na fleksibilan način, upotrebićemo koncept **Generator**a predstavljen u poglavlju *Generički tipovi*. Ako neka alatka upotrebljava **Generator**, napravljeni podaci se menjaju u zavisnosti od izabranog **Generator**a (to je

primer projektnog obrasca *Strategy* (Strategija) – različiti **Generatori** predstavljaju različite strategije[1]).

U ovom odeljku napravićemo nekoliko **Generatora**; kao što ste već videli, lako ćete definisati sopstvene.

Prvo, napravićemo osnovni skup generatora za brojanje svih omotača prostih tipova i objekata tipa **String**. Klase generatora su ugnežđene u klasi **GeneratorDateKolicine** da bi mogle biti nazvane imenom tipa objekata koje generišu; na primer, generator koji pravi objekte tipa **Integer** pravimo izrazom **new GeneratorDateKolicine.Integer()**:

```
//: net/mindview/util/GeneratorDateKolicine.java
// Jednostavne realizacije generatora.
package net.mindview.util;

public class GeneratorDateKolicine {
  public static class
  Boolean implements Generator<java.lang.Boolean> {
    private boolean vrednost = false;
    public java.lang.Boolean next() {
      vrednost = !vrednost; // Napred - nazad
      return vrednost;
    }
  }
  public static class
  Byte implements Generator<java.lang.Byte> {
    private byte vrednost = 0;
    public java.lang.Byte next() { return vrednost++; }
  }
  static char[] znakovi = ("abcdefghijklmnopqrstuvwxyz" +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
  public static class
  Character implements Generator<java.lang.Character> {
    int indeks = -1;
    public java.lang.Character next() {
      indeks = (indeks + 1) % znakovi.length;
      return znakovi[indeks];
    }
  }
  public static class
  String implements Generator<java.lang.String> {
    private int duzina = 7;
    Generator<java.lang.Character> cg = new Character();
    public String() {}
    public String(int duzina) { this.duzina = duzina; }
    public java.lang.String next() {
      char[] baf = new char[duzina];
      for(int i = 0; i < duzina; i++)
```

[1] Iako to nije baš sasvim jasno. Moglo bi se tvrditi i da **Generator** predstavlja obrazac *Command* (Komanda). Međutim, ja smatram da je zadatak popuniti niz i da **Generator** obavlja deo tog zadatka, pa mi to više liči na strategiju nego na komandu.

```
        baf[i] = cg.next();
      return new java.lang.String(baf);
    }
  }
  public static class
  Short implements Generator<java.lang.Short> {
    private short vrednost = 0;
    public java.lang.Short next() { return vrednost++; }
  }
  public static class
  Integer implements Generator<java.lang.Integer> {
    private int vrednost = 0;
    public java.lang.Integer next() { return vrednost++; }
  }
  public static class
  Long implements Generator<java.lang.Long> {
    private long vrednost = 0;
    public java.lang.Long next() { return vrednost++; }
  }
  public static class
  Float implements Generator<java.lang.Float> {
    private float vrednost = 0;
    public java.lang.Float next() {
      float rezultat = vrednost;
      vrednost += 1.0;
      return rezultat;
    }
  }
  public static class
  Double implements Generator<java.lang.Double> {
    private double vrednost = 0.0;
    public java.lang.Double next() {
      double rezultat = vrednost;
      vrednost += 1.0;
      return rezultat;
    }
  }
} ///:~
```

Svaka klasa realizuje neko značenje pojma količina. U slučaju klase **GeneratorDateKolicine.Character**, radi se o velikim i malim slovima koja se ponavljaju. Klasa **GeneratorDateKolicine.String** upotrebljava **GeneratorDateKolicine.Character** za popunjavanje niza znakova, koje zatim pretvara u **String**. Veličinu niza određuje argument konstruktora. Obratite pažnju na to da **GeneratorDateKolicine.String** upotrebljava elementarni **Generator<java.lang.Character>** umesto specifične reference na **GeneratorDateKolicine.Character**. Kasnije taj generator može zameniti **RandomGenerator.String** iz paketa **RandomGenerator.java**.

Evo jedne alatke za testiranje koja upotrebljava refleksiju sa ugneždenim **Generator**om, pa se može primeniti za testiranje proizvoljnog skupa **Generator**a ovog oblika:

```
//: nizovi/TestiranjeGeneratora.java
import net.mindview.util.*;

public class TestiranjeGeneratora {
  public static int velicina = 10;
  public static void test(Class<?> okolnaKlasa) {
    for(Class<?> tip : okolnaKlasa.getClasses()) {
      System.out.print(tip.getSimpleName() + ": ");
      try {
        Generator<?> g = (Generator<?>)tip.newInstance();
        for(int i = 0; i < velicina; i++)
          System.out.printf(g.next() + " ");
        System.out.println();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  public static void main(String[] args) {
    test(GeneratorDateKolicine.class);
  }
} /* Ispis:
Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Long: 0 1 2 3 4 5 6 7 8 9
Integer: 0 1 2 3 4 5 6 7 8 9
Short: 0 1 2 3 4 5 6 7 8 9
String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP QRSTUVW XYZabcd
efghijk lmnopqr
Character: a b c d e f g h i j
Byte: 0 1 2 3 4 5 6 7 8 9
Boolean: true false true false true false true false true false
*///:~
```

Ovde se pretpostavlja da klasa koja se ispituje sadrži skup ugneždenih objekata tipa **Generator**, od kojih svaki ima podrazumevani konstruktor (onaj bez argumenata). Sve ugneždene klase proizvodi reflektivna metoda **getClasses()**. Zatim metoda **test()** pravi instancu svakog od tih generatora i ispisuje rezultat koji se dobija s deset poziva metode **next()**.

Evo skupa **Generator**a koji upotrebljavaju generator slučajnih brojeva. Pošto se konstruktor klase **Random** inicijalizuje konstantom, rezultat se ponavlja svaki put kada pomoću jednog od ovih **Generator**a pozovete program:

```
//: net/mindview/util/GeneratorSlucajnih.java
// Generatori koji daju slučajne vrednosti.
package net.mindview.util;
import java.util.*;
```

```
public class GeneratorSlucajnih {
     private static Random r = new Random(47);
  public static class
  Boolean implements Generator<java.lang.Boolean> {
    public java.lang.Boolean next() {
      return r.nextBoolean();
    }
  }
  public static class
  Byte implements Generator<java.lang.Byte> {
    public java.lang.Byte next() {
      return (byte)r.nextInt();
    }
  }
  public static class
  Character implements Generator<java.lang.Character> {
    public java.lang.Character next() {
      return GeneratorDateKolicine.znakovi[
        r.nextInt(GeneratorDateKolicine.znakovi.duzina)];
    }
  }
  public static class
     String extends GeneratorDateKolicine.String
    // Uključićemo generator slučajnih znakova:
    { cg = new Character(); } // Inicijalizator instance
    public String() {}
    public String(int duzina) { super(duzina); }
  }
  public static class
  Short implements Generator<java.lang.Short> {
    public java.lang.Short next() {
      return (short)r.nextInt();
    }
  }
  public static class
  Integer implements Generator<java.lang.Integer> {
    private int mod = 10000;
    public Integer() {}
    public Integer(int modulo) { mod = modulo; }
    public java.lang.Integer next() {
      return r.nextInt(mod);
    }
  }
  public static class
  Long implements Generator<java.lang.Long> {
    private int mod = 10000;
    public Long() {}
    public Long(int modulo) { mod = modulo; }
    public java.lang.Long next() {
```

```
      return new java.lang.Long(r.nextInt(mod));
    }
  }
  public static class
  Float implements Generator<java.lang.Float> {
    public java.lang.Float next() {
      // Odseci sva decimalna mesta posle drugog:
      int odseceno = Math.round(r.nextFloat() * 100);
      return ((float)odseceno) / 100;
    }
  }
  public static class
  Double implements Generator<java.lang.Double> {
    public java.lang.Double next() {
      long odseceno = Math.round(r.nextDouble() * 100);
      return ((double)odseceno) / 100;
    }
  }
} ///:~
```

Vidite da **GeneratorSlucajnih.String** nasleduje **GeneratorDateKolicine.String** i jednostavno uključuje **Character** generator.

Za generisanje brojeva koji nisu preveliki, **GeneratorSlucajnih.Integer** koristi standardnu vrednost modula 10.000, ali preklopljeni konstruktor omogućuje i izbor manje vrednosti. Isti pristup je upotrebljen i za **GeneratorSlucajnih.Long**. Za **Generatore Float** i **Double**, vrednosti iza decimalne tačke se odsecaju.

Za testiranje klase **GeneratorSlucajnih** ponovo ćemo upotrebiti **TestiranjeGeneratora**:

```
//: nizovi/TestiranjeGeneratoraSlucajnih.java
import net.mindview.util.*;

public class TestiranjeGeneratoraSlucajnih {

  public static void main(String[] args) {
    TestiranjeGeneratora.test(GeneratorSlucajnih.class);
  }
} /* Ispis:
Double: 0.73 0.53 0.16 0.19 0.52 0.27 0.26 0.05 0.8 0.76
Float: 0.53 0.16 0.53 0.4 0.49 0.25 0.8 0.11 0.02 0.8
Long: 7674 8804 8950 7826 4322 896 8033 2984 2344 5810
Integer: 8303 3141 7138 6012 9966 8689 7185 6992 5746 3976
Short: 3358 20592 284 26791 12834 -8092 13656 29324 -1423 5327
String: bkInaMe sbtWHkj UrUkZPg wsqPzDy CyRFJQA HxxHvHq XumcXZJ oogoYWM
NvqeuTp nXsgqia
Character: x x E A J J m z M s
Byte: -60 -17 55 -14 -5 115 39 -37 79 115
Boolean: false true false false true true true true true true
*///:~
```

Broj ispitnih vrednosti određuje javno polje **TestiranjeGeneratora.velicina** koje možete menjati.

Pravljenje nizova od Generatora

Za pravljenje niza od **Generatora** potrebne su nam dve alatke za konverziju. Prva pomoću bilo kojeg **Generatora** pravi niz podtipova klase **Object**. Da bismo rešili problem prostih tipova, druga alatka prima proizvoljan niz omotača prostih tipova i proizvodi odgovarajući niz prostih tipova.

Prva alatka ima dve opcije koje predstavlja preklopljena statična metoda **array**(). Prva verzija metode prima postojeći niz i popunjava ga **Generatorom**, a druga uzima jedan objekat tipa **Class**, jedan **Generator** i željeni broj elemenata, i pravi nov niz koji se opet popunjava **Generatorom**. Vodite računa o tome da ova alatka proizvodi samo nizove podtipova klase **Object** i da ne može da pravi nizove prostih tipova:

```
//: net/mindview/util/Generisani.java
package net.mindview.util;
import java.util.*;

public class Generisani {
  // Popunjavanje postojećeg niza:
  public static <T> T[] niz(T[] a, Generator<T> gen) {
    return new CollectionData<T>(gen, a.duzina).toArray(a);
  }
  // Napravi nov niz:
  @SuppressWarnings("unchecked")
  public static <T> T[] niz(Class<T> tip,
      Generator<T> gen, int velicina) {
    T[] a =
      (T[])java.lang.reflect.Array.newInstance(tip, velicina);
    return new CollectionData<T>(gen, velicina).toArray(a);
  }
} ///:~
```

Definicija klase **CollectionData** navedena je u poglavlju *Detaljno razmatranje kontejnera*. Ona pravi **Collection** objekat popunjen elementima koje je napravio **Generator gen**. Broj tih elemenata određuje drugi argument konstruktora. Svi podtipovi klase **Collection** (kontejneri) imaju metodu **toArray**() koja elementima kontejnera popunjava niz zadat argumentom.

Druga metoda upotrebljava refleksiju za dinamičko pravljenje novog niza odgovarajućeg tipa i veličine. On se zatim popunjava isto kao u prvoj metodi.

Klasu **Generisani** možemo ispitati nekom od klasa **GeneratorDateKolicine** definisanih u prethodnom odeljku:

```
//: nizovi/TestGenerisanih.java
import java.util.*;
import net.mindview.util.*;
```

```
public class TestGenerisanih {
  public static void main(String[] args) {
    Integer[] a = { 9, 8, 7, 6 };
    System.out.println(Arrays.toString(a));
    a = Generisani.array(a,new GeneratorDateKolicine.Integer());
    System.out.println(Arrays.toString(a));
    Integer[] b = Generisani.array(Integer.class,
        new GeneratorDateKolicine.Integer(), 15);
    System.out.println(Arrays.toString(b));
  }
} /* Ispis:
[9, 8, 7, 6]
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Iako je niz **a** inicijalizovan, njegove vrednosti nestaju kada se propusti kroz metodu **Generisani.array()** koja ih zamenjuje (ali ne menja prvobitno mesto niza u memoriji). Inicijalizacija niza **b** pokazuje kako se popunjen niz pravi od nule.

Generički tipovi ne rade s prostim tipovima, a mi generatore hoćemo da upotrebimo za popunjavanje nizova prostih tipova. Problem smo rešili tak što smo napravili konvertor koji prima proizvoljan niz omotačkih objekata i konvertuje ga u niz odgovarajućih prostih tipova. Da nema te alatke, morali bismo da pravimo zaseban generator za svaki prost tip.

```
//: net/mindview/util/KonvertujU.java
package net.mindview.util;

public class KonvertujU {
  public static boolean[] prost(Boolean[] ulaz) {
    boolean[] rezultat = new boolean[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i]; // Automatsko raspakivanje
    return rezultat;
  }
  public static char[] prost(Character[] ulaz) {
    char[] rezultat = new char[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static byte[] prost(Byte[] ulaz) {
    byte[] rezultat = new byte[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static short[] prost(Short[] ulaz) {
```

```
    short[] rezultat = new short[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static int[] prost(Integer[] ulaz) {
    int[] rezultat = new int[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static long[] prost(Long[] ulaz) {
    long[] rezultat = new long[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static float[] prost(Float[] ulaz) {
    float[] rezultat = new float[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
  public static double[] prost(Double[] ulaz) {
    double[] rezultat = new double[ulaz.duzina];
    for(int i = 0; i < ulaz.duzina; i++)
      rezultat[i] = ulaz[i];
    return rezultat;
  }
} ///:~
```

Svaka verzija metode **prost**() pravi odgovarajući niz prostih tipova tačne dužine i zatim u njega kopira elemente niza omotačkih tipova **ulaz**. Obratite pažnju na to da se automatsko raspakivanje dešava u izrazu:

```
rezultat[i] = ulaz[i];
```

Evo primera iz kojega možete videti kako se klasa **KonvertujU** upotrebljava sa obe verzije metode **Generisani.array**():

```
//: nizovi/PrimerKonverzijeProstih.java
import java.util.*;
import net.mindview.util.*;

public class PrimerKonverzijeProstih {
  public static void main(String[] args) {
    Integer[] a = Generisani.niz(Integer.class,
        new GeneratorDateKolicine.Integer(), 15);
    int[] b = KonvertujU.prost(a);
```

```
    System.out.println(Arrays.toString(b));
    boolean[] c = KonvertujU.prost(
      Generisani.niz(Boolean.class,
        new GeneratorDateKolicine.Boolean(), 7));
    System.out.println(Arrays.toString(c));
  }
} /* Ispis:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[true, false, true, false, true, false, true]
*///:~
```

Najzad, ovo je program za testiranje alatki za generisanje nizova pomoću klasa **GeneratorSlucajnih**:

```
//: nizovi/TestGenerisanjaNizova.java
// Testiranje alatki koje nizove popunjavaju generatorima.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestGenerisanjaNizova {
  public static void main(String[] args) {
    int velicina = 6;
    boolean[] a1 = KonvertujU.prost(Generisani.niz(
      Boolean.class, new GeneratorSlucajnih.Boolean(), velicina));
    print("a1 = " + Arrays.toString(a1));
    byte[] a2 = KonvertujU.prost(Generisani.niz(
      Byte.class, new GeneratorSlucajnih.Byte(), velicina));
    print("a2 = " + Arrays.toString(a2));
    char[] a3 = KonvertujU.prost(Generisani.niz(
      Character.class,
      new GeneratorSlucajnih.Character(), velicina));
    print("a3 = " + Arrays.toString(a3));
    short[] a4 = KonvertujU.prost(Generisani.niz(
      Short.class, new GeneratorSlucajnih.Short(), velicina));
    print("a4 = " + Arrays.toString(a4));
    int[] a5 = KonvertujU.prost(Generisani.niz(
      Integer.class, new GeneratorSlucajnih.Integer(), velicina));
    print("a5 = " + Arrays.toString(a5));
    long[] a6 = KonvertujU.prost(Generisani.niz(
      Long.class, new GeneratorSlucajnih.Long(), velicina));
    print("a6 = " + Arrays.toString(a6));
    float[] a7 = KonvertujU.prost(Generisani.niz(
      Float.class, new GeneratorSlucajnih.Float(), velicina));
    print("a7 = " + Arrays.toString(a7));
    double[] a8 = KonvertujU.prost(Generisani.niz(
      Double.class, new GeneratorSlucajnih.Double(), velicina));
    print("a8 = " + Arrays.toString(a8));
  }
```

```
} /* Ispis:
a1 = [true, false, true, false, false, true]
a2 = [104, -79, -76, 126, 33, -64]
a3 = [Z, n, T, c, Q, r]
a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
a5 = [7704, 7383, 7706, 575, 8410, 6342]
a6 = [7674, 8804, 8950, 7826, 4322, 896]
a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:~
```

Time se obezbeđuje i da sve verzije metode **KonvertujU.prost()** ispravno rade.

Vežba 11: (2) Pokažite da automatsko raspakivanje ne radi s nizovima.

Vežba 12: (1) Napravite inicijalizovan niz tipa **double** pomoću klase **GeneratorDateKolicine**. Ispišite rezultate.

Vežba 13: (2) Popunite objekat tipa **String** pomoću klase **GeneratorDateKolicine.Character**.

Vežba 14: (6) Napravite po jedan niz svakog od prostih tipova, zatim ih popunite pomoću klase **GeneratorDateKolicine**. Ispišite svaki niz.

Vežba 15: (2) Izmenite **PoredjenjeSKontejnerima.java** tako što ćete napraviti **Generator** za objekte tipa **SferaOdBerilijuma**, i prepravite metodu **main()** tako da taj **Generator** upotrebi za **Generisani.array()**.

Vežba 16: (3) Na osnovu programa **GeneratorDateKolicine.java**, napravite klasu **PreskociGenerator** koja nove vrednosti proizvodi povećavanjem za 1, u skladu s argumentom konstruktora. Prepravite program **TestGenerisanjaNizova.java** tako da pokaže da vaša nova klasa radi ispravno.

Vežba 17: (5) Napravite i ispitajte **Generator** za tip **BigDecimal**, i obezbedite da radi s metodama klase **Generisani**.

Metode klase **Arrays**

U paketu **java.util** pronaći ćete klasu **Arrays** koja sadrži skup statičkih uslužnih funkcija za nizove. Postoji šest osnovnih funkcija: **equals()**, za poređenje jednakosti dva niza (i **deepEquals()** za višedimenzionalne nizove), **fill()** za popunjavanje niza određenom vrednošću (već ste je upoznali u prethodnom delu poglavlja), **sort()** za uređivanje elemenata niza, **binarySearch()** za pronalaženje elementa u uređenom nizu, **toString()** za pravljenje **String** objekta koji predstavlja niz, i **hashCode()**, za transformisanje ključa niza (saznaćete šta to znači u poglavlju *Detaljno razmatranje kontejnera*). Sve ove metode postoje i za nizove prostih tipova i za nizove tipa objekata. Osim toga, postoji i jedinstvena metoda **asList()** koja proizvoljan niz pretvara u kontejner tipa **List**, o čemu ste saznali u poglavlju *Čuvanje objekata*.

Pre nego što razmotrimo metode klase **Arrays**, pogledaćemo jednu korisnu metodu koja nije njen deo.

Kopiranje niza

Standardna biblioteka u Javi obezbeđuje statičku metodu **System.arraycopy()** pomoću koje se niz mnogo brže kopira nego kada se koristi petlja **for** za ručno kopiranje. **System.arraycopy()** je preklopljena i radi s nizovima svih tipova podataka. Evo primera kako ova metoda radi s nizovima tipa **int**:

```
//: nizovi/KopiranjeNizova.java
// Upotreba metode System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class KopiranjeNizova {
  public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    print("i = " + Arrays.toString(i));
    print("j = " + Arrays.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    print("j = " + Arrays.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    print("k = " + Arrays.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    print("i = " + Arrays.toString(i));
    // Objekti:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    print("u = " + Arrays.toString(u));
    print("v = " + Arrays.toString(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    print("u = " + Arrays.toString(u));
  }
} /* Ispis:
i = [47, 47, 47, 47, 47, 47, 47]
j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]
k = [47, 47, 47, 47, 47]
i = [103, 103, 103, 103, 103, 47, 47]
u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]
v = [99, 99, 99, 99, 99]
u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]
*///:~
```

Argumenti metode **arraycopy()** su izvorni niz, pomeraj u izvornom nizu od koga treba započeti kopiranje, odredišni niz, pomeraj u odredišnom nizu od koga treba da počne kopiranje i broj elemenata za kopiranje. Prirodno, svako prekoračenje granica niza prouzrokovaće izuzetak.

Primer pokazuje da se mogu kopirati i nizovi prostih tipova i nizovi objekata. Međutim, ako kopirate nizove objekata, kopiraju se samo reference, ali ne i sami objekti. To se zove *površno kopiranje* (engl. *shallow copy*, videti dodatke knjige na Webu).

Metoda **System.arraycopy()** ne obavlja automatsko pakovanje niti automatsko raspakivanje – oba niza moraju biti tačno istog tipa.

Vežba 18: (3) Napravite i popunite niz objekata tipa **SferaOdBerilijuma**. Kopirajte taj niz u drugi, nov niz i pokažite da je to površna kopija.

Poređenje nizova

Klasa **Arrays** nudi preklopljene verzije metode **equals()** za poređenje jednakosti dva niza. I ova metoda postoji za nizove svih prostih tipova i tipa **Object**. Da bi bili jednaki, nizovi moraju da imaju isti broj elemenata, i svaki element mora da bude jednak odgovarajućem elementu drugog niza, pri čemu se metoda **equals ()** poziva za svaki element. (Za proste tipove se koristi metoda **equals()** omotačke klase, npr. **Integer.equals()** za tip **int**.) Evo primera:

```
//: nizovi/PoredjenjeNizova.java
// Korišćenje metode Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class PoredjenjeNizova {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    print(Arrays.equals(a1, a2));
    a2[3] = 11;
    print(Arrays.equals(a1, a2));
    String[] s1 = new String[4];
    Arrays.fill(s1, "Cao");
    String[] s2 = {new String("Cao"), new String("Cao"),
       new String("Cao"),  new String("Cao") };
    print(Arrays.equals(s1, s2));
  }
} /* Ispis:
true
false
true
*///:~
```

U početku, nizovi **a1** i **a2** su jednaki pa je povratna vrednost **true**, ali se zatim jedan element izmeni pa je drugi red rezultata **false**. U poslednjem slučaju, svi elementi **s1** ukazuju na isti objekat, dok **s2** ima pet jedinstvenih objekata. Međutim, jednakost nizova se određuje prema sadržaju (metodom **Object.equals()**), pa je rezultat **true**.

Vežba 19: (2) Napravite klasu sa **int** poljem koje se inicijalizuje pomoću argumenta konstruktora. Napravite dva niza tih objekata koristeći identične inicijalizacione vrednosti za oba niza i pokažite da metoda **Arrays.equals()** kaže da oni nisu jednaki. Rešite problem tako što ćete dodati metodu **equals()** u svoju klasu.

Vežba 20: (4) Pokažite rad metode **deepEquals()** za višedimenzionalne nizove.

Poređenje elemenata niza

Uređivanje se mora izvoditi na osnovu stvarnog tipa objekta. Naravno, jedan pristup bio bi da se piše posebna metoda uređivanja za svaki tip podataka, ali verovatno već shvatate da se tako ne dobija kôd koji se može lako koristiti s različitim tipovima.

Najvažniji cilj programiranja jeste da se razdvoji ono što se menja od onoga što ostaje isto, a ovde je kôd koji se ne menja opšti algoritam uređivanja, dok je ono što se menja od jedne do druge primene zapravo način poređenja objekata. Dakle, umesto ugrađivanja koda za poređenje u različite potprograme za uređivanje, koristi se projektni obrazac *Strategy*.[2] Strategija znači da se deo koda koji se razlikuje od slučaja do slučaja kapsulira unutar posebne klase (objekta tipa Strategija). Objekat tipa Strategija predajete delu koda koji je uvek isti, a on pomoću te Strategije izvršava svoj algoritam. Na taj način se mogu napraviti različiti objekti koji će izražavati različite načine poređenja i dostavljaće ih istom kodu za uređivanje.

U Javi postoje dva načina za uređivanje. Prva je metoda „prirodnog" poređenja koja se ugrađuje u klasu tako što se realizuje interfejs **java.lang.Comparable**. To je veoma jednostavan interfejs s jednom metodom, **compareTo()**. Ova metoda prima kao argument drugi objekat istog tipa i vraća negativnu vrednost ako je tekući objekat manji od argumenta, nulu ako je tekući objekat jednak argumentu, odnosno pozitivnu vrednost ako je tekući objekat veći od argumenta.

Evo klase koja realizuje interfejs **Comparable** i ilustruje poređenje nizova pomoću metode Javine standardne biblioteke **Arrays.sort()**:

```
//: nizovi/UporedivTip.java
// Primena interfejsa Comparable.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class UporedivTip implements Comparable {
  int i;
  int j;
```

[2] *Design Patterns*, Erich Gamma i dr. (Addison-Wesley, 1995). Videti *Thinking in Patterns (with Java)* na lokaciji *www.MindView.net*.

```
  private static int broj = 1;
  public UporedivTip(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString() {
    String rezultat = "[i = " + i + ", j = " + j + "]";
    if(broj++ % 3 == 0)
      rezultat += "\n";
    return rezultat;
  }
  }
  public int compareTo(UporedivTip rv) {
    return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
  }
  private static Random r = new Random(47);
  public static Generator<UporedivTip> generator() {
    return new Generator<UporedivTip>() {
      public UporedivTip next() {
        return new UporedivTip(r.nextInt(100), r.nextInt(100));
      }
    };
  }
  public static void main(String[] args) {
    UporedivTip[] a =
      Generisan.niz(new UporedivTip[12], generator());
    print("pre uređivanja: ");
    print(Arrays.toString(a));
    Arrays.sort(a);
    print("nakon uređivanja: ");
    print(Arrays.toString(a));
  }
} /* Ispis:
pre uređivanja:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
nakon uređivanja:
[[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
, [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
, [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
, [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
]
*///:~
```

Kada definišete funkciju poređenja, sami definišete šta znači uporediti jedan objekat vaše klase s drugim. Ovde se za poređenje koriste samo vrednosti polja **i**, a polje **j** se ne uzima u obzir.

Metoda **generator**() vraća objekat koji realizuje interfejs **Generator**, tako što pravi anonimnu unutrašnju klasu. Na taj način, inicijalizacijom slučajnim vrednostima, prave se objekti klase **UporedivTip**. U metodi **main**(), za popunjavanje niza elemenata tipa **UporedivTip** koristi se generator, a niz se potom uređuje. Da interfejs **Comparable** nije bio realizovan, pojavila bi se greška **ClassCastException** u vreme prevođenja prilikom pokušaja pozivanja metode **sort**(). To bi se desilo zato što metoda **sort**() tip svog argumenta konvertuje u **Comparable**.

Pretpostavimo da radite s klasom koja ne realizuje interfejs **Comparable**, ili da klasa *realizuje* taj interfejs, ali vam se ne sviđa kako ona radi i za određeni tip podataka radije biste koristili neku drugu funkciju za poređenje. Da biste to postigli, morate da primenite drugi pristup za poređenje objekata. Treba da napravite posebnu klasu koja realizuje interfejs **Comparator** (ukratko predstavljen u poglavlju *Čuvanje objekata*). Ovo je primer projektnog obrasca *Strategy*. **Comparator** ima dve metode, **compare**() i **equals**(). Međutim, **equals**() se primenjuje (realizuje) samo kada su potrebne specijalne performanse, jer se pri svakom pravljenju klase ona automatski nasleđuje iz klase **Object** koja ima svoju metodu **equals**(). Znači, možete da koristite standardnu metodu **equals**() klase **Object** i da zadovoljite uslove koje nameće interfejs.

Klasa **Collections** (njome ćemo se pozabaviti u narednom poglavlju) sadrži metodu **reverseOreder**(); metoda pravi **Comparator** koji obrće prirodan redosled uređivanja. To se jednostavno primenjuje na našu klasu **UporedivTip**:

```
//: nizovi/ObrnutiRedosled.java
// Prikazuje Collections.reverseOrder().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class ObrnutiRedosled {
  public static void main(String[] args) {
    UporedivTip[] a = Generisan.niz(
       new UporedivTip[12], UporedivTip.generator());
     print("pre uređivanja: ");
     print(Arrays.toString(a));
     Arrays.sort(a, Collections.reverseOrder());
     print("nakon uređivanja: ");
     print(Arrays.toString(a));
  }
} /* Ispis:
pre uređivanja:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
nakon uređivanja:
[[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
, [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
, [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
, [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
]
*///:~
```

Možete napisati i sopstveni **Comparator**. Sledeći **Comparator** poredi objekte tipa **UporedivTip** po vrednosti njihovih polja **j**, a ne po vrednosti polja **i**:

```
//: nizovi/PrimerZaComparator.java
// Primena interfejsa Comparator za neku klasu.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class ComparatorZaUporedivTip implements Comparator<UporedivTip> {
  public int compare(UporedivTip o1, UporedivTip o2) {
    return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
  }
}
public class PrimerZaComparator {
  public static void main(String[] args) {
    UporedivTip[] a = Generisan.niz(
       new UporedivTip[12], UporedivTip.generator());
     print("pre uređivanja: ");
     print(Arrays.toString(a));
     Arrays.sort(a, new CompTypeComparator());
     print("nakon uređivanja: ");
     print(Arrays.toString(a));
  }
} /* Ispis:
pre uređivanja:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
nakon uređivanja:
[[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
, [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
, [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
, [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
]
*///:~
```

Vežba 21: (3) Pokušajte da uredite niz objekata iz vežbe 18. Primenite **Comparable** da biste rešili problem. Potom napravite **Comparator** za sortiranje objekata obrnutim redosledom.

Uređivanje niza

Pomoću ugrađenih metoda možete da uredite proizvoljan niz elemenata prostog tipa, odnosno bilo koji niz objekata koji realizuju **Comparable** ili imaju pridruženi **Comparator**.[3] Evo primera kojim se generišu, a potom i sortiraju, slučajni objekti tipa **String**:

```
//: nizovi/UredjivanjeTipaString.java
// Uredjivanje niza elemenata tipa String.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class UredjivanjeTipaString{
  public static void main(String[] args) {
    String[] sa = Generisani.niz(new String[20],
      new GeneratorSlucajnih.String(5));
      print("Pre uređivanja: ", Arrays.toString(sa));
     Arrays.sort(sa);
     Arrays2.print("Nakon uređivanja: ", Arrays.toString(sa));
     Arrays.sort(sa, Collections.reverseOrder());
     print("Uređivanje obrnutim redom: " + Arrays.toString(sa));
     Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
     print("Uređivanje bez obzira na veličinu slova: " +
Arrays.toString(sa));
  }
} /* Ispis:
Pre uređivanja: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMRoE, suEcU, OneOE,
dLsmw, HLGEa, hKcxr, EqUCB, bkIna, Mesbt, WHkjU, rUkZP, gwsqP, zDyCy,
RFJQA, HxxHv]
Nakon uređivanja: [EqUCB, HLGEa, HxxHv, JMRoE, Mesbt, OWZnT, OneOE,
RFJQA, WHkjU, YNzbr, bkIna, cQrGs, dLsmw, eGZMm, gwsqP, hKcxr, nyGcF,
rUkZP, suEcU, zDyCy]
Uređivanje obrnutim redom: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP,
eGZMm, dLsmw, cQrGs, bkIna, YNzbr, WHkjU, RFJQA, OneOE, OWZnT, Mesbt,
JMRoE, HxxHv, HLGEa, EqUCB]
Uređivanje bez obzira na veličinu slova: [bkIna, cQrGs, dLsmw, eGZMm,
EqUCB, gwsqP, hKcxr, HLGEa, HxxHv, JMRoE, Mesbt, nyGcF, OneOE, OWZnT,
RFJQA, rUkZP, suEcU, WHkjU, YNzbr, zDyCy]
*///:~
```

Verovatno ćete primetiti da je rezultat ovog algoritma za uređivanje niza elemenata tipa **String** *leksikografski*, odnosno na početak postavlja reči koje počinju velikim slovom, a potom sledе reči koje počinju malim slovom. (Ovako se obično uređuju stavke u telefonskim imenicima.) Ako hoćete da grupišete reči abecedno, tj. bez obzira na to da li počinju velikim ili malim slovom, upotrebite **String.CASE_INSENSITIVE_ORDER** kao u poslednjem pozivu metode **sort()** u gornjem primeru.

[3] Začudo, u Javi 1.0 i 1.1 nije postojala podrška za uređivanje objekata tipa **String**.

Algoritam uređivanja koji se koristi u Javinoj standardnoj biblioteci optimalan je za odredeni tip koji se uređuje: Quicksort za proste tipove, odnosno stabilan MergeSort za objekte. Dakle, ne morate da brinete o učinku, osim ako vam alatka za merenje performansi ne ukaže na to da je postupak uređivanja usko grlo.

Pretraživanje uređenog niza

Kada se niz uredi, određeni element se može brzo potražiti pomoću metode **Arrays.binarySearch()**. Međutim, veoma je važno da ne pokušate da primenite metodu **binarySearch()** na niz koji nije uređen, jer se rezultati toga ne mogu predvideti. U sledećem primeru, klasa **GeneratorSlucajnih** se koristi prvo za popunjavanje niza, a zatim i za dobijanje vrednosti koja će se tražiti:

```
//: nizovi/PretrazivanjeNiza.java
// Koriscenje Arrays.binarySearch().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PretrazivanjeNiza {
  public static void main(String[] args) {
    Generator<Integer> gen =
      new GeneratorSlucajnih.Integer(1000);
    int[] a = KonvertujU.prost(
      Generisani.niz(new Integer[25], gen));
    Arrays.sort(a);
    print("Uređen niz: " + Arrays.toString(a));
    while(true) {
      int r = gen.next();
      int pozicija = Arrays.binarySearch(a, r);
      if(pozicija >= 0) {
        print("Pozicija od " + r + " jeste " + pozicija +
          ", a[" + pozicija + "] = " + a[pozicija]);
        break; // Izlazak iz petlje while
      }
    }
  }
} /* Ispis:
Uređen niz: [128, 140, 200, 207, 258, 258, 278, 288, 322, 429, 511, 520, 522,
551, 555, 589, 693, 704, 809, 861, 861, 868, 916, 961, 998]
Pozicija od 322 jeste 8, a[8] = 322
*///:~
```

U petlji **while** generišu se slučajne vrednosti za elemente koji se traže, sve dok se jedna vrednost ne pronađe.

Metoda **Arrays.binarySearch()** vraća vrednost koja je veća od nule ili jednaka nuli ako se pronađe traženi element. U suprotnom, vraća negativnu vrednost koja predstavlja mesto na koje bi taj element trebalo da se umetne kada bi se redosled niza ručno održavao. Vraćena vrednost je:

```
-(tačka umetanja) - 1
```

Tačka umetanja je indeks prvog elementa koji je veći od traženog, odnosno **a.size()** ako su svi elementi niza manji od traženog.

Ako niz sadrži više elemenata iste vrednosti, ne zna se koji će od njih biti pronađen. To znači da algoritam ne podržava duplikate elemenata, već ih toleriše. Ako vam je potrebna uređena lista bez ponovljenih elemenata, upotrebite klasu **TreeSet** (koja održava uređeni redosled) ili **LinkedHashSet** (koja održava redosled umetanja). Ove klase se umesto vas automatski staraju o svim pojedinostima. Samo u slučajevima kada vam je neophodno poboljšanje performansi trebalo bi zameniti, jednu od tih klasa nizom čiji se redosled održava ručno.

Ako ste za uređivanje niza objekata koristili komparator (nizovi prostih tipova se ne mogu uređivati pomoću komparatora), morate da upotrebite isti **Comparator** kada pretražujete metodom **binarySearch()** (pomoću preklopljene verzije ove funkcije). Na primer, program **UređivanjeTipaString.java** može se prilagoditi tako da pretražuje niz:

```
//: nizovi/AbecednaPretraga.java
// Pretraživanje pomoću komparatora.
import java.util.*;
import net.mindview.util.*;

public class AbecednaPretraga {
  public static void main(String[] args) {
    String[] sa = Generisani.niz(new String[30],
      new GeneratorSlucajnih.String(5));
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    System.out.println(Arrays.toString(sa));
    int indeks = Arrays.binarySearch(sa, sa[10],
      String.CASE_INSENSITIVE_ORDER);
    System.out.println("Indeks: "+ indeks + "\n"+ sa[indeks]);
  }
} /* Ispis:
[bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr, HLGEa, HqXum, HxxHv,
JMRoE, JmzMs, Mesbt, MNvqe, nyGcF, ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia,
sljrL, suEcU, uTpnX, vpfFv, WHkjU, xxEAJ, YNzbr, zDyCy]
Indeks: 10
HxxHv
*///:~
```

Comparator se mora proslediti preklopljenoj metodi **binarySearch()** kao treći argument. U gornjem primeru će element koji se traži sigurno biti pronađen jer je dobijen iz samog niza.

Vežba 22: (2) Pokažite da su rezultati primene metode **binarySearch()** na neuređen niz nepredvidljivi.

Vežba 23: (2) Napravite niz objekata tipa **Integer**, popunite ga nasumičnim **int** vrednostima (primenite automatsko pakovanje) i uredite ga obrnutim redosledom pomoću **Comparatora**.

Vežba 24: (3) Pokažite da se klasa iz vežbe 19 može pretraživati.

Sažetak

U ovom poglavlju, videli ste da Java pruža pristojnu podršku za nizove nepromenljive veličine i niskog nivoa. Takvi nizovi imaju dobre performanse, ali ne i fleksibilnost kao u jezicima C i C++. U prvoj verziji Jave, nizovi nepromenljive veličine i niskog nivoa bili su apsolutno neophodni, ne samo zato što su projektanti Jave (i zbog performansi) odlučili da Java ima i proste tipove, nego i zato što je podrška za kontejnere u toj verziji bila minimalna. Dakle, u ranim verzijama Jave, uvek je bilo preporučljivo koristiti nizove.

U kasnijim verzijama Jave, znatno se poboljšala podrška za kontejnere, pa oni sada zasenjuju nizove po svemu sem po performansama, a i performanse kontejnera su u međuvremenu znatno poboljšane. Već smo na više mesta rekli da probleme s performansama obično ne prouzrokuje ono na šta programer sumnja.

Nakon uvođenja automatskog pakovanja i generičkih tipova, držanje prostih tipova u kontejnerima postalo je lako, pa je i to razlog da nizove zamenite kontejnerima. Nizovi više nemaju ni tu isključivu prednost da omogućuju bezbedan rad s tipovima, pošto i generički tipovi daju kontejnere čiji se tipovi automatski proveravaju.

Rekosmo u ovom poglavlju da su generički tipovi prilično nepodesni za rad s nizovima, a i sami ćete se uveriti kada pokušate da ih upotrebite. Često se dešava da u vreme prevođenja dobijate upozorenja unchecked čak i kada postignete da generički tipovi i nizovi na neki način sarađuju (kao što ćete videti u narednom poglavlju).

Kada se raspravljalo o konkretnim primerima, projektanti jezika Java u više navrata su mi govorili da bi umesto nizova trebalo da koristim kontejnere (pomoću nizova sam pokazivao specifične tehnike, pa tu mogućnost nisam imao).

Sve navedeno pokazuje da bi „kontejnerima trebalo da date prednost nad nizovima" kada programirate u novijim verzijama Jave. Tek kada se dokaže da su performanse problem (i da će ih prelazak na nizove znatno popraviti), trebalo bi da ponovo „podelite program na proste faktore" i pređete na nizove.

Ovo je prilično hrabra tvrdnja, ali neki jezici uopšte nemaju nizove niskog nivoa i nepromenljive veličine. Imaju samo kontejnere promenljive veličine koji su mnogo funkcionalniji od Javinih nizova. Primera radi, Python[4] ima tip **list** čija sintaksa liči na onu elementarnih nizova, ali mnogo funkcionalniji – može se čak i naslediti:

```
#: nizovi/PythonoveListe.py

aLista = [1, 2, 3, 4, 5]
print type(aLista) # <type 'lista'>
print aLista # [1, 2, 3, 4, 5]
print aLista[4] # 5   Indeksiranje liste
aLista.append(6) # Veličina lista je promenljiva
aLista += [7, 8] # Dodavanje jedne liste drugoj
print aLista # [1, 2, 3, 4, 5, 6, 7, 8]
deoAListe = aLista[2:4]
print deoAListe # [3, 4]

class MojaLista(list): # Nasleđivanje liste
```

[4] Videti *www.Python.org*.

```
        # Definicija metode, pokazivač 'this' se piše eksplicitno:
        def dajObrnutu(self):
            obrnuta = self[:] # Kopiranje liste pomoću delova
            obrnuta.reverse() # Ugrađena metoda klase list
            return obrnuta

lista2 = MojaLista(aLista) # Za pravljenje objekta nije potrebno 'new'
print type(lista2) # <klasa '__main__.MojaLista'>
print lista2.dajObrnutu() # [8, 7, 6, 5, 4, 3, 2, 1]
#:~
```

Osnove Pythonove sintakse date su u prethodnom poglavlju. Ovde smo napravili listu navodeći unutar uglastih zagrada niz objekata radvojenih zarezima. Rezultat je objekat tipa **list** u vreme izvršavanja (izlaz naredbe **print** prikazan je u istom redu, u obliku komentara). Ispisivanje **liste** daje isti rezultat kao metoda **Arrays.toString()** u Javi.

Pravljenje podniza **liste** postižemo „deljenjem", tako što operator **:** smestimo unutar operacije indeksiranja. Tip **list** ima još mnogo ugrađenih operacija.

MojaLista je definicija klase; osnovne klase se pišu u zagradama. Unutar klase, naredbe **def** proizvode metode, a prvi argument metode automatski postaje ekvivalentan rezervisanoj reči **this** u Javi, samo što u Pythonu on mora biti eksplicitno naveden, a identifikator **self** se koristi po konvenciji (nije rezervisana reč). Obratite pažnju na automatsko nasleđivanje konstruktora.

Iako u Pythonu zapravo postoje *samo* objekti (među kojima i tipovi za cele brojeve i brojeve s pokretnim zarezom), izlaz u nuždi za optimizovanje performansi kritičnih delova programa predstavlja pisanje dodataka na jezicima C, C++ ili pomoću Pyrexa, specijalne alatke koja olakšava ubrzavanje koda. Na taj način možete ostvariti i čistoću objekata i poboljšane performanse.

Jezik PHP[5] ide još dalje u tom smeru, jer ima samo jedan tip niza koji radi i kao niz čiji su indeksi celi brojevi i kao asocijativan niz (mapa).

Zanimljivo je nagađati sada, posle toliko godina evolucije Jave, da li bi projektanti opet stavili u jezik proste tipove i nizove niskog nivoa da sve počinju iz početka. Da je to bilo izostavljeno, mogao se napraviti zaista čist objektno orijentisan jezik (uprkos takvim tvrdnjama, Java to nije, upravo zbog elemenata niskog nivoa). Zahtev da se efikasno radi uvek izgleda nezaobilazan, ali s godinama smo videli evoluciju od te ideje ka upotrebi komponenata višeg nivoa kao što su kontejneri. Dodajte tome sledeću činjenicu: ukoliko su kontejneri ugrađeni u osnovu jezika (kao što jesu u nekim jezicima), onda prevodilac ima mnogo bolju priliku da optimizuje.

Bilo kako bilo, od nizova ne možemo pobeći; nailazićete na njih u kodu koji čitate. Međutim, kontejneri su gotovo uvek bolje rešenje.

Vežba 25: (3) Napišite **PythonoveListe.py** u Javi.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

[5] Videti *www.php.net*.

Detaljno razmatranje kontejnera

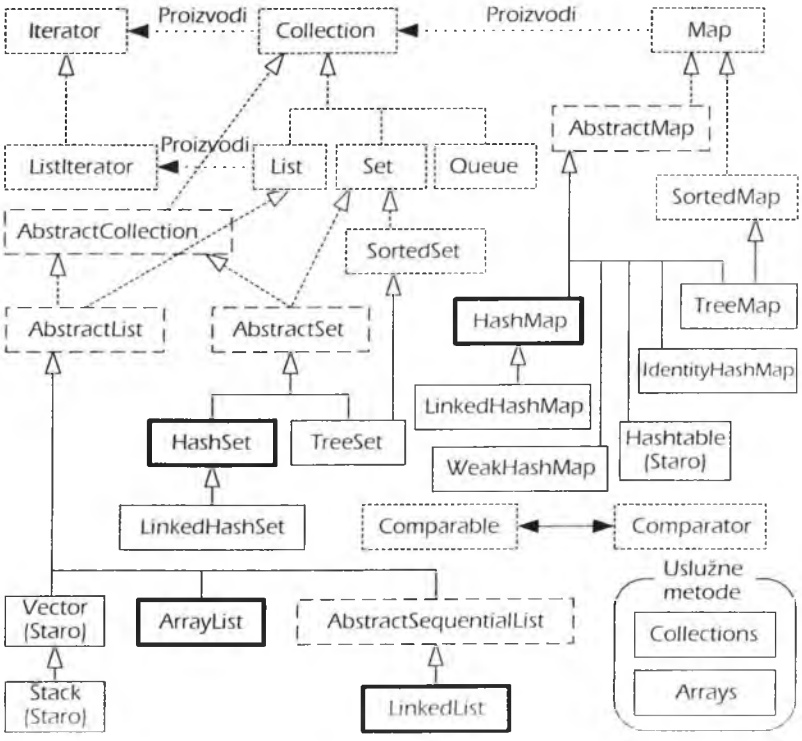
U poglavlju Čuvanje objekata *predstavili smo Javinu biblioteku kontejnera i njenu osnovnu funkcionalnost, što je dovoljno za početak rada s njima. U ovom poglavlju iscrpnije ćemo istražiti ovu važnu biblioteku.*

DA BISTE BIBLIOTEKU S KONTEJNERIMA MOGLI DA KORISTITE U POTPUNOSTI, MORATE ZNATI više nego što sadrži poglavlje *Čuvanje objekata*. Pošto je ovo poglavlje zasnovano na naprednijem gradivu (kao što su generički tipovi), odložili smo ga do ovog mesta u knjizi.

Nakon što damo potpuniji pregled kontejnera, saznaćete kako radi transformisanje ključa (engl. *hashing*) i kako treba pisati metode **hashCode( )** i **equals( )** da bi radile s kontejnerima transformisanih ključeva. Naučićete zašto postoje različite verzije nekih kontejnera i kada koji treba upotrebiti. Poglavlje ćemo završiti razmatranjem uslužnih metoda opšte namene i specijalnih klasa.

Potpuna taksonomija kontejnera

U „Sažetku" poglavlja *Čuvanje objekata* prikazan je pojednostavljen dijagram Javine biblioteke kontejnera. Ovo je njen potpuniji dijagram koji obuhvata apstraktne klase i stare komponente (sem realizacija interfejsa **Queue**):



Potpuna taksonomija kontejnera

U Javu SE5 je dodato sledeće:

- Interfejs **Queue** (klasa **LinkedList** je izmenjena tako da ga realizuje, kao što ste videli u poglavlju *Čuvanje objekata*) i njegove realizacije **PriorityQueue** i razni podtipovi klase **BlockingQueue** koje ćemo razmotriti u poglavlju *Paralelno izvršavanje.*
- Interfejs **ConcurrentMap** i njegova realizacija **ConcurrentHashMap**, takođe za upotrebu u višenitnom radu i to je prikazano u poglavlju *Paralelno izvršavanje.*
- **CopyOnWriteArrayList** i **CopyOnWriteArraySet**, takođe za paralelno izvršavanje.
- **EnumSet** i **EnumMap**, specijalne realizacije klasa **Set** odnosno **Map** za upotrebu s nabrojanim tipovima (**enum**); razmotrene su u poglavlju *Nabrojani tipovi.*
- Više uslužnih metoda u klasi **Collections.**

Pravougaonici oivičeni isprekidano (ali ne najkraćim crticama) predstavljaju **apstraktne** klase. Vidi se da imena više klasa počinju sa **Abstract.** Isprva to može da zbuni, ali radi se o alatkama koje delimično realizuju određeni interfejs. Da pravite sopstveni skup, na primer, ne biste počeli od interfejsa **Set** i realizovali sve njegove metode; valjda biste nasledili klasu **AbstractSet** i napisali minimum onoga što je potrebno za pravljenje nove klase. Međutim, biblioteka kontejnera sadrži gotovo svu funkcionalnost koja vam ikada može zatrebati, pa najčešće možete zanemariti sve klase čija imena počinju sa **Abstract.**

Popunjavanje kontejnera

Mada je problem sa štampanjem sadržaja kontejnera rešen, popunjavanje kontejnera ima isti nedostatak kao **java.util.Arrays**. Kao u paketu **Arrays**, postoji prateća klasa nazvana **Collections** koja sadrži statične uslužne metode, među kojima i jednu nazvanu **fill ( )**. Kao u verziji paketa **Arrays**, ta metoda **fill ( )** po celom kontejneru samo duplira jednu referencu na objekat. Sem toga, ona radi samo za **List** objekte, ali se proizvedena lista može proslediti konstruktoru ili nekoj metodi **addAll( )**:

```
//: kontejneri/PopunjavanjeLista.java
// Metode Collections.fill() i Collections.nCopies().
import java.util.*;

class AdresaStringa {
  private String s;
  public AdresaStringa(String s) { this.s = s; }
  public String toString() {
    return super.toString() + " " + s;
  }
}

public class PopunjavanjeLista {
  public static void main(String[] args) {
    List<AdresaStringa> list= new ArrayList<AdresaStringa>(
      Collections.nCopies(4, new AdresaStringa("Zdravo")));
    System.out.println(list);
    Collections.fill(list, new AdresaStringa("svima!"));
```

```
    System.out.println(list);
  }
} /* Ispis: (primer)
[AdresaStringa@82ba41 Zdravo, AdresaStringa@82ba41 Zdravo,
AdresaStringa@82ba41 Zdravo, AdresaStringa@82ba41 Zdravo]
[AdresaStringa@923e30 svima!, AdresaStringa@923e30 svima!,
AdresaStringa@923e30 svima!, AdresaStringa@923e30 svima!]
*///:~
```

U primeru su prikazana dva načina popunjavanja kontejnera (objekta tipa **Collection**) referencama na jedan objekat. Prvo, metoda **Collections.nCopies( )** pravi listu koja se prosleđuje konstruktoru; on popunjava **ArrayList**.

Metoda **toString( )** klase **AdresaStringa** poziva metodu **Object.toString( )** koja daje ime klase i heksadecimalno prikazan ključ za heširanje (engl. *hash code*) tog objekta – generisan metodom **hashCode( )**. Iz rezultata vidite da sve reference upućuju na isti objekat, a isto važi i nakon poziva druge metode, **Collections.fill( )**. Metodu **fill( )** još manje korisnom čini to što može da zameni samo elemente koji su već u listi – ona ne može da dodaje nove elemente.

Generatorsko rešenje

Gotovo svi podtipovi interfejsa **Collection** imaju konstruktor koji prima drugi objekat tipa **Collection**, iz kojega može da popuni nov kontejner. Dakle, da bismo sačinili podatke za testiranje, samo treba da napravimo klasu koja kao argumente prima konstruktore nekog **Generator**a (oni su definisani u poglavlju *Generički tipovi* i pobliže objašenjeni u poglavlju *Nizovi*) i neki broj **kolicina**:

```
//: net/mindview/util/PodaciKontejnera.java
// Kontejner koji podacima popunjava generator.
package net.mindview.util;
import java.util.*;

public class PodaciKontejnera<T> extends ArrayList<T> {
  public PodaciKontejnera(Generator<T> gen, int kolicina) {
    for(int i = 0; i < kolicina; i++)
      add(gen.next());
  }
  // Generička pomoćna metoda:
  public static <T> PodaciKontejnera<T>
  list(Generator<T> gen, int kolicina) {
    return new PodaciKontejnera<T>(gen, kolicina);
  }
} ///:~
```

Ovde **Generator** puni kontejner željenim brojem objekata. Dobijeni kontejner se može proslediti konstruktoru bilo kojeg drugog kontejnera koji će podatke kopirati u sebe. Za popunjavanje postojećeg kontejnera može se upotrebiti i metoda **addAll( )** koju imaju svi podtipovi klase **Collection**.

Kada se koristi pomoćna generička metoda, smanjuje se količina koda potrebnog za upotrebu klase.

Program **PodaciKontejnera** je primer projektnog obrasca *Adapter* (Adapter);[1] on prilagođava određeni **Generator** konstruktoru nekog kontejnera, tj. podtipa klase **Collection**.

Evo primera u kojem se inicijalizuje objekat tipa **LinkedHashSet**:

```
//: kontejneri/PodaciZaTestKontejnera.java
import java.util.*;
import net.mindview.util.*;

class Vlada implements Generator<String> {
  String[] foundation = ("strange women lying in ponds " +
    "distributing swords is no basis for a system of " +
    "government").split(" ");
  private int indeks;
  public String next() { return foundation[indeks++]; }
}

public class PodaciZaTestKontejnera {
  public static void main(String[] args) {
    Set<String> set = new LinkedHashSet<String>(
      new PodaciKontejnera<String>(new Vlada(), 15));
    // Korišćenje pomoćne metode:
    set.addAll(PodaciKontejnera.list(new Vlada(), 15));
    System.out.println(set);
  }
} /* Ispis:
[strange, women, lying, in, ponds, distributing, swords, is, no,
basis, for, a, system, of, government]
*///:~
```

Elementi su u poretku kojim su bili umetani, pošto **LinkedHashSet** održava ulančanu listu koja zadržava redosled umetanja.

Svi generatori definisani u poglavlju *Nizovi* sada su dostupni preko adaptera **PodaciKontejnera**. U ovom primeru upotrebićemo dva od njih:

```
//: kontejneri/GenerisanjePodatakaKontejnera.java
// Korišćenje generatora definisanih u poglavlju Nizovi.
import java.util.*;
import net.mindview.util.*;

public class GenerisanjePodatakaKontejnera {
  public static void main(String[] args) {
    System.out.println(new ArrayList<String>(
      PodaciKontejnera.list( // Pomoćna metoda
        new RandomGenerator.String(9), 10)));
```

[1] Ovo nije stroga definicija adaptera kao ona iz knjige *Projektni obrasci*, ali smatram da je dovoljno precizna.

```
    System.out.println(new HashSet<Integer>(
      new PodaciKontejnera<Integer>(
        new RandomGenerator.Integer(), 10)));
  }
} /* Ispis:
[YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw, HLGEahKcx,
rEqUCBbkI, naMesbtWH, kjUrUkZPg, wsqPzDyCy]
[573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
*///:~
```

Dužinu objekta tipa **String** koji pravi **RandomGenerator.String** određuje konstruktor u argumentu.

Generatori **mapa**

Na isti način možemo napraviti mapu, ali za nju nam treba klasa **Par**, pošto se za svaki poziv **Generatorove** metode **next( )** mora napraviti par objekata (jedan ključ i jedna vrednost) za popunjavanje mape:

```
//: net/mindview/util/Par.java
package net.mindview.util;

public class Par<K,V> {
  public final K kljuc;
  public final V vrednost;
  public Par(K k, V v) {
    kljuc = k;
    vrednost = v;
  }
} ///:~
```

Polja **kljuc** i **vrednost** su javna i finalna, pa **Par** postaje *Objekat za prenos podataka* (ili *Prenosilac*).

Ovaj adapter interfejsa **Map** sada može da popunjava objekte za inicijalizaciju mape raznim kombinacijama **Generatora**, objekata tipa **Iterable** i konstanti:

```
//: net/mindview/util/PodaciIzMape.java
// Mapa koju podacima popunjava generatorski objekat.
package net.mindview.util;
import java.util.*;

public class PodaciIzMape<K,V> extends LinkedHashMap<K,V> {
  // Jedan Generator za Par:
  public PodaciIzMape(Generator<Par<K,V>> gen, int kolicina) {
    for(int i = 0; i < kolicina; i++) {
      Par<K,V> p = gen.next();
      put(p.kljuc, p.vrednost);
    }
```

```
  }
  // Dva zasebna Generatora:
  public PodaciIzMape(Generator<K> genK, Generator<V> genV,
      int kolicina) {
    for(int i = 0; i < kolicina; i++) {
      put(genK.next(), genV.next());
    }
  }
  // Generator za ključ i jedna vrednost:
  public PodaciIzMape(Generator<K> genK, V vrednost, int kolicina){
    for(int i = 0; i < kolicina; i++) {
      put(genK.next(), vrednost);
    }
  }
  // Objekat tipa Iterable i Generator vrednosti:
  public PodaciIzMape(Iterable<K> genK, Generator<V> genV) {
    for(K kljuc : genK) {
      put(kljuc, genV.next());
    }
  }
  // Objekat tipa Iterable i jedna vrednost:
  public PodaciIzMape(Iterable<K> genK, V vrednost) {
    for(K kljuc : genK) {
      put(kljuc, vrednost);
    }
  }
  // Generičke pomoćne metode:
  public static <K,V> PodaciIzMape<K,V>
  map(Generator<Par<K,V>> gen, int kolicina) {
    return new PodaciIzMape<K,V>(gen, kolicina);
  }
  public static <K,V> PodaciIzMape<K,V>
  map(Generator<K> genK, Generator<V> genV, int kolicina) {
    return new PodaciIzMape<K,V>(genK, genV, kolicina);
  }
  public static <K,V> PodaciIzMape<K,V>
  map(Generator<K> genK, V vrednost, int kolicina) {
    return new PodaciIzMape<K,V>(genK, vrednost, kolicina);
  }
  public static <K,V> PodaciIzMape<K,V>
  map(Iterable<K> genK, Generator<V> genV) {
    return new PodaciIzMape<K,V>(genK, genV);
  }
  public static <K,V> PodaciIzMape<K,V>
  map(Iterable<K> genK, V vrednost) {
    return new PodaciIzMape<K,V>(genK, vrednost);
  }
} ///:~
```

Program pruža izbor – može se koristiti: jedan objekat tipa **Generator<Par<K,V>>**, dva zasebna **Generatora**, jedan **Generator** i jedna konstantna vrednost, jedan objekat tipa **Iterable** (koji obuhvata svaki objekat tipa **Collection**) i jedan **Generator**, ili objekat tipa **Iterable** i jedna vrednost. Pomoćne generičke metode smanjuju količinu koda potrebnu za pravljenje objekta tipa **PodaciIzMape**.

Evo primera korišćenja programa **PodaciIzMape**. I **Generator** klase **Slova** realizuje interfejs **Iterable** pravljenjem **Iteratora**; zato se može upotrebiti za testiranje metoda **PodaciIzMape.map( )** koje rade sa svim objektima koji realizuju **Iterable**:

```
//: kontejneri/TestiranjePodatakaMape.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Slova implements Generator<Par<Integer,String>>,
  Iterable<Integer> {
  private int velicina = 9;
  private int broj = 1;
  private char slovo = 'A';
  public Par<Integer,String> next() {
    return new Par<Integer,String>(
      broj++, "" + slovo++);
  }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      public Integer next() { return broj++; }
      public boolean hasNext() { return broj < velicina; }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}

public class TestiranjePodatakaMape {
  public static void main(String[] args) {
    // Generator para:
    print(PodaciIzMape.map(new Slova(), 11));
    // Dva zasebna generatora:
    print(PodaciIzMape.map(new CountingGenerator.Character(),
      new RandomGenerator.String(3), 8));
    // Generator ključa i jedna vrednost:
    print(PodaciIzMape.map(new CountingGenerator.Character(),
      "Vrednost", 6));
    // Objekat tipa Iterable i Generator vrednosti:
    print(PodaciIzMape.map(new Slova(),
      new RandomGenerator.String(3)));
```

```
    // Objekat tipa Iterable i jedna vrednost:
    print(PodaciIzMape.map(new Slova(), "Pop"));
  }
} /* Ispis:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
{a=YNz, b=brn, c=yGc, d=FOW, e=ZnT, f=cQr, g=Gse, h=GZM}
{a=Value, b=Value, c=Value, d=Value, e=Value, f=Value}
{1=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
{1=Pop, 2=Pop, 3=Pop, 4=Pop, 5=Pop, 6=Pop, 7=Pop, 8=Pop}
*///:~
```

I u ovom primeru smo koristili generatore iz poglavlja *Nizovi*.

Ovim alatkama možete da generišete proizvoljan skup podataka za **Mape** i kontejnere (objekte tipa **Collection**) i zatim da ih inicijalizujete konstruktorom ili metodama **Map.putAll( )** odnosno **Collection.addAll( )**.

Korišćenje **apstraktnih** klasa

Podaci za testiranje kontejnera mogu se napraviti i namenskim realizacijama klasa **Collection** i **Map**. Svaki kontejner iz paketa **java.util** ima svoju apstraktnu klasu koja delimično realizuje taj kontejner, a vama preostaje samo da realizujete potrebne metode za pravljenje željenog kontejnera. Ukoliko je dobijeni kontejner samo za čitanje, što je uobičajeno u slučaju podataka za testiranje, broj metoda koje morate da napravite je minimalan.

Iako to nije bilo neophodno, sledeće rešenje nam je dalo priliku da prikažemo još jedan projektni obrazac *Flyweight* (Zamajac). *Flyweight* se koristi kada je za uobičajeno rešenje neophodno previše objekata ili kada pravljenje normalnih objekata zauzima previše prostora. Projektni obrazac *Flyweight* eksternalizuje deo objekta tako što, umesto da se ceo objekat drži u samom objektu, deo objekta ili ceo objekat traži se u efikasnijoj, spoljnoj tabeli (ili pravi nekim drugim proračunom koji štedi prostor).

Važno je da uočite kako se namenski objekti tipa **Map** i **Collection** lako prave nasleđivanjem klasa **java.util.Abstract**. Da biste napravili objekat tipa **Map** samo za čitanje, nasleđujete klasu **AbstractMap** i realizujete metodu **entrySet( )**. Da biste napravili objekat tipa **Set** samo za čitanje, nasleđujete klasu **AbstractSet** i realizujete metode **iterator( )** i **size( )**.

Skup podataka u ovom primeru jeste mapa država i njihovih glavnih gradova.[2] Metoda **glavni_gradovi( )** pravi mapu država i glavnih gradova. Metoda **imena( )** pravi listu engleskih imena država. U oba slučaja dobićete delimični listing ukoliko željenu veličinu naznačite celobrojnim argumentom:

```
//: net/mindview/util/Countries.java
// "Flyweight" Mape i Liste podataka za primer.
package net.mindview.util;
import java.util.*;
import static net.mindview.util.Print.*;
```

[2] Ovi podaci su pronađeni na Internetu. Čitaoci su s vremenom slali razne ispravke.

```
public class Countries {
  public static final String[][] PODACI = {
    // Afrika
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"}, {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"BISSAU","Bissau"},
    {"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
    {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
    {"RWANDA","Kigali"},
    {"SAO TOME E PRINCIPE","Sao Tome"},
    {"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
    {"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
    {"SOUTH AFRICA","Pretoria/Cape Town"},
    {"SUDAN","Khartoum"},
    {"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
    {"TOGO","Lome"}, {"TUNISIA","Tunis"},
    {"UGANDA","Kampala"},
    {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
    {"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
    // Azija
    {"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
    {"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
    {"BRUNEI","Bandar Seri Begawan"},
    {"CAMBODIA","Phnom Penh"},
    {"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
    {"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
    {"IRAN","Tehran"}, {"IRAQ","Baghdad"},
    {"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
    {"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
    {"LAOS","Vientiane"}, {"LEBANON","Beirut"},
    {"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
    {"MONGOLIA","Ulan Bator"},
    {"MYANMAR (BURMA)","Rangoon"},
```

```
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"},
{"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"},
{"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
// Australjia i Okeanija
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Istočna Evropa i bivši SSSR
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"},
{"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Evropa
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA AND HERZEGOVINA","Sarajevo"},
{"BULGARIA","Sofia"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljubljana"}, {"SPAIN","Madrid"},
```

```
    {"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
    {"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
    // Severna i Centralna Amerika
    {"ANTIGUA AND BARBUDA","Saint John's"},
    {"BAHAMAS","Nassau"},
    {"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
    {"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
    {"CUBA","Havana"}, {"DOMINICA","Roseau"},
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"},
    {"GRENADA","Saint George's"},
    {"GUATEMALA","Guatemala City"},
    {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
    {"PANAMA","Panama City"}, {"ST. KITTS","-"},
    {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // Južna Amerika
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
  // Realizovaćemo metodu entrySet() da bismo koristili AbstractMap
  private static class FlyweightMapa
  extends AbstractMap<String,String> {
    private static class Stavka
    implements Map.Entry<String,String> {
      int indeks;
      Stavka(int indeks) { this.indeks = indeks; }
      public boolean equals(Object o) {
        return PODACI[indeks][0].equals(o);
      }
      public String getKey() { return PODACI[indeks][1]; }
      public String getValue() { return PODACI[indeks][1]; }
      public String setValue (String vrednost) {
        throw new UnsupportedOperationException();
      }
      public int hashCode() {
        return PODACI[indeks][0].hashCode();
      }
    }
```

```
    // Realizacijom metoda size() i iterator() koristimo
    // klasu AbstractSet
    static class SkupStavki
    extends AbstractSet<Map.Entry<String,String>> {
      private int velicina;
      EntrySet(int velicina) {
        if(velicina < 0)
          this.velicina = 0;
        // Ne može biti veća od niza:
        else if(velicina > PODACI.length)
          this.velicina = PODACI.length;
        else
          this.velicina = velicina;
      }
      public int size() { return velicina; }
      private class Iter
      implements Iterator<Map.Entry<String,String>> {
        // Samo jedan objekat tipa Stavka po Iteratoru:
        private Stavka stavka = new Stavka(-1);
        public boolean hasNext() {
          return stavka.indeks < velicina - 1;
        }
        public Map.Entry<String,String> next() {
          stavka.indeks++;
          return stavka;
        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
      }
      public
      Iterator<Map.Entry<String,String>> iterator() {
        return new Iter();
      }
    }
    private static Set<Map.Entry<String,String>> stavke =
      new EntrySet(PODACI.length);
    public Set<Map.Entry<String,String>> entrySet() {
      return stavke;
    }
  }
  // Pravljenje delimične mape država, date veličine:
  static Map<String,String> select(final int velicina) {
    return new FlyweightMapa() {
      public Set<Map.Entry<String,String>> entrySet() {
        return new EntrySet(velicina);
      }
    };
  }
```

```
  static Map<String,String> mapa = new FlyweightMapa();
  public static Map<String,String> glavni_gradovi() {
    return mapa; // Cela mapa
  }
  public static Map<String,String> glavni_gradovi(int velicina) {
    return select(velicina); // Delimična mapa
  }
  static List<String> imena =
  // Sva imena:
  public static List<String> imena() { return imena; }
  // Delimična lista:
  public static List<String> imena(int velicina) {
    return new ArrayList<String>(select(velicina).keySet());
  }
  public static void main(String[] args) {
    print(glavni_gradovi(10));
    print(imena(10));
       print HashMap<String,String>(glavni_gradovi(3)));
    print(new LinkedHashMap<String,String>(glavni_gradovi(3)));
    print(new TreeMap<String,String>(glavni_gradovi(3)));
    print(new Hashtable<String,String>(glavni_gradovi(3)));
    print(new HashSet<String>(imena(6)));
    print(new LinkedHashSet<String>(imena(6)));
    print(new TreeSet<String>(imena(6)));
    print(new ArrayList<String>(imena(6)));
    print(new LinkedList<String>(imena(6)));
    print(glavni_gradovi().get("BRAZIL"));
  }
} /* Ispis:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo, BOTSWANA=Gaberone,
BURKINA FASO=Ouagadougou, BURUNDI=Bujumbura, CAMEROON=Yaounde,
CAPE VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui, CHAD=N'djamena}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC, CHAD]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BURKINA FASO, BURUNDI, BOTSWANA, BENIN, ANGOLA, ALGERIA]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
Brasilia
*///:~
```

Dvodimenzionalni niz **PODACI** tipa **String** je javan, pa se može upotrebljavati i drugde. **FlyweightMapa** mora realizovati metodu **entrySet( )** koja zahteva namensku realizaciju i klase **Set** i klase **Map.Entry**. Uloga zamajca je u tome što svaki objekat tipa

Map.Entry jednostavno skladišti samo svoj indeks, a ne stvarni ključ i vrednost. Kada pozovete metodu **getKey( )** ili **getValue( )**, ona će vam vratiti odgovarajući element **PODACI** određen pomoću tog indeksa. Klasa **SkupStavki** se stara da njena **velicina** ne bude veća od niza **PODACI**.

Drugi deo zamajca realizovan je u programu **SkupStavki.Iterator**. Umesto da se za svaki par u nizu **PODACI** pravi objekat tipa **Map.Entry**, pravi se samo jedna stavka mape (**Map.Entry**) *za svaki iterator*. Objekat **Stavka** se upotrebljava kao prozor u podatke; on sadrži samo **indeks** statičnog niza znakovnih nizova. Svaki put kada pozovete metodu **next( )** za taj iterator, **indeks** u objektu **Stavka** povećava se za 1, tako da pokazuje na sledeći par elemenata, i potom se jedini objekat tipa **Stavka** tog **Iteratora** vraća iz metode **next( )**.[3]

Metoda **select( )** pravi objekat tipa **FlyweightMapa** koji sadrži **SkupStavki** željene veličine, a ovaj se koristi u metodama **glavni gradovi( )** i **imena( )** pokazanim u metodi **main( )**.

Za neka ispitivanja, ograničena veličina klase **Countries** predstavlja problem. Na isti način možemo napraviti inicijalizovane namenske kontejnere koji imaju skup podataka proizvoljne veličine. Sledeća klasa je lista proizvoljne veličine koju (zapravo) unapred inicijalizujemo **Integer** podacima:

```
//: net/mindview/util/BrojackaListaIntegera.java
// Lista proizvoljne dužine s probnim podacima.
package net.mindview.util;
import java.util.*;

public class BrojackaListaIntegera
extends AbstractList<Integer> {
  private int velicina;
  public BrojackaListaIntegera(int velicina) {
    this.velicina = velicina < 0 ? 0 : velicina;
  }
  public Integer get(int indeks) {
    return Integer.valueOf(indeks);
  }
  public int size() { return velicina; }
  public static void main(String[] args) {
    System.out.println(new BrojackaListaIntegera(30));
  }
} /* Ispis:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Morate realizovati metode **get( )** i **size( )** da biste od natklase **AbstractList** napravili listu samo za čitanje. Opet smo upotrebili rešenje sa zamajcem: **get( )** pravi vrednost kada je zatražite, pa lista zapravo ne mora da bude popunjena.

---

[3] **Mape** u paketu **java.util** prave grupne kopije metodama **getKey( )** i **getValue( )** za **Mape**, pa ovo radi. Kada bi namenska mapa samo kopirala celu **Map.Entry**, ovakav pristup bi prouzrokovao problem.

Evo jedne mape koja sadrži unapred inicijalizovane jedinstvene objekte tipa **Integer** i **String**; i ona može biti proizvoljne veličine:

```
//: net/mindview/util/BrojackaMapaPodataka.java
// Mapa neograničene dužine s probnim podacima.
package net.mindview.util;
import java.util.*;

public class BrojackaMapaPodataka
extends AbstractMap<Integer,String> {
  private int velicina;
  private static String[] znakovi =
    "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
    .split(" ");
  public BrojackaMapaPodataka(int velicina) {
    if(velicina < 0) this.velicina = 0;
    this.velicina = velicina;
  }
  private static class Stavka
  implements Map.Entry<Integer,String> {
    int indeks;
    Stavka(int indeks) { this.indeks = indeks; }
    public boolean equals(Object o) {
      return Integer.valueOf(indeks).equals(o);
    }
    public Integer getKey() { return indeks; }
    public String getValue() {
      return
        znakovi[indeks % znakovi.length] +
        Integer.toString(indeks / znakovi.length);
    }
    public String setValue(String vrednost) {
      throw new UnsupportedOperationException();
    }
    public int hashCode() {
      return Integer.valueOf(indeks).hashCode();
    }
  }
  public Set<Map.Entry<Integer,String>> entrySet() {
    // LinkedHashSet zadržava poredak ustanovljen prilikom
    // inicijalizacije:
    Set<Map.Entry<Integer,String>> stavke =
      new LinkedHashSet<Map.Entry<Integer,String>>();
    for(int i = 0; i < velicina; i++)
      stavke.add(new Stavka(i));
    return stavke;
  }
  public static void main(String[] args) {
    System.out.println(new BrojackaMapaPodataka(60));
  }
```

```
} /* Ispis:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0, 10=K0,
11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0, 17=R0, 18=S0, 19=T0, 20=U0,
21=V0, 22=W0, 23=X0, 24=Y0, 25=Z0, 26=A1, 27=B1, 28=C1, 29=D1, 30=E1,
31=F1, 32=G1, 33=H1, 34=I1, 35=J1, 36=K1, 37=L1, 38=M1, 39=N1, 40=O1,
41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1, 49=X1, 50=Y1,
51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2, 57=F2, 58=G2, 59=H2}
*///:~
```

Ovde se koristi **LinkedHashSet** umesto da smo napravili namenski podtip natklase **Set**, pa *Flyweight* nije potpuno realizovan.

Vežba 1: (1) Napravite listu (pokušajte da sačinite i **ArrayList** i **LinkedList**) i popunite je pomoću klase **Countries**. Listu uredite i ispišite, zatim na nju više puta primenite **Collections.shuffle( )** i svaki put je ispišite, da biste videli kako metoda **shuffle( )** svaki put drugačije ispremeša listu.

Vežba 2: (2) Napravite mapu i skup koji sadrže imena svih država koja počinju na A.

Vežba 3: (1) Pomoću klase **Countries**, više puta popunite neki objekat tipa **Set** istim podacima i dokažite da dobijeni skup (**Set**) ima samo po jedan primerak svakog objekta. Isprobajte to s klasama **HashSet**, **LinkedHashSet** i **TreeSet**.

Vežba 4: (2) Napravite inicijalizator kolekcije koji otvara datoteku i deli je na reči metodom **TextFile( )** (interfejsa **Collection**), a zatim te reči koristi kao izvor podataka za dobijeni kontejner. Pokažite da to radi.

Vežba 5: (3) Izmenite program **BrojackaMapaPodataka.java** tako da potpuno realizuje projektni obrazac *Flyweight* tako što dodaje namensku klasu **SkupStavki**, poput one u programu **Countries.java**.

Funkcije interfejsa **Collection**

U tabeli koja sledi prikazano je sve što možete da uradite s kolekcijama (nisu obuhvaćene metode koje se automatski nasleđuju iz klase **Object**), što znači sa skupom ili listom. (Interfejs **List** ima i dodatne funkcije.) Mape nisu izvedene iz interfejsa **Collection**, pa će biti razmotrene zasebno.

boolean add(T)	Obezbeđuje da argument generičkog tipa **T** bude u kontejneru. Vraća **false** ako ne doda argument. (Ovo je opciona metoda, opisana u sledećem odeljku.)
boolean addAll(Kolekcija<? extends T>)	Dodaje kolekciji sve elemente iz argumenta. Vraća **true** ako je dodat neki element. (Opciona metoda.)
void clear( )	Uklanja sve elemente iz kontejnera. (Opciona metoda.)
boolean contains(T)	Vraća **true** ako kontejner sadrži argument.
boolean containsAll(Kolekcija<?>)	Vraća **true** ako kontejner sadrži sve elemente iz argumenta.
boolean isEmpty( )	Vraća **true** ako u kontejneru nema elemenata.

Iterator<T> iterator( )	Vraća **Iterator<T>** koji se može koristiti za kretanje kroz elemente kontejnera.
boolean remove(Object)	Ako se argument nalazi u kontejneru, uklanja se jedna instanca tog elementa. Vraća **true** ako je nešto uklonjeno. (Opciona metoda).
boolean removeAll(Kolekcija<?>)	Uklanja sve elemente koje sadrži argument. Vraća **true** ako je nešto uklonjeno. (Opciona metoda).
boolean retainAll(Kolekcija<?>)	Zadržava samo elemente koji se nalaze u argumentu (skupovni presek). Vraća **true** ako se nešto promenilo. (Opciona metoda).
int size( )	Vraća broj elemenata u kontejneru.
Object[] toArray( )	Vraća niz koji sadrži sve elemente kontejnera.
<T> T[] toArray(T[] a)	Vraća niz koji sadrži sve elemente kontejnera čiji je tip isti kao tip niza **a**, a ne prosto **Object**. (Niz se mora konvertovati u odgovarajući tip).

Imajte u vidu da ne postoji metoda **get( )** za izbor elementa nasumičnim pristupom. Razlog je to što interfejs **Collection** sadrži i **Set** koji održava svoj unutrašnji poredak (pa bi pretraživanje nasumičnim pristupom bilo besmisleno). Dakle, za pregled elemenata kolekcije morate upotrebiti iterator.

U narednom primeru prikazane su sve te metode. Iako one rade sa svim klasama koje realizuju interfejs **Collection**, kao „najmanji zajednički imenitelj" upotrebljen je **ArrayList**.

```
//: kontejneri/MetodeKolekcija.java
// Šta sve možete da uradite sa svim kolekcijama.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class MetodeKolekcija {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    c.addAll(Countries.imena(6));
    c.add("deset");
    c.add("jedanaest");
    print(c);
    // Pravljenje niza od liste:
    Object[] niz = c.toArray();
    // Pravljenje niza tipa String od liste:
    String[] str = c.toArray(new String[0]);
    // Pronalaženje najvećeg i najmanjeg elementa;
    // to podrazumeva različite metode, u zavisnosti
    // od načina realizacije interfejsa Comparable:
    print("Collections.max(c) = " + Collections.max(c));
    print("Collections.min(c) = " + Collections.min(c));
    // Dodavanje jedne kolekcije drugoj
    Collection<String> c2 = new ArrayList<String>();
    c2.addAll(Countries.imena(6));
    c.addAll(c2);
    print(c);
    c.remove(Countries.PODACI[0][0]);
```

```
    print(c);
    c.remove(Countries.PODACI[1][0]);
    print(c);
    // Uklanjanje svih komponenata koje se nalaze
    // u kolekciji – argumentu:
    c.removeAll(c2);
    print(c);
    c.addAll(c2);
    print(c);
    // Da li se određeni element nalazi u ovoj kolekciji?
    String vre = Countries.PODACI[3][0];
    print("c.contains(" + vre  + ") = " + c.contains(vre));
    // Da li je kolekcija u ovoj kolekciji?
    print("c.containsAll(c2) = " + c.containsAll(c2));
    Collection<String> c3 =
      ((List<String>)c).subList(3, 5);
    // Zadržavanje samo onih elemenata koji se nalaze
    // u skupovima c2 i c3 (presek skupova):
    c2.retainAll(c3);
    print(c2);
    // Odbacivanje svih elemenata iz skupa c2
    // koji se pojavljuju i u skupu c3:
    c2.removeAll(c3);
       print("c2.isEmpty() = " +  c2.isEmpty());
    c = new ArrayList<String>();
    c.addAll(Countries.imena(6));
    print(c);
    c.clear(); // Uklanjanje svih elemenata
    print("posle c.clear():" + c);
  }
} /* Ispis:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
deset, jedanaest]
Collections.max(c) = deset
Collections.min(c) = ALGERIA
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI, deset,
jedanaest, ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI, deset, jedanaest,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[BENIN, BOTSWANA, BURKINA FASO, BURUNDI, deset, jedanaest,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[deset, jedanaest]
[deset, jedanaest, ALGERIA, ANGOLA, BENIN, BOTSWANA,
BURKINA FASO, BURUNDI]
c.contains(BOTSWANA) = true
c.containsAll(c2) = true
[ANGOLA, BENIN]
c2.isEmpty() = true
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
posle c.clear():[]
*///:~
```

Prave se **ArrayList**e koje sadrže različite skupove podataka i bivaju svedene naviše na **Collection** objekte, pa je jasno da se sem interfejsa **Collection** ne upotrebljava ništa drugo. U metodi **main( )**, pomoću jednostavnih vežbi prikazane su sve metode klase **Collection**.

Naredni odeljci u ovom poglavlju opisuju različite realizacije interfejsa **List**, **Set** i **Map**. Zvezdicom (*) je uvek označena realizacija koju bi trebalo podrazumevano izabrati. Opisi starih klasa **Vector**, **Stack** i **Hashtable** dati su tek na kraju poglavlja – iako ne bi trebalo da ih sami upotrebljavate, sretaćete se s njima u starom kodu.

Opcione operacije

Metode koje obavljaju razne vrste dodavanja i uklanjanja jesu *opcione operacije* interfejsa **Collection**. To znači da klasa koja realizuje interfejs ne mora da sadrži funkcionalne definicije tih metoda.

To je veoma neobičan način definisanja interfejsa. Kao što ste videli, u objektno orijentisanom programiranju interfejs je vrsta ugovora. Pomoću njega se kaže: „Bez obzira na to kako odlučite da realizujete ovaj interfejs, jemčim da ovom objektu možete da šaljete ove poruke."[4] Ali opciona operacija krši to temeljno obećanje: ne samo da pozivanje ovih metoda *neće* uraditi ništa korisno, već će i izazvati izuzetak! Kao da smo izgubili bezbednost tipova u vreme prevođenja.

Nije sve baš tako strašno. U kolekciji, listi, skupu ili mapi, prevodilac vas još uvek ograničava na pozivanje isključivo onih metoda koje se nalaze u tom interfejsu. To je ipak bolje od dinamičkih jezika koji pružaju mogućnost pozivanja svake metode za svaki objekat, da bi se tek pri pokretanju programa ustanovilo da li pozvana metoda uopšte može da radi[5]. Osim toga, mnoge metode čiji je argument kolekcija, isključivo *čitaju* tu kolekciju: nijedna metoda za čitanje kolekcije *nije* opciona.

Zašto bi iko zadao da je metoda opciona? Ovakav pristup sprečava eksploziju broja interfejsa. Drugačiji načini projektovanja biblioteke kontejnera uvek se završe sa nerazmrsivom gomilom interfejsa koji opisuju sve varijacije glavne teme i zbog toga ih je teško savladati. Nije čak ni moguće obuhvatiti sve specijalne slučajeve u obliku interfejsa, pošto uvek mogu da se izmisle novi interfejsi. Pristupom „operacija koje nisu podržane" postiže se važna svrha Javine biblioteke kontejnera: kontejneri se lako savladavaju i koriste; operacije koje nisu podržane specijalan su slučaj koji se može naučiti i kasnije. Međutim, da bi takav pristup delovao, potrebno je sledeće:

1. Izuzetak tipa **UnsupportedOperationException** mora da se javlja retko, tj. u većini klasa trebalo bi da rade sve operacije, a samo u specijalnim slučajevima operacija ne bi trebalo da bude podržana. To važi u Javinoj biblioteci kontejnera pošto klase koje najčešće koristite (**ArrayList**, **LinkedList**, **HashSet** i **HashMap**), kao i druge konkretne realizacije, podržavaju sve operacije. Ovakav pristup omogućuje da napravite novu kolekciju, da pri tom ne obezbedite definicije svih metoda u interfejsu **Collection**, a da je ipak uklopite u postojeću biblioteku.

[4] Termin „interfejs" je ovde upotrebljen tako da obuhvata i formalnu rezervisanu reč **interface** i opštije značenje: „metode podržane u svim klasama i potklasama".

[5] Mada ovo zvuči čudno i možda beskorisno kada je opisano na taj način, videli ste, naročito u poglavlju *Podaci o tipu*, da takvo dinamičko ponašanje može biti veoma moćno.

2. Ako operacija nije podržana, obično postoji velika verovatnoća da će se izuzetak tipa **UnsupportedOperationException** pojaviti u vreme testiranja, a ne kada prodate program korisniku. Najzad, on ukazuje na programersku grešku: korišćenje realizacije na nepravilan način.

Imajte u vidu da se nepodržane operacije mogu otkriti tek u vreme izvršavanja, te stoga predstavljaju dinamičku proveru tipova. Ako ste koristili jezik sa statičnom proverom tipova kao što je C++, možda vam i Java izgleda kao jezik u kojem se tipovi proveravaju statično. Java svakako *ima* statičnu proveru tipova, ali i znatnu količinu dinamičke provere tipova, pa je teško reći da je tačno jednog ili drugog tipa. Kada postanete svesni toga, počećete da uočavate i druge slučajeve dinamičke provere tipova u Javi.

Nepodržane operacije

U Javi su čest izvor nepodržanih operacija kontejner i pripadna struktura podataka nepromenljive dužine. Takav kontejner (u gornjem primeru, listu) od niza proizvodi metoda **Arrays.asList( )**. S druge strane, *vi odlučujete* da li će bilo koji kontejner (uključujući tu i mape) generisati izuzetke **UnsupportedOperationException** zbog upotrebe nepromenljivih metoda u klasi **Collections**. Primer koji sledi sadrži oba slučaja:

```
//: kontejneri/Nepodrzane.java
// Nepodržane operacije u Javinim kontejnerima.
import java.util.*;

public class Nepodrzane {
  static void test(String prk, List<String> lista) {
    System.out.println("--- " + prk + " ---");
    Collection<String> c = lista;
    Collection<String> podLista = lista.podLista(1,8);
    // Kopija podliste:
    Collection<String> c2 = new ArrayList<String>(podLista);
    try { c.retainAll(c2); } catch(Exception e) {
      System.out.println("retainAll(): " + e);
    }
    try { c.removeAll(c2); } catch(Exception e) {
      System.out.println("removeAll(): " + e);
    }
    try { c.clear(); } catch(Exception e) {
      System.out.println("clear(): " + e);
    }
    try { c.add("X"); } catch(Exception e) {
      System.out.println("add(): " + e);
    }
    try { c.addAll(c2); } catch(Exception e) {
      System.out.println("addAll(): " + e);
    }
    try { c.remove("C"); } catch(Exception e) {
      System.out.println("remove(): " + e);
    }
```

```
    // Metoda List.set() menja vrednost, ali
    // ne menja veličinu strukture podataka:
    try {
      lista.set(0, "X");
    } catch(Exception e) {
      System.out.println("List.set(): " + e);
    }
  }
  public static void main(String[] args) {
    List<String> lista =
      Arrays.asList("A B C D E F G H I J K L".split(" "));
    test("Promenljiva kopija", new ArrayList<String>(lista));
    test("Arrays.asList()", lista);
    test("unmodifiableList()",
      Collections.unmodifiableList(
        new ArrayList<String>(lista)));
  }
} /* Ispis:
--- Promenljiva kopija ---
--- Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*///:~
```

Pošto metoda **Arrays.asList( )** pravi listu od niza odredene dužine, ima smisla da budu podržane samo one operacije koje ne menjaju *dužinu* niza. Svaka metoda koja bi promenila dužinu pripadne strukture podataka izazvala bi izuzetak **UnsupportedOperationException** koji ukazuje na poziv nepodržane metode (grešku programera).

Vodite računa o tome da svakoj kolekciji uvek možete da prosledite rezultat metode **Arrays.asList( )** kao konstruktorski argument (ili da pozovete metodu **addAll( )**, ili statičnu metodu **Collections.addAll( )**) da biste napravili pravi kontejner koji omogućuje upotrebu svih metoda – to se vidi iz prvog poziva metode **test( )** u funkciji **main( )**. Takav poziv proizvodi novu strukturu podataka promenljive dužine.

Nepromenljive metode klase **Collections** omotavaju kontejner posrednikom koji izaziva **UnsupportedOperationException** ako izvršite bilo koju operaciju koja na bilo koji način menja taj kontejner. Cilj korišćenja tih metoda jeste pravljenje konstantnog kontejnerskog objekta. Naveščemo kasnije celokupan spisak nepromenljivih metoda klase **Collections**.

Poslednji blok **try** u metodi **test( )** ispituje metodu **set( )** koja je deo klase **List**. To je zanimljivo, zato što vidite kako dobro dođe granularnost tehnike nepodržana operacija – dobijeni interfejs može da se razlikuje za jednu metodu između objekta koji vraća **Arrays.asList( )** i objekta koji vraća metoda **Collections.nepromenljivaLista( )**. **Arrays.asList( )** vraća listu fiksne dužine, dok **Collections.nepromenljivaLista( )** proizvodi listu koja se ne može menjati. Kao što vidite iz rezultata, možete *modifikovati* elemente liste koju vraća metoda **Arrays.asList( )**, jer to ne bi narušilo nepromenljivu dužinu te liste. S druge strane, jasno je da rezultat metode **unmodifiableList( )** ne bi trebalo da bude promenljiv ni na koji način. Da smo koristili interfejse, bila bi potrebna dva dodatna interfejsa, jedan s funkcionalnom metodom **set( )** i drugi bez nje. Za razne nepromenljive podtipove od **Collection** bili bi potrebni dodatni interfejsi.

U dokumentaciji za metodu koja koristi kolekciju, listu, skup ili mapu kao argument treba odrediti koje opcione metode moraju da budu realizovane.

Vežba 6: (2) Obratite pažnju na to da klasa **List** ima dodatne opcione operacije koje **Collection** ne obuhvata. Napišite verziju programa **Nepodrzane.java** za testiranje tih dodatnih opcionih operacija.

Funkcije **liste**

Kao što ste videli, prost tip **List** se prilično lako koristi. Iako najćešće koristite metodu **add( )** za umetanje objekata, metodu **get( )** za uzimanje objekata jedan po jedan i **iterator( )** za dobijanje iteratora niza, postoje i druge metode koje mogu da budu korisne.

Svaka od metoda u primeru koji sledi obavlja drugu grupu aktivnosti: ono što svaka lista može da uradi (**osnovniTest( )**), kretanje po listi pomoću iteratora (**kretanjeIteratorom( )**) u odnosu na izmenu objekata pomoću iteratora (**promenaIteratorom( )**), prikaz dejstva obrade listi (**vidljivTest( )**), i operacije dostupne samo u ulančanim listama (**LinkedList**):

```
//: kontejneri/Liste.java
// Šta se sve može raditi s listama.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Liste {
  private static boolean b;
  private static String s;
  private static int i;
  private static Iterator<String> it;
  private static ListIterator<String> lit;
  public static void osnovniTest(List<String> a) {
    a.add(1, "x"); // Dodavanje na lokaciju 1
    a.add("x"); // Dodavanje na kraj
    // Dodavanje kolekcije:
    a.addAll(Countries.imena(25));
```

```
    // Dodavanje kolekcije počev od lokacije 3:
    a.addAll(3, Countries.imena(25));
    b = a.contains("1"); // Da li je tu?
    // Da li je cela kolekcija tu?
    b = a.containsAll(Countries.imena(25));
    // Liste omogućuju nasumičan pristup, što je brzo
    // kod ArrayList, sporo kod LinkedList:
    s = a.get(1); // Uzmi objekat (određenog tipa) na lokaciji 1
    i = a.indexOf("1"); // Koji je indeks objekta?
    b = a.isEmpty(); // Da li ima elemenata?
    it = a.iterator(); // Običan Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Počni od lokacije 3
    i = a.lastIndexOf("1"); // Poslednje poklapanje
    a.remove(1); // Ukloni element na lokaciji 1
    a.remove("3"); // Ukloni ovaj objekat
    a.set(1, "y"); // Postavi lokaciju 1 na "y"
    // Zadrži sve što je u argumentu
    // (presek dva skupa):
    a.retainAll(Countries.imena(25));
    // Obriši sve što se nalazi u argumentu:
    a.removeAll(Countries.imena(25));
    i = a.size(); // Kolika je lista?
    a.clear(); // Obriši sve elemente
  }
  public static void kretanjeIteratorom (List<String> a) {
    ListIterator<String> it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    s = it.next();
    i = it.nextIndex();
    s = it.previous();
    i = it.previousIndex();
  }
  public static void promenaIteratorom(List<String> a) {
    ListIterator<String> it = a.listIterator();
    it.add("47");
    // Mora se pomeriti na element posle  add():
    it.next();
    // Obriši element iza upravo napravljenog:
    it.remove();
    // Mora se pomeriti na element posle  remove():
    it.next();
    // Promeni element iza obrisanog:
    it.set("47");
  }
  public static void vidljivTest(List<String> a) {
    print(a);
```

```
    List<String> b = Countries.imena(25);
    print("b = " + b);
    a.addAll(b);
    a.addAll(b);
    print(a);
    // Umetanje, uklanjanje i zamena elemenata
    // pomoću ListIteratora:
    ListIterator<String> x = a.listIterator(a.size()/2);
    x.add("jedan");
    print(a);
    print(x.next());
    x.remove();
    print(x.next());
    x.set("47");
    print(a);
    // Kretanje kroz listu unazad:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      printnb(x.previous() + " ");
    print();
    print("metoda vidljivTest gotova");
  }
  // Postoje operacije koje mogu obaviti
  // samo ulančane liste:
  public static void testirajUlancanu() {
    LinkedList<String> ll = new LinkedList<String>();
    ll.addAll(Countries.imena(25));
    print(ll);
    // Lista kao stek, stavljanje na stek:
    ll.addFirst("jedan");
    ll.addFirst("dva");
    print(ll);
    // "Zavirivanje" na vrh steka:
    print(ll.getFirst());
    // Kao skidanje sa steka:
    print(ll.removeFirst());
    print(ll.removeFirst());
    // Lista kao red za čekanje,
    // izvlačenje elemenata iz "repa":
    print(ll.removeLast());
    print(ll);
  }
  public static void main(String[] args) {
    // Pravljenje i popunjavanje nove liste svaki put:
    osnovniTest(
      new LinkedList<String>(Countries.imena(25)));
    osnovniTest(
      new ArrayList<String>(Countries.imena(25)));
```

```
      kretanjeIteratorom(
        new LinkedList<String>(Countries.imena(25)));
      kretanjeIteratorom(
        new ArrayList<String>(Countries.imena(25)));
      promenaIteratorom(
        new LinkedList<String>(Countries.imena(25)));
      promenaIteratorom(
        new ArrayList<String>(Countries.imena(25)));
      vidljivTest(
        new LinkedList<String>(Countries.imena(25)));
      testirajUlancanu();
    }
  } /* (Pokrenite da biste videli rezultat)
  *///:~
```

Metode **osnovniTest( )** i **kretanjeIteratorom( )** pozivaju se samo da bi se prikazala odgovarajuća sintaksa, a njihova povratna vrednost se nigde ne koristi, iako se pamti. U nekim slučajevima, povratna vrednost se ne pamti jer se obično ne koristi. Pre nego što upotrebite ove funkcije, proučite sve načine njihovog korišćenja u dokumentaciji na Web adresi *java.sun.com*.

Vežba 7: (4) Napravite **ArrayList** i **LinkedList** i popunite obe generatorom **Countries.imena( )**. Ispišite svaku listu pomoću običnog iteratora, zatim **ListIterator**om umetnite jednu listu na svaku drugu lokaciju druge liste. Potom obavite umetanje počev od kraja prve liste unazad.

Vežba 8: (7) Napravite generičku klasu jednostruko ulančane liste i nazovite je **SList**. Neka ona *ne* realizuje interfejs **List**, tako je jednostavnije. Svaki **Link** objekat u listi treba da sadrži referencu na sledeći element liste, ali ne na prethodni (za razliku od nje, **LinkedList** je dvostruko ulančana lista, što znači da održava veze u oba smera). Napravite sopstveni **SListIterator** koji ne realizuje **ListIterator**, opet zbog jednostavnosti. Neka jedina metoda u klasi **SList** sem **toString( )** bude **iterator( )** koji proizvodi **SListIterator**. Jedini način umetanja i uklanjanja elemenata iz objekta tipa **SList** omogućuje **SListIterator**. Napišite kôd koji pokazuje rad liste **Slist**.

Skupovi i redosled skladištenja

Primeri skupova u poglavlju *Čuvanje objekata* predstavljaju dobar uvod u operacije koje se mogu obaviti sa osnovnim skupovima. Međutim, u tim primerima bili su prigodno korišćeni unapred definisani Javini tipovi kao što su **Integer** i **String**, koji su projektovani za primenu u kontejnerima. Kada pravite sopstvene tipove, imajte u vidu da **Set** zahteva metodu za održavanje redosleda smeštanja elemenata. Način održavanja redosleda smeštanja menja se u zavisnosti od realizacije interfejsa **Set**. Dakle, različite realizacije interfejsa **Set** ne samo da imaju različita ponašanja, nego su različiti i zahtevi u pogledu tipa objekta koji se može staviti u određeni **Set**:

Set (interfejs)	Svaki element koji dodajete u skup mora da bude jedinstven; u suprotnom, **Set** ne dodaje duplikat elementa. Objekti koji se dodaju u skup moraju da definišu metodu **equals( )** kojom se utvrđuje jedinstvenost objekta. **Set** ima istovetan interfejs kao **Collection**. Interfejs skupa ne garantuje održavanje elemenata u određenom rasporedu.
HashSet*	Klasa koja se koristi za skupove gde je važno brzo pronalaženje elemenata. Objekti moraju da definišu i metodu **hashCode( )**.
TreeSet	Uređen skup u obliku stabla. Iz ovog skupa možete da izdvojite niz uređen u određenom redosledu. Elementi moraju da definišu i interfejs **Comparable**.
LinkedHashSet	Ima brzinu pretraživanja klase **HashSet**, ali interno održava redosled kojim su elementi bili umetani (redosled umetanja) pomoću ulančane liste. Zato se rezultati pojavljuju u redosledu umetanja kada iterirate kroz **Set**. Elementi moraju definisati i metodu **hashCode( )**.

Zvezdica pored **HashSet** pokazuje sledeće: ako nema drugih ograničenja, to treba da bude podrazumevani izbor zato što je optimalan po brzini.

Obrazac za definiciju metode **hashCode( )** biće opisan u nastavku poglavlja. **equals( )** morate da definišete i za skladištenje s transformacijom ključa (heširanje) i za skladištenje u stablu, ali je metoda **hashCode( )** apsolutno neophodna samo ako će se klasa nalaziti u skupu tipa **HashSet** (što je verovatno, pošto bi ova klasa u suštini trebalo da bude vaš prvi izbor kao realizacija za **Set**) ili **LinkedHashSet**. Međutim, dobar stil programiranja nalaže da se uvek redefiniše metoda **hashCode( )** ako se redefiniše **equals( )**.

Ovaj primer ilustruje metode koje moraju biti definisane da bi se određeni tip uspešno upotrebljavao sa određenom realizacijom interfejsa **Set**:

```
//: kontejneri/TipoviZaSkupove.java
// Metode potrebne za stavljanje sopstvenog tipa u skup.
import java.util.*;

class TipSkupa {
  int i;
  public TipSkupa(int n) { i = n; }
  public boolean equals(Object o) {
    return o instanceof TipSkupa && (i == ((TipSkupa)o).i);
  }
  public String toString() { return Integer.toString(i); }
}

class TipHesiranja extends TipSkupa {
  public TipHesiranja(int n) { super(n); }
  public int hashCode() { return i; }
}

class TipStabla extends TipSkupa
implements Comparable<TipStabla> {
```

```
  public TipStabla(int n) { super(n); }
  public int compareTo(TipStabla arg) {
    return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
  }
}

public class TipoviZaSkupove {
  static <T> Set<T> fill(Set<T> skup, Class<T> tip) {
    try {
      for(int i = 0; i < 10; i++)
          skup.add(
            tip.getConstructor(int.class).newInstance(i));
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return skup;
  }
  static <T> void test(Set<T> skup, Class<T> tip) {
    fill(skup, tip);
    fill(skup, tip); // Pokušaj dodavanja duplikata
    fill(skup, tip);
    System.out.println(skup);
  }
  public static void main(String[] args) {
    test(new HashSet<TipHesiranja>(), TipHesiranja.class);
    test(new LinkedHashSet<TipHesiranja>(), TipHesiranja.class);
    test(new TreeSet<TipStabla>(), TipStabla.class);
    // Ovo ne radi:
    test(new HashSet<TipSkupa>(), TipSkupa.class);
    test(new HashSet<TipStabla>(), TipStabla.class);
    test(new LinkedHashSet<TipSkupa>(), TipSkupa.class);
    test(new LinkedHashSet<TipStabla>(), TipStabla.class);
    try {
      test(new TreeSet<TipSkupa>(), TipSkupa.class);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
    try {
      test(new TreeSet<TipHesiranja>(), TipHesiranja.class);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
} /* Ispis: (primer)
[2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 9, 7, 5, 1, 2, 6, 3, 0, 7, 2, 4, 4, 7, 9, 1, 3, 6, 2, 4, 3, 0, 5, 0,
```

```
8, 8, 8, 6, 5, 1]
[0, 5, 5, 6, 5, 0, 3, 1, 9, 8, 4, 2, 3, 9, 7, 3, 4, 4, 0, 7, 1, 9, 6, 2,
1, 8, 2, 8, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3,
4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3,
4, 5, 6, 7, 8, 9]
java.lang.ClassCastException: TipSkupa cannot be cast to
java.lang.Comparable
java.lang.ClassCastException: TipHesiranja cannot be cast to
java.lang.Comparable
*///:~
```

Da bi se pokazalo koje metode su neophodne za određeni skup i da bi se istovremeno izbeglo dupliranje koda, napravljene su tri klase. Osnovna klasa **TipSkupa** prosto skladišiti jedan **int** i ispisuje ga metodom **toString( )**. Pošto sve klase uskladištene u skupovima moraju imati metodu **equals( )**, i ona je uskladištena u osnovnoj klasi. Jednakost je zasnovana na vrednosti celog broja **i**.

TipHesiranja nasleđuje **TipSkupa** i dodaje metodu **hashCode( )** neophodnu da bi objekat bio smešten u realizaciju interfejsa **Set** s transformisanim ključem.

TipStabla realizuje interfejs **Comparable** koji je neophodan ako će objekat biti korišćen u uređenom kontejneru, kao što je **SortedSet**. (Jedina trenutno dostupna klasa koja ga realizuje je **TreeSet**.) Obratite pažnju na to da u metodi **compareTo( )** nisam koristio jednostavnu i očiglednu konstrukciju **return i-i2**. Iako je to česta greška u programiranju, sve bi radilo kako treba samo kada bi **i** i **i2** bili neoznačeni celi brojevi (pod uslovom da Java ima rezervisanu reč **unsigned** za neoznačen ceo broj, što nije slučaj). Greška se pojavljuje za označen ceo broj koji nije dovoljno velik da predstavi razliku dva označena cela broja (tip **int**). Ako je **i** veliki pozitivan ceo broj, a **j** veliki negativan ceo broj, **i-j** će dovesti do prekoračenja i daće negativnu vrednost, što nije ispravno.

Od metode **compareTo( )** obično očekujete da proizvodi prirodan redosled usaglašen s metodom **equals( )**. Ako **equals( )** daje **true** za određeno poređenje, onda bi **compareTo( )** trebalo da dâ nulu kao rezultat istog poređenja, a ukoliko **equals( )** daje **false** za neko poređenje, onda bi **compareTo( )** za isto poređenje trebalo da dâ rezultat različit od nule.

U klasi **TipoviZaSkupove**, metode **fill( )** i **test( )** definisane su pomoću generičkih tipova da bi se sprečilo dupliranje koda. Ponašanje skupa (objekta tipa **Set**) metoda **test( )** proverava s tri poziva metode **fill( )** za ispitni **skup**, pri čemu pokušava da u njega ubaci duplikate objekata. Metoda **fill( )** prima skup proizvoljnog tipa i **Class** objekat istog tipa. Pomoću objekta tipa **Class** pronalazi konstruktor koji prima celobrojni argument i poziva taj konstruktor da bi skupu dodavala elemente.

Iz rezultata vidite da **HashSet** čuva elemente u nekom misterioznom poretku (koji ćemo objasniti u nastavku poglavlja), **LinkedHashSet** čuva elemente u poretku kojim su bili umetnuti, a **TreeSet** čuva elemente u uređenom poretku (zbog načina realizacije metode **compareTo( )**, to je slučajno opadajući poredak).

Ukoliko pokušamo da upotrebimo tipove koji ne podržavaju ispravno operacije sa skupovima koji te operacije zahtevaju, sve se ruši. Smeštanje objekta tipa **TipSkupa** ili **TipStabla** koji ne obuhvata redefinisanu metodu **hashCode**( ) u bilo koju realizaciju s transformisanjem ključa rezultuje dupliranjem vrednosti, čime se krši osnovno pravilo skupova (interfejsa **Set**). To prilično obespokojava, pošto se greška ne generiše čak ni tokom izvršavanja. Međutim, podrazumevana metoda **hashCode**( ) legitimna je, pa je takvo ponašanje dozvoljeno, uprkos tome što je nekorektno. Jedini pouzdan način da se obezbedi ispravnost takvog programa jeste uključivanje testiranja anotacijama &unit u *build* sistem (više informacija potražite u dodatku na adresi *http://MindView.net/Books/BetterJava*).

Ako u objektu tipa **TreeSet** pokušate da upotrebite tip koji ne realizuje interfejs **Comparable**, dobićete određeniji rezultat: kada **TreeSet** pokuša da upotrebi taj objekat kao uređen (jer metoda **compareTo**( ) definiše neki poredak), biće generisan izuzetak.

SortedSet

Elementi u kontejneru tipa **SortedSet** zajemćeno su u uređenom poretku, zbog čega sledeće metode u interfejsu **SortedSet** mogu da pruže dodatnu funkcionalnost:

Comparator comparator( ): Proizvodi **Comparator** koji se upotrebljava za ovaj skup (**Set**) ili **null** za prirodan poredak.
Object first( ): Proizvodi najniži element.
Object last( ): Proizvodi najviši element.
SortedSet subSet(odElement, doElement): Proizvodi prikaz ovog skupa sa elementima od **odElement**, uključivo, do **doElement**, isključivo.
SortedSet headSet(doElement): Proizvodi prikaz ovog skupa sa elementima manjim od **doElement**.
SortedSet tailSet(odElement): Proizvodi prikaz ovog skupa sa elementima jednakim ili većim od elementa **odElement**.

Evo jednostavnog primera:

```
//: kontejneri/PrimerZaSortedSet.java
// Šta se može uraditi sa objektom tipa TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class PrimerZaSortedSet {
  public static void main(String[] args) {
    SortedSet<String> sortedSet = new TreeSet<String>();
    Collections.addAll(sortedSet,
      "jedan dva tri cetiri pet sest sedam osam"
        .split(" "));
    print(sortedSet);
    String najmanji = sortedSet.first();
    String najveci = sortedSet.last();
    print(najmanji);
    print(najveci);
```

```
    Iterator<String> it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) najmanji = it.next();
      if(i == 6) najveci = it.next();
      else it.next();
    }
    print(najmanji);
    print(najveci);
    print(sortedSet.subSet(najmanji, najveci));
    print(sortedSet.headSet(najveci));
    print(sortedSet.tailSet(najmanji));
  }
} /* Ispis:
[cetiri, dva, jedan, osam, pet, sedam, sest, tri]
cetiri
tri
osam
tri
[osam, pet, sedam, sest]
[cetiri, dva, jedan, osam, pet, sedam, sest]
[osam, pet, sedam, sest, tri]
*///:~
```

Imajte u vidu da **SortedSet** znači „uređeno u skladu s funkcijom poređenja objekta", a ne „uređeno po redosledu umetanja". Redosled umetanja se može očuvati pomoću kontejnera **LinkedHashSet**.

Vežba 9: (2) Upotrebite **RandomGenerator.String** za popunjavanje objekta tipa **TreeSet**, ali uz abecedni redosled. Ispišite **TreeSet** da biste proverili redosled sortiranja.

Vežba 10: (7) Definišite sopstveni **SortedSet** pomoću objekta tipa **LinkedList** kao pripadne realizacije.

Redovi za čekanje

Sem u aplikacijama za paralelno izvršavanje, Java SE5 je realizovala **Queue** (red za čekanje) samo u kontejnerima **LinkedList** i **PriorityQueue** koji se razlikuju po ponašanju, a ne po performansama. Sledi elementarni primer s većinom realizacija reda za čekanje (koje u ovom primeru neće sve funkcionisati), uključujući tu i redove za čekanje napravljene za paralelno izvršavanje. Elemente treba umetati od jednog kraja, a vaditi od drugog:

```
//: kontejneri/PonasanjeRedaZaCekanje.java
// Poredi ponašanje nekih redova za čekanje
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
```

```
public class PonasanjeRedaZaCekanje {
  private static int brojac = 10;
  static <T> void test(Queue<T> queue, Generator<T> gen) {
    for(int i = 0; i < brojac; i++)
      queue.offer(gen.next());
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  static class Gen implements Generator<String> {
    String[] s = ("jedan dva tri cetiri pet sest sedam " +
      "osam devet deset").split(" ");
    int i;
    public String next() { return s[i++]; }
  }
  public static void main(String[] args) {
    test(new LinkedList<String>(), new Gen());
    test(new PriorityQueue<String>(), new Gen());
    test(new ArrayBlockingQueue<String>(count), new Gen());
    test(new ConcurrentLinkedQueue<String>(), new Gen());
    test(new LinkedBlockingQueue<String>(), new Gen());
    test(new PriorityBlockingQueue<String>(), new Gen());
  }
} /* Ispis:
jedan dva tri cetiri pet sest sedam osam devet deset
osam pet četiri devet jedan sedam šest deset tri dva
cetiri deset devet dva jedan osam pet sedam sest tri
cetiri deset devet dva jedan osam pet sedam sest tri
cetiri deset devet dva jedan osam pet sedam sest tri
osam pet četiri devet jedan sedam šest deset tri dva
*///:~
```

Vidite da objekat tipa **Queue** daje elemente u istom poretku u kojem su u njega bili umetani, izuzev u redovima za čekanje s prioritetom.

Redovi za čekanje s prioritetom

Osnovno objašnjenje redova za čekanje s prioritetom naći ćete u poglavlju *Čuvanje objekata*. Zanimljiviji problem je lista zadataka, gde svaki objekat sadrži znakovni niz i po jednu vrednost primarnog i sekundarnog prioriteta. Uređenje te liste takođe je određeno realizacijom interfejsa **Comparable**:

```
//: kontejneri/ListaZadataka.java
// Složenija upotreba kontejnera PriorityQueue.
import java.util.*;

class ListaZadataka extends PriorityQueue
                    <ListaZadataka.StavkaListeZadataka> {
```

```
  static class StavkaListeZadataka implements Comparable
                      <StavkaListeZadataka> {
    private char primarni;
    private int sekundarni;
    private String item;
    public StavkaListeZadataka(String td, char pri, int sek) {
      primarni = pri;
      sekundarni = sek;
      stavka = uraditi;
    }
    public int compareTo(StavkaListeZadataka arg) {
      if(primarni > arg.primarni)
        return vrednost+1;
      if(primarni == arg.primarni)
        if(sekundarni > arg.sekundarni)
          return +1;
        else if(sekundarni == arg.sekundarni)
          return 0;
      return -1;
    }
    public String toString() {
      return Character.toString(primarni) +
        sekundarni + ": " + stavka;
    }
  }
  public void add(String uraditi, char pri, int sek) {
    super.add(new StavkaListeZadataka(uraditi, pri, sek));
  }
  public static void main(String[] args) {
    ListaZadataka ListaZadataka = new ListaZadataka();
    ListaZadataka.add("Baciti smeće", 'C', 4);
    ListaZadataka.add("Nahraniti psa", 'A', 2);
    ListaZadataka.add("Nahraniti ptičicu", 'B', 7);
    ListaZadataka.add("Pokositi travu", 'C', 3);
    ListaZadataka.add("Zaliti travnjak", 'A', 1);
    ListaZadataka.add("Nahraniti mačku", 'B', 1);
    while(!ListaZadataka.isEmpty())
      System.out.println(ListaZadataka.remove());
  }
} /* Ispis:
A1: Zaliti travnjak
A2: Nahraniti psa
B1: Nahraniti mačku
B7: Nahraniti ptičicu
C3: Pokositi travu
C4: Baciti smeće
*///:-
```

Obratite pažnju na to da su u redu za čekanje s prioritetom stavke sortirane automatski.

Vežba 11: (2) Napravite klasu koja sadrži objekat tipa **Integer** inicijalizovan metodom **java.util.Random** na vrednost izmedu 0 i 100. Realizujte **Comparable** pomoću tog **Integer** polja. Popunite objekat tipa **PriorityQueue** objektima vaše klase i izvucite te vrednosti metodom **poll( )** da bi se pokazalo da kontejner proizvodi očekivani poredak.

Dvostrani redovi za čekanje

Dvored (*dvostrani red za čekanje*, engl. *double-ended queue, dequeue*) jeste red za čekanje u koji elemente možete dodavati i iz njega ih uklanjati sa oba kraja. Kontejner **LinkedList** ima i metode koje podržavaju operacije u dvostranim redovima za čekanje, ali standardne Javine biblioteke ne sadrže eksplicitan interfejs za njih. Zato **LinkedList** ne može da realizuje taj interfejs i nije moguće svođenje naviše na interfejs **Deque**, dok je u prethodnom primeru bilo moguće svođenje na **Queue**. Međutim, klasu tipa **Deque** možete napraviti pomoću kompozicije i jednostavno eksponirati relevantne metode kontejnera **LinkedList**:

```
//: net/mindview/util/Dvored.java
// Pravljenje dvostranog reda za čekanje od kontejnera LinkedList.
package net.mindview.util;
import java.util.*;

public class Dvored<T> {
  private LinkedList<T> dvored = new LinkedList<T>();
  public void addFirst(T e) { dvored.addFirst(e); }
  public void addLast(T e) { dvored.addLast(e); }
  public T getFirst() { return dvored.getFirst(); }
  public T getLast() { return dvored.getLast(); }
  public T removeFirst() { return dvored.removeFirst(); }
  public T removeLast() { return dvored.removeLast(); }
  public int size() { return dvored.size(); }
  public String toString() { return dvored.toString(); }
  // I druge metode po potrebi...
} ///:~
```

Ako ovaj **Dvored** upotrebite u sopstvenim programima, verovatno ćete morati da dodate još metoda da bi postao upotrebljiv.

Sledi jednostavna provera klase **Dvored**:

```
//: kontejneri/DvoredTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DvoredTest {
  static void testPopunjavanja(Dvored<Integer> dvored) {
    for(int i = 20; i < 27; i++)
      dvored.addFirst(i);
    for(int i = 50; i < 55; i++)
      dvored.addLast(i);
  }
```

```
  public static void main(String[] args) {
    Dvored<Integer> di = new Dvored<Integer>();
    testPopunjavanja(di);
    print(di);
    while(di.size() != 0)
      printnb(di.removeFirst() + " ");
    print();
    testPopunjavanja(di);
    while(di.size() != 0)
      printnb(di.removeLast() + " ");
  }
} /* Ispis:
[26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
26 25 24 23 22 21 20 50 51 52 53 54
54 53 52 51 50 20 21 22 23 24 25 26
*///:~
```

Elementi se ređe umeću na oba kraja i vade sa oba kraja, pa se **Dvored** koristi rede od običnog reda za čekanje (**Queue**).

Iscrpnije o **Map**ama

Kao što ste saznali u poglavlju *Čuvanje objekata*, osnovna namena mape (*asocijativnog niza*) jeste da održava parove ključ–vrednost (asocijacije, pridruživanja, mapiranja, preslikavanja), tako da vrednost možete pronaći pomoću ključa. Standardna Javina biblioteka sadrži različite osnovne realizacije **Map**e: **HashMap**, **TreeMap**, **LinkedHashMap**, **WeakHashMap**, **ConcurrentHashMap** i **IdentityHashMap**. Svima im je zajednički osnovni interfejs **Map**, a razlikuju se u ponašanju i efikasnosti, poretku čuvanja i prezentovanja parova, roku čuvanja objekata u mapi, radu mape u višenitnim programima i načinu utvrđivanja jednakosti ključa. Iz broja realizacija interfejsa **Map** trebalo bi da zaključite koliko je ova alatka važna.

Da biste mogli bolje da shvatite **Map**e, pogledaćemo kako se pravi asocijativni niz. Evo jedne izuzetno jednostavne realizacije:

```
//: kontejneri/ AsocijativniNiz.java
// Ključeve pridružuje vrednostima.
import static net.mindview.util.Print.*;

public class AsocijativniNiz<K,V> {
  private Object[][] parovi;
  private int indeks;
  public AsocijativniNiz(int duzina) {
    parovi = new Object[duzina][2];
  }
  public void put(K kljuc, V vrednost) {
    if(indeks >= parovi.duzina)
      throw new ArrayIndexOutOfBoundsException();
```

```
    parovi[indeks++] = new Object[]{ kljuc, vrednost };
  }
  @SuppressWarnings("unchecked")
  public V get(K kljuc) {
    for(int i = 0; i < indeks; i++)
      if(kljuc.equals(parovi[i][0]))
        return (V)parovi[i][1];
    return null; // Ključ nije pronađen
  }
  public String toString() {
    StringBuilder rezultat = new StringBuilder();
    for(int i = 0; i < indeks; i++) {
      rezultat.append(parovi[i][0].toString());
      rezultat.append(" : ");
      rezultat.append(parovi[i][1].toString());
      if(i < indeks - 1)
        rezultat.append("\n");
    }
    return rezultat.toString();
  }
  public static void main(String[] args) {
    AsocijativniNiz<String,String> mapa =
      new AsocijativniNiz<String,String>(6);
    mapa.put("nebo", "plavo");
    mapa.put("trava", "zelena");
    mapa.put("okean", "nemiran");
    mapa.put("stablo", "visoko");
    mapa.put("zemlja", "smeđa");
    mapa.put("sunce", "toplo");
    try {
      map.put("dodatni", "objekat"); // Iza kraja
    } catch(ArrayIndexOutOfBoundsException e) {
      print("Previše objekata!");
    }
    print(map);
    print(map.get("okean"));
  }
} /* Ispis:
Previše objekata!
nebo : plavo
trava : zelena
okean : nemiran
stablo : visoko
zemlja : smeđa
sunce : toplo
nemiran
*///:~
```

Osnovne metode asocijativnog niza su **put( )** i **get( )**, ali je radi lakšeg prikazivanja metoda **toString( )** redefinisana tako da ispisuje parove ključ–vrednost. Da bismo pokazali da to funkcioniše, **main( )** učitava jedan **AsocijativniNiz** parova znakovnih nizova i ispisuje tako dobijenu mapu, a iza toga **get( )** vadi iz mape jednu od vrednosti.

Metodu **get( )** upotrebljavate tako što joj prosledite **kljuc** koji želite da pronađe, a ona vadi i kao rezultat vraća njemu pridruženu vrednost ili **null** ako ga ne pronađe. Metoda **get( )** upotrebljava verovatno najmanje efikasan način lociranja vrednosti koji se može zamisliti: krene od vrha niza i metodom **equals( )** poredi sve ključeve redom. Ali poenta je jednostavnost, a ne efikasnost.

Dakle, gornja verzija je poučna, ali ne i veoma efikasna, a ima i nepromenljivu veličinu, što nije prilagodljivo. Srećom, **Map**e u paketu **java.util** nemaju te probleme, a možete ih zameniti u gornjem primeru.

Vežba 12: (1) Zamenite po jedan kontejner tipa **HashMap**, **TreeMap** odnosno **LinkedHashMap** u metodi **main( )** programa **AsocijativniNiz.java**.

Vežba 13: (4) Upotrebite **AsocijativniNiz.java** za prebrojavanje reči, pri čemu se **String** mapira (preslikava) u **Integer**. Uslužnom metodom **net.mindview.util.TextFile** (iz ove knjige) otvorite tekstualnu datoteku i podelite je na reči, pri čemu su graničnici razmak i znakovi interpunkcije, i prebrojte reči u toj datoteci.

Performanse

Performanse su temeljni problem pri radu s mapama, jer je linearno pretraživanje metodom **get( )** veoma spor način traženja ključa. Ovde pomaže brži kontejner **HashMap**. Umesto sporog traženja ključa, on upotrebljava posebnu vrednost nazvanu *ključ za heširanje* (engl. *hash code*). Ključ za heširanje predstavlja način da se određena informacija u objektu pretvori u „relativno jedinstven" celi broj (**int**) za taj objekat. Metoda **hashCode( )** je definisana u korenskoj klasi **Object**, pa svi Java objekti mogu da proizvedu ključ za heširanje. Kontejner **HashMap** uzima **hashCode( )** objekta i njome brzo traga za ključem. Time se dobija ogromno poboljšanje performansi.[6]

Navešćemo osnovne realizacije interfejsa **Map**. Zvezdica kod **HashMap**e pokazuje da bi je trebalo podrazumevano izabrati (ako nema drugih ograničenja), zato što je optimizovana po brzini. Ostale realizacije imaju naglašenija druga obeležja, te su sporije od **HashMap**e.

Transformacija ključeva je najčešći način skladištenja elemenata u mapi. Kasnije ćemo objasniti kako se obavlja ta transformacija.

---

[6] Ako su uprkos ovim ubrzanjima performanse još uvek nezadovoljavajuće, pretraživanje tabela možete dodatno ubrzati ukoliko napišete sopstvenu **Map**u i prilagodite je svojim tipovima, da ne biste trošili vreme na konverziju u tip **Object** i iz njega. Za još bolje performanse, zaljubljenici u brzinu mogu iskoristiti knjigu *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition* Donalda Knutha; ona će im pomoći da nizovima zamene liste s prelivajućim kofama, što daje još dve pogodnosti: njih možete optimizovati prema karakteristikama skladištenja na disku i oni će vam uštedeti najveći deo vremena potrebnog za pravljenje pojedinačnih zapisa i njihovo prikupljanje kada postanu smeće.

HashMap*	Realizacija zasnovana na tabeli transformisanih ključeva. (Upotrebite je umesto **Hashtable**.) Umetanje i pronalaženje parova obavlja u konstantnom vremenu. Performanse se mogu podesiti preko konstruktora koji omogućuju zadavanje *kapaciteta* i *faktora opterećenja* tabele transformisanih ključeva.
LinkedHashMap	Poput **HashMape**, ali kada kroz nju iterirate, parove dobijate u redosledu umetanja ili u redosledu najdavnijeg korišćenja (engl. *least-recently-used*, LRU). Samo malo sporija od **HashMape**, sem za iteriranje, kada je brža zbog ulančane liste koja održava interni poredak.
TreeMap	Realizacija napravljena na osnovu crveno-crnog stabla. Kada pogledate ključeve ili parove, oni će biti u uređenom poretku (koji određuje **Comparable** ili **Comparator**). Suština primene kontejnera **TreeMap** jeste da rezultate dobijate u uređenom poretku. **TreeMap** je jedina **Mapa** koja ima metodu **subMap( )**, dakle jedina koja može da vrati deo stabla.
WeakHashMap	Mapa *slabih ključeva* koji omogućuju da objekti na koje mapa upućuje budu oslobođeni (sakupljeni u smeće); koristi se za rešavanje određenih vrsta problema. Ako izvan mape ne postoje reference na određeni ključ, on može biti pokupljen kao smeće.
ConcurrentHashMap	**Mapa** koja bezbedno radi u višenitnom radu, a da nema zaključavanje zbog sinhronizacije. Ovo je objašnjeno u poglavlju *Paralelno izvršavanje*.
IdentityHashMap	Mapa transformisanih ključeva koja za poređenje ključeva upotrebljava == umesto metode **equals( )**. Samo za rešavanje posebnih vrsta problema; nije za opštu upotrebu.

Za ključeve za **Map**u postavljaju se isti zahtevi kao za elemente skupa (**Set**). Upoznali ste ih u primeru **TipoviZaSkupove.java**. Svaki ključ mora imati metodu **equals( )**. Ako se ključ upotrebljava u **Map**i transformisanih ključeva, on mora imati i propisnu metodu **hashCode( )**. Ključ koji se upotrebljava u **TreeMap**i mora realizovati **Comparable**.

U narednom primeru pokazaćemo operacije dostupne preko interfejsa **Map**, uz pomoć prethodno definisanog skupa ispitnih podataka **BrojackaMapaPodataka**:

```
//: kontejneri/Mape.java
// Šta se može uraditi s mapama.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Mape {
  public static void printKeys(Map<Integer,String> mapa) {
    printnb("Veličina = " + mapa.size() + ", ");
    printnb("Ključevi: ");
    print(mapa.keySet()); // Daje skup ključeva
  }
  public static void test(Map<Integer,String> mapa) {
    print(mapa.getClass().getSimpleName());
    mapa.putAll(new BrojackaMapaPodataka(25));
```

```
    // Ponašanje ključeva u mapi jednako je onome u skupu ('Set'):
    map.putAll(new BrojackaMapaPodataka(25));
    printKeys(mapa);
    // Pravljenje kolekcije vrednosti:
    printnb("Vrednosti: ");
    print(mapa.values());
    print(mapa);
    print("mapa.containsKey(11): " + map.containsKey(11));
    print("mapa.get(11): " + map.get(11));
    print("mapa.containsValue(\"F0\"): "
      + mapa.containsValue("F0"));
    Integer kljuc = mapa.keySet().iterator().next();
    print("Prvi ključ u mapi: " + kljuc);
    mapa.remove(kljuc);
    printKeys(mapa);
    mapa.clear();
    print("mapa.isEmpty(): " + mapa.isEmpty());
    mapa.putAll(new BrojackaMapaPodataka(25));
    // Operacije nad tim skupom menjaju mapu:
    mapa.keySet().removeAll(mapa.keySet());
    print("mapa.isEmpty(): " + mapa.isEmpty());
  }
  public static void main(String[] args) {
    test(new HashMap<Integer,String>());
    test(new TreeMap<Integer,String>());
    test(new LinkedHashMap<Integer,String>());
    test(new IdentityHashMap<Integer,String>());
    test(new ConcurrentHashMap<Integer,String>());
    test(new WeakHashMap<Integer,String>());
  }
} /* Ispis:
HashMap
Veličina = 25, Ključevi: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4,
19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
Vrednosti: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0, E0, T0, L0,
S0, D0, M0, R0, C0, N0, U0, K0, F0, A0]
{15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0, 1=B0,
14=O0, 24=Y0, 4=E0, 19=T0, 11=L0, 18=S0, 3=D0, 12=M0, 17=R0, 2=C0,
13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
mapa.containsKey(11): true
mapa.get(11): L0
mapa.containsValue("F0"): true
Prvi ključ u mapi: 15
Veličina = 24, Ključevi: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4, 19,
11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
mapa.isEmpty(): true
mapa.isEmpty(): true
...
*///:~
```

Metoda **printKeys( )** pokazuje kako se pravi kolekcija (**Collection**) od **Map**e. Metoda **keySet( )** proizvodi skup (**Set**) ključeva iz **Map**e. Rezultate metode **values( )** lako ćete štampati zbog poboljšane podrške za štampanje u Javi SE5; ta metoda pravi kolekciju od svih vrednosti u **Map**i. (Vodite računa o tome da ključne reči moraju biti jedinstvene, dok vrednosti mogu imati duplikate.) Pošto **Map**a održava te kolekcije, svaka izmena u kolekciji odražava se u njoj pridruženoj **Map**i.

Ostatak programa pruža jednostavne primere svih operacija s **Map**ama i testira sve osnovne vrste **Map**a.

Vežba 14: (3) Pokažite da **java.util.Properties** radi u gornjem programu.

SortedMap

Ako imate neku realizaciju interfejsa **SortedMap** (od njih je dostupna samo **TreeMap**), ključevi su zajamčeno u uređenom poretku, zbog čega sledeće metode interfejsa **SortedMap** mogu da pruže dodatnu funkcionalnost:

Comparator comparator( ): Proizvodi komparator za upotrebu u ovoj **Map**i ili **null** za prirodni poredak.

T firstKey( ): Proizvodi najmanji ključ.

T lastKey( ): Proizvodi najveći ključ.

SortedMap subMap(odKljuč, doKljuč): Proizvodi prikaz dela mape s ključevima od ključa **odKljuč**, uključivo, do ključa **doKljuč**, isključivo.

SortedMap headMap(toKey): Proizvodi prikaz dela mape s ključevima manjim od ključa **doKljuč**.

SortedMap tailMap(fromKey): Proizvodi prikaz dela mape s ključevima jednakim ili većim od ključa **odKljuč**.

Ovaj primer liči na **PrimerZaSortedSet.java** i pokazuje to dodatno ponašanje **TreeMap**e:

```
//: kontejneri/PrimerZaSortedMap.java
// Šta se može uraditi s mapom TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PrimerZaSortedMap {
  public static void main(String[] args) {
    TreeMap<Integer,String> sortedMap =
      new TreeMap<Integer,String>(new BrojackaMapaPodataka(10));
    print(sortedMap);
    Integer najmanji = sortedMap.firstKey();
    Integer najveci = sortedMap.lastKey();
    print(najmanji);
    print(najveci);
    Iterator<Integer> it = sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
```

```
      if(i == 3) najmanji = it.next();
      if(i == 6) najveci = it.next();
      else it.next();
    }
    print(najmanji);
    print(najveci);
    print(sortedMap.subMap(najmanji, najveci));
    print(sortedMap.headMap(najveci));
    print(sortedMap.tailMap(najmanji));
  }
} /* Ispis:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
0
9
3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*///:~
```

Ovde su parovi uskladišteni po redosledu ključeva. Pošto je mapa **TreeMap** uređena, koncept „mesta" ima smisla, pa se zna šta je prvi i poslednji element i šta je podmapa sa elementima od – do.

LinkedHashMap

Mapa **LinkedHashMap** transformiše sve ključeve radi brzine, ali tokom prolaska parove daje u redosledu umetanja (**System.out.println**( ) iterira kroz mapu, pa rezultate prolaska možete videti). Sem toga, realizacija **LinkedHashMap**e može se u konstruktoru konfigurisati tako da upotrebljava algoritam *najdavnijeg korišćenja* (engl. *least-recently-used*, LRU) odnosno pristupanja, pa su na početku liste elementi kojima nije pristupano (te su stoga kandidati za uklanjanje). Time je olakšano pravljenje programa koji periodično čiste smeće da bi se uštedeo memorijski prostor. Evo jednostavnog primera u kojem su prikazane obe funkcionalnosti:

```
//: kontejneri/PrimerZaLinkedHashMap.java
// Šta se može uraditi s kontejnerom LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PrimerZaLinkedHashMap {
  public static void main(String[] args) {
    LinkedHashMap<Integer,String> linkedMap =
      new LinkedHashMap<Integer,String>(
        new BrojackaMapaPodataka(9));
```

```
    print(linkedMap);
    // Poredak najdavnijeg korišćenja:
    linkedMap =
      new LinkedHashMap<Integer,String>(16, 0.75f, true);
    linkedMap.putAll(new BrojackaMapaPodataka(9));
    print(linkedMap);
    for(int i = 0; i < 6; i++) // Prouzrokuj pristupanje:
      linkedMap.get(i);
    print(linkedMap);
    linkedMap.get(0);
    print(linkedMap);
  }
} /* Ispis:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*///:~
```

Iz ispisa vidite da prolazak kroz mapu zaista daje parove u redosledu umetanja, čak i u LRU verziji. Međutim, nakon (samo) prvih šest pristupanja u LRU verziji, poslednje tri stavke prelaze na početak liste. Zatim, kada se ponovo pristupi stavci 0, ona prelazi na začelje liste.

Transformisanje ključeva i ključevi za heširanje

U primerima iz poglavlja *Čuvanje objekata*, kao **HashMap** ključevi bile su upotrebljene unapred definisane klase. Ti primeri su funkcionisali zato što su te unapred definisane klase imale potreban kôd da mogu ispravno da se ponašaju u ulozi ključeva.

Uobičajena greška je pravljenje sopstvenih klasa za ključeve u **HashMap**i, ali bez potrebnog koda. Na primer, zamislite sistem za prognoziranje vremena koji objekte tipa **Medved** pridružuje objektima tipa **Prognoza**. To izgleda prilično jednostavno – napravite te dve klase i objekat tipa **Medved** upotrebite kao ključ, a objekat tipa **Prognoza** kao vrednost:

```
//: kontejneri/Medved.java
// Izgleda uverljivo, ali ne radi kao ključ HashMape.

public class Medved {
  protected int broj;
  public Medved(int n) { broj = n; }
  public String toString() {
    return "Medved #" + broj;
  }
} ///:~

//: kontejneri/Prognoza.java
```

```
// Prognoziranje vremena s medvedima.
import java.util.*;

public class Prognoza {
  private static Random slucajan = new Random(47);
  private boolean senka = slucajan.nextDouble() > 0.5;
  public String toString() {
    if(senka)
      return "Još šest sedmica zime!";
    else
      return "Rano proleće!";
  }
} ///:~

//: kontejneri/DetektorProleca.java
// Kakvo će biti vreme?
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class DetektorProleca {
  // Koristi klasu Medved ili neku njenu potklasu:
  public static <T extends Medved>
  void otkrivanjeProleca(Class<T> type) throws Exception {
    Constructor<T> meda = type.getConstructor(int.class);
    Map<Medved,Prognoza> mapa =
      new HashMap<Medved,Prognoza>();
    for(int i = 0; i < 10; i++)
      mapa.put(meda.newInstance(i), new Prognoza());
    print("mapa = " + mapa);
    Medved md = meda.newInstance(3);
    print("Traženje prognoze za " + md);
    if(mapa.containsKey(md))
      print(mapa.get(md));
    else
      print("Nije pronađen ključ: " + md);
  }
  public static void main(String[] args) throws Exception {
    otkrivanjeProleca(Medved.class);
  }
} /* Ispis:
mapa= {Medved #3=Rano proleće!, Medved #7=Rano proleće!,
Medved #5=Rano proleće!, Medved #9=Još šest sedmica zime!,
Medved #8=Još šest sedmica zime!, Medved #0=Još šest sedmica zime!,
Medved #6=Rano proleće!, Medved #4=Još šest sedmica zime!,
Medved #1=Još šest sedmica zime!, Medved #2=Rano proleće!}
Traženje prognoze za Medved #3
Nije pronađen ključ: Medved #3
*///:~
```

Svaki **Medved** dobija identifikacioni broj da biste u **HashMap**i mogli da pronađete određeni objekat tipa **Prognoza** tako što ćete kazati: „Daj mi objekat tipa **Prognoza** pridružen objektu #3 tipa **Medved**". Klasa **Prognoza** sadrži objekat tipa **boolean** inicijalizovan metodom **java.util.random( )** i metodu **toString( )** koja tumači rezultate. Metoda **otkrivanjeProleca( )** napravljena je procesom refleksije da generiše primerak klase **Medved** ili njene potklase i da ga upotrebljava. To će nam dobro doći kasnije, kada za rešavanje ovde prikazanog problema nasledimo nov tip klase **Medved**.

HashMapa se popunjava objektima tipa **Medved** i njima pridruženim objektima tipa **Prognoza**. Ta **HashMap**a se ispisuje da biste videli da je popunjena. Zatim se **Medved** broj 3 koristi kao ključ za pronalaženje prognoze za objekat #3 tipa **Medved** (za koji možete videti da mora biti u **Map**i).

Sve izgleda veoma jednostavno, ali ne radi – ne može da se pronađe ključ za #3. Problem je to što se **Medved** automatski nasleđuje od zajedničke korenske klase **Object**, pa se za generisanje ključa za heširanje za svaki objekat upotrebljava metoda **hashCode( )** klase **Object**. Ona za heširanje podrazumevano upotrebljava samo adresu svog objekta. Zato prvi primerak **Medved(3)** *ne* proizvodi ključ za heširanje jednak ključu za heširanje drugog primerka **Medved(3)**, koji smo pokušali da upotrebimo za pretraživanje.

Možda mislite da je dovoljno napisati odgovarajuću redefiniciju metode **hashCode( )**. Ali ni to neće funkcionisati sve dok ne uradite još nešto: redefinišete metodu **equals( )** koja je takođe deo klase **Object**. **HashMap**a upotrebljava **equals( )** prilikom utvrđivanja da li je ključ jednak nekom ključu u tabeli.

Prava metoda **equals( )** mora zadovoljiti sledećih pet uslova:

1. Refleksivna: Za svaki **x**, **x.equals(x)** treba da vraća **true**.
2. Simetrična: Za svaki **x** i **y**, **x.equals(y)** treba da vraća **true** ako i samo ako **y.equals(x)** vraća **true**.
3. Tranzitivna: Za svaki **x**, **y** i **z**, ako **x.equals(y)** vraća **true** i **y.equals(z)** vraća **true**, onda **x.equals(z)** treba da vraća **true**.
4. Konsistentna: Za svaki **x** i **y**, svaki poziv **x.equals(y)** treba dosledno da vraća **true** ili dosledno da vraća **false**, ukoliko se informacije upotrebljene pri utvrđivanju jednakosti objekata ne promene.
5. Za svaki **x** različit od **null**, **x.equals(null)** treba da vraća **false**.

Ponavljam, podrazumevana metoda **Object.equals( )** jednostavno poredi adrese objekata, pa jedan **Medved(3)** nije jednak drugom objektu **Medved(3)**. Dakle, da biste u **HashMap**i mogli da upotrebljavate sopstvene klase kao ključeve, morate redefinisati i **hashCode( )** i **equals( )**, kao što je urađeno u sledećem rešenju problema s medvedom:

```
//: kontejneri/Medved2.java
// Klasa upotrebljena kao ključ u HashMapi
// mora redefinisati metode hashCode() i equals().

public class Medved2 extends Medved {
  public Medved2(int n) { super(n); }
  public int hashCode() { return broj; }
```

```
  public boolean equals(Object o) {
    return o instanceof Medved2 &&
      (broj == ((Medved2)o).broj);
  }
} ///:~

//: kontejneri/DetektorProleca2.java
// Ključ koji funkcioniše.

public class DetektorProleca2 {
  public static void main(String[] args) throws Exception {
    DetektorProleca.otkrivanjeProleca(Medved2.class);
  }
} /* Ispis:
mapa = {Medved #2=Rano proleće!, Medved #4=Još šest sedmica zime!,
Medved #9=Još šest sedmica zime!, Medved #8=Još šest sedmica zime!,
Medved #6=Rano proleće!, Medved #1=Još šest sedmica zime!, Medved
#3=Rano proleće!, Medved #7=Rano proleće!, Medved #5=Rano proleće!,
Medved #0=Još šest sedmica zime!}
Traženje prognoze za Medved #3
Rano proleće!
*///:~
```

Medved2.hashCode( ) vraća broj medveda kao vrednost transformisanog ključa. U ovom primeru, programer treba da se postara da svaki medved ima jedinstven ID broj. Od metode **hashCode( )** ne zahteva se da vraća jedinstvene identifikatore (što ćemo detaljnije objasniti u nastavku poglavlja), ali metoda **equals( )** mora strogo da utvrdi da li su dva objekta ekvivalentna. Ovde metoda **equals( )** proverava broj medveda, pa ako u mapi **HashMap** postoje dva objekta tipa **Medved2** sa istim brojem medveda, stvar neće funkcionisati.

Iako izgleda kao da metoda **equals( )** proverava samo da li je njen argument instanca od **Medved2** (pomoću rezervisane reči **instanceof**, objašnjene u poglavlju *Podaci o tipu*), **instanceof** zapravo potiho proverava i da li je objekat jednak **null**, pošto **instanceof** daje rezultat **false** ako je njen levi argument jednak **null**. Ako je tip odgovarajući i nije **null**, porede se vrednosti **broj** u oba objekta. Iz ispisa rezultata vidite da je ponašanje sada ispravno.

Kada pravite klase za upotrebu u kontejneru tipa **HashSet**, morate obratiti pažnju na iste stvari kao kada ih upotrebljavate kao ključeve u kontejneru tipa **HashMap**.

Način rada metode **hashCode( )**

Prethodni primer je tek prvi korak u ispravnom rešavanju problema. Iz njega vidite sledeće: ukoliko ne redefinišete metode **hashCode( )** i **equals( )** za svoj ključ, struktura transformisanih podataka (**HashSet**, **HashMap**, **LinkedHashSet** ili **LinkedHashMap**) verovatno neće ispravno raditi s tim ključem. Da biste pronašli *dobro* rešenje problema, treba da razumete šta se dešava unutar strukture transformisanih podataka.

Prvo, prisetite se zašto transformišemo ključeve: hteli smo mogućnost da jedan objekat pronađemo pomoću drugog. To se može postići i pomoću kontejnera **TreeMap** ili čak i realizacijom sopstvene **Mape**. Nasuprot realizaciji s transformisanjem ključeva, naredni primer realizuje **Mapu** pomoću para **ArrayList** kontejnera. Za razliku od primera **AsocijativniNiz.java**, u narednom se interfejs **Map** potpuno realizuje, što objašnjava postojanje metode **entrySet( )**:

```
//: kontejneri/SporaMapa.java
// Mapa realizovana ArrayListama.
import java.util.*;
import net.mindview.util.*;

public class SporaMapa<K,V> extends AbstractMap<K,V> {
  private List<K> kljucevi = new ArrayList<K>();
  private List<V> vrednosti = new ArrayList<V>();
  public V put(K kljuc, V vrednost) {
    V staraVrednost = get(kljuc); // Stara vrednost ili null
    if(!kljucevi.contains(kljuc)) {
      kljucevi.add(kljuc);
      vrednosti.add(vrednost);
    } else
      vrednosti.set(kljucevi.indexOf(kljuc), vrednost);
    return staraVrednost;
  }
  public V get(Object kljuc) { // kljuc je tipa Object, ne tipa K
    if(!kljucevi.contains(kljuc))
      return null;
    return vrednosti.get(kljucevi.indexOf(kljuc));
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> skup= new HashSet<Map.Entry<K,V>>();
    Iterator<K> ki = kljucevi.iterator();
    Iterator<V> vi = vrednosti.iterator();
    while(ki.hasNext())
      skup.add(new Map.Entry<K,V>(ki.next(), vi.next()));
    return skup;
  }
  public static void main(String[] args) {
    SporaMapa<String,String> m= new SporaMapa<String,String>();
    m.putAll(Countries.glavni_gradovi(15));
    System.out.println(m);
    System.out.println(m.get("BOTSWANA"));
    System.out.println(m.entrySet());
  }
} /* Ispis:
{CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo,
```

```
EQUATORIAL GUINEA=Malabo, EGYPT=Cairo, ANGOLA=Luanda,
BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti}
Gaberone
[CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo,
EQUATORIAL GUINEA=Malabo, EGYPT=Cairo, ANGOLA=Luanda,
BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti]
*///:~
```

Metoda **put( )** jednostavno smešta ključeve i vrednosti u odgovarajuće **ArrayLis**te. U skladu sa interfejsom **Map**, mora da vrati stari ključ ili **null** ako stari ključ **ne** pronađe.

Takođe u skladu sa specifikacijama za interfejs **Map**, metoda **get( )** daje **null** ako se traženi ključ ne pronađe u mapi **SporaMapa**. Ukoliko traženi ključ postoji, upotrebljava se za pronalaženje numeričkog indeksa koji pokazuje njegovo mesto u **Listi kljucevi** i taj broj se koristi kao indeks kojim se proizvodi pridružena vrednost iz **Liste vrednosti**. Obratite pažnju na to da je **kljuc** u metodi **get( )** tipa **Object**, a ne parametrizovanog tipa **K** kao što biste mogli očekivati (i koji je naravno bio upotrebljen u primeru **AsocijativniNiz.java**). Ovo je posledica tako kasnog ubacivanja generičkih tipova u Javu – da su generički tipovi postojali u prvobitnom izdanju jezika, metoda **get( )** mogla bi da specificira tip svog parametra.

Metoda **Map.entrySet( )** mora da proizvede skup objekata tipa **Map.Entry**. Međutim, interfejs **Map.Entry** opisuje strukturu koja se menja u zavisnosti od realizacije, pa ako želite da pravite sopstveni tip **Mape**, morate da definišete i realizaciju od **Map.Entry**:

```
//: kontejneri/StavkaMape.java
// Jednostavan interfejs Map.Entry za primere realizovanja interfejsa
Map.
import java.util.*;

public class StavkaMape<K,V> implements Map.Entry<K,V> {
  private K kljuc;
  private V vrednost;
  public StavkaMape(K kljuc, V vrednost) {
    this.kljuc = kljuc;
    this.vrednost = vrednost;
  }
  public K getKey() { return kljuc; }
  public V getValue() { return vrednost; }
  public V setValue(V v) {
    V rezultat = vrednost;
    vrednost = v;
    return rezultat;
  }
  public int hashCode() {
    return (kljuc==null ? 0 : kljuc.hashCode()) ^
      (vrednost==null ? 0 : vrednost.hashCode());
```

```
  }
  public boolean equals(Object o) {
    if(!(o instanceof StavkaMape)) return false;
    StavkaMape sm = (StavkaMape)o;
    return
      (kljuc == null ?
       sm.getKey() == null : kljuc.equals(sm.getKey())) &&
      (vrednost == null ?
       sm.getValue()== null : vrednost.equals(sm.getValue()));
  }
  public String toString() { return kljuc + "=" + vrednost; }
} ///:~
```

Ovde veoma jednostavna klasa **StavkaMape** čuva i vraća ključeve i vrednosti. To je u metodi **entrySet( )** iskorišćeno za pravljenje skupa parova ključ – vrednost. Obratite pažnju na to da **entrySet( )** upotrebljava **HashSet** za čuvanje parova, pa **StavkaMape** jednostavno koristi metodu **hashCode( )** objekta **kljuc**. Iako je ovo rešenje veoma jednostavno i naizgled funkcioniše u trivijalnom testu u metodi **SporaMapa.main( )**, to nije korektna realizacija zato što se pravi kopija skupova kljucevi i vrednosti. Korektna realizacija skupa **entrySet( )** treba da pruži *uvid* u **Mapu**, a ne da pravi njenu kopiju, i taj uvid će omogućiti modifikaciju prvobitne mape (što kopija ne omogućuje). Priliku da rešite ovaj problem naći ćete u vežbi 16.

Vodite računa o tome da metoda **equals( )** klase **StavkaMape** mora da proverava i ključeve i vrednosti. Značenje metode **hashCode( )** biće opisano uskoro.

Predstavu sadržaja klase **SporaMapa** u obliku znakovnog niza (**String**) automatski pravi metoda **toString( )** definisana u klasi **AbstractMap**.

U metodi **SporaMapa.main( )** učitava se **SporaMapa** i zatim se prikazuje njen sadržaj. Poziv metode **get( )** pokazuje da to funkcioniše.

Vežba 15: (1) Ponovite vežbu 13 uz upotrebu mape **SporaMapa**.

Vežba 16: (7) Primenite testove iz programa **Maps.java** na klasu **SporaMapa** da biste proverili kako funkcioniše. Popravite sve što u klasi **SporaMapa** ne funkcioniše korektno.

Vežba 17: (2) Realizujte ostatak interfejsa **Map** za klasu **SporaMapa**.

Vežba 18: (3) Napravite **SporSkup** po uzoru na mapu **SporaMapa.java**.

Transformisanje ključeva zbog brzine

SporaMapa.java pokazuje da nije teško napraviti nov tip **Map**e. Ali kao što joj ime nagoveštava, **SporaMapa** nije baš brza, pa je verovatno ne bi koristio niko ko ima neku drugu mogućnost. Problem je pronalaženje ključa; ključevi se ne drže u određenom poretku, pa se za njima traga jednostavnom linearnom pretragom. Linearno pretraživanje je najsporiji način pronalaženja.

Razlog za transformisanje ključeva je brzina: zbog toga je pretraživanje brzo. Pošto je brzina pronalaženja ključeva usko grlo, jedno od rešenja problema jeste da se ključevi drže u uređenom poretku i da se zatim pretraživanje obavlja metodom **Collections.binarySearch( )** (u jednoj vežbi proći ćete kroz taj postupak).

Transformisanje ide još korak dalje jer implicira da samo želite da ključ smestite *negde* gde se može brzo pronaći. Najbrža struktura za skladištenje grupe elemenata je niz, pa će on biti upotrebljen za predstavljanje informacija o ključu (rekao sam „informacija o ključu", a ne samog ključa). Budući da je veličina niza nepromenljiva, imamo problem: želimo da smestimo neodređen broj vrednosti u **Mapu**, ali ako je broj ključeva određen veličinom niza, kako ćemo to postići?

Odgovor je da taj niz neće sadržati ključeve. Iz objekta ključa biće izveden broj – indeks u tom nizu. Taj broj je *ključ za heširanje* (engl. *hash code*) koji daje metoda **hashCode( )** (u rečniku računarske nauke nju nazivamo *heš funkcija* ili *funkcija za transformisanje ključeva*), definisana u klasi **Object** i verovatno redefinisana u vašoj klasi.

Da bi se rešio problem niza nepromenljive veličine, više ključeva može dati isti indeks. Dakle, mogu nastati *sudari* (engl. *collisions*). Stoga nije važno koliki je niz; u njemu će biti mesta za ključeve za heširanje svih objekata ključeva.

Zato postupak pronalaženja vrednosti počinje izračunavanjem ključa za heširanje koji se upotrebljava kao indeks niza. Kada biste mogli jemčiti da neće biti sudaranja (što je moguće ako imate nepromenljiv broj vrednosti), onda biste imali *savršenu funkciju za transformisanje ključeva*, ali to je poseban slučaj.[7] U svim drugim slučajevima, sudaranjem se bavi *spoljno nadovezivanje* (engl. *external chaining*): niz ne pokazuje neposredno na vrednost, nego na listu vrednosti. Za tim vrednostima traga se metodom **equals( )** na linearan način. Naravno, taj deo pretraživanja je mnogo sporiji, ali ako je funkcija za transformisanje ključa dobra, u svakom odeljku liste biće tek nekoliko vrednosti. Stoga se umesto pretraživanja cele liste, brzo skače na odeljak u kojem za pronalaženje vrednosti treba uporediti tek nekoliko stavki. To je mnogo brže i zato je kontejner **HashMap** tako brz.

Pošto sada znate osnove transformisanja ključeva, možemo da realizujemo jednostavnu **Mapu** s transformisanim ključevima:

```
//: kontejneri/JednostavnaHashMapa.java
// Primer mape s transformisanim ključevima.
import java.util.*;
import net.mindview.util.*;

public class JednostavnaHashMapa<K,V> extends AbstractMap<K,V> {
  // Izaberite prim broj za veličinu heš tabele,
  // da bi se postigla ravnomerna raspodela:
  static final int VELICINA = 997;
  // Ne možete imati fizički niz generičkih tipova,
  // ali možete svesti naviše na njega:
  @SuppressWarnings("unchecked")
  LinkedList<StavkaMape<K,V>>[] kofe =
    new LinkedList[VELICINA];
  public V put(K kljuc, V vrednost) {
    V staraVrednost = null;
    int indeks = Math.abs(kljuc.hashCode()) % VELICINA;
```

[7] U Javi SE5, savršena funkcija za transformisanje ključeva (heš funkcija) realizovana je u kontejnerima **EnumMap** i **EnumSet**, zato što **enum** definiše nepromenljiv broj instanci. Videti poglavlje *Nabrojani tipovi*.

```
    if(kofe[indeks] == null)
      kofe[indeks] = new LinkedList<StavkaMape<K,V>>();
    LinkedList<StavkaMape<K,V>> kofa = kofe[indeks];
    StavkaMape<K,V> par = new StavkaMape<K,V>(kljuc, vrednost);
    boolean pronadjen = false;
    ListIterator<StavkaMape<K,V>> it = kofa.listIterator();
    while(it.hasNext()) {
      StavkaMape<K,V> iPar = it.next();
      if(iPar.getKey().equals(kljuc)) {
        staraVrednost = iPar.getValue();
        it.set(par); // Zameni staro novim
        pronadjen = true;
        break;
      }
    }
    if(!pronadjen)
      kofe[indeks].add(par);
    return staraVrednost;
  }
  public V get(Object kljuc) {
    int indeks = Math.abs(kljuc.hashCode()) % VELICINA;
    if(kofe[indeks] == null) return null;
    for(StavkaMape<K,V> iPar : kofe[indeks])
      if(iPar.getKey().equals(kljuc))
        return iPar.getValue();
    return null;
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> skup= new HashSet<Map.Entry<K,V>>();
    for(LinkedList<StavkaMape<K,V>> kofa : kofe) {
      if(kofa == null) continue;
      for(StavkaMape<K,V> mpar : kofa)
        skup.add(mpar);
    }
    return skup;
  }
  public static void main(String[] args) {
    JednostavnaHashMapa<String,String> m =
      new JednostavnaHashMapa<String,String>();
    m.putAll(Countries.glavni_gradovi(25));
    System.out.println(m);
    System.out.println(m.get("ERITREA"));
    System.out.println(m.entrySet());
  }
} /* Ispis:
{CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE D'IVOIR
(IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui,
GUINEA=Conakry, BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo,
ANGOLA=Luanda, BURKINA FASO=Ouagadougou, ERITREA=Asmara,
THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
```

```
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, LESOTHO=Maseru, mdANA=Accra,
DJIBOUTI=Dijibouti, ETHIOPIA=Addis Ababa}
Asmara
[CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE D'IVOIR
(IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui,
GUINEA=Conakry, BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo,
ANGOLA=Luanda, BURKINA FASO=Ouagadougou, ERITREA=Asmara,
THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, LESOTHO=Maseru, mdANA=Accra,
DJIBOUTI=Dijibouti, ETHIOPIA=Addis Ababa]
*///:~
```

Pošto „odeljke" heš tabele često nazivaju *kofe*, niz koji u programu predstavlja tabelu nazvan je **kofe**. Da bi se postigla ravnomerna raspodela, broj kofa je obično prim broj.[8] Obratite pažnju na to da se radi o ulančanoj listi **LinkedList** koja automatski razešava sudare: svaka nova stavka se jednostavno dodaje na kraj liste u određenu kofu. Iako Java ne dozvoljava pravljenje niza generičkih objekata, moguće je napraviti *referencu* na takav niz. Ovde je podesno svesti naviše na takav niz, da bi se sprečilo dodatno svođenje u nastavku programa.

Kada – metodom **put( )** – treba obaviti umetanje u mapu, **hashCode( )** se poziva za ključ, a rezultat se pretvara u pozitivan broj. Za uklapanje rezultujućeg broja u niz **kofe** koristi se operator modulo (%) i veličina niza. Ako se za mesto na koje rezultat treba ubaciti dobije **null**, to znači da nema elemenata koji su transformisanjem dospeli na to mesto, pa se pravi nova ulančana lista (**LinkedList**) za čuvanje objekta koji je transformisanjem upravo dospeo na njega. Međutim, normalan postupak se sastoji od toga da se u listi potraže duplikati, pa ako postoje, stara vrednost se smešta u promenljivu **staraVrednost**, a nova vrednost zamenjuje staru. Indikator **pronadjen** pamti da li je pronađen stari par ključ – vrednost, i ako nije, novi par se dodaje na kraj liste.

Metoda **get( )** za vađenje vrednosti pomoću ključa, izračunava indeks u nizu **kofe** na isti način kao **put( )** (time se jemči da ćete dospeti u isti odeljak liste). Ukoliko postoji neka ulančana lista, u njoj se traži vrednost koja odgovara datom ključu.

Imajte u vidu podatak da ova realizacija nije optimizovana za performanse, nego samo pokazuje operacije koje obavlja mapa transformisanih ključeva. Pogledajte izvorni kôd za **java.util.HashMap** ako vas zanima optimizovana realizacija. Takođe, jednostavnosti radi, **JednostavnaHashMapa** metodi **entrySet( )** pristupa jednako kao **SporaMapa**, što je previše pojednostavljeno i ne bi funkcionisalo u **Map**i opšte namene.

Vežba 19: (1) Ponovite vežbu 13 uz upotrebu mape **JednostavnaHashMapa**.

Vežba 20: (3) Prepravite program **JednostavnaHashMapa** tako da prijavljuje sudare i testirajte ga dodavanjem istog skupa podataka dvaput, čime ćete izazvati sudare.

---

[8] Ispostavlja se da prim broj zapravo nije idealan kao veličina kofa za heširanje, i novije realizacije heširanja u Javi upotrebljavaju veličinu jednaku nekom stepenu broja dva (a to je rezultat opsežnog ispitivanja). Deljenje i ostatak od deljenja su najsporije operacije koje obavljaju savremeni procesori. Ako je dužina tabele jednaka nekom stepenu broja dva, umesto deljenja može se upotrebiti maskiranje. Pošto je **get( )** najčešća operacija, deljenje (%) prouzrokuje veliki deo troškova koje eliminiše pristup „stepen broja dva" (ali može da utiče i na neke **hashCode( )** metode).

Vežba 21: (2) Prepravite program **JednostavnaHashMapa** tako da prijavljuje broj „proba" potrebnih kada dođe do sudara. Drugim rečima, koliko puta se mora pozvati **next( )** za **Iterator**e koji prolaze ulančanim listama u potrazi za podudarnim elementima?

Vežba 22: (4) Realizujte metode **clear( )** i **remove( )** za kontejner **JednostavnaHashMapa**.

Vežba 23: (3) Realizujte ostatak interfejsa **Map** za kontejner **JednostavnaHashMapa**.

Vežba 24: (5) Po uzoru na primer u programu **JednostavnaHashMapa.java**, napravite i testirajte **JednostavanHashSkup**.

Vežba 25: (6) Umesto da za svaku kofu upotrebljavate po jedan **ListIterator**, prepravite klasu **StavkaMape** tako da bude samostalna jednostruko ulančana lista (svaka **StavkaMape** treba da ima vezu unapred ka sledećoj klasi **StavkaMape**). Prepravite ostatak koda u programu **JednostavnaHashMapa.java** tako da novi pristup ispravno funkcioniše.

Redefinisanje metode **hashCode( )**

Pošto sada razumete kako radi transformisanje ključeva, ima smisla da i sami napišete metodu **hashCode( )**.

Pre svega, vi ne pravite stvarnu vrednost koja se upotrebljava za indeksiranje niza kofa. Ona se menja u zavisnosti od kapaciteta određenog **HashMap** objekta, a taj kapacitet se menja u zavisnosti od napunjenosti kontejnera i *faktora opterećenja* (objasnićemo taj termin kasnije). Stoga će vrednost koju proizvede vaša metoda **hashCode( )** biti još menjana da bi se napravio indeks niza kofa (u programu **JednostavnaHashMapa**, proračun se svodi na operaciju modulo s veličinom niza kofa).

Pri pravljenju metode **hashCode( )** najvažnije je da ona daje istu vrednost za određeni objekat svaki put kada bude pozvana, bez obzira na to kada je pozvana. Ukoliko dobijete objekat koji pravi jednu **hashCode( )** vrednost kada se metodom **put( )** umeće u **HashMap**u, a drugu kada se metodom **get( )** vadi iz nje, nećete moći da pronalazite i vadite objekte iz mape. Stoga ako se rezultat vaše metode **hashCode( )** bude menjao u zavisnosti od promenljivih podataka u objektu, korisnik mora biti svestan da će izmena tih podataka prouzrokovati pravljenje različitog ključa, jer se generiše drugačiji **hashCode( )**.

Sem toga, **hashCode( )** verovatno *nećete* generisati na osnovu jedinstvenih informacija objekta – konkretno, vrednost koju daje rezervisana reč **this** predstavlja loš **hashCode( )**, zato što u tom slučaju ne možete generisati ključ identičan onome koji je upotrebljen za umetanje (metodom **put( )**) prvobitnog para ključ – vrednost. To je bio problem u programu **DetektorProleca.java**, zato što podrazumevana realizacija metode **hashCode( )** kao ulazni podatak objekta *koristi* njegovu adresu. Zbog toga treba upotrebiti one informacije iz objekta koje ga identifikuju smisleno.

Jedan primer vidi se u klasi **String**. Znakovni nizovi (**String**ovi) imaju posebno obeležje: ukoliko program ima više **String** objekata koji sadrže identične sekvence znakova, onda se svi ti **String** objekti preslikavaju (mapiraju) na isto mesto u memoriji. Stoga ima smisla da funkcije **hashCode( )** proizvedene za dve zasebne instance znakovnog niza **Zdravo** budu identične. To vidite u ovom programu:

```
//: kontejneri/KljučeviZaTransformisanjeZnakovnihNizova.java

public class KljučeviZaTransformisanjeZnakovnihNizova {
  public static void main(String[] args) {
    String[] dvazdravo = "Zdravo Zdravo".split(" ");
    System.out.println(dvazdravo .hashCode());
    System.out.println(dvazdravo .hashCode());
  }
} /* Ispis: (primer)
69609650
69609650
*///:~
```

Očigledno je da se **hashCode( )** za tip **String** izračunava na osnovu sadržaja datog znakovnog niza (objekta tipa **String**).

Dakle, da bi metoda **hashCode( )** bila delotvorna, mora biti brza i smisaona, tj. mora generisati vrednost na osnovu sadržaja objekta. Ne zaboravite da ta vrednost ne mora biti jedinstvena – bavite se brzinom, a ne jedinstvenošću – ali kombinacija metoda **hashCode( )** i **equals( )** mora u potpunosti odrediti identitet objekta.

Pošto se **hashCode( )** dodatno obrađuje pre pretvaranja u indeks kofa, opseg vrednosti rezultata te metode nije važan; dovoljno je da ona generiše ceo broj (**int**).

Ima tu još nešto: dobra metoda **hashCode( )** treba da daje ravnomerno raspodeljene vrednosti. Ako se te vrednosti negde gomilaju, onda će kontejner **HashMap** ili **HashSet** na nekim mestima biti gušće popunjen i neće biti onoliko brz kao što može biti uz ravnomerno raspodeljene rezultate funkcije za heširanje.

U knjizi *Efikasno programiranje na Javi* (Mikro knjiga, 2004), Joshua Bloch daje osnovni recept za generisanje pristojne metode **hashCode( )**:

1. U celobrojnu promenljivu (tipa **int**) nazvanu **rezultat** smestite neki broj različit od nule, recimo 17.
2. Za svako značajno polje **f** u objektu (tj. svako polje koje metoda **equals( )** uzima u obzir) izračunajte celobrojni (**int**) **hashCode( ) c**:

Tip polja	Algoritam za izračunavanje
boolean	**c = (f ? 0 : 1)**
byte, char, short ili **int**	**c = (int)f**
long	**c = (int)(f ^ (f >>>32))**
float	**c = Float.floatToIntBits(f);**
double	**long l = Double.doubleToLongBits(f);** **c = (int)(l ^ (l >>> 32))**
Object, gde metoda **equals( )** poziva **equals( )** za to polje	**c = f.hashCode( )**
Niz	Na svaki element primenite gornja pravila

3. Ksombinujte goreizračunate ključeve za transformisanje:
   rezultat = 37 * **rezultat** + **c**;
4. Vratite **rezultat**.
5. Pogledajte dobijenu metodu **hashCode**( ) i postarajte se da jednake instance imaju jednake ključeve za transformisanje.

Naredni primer je napisan po prethodnim smernicama:

```
//: kontejneri/PrebrojanZnakovniNiz.java
// Pravljenje dobre metode hashCode().
import java.util.*;
import static net.mindview.util.Print.*;

public class PrebrojanZnakovniNiz {
  private static List<String> napravljeno =
    new ArrayList<String>();
  private String s;
  private int id = 0;
  public PrebrojanZnakovniNiz(String znn) {
    s = znn;
    napravljeno.add(s);
    // id je ukupan broj instanci datog znakovnog
    // niza koje koristi PrebrojanZnakovniNiz:
    for(String s2 : napravljeno)
      if(s2.equals(s))
        id++;
  }
  public String toString() {
    return "Znakovni niz: " + s + " id: " + id +
      " hashCode(): " + hashCode();
  }
  public int hashCode() {
    // Veoma jednostavan pristup:
    // return s.hashCode() * id;
    // Upotrebićemo recept Joshue Blocha:
    int rezultat = 17;
    rezultat = 37 * rezultat + s.hashCode();
    rezultat = 37 * rezultat + id;
    return rezultat;
  }
  public boolean equals(Object o) {
    return o instanceof PrebrojanZnakovniNiz &&
      s.equals(((PrebrojanZnakovniNiz)o).s) &&
      id == ((PrebrojanZnakovniNiz)o).id;
  }
  public static void main(String[] args) {
    Map<PrebrojanZnakovniNiz,Integer> mapa =
      new HashMap<PrebrojanZnakovniNiz,Integer>();
    PrebrojanZnakovniNiz[] pzn = new PrebrojanZnakovniNiz;
    for(int i = 0; i < pzn.duzina; i++) {
```

```
      pzn[i] = new PrebrojanZnakovniNiz("zdravo");
      mapa.put(pzn[i], i); // Automatsko pakovanje int -> Integer
    }
    print(mapa);
    for(PrebrojanZnakovniNiz pznakovniniz : pzn) {
      print("Tražim " + pznakovniniz);
      print(mapa.get(pznakovniniz));
    }
  }
} /* Ispis: (primer)
{String: zdravo id: 4 hashCode(): 146450=3, String: zdravo id: 1
hashCode(): 146447=0, String: zdravo id: 3 hashCode(): 146449=2,
String: zdravo id: 5 hashCode(): 146451=4, String: zdravo id:
2 hashCode(): 146448=1}
Tražim Znakovni niz: zdravo id: 1 hashCode(): 146447
0
Tražim Znakovni niz: zdravo id: 2 hashCode(): 146448
1
Tražim Znakovni niz: zdravo id: 3 hashCode(): 146449
2
Tražim Znakovni niz: zdravo id: 4 hashCode(): 146450
3
Tražim Znakovni niz: zdravo id: 5 hashCode(): 146451
4
*///:~
```

PrebrojanZnakovniNiz sadrži jedan znakovni niz (**String**) i jedan **id** koji predstavlja broj objekata tipa **PrebrojanZnakovniNiz** koji sadrže identičan znakovni niz. Prebrojavanje se obavlja u konstruktoru, iteriranjem kroz statičnu listu **ArrayList** u kojoj su uskladišteni svi znakovni nizovi.

Obe metode, i **hashCode( )** i **equals( )**, proizvode rezultate na osnovu oba polja; da uzimaju u obzir samo objekat tipa **String** ili samo **id**, dešavalo bi se da različite vrednosti dobiju duplikate istog ključa.

U metodi **main( )**, pomoću istog znakovnog niza napravljeno je više objekata tipa **PrebrojanZnakovniNiz**, kako bi se pokazalo da duplikati prave jedinstvene vrednosti zbog uzimanja u obzir **id**(entifikatora) broja duplikata. Prikazana je i **HashMap**a da biste videli kakav je njen unutrašnji poredak (ne može se uočiti nikakva pravilnost), a zatim se svaki ključ traži pojedinačno, kako bi se pokazalo da mehanizam pronalaženja radi kako treba.

Kao drugi primer, razmotrićemo klasu **Jedinka** koja je bila upotrebljena kao osnovna klasa biblioteke **podaciotipu.ljubimci** definisane u poglavlju *Podaci o tipu*. Klasa **Jedinka** je bila upotrebljena u tom poglavlju, ali je njena definicija odložena do ovoga da biste shvatili realizaciju:

```
//: podaciotipu/ljubimci/Jedinka.java
package podaciotipu.ljubimci;

public class Jedinka implements Comparable<Jedinka> {
  private static long brojac = 0;
  private final long id = brojac++;
```

```
  private String ime;
  public Jedinka(String ime) { this.ime = ime; }
  // 'ime' is opciono:
  public Jedinka() {}
  public String toString() {
    return getClass().getSimpleName() +
      (ime == null ? "" : " " + ime);
  }
  public long id() { return id; }
  public boolean equals(Object o) {
    return o instanceof Jedinka &&
      id == ((Jedinka)o).id;
  }
  public int hashCode() {
    int rezultat = 17;
    if(ime != null)
      rezultat = 37 * rezultat + ime.hashCode();
    rezultat = 37 * rezultat + (int)id;
    return rezultat;
  }
  public int compareTo(Jedinka arg) {
    // Prvo se porede imena klasa:
    String prvo = getClass().getSimpleName();
    String argPrvi = arg.getClass().getSimpleName();
    int prvoPoredjenje = prvo.compareTo(argPrvi);
    if(prvoPoredjenje != 0)
    return prvoPoredjenje;
    if(ime != null && arg.ime != null) {
      int drugoPoredjenje = ime.compareTo(arg.ime);
      if(drugoPoredjenje != 0)
        return drugoPoredjenje;
    }
    return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
  }
} ///:~
```

Metoda **compareTo( )** ima hijerarhiju poređenja, tako da proizvodi sekvencu uređenu prvo po stvarnom tipu, zatim po imenu ako ono postoji, i najzad po redosledu pravljenja. U ovom primeru videćete kako sve to radi:

```
//: kontejneri/TestiranjeJedinki.java
import holding.MapOfList;
import podaciotipu.ljubimci.*;
import java.util.*;

public class TestiranjeJedinki {
  public static void main(String[] args) {
    Set<Jedinka> ljubimci = new TreeSet<Jedinka>();
    for(List<? extends LJubimac> llj :
```

```
        MapOfList.ljubiteljiLjubimaca.vrednosti())
      for(LJubimac lj : llj)
        ljubimci.add(lj);
    System.out.println(ljubimci);
  }
} /* Ispis:
[Cat Elsie May, Cat Pinkola, Cat Shackleton, Cat Stanford aka
Stinky el Negro, Cymric Molly, Dog Margrett, Mutt Spot, Pug Louie
aka Louis Snorkelstein Dupree, Rat Fizzy, Rat Freckly, Rat Fuzzy
*///:~
```

Pošto svi ovi ljubimci imaju imena, uređuje se prvo po tipu, zatim po imenu i najzad po imenu unutar njihovog tipa.

Napisati odgovarajuće metode **hashCode( )** i **equals( )** za novu klasu ume da bude zapetljano. Alatke koje će vam pri tome pomoći pronaći ćete u Apache projektu „Jakarta Commons" na adresi *jakarta.apache.org/commons*, pod „lang" (ovaj projekat ima i mnogo drugih potencijalno korisnih biblioteka, pa izgleda da je to odgovor Javine zajednice na lokaciju *www.boost.org* zajednice korisnika C++-a).

Vežba 26: (2) U program **PrebrojanZnakovniNiz** dodajte **char** polje koje se takođe inicijalizuje u konstruktoru i prepravite metode **hashCode( )** i **equals( )** tako da obuhvate vrednost ovog polja.

Vežba 27: (3) Prepravite metodu **hashCode( )** u programu **PrebrojanZnakovniNiz.java** tako što ćete ukloniti kombinaciju sa **id**, i pokažite da **PrebrojanZnakovniNiz** i dalje radi kao ključ. Zašto ovaj pristup ne valja?

Vežba 28: (4) Dodavanjem metoda **hashCode( )**, **equals( )** i realizovanjem interfejsa **Comparable** za svaki tip **N_torke** prepravite **net/mindview/util/N_torka.java** tako da postane klasa opšte namene.

Izbor realizacije

Dosad je trebalo da shvatite sledeće: iako postoje samo četiri osnovna tipa kontejnera – **Map**, **List**, **Set** i **Queue** – svaki od tih interfejsa ima više realizacija. Ako vam treba funkcionalnost određenog interfejsa, kako da izaberete njegovu realizaciju?

Svaka realizacija ima svoja obeležja, prednosti i nedostatke. Na primer, na slici s početka ovog poglavlja vidite da je obeležje klasa **Hashtable**, **Vector** i **Stack** to što su nasleđene iz prethodnih verzija Jave (engl. *legacy*), kako se stari kôd ne bi rušio zbog prevođenja novim prevodiocima (ali ih ne treba koristiti u novim programima).

Vrste redova za čekanje u (**Queues**) u Javinim bibliotekama razlikuju se samo po načinu primanja i vraćanja vrednosti (važnost toga objasnićemo u poglavlju *Paralelno izvršavanje*).

Razlika između kontejnera često se svodi na to kakvi su „u pozadini" – tj. koje strukture podataka fizički realizuju željeni interfejs. Na primer, pošto **ArrayList** i **LinkedList** realizuju interfejs **List**, *elementarne* **List** operacije su iste bez obzira na to koju od njih upotrebite. Međutim, u pozadini **ArrayList**e je niz, dok je **LinkedList** realizovana na način uobičajen za dvostruko ulančanu listu, kao grupa pojedinačnih objekata koji sadrže

podatke i reference na prethodni i sledeći element u listi. Upravo zato, ako nameravate da mnogo umećete i uklanjate elemente usred liste, koristite **LinkedList**. (**LinkedList** ima i dodatnu funkcionalnost, utvrđenu u listi **AbstractSequentialList**.) Ali ako vam to nije potrebno, uzmite listu **ArrayList** koja je obično brža.

Kao drugi primer, **Set** se može realizovati kao **TreeSet**, **HashSet** ili **LinkedHashSet**.[9] Ponašanje svakog od tih skupova je drugačije: **HashSet** je podesan za tipičnu upotrebu i najbrže obavlja pretraživanje, **LinkedHashSet** čuva parove u poretku umetanja, a u pozadini skupa **TreeSet** nalazi se mapa **TreeMap** koja uvek daje uređen skup. Realizaciju birate u skladu s ponašanjem koje vam je potrebno.

Ponekad različite realizacije određenog kontejnera imaju neke zajedničke operacije, ali su performanse tih operacija različite. U tom slučaju, realizaciju birate na osnovu toga koliko često koristite određenu operaciju i koliko brza ona mora da bude. U takvim slučajevima se o razlikama između realizacija kontejnera može zaključivati i kroz ispitivanje performansi.

Struktura za ispitivanje performansi

Napravio sam strukturu (engl. *framework*) za ispitivanje u koju sam stavio osnovne funkcije za postupak ispitivanja, da bih izbegao dupliranje koda i obezbedio ujednačenost ispitivanja. Sledeći program pravi osnovnu klasu od koje možete napraviti listu anonimnih unutrašnjih klasa, po jednu za svako ispitivanje. Postupak ispitivanja se sastoji od pozivanja svake od tih unutrašnjih klasa. Time je olakšano dodavanje i uklanjanje novih vrsta ispitivanja.

Ovo je još jedan primer projektnog obrasca *Template Method* (Šablonska metoda). Iako redefinisanjem metode **Test.ispit( )** za svako različito ispitivanje sledimo tipičan pristup *Template Method*, u ovom slučaju je osnovni kod (onaj koji se ne menja) u zasebnoj klasi **Tester**.[10] Tip kontejnera koji se ispituje opisan je generičkim parametrom C:

```
//: kontejneri/Test.java
// Struktura za vremensko ispitivanje kontejnera.

public abstract class Test<C> {
  String ime;
  public Test(String ime) { this.ime = ime; }
  // Redefinišite ovu metodu za svaku vrstu ispitivanja.
  // Vraća stvarni broj ponavljanja ispitivanja.
  abstract int ispit(C kontejner, ParamIspita pi);
} ///:~
```

Svaki objekat tipa **Test** skladišti ime tog testa (ispitivanja). Kada pozovete metodu **ispit( )**, morate joj dati kontejner koji treba ispitati i „prenosioca" ili „objekat za prenos podataka" koji sadrži sve parametre tog ispitivanja. To su parametri **velicina**, jednak bro-

[9] Ili kao **EnumSet** odnosno **CopyOnWriteArraySet**, što su specijalni slučajevi. Ne poričem da postoje druge specijalizovane realizacije raznih kontejnerskih interfejsa, ali u ovom odeljku pokušavam da sagledam opštiju sliku.

[10] U smišljanju generičkih tipova podesnih za ovo poglavlje pomogao mi je Krzysztof Sobolewski.

ju elemenata u kontejneru, i **petlje**, koji određuje broj iteracija za to ispitivanje. Ti parametri se mogu koristiti u svakom ispitivanju ali i ne moraju.

Svaki kontejner će biti podvrgnut nizu poziva metode **ispit( )**, svaki put s različitim parametrima ispitivanja (objektom tipa **ParamIspita**), pa **ParamIspita** sadrži i dve statične metode **niz( )** za lako pravljenje nizova **ParamIspita** objekata. Prva verzija metode **niz( )** prima listu promenljivih argumenata koja naizmenično sadrži vrednosti **velicina** i **petlji**, a druga verzija prima listu iste vrste, sem što su vrednosti smeštene u znakovne nizove (objekte tipa **String**) – pa se može upotrebljavati za raščlanjivanje argumenata na komandnoj liniji:

```
//: kontejneri/ParamIspita.java
// "Objekat za prenos podataka".

public class ParamIspita {
  public final int velicina;
  public final int petlje;
  public ParamIspita(int velicina, int petlje) {
    this.velicina = velicina;
    this.petlje = petlje;
  }
  // Pravljenje niza objekata tipa ParamIspita od sekvence promenljivih
  // argumenata:
  public static ParamIspita[] niz(int... vrednosti) {
    int velicina = vrednosti.length/2;
    ParamIspita[] rezultat = new ParamIspita[velicina];
    int n = 0;
    for(int i = 0; i < velicina; i++)
      rezultat[i] = new ParamIspita(vrednosti[n++], vrednosti[n++]);
    return rezultat;
  }
  // Pretvori niz objekata tipa String u niz objekata tipa ParamIspita:
  public static ParamIspita[] niz(String[] vrednosti) {
    int[] vrdnsti = new int[vrednosti.length];
    for(int i = 0; i < vrdnsti.length; i++)
      vrdnsti[i] = Integer.decode(vrednosti[i]);
    return niz(vrdnsti);
  }
} ///:~
```

Strukturu za ispitivanje koristite tako što određenoj metodi **Tester.pokreni( )** (to su prigodne preklopljene generičke metode zahvaljujući kojima ne morate toliko da kucate) prosledite kontejner koji treba ispitati i **Listu Test** objekata. **Tester.pokreni( )** poziva odgovarajući preklopljeni konstruktor, a zatim poziva metodu **vremenskoIspit( )** koja izvršava svako ispitivanje navedeno u listi za taj kontejner. **vremenskoIspit( )** ponavlja svako ispitivanje za svaki od **ParamIspita** objekata u listi **listaParam**. Pošto se **listaParam** inicijalizuje pomoću statičnog niza **podrazumevaniParam**, zadavanjem novih vrednosti niza **podrazumevaniParam** možete da promenite parametre **listaParam** za sva ispiti-

vanja, a za jedno ispitivanje parametre **listaParam** možete da promenite prosleđivanjem liste **listaParam** prilagođene za to ispitivanje:

```
//: kontejneri/Tester.java
// Primenjuje Test objekte na liste različitih kontejnera.
import java.util.*;

public class Tester<C> {
  public static int sirinaPolja = 8;
  public static ParamIspita[] podrazumevaniParam= ParamIspita.niz(
    10, 5000, 100, 5000, 1000, 5000, 10000, 500);
  // Redefinišite ovo da biste modifikovali inicijalizaciju
  // pre ispitivanja:
  protected C inicijalizuj(int velicina) { return kontejner; }
  protected C kontejner;
  private String naslov = "";
  private List<Test<C>> ispitivanja;
  private static String poljeString() {
    return "%" + sirinaPolja + "s";
  }
  private static String poljeBroj() {
    return "%" + sirinaPolja + "d";
  }
  private static int sirinaVelicine = 5;
  private static String poljeVelicine = "%" + sirinaVelicine + "s";
  private ParamIspita[] listaParam = podrazumevaniParam;
  public Tester(C kontejner, List<Test<C>> ispitivanja) {
    this.kontejner = kontejner;
    this.ispitivanja = ispitivanja;
    if(kontejner != null)
      naslov = kontejner.getClass().getSimpleName();
  }
  public Tester(C kontejner, List<Test<C>> ispitivanja,
      ParamIspita[] listaParam) {
    this(kontejner, ispitivanja);
    this.listaParam = listaParam;
  }
  public void zadajNaslov(String novNaslov) {
    naslov = novNaslov;
  }
  // Pomoćne generičke metode :
  public static <C> void pokreni(C kntnr, List<Test<C>> ispitivanja){
    new Tester<C>(kntnr, ispitivanja).vremenskoIspit();
  }
  public static <C> void pokreni(C kntnr,
      List<Test<C>> ispitivanja, ParamIspita[] listaParam) {
    new Tester<C>(kntnr, ispitivanja, listaParam).vremenskoIspit();
  }
```

```
  private void prikaziZaglavlje() {
    // Izračunaj širinu i dopuni znacima '-':
    int sirina = sirinaPolja * ispitivanja.size() + sirinaVelicine;
    int duzinaCrtica = sirina - naslov.length() - 1;
    StringBuilder zglavlje = new StringBuilder(sirina);
    for(int i = 0; i < duzinaCrtica/2; i++)
      zglavlje.append('-');
    zglavlje.append(' ');
    zglavlje.append(naslov);
    zglavlje.append(' ');
    for(int i = 0; i < duzinaCrtica/2; i++)
      zglavlje.append('-');
    System.out.println(zglavlje);
    // Ispisivanje zaglavlja kolona:
    System.out.format(poljeVelicine, "vel.");
    for(Test ispit : ispitivanja)
      System.out.format(poljeString(), ispit.ime);
    System.out.println();
  }
  // Pokreni ispitivanja za ovaj kontejner:
  public void vremenskoIspit() {
    prikaziZaglavlje();
    for(ParamIspita param : listaParam) {
      System.out.format(poljeVelicine, param.velicina);
      for(Test<C> ispit : ispitivanja) {
        C kkontejner = inicijalizuj(param.velicina);
        long start = System.nanoTime();
        // Poziv redefinisane metode:
        int ponavljanja = ispit.ispit(kkontejner, param);
        long trajanje = System.nanoTime() - start;
        long vremeZaJednoPonavljanje = trajanje / ponavljanja;
                                              // Nanosekunde
        System.out.format(poljeBroj(), vremeZaJednoPonavljanje);
      }
      System.out.println();
    }
  }
} ///:~
```

Metode **poljeString( )** i **poljeBroj( )** proizvode znakovne nizove za formatiranje rezultata prilikom ispisa. Standardnu širinu za formatiranje menjate modifikovanjem statične vrednosti **sirinaPolja**. Metoda **prikaziZaglavlje( )** formatira i ispisuje zaglavlje s podacima iz svakog ispitivanja.

Ako vam je potrebna specijalna inicijalizacija, redefinišite metodu **inicijalizuj( )**. Ona proizvodi inicijalizovan objekat kontejner odgovarajuće veličine – možete modifikovati postojeći objekat kontejner ili napraviti novi. U metodi **ispit( )** vidite da se rezultat hvata u lokalnoj referenci nazvanoj **kontejner**, što omogućuje da uskladišteni član **kontejner** zamenite potpuno drugačije inicijalizovanim kontejnerom.

Povratna vrednost svake metode **Test.ispit( )** mora biti broj operacija obavljenih tokom tog ispitivanja, što se koristi za izračunavanje broja nanosekundi potrebnih za svaku operaciju. Imajte u vidu da metoda **System.nanoTime( )** obično daje vrednosti čija granularnost premašuje jedan (i menja se u zavisnosti od računara i operativnog sistema), a to će prouzrokovati određenu količinu šuma u rezultatima.

Rezultati se mogu menjati u zavisnosti od računara na kojem se ispitivanja obavljaju; ova ispitivanja su namenjena samo za relativno poređenje performansi različitih kontejnera, a ne za apsolutno merenje tih performansi.

Performanse različitih **Lista**

Evo ispitivanja performansi osnovnih operacija u listama. Poređenja radi, prikazane su i najvažnije operacije u redovima za čekanje (**Queue**). Za ispitivanje svake klase kontejnera napravljene su dve zasebne liste testova. U ovom slučaju, operacije u redovima za čekanje odnose se samo na ulančane liste (**LinkedList**).

```
//: kontejneri/PerformanseLista.java
// Pokazuje razlike u performansama Lista.
// {Argumenata: 100 500} Malo, da bi ispitivanje builda bilo kratko
import java.util.*;
import net.mindview.util.*;

public class PerformanseLista {
  static Random slucajan = new Random();
  static int ponavljanja = 1000;
  static List<Test<List<Integer>>> ispitivanja =
    new ArrayList<Test<List<Integer>>>();
  static List<Test<LinkedList<Integer>>> ispitivanjaRedaZaCekanje =
    new ArrayList<Test<LinkedList<Integer>>>();
  static {
    ispitivanja.add(new Test<List<Integer>>("add") {
      int ispit(List<Integer> lista, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicinaListe = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          lista.clear();
          for(int j = 0; j < velicinaListe; j++)
            lista.add(j);
        }
        return petlje * velicinaListe;
      }
    });
    ispitivanja.add(new Test<List<Integer>>("get") {
      int ispit(List<Integer> lista, ParamIspita pi) {
        int petlje = pi.petlje * ponavljanja;
        int velicinaListe = lista.size();
        for(int i = 0; i < petlje; i++)
```

```
        lista.get(slucajan.nextInt(velicinaListe));
      return petlje;
    }
  });
  ispitivanja.add(new Test<List<Integer>>("set") {
    int ispit(List<Integer> lista, ParamIspita pi) {
      int petlje = pi.petlje * ponavljanja;
      int velicinaListe = lista.size();
      for(int i = 0; i < petlje; i++)
        lista.set(slucajan.nextInt(velicinaListe), 47);
      return petlje;
    }
  });
  ispitivanja.add(new Test<List<Integer>>("iteradd") {
    int ispit(List<Integer> lista, ParamIspita pi) {
      final int PETLJE = 1000000;
      int polovina = lista.size() / 2;
      ListIterator<Integer> it = lista.listIterator(polovina);
      for(int i = 0; i < PETLJE; i++)
        it.add(47);
      return PETLJE;
    }
  });
  ispitivanja.add(new Test<List<Integer>>("insert") {
    int ispit(List<Integer> lista, ParamIspita pi) {
      int petlje = pi.petlje;
      for(int i = 0; i < petlje; i++)
        lista.add(5, 47); // Minimizovanje troškova
                          // nasumičnog pristupanja
      return petlje;
    }
  });
  ispitivanja.add(new Test<List<Integer>>("remove") {
    int ispit(List<Integer> lista, ParamIspita pi) {
      int petlje = pi.petlje;
      int velicina = pi.velicina;
      for(int i = 0; i < petlje; i++) {
        lista.clear();
        lista.addAll(new BrojackaListaIntegera(velicina));
        while(lista.size() > 5)
          lista.remove(5); // Minimizovanje troškova
                           // nasumičnog pristupanja
      }
      return petlje * velicina;
    }
  });
  // Ispitivanja ponašanja reda za čekanje:
  ispitivanjaRedaZaCekanje.add(new Test<LinkedList<
                               Integer>>("addFirst") {
```

```
      int ispit(LinkedList<Integer> lista, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicina = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          lista.clear();
          for(int j = 0; j < velicina; j++)
            lista.addFirst(47);
        }
        return petlje * velicina;
      }
    });
    ispitivanjaRedaZaCekanje.add(new Test<LinkedList<
                                 Integer>>("addLast") {
      int ispit(LinkedList<Integer> lista, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicina = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          lista.clear();
          for(int j = 0; j < velicina; j++)
            lista.addLast(47);
        }
        return petlje * velicina;
      }
    });
    ispitivanjaRedaZaCekanje.add(
      new Test<LinkedList<Integer>>("rmFirst") {
        int ispit(LinkedList<Integer> lista, ParamIspita pi) {
          int petlje = pi.petlje;
          int velicina = pi.velicina;
          for(int i = 0; i < petlje; i++) {
            lista.clear();
            lista.addAll(new BrojackaListaIntegera(velicina));
            while(lista.size() > 0)
              lista.removeFirst();
          }
          return petlje * velicina;
        }
      });
    ispitivanjaRedaZaCekanje.add(new Test<LinkedList<Inte-
ger>>("rmLast") {
      int ispit(LinkedList<Integer> lista, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicina = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          lista.clear();
          lista.addAll(new BrojackaListaIntegera(velicina));
          while(lista.size() > 0)
            lista.removeLast();
```

```
        }
        return petlje * velicina;
      }
    });
  }
  static class IspitivacLista extends Tester<List<Integer>> {
    public IspitivacLista(List<Integer> kontejner,
        List<Test<List<Integer>>> ispitivanja) {
      super(kontejner, ispitivanja);
    }
    // Pre svakog ispitivanja popuni do odgovarajuće veličine:
    @Override protected List<Integer> inicijalizuj(int velicina){
      kontejner.clear();
      kontejner.addAll(new BrojackaListaIntegera(velicina));
      return kontejner;
    }
    // Pomočna metoda:
    public static void pokreni(List<Integer> lista,
        List<Test<List<Integer>>> ispitivanja) {
      new IspitivacListi(lista, ispitivanja).vremenskoIspit();
    }
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.podrazumevaniParam = ParamIspita.niz(args);
    // Na niz se mogu primeniti samo sledeća dva ispitivanja:
    Tester<List<Integer>> ispitNiza =
      new Tester<List<Integer>>(null, ispitivanja.podLista(1, 3)){
        // Ovo će biti pozvano pre svakog ispitivanja.
        // Proizvodi listu nepromenljive veličine s nizom u pozadini:
        @Override protected
        List<Integer> inicijalizuj(int velicina) {
          Integer[] ia = Generated.niz(Integer.class,
            new CountingGenerator.Integer(), size);
          return Arrays.asList(ia);
        }
      };
    ispitNiza.zadajNaslov("Niz kao lista");
    ispitNiza.vremenskoIspit();
    Tester.podrazumevaniParam= ParamIspita.niz(
      10, 5000, 100, 5000, 1000, 1000, 10000, 200);
    if(args.length > 0)
      Tester.podrazumevaniParam = ParamIspita.niz(args);
    IspitivacLista.pokreni(new ArrayList<Integer>(), ispitivanja);
    IspitivacLista.pokreni(new LinkedList<Integer>(), ispitivanja);
    IspitivacLista.pokreni(new Vector<Integer>(), ispitivanja);
    Tester.sirinaPolja = 12;
    Tester<LinkedList<Integer>> ispitivanjeRedaZaCekanje =
```

```
      new Tester<LinkedList<Integer>>(
        new LinkedList<Integer>(), ispitivanjaRedaZaCekanje);
    ispitivanjeRedaZaCekanje.zadajNaslov("Ispitivanja reda za čekanje");
    ispitivanjeRedaZaCekanje.vremenskoIspit();
  }
} /* Ispis: (primer)
--- Niz kao lista ---
 vel.     get     set
   10     130     183
  100     130     164
 1000     129     165
10000     129     165
--------------------- ArrayList ---------------------
 vel.     add     get     set  iteradd  insert  remove
   10     121     139     191     435    3952     446
  100      72     141     191     247    3934     296
 1000      98     141     194     839    2202     923
10000     122     144     190    6880   14042    7333
--------------------- LinkedList ---------------------
 vel.     add     get     set  iteradd  insert  remove
   10     182     164     198     658     366     262
  100     106     202     230     457     108     201
 1000     133    1289    1353     430     136     239
10000     172   13648   13187     435     255     239
---------------------- Vector ----------------------
 vel.     add     get     set  iteradd  insert  remove
   10     129     145     187     290    3635     253
  100      72     144     190     263    3691     292
 1000      99     145     193     846    2162     927
10000     108     145     186    6871   14730    7135
------------ Ispitivanja reda za čekanje ------------
 vel.    addFirst     addLast     rmFirst      rmLast
   10         199         163         251         253
  100          98          92         180         179
 1000          99          93         216         212
10000         111         109         262         384
*///:~
```

O svakom ispitivanju treba dobro razmisliti, da ne biste dobijali besmislene rezultate. Na primer, ispitivanje operacije **add** briše **List**u i zatim je popunjava do zadate veličine liste. Zato je poziv metode **clear( )** deo ispitivanja i može uticati na izmereno vreme, naročito u malim ispitivanjima. Iako prethodni rezultati izgledaju prilično razumno, struktura za ispitivanje bi se mogla preraditi tako da poziv pripremne metode – što bi, u ovom slučaju, obuhvatilo i poziv metode **clear( )** – bude *izvan* petlje u kojoj se meri vreme.

Vodite računa o tome da svako ispitivanje mora tačno izračunati broj obavljenih operacija i vratiti tu vrednost iz metode **ispit( )**, da bi merenje vremena bilo tačno.

U ispitivanjima operacija **get** i **set** upotrebljava se generator slučajnih brojeva za nasumično pristupanje **Listi**. Iz ispisa vidite da su za **List**u napravljenu na osnovu niza i za **ArrayList** ta pristupanja brza i ujednačenog trajanja bez obzira na veličinu liste, dok za ulančanu listu (**LinkedList**) trajanje pristupanja znatno raste za veće liste. Očigledno je da ulančane liste nisu dobro rešenje ukoliko nameravate da obavljate mnogo nasumičnih pristupanja.

U ispitivanju operacije **iteradd**, za umetanje novih elemenata upotrebljen je iterator u sredini liste. Za kontejner **ArrayList** to postaje skupo kako lista raste, ali za **LinkedList** relativno je jeftino, a i konstantnog trajanja bez obzira na veličinu. To ima smisla zato što **ArrayList** tokom umetanja mora da napravi prostor i kopira sve svoje reference od tog mesta unapred. To postaje skupo kako **ArrayList** postaje sve veća. Objekat tipa **LinkedList** treba samo da poveže nov element i ne mora da menja ostatak liste, pa bi režijski troškovi te operacije trebalo da budu približno jednaki, bez obzira na veličinu liste.

U ispitivanju operacija **insert** i **remove** kao mesto umetanja odnosno uklanjanja koristi se lokacija broj 5, a ne neki kraj **List**e. Ulančana lista (**LinkedList**) ima poseban algoritam za krajnje lokacije **List**e – time se povećava brzina kada se ulančana lista upotrebljava kao red za čekanje (**Queue**). Međutim, ukoliko elemente dodajete i uklanjate u sredini liste, prouzrokujete troškove nasumičnog pristupanja koji se razlikuju za razne realizacije **List**a (kao što ste videli). Pošto su umetanja i uklanjanja na istoj lokaciji broj 5, trebalo bi da su troškovi nasumičnog pristupanja zanemarivi i da vidimo samo troškove umetanja i uklanjanja, ali i da ne vidimo bilo kakvu posebnu optimizaciju za krajeve ulančane liste (**LinkedList**). Iz rezultata vidite da su troškovi umetanja i uklanjanja iz ulančane liste veoma mali i da se ne menjaju u zavisnosti od veličine liste, ali su troškovi (naročito umetanja) u objekat tipa **ArrayList** *veoma* veliki i povećavaju se s veličinom liste.

Iz ispitivanja redova za čekanje (objekata tipa **Queue**) vidite koliko brzo ulančana lista umeće i uklanja elemente s krajnjih tačaka liste, što je optimalno za ponašanje redova za čekanje.

Obično je dovoljno samo pozvati metodu **Tester.pokreni**( ) i proslediti joj kontejner i spisak **ispitivanja**. Međutim, ovde moramo redefinisati metodu **inicijalizuj**( ) da bi **List**a bila obrisana i ponovo popunjena pre svakog ispitivanja – inače bi **List**a tokom raznih ispitivanja izgubila kontrolu nad svojom veličinom. **IspitivacLista** nasleđuje klasu **Tester** i obavlja inicijalizaciju uz pomoć objekta tipa **BrojackaListaIntegera**. Redefinisana je i pomoćna metoda **pokreni**( ).

Želeli bismo i da uporedimo pristupanje nizovima s pristupanjem kontejnerima (prvenstveno kontejnerima tipa **ArrayList**). U prvom ispitivanju unutar metode **main**( ), pomoću anonimne unutrašnje klase napravljen je poseban objekat tipa **Test**. Metoda **inicijalizuj**( ) tako je redefinisana da pravi nov objekat svaki put kada bude pozvana (zanemaruje uskladišteni objekat **kontejner**, pa je za ovaj konstruktor klase **Tester** argument **kontejner** jednak **null**). Novi objekat se pravi metodama **Generated.niz**( ) (definisanom u poglavlju *Nizovi*) odnosno **Arrays.asList**( ). U ovom slučaju mogu se obaviti samo dva ispitivanja, zato što elemente ne možete umetati u **List**u napravljenu na osnovu niza niti ih uklanjati, pa je za izbor ispitivanja iz spiska **ispitivanja** upotrebljena metoda **List.podLista**( ).

Za operacije nasumičnog (nesekvencijalnog) pristupanja **get( )** i **set( )**, **List**a napravljena na osnovu niza neznatno je brža od kontejnera tipa **ArrayList**, ali su iste te operacije mnogo skuplje kada se koriste ulančane liste, pošto klasa **LinkedList** nije namenjena za operacije nasumičnog (nesekvencijalnog) pristupanja.

Klasu **Vector** treba izbegavati; ona je u biblioteci samo da bi se obezbedila podrška za stari kôd (u ovom programu deluje samo zato što je radi kompatibilnosti s novim kodom prerađena tako da postane **List**a).

CopyOnWriteArrayList je posebna realizacija klase **List** koja se upotrebljava u programiranju za paralelno izvršavanje. Nju ćemo razmotriti u poglavlju *Paralelno izvršavanje*.Verovatno je najbolja politika uzeti **ArrayList** za podrazumevani tip kontejnera i preći na **LinkedList** ako je potrebna dodatna funkcionalnost ili se ispostavi da mnogo umetanja u sredinu liste i uklanjanja iz nje oslabljuje performanse. Ako radite s grupom elemenata nepromenljive veličine, upotrebite **List**u – metodom **Arrays.asList( )** – napravljenu na osnovu niza ili pravi niz, ako je potrebno.

Vežba 29: (2) Izmenite program **PerformanseLista.java** tako da **List**e umesto **Integer**a skladište objekte tipa **String**. Za pravljenje niza ispitnih vrednosti upotrebite **Generator** iz poglavlja *Nizovi*.

Vežba 30: (3) Uporedite performanse operacije **Collections.sort( )** u kontejnerima tipa **ArrayList** odnosno **LinkedList**.

Vežba 31: (5) Napravite kontejner koji kapsulira niz objekata tipa **String** i dozvoljava samo dodavanje i vađenje znakovnih nizova, pa se tokom upotrebe ne postavlja pitanje svođenja na druge tipove. Ako unutrašnji niz nije dovoljno velik za sledeće dodavanje, kontejner treba da automatski prilagodi njegovu veličinu. U metodi **main( )** uporedite performanse tog kontejnera i objekta tipa **ArrayList<String>**.

Vežba 32: (2) Ponovite prethodnu vežbu za kontejner celih brojeva (**int**) i uporedite njegove performanse i performanse objekta tipa **ArrayList<Integer>**. Neka poređenje performansi obuhvati i postupak povećavanja svakog objekta u kontejneru za 1.

Vežba 33: (5) Napravite objekat tipa **FastTraversalLinkedList** koji interno upotrebljava ulančanu listu (objekat tipa **LinkedList**) za brza dodavanja i uklanjanja, a objekat tipa **ArrayList** za brza prolaženja kroz kontejner i operacije **get( )**. Ispitajte ga prepravljenim programom **PerformanseLista.java**.

Opasnosti od mikropoređenja performansi

Kada pišete programe za *mikropoređenje performansi*, čuvajte se da ne pretpostavljate previše i toliko suzite testove da oni mere vreme izvršavanja samo onih operacija koje se ispituju. Takođe, morate paziti na to da ispitivanje bude dovoljno dugačko da njegovi rezultati budu statistički pouzdani i uzmite u obzir to da se neke Java HotSpot tehnologije uključuju tek nakon što se program izvršava određeno vreme (to se mora uzeti u obzir i kad se prave kratki programi).

Rezultati će se razlikovati u zavisnosti od računara i JVM-a koje upotrebljavate, pa bi trebalo da sami obavite prethodna ispitivanja kako biste proverili da li su rezultati slični

onima navedenim u ovoj knjizi. Ne pokušavajte da izmerite apsolutna trajanja izvršavanja pojedinih operacija, nego uporedite performanse raznih vrsta kontejnera.

Pored toga, *program za optimizaciju* će verovatno bolje analizirati performanse nego što vi to možete. Java se isporučuje s jednim programom za optimizaciju (videti dodatak na adresi *http://MindView.net/Books/BetterJava*), a postoje i programi za optimizaciju nezavisnih proizvođača, te besplatni/otvorenog izvornog koda i komercijalni.

Srodan je primer koji se bavi metodom **Math.random( )**. Daje li ona brojeve između nule i jedinice, uključivo ili isključivo s brojem 1? Matematički rečeno, jesu li njeni rezultati u intervalu (0,1), ili [0,1], ili (0,1] ili [0,1)? (Uglasta zagrada znači „obuhvata", a okrugla „ne obuhvata".) *Možda* će nam odgovoriti program za ispitivanje:

```
//: kontejneri/GraniceOdRandom.java
// Daje li metoda Math.random() brojeve 0.0 i 1.0?
// {RunByHand}
import static net.mindview.util.Print.*;

public class GraniceOdRandom {
  static void upotreba() {
    print("Upotreba:");
    print("\tGraniceOdRandom donja");
    print("\tGraniceOdRandom gornja");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) upotreba();
    if(args.equals("donja")) {
      while(Math.random() != 0.0)
        ; // Pokušavaj i dalje
      print("Dala 0.0!");
    }
    else if(args.equals("gornja")) {
      while(Math.random() != 1.0)
        ; // Pokušavaj i dalje
      print("Dala 1.0!");
    }
    else
      upotreba();
  }
} ///:~
```

Program pokrećete upisivanjem jedne od sledećih instrukcija na komandnu liniju:

```
java GraniceOdRandom donja
```

ili

```
java GraniceOdRandom gornja
```

U oba slučaja moraćete ručno da prekinete izvršavanje programa, pa izgleda kao da **Math.random( )** nikada ne daje ni 0.0 niti 1.0. Ali tu vas ovakav eksperiment može prevariti. Ako uzmete u obzir da između 0 i 1 postoje oko 262 različita dvostruka razlomka, verovatnoća da će se eksperimentalno dobiti bilo koji od tih pojedinačnih brojeva tako je mala da prevazilazi životni vek računara, pa čak i eksperimentatora. Ispostavlja se da metoda **Math.random( )** među svojim rezultatima *daje* i 0.0. Ili, matematički rečeno, njeni rezultati su u intervalu [0,1). Dakle, morate pažljivo analizirati svoje eksperimente i shvatiti njihova ograničenja.

Izbor skupa (realizacije interfejsa Set)

U zavisnosti od traženog ponašanja, možete izabrati **TreeSet**, **HashSet** ili **LinkedHashSet**. Naredni program za ispitivanje pokazuje različite performanse tih realizacija:

```
//: kontejneri/PerformanseSkupova.java
// Pokazuje različite performanse Skupova.
// {Args: 100 5000} Malo, da bi testiranje builda bilo kratko
import java.util.*;

public class PerformanseSkupova {
  static List<Test<Set<Integer>>> ispitivanja =
    new ArrayList<Test<Set<Integer>>>();
  static {
    ispitivanja.add(new Test<Set<Integer>>("add") {
      int ispit(Set<Integer> skup, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicina = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          skup.clear();
          for(int j = 0; j < velicina; j++)
            skup.add(j);
        }
        return petlje * velicina;
      }
    });
    ispitivanja.add(new Test<Set<Integer>>("contains") {
      int ispit(Set<Integer> skup, ParamIspita pi) {
        int petlje = pi.petlje;
        int opseg = pi.velicina * 2;
        for(int i = 0; i < petlje; i++)
          for(int j = 0; j < opseg; j++)
            skup.contains(j);
        return petlje * opseg;
      }
    });
    ispitivanja.add(new Test<Set<Integer>>("iterate") {
      int ispit(Set<Integer> skup, ParamIspita pi) {
        int petlje = pi.petlje * 10;
        for(int i = 0; i < petlje; i++) {
```

```
          Iterator<Integer> it = skup.iterator();
          while(it.hasNext())
            it.next();
        }
        return petlje * skup.size();
      }
    });
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.podrazumevaniParam = ParamIspita.niz(args);
    Tester.sirinaPolja = 10;
    Tester.pokreni(new TreeSet<Integer>(), ispitivanja);
    Tester.pokreni(new HashSet<Integer>(), ispitivanja);
    Tester.pokreni(new LinkedHashSet<Integer>(), ispitivanja);
  }
} /* Ispis: (primer)
------------- TreeSet -------------
 vel.       add  contains   iterate
   10       746       173        89
  100       501       264        68
 1000       714       410        69
10000      1975       552        69
------------- HashSet -------------
 vel.       add  contains   iterate
   10       308        91        94
  100       178        75        73
 1000       216       110        72
10000       711       215       100
---------- LinkedHashSet ----------
 vel.       add  contains   iterate
   10       350        65        83
  100       270        74        55
 1000       303       111        54
10000      1615       256        58
*///:~
```

Performanse skupa **HashSet** po pravilu su bolje od onih skupa **TreeSet**, naročito pri dodavanju elemenata i traganju za njima (metodom **contains**( )), što su dve najvažnije operacije. **TreeSet** postoji zato što svoje elemente održava u uređenom poretku, pa ga upotrebljavamo samo onda kada nam treba uređen **Set**. Zbog unutrašnje strukture potrebne za uređivanje *i* zato što se iteriranje češće izvodi, ono je obično brže u kontejneru tipa **TreeSet** nego tipa **HashSet**.

Obratite pažnju na to da su umetanja u skupu **LinkedHashSet** dugotrajnija nego u skupu **HashSet**; to važi zato što održavanje ulančane liste prouzrokuje dodatne troškove, jer se dodaje onima za kontejner s transformisanjem ključa.

Vežba 34: (1) Izmenite **PerformanseSkupova.java** tako da objekti tipa **Set** skladište znakovne nizove (**String**), a ne **Integer**e. Za pravljenje ispitnih vrednosti upotrebite **Generator** iz poglavlja *Nizovi*.

Izbor mapa (realizacija interfejsa **Map**)

Ovaj program pokazuje različite performanse raznih realizacija interfejsa **Map**:

```
//: kontejneri/PerformanseMapa.java
// Pokazuje različite performanse raznih Mapa.
// {Args: 100 5000} Malo, da bi testiranje builda bilo kratko
import java.util.*;

public class PerformanseMapa {
  static List<Test<Map<Integer,Integer>>> ispitivanja =
    new ArrayList<Test<Map<Integer,Integer>>>();
  static {
    ispitivanja.add(new Test<Map<Integer,Integer>>("put") {
      int ispit(Map<Integer,Integer> mapa, ParamIspita pi) {
        int petlje = pi.petlje;
        int velicina = pi.velicina;
        for(int i = 0; i < petlje; i++) {
          mapa.clear();
          for(int j = 0; j < velicina; j++)
            mapa.put(j, j);
        }
        return petlje * velicina;
      }
    });
    ispitivanja.add(new Test<Map<Integer,Integer>>("get") {
      int ispit(Map<Integer,Integer> mapa, ParamIspita pi) {
        int petlje = pi.petlje;
        int opseg = pi.velicina * 2;
        for(int i = 0; i < petlje; i++)
          for(int j = 0; j < opseg; j++)
            mapa.get(j);
        return petlje * opseg;
      }
    });
    ispitivanja.add(new Test<Map<Integer,Integer>>("iterate") {
      int ispit(Map<Integer,Integer> mapa, ParamIspita pi) {
        int petlje = pi.petlje * 10;
        for(int i = 0; i < petlje; i ++) {
          Iterator it = mapa.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
        return petlje * mapa.size();
      }
    });
  }
  public static void main(String[] args) {
    if(args.length > 0)
```

```
      Tester.podrazumevaniParam = ParamIspita.niz(args);
    Tester.pokreni(new TreeMap<Integer,Integer>(), ispitivanja);
    Tester.pokreni(new HashMap<Integer,Integer>(), ispitivanja);
    Tester.pokreni(new LinkedHashMap<Integer,Integer>(),ispitivanja);
    Tester.pokreni(
      new IdentityHashMap<Integer,Integer>(), ispitivanja);
    Tester.pokreni(new WeakHashMap<Integer,Integer>(), ispitivanja);
    Tester.pokreni(new Hashtable<Integer,Integer>(), ispitivanja);
  }
} /* Ispis: (primer)
---------- TreeMap ----------
 vel.     put     get iterate
   10     748     168     100
  100     506     264      76
 1000     771     450      78
10000    2962     561      83
---------- HashMap ----------
 vel.     put     get iterate
   10     281      76      93
  100     179      70      73
 1000     267     102      72
10000    1305     265      97
------- LinkedHashMap -------
 vel.     put     get iterate
   10     354     100      72
  100     273      89      50
 1000     385     222      56
10000    2787     341      56
------ IdentityHashMap ------
 vel.     put     get iterate
   10     290     144     101
  100     204     287     132
 1000     508     336      77
10000     767     266      56
-------- WeakHashMap --------
 vel.     put     get iterate
   10     484     146     151
  100     292     126     117
 1000     411     136     152
10000    2165     138     555
--------- Hashtable ---------
 vel.     put     get iterate
   10     264     113     113
  100     181     105      76
 1000     260     201      80
10000    1245     134      77
*///:-
```

U svim realizacijama interfejsa **Map**, sem u **IdentityHashMap**, s porastom **Map**e umetanje postaje znatno sporije. Medutim, pretraživanje je po pravilu mnogo brže od umetanja, što je dobro zato što se stavke obično mnogo češće traže nego što se umeću.

Performanse mape **Hashtable** približno su jednake onima mape **HashMap**. Pošto je **HashMap** predviđena da zameni **Hashtable**, te zato u pozadini upotrebljava isti mehanizam za skladištenje i pretraživanje (o kojem ćemo kasnije govoriti), ovo nas ne čudi.

Kontejner **TreeMap** po pravilu je sporiji od kontejnera **HashMap**. Kao i **TreeSet**, **TreeMap** omogućuje pravljenje uredene liste. Stablo (engl. *tree*) je uvek uredeno, pa eksplicitno sortiranje nije neophodno. Kada popunite **TreeMap**, možete pozvati metodu **keySet( )** koja će dati skup (**Set**) ključeva, a zatim metodu **toArray( )** koja će dati njihov niz. Zatim možete pozvati statičnu metodu **Arrays.binarySearch( )** koja brzo pronalazi objekte u tom uređenom nizu. Naravno, to ima smisla samo ako je ponašanje kontejnera **HashMap** neprihvatljivo, pošto baš **HashMap** brzo pronalazi ključeve. Sem toga, **HashMap** ćete lako napraviti od kontejnera **TreeMap** – treba da napravite samo jedan objekat ili da pozovete metodu **putAll( )**. Najzad, ako vam treba mapa, trebalo bi da upotrebljavate **HashMap**; **TreeMap** koristite samo ukoliko vam treba stalno uredena mapa.

Za umetanja je **LinkedHashMap** sporiji od mape **HashMap**, zato što pored strukture podataka s transformisanim ključevima održava i ulančanu listu (da bi očuvao redosled umetanja). Ali zbog te liste brža je iteracija.

Performanse kontejnera **IdentityHashMap** razlikuju se zato što on za poredenja upotrebljava ==, a ne metodu **equals( )**. **WeakHashMap** je opisana u nastavku poglavlja.

Vežba 35: (1) Izmenite program **PerformanseMapa.java** tako da obuhvati ispitivanja kontejnera **SporaMapa**.

Vežba 36: (5) Izmenite kontejner **SporaMapa** tako da umesto dve **ArrayList**e ima jedan kontejner **ArrayList** čiji su objekti tipa **StavkaMape**. Dokažite da izmenjena verzija radi ispravno. Ispitajte brzinu nove mape programom **PerformanseMapa.java**. Potom izmenite metodu **put( )** tako da nakon unošenja svakog para poziva **sort( )**, a izmenite i metodu **get( )** tako da za pronalaženje ključa koristi **Collections.binarySearch( )**. Uporedite performanse nove verzije i starih verzija.

Vežba 37: (2) Izmenite kontejner **JednostavnaHashMapa** tako da upotrebljava **ArrayList**e umesto **LinkedList**a. Izmenite program **PerformanseMapa.java** tako da poredi performanse te dve realizacije.

Od čega zavise performanse kontejnera HashMap

Kontejner tipa **HashMap** može se ručno podesiti tako da ima bolje performanse u konkretnoj aplikaciji. Morate poznavati sledeću terminologiju da biste shvatili od čega zavise performanse prilikom podešavanja kontejnera **HashMap**:

Kapacitet: Broj kofa u tabeli.

Početni kapacitet: Broj kofa nakon pravljenja tabele. **HashMap** i **HashSet** imaju konstruktore koji omogućuju zadavanje početnog kapaciteta.

Veličina: Broj stavki trenutno u tabeli.

Faktor opterećenja: veličina/kapacitet. Faktor opterećenja od 0 ima prazna tabela, 0,5 napola puna tabela itd. Malo popunjena tabela nema mnogo sudara i zato je optimalna za umetanje i pretraživanje (ali će usporiti postupak prolaska kroz kontejner iteratorom). **HashMap** i **HashSet** imaju konstruktore koji omogućuju zadavanje faktora opterećenja, pa kada se ovaj faktor opterećenja dostigne, kontejner će automatski približno dvaput povećati kapacitet (broj kofa) i preraspodeliće postojeće objekte u novi skup kofa (to se naziva *ponovno heširanje*, tj. *ponovno transformisanje ključeva*).

Podrazumevani faktor opterećenja kontejnera **HashMap** iznosi 0,75 (*ponovno transformisanje ključeva* kontejnera pokreće se tek kada se popuni tri četvrtine tabele). Izgleda da je to dobar kompromis između utrošenog vremena i prostora. Veći faktor opterećenja smanjuje prostor potreban za tabelu ali povećava troškove pretraživanja, što je važno pošto je pretraživanje – zajedno s **get( )** i **put( )** – najčešća operacija.

Ukoliko znate da ćete u kontejneru tipa **HashMap** skladištiti mnogo stavki, napravite ga s dovoljno velikim početnim kapacitetom da biste sprečili dodatne troškove zbog automatskog ponovnog transformisanja ključeva.[11]

Vežba 38: (3) Potražite klasu **HashMap** u JDK dokumentaciji. Napravite kontejner tipa **HashMap**, popunite ga elementima i izračunajte faktor opterećenja. Ispitajte brzinu pretraživanja ove mape, zatim pokušajte da povećate brzinu tako što ćete napraviti nov kontejner tipa **HashMap** većeg početnog kapaciteta i kopirati stare mape u novu, a potom ponovo pokrenite svoje ispitivanje brzine nove mape.

Vežba 39: (6) Klasi **JednostavnaHashMapa** dodajte privatnu metodu **rehash( )** koja se poziva kada faktor opterećenja premaši 0,75. Tokom ponovnog transformisanja ključeva udvostručite broj kofa, a zatim pronađite prvi veći prim broj i neka to bude novi broj kofa.

Uslužne metode

Za kontejnere postoji više samostalnih uslužnih metoda, napisanih u obliku statičnih metoda klase **java.util.Collections**. Već ste upoznali neke od njih, kao što su **addAll( )**, **reverseOrder( )** i **binarySearch( )**. U sledećoj tabeli navodimo ostale (sinhronizovane i nepromenljive uslužne metode biće opisane u narednim odeljcima). U tabeli su generički tipovi upotrebljeni samo tamo gde su relevantni:

[11] Joshua Bloch mi je pisao: „... Smatram da smo pogrešili kada smo dozvolili da pojedinosti realizacije (kao što su veličina hes tabele i faktor opterećenja) udu u naše interfejse za programiranje aplikacija. Klijent bi možda trebalo da nam saopšti maksimalnu očekivanu veličinu kontejnera, a mi bi trebalo da preuzmemo dalji posao. Klijenti će lako naneti više štete nego koristi ukoliko sami biraju vrednosti za ove parametre. Razmotrimo parametar **capacityIncrement** klase **Vector** kao krajnji slučaj. To niko nikada ne bi trebalo da zadaje, a mi smo pogrešili što smo to zadavanje omogućili. Ukoliko mu zadate bilo koju vrednost različitu od nule, linearni asimptotski troškovi sekvence dodavanja postaju kvadratni. Drugim rečima, uništavaju se performanse. S vremenom smo se opametili u tom pogledu. Kontejner **IdentityHashMap**, uopšte nema parametre za podešavanje niskog nivoa."

checkedCollection(**Kolekcija<T>, Class<T> tip)** **checkedList(** **List<T>, Class<T> tip)** **checkedMap(Map<K,V>,** **Class<K> keyType,** **Class<V> valueType)** **checkedSet(Set<T>,** **Class<T> tip)** **checkedSortedMap(** **SortedMap<K,V>,** **Class<K> keyType,** **Class<V> valueType)** **checkedSortedSet(** **SortedSet<T>,** **Class<T> tip)**	Proizvode *dinamički* bezbednu (u pogledu tipova) realizaciju **Kolekcije** ili nekog njenog konkretnog podtipa. Koristite ih kada nije moguće upotrebiti statički proverenu verziju. U poglavlju *Generički tipovi* ove metode su bile navedene pod naslovom „Dinamička bezbednost tipova".
max(Kolekcija) **min(Kolekcija)**	Daje najveći ili najmanji element argumenta koristeći metodu prirodnog poređenja objekata u **Kolekciji**.
max(Kolekcija, Komparator) **min(Kolekcija, Komparator)**	Daje najveći ili najmanji element **Kolekcije** koristeći **Komparator**.
indexOfSubList(List izvor, **List odredište)**	Proizvodi početni indeks *prvog* mesta na kojem se **odredište** pojavljuje unutar **izvora** ili -1 ako takvog mesta nema.
lastIndexOfSubList(List izvor, **List odredište)**	Proizvodi početni indeks *poslednjeg* mesta na kojem se **odredište** pojavljuje unutar **izvora** ili -1 ako takvog mesta nema.
replaceAll(List<T>, **T staraVrednost, T novaVrednost)**	Zamenjuje sve primerke objekta **staraVrednost** objektom **novaVrednost**.
reverse(List)	Obrće redosled svih elemenata liste.
reverseOrder( ) **reverseOrder(** **Komparator<T>)**	Vraća **Komparator** koji obrće prirodni poredak kolekcije objekata koja realizuje interfejs **Comparable<T>**. Druga verzija obrće redosled datog **Komparatora**.
rotate(List, int udaljenost)	Premešta sve elemente unapred za **udaljenost**, one s kraja vraća na početak.
shuffle(List) **shuffle(List, Random)**	Nasumično permutuje datu listu. Prvi oblik ima sopstveni izvor slučajnih brojeva, dok ga u drugom obliku sami zadajete.
sort(List<T>) **sort(List<T>,** **Komparator<? super T> c)**	Uređuje listu **List<T>** po njenom prirodnom redosledu. U drugom obliku sami zadajete **Komparator** za uređivanje.
copy(List<? super T> odredište, **List<? extends T> izvor)**	Kopira **elemente** iz izvora u **odredište**.
swap(List, int i, int j)	Zamenjuje elemente na mestima **i** i **j** u listi **List**. Verovatno brža od one koju možete sami da napišete.
fill(List<? super T>, T x)	Zamenjuje sve elemente liste objektom **x**.

nCopies(int n, T x)	Vraća nepromenljivu listu **List<T>** veličine **n** čije sve reference upućuju na objekat **x**
disjoint(Kolekcija, Kolekcija)	Vraća **true** ako date kolekcije nemaju zajedničkih elemenata.
frequency(Kolekcija, Object x)	Vraća broj elemenata **Kolekcije** jednakih objektu **x**.
emptyList( ) **emptyMap( )** **emptySet( )**	Vraća nepromenljivu praznu **listu**, **mapu** ili **skup**. Oni su generički, pa će rezultujuća **Kolekcija** biti parametrizovana u željeni tip.
singleton(T x) **singletonList(T x)** **singletonMap(K kljuc, V vrednost)**	Vraća nepromenljivu listu, mapu ili skup (**Set<T>**, **List<T>** odnosno **Map<K,V>**) sa samo jednom stavkom, napravljenom na osnovu datih argumenata.
list(Enumeration<T> e)	Proizvodi **ArrayList<T>** koja sadrži elemente u poretku kojim ih vraća (stari) **Enumeration** (prethodnik **Iteratora**). Za konvertovanje starog koda.
enumeration(Kolekcija <T>)	Proizvodi stari **Enumeration<T>** za dati argument.

Imajte u vidu da **min**( ) i **max**( ) rade sa objektima tipa **Collection**, ne sa **List**ama, pa ne morate brinuti da li je data kolekcija uređena ili nije. Već smo spomenuli da za **List**u ili niz *treba* da pozovete metodu **sort**( ) pre poziva metode **binarySearch**( ).

U narednom primeru prikazaćemo osnove korišćenja uslužnih metoda iz gornje tabele:

```
//: kontejneri/UsluzneMetode.java
// Jednostavni primeri uslužnih metoda klase Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class UsluzneMetode {
  static List<String> lista = Arrays.asList(
    "jedan Dva tri Cetiri pet sest jedan".split(" "));
  public static void main(String[] args) {
    print(lista);
    print("'lista' disjoint (Cetiri)?: " +
      Collections.disjoint(lista,
        Collections.singletonList("Cetiri")));
    print("mks: " + Collections.max(lista));
    print("min: " + Collections.min(lista));
    print("mks uz komparator: " + Collections.max(lista,
      String.CASE_INSENSITIVE_ORDER));
    print("min uz komparator: " + Collections.min(lista,
      String.CASE_INSENSITIVE_ORDER));
    List<String> podlista =
      Arrays.asList("Cetiri pet sest".split(" "));
    print("indeksPodliste: " +
      Collections.indeksPodliste(lista, podlista));
    print("poslednjiIndeksPodliste: " +
      Collections.poslednjiIndeksPodliste(lista, podlista));
```

```
    Collections.replaceAll(lista, "jedan", "Da");
    print("replaceAll: " + lista);
    Collections.reverse(lista);
    print("obrnuto: " + lista);
    Collections.rotate(lista, 3);
    print("rotirano: " + lista);
    List<String> izvor =
      Arrays.asList("u matrici".split(" "));
    Collections.copy(lista, izvor);
    print("kopija: " + lista);
    Collections.swap(lista, 0, lista.size() - 1);
    print("zamena: " + lista);
    Collections.shuffle(lista, new Random(47));
    print("ispreturano: " + lista);
    Collections.fill(lista, "pop");
    print("popuna: " + lista);
    print("broj primeraka reči 'pop': " +
      Collections.frequency(lista, "pop"));
    List<String> duplikati = Collections.nCopies(3, "snap");
    print("duplikati: " + duplikati);
    print("'lista' disjoint 'duplikati'?: " +
      Collections.disjoint(lista, duplikati));
    // Pravljenje starog iteratora Enumeration:
    Enumeration<String> e = Collections.enumeration(duplikati);
    Vector<String> v = new Vector<String>();
    while(e.hasMoreElements())
      v.addElement(e.nextElement());
    // Pretvaranje starog Vectora u objekat
    // tipa List pomoću iteratora Enumeration:
    ArrayList<String> arrayList =
      Collections.lista(v.elements());
    print("arrayList: " + arrayList);
  }
} /* Ispis:
[jedan, Dva, tri, Cetiri, pet, sest, jedan]
'lista' disjoint (Cetiri)?: false
mks: tri
min: Četiri
mks uz komparator: Dva
min uz komparator: pet
indeksPodliste: 3
poslednjiIndeksPodliste: 3
replaceAll: [Da, Dva, tri, Cetiri, pet, sest, Da]
obrnuto: [Da, sest, pet, Cetiri, tri, Dva, Da]
rotirano: [tri, Dva, Da, Da, sest, pet, Cetiri]
kopija: [u, matrici, Da, sest, pet, Cetiri]
zamena: [Cetiri, matrici, Da, sest, pet, u]
ispreturano: [sest, matrici, Cetiri, Da, pet, u]
```

```
popuna: [pop, pop, pop, pop, pop, pop, pop]
broj primeraka reči 'pop': 7
duplikati: [snap, snap, snap]
'lista' disjoint 'duplikati'?: true
arrayList: [snap, snap, snap]
*///:~
```

Ponašanje svake uslužne metode vidi se iz ispisa rezultata. Obratite pažnju na razliku između rezultata metoda **min( )** i **max( )** uz **String.CASE_INSENSITIVE_ORDER Comparator**; razlog je zanemarivanje razlike između velikih i malih slova.

Uređivanje i pretraživanje lista (realizacija interfejsa **List**)

Uslužne metode za sortiranje i pretraživanje lista imaju ista imena i potpise kao one za uređivanje nizova objekata, ali to su statične metode klase **Collections**, a ne klase **Arrays**. Evo primera u kojem se koriste metode za liste iz paketa **Utilities.java**:

```
//: kontejneri/ListSortSearch.java
// Uređivanje i pretraživanje lista uslužnim metodama klase Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List<String> lista =
      new ArrayList<String>(Utilities.lista);
    lista.addAll(Utilities.lista);
    print(lista);
    Collections.shuffle(lista, new Random(47));
    print("Ispreturana: " + lista);
    // Uklanjanje poslednjih elemenata pomoću ListIteratora:
    ListIterator<String> it = lista.listIterator(10);
    while(it.hasNext()) {
      it.next();
      it.remove();
    }
    print("Skraćena: " + lista);
    Collections.sort(lista);
    print("Uređena: " + lista);
    String kljuc = lista.get(7);
    int indeks = Collections.binarySearch(lista, kljuc);
    print("Mesto ključa " + kljuc + " je " + indeks +
      ", lista.get(" + indeks + ") = " + lista.get(indeks));
    Collections.sort(lista, String.CASE_INSENSITIVE_ORDER);
    print("Uređena bez obzira na velika i mala slova: " + lista);
    kljuc = lista.get(7);
    indeks = Collections.binarySearch(lista, kljuc,
      String.CASE_INSENSITIVE_ORDER);
```

```
      print("Mesto ključa " + kljuc + " is " + indeks +
        ", lista.get(" + indeks + ") = " + lista.get(indeks));
    }
} /* Ispis:
[jedan, Dva, tri, Cetiri, pet, sest, jedan, jedan, Dva, tri, Cetiri,
pet,
sest, jedan]
Ispreturana: [Cetiri, pet, jedan, jedan, Dva, sest, sest, tri, tri, pet,
Cetiri, Dva, jedan, jedan]
Skraćena: [Cetiri, pet, jedan, jedan, Dva, sest, sest, tri, tri, pet]
Uređena: [Cetiri, Dva, jedan, jedan, pet, pet, sest, sest, tri, tri]
Mesto ključa sest je 7, lista.get(7) = sest
Uređena bez obzira na velika i mala slova: [Cetiri, Dva, jedan,
jedan, pet, pet, sest, sest, tri, tri]
Mesto ključa tri je 7, lista.get(7) = tri
*///:~
```

Kao pri pretraživanju i uređivanju nizova, i ovde važi sledeće: ukoliko listu uredite pomoću nekog komparatora, pomoću istog komparatora morate obaviti i binarno pretraživanje – metodom **binarySearch**( ).

U ovom programu prikazana je i metoda **shuffle**( ) klase **Collections** koja nasumično menja redosled elemenata liste. Jedan **ListIterator** je napravljen na određenom mestu ispreturane (engl. *shuffled*) liste i upotrebljen za uklanjanje elemenata od tog mesta do kraja liste.

Vežba 40: (5) Napravite klasu s dva objekta tipa **String** i realizujte u njoj interfejs **Comparable** tako da poređenje uzima u obzir samo prvi objekat tipa **String**. Popunite jedan niz i jedan objekat tipa **ArrayList** objektima vaše klase, koristeći generator **RandomGenerator**. Pokažite da se sortiranje obavlja ispravno. Potom napravite **Comparator** koji uzima u obzir samo drugi objekat tipa **String** i pokažite da je sortiranje opet ispravno. Obavite i binarno pretraživanje pomoću svoje realizacije interfejsa **Comparator**.

Vežba 41: (3) Izmenite klasu iz prethodne vežbe tako da radi s kontejnerima tipa **HashSet** i kao ključ u kontejnerima tipa **HashMap**.

Vežba 42: (2) Izmenite vežbu 40 tako da lista bude sortirana abecedno.

Kako **kolekciju** ili **mapu** učiniti nepromenljivom

Često je pogodno imati verziju kolekcije ili mape samo za čitanje. Klasa **Collections** omogućuje da to uradite prosledivanjem originalnog kontejnera metodi koja vraća njegovu verziju samo za čitanje. Ta metoda ima više varijanata: za kolekcije (ako datu kolekciju ne možete tretirati kao konkretniji tip), liste, skupove i mape. U sledećem primeru prikazan je pravilan način pravljenja verzije samo za čitanje svakog od nabrojanih vrsta kontejnera:

```
//: kontejneri/SamoZaCitanje.java
// Pomoću metoda Collections.unmodifiable.
import java.util.*;
import net.mindview.util.*;
```

```
import static net.mindview.util.Print.*;

public class SamoZaCitanje {
  static Collection<String> podaci =
    new ArrayList<String>(Countries.imena(6));
  public static void main(String[] args) {
    Collection<String> c =
      Collections.unmodifiableCollection(
        new ArrayList<String>(podaci));
    print(c); // Čitanje funkcioniše
    //! c.add("jedan"); // Nema menjanja

    List<String> a = Collections.unmodifiableList(
        new ArrayList<String>(podaci));
    ListIterator<String> lit = a.listIterator();
    print(lit.next()); // Čitanje funkcioniše
    //! lit.add("jedan"); // Nema menjanja

    Set<String> s = Collections.unmodifiableSet(
      new HashSet<String>(podaci));
    print(s); // Čitanje funkcioniše
    //! s.add("jedan"); // Nema menjanja

    // Za SortedSet:
    Set<String> ss = Collections.unmodifiableSortedSet(
      new TreeSet<String>(podaci));

    Map<String,String> m = Collections.unmodifiableMap(
      new HashMap<String,String>(Countries.glavni_gradovi(6)));
    print(m); // Čitanje funkcioniše
    //! m.put("Ralf", "Zdravo!");

    // Za SortedMap:
    Map<String,String> sm =
      Collections.unmodifiableSortedMap(
        new TreeMap<String,String>(Countries.glavni_gradovi(6)));
  }
} /* Ispis:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
ALGERIA
[BURKINA FASO, BURUNDI, BOTSWANA, BENIN, ANGOLA, ALGERIA]
{BURKINA FASO=Ouagadougou, BURUNDI=Bujumbura, BOTSWANA=Gaberone,
BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
*///:~
```

Poziv nepromenljive metode za određeni tip ne prouzrokuje proveru u vreme prevođenja, ali nakon što se ta transformacija obavi, poziv bilo koje metode koja menja sadržaj tog kontejnera prouzrokuje **UnsupportedOperationException**.

U svakom slučaju, kontejner morate popuniti smislenim podacima *pre* nego što ga pretvorite u verziju samo za čitanje. Nakon učitavanja, najbolje je zameniti postojeću referencu kontejnera referencom koju proizvodi poziv „unmodifiable" metode. Time izbegavate rizik od nehotičnog pokušaja menjanja sadržaja nakon što ste ga učinili nepromenljivim. S druge strane, ova alatka omogućuje i da promenljivi kontejner zadržite u obliku privatnog člana određene klase i da iz poziva neke metode vratite referencu tog kontejnera samo za čitanje. Dakle, kontejner možete izmeniti iznutra iz klase, a svi drugi mogu samo da ga čitaju.

Sinhronizovanje **kolekcije** ili **mape**

Rezervisana reč **synchronized** čini važan deo *višenitnog izvršavanja*, ali je ona toliko komplikovana da je nećemo načinjati do poglavlja *Paralelno izvršavanje*. Ovde navodimo samo to da klasa **Collections** može automatski da sinhronizuje ceo kontejner. Sintaksa je slična „unmodifiable" metodama:

```
//: kontejneri/Sinhronizacija.java
// Upotreba Collections.synchronized metoda.
import java.util.*;

public class Sinhronizacija {
  public static void main(String[] args) {
    Collection<String> c =
      Collections.synchronizedCollection(
        new ArrayList<String>());
    List<String> lista = Collections.synchronizedList(
      new ArrayList<String>());
    Set<String> s = Collections.synchronizedSet(
      new HashSet<String>());
    Set<String> ss = Collections.synchronizedSortedSet(
      new TreeSet<String>());
    Map<String,String> m = Collections.synchronizedMap(
      new HashMap<String,String>());
    Map<String,String> sm =
      Collections.synchronizedSortedMap(
        new TreeMap<String,String>());
  }
} ///:~
```

Najbolje je novi kontejner odmah proslediti preko odgovarajuće „synchronized" metode, kao u gornjem primeru. Tako se ne može dogoditi da slučajno eksponirate nesinhronizovanu verziju.

Brzo otkazivanje

Javini kontejneri imaju i mehanizam koji sprečava da više procesa istovremeno menja sadržaj kontejnera. Problem nastaje ako ste usred iteriranja kroz kontejner, a neki drugi proces uskoči i umetne, ukloni ili izmeni neki objekat u tom kontejneru. Možda ste taj ele-

ment kontejnera već prošli, možda je on ispred vas, možda će se kontejner smanjiti nakon što pozovete metodu **size( )** – scenarija za katastrofu ima koliko hoćete. Biblioteka Java kontejnera upotrebljava mehanizam *brzog otkazivanja* (engl. *fail-fast*); u kontejneru traži izmene koje nije prouzrokovao vaš proces lično. Ako otkrije da neko drugi menja sadržaj kontejnera, odmah generiše izuzetak **ConcurrentModificationException**. Taj aspekat se naziva „brzo otkazivanje" – problem se ne traži naknadno, nekim složenijim algoritmom.

Mehanizam brzog otkazivanja je sasvim lako videti u praksi – dovoljno je napraviti iterator i zatim nešto dodati kolekciji na koju iterator pokazuje:

```
//: kontejneri/BrzoOtkazivanje.java
// Pokazuje ponašanje nazvano "brzo otkazivanje".
import java.util.*;

public class BrzoOtkazivanje {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    Iterator<String> it = c.iterator();
    c.add("Neki objekat");
    try {
      String s = it.next();
    } catch(ConcurrentModificationException e) {
      System.out.println(e);
    }
  }
} /* Ispis:
java.util.ConcurrentModificationException
*///:~
```

Izuzetak se dogodio zato što je nešto smešteno u kontejner *nakon* pribavljanja iteratora od tog kontejnera. Mogućnost da dva dela programa menjaju isti kontejner proizvodi neodređeno stanje, pa vas izuzetak obaveštava kako bi trebalo da izmenite kôd – u ovom slučaju, pribavite iterator tek *nakon* dodavanja svih elemenata u kontejner.

Kontejneri **ConcurrentHashMap**, **CopyOnWriteArrayList** i **CopyOnWriteArraySet** koriste tehnike kojima se izbegava generisanje izuzetka **ConcurrentModificationException**s.

Čuvanje referenci

Biblioteka **java.lang.ref** sadrži skup klasa koje omogućuju veću fleksibilnost u sakupljanju smeća. Te klase su naročito korisne kada radite s velikim objektima koji mogu iscrpeti memoriju. Od apstraktne klase **Reference** nasleđene su tri klase: **SoftReference**, **WeakReference** i **PhantomReference**. Svaka od njih sakupljaču smeća pruža drugačiji nivo indirekcije ukoliko je objekat o kojem se radi dostižan samo preko jednog od tih **Reference** objekata.

Ako je objekat *dostižan*, može se pronaći negde u programu. To može značiti da imate običnu referencu steka koja upućuje pravo na objekat, ali mogli biste imati i referencu objekta koji sadrži referencu objekta o kojem se radi; tih međuveza može biti mnogo.

Ukoliko je objekat dostižan, sakupljač smeća ga ne može osloboditi jer ga program još uvek koristi. Ako objekat nije dostižan, ne postoji način da ga program upotrebi, pa je bezbedno sakupiti ga u smeće.

Objekte tipa **Reference** koristite kada i dalje hoćete da čuvate referencu određenog objekta – taj objekat hoćete da dosegnete – ali takođe hoćete da omogućite sakupljaču smeća da taj objekat oslobodi. Dakle, postoji način da objekat koristite, ali ako se ukaže mogućnost da se memorija iscrpe, dozvoljavate da taj objekat bude oslobođen.

To se postiže korišćenjem objekta tipa **Reference** kao *posrednika* (engl. *proxy*) između vas i obične reference. Sem toga, taj objekat ne sme imati obične reference (one koje nisu omotane u **Reference** objekte). Ako sakupljač smeća otkrije da je određeni objekat dostižan preko obične reference, taj objekat neće biti oslobođen.

Reference tipa **SoftReference**, **WeakReference** i **PhantomReference** sve su „slabije" i odgovaraju različitim nivoima dostižnosti. Meke reference (engl. *soft references*) služe za realizovanje keša, tj. ostava koje čuvaju memoriju. Slabe reference (engl. *weak references*) služe za realizovanje „kanonizujućih mapiranja (preslikavanja)" – gde se instance objekata mogu istovremeno upotrebljavati na više mesta u programu, da bi se uštedela memorija – što ne sprečava da njihovi ključevi (ili vrednosti) budu sakupljeni u smeće. Fantomske reference (engl. *phantom references*) služe za planiranje operacija čišćenja „pre (memorijske) smrti" na fleksibilniji način nego što bi to bilo moguće s Javinim mehanizmom finalizacije.

Reference tipa **SoftReference** i **WeakReference** možete ali i ne morate smestiti u „red referenci za čekanje" (**ReferenceQueue**) – tako je nazvan uređaj za akcije čišćenja „pre (memorijske) smrti") – dok se **PhantomReference** može napraviti samo u tom **ReferenceQueue**. Evo jednostavnog primera:

```
//: kontejneri/Reference.java
// Primer objekata tipa Reference
import java.lang.ref.*;
import java.util.*;

class VeomaVeliki {
  private static final int VELICINA = 10000;
  private long[] la = new long[VELICINA];
  private String ident;
  public VeomaVeliki(String id) { ident = id; }
  public String toString() { return ident; }
  protected void finalize() {
    System.out.println("Finalizovanje " + ident);
  }
}

public class Reference {
  private static ReferenceQueue<VeomaVeliki> rq =
    new ReferenceQueue<VeomaVeliki>();
  public static void checkQueue() {
    Reference<? extends VeomaVeliki> urd = rq.poll();
    if(urd != null)
```

```
      System.out.println("Unutar reda: " + urd.get());
    }
    public static void main(String[] args) {
      int velicina = 10;
      // Ili izaberite veličinu sa komandne linije:
      if(args.length > 0)
        velicina = new Integer(args);
      LinkedList<SoftReference<VeomaVeliki>> sa =
        new LinkedList<SoftReference<VeomaVeliki>>();
      for(int i = 0; i < velicina; i++) {
        sa.add(new SoftReference<VeomaVeliki>(
          new VeomaVeliki("Soft " + i), rq));
        System.out.println("Tek napravljeno: " + sa.getLast());
        checkQueue();
      }
      LinkedList<WeakReference<VeomaVeliki>> wa =
        new LinkedList<WeakReference<VeomaVeliki>>();
      for(int i = 0; i < velicina; i++) {
        wa.add(new WeakReference<VeomaVeliki>(
          new VeomaVeliki("Weak " + i), rq));
        System.out.println("Tek napravljeno: " + wa.getLast());
        checkQueue();
      }
      SoftReference<VeomaVeliki> s =
        new SoftReference<VeomaVeliki>(new VeomaVeliki("Soft"));
      WeakReference<VeomaVeliki> w =
        new WeakReference<VeomaVeliki>(new VeomaVeliki("Weak"));
      System.gc();
      LinkedList<PhantomReference<VeomaVeliki>> pa =
        new LinkedList<PhantomReference<VeomaVeliki>>();
      for(int i = 0; i < velicina; i++) {
        pa.add(new PhantomReference<VeomaVeliki>(
          new VeomaVeliki("Phantom " + i), rq));
        System.out.println("Tek napravljeno: " + pa.getLast());
        checkQueue();
      }
    }
  } /* (Pokrenite da biste videli rezultate) *///:~
```

Kada pokrenete ovaj program (preusmerite njegov izlaz u tekstualnu datoteku da biste rezultate mogli da gledate stranicu po stranicu), videćete da objekti bivaju sakupljani u smeće iako im i dalje možete pristupati preko objekta tipa **Reference** – referencu objekta pribavljate metodom **get( )**. Videćete i to da **ReferenceQueue** uvek proizvodi **null** objekat tipa **Reference**. Da bi taj objekat postao upotrebljiv, nasledite određenu klasu **Reference**, a novoj klasi dodajte upotrebljivije metode.

Kontejner **WeakHashMap**

Biblioteka kontejnera ima posebnu mapu za skladištenje slabih referenci: to je **WeakHashMap**. Ta klasa olakšava pravljenje kanonizovanih mapiranja (preslikavanja). Takvim mapiranjem štedite memoriju jer pravite samo jednu instancu određene vrednosti. Kada programu zatreba ta vrednost, on potraži postojeći objekat u tom mapiranju i upotrebi njega (umesto da ga pravi od nule). Mapiranje može napraviti vrednosti u sklopu svoje inicijalizacije, ali se vrednosti češće prave tek na zahtev.

Pošto je ovo tehnika za uštedu memorije, baš je podesno što **WeakHashMap** omogućuje skupljaču smeća da automatski čisti njene ključeve i vrednosti. S ključevima i vrednostima koje želite da smestite u mapu **WeakHashMap** ne morate da radite ništa posebno; sama mapa ih automatski omotava u objekte tipa **WeakReference**. Okidač koji dozvoljava čišćenje jeste činjenica da se ključ više ne upotrebljava, kao što je prikazano u narednom primeru:

```
//: kontejneri/KanonskoMapiranje.java
// Primer s kontejnerom WeakHashMap.
import java.util.*;

class Element {
  private String ident;
  public Element(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() { return ident.hashCode(); }
  public boolean equals(Object r) {
    return r instanceof Element &&
      ident.equals(((Element)r).ident);
  }
  protected void finalize() {
    System.out.println("Finalizovanje " +
      getClass().getSimpleName() + " " + ident);
  }
}

class Kljuc extends Element {
  public Kljuc(String id) { super(id); }
}

class Vrednost extends Element {
  public Vrednost(String id) { super(id); }
}

public class KanonskoMapiranje {
  public static void main(String[] args) {
    int velicina = 1000;
    // Ili izaberite veličinu na komandnoj liniji:
    if(args.length > 0)
      velicina = new Integer(args[0]);
```

```
    Kljuc[] kljucevi = new Kljuc[velicina];
    WeakHashMap<Kljuc,Vrednost> mapa =
      new WeakHashMap<Kljuc,Vrednost>();
    for(int i = 0; i < velicina; i++) {
      Kljuc k = new Kljuc(Integer.toString(i));
      Vrednost v = new Vrednost(Integer.toString(i));
      if(i % 3 == 0)
        kljucevi[i] = k; // Sačuvaj kao "prave" reference
      mapa.put(k, v);
    }
    System.gc();
  }
} /* (Pokrenite da biste videli rezultate) *///:~
```

Klasa **Kljuc** mora imati svoje metode **hashCode( )** i **equals( )**, pošto se upotrebljava kao ključ u strukturi podataka s transformisanjem ključeva. Metoda **hashCode( )** opisana je u prethodnom delu poglavlja.

Kada pokrenete program, videćete da skupljač smeća preskače svaki treći ključ, zato što su u niz **kljucevi** smeštene i obične reference tih ključeva, te se ti objekti ne mogu sakupljati kao smeće.

Kontejneri Jave 1.0/1.1

Nažalost, mnogo koda je napisano uz korišćenje kontejnera iz Jave 1.0/1.1, a ponekad se čak i nov kôd piše uz korišćenje ovih klasa. Zbog toga morate znati sve o starim kontejnerima, mada nikada ne bi trebalo da ih koristite dok pišete nov kôd. Međutim, stari kontejneri su bili prilično ograničeni, pa o njima nema mnogo toga da se kaže. (Pošto su oni sad prošlost, uzdržaću se od ismevanja nekih ishitrenih odluka primenjenih pri njihovom projektovanju.)

Vector i Enumeration

Jedini niz koji je mogao da se proširuje u Javi 1.0/1.1 bio je **Vector**, pa se dosta koristio. Njegovi nedostaci su suviše brojni da bi se ovde naveli (potražite prvo izdanje ove knjige na lokaciji *www.BruceEckel.com*). U osnovi, to je bio **ArrayList** s dugim, nezgodnim imenima metoda. U biblioteci kontejnera nove Jave, **Vector** je prilagođen da bi mogao da radi i kao kolekcija i kao lista. To je pomalo čudno zato što bi neki ljudi mogli pomisliti kako je klasa **Vector** poboljšana, a ona u stvari postoji samo da bi podržavala stari Java kôd.

Verzija iteratora u Javi 1.0/1.1 drugačije se zvala – **Enumeration** – tj. nije korišćen termin koji je svima poznat. Interfejs **Enumeration** je manji od interfejsa **Iterator** i sadrži samo dve metode s dugačkim imenima: **boolean hasMoreElements( )** koja vraća **true** ako kontejner sadrži još elemenata, i **Object nextElement( )** koja vraća sledeći element kolekcije ako on postoji (inače generiše izuzetak).

Enumeration je samo interfejs bez realizacije, pa ga čak i nove biblioteke ponekad koriste, što nije preporučljivo ali je barem bezopasno. Iako bi prilikom pisanja novih programa uvek trebalo koristiti **Iterator**, morate znati da postoje biblioteke koje žele da vam prosleđe objekat tipa **Enumeration**.

Osim toga, pomoću metode **Collections.enumeration( )** možete da napravite objekat tipa **Enumeration** za bilo koju kolekciju, kao u ovom primeru:

```
//: kontejneri/Nabrajanje.java
// Java 1.0/1.1 klase Vector i Enumeration.
import java.util.*;
import net.mindview.util.*;

public class Nabrajanje {
  public static void main(String[] args) {
    Vector<String> v = new Vector<String> (Countries.names(10));
    Enumeration<String> e = v.elements();
    while(e.hasMoreElements())
      System.out.print(e.nextElement() + ", ");
    // Pravi Enumeration od kolekcije:
    e = Collections.enumeration(new ArrayList<String> ());
  }
} /* Ispis:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC, CHAD
*///:~
```

Da biste napravili objekat tipa **Enumeration**, pozovite metodu **elements( )**, koju potom možete da upotrebite za prolazak kroz listu.

U poslednjem redu pravi se objekat klase **ArrayList** i koristi metoda **enumeration( )** za dobijanje objekta tipa **Enumeration** na osnovu iteratora klase **ArrayList**. Dakle, možete da koristite nove kontejnere i sa starim kodom u kome se upotrebljava **Enumeration**.

Klasa **Hashtable**

Kao što ste videli u delovima ovog poglavlja posvećenim upoređivanju performansi, klasa **Hashtable** je veoma slična klasi **HashMap**, čak su i imena metoda ista. Zato nema razloga da se u novim programima umesto klase **HashMap** koristi **Hashtable**.

Klasa **Stack**

Pojam steka objašnjen je ranije, u odeljku o klasi **LinkedList**. Klasa **Stack** iz Jave 1.0/1.1 čudna je: umesto da koristi klasu **Vector** i kompoziciju, ona je *izvedena* (nasleđena) iz klase **Vector**. Zato ima sva obeležja i ponašanje klase **Vector**, uz dodatne funkcije steka. Teško je razaznati da li su projektanti smatrali da je to posebno korisno, ili se radi o neznanju; ipak, jasno je da projekat nije revidiran pre puštanja u distribuciju, pa se i tako loš dizajn *još uvek* sreće (ali ga vi nemojte koristiti).

Evo jednostavnog primera steka na koji se stavlja svaki red iz niza objekata tipa **String**. Pokazano je kako je kao stek jednako lako koristiti ulančanu listu (objekat tipa **LinkedList**) ili stek napravljen u poglavlju *Čuvanje objekata*:

```
//: kontejneri/Stekovi.java
// Prikazivanje klase Stack.
import java.util.*;
import static net.mindview.util.Print.*;

enum Mesec { JANUAR, FEBRUAR, MART, APRIL, MAJ, JUN,
  JUL, AVGUST, SEPTEMBAR, OKTOBAR, NOVEMBAR};

public class Stekovi {
  public static void main(String[] args) {
    Stack<String> stk = new Stack<String> ();
    for(Mesec m : Mesec.values())
      stk.push(m.toString());
    print("stek = " + stk);
    // Stek kao Vector:
    stk.addElement("Poslednji red");
    print("element 5 = " + stk.elementAt(5));
    print("Vadim elemente:");
    while(!stk.empty())
      printnb(stk.pop() + " ");

// Ulančana lista kao stek:
LinkedList<String> lstek = new LinkedList<String> ();
for(Mesec m : Mesec.values())
   lstek.addFirst(m.toString());
print("lstek = " + lstek);
while(!lstek.isEmpty())
      printnb(lstek.removeFirst( ) + " ");

// Klasa Stek iz poglavlja Čuvanje objekata
net.mindview.util.Stek<String> stek2 =
   new net.mindview.util.Stek<String> ();
for(Mesec m : Mesec.values())
   stek2.push(m.toString());
print("stek2 = " + stek2);
while(!stek2.empty())
      printnb(stek2.pop( ) + " ");
  }
} /* Ispis:
stek = [JANUAR, FEBRUAR, MART, APRIL, MAJ, JUN, JUL, AVGUST,
SEPTEMBAR, OKTOBAR, NOVEMBAR]
element 5 = JUN
Vadim elemente:
Poslednji red NOVEMBAR OKTOBAR SEPTEMBAR AVGUST JUL JUN MAJ APRIL MART
FEBRUAR JANUAR lstek = [NOVEMBAR OKTOBAR SEPTEMBAR AVGUST JUL JUN MAJ
APRIL MART FEBRUAR JANUAR]
NOVEMBAR OKTOBAR SEPTEMBAR AVGUST JUL JUN MAJ APRIL MART FEBRUAR
JANUAR stek2 = [NOVEMBAR OKTOBAR SEPTEMBAR AVGUST JUL JUN MAJ APRIL
MART FEBRUAR JANUAR]
NOVEMBAR OKTOBAR SEPTEMBAR AVGUST JUL JUN MAJ APRIL
MART FEBRUAR JANUAR
*///:~
```

Od klase **Mesec** definisane nabrajanjem konstanti generišu se odgovarajući znakovni nizovi (objekti tipa **String**), svaki se stavlja na stek metodom **push( )**, a kasnije se skida s vrha steka metodom **pop( )**. Radi prikaza, operacije klase **Vector** takođe se izvode na objektu klase **Stack**. To je moguće zato što je, zahvaljujući nasleđivanju, stek jedna vrsta vektora. Stoga se sve operacije koje se mogu obaviti s vektorom mogu takođe obaviti i sa stekom, npr. **elementAt( )**.

Kao što je ranije pomenuto, kada želite ponašanje tipa steka, treba da koristite **LinkedList** ili klasu **net.mindview.util.Stek** izvedenu iz klase **LinkedList**.

Klasa **BitSet**

Klasa **BitSet** se koristi za efikasno čuvanje velike količine informacija tipa *uključeno--isključeno*. Ona je efikasna samo s tačke gledišta veličine; ako vam je potreban brz pristup, treba da znate da je ova klasa nešto sporija od nizova prirodnog tipa.

Osim toga, minimalna veličina skupa bitova je **long**, tj. 64 bita. To znači da klasa **BitSet** neće biti efikasna za čuvanje kraćih podataka, npr. onih od 8 bitova. U tom slučaju, bolje je da napravite sopstvenu klasu, ili samo niz u koji ćete smestiti podatke ako vam je bitna veličina. (To će biti slučaj samo ako pravite *mnogo* objekata koji sadrže liste informacija tipa uključeno-isključeno, a odluku treba doneti tek na osnovu rezultata programa za optimizaciju i ostalih merenja. Ukoliko ste to odlučili samo zato što ste sami nešto proglasili za preveliko, imaćete veoma složen program i izgubićete mnogo vremena.)

Običan kontejner se širi kada mu dodajete nove elemente, a to radi i **BitSet**. Sledeći primer pokazuje kako radi klasa **BitSet**:

```
//: kontejneri/Bitovi.java
// Prikazivanje klase BitSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bitovi {
  public static void ispisiBitove(BitSet b) {
    print("bitovi: " + b);
    StringBuilder bbitovi = new StringBuilder( );
    for(int j = 0; j < b.size() ; j++)
      bbitovi.append(b.get(j) ? "1" : "0");
    print("bit-maska: " + bbitovi);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Uzmi bajt najmanje težine od nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
```

```
    print("vrednost tipa byte: " + bt);
    ispisiBitove(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    print("vrednost tipa short: " + st);
    ispisiBitove(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    print("vrednost tipa int: " + it);
    ispisiBitove(bi);

    // Testiraj skupove bita >= 64 bita:
    BitSet b127 = new BitSet();
    b127.set(127);
    print("bit broj 127 je uključen: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    print("bit broj 255 je uključen: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    print("bit broj 1023 je uključen: " + b1023);
} /* Ispis:  }
vrednost tipa byte: -107
bitovi: {0, 2, 4, 7}
bit-maska:
10101001000000000000000000000000000000000000000000000000000000000000000000000000
vrednost tipa short: 1302
bitovi: {1, 2, 4, 8, 10}
bit-maska:
10101001000000000000000000000000000000000000000000000000000000000000000000000000
vrednost tipa int: -2014573909
bitovi: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24, 25, 26, 31}
bit-maska:
10101001000000000000000000000000000000000000000000000000000000000000000000000000
bit broj 127 je uključen: {127}
bit broj 255 je uključen: {255}
bit broj 1023 je uključen: {1023, 1024}
* ///:~
```

Generator slučajnih brojeva koristi se za generisanje slučajnih brojeva tipa **byte**, **short** i **int**, a svaki od njih se pretvara u odgovarajuću bit masku u klasi **BitSet**. Ovo lepo radi zato što je **BitSet** 64-bitni, pa nijedan od ovih brojeva ne prouzrokuje povećavanje njegove dužine. Potom se pravi skup bitova veće dužine. Vidite da se **BitSet** širi po potrebi.

Ukoliko imate nepromenljiv skup indikatora koje možete da imenujete, obično je bolje koristiti klasu **EnumSet** (videti poglavlje *Nabrojani tipovi*) nego klasu **BitSet** pošto **EnumSet** omogućuje rukovanje imenima, a ne lokacijama numeričkih bitova, i time smanjuje količinu grešaka. **EnumSet** vas sprečava i da nehotice dodate nove lokacije indikatora, što bi moglo da prouzrokuje ozbiljne greške koje se teško pronalaze. Koristite klasu **BitSet** umesto klase **EnumSet** samo u slučaju da sve do trenutka izvršavanja programa ne znate broj indikatora ili ako nije praktično dodeliti im imena ili vam treba neka od specijalnih operacija klase **BitSet** (pročitajte JDK dokumentaciju klasa **BitSet** i **EnumSet**).

Sažetak

Ima onih koji tvrde da je biblioteka kontejnera najvažnija biblioteka objektno orijentisanog jezika. U većini programa kontejneri se upotrebljavaju više od ijedne druge komponente biblioteke. Neki jezici (Python, na primer) imaju ugrađene čak i osnovne kontejnerske komponente (liste, mape i skupove).

Kao što ste videli u poglavlju *Čuvanje objekata*, pomoću kontejnera se može uraditi mnogo toga zanimljivog, a bez mnogo napora. Međutim, doći će trenutak kada ćete morati da znate više o kontejnerima kako biste mogli pravilno da ih upotrebljavate – konkretno, morate znati dovoljno o operacijama transformisanja ključeva (heširanja) da biste napisali sopstvenu metodu **hashCode**( ) (a morate znati i kada je to neophodno), i morate znati dovoljno o raznim realizacijama kontejnera da biste izabrali onu koja vam treba. U ovom poglavlju objasnili smo te pojmove i razmotrili još neke korisne detalje o biblioteci kontejnera. U ovom trenutku trebalo bi da ste prilično dobro pripremljeni za korišćenje Java kontejnera u svakodnevnim programerskim zadacima.

Biblioteku kontejnera je teško projektovati (to važi za većinu dizajnerskih problema s bibliotekama). U C++-u, kontejnerske klase su obuhvatale mnogo različitih klasa. To je bilo bolje od onoga što je postojalo pre kontejnerskih klasa jezika C++ (a to je ništa), ali se nije moglo lepo prevesti u Javu. Druga je krajnost kontejnerska biblioteka koja se sastoji od jedne jedine klase, container, koja istovremeno radi kao linearna sekvenca i asocijativni niz. Javina biblioteka kontejnera održava neku ravnotežu: ima svu funkcionalnost koju očekujete od zrele biblioteke kontejnera, ali ju je lakše naučiti i lakše se koristi od kontejnerskih klasa jezika C++ i ostalih sličnih biblioteka kontejnera. Rezultat ponegde izgleda čudno. Za razliku od nekih drugih odluka sprovedenih u starim Java bibliotekama, te čudnovatosti nisu nastale slučajno, nego su posledica odluka pažljivo odvaganih kako bi se postigao kompromis po pitanju složenosti.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Javin ulazno-izlazni sistem

Stvaranje dobrog ulazno-izlaznog (U/I) sistema jedan je od težih zadataka za projektanta jezika. Posledica toga je veliki broj različitih pristupa.

IZGLEDA DA JE PROBLEM UZETI U OBZIR SVE MOGUĆNOSTI. NE SAMO DA POSTOJE RAZLIČITI izvori ili ponori podataka U/I sistema s kojima možete da radite (datoteka, konzola, mrežne veze itd.), već ima i raznih načina za to (sekvencijalni, nasumičan pristup, baferi, binarni, znakovni, po redovima, po rečima itd.).

Projektanti Javinih biblioteka pristupili su rešavanju ovog problema tako što su napravili velikog broja klasa. Zapravo, Javin U/I sistem ima toliko mnogo klasa da na prvi pogled može da izgleda zastrašujuće (ironično, ali projekat Javinog U/I sistema zapravo sprečava eksploziju broja klasa). Napravljena je i velika izmena u U/I biblioteci nakon Jave 1.0, kada je prvobitna binarno orijentisana biblioteka dopunjena znakovno orijentisanim, Unicode U/I klasama. Klase **nio** (akronim od „new I/O", i to ćemo ime koristiti još godinama iako su one bile uvedene u Javu 1.4 i stoga su već „stare") dodate su radi poboljšanja performansi i funkcionalnosti. Usled toga postoji prilično veliki broj klasa koje treba savladati da bi se stekla dovoljno dobra osnovna znanja o Javinom U/I sistemu. Prilično je važno poznavati istorijat razvoja U/I biblioteke, čak i ako je vaša prva reakcija: „Ne gnjavi me istorijom, već mi pokaži kako se koristi"! Problem je to što će vas bez predznanja iz istorije ubrzo zbuniti neke klase i nećete znati ni kada da ih koristite, ni kada ne.

Ovo poglavlje je uvod u razne U/I klase u standardnoj Javinoj biblioteci i u način njihovog korišćenja.

Klasa **File**

Pre nego što se pozabavimo klasama koje i čitaju i upisuju podatke u tokove, proučićemo uslužne klase koje postoje u biblioteci, a pomažu u radu s datotekama.

Klasa **File** ima zbunjujuće ime: možete se prevariti i pomisliti da se odnosi na datoteku. Bolje ime za klasu bilo bi FilePath. Ona može da predstavlja *ime* određene datoteke ili *imena* skupa datoteka u direktorijumu. Ako je reč o skupu datoteka, možete da ga potražite metodom **list( )** koja vraća niz podataka tipa **String**. Vraćanje niza umesto neke prilagodljive kontejnerske klase opravdano je zato što je broj elemenata nepromenljiv, a ako želite drugačiji spisak sadržaja direktorijuma, treba da napravite drugačiji objekat klase **File**. U ovom odeljku prikazuje se korišćenje ove klase i njoj pridruženog interfejsa **FilenameFilter**.

Listanje direktorijuma

Pretpostavićemo kako hoćete da vidite spisak sadržaja direktorijuma. Objekat klase **File** možete upotrebiti na dva načina. Ako pozovete metodu **list( )** bez argumenata, dobićete kompletnu listu datoteka u direktorijumu koji opisuje taj objekat. Ukoliko želite ograničenu listu, npr. sve datoteke s nastavkom **.java**, upotrebite „filtar direktorijuma", klasu koja pokazuje kako se biraju objekti klase **File** koji će se prikazivati.

Evo kratkog primera. Obratite pažnju na to da se rezultati veoma lako uređuju po abecednom redosledu pomoću metode **java.util.Arrays.sort( )** i komparatora **String.CASE_INSENSITIVE_ORDER**:

```
//: ui/ListanjeDirektorijuma.java
// Prikazuje spisak direktorijuma pomoću regularnih izraza.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class ListanjeDirektorijuma {
  public static void main(String[] args) {
    File putanja = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = putanja.list();
    else
      lista = putanja.list(new FiltarImena(args));
    Arrays.sort(lista, String.CASE_INSENSITIVE_ORDER);
    for(String stavkaDir : lista)
      System.out.println(stavkaDir);
  }
}
class FiltarImena implements FilenameFilter {
  private Pattern uzorak;
  public FiltarImena(String regiz) {
     uzorak = Pattern.compile(regiz);
  }
  public boolean accept(File dir, String ime) {
    return uzorak.matcher(ime).matches( );
  }
} /* Ispis:
PrimerZaDirektorijum.java
ListanjeDirektorijuma.java
ListanjeDirektorijuma2.java
ListanjeDirektorijuma3.java
* ///:~
```

Klasa **FiltarImena** realizuje interfejs **FilenameFilter**. Korisno je videti koliko je interfejs **FilenameFilter** jednostavan:

```
public interface FilenameFilter {
  boolean accept(File dir, String ime);
}
```

Ova klasa postoji samo da bi metodi **list( )** pružila metodu **accept( )**, da bi **list( )** mogla „povratno da pozove" **accept( )** i tako utvrdi koja imena datoteka bi trebalo da se nađu u listi. Stoga se ovakva struktura često naziva *povratni poziv* (engl. *callback*). Konkretnije, ovo je primer projektnog obrasca *Strategy* (Strategija), pošto **list( )** realizuje tek osnovnu funkcionalnost, a vi dajete *Strategy* u obliku interfejsa **FilenameFilter** koji dovršava algo-

ritam potreban metodi **list( )** za pružanje usluge. Pošto je argument metode **list( )** objekat klase **FilenameFilter**, to znači da joj možete proslediti objekat bilo koje klase koja realizuje interfejs **FilenameFilter**, i tako (čak i u vreme izvršavanja) odrediti način ponašanja metode **list( )**. Svrha Strategije je da se obezbedi prilagodljivost koda.

Argument metode **accept( )** mora da bude objekat klase **File** koji predstavlja direktorijum, kao i **String** koji sadrži ime datoteke. Zapamtite da metoda **list( )** poziva metodu **accept( )** za svako ime datoteke u direktorijumu, da bi videla koje od njih treba uključiti u listu. Na to ukazuje rezultat metode **accept( )** koji je tipa **boolean**.

Metoda **accept( )** koristi metodu **matcher(ime)** da bi proverila da li se regularni izraz **regiz** podudara sa **ime**nom datoteke. Rezultat metode **list( )** je niz koji se postepeno pravi pomoću metode **accept( )**.

Anonimne unutrašnje klase

Ovaj primer je savršen za prepravljanje pomoću anonimne unutrašnje klase (takve klase su opisane u poglavlju *Unutrašnje klase*). Kao prva promena, uvodi se metoda **filter( )** koja vraća referencu na **FilenameFilter**:

```
//: ui/ListanjeDirektorijuma2.java
// Koristi anonimne unutrašnje klase.
import java.util.regex.*;
import java.io.*;
import java.util.*;
public class ListanjeDirektorijuma2 {
  public static FilenameFilter filter(final String regiz) {
    // Pravljenje anonimne unutrašnje klase:
    return new FilenameFilter() {
      private Pattern uzorak = Pattern.compile(regiz);
      public boolean accept(File dir, String ime) {
         return uzorak.matcher(ime).matches( );
      }
    }; // Kraj anonimne unutrašnje klase
  }
  public static void main(String[] args) {
    File putanja = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = putanja.list();
    else
      lista = putanja.list(filter(args));
    Arrays.sort(lista, String.CASE_INSENSITIVE_ORDER);
    for(String stavkaDir : lista)
      System.out.println(stavkaDir);
  }
} /* Ispis:
PrimerZaDirektorijum.java
ListanjeDirektorijuma.java
ListanjeDirektorijuma2.java
ListanjeDirektorijuma3.java
*///:~
```

Obratite pažnju na to da argument metode **filter( )** mora da bude označen kao **final**. To je neophodno da bi anonimna unutrašnja klasa mogla da koristi objekat koji je izvan njene oblasti važenja.

Ovakav pristup je bolji zato što je klasa koja realizuje interfejs **FilenameFilter** sada čvrsto povezana s klasom **ListanjeDirektorijuma2**. Međutim, ovaj pristup se može dodatno poboljšati kada se definiše anonimna unutrašnja klasa kao argument metode **list( )**; u tom slučaju, program je još kraći:

```
//: ui/ListanjeDirektorijuma3.java
// Pravljenje anonimne unutrašnje klase "na licu mesta".
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;
public class ListanjeDirektorijuma3 {
  public static void main(final String[] args) {
    File putanja = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = putanja.list();
    else
      lista = putanja.list(new FilenameFilter() {
        private Pattern uzorak = Pattern.compile(args);
        public boolean accept(File dir, String ime) {
           return uzorak.matcher(ime).matches( );
        }
      });
    Arrays.sort(lista, String.CASE_INSENSITIVE_ORDER);
    for(String  stavkaDir  :  lista)
        System.out.println(stavkaDir);
  }
} /* Ispis:
PrimerZaDirektorijum.java
ListanjeDirektorijuma.java
ListanjeDirektorijuma2.java
ListanjeDirektorijuma3.java
*///:~
```

Argument metode **main( )** sada je označen kao **final**, pošto anonimna unutrašnja klasa direktno koristi **args**.

Ovo pokazuje kako anonimne unutrašnje klase omogućuju pravljenje specifičnih, unikatnih klasa za brzo rešavanje problema. Prednost ovog pristupa jeste to što je kôd koji rešava odredeni problem izolovan na jednom mestu. S druge strane, kôd nije uvek lako čitljiv, pa morate mudro da ga koristite.

Vežba 1: (3) Izmenite program **ListanjeDirektorijuma.java** (ili neku od njegovih varijanata) tako da **FilenameFilter** otvara i čita svaku datoteku (uslužnom metodom **net.mindview.util.TextFile**) i prihvata je na osnovu toga sadrži li ona bilo koji od argumenata s komandne linije koji slede iza imena datoteke.

Vežba 2: (2) Napravite klasu **ListanjeUređenogDirektorijuma** s konstruktorom koji prima objekat tipa **File** i od njegovih datoteka pravi uređenu listu direktorijuma. Klasi dodajte dve preklopljene metode **list( )**: prvu koja proizvodi celu listu i drugu koja proizvodi podskup te liste koji odgovara njenom argumentu (što je regularan izraz).

Vežba 3: (3) Izmenite program **ListanjeDirektorijuma.java** (ili neku od njegovih varijanata) tako da sabere veličine izabranih datoteka.

Uslužne metode za direktorijume

U programiranju se često obavljaju operacije nad skupovima datoteka, bilo u lokalnom direktorijumu bilo prolaskom kroz celo stablo direktorijuma. Dobro bi poslužila alatka koja proizvodi taj skup datoteka. Sledeća uslužna klasa proizvodi bilo niz **File** objekata u lokalnom direktorijumu (metodom **lokal( )**), bilo **List<File>** celog stabla direktorijuma počev od datog direktorijuma (metodom **prolazak( )**). (**File** objekti su korisniji od imena datoteka zato što sadrže više informacija). Datoteke se biraju na osnovu regularnog izraza koji zadajete:

```
//: net/mindview/util/Direktorijum.java
// Proizvodi sekvencu File objekata koji odgovaraju
// regularnom izrazu, bilo u lokalnom direktorijumu,
// bilo prolaskom kroz stablo direktorijuma.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

public final class Direktorijum {
  public static File[]
  lokal(File dir, final String regiz) {
    return dir.listFiles(new FilenameFilter() {
      private Pattern uzorak = Pattern.compile(regiz);
      public boolean accept(File dir, String ime) {
        return uzorak.matcher(
          new File(ime).getName()).matches();
      }
    });
  }
  public static File[]
  lokal(String putanja, final String regiz) { // Preklopljeno
    return lokal(new File(putanja), regiz);
  }
  // Dvojka za vraćanje para objekata:
  public static class InfoStabla implements Iterable<File> {
    public List<File> datoteke = new ArrayList<File>();
    public List<File> dirmi = new ArrayList<File>();
    // Podrazumevani iterabilni element je lista datoteka:
    public Iterator<File> iterator() {
      return datoteke.iterator();
    }
```

```
    void addAll(InfoStabla ostalo) {
      datoteke.addAll(ostalo.datoteke);
      dirmi.addAll(ostalo.dirmi);
    }
    public String toString() {
      return "direktorijumi: " + PPrint.pformat(dirmi) +
        "\n\ndatoteke: " + PPrint.pformat(datoteke);
    }
  }
  public static InfoStabla
  prolazak(String pocetak, String regiz) { // Počni rekurziju
    return rekurzDirme(new File(pocetak), regiz);
  }
  public static InfoStabla
  prolazak(File pocetak, String regiz) { // Preklopljeno
    return rekurzDirme(pocetak, regiz);
  }
  public static InfoStabla prolazak(File pocetak) { // Sve
    return rekurzDirme(pocetak, ".*");
  }
  public static InfoStabla prolazak(String pocetak) {
    return rekurzDirme(new File(pocetak), ".*");
  }
  static InfoStabla rekurzDirme(File pocDir, String regiz){
    InfoStabla result = new InfoStabla();
    for(File stavka : pocDir.listFiles()) {
      if(stavka.isDirectory()) {
        rezultat.dirmi.add(stavka);
        rezultat.addAll(rekurzDirme(stavka, regiz));
      } else // Regularna datoteka
        if(stavka.getName().matches(regiz))
          rezultat.datoteke.add(stavka);
    }
    return rezultat;
  }
  // Jednostavna provera validnosti:
  public static void main(String[] args) {
    if(args.length == 0)
    System.out.println(prolazak("."));
  else
    for(String arg : args)
      System.out.println(prolazak(arg));
  }
} ///:~
```

Metoda **lokal( )** koristi varijantu metode **File.list( )** nazvanu **listFiles( )** koja proizvodi niz objekata tipa **File**. Vidite da koristi i **FilenameFilter**. Ukoliko umesto niza želite listu, sami pretvorite njen rezultat u niz metodom **Arrays.asList( )**.

Metoda **prolazak( )** pretvara ime početnog direktorijuma u objekat tipa **File** i poziva **rekurzDirme( )** koja obavlja rekurzivni prolazak kroz direktorijume, prikupljajući sve

više informacija u svakoj rekurziji. Da bismo mogli da razlikujemo obične datoteke od direktorijuma, povratna vrednost je zapravo „n-torka" objekata – jedna lista koja sadrži obične datoteke i druga koja sadrži direktorijume. Polja su namerno deklarisana kao javna (**public**), pošto je svrha **InfoStabla** da prikupi objekte – da vraćate samo jednu listu, ne biste je napravili privatnom, pa ni vraćanje para objekata ne znači da ih morate napraviti privatnim. Obratite pažnju na to da **InfoStabla** realizuje **Iterable<File>** koji proizvodi datoteke, pa je „podrazumevana iteracija" po listi datoteka, dok iteraciju po direktorijumima zadajete sa **dirmi**.

Metoda **InfoStabla.toString( )** upotrebljava klasu „pretty printer" da bi se ispis lakše pregledao. Podrazumevane metode **toString( )** za kontejnere, štampaju sve elemente jednog kontejnera u istom redu. U velikim kolekcijama to postaje teško čitljivo, pa bi se moglo poželeti i neko drugo formatiranje. Sledeća alatka dodaje prelaske u novi red i uvlačenja za svaki element:

```
//: net/mindview/util/PPrint.java
// Pretty-printer za kolekcije
package net.mindview.util;
import java.util.*;

public class PPrint {
  public static String pformat(Collection<?> k) {
    if(k.size() == 0) return "[]";
    StringBuilder rezultat = new StringBuilder("[");
    for(Object elem : k) {
      if(k.size() != 1)
        rezultat.append("\n  ");
      rezultat.append(elem);
    }
    if(k.size() != 1)
      rezultat.append("\n");
    rezultat.append("]");
    return rezultat.toString();
  }
  public static void pprint(Collection<?> k) {
    System.out.println(pformat(k));
  }
  public static void pprint(Object[] k) {
    System.out.println(pformat(Arrays.asList(k)));
  }
} ///:~
```

Metoda **pformat( )** od kolekcije proizvodi formatiran znakovni niz, a **pprint( )** poziva **pformat( )** da to obavi. Imajte u vidu da se drugačije tretiraju posebni slučajevi kada nema elemenata, odnosno kada je samo jedan element. Postoji i verzija metode **pprint( )** za nizove.

Uslužne metode klase **Direktorijum** deo su paketa **net.mindview.util** i stoga lako dostupne. Evo primera kako se upotrebljavaju:

```
//: io/PrimerZaDirektorijum.java
// Primer upotrebe uslužnih metoda za Direktorijum.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PrimerZaDirektorijum {
  public static void main(String[] args) {
    // Svi direktorijumi:
    PPrint.pprint(Directory.prolazak(".").direktorijumi);
    // Sve datoteke koje počinju na 'T'
    for(File datoteka : Direktorijum.lokal(".", "T.*"))
      print(datoteka);
    print("----------------------");
    // Sve Java datoteke koje počinju na 'T':
    for(File datoteka : Direktorijum.prolazak(".", "T.*\\.java"))
      print(datoteka);
    print("======================");
    // Class datoteke koje sadrže "Z" ili "z":
    for(File datoteka : Direktorijum.prolazak(".",".*[Zz].*\\.class"))
      print(datoteka);
  }
} /* Ispis: (primer)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
----------------------
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
======================
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:~
```

Možda je potrebno da osvežite svoje znanje o regularnim izrazima, pa se vratite na poglavlje *Znakovni nizovi* da biste shvatili druge argumente u metodama **lokal**( ) i **prolazak**( ).

Da bismo otišli korak dalje, napravićemo alatku koja radi i jedno i drugo: prolazi kroz direktorijume *i* obraduje datoteke u njima u skladu s datim objektom tipa **Strategy** (ovo je još jedan primer projektnog obrasca *Strategy*):

```
//: net/mindview/util/ObradaDatoteka.java
package net.mindview.util;
import java.io.*;

public class ObradaDatoteka {
```

```
  public interface Strategy {
    void process(File datoteka);
  }
  private Strategy strategija;
  private String spolj;
  public ObradaDatoteka(Strategy strategija, String spolj) {
    this.strategija = strategija;
    this.spolj = spolj;
  }
  public void pocetak(String[] args) {
    try {
      if(args.length == 0)
        obradaStablaDirma(new File("."));
      else
        for(String arg : args) {
          File fileArg = new File(arg);
          if(fileArg.isDirectory())
            obradaStablaDirma(fileArg);
          else {
            // Dozvoli korisniku da izostavi nastavak imena datoteke:
            if(!arg.endsWith("." + spolj))
              arg += "." + spolj;
            strategija.process(
              new File(arg).getCanonicalFile());
          }
        }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void
  obradaStablaDirma(File koren) throws IOException {
    for(File datoteka : Direktorijum.prolazak(
        koren.getAbsolutePath(), ".*\\." + spolj))
      strategija.process(datoteka.getCanonicalFile());
  }
  // Primer upotrebe:
  public static void main(String[] args) {
    new ObradaDatoteka(new ObradaDatoteka.Strategy() {
      public void process(File datoteka) {
        System.out.println(datoteka);
      }
    }, "java").pocetak(args);
  }
} /* (Pokrenite da biste videli rezultate) *///:~
```

Interfejs **Strategy** ugnežden je u klasu **ObradaDatoteka**. Da biste ga realizovali, morate da implementirate **ObradaDatoteka.Strategy** i time čitaocu date veći kontekst. **ObradaDatoteka** pronalazi datoteke koje imaju određeni nastavak (**spolj** kao argument konstruktora) i predaje ih objektu **Strategy** (koji je takođe argument konstruktora).

Ako ne zadate argumente, u klasi **ObradaDatoteka** pretpostavlja se da želite da prođete kroz sve direktorijume koji izlaze iz tekućeg direktorijuma. S druge strane, možete zadati određenu datoteku, s nastavkom ili bez njega(ona će dodati nastavak ako bude potrebno), odnosno jedan ili više direktorijuma.

U metodi **main( )** vidite elementarni način upotrebe ove alatke; ona štampa imena svih Java datoteka izvornog koda u skladu sa argumentima koje zadate na komandnoj linijii.

Vežba 4: (2) Upotrebite **Direktorijum.prolazak( )** za sabiranje veličina svih datoteka u stablu direktorijuma čija imena odgovaraju određenom regularnom izrazu.

Vežba 5: (1) Izmenite program **ObradaDatoteka.java** tako da ispituje podudaranje s regularnim izrazom, a ne s nepromenljivim nastavkom imena datoteke.

Provera postojanja i pravljenje direktorijuma

Klasa **File** je više od pukog prikaza postojeće datoteke ili direktorijuma. Objekat klase **File** možete da koristite i za pravljenje novog direktorijuma ili celokupne putanje direktorijuma ako ona ne postoji. Možete da ispitujete i karakteristike datoteka (veličinu, datum poslednje promene, mogućnost za čitanje/upisivanje), utvrdujete da li objekat klase **File** predstavlja datoteku ili direktorijum i brišete datoteke. Sledeći program prikazuje neke od ostalih metoda klase **File** (potpun skup metoda potražite u HTML dokumentaciji na lokaciji *java.sun.com*):

```
//: ui/NapraviDirektorijume.java
// Prikazuje korišćenje klase File
// za pravljenje direktorijuma i rad s datotekama.
// {Args: NapraviDirektorijumeProba}
import java.io.*;

public class NapraviDirektorijume {
  private static void upotreba() {
    System.err.println(
    "Korišćenje:NapraviDirektorijume putanja1 ...\n" +
    "Pravi sve putanje \n" +
    "Korišćenje:NapraviDirektorijume -d putanja1 ...\n" +
    "Briše sve putanje \n" +
    "Korišćenje:NapraviDirektorijume -r putanja1 putanja2\n" +
    "Preimenuje putanju1 u putanju2";
    System.exit(1);
  }
  private static void podaciDatoteke(File f) {
    System.out.println(
      "Apsolutna putanja: " + f.getAbsolutePath() +
      "\n može da se čita: " + f.canRead() +
      "\n može da se upisuje: " + f.canWrite() +
      "\n ime: " + f.getName() +
      "\n roditelj: " + f.getParent() +
      "\n putanja: " + f.getPath() +
      "\n dužina: " + f.length() +
      "\n datum poslednje promene: " + f.lastModified());
```

```
    if(f.isFile())
      System.out.println("to je datoteka");
    else if(f.isDirectory())
      System.out.println("to je direktorijum");
  }
  public static void main(String[] args) {
    if(args.length < 1) upotreba();
    if(args.equals("-r")) {
      if(args.length != 3) upotreba();
      File
        stari = new File(args),
        novoIme = new File(args);
      stari.renameTo(novoIme);
      podaciDatoteke(stari);
      podaciDatoteke(novoIme);
      return; // Izlazak iz funkcije main
    }
    int broj = 0;
    boolean brisanje = false;
    if(args.equals("-d")) {
      broj++;
      brisanje = true;
    }
    broj--;
    while(++broj < args.length) {
      File f = new File(args[broj]);
      if(f.exists()) {
        System.out.println(f + " postoji");
        if(brisanje) {
          System.out.println("brišem..." + f);
          f.delete();
        }
      }
      else { // Ne postoji
        if(!brisanje) {
          f.mkdirs();
          System.out.println("napravljena " + f);
        }
      }
      podaciDatoteke(f);
    }
  }
} /* Ispis: (80% podudaranja)
napravljeno NapraviDirektorijumeProba
Apsolutna putanja: d:\aaa-TIJ4\code\io\MakeDirectoriesTest
 može da se čita: true
 može da se upisuje: true
 ime: NapraviDirektorijumeProba
 roditelj: null
 putanja: NapraviDirektorijumeProba
 dužina: 0
```

```
 datum poslednje promene: 1101690308831
to je direktorijum
*///:~
```

U metodi **podaciDatoteke( )** možete da vidite razne metode za ispitivanje datoteka koje se koriste za prikazivanje informacija o datoteci ili putanji direktorijuma.

Prva metoda koja se koristi je **renameTo( )**; ona preimenuje (ili premešta) datoteke na potpuno novu putanju predstavljenu argumentom tipa **File**. Ovaj pristup se može koristiti i za direktorijume proizvoljne dužine.

Ako budete eksperimentisali s gornjim programom, ustanovićete da možete napraviti putanju direktorijuma proizvoljne složenosti zato što će **mkdirs( )** obaviti sav posao umesto vas.

Vežba 6: (5) Iskoristite program **ObradaDatoteka** i pronađite u nekom podstablu direktorijuma sve datoteke Java izvornog koda koje su promenjene nakon određenog datuma.

Ulaz i izlaz

U U/I bibliotekama često se koristi apstraktan pojam – *tok* podataka (engl. *stream*). Tok predstavlja neki izvor ili ponor podataka u obliku objekta koji je u stanju da daje ili prima delove podataka. Tok skriva detalje o tome šta se dešava s podacima unutar stvarnog U/I uređaja.

Klase Javine biblioteke za U/I podeljene su na ulazne i izlazne, što možete videti u hijerarhiji Javinih klasa u dokumentaciji na Webu. Zbog nasleđivanja, sve klase izvedene iz **InputStream** ili **Reader** imaju osnovnu metodu **read( )** za čitanje jednog bajta ili niza bajtova. Slično, sve klase izvedene iz **OutputStream** ili **Writer** imaju osnovnu metodu **write( )** za upisivanje jednog bajta ili niza bajtova. Međutim, obično nećete koristiti te metode, jer one postoje da bi ih koristile druge klase koje obezbeđuju korisnije metode. Retko ćete praviti objekat toka korišćenjem samo jedne klase; umesto toga, grupisaćete više objekata da biste postigli željenu funkcionalnost. Mogućnost da se napravi više objekata radi dobijanja jednog toka, glavni je krivac za nerazumljivost Javine biblioteke tokova.

Korisno je kategorizovati klase prema njihovoj funkcionalnosti. U Javi 1.0 projektanti biblioteka su se vodili idejom da se sve klase koje imaju veze sa ulazom, izvode iz klase **InputStream**, a da sve klase povezane sa izlazom nasleđuju klasu **OutputStream**.

Kao i dosad u knjizi, pokušaću da prikažem klase, ali ću podrazumevati da ćete sve detalje, npr. kompletne spiskove metoda, tražiti u dokumentaciji na Webu.

Vrste ulaznih tokova

Klasa **InputStream** treba da predstavlja ulazne tokove iz različitih izvora. Ti izvori mogu da budu:

1. niz bajtova
2. objekat klase **String**
3. datoteka

4. cev (engl. *pipe*) koja radi kao prava cev: podaci se stavljaju na jednom kraju, a izlaze na drugom
5. niz drugih tokova koji se mogu objediniti u jedan tok
6. drugi izvori, npr. veza sa Internetom (o ovome se govori u knjizi *Thinking in Enterprise Java*, dostupnoj na adresi *www.MindView.net*).

Svaki od ovih tipova ulaza povezan je sa određenom potklasom klase **InputStream**. Pored toga, **FilterInputStream** takođe je vrsta ulaznog toka koji služi kao osnova za dopunske klase, tj. za dodeljivanje atributa ili korisnih interfejsa ulaznim tokovima. O ovome će biti reči u nastavku.

Tabela U/I-1 Tipovi ulaznih tokova

Klasa	Funkcija	Argumenti konstruktora Kako se koristi
ByteArrayInputStream	Omogućuje da se memorijski blok koristi kao ulazni tok	Bafer iz kojeg se izvlače bajtovi. Kao izvor podataka. Povežite je sa objektom klase **FilterInputStream** da biste dobili korisan interfejs.
StringBufferInputStream	Konvertuje **String** u **InputStream**	**String**. Stvarna realizacija zapravo koristi **StringBuffer** Kao izvor podataka. Povežite je sa objektom klase **FilterInputStream** da biste dobili korisan interfejs.
FileInputStream	Čitanje informacija iz datoteke	**String** koji predstavlja ime datoteke, ili objekti klase **File** ili **FileDescriptor** Kao izvor podataka. Povežite je sa objektom klase **FilterInputStream** da biste dobili korisan interfejs.
PipedInputStream	Daje podatke koji se upisuju u pridruženi **PipedOutputStream**. Realizuje slanje kroz cevovod.	**PipedOutputStream** Kao izvor podataka u višenitnom radu. Povežite je sa objektom klase **FilterInputStream** da biste dobili korisan interfejs.
SequenceInputStream	Konvertuje dva ili više objekata klase **InputStream** u jedan ulazni tok.	Dva objekta klase **InputStream** ili **Enumeration** za kontejner objekata klase **InputStream** Kao izvor podataka. Povežite je sa objektom klase **FilterInputStream** da biste dobili korisan interfejs.
FilterInputStream	Apstraktna klasa, interfejs za dopunske klase koje obezbeđuju korisne funkcije za ulazne tokove. Pogledajte tabelu U/I -3.	Pogledajte tabelu U/I -3. Pogledajte tabelu U/I -3.

Vrste izlaznih tokova (OutputStream)

Ova kategorija obuhvata klase koje odlučuju gde će se usmeriti izlaz: u niz bajtova (ali ne i **String**; znakovni niz možete da napravite korišćenjem niza bajtova), datoteku ili cev.

Pored toga, **FilterOutputStream** služi kao osnovna klasa za dopunske klase koje dodaju atribute ili korisne interfejse u izlazne tokove. O ovome će biti reči u nastavku.

Tabela U/I-2 Vrste izlaznih tokova

Klasa	Funkcija	Argumenti konstruktora Kako se koristi
ByteArray-OutputStream	Pravi bafer u memoriji. Svi podaci koje šaljete u tok smeštaju se u taj bafer.	Opciona početna veličina bafera. Za zadavanje odredišta podataka. Povežite je sa objektom klase **Filter-OutputStream** da biste dobili koristan interfejs.
FileOutput-Stream	Slanje informacija u datoteku.	Znakovni niz koji predstavlja ime datoteke, ili objekte klase **File** ili **File-Descriptor** Za zadavanje odredišta podataka. Povežite je sa objektom klase **Filter-OutputStream** da biste dobili koristan interfejs.
PipedOutput-Stream	Sve informacije koje upisujete u ovaj tok automatski završavaju kao ulaz za odgovarajući **Piped-InputStream**. Realizuje prijem iz cevovoda.	**PipedInputStream** Za zadavanje odredišta podataka u višenitnom radu. Povežite je sa objektom klase **FilterOutputStream** da biste dobili koristan interfejs.
FilterOutput-Stream	Apstraktna klasa, osnova za dopunske klase koje obezbeđuju korisne funkcije drugim izlaznim tokovima. Pogledajte tabelu U/I-4.	Pogledajte tabelu U/I-4. Pogledajte tabelu U/I-4.

Dodavanje atributa i korisnih interfejsa

Dekoratori su predstavljeni u poglavlju *Generički tipovi*, na stranici 484. Javina U/I biblioteka zahteva veliki broj kombinacija funkcija i zato se koristi projektni obrazac *Decorator* (Dekorator).[1] Zbog toga u Javinoj U/I biblioteci postoje filtarske klase: apstraktna filtarska klasa je osnovna klasa za sve dekoratore. Decorator mora da ima isti interfejs kao objekat koji omotava, ali može i da ga proširi, što se dešava u nekim filtarskim klasama.

Ovakav obrazac, ipak, ima i nedostatak. Projektni obrasci *Decorator* omogućuju mnogo veću prilagodljivost programa (pošto se atributi lako kombinuju), ali povećavaju složenost koda. Javinu U/I biblioteku nije pogodno koristiti zato što morate da pravite veliki broj klasa, tj. jezgro ulaza/izlaza i razne dekoratore da biste dobili jedan U/I objekat koji želite.

[1] Nije jasno da li je to bila dobra projektantska odluka, naročito kada se uzme u obzir jednostavnost U/I biblioteka u drugim jezicima. Ali odluka se time opravdava.

Klase **FilterInputStream** i **FilterOutputStream** koje nemaju preterano intuitivna imena, obezbeđuju dekoratorski interfejs za kontrolisanje određenog ulaznog ili izlaznog toka. **FilterInputStream** i **FilterOutputStream** su apstraktne klase izvedene iz osnovnih klasa U/I biblioteke, **InputStream** i **OutputStream**, što je i ključni zahtev dekoratora (da bi obezbedio zajednički interfejs za sve omotane objekte).

Filtriranje ulaznog toka

Klase koje realizuju **FilterInputStream** izvršavaju dva prilično različita zadatka. **DataInputStream** omogućuje čitanje različitih prostih tipova, kao i objekata klase **String** – za to služe metode koje počinju rečju read (čitanje), npr. **readByte( )**, **readFloat( )** itd. Ova klasa, uz srodnu klasu **DataOutputStream**, omogućuje premeštanje prostih tipova podataka s jednog mesta na drugo pomoću toka. Ta mesta određuju klase iz tabele U/I-1.

Preostale klase menjaju način internog ponašanja ulaznog toka: određuju da li je on baferisan ili nije, da li prati broj redova koje čita (i omogućuje da zatražite red po broju ili da zadate broj reda) i da li se jedan znak može vratiti u bafer. Poslednje dve klase prilično liče na podršku za pravljenje prevodioca (verovatno su dodate da bi se podržao eksperimentalni „Javin prevodilac pisan na Javi"), pa ih verovatno nećete koristiti u svakodnevnom programiranju.

Gotovo uvek morate da baferišete ulaz, bez obzira na to s kojim U/I uređajem ste povezani, pa bi bilo korisnije da U/I biblioteka ima specijalan slučaj (ili prosto poziv metode) za nebaferisani ulaz umesto za baferisan ulaz.

Tabela U/I-3 Vrste filtriranih ulaznih tokova

Klasa	Funkcija	Argumenti konstruktora Kako se koristi
DataInputStream	Koristi se u kombinaciji s klasom **DataOutputStream**, da bi se iz toka na prenosiv način čitali prosti tipovi (**int**, **char**, **long** itd.).	**InputStream** Sadrži kompletan interfejs koji omogućuje čitanje prostih tipova.
BufferedInputStream	Koristite je za smanjenje broja fizičkih operacija čitanja svaki put kada vam zatreba još podataka. Ovom klasom kažete toku: „Upotrebi bafer".	**InputStream**, uz opcionu veličinu bafera. Ne obezbeđuje interfejs sama po sebi, već samo procesu dodaje bafer. Pridružite joj objekat koji realizuje interfejs.
LineNumberInputStream	Prati brojeve redova u ulaznom toku; možete da pozovete **getLineNumber( )** i **setLineNumber(int)**	**InputStream** Samo dodaje numerisanje redova, pa ćete joj verovatno pridružiti objekat koji realizuje interfejs.
PushbackInputStream	Ima skladište za vraćanje jednog bajta u koji možete da smestite poslednji pročitani znak.	**InputStream** Obično se koristi u analizatoru za prevodioca. Verovatno je nećete koristiti.

Filtriranje izlaznog toka

Odgovarajuća klasa za **DataInputStream** je **DataOutputStream**; ona formatira sve proste tipove i objekte klase **String** u tok na takav način da ih može čitati svaki tok tipa **DataInputStream** na bilo kojoj platformi. Sve njene metode počinju rečju write (upis), npr. **writeByte( )**, **writeFloat( )** itd.

Prvobitna svrha klase **PrintStream** bila je ispis svih prostih tipova podataka i objekata klase **String** u formatu koji se može prikazivati ljudima. To se razlikuje od **DataOutputStream**, čiji je cilj da elemente podataka postavi u binarni niz na takav način da ih **DataInputStream** može rekonstruisati nakon prenosa.

Dve važne metode klase **PrintStream** su **print( )** i **println( )** koje se redefinišu za ispis svih tipova. Razlika između ove dve metode je to što **println( )** prelazi u novi red po završetku štampanja.

Klasa **PrintStream** može da bude problematična zato što hvata sve izuzetke tipa **IOException** (metodom **checkError( )** morate eksplicitno da proverite da li je nastala do greška). Klasa **PrintStream** nije primenljiva za neke jezike i ne radi s prelomima redova na način nezavisan od platforme (ti problemi su rešeni pomoću klase **PrintWriter** o kojoj će biti reči kasnije.

BufferedOutputStream je dopunska klasa koja nalaže toku da koristi baferisanje da se podaci ne bi fizički upisivali kad god se upisuje u tok. Verovatno ćete ovu klasu uvek koristiti za ispisivanje rezultata.

Tabela U/I-4 Vrste filtriranih izlaznih tokova

Klasa	Funkcija	Argumenti konstruktora Kako se koristi
DataOutputStream	U kombinaciji s klasom **DataInputStream** koristi se za upisivanje prostih tipova (**int**, **char**, **long** itd.) u tok na prenosiv način.	**OutputStream** Sadrži kompletan interfejs koji omogućuje upisivanje prostih tipova
PrintStream	Koristi se za formatiranje izlaza. Za razliku od klase **DataOutputStream** koja *skladišti* podatke, **PrintStream** ih *ispisuje*.	**OutputStream**, uz opcioni argument tipa **boolean** koji određuje da li se bafer prazni prilikom prelaska u sledeći red. Trebalo bi da bude „konačno" omotavanje za objekat klase **OutputStream**. Verovatno ćete je dosta koristiti.
BufferedOutputStream	Koristite je da biste izbegli fizičko upisivanje pri svakom slanju podataka. Ovom klasom kažete toku da „koristi bafer". Za pražnjenje sadržaja bafera možete da pozovete metodu **flush( )**	**OutputStream**, uz opcionu veličinu bafera. Ne obezbeđuje interfejs sama po sebi već samo procesu dodaje bafer. Dodeli te joj objekat koji realizuje interfejs

Klase za čitanje i upisivanje

U Javi 1.1 značajno su izmenjene osnovne biblioteke U/I tokova. Kada vidite klase **Reader** i **Writer**, prvo ćete pomisliti (kao i ja) da zamenjuju klase **InputStream** i **OutputStream**.

Ipak, nije tako. Iako su neki elementi prvobitne biblioteke tokova zastareli (ako ih koristite, dobićete upozorenje od prevodioca), klase **InputStream** i **OutputStream** i dalje obezbeđuju važne funkcije binarno orijentisanog ulaza/izlaza, dok klase **Reader** i **Writer** obezbeđuju znakovno orijentisani Unicode ulaz/izlaz. Pored toga:

1. U Javu 1.1 dodate su nove klase u hijerarhiji ulaznih i izlaznih tokova, pa je očigledno da klase **InputStream** i **OutputStream** nisu zamenjene.
2. Postoje prilike kada morate da koristite klase iz binarne hijerarhije *u kombinaciji* s klasama u znakovnoj hijerarhiji. Da bi se to postiglo, postoje posredničke klase: **InputStreamReader** konvertuje **InputStream** u **Reader**, a **OutputStreamWriter** konvertuje **OutputStream** u **Writer**.

Hijerarhije klasa **Reader** i **Writer** postoje prvenstveno zbog internacionalizacije. Stara U/I hijerarhija podržava samo osmobitne binarne tokove, odnosno ne radi dobro sa šesnaestobitnim Unicode znakovima. Pošto se Unicode koristi za internacionalizaciju (a Javin izvorni tip **char** je šesnaestobitni Unicode), hijerarhije klasa **Reader** i **Writer** su dodate da bi se podržao standard Unicode u svim U/I operacijama. Pored toga, projektovane su nove biblioteke koje rade brže nego stare.

Izvori i ponori podataka

Za gotovo sve originalne ulazno/izlazne Javine klase postoje ekvivalentne potklase **Reader** i **Writer** koje obezbeđuju osnovni rad sa Unicode znakovima. Međutim, postoje prilike kada su binarni ulazni i izlazni tokovi ispravno rešenje. To se naročito odnosi na biblioteke **java.util.zip** koje su binarno, a ne znakovno orijentisane. Zbog toga je najmudrije *pokušati* koristiti klase **Reader** i **Writer** kad god je to moguće, a situacije kada morate da koristite binarno orijentisane biblioteke otkrićete sami zato što kôd neće moći da se prevede.

Evo tabele koja prikazuje vezu između izvora i ponora informacija (tj. mesta s kojih podaci fizički stižu i kuda odlaze) u dve hijerarhije.

Izvori i ponori: klasa Jave 1.0	Odgovarajuća klasa Jave 1.1
InputStream	Reader Konvertor: **InputStreamReader**
OutputStream	Writer Konvertor: **OutputStreamWriter**
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream (zastarelo)	**StringReader**
(nema odgovarajuće klase)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

U većini slučajeva otkrićete da su interfejsi dve različite hijerarhije slični, ako ne i istovetni.

Menjanje ponašanja toka

U klasama **InputStream** i **OutputStream**, tokovi su prilagođeni specifičnim potrebama pomoću dopunskih potklasa klasa **FilterInputStream** i **FilterOutputStream**. I u hijerarhiji klasa **Reader** i **Writer** koristi se ovaj pristup, ali ne na potpuno isti način.

U sledećoj tabeli veza nije toliko direktna kao u prethodnoj tabeli. To je posledica organizacije klasa: za razliku od toka **BufferedOutputStream** koji je potklasa klase **FilterOutputStream**, **BufferedWriter** *nije* potklasa klase **FilterWriter** (koja, iako je apstraktna, nema potklase, pa izgleda da je uvedena ili zbog nekih kasnijih proširenja, ili samo da se ne biste pitali zašto je nema). Međutim, interfejsi klasa su veoma slični.

Filtri: klasa Jave 1.0	**Odgovarajuća klasa Jave 1.1**
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (apstraktna klasa bez potklasa)
BufferedInputStream	**BufferedReader** (ima i metodu **readLine( )**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	Koristite **DataInputStream**, osim kada morate da koristite metodu **readLine( )**. U tom slučaju bi trebalo da koristite klasu **BufferedReader**
PrintStream	**PrintWriter**
LineNumberInputStream (zastarelo)	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (koristite konstruktor koji prihvata argument tipa **Reader**)
PushBackInputStream	**PushBackReader**

Jedno pravilo je sasvim jasno: kad god želite da koristite **readLine( )**, to više ne bi trebalo da radite pomoću klase **DataInputStream** (o tome ćete dobiti upozorenje tokom prevođenja), već pomoću klase **BufferedReader**. U svim ostalim slučajevima, **DataInputStream** se i dalje preporučuje.

Da bi se olakšao prelazak na korišćenje klase **PrintWriter**, njeni konstruktori prihvataju sve objekte tipa **OutputStream**, ali i **Writer**. Međutim, **PrintWriter** ne omogućuje ništa bolje formatiranje od klase **PrintStream**; njihovi interfejsi su doslovno isti.

Konstruktor klase **PrintWriter** takođe ima opciju za automatsko pražnjenje internih bafera, što se dešava nakon svakog poziva metode **println( )** ako je indikator postavljen u konstruktoru.

Klase koje nisu promenjene

Neke klase su ostale nepromenjene i u Javi 1.1:

Naročito se bez ikakvih izmena koristi klasa **DataOutputStream**, pa za čuvanje i pronalaženje podataka u prenosivom formatu možete da primenite hijerarhije **InputStream** i **OutputStream**.

Klase Jave 1.0 bez odgovarajućih klasa u Javi 1.1
DataOutputStream
File
RandomAccessFile
SequenceInputStream

Poseban slučaj: klasa RandomAccessFile

Klasa **RandomAccessFile** koristi se za datoteke sa zapisima poznate veličine, pa s jednog zapisa na drugi možete da se pomerate metodom **seek( )**, a zatim da čitate ili menjate zapise. Zapisi ne moraju da budu iste veličine; treba samo da bude moguće odrediti njihovu veličinu i mesto u datoteci u kojoj se nalaze.

Isprva je pomalo teško poverovati da **RandomAccessFile** nije deo hijerarhija klasa **InputStream** ili **OutputStream**. Međutim, ona nema nikakve veze s tim hijerarhijama osim što realizuje interfejse **DataInput** i **DataOutput** (koje realizuju i **DataInputStream** i **DataOutputStream**). Ona čak ne koristi ni funkcije postojećih klasa **InputStream** ili **OutputStream** – to je sasvim posebna klasa, napisana kompletno od početka, sa sopstvenim, većinom osnovnim, metodama. **RandomAccessFile** nije deo hijerarhije zato što se ponaša sasvim drugačije od ostalih U/I tipova: omogućuje kretanje unapred i unazad po datoteci. U svakom slučaju, ona postoji kao samostalan, direktan naslednik klase **Object**.

U osnovi, **RandomAccessFile** radi kao **DataInputStream** u kombinaciji s klasom **DataOutputStream**, uz metode **getFilePointer( )** za pronalaženje pozicije unutar datoteke, **seek( )** za prelazak na novu poziciju unutar datoteke i **length( )** za određivanje maksimalne veličine datoteke. Osim toga, njeni konstruktori zahtevaju još jedan argument (isti kao za funkciju **fopen( )** u jeziku C) koji ukazuje na to da li se radi samo o nasumičnom čitanju (**r**) ili o čitanju i upisivanju (**rw**). Nema podrške za datoteke u koje može samo da se upisuje, što ukazuje na to da bi klasa **RandomAccessFile** mogla da radi dobro i da je izvedena iz klase **DataInputStream**.

Metode za pretraživanje dostupne su samo unutar klase **RandomAccessFile** koja radi isključivo s datotekama. Klasa **BufferedInputStream** omogućuje označavanje pozicije (čija se vrednost čuva u internoj promenljivoj) metodom **mark( )** i vraćanje na tu poziciju metodom **reset( )**, ali su te metode ograničene i nisu naročito korisne.

Od Jave 1.4, umesto većine, ako ne i svih funkcija klase **RandomAccessFile** koriste se **nio** *datoteke preslikane u memoriju* (engl. *memory-mapped files*), koje će biti opisane u nastavku poglavlja.

Tipične primene U/I tokova

Iako klase U/I tokova možete da kombinujete na razne načine, verovatno ćete upotrebljavati samo nekoliko kombinacija. Sledeći primer se može koristiti kao osnovna referenca; on prikazuje pravljenje i korišćenje tipičnih U/I konfiguracija.

U ovim primerima, obradu izuzetaka pojednostavićemo tako što ćemo proslediti izuzetke na konzolu, ali to je prikladno samo u malim primerima i uslužnim klasama. Vi ćete u kodu morati da primenite bolju obradu izuzetaka.

Baferisana ulazna datoteka

Kako biste otvorili datoteku za unos znakova, upotrebite **FileInputReader** sa objektima klasa **String** ili **File** kao imenima datoteka. Da bi se povećala brzina, ta datoteka bi trebalo da bude baferisana, pa dobijenu referencu prosledite konstruktoru klase **BufferedReader**. Pošto ta klasa ima i metodu **readLine( )**, to je konačan objekat i interfejs iz koga ćete čitati. Kada stignete do kraja datoteke, metoda **readLine( )** vraća **null**, što se koristi kao uslov izlaska iz petlje **while**.

```
//: io/BaferisanaUlaznaDatoteka.java
import java.io.*;

public class BaferisanaUlaznaDatoteka {
  // Baci izuzetke na konzolu:
  public static String
  read(String imedatoteke) throws IOException {
    // Čitanje ulaza red po red:
    BufferedReader in = new BufferedReader(
      new FileReader(imedatoteke));
    String s;
    StringBuilder sb = new StringBuilder();
    while((s = in.readLine())!= null)
      sb.append(s + "\n");
    in.close();
    return sb.toString();
  }
  public static void main(String[] args)
  throws IOException {
    System.out.print(read("BaferisanaUlaznaDatoteka.java"));
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Objekat **sb** tipa **StringBuilder** koristi se za sakupljanje celokupnog sadržaja datoteke (uključujući i znakove za prelazak u novi red koji se moraju dodati pošto ih **readLine( )** odseca). Na kraju se poziva metoda **close( )** za zatvaranje datoteke.[2]

Vežba 7: (2) Otvorite tekstualnu datoteku tako da je možete čitati red po red. Pročitajte svaki red kao znakovni niz i stavite ga u listu tipa **LinkedList**. Ispišite sve redove povezane (ulančane) liste obrnutim redosledom.

Vežba 8: (1) Promenite vežbu 7 tako da se ime datoteke koju čitate prosleđuje kao argument s komandne linije.

Vežba 9: (1) Promenite vežbu 8 tako da pretvara sva slova redova teksta iz liste **ArrayList** u velika i šalje rezultate u tok **System.out**.

---

[2] Metoda **close( )** biće pozvana automatski tokom izvršavanja metode **finalize( )**, što bi trebalo da se desi pri izlasku iz programa (bez obzira na to da li će se smeće skupiti). Međutim, na drugim mestima u knjizi objašnjeno je da to ne radi kako su projektanti Jave očekivali (tj. prosto ne radi), pa je jedini bezbedan pristup da se eksplicitno pozove metoda **close( )** za datoteke.

Vežba 10: (2) Promenite vežbu 8 tako da kao dodatne argumente s komandne linije prihvata reči koje će tražiti u datoteci. Ispišite sve redove u kojima se reči pronađu.

Vežba 11: (2) U primeru **unutrasnjeklase/UpravljanjeStaklenomBastom.java, KontrolerStakleneBaste** sadrži fiksiran skup događaja. Promenite program tako da čita događaje i njihova relativna vremena iz tekstualne datoteke. (nivo težine 8): Za pravljenje događaja upotrebite projektni obrazac *Factory Method* (Proizvodna metoda) pronaći ćete ga u knjizi *Thinking in Patterns (with Java)* koja se može preuzeti s lokacije *www.MindView.net.*

Čitanje iz memorije

U ovom odeljku se na osnovu **String** rezultata metode **BufferedInputFile.read( )** pravi objekat klase **StringReader.** Potom se poziva metoda **read( )** koja čita jedan po jedan znak i šalje ga na konzolu.

```
//: io/CitanjeIzMemorije.java
import java.io.*;

public class CitanjeIzMemorije {
  public static void main(String[] args)
  throws IOException {
    StringReader in = new StringReader(
      BufferedInputFile.read("CitanjeIzMemorije.java"));
    int c;
    while((c = in.read()) != -1)
      System.out.print((char)c);
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Obratite pažnju na to da metoda **read( )** vraća sledeći bajt kao ceo broj (**int**) i stoga se on mora konvertovati u tip **char** da bi se pravilno ispisivao.

Formatiran ulaz iz memorije

Da biste pročitali formatirane podatke, upotrebite binarno orijentisanu U/I klasu **DataInputStream** (a ne neku znakovno orijentisanu). To znači da morate koristiti klase hijerarhije **InputStream**, a ne klase hijerarhije **Reader.** Naravno, pomoću klasa tipa **InputStream** možete da pročitate bilo šta (pa i datoteku) kao niz bajtova, ali se ovde koristi **String.**

```
//: io/FormatiranUlazIzMemorije.java
import java.io.*;

public class FormatiranUlazIzMemorije {
  public static void main(String[] args)
  throws IOException {
    try {
      DataInputStream ulaz = new DataInputStream(
        new ByteArrayInputStream(
          BaferisanaUlaznaDatoteka.read(
```

```
            "FormatiranUlazIzMemorije.java").getBytes()));
        while(true)
          System.out.print((char)ulaz.readByte());
      } catch(EOFException e) {
        System.err.println("Kraj toka");
      }
    }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Da bi se objekat klase **String** pretvorio u niz bajtova, što je potrebno za klasu **ByteArrayInputStream**, koristi se metoda **getBytes( )** klase **String**. Nakon toga imate odgovarajući ulazni tok koji ćete proslediti klasi **DataInputStream**.

Ako čitate znakove iz toka tipa **DataInputStream** bajt po bajt pomoću metode **readByte( )**, svaka vrednost može biti važeći rezultat, pa se povratna vrednost ne može koristiti za otkrivanje kraja ulaznih podataka. Umesto toga možete da upotrebite metodu **available( )** da biste saznali koliko još znakova ima. Evo primera koji pokazuje kako se čita datoteka bajt po bajt:

```
//: ui/ProveraKraja.java
// Provera kraja datoteke tokom njenog čitanja bajt po bajt.
import java.io.*;

public class ProveraKraja {
  public static void main(String[] args)
  throws IOException {
    DataInputStream ulaz =
      new DataInputStream(
       new BufferedInputStream(
        new FileInputStream("ProveraKraja.java")));
    while(ulaz.available() != 0)
      System.out.print((char)ulaz.readByte());
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Obratite pažnju na to da metoda **available( )** radi različito, u zavisnosti od toga koja se vrsta medija čita; u opštem slučaju, ona vraća broj bajtova koji se mogu pročitati *bez blokiranja*. Ako se radi o datoteci, to podrazumeva celu datoteku, ali za neku drugu vrstu toka možda ima drugačije značenje, pa ovu metodu pažljivo koristite.

Kraj ulaznih podataka u slučajevima poput ovog mogli biste da otkrijete i hvatanjem izuzetka. Međutim, korišćenje izuzetaka za kontrolu toka smatra se njihovom zloupotrebom.

Osnove pisanja u datoteku

Objekat klase **FileWriter** upisuje podatke u datoteku. Gotovo uvek će biti potrebno da se izlaz baferiše omotavanjem u **BufferedWriter** (pokušajte da izostavite omotavanje da biste videli kakav će to uticaj imati na performanse: baferisanje neverovatno poboljšava performanse U/I operacija). U ovom primeru, izlazni tok se radi formatiranja dekoriše kao **PrintWriter**. Datoteka koja je napravljena na ovaj način može se čitati kao obična tekstualna datoteka:

```
//: io/OsnovePisanjaUDatoteku.java
import java.io.*;

public class OsnovePisanjaUDatoteku {
  static String datoteka = "OsnovePisanjaUDatoteku.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader ulaz = new BufferedReader(
      new StringReader(
        BaferisanaUlaznaDatoteka.read("OsnovePisanjaUDatoteku.java")));
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(datoteka)));
    int brojRedova = 1;
    String s;
    while((s = ulaz.readLine()) != null )
      out.println(brojRedova++ + ": " + s);
    out.close();
    // Prikaži uskladištenu datoteku:
    System.out.println(BaferisanaUlaznaDatoteka.read(datoteka));
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Dok se redovi upisuju u datoteku, uvećava se ukupan broj redova. Obratite pažnju na to da *nisam* upotrebio klasu **LineNumberInputStream**, pošto je sasvim beskorisna. Kao što je ovde pokazano, veoma je lako pratiti ukupan broj redova.

Kada se ulazni tok iscrpe, metoda **readLine**( ) vraća **null**. Videćete eksplicitan poziv metode **close**( ) za datoteku **out**. Ako ne pozovete metodu **close**( ) za izlaznu datoteku, može se desiti da baferi ne budu ispražnjeni i da se zato izgube podaci.

Prečica za pisanje u tekstualnu datoteku

U Javi SE5, klasi **PrintWriter** dodat je pomoćni konstruktor, da ne biste morali ručno da radite dekoraciju svaki put kada hoćete da napravite tekstualnu datoteku i nešto u nju upišete. Izmenio sam program **OsnovePisanjaUDatoteku.java** tako da koristi ovu prečicu:

```
//: io/PrecicaZaPisanjeuDatoteku.java
import java.io.*;

public class PrecicaZaPisanjeuDatoteku {
  static String datoteka = "PrecicaZaPisanjeuDatoteku.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader ulaz = new BufferedReader(
      new StringReader(
        BaferisanaUlaznaDatoteka.read("PrecicaZaPisanjeuDatoteku.java")));
    // Evo prečice:
    PrintWriter out = new PrintWriter(datoteka);
    int brojRedova = 1;
    String s;
    while((s = ulaz.readLine()) != null )
```

```
        out.println(brojRedova++ + ": " + s);
      out.close();
      // Prikaži uskladištenu datoteku:
      System.out.println(BaferisanaUlaznaDatoteka.read(datoteka));
    }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Baferisanje je i dalje tu, samo ne morate sami da ga radite. Nažalost, za ostale uobičajene programerske zadatke nisam napisao prečice, pa tipičan U/I i dalje zahteva mnogo redundantnog teksta. Međutim, u ovoj knjizi se upotrebljava uslužna metoda **TextFile** koja će biti definisana malo kasnije u ovom poglavlju – ona pojednostavljuje te uobičajene programerske zadatke.

Vežba 12: (3) Promenite vežbu 8 tako da se otvara tekstualna datoteka u koju tekst može i da se upisuje. Upišite u datoteku redove koji se nalaze u ulančanoj listi (**LinkedList**) i njihove brojeve. (Nemojte koristiti klase LineNumber.)

Vežba 13: (3) Promenite vežbu **OsnovePisanjaUDatoteku.java** tako da za praćenje broja redova upotrebljava **LineNumberReader**. Obratite pažnju na to koliko je lakše pratiti taj broj programski.

Vežba 14: (2) Na osnovu programa **OsnovePisanjaUDatoteku.java**, napišite program koji poredi performanse upisivanja u datoteku pri korišćenju baferisanog i nebaferisanog ulaza/izlaza.

Čuvanje i rekonstruisanje podataka

Klasa **PrintWriter** formatira podatke u oblik koji čovek može da čita. Međutim, da bi se podaci iskazali u obliku koji drugi tok može da prepozna, koristi se **DataOutputStream** za upisivanje, odnosno **DataInputStream** za rekonstruisanje podataka. Naravno, ovi tokovi bi mogli da sadrže bilo koje podatke, ali se ovde koristi datoteka baferisana i za upis i za čitanje. Klase **DataOutputStream** i **DataInputStream** binarno su orijentisane, pa je potrebno koristiti klase tipa **InputStream** i **OutputStream**.

```
//: io/CuvanjeIRekonstruisanjePodataka.java
import java.io.*;

public class CuvanjeIRekonstruisanjePodataka {
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Podaci.txt")));
    out.writeDouble(3.14159);
    out.writeUTF("To je broj pi");
    out.writeDouble(1.41413);
    out.writeUTF("Kvadratni koren od 2");
    out.close();
    DataInputStream ulaz = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Podaci.txt")));
```

```
    System.out.println(ulaz.readDouble());
    // Samo readUTF() pravilno pronalazi
    // Java-UTF String:
    System.out.println(ulaz.readUTF());
    System.out.println(ulaz.readDouble());
    System.out.println(ulaz.readUTF());
  }
} /* Ispis:
3.14159
To je broj pi
1.41413
Kvadratni koren od 2
*///:~
```

Ako koristite **DataOutputStream** za upisivanje podataka, Java garantuje da ćete podatke moći tačno da rekonstruišete pomoću klase **DataInputStream**, bez obzira na to na kojoj platformi se podaci upisuju i čitaju. To je izuzetno korisno, što će biti jasno svima koji su mnogo vremena proveli brinući o prilagođavanju programa različitim platformama. Takav problem nestaje ako Java postoji na obe platforme.[3]

Kada koristite **DataOutputStream**, *jedini* način pisanja znakovnog niza (objekta tipa **String**) koji osigurava njegovu pouzdanu rekonstrukciju pomoću ulaznog toka **DataInputStream** jeste kodiranje UTF-8 koje se u ovom primeru obavlja metodama **writeUTF( )** i **readUTF( )**. UTF-8 je višebajtni format čija se dužina kodiranih znakova menja u zavisnosti od upotrebljenog skupa znakova. Ako radite (isključivo ili pretežno) sa ASCII znakovima (koji zauzimaju samo sedam bitova), korišćenjem Unicode zauzima se ogroman prostor i/ili propusni opseg; pa UTF-8 kodira ASCII znakove u jednom bajtu, a ne-ASCII znakove u dva ili tri bajta. Sem toga, dužina znakovnog niza se smešta u prva dva bajta UTF-8 znakovnog niza. Međutim, **writeUTF( )** i **readUTF( )** upotrebljavaju posebnu varijantu kodiranja UTF-8 za Javu (detaljno opisanu u HTML dokumentaciji tih metoda na Webu), pa ako znakovni niz zapisan metodom **writeUTF( )** čitate ne-Java programom, morate sami za to pisati poseban kôd, inače čitanje neće biti ispravno.

Kada koristite **DataOutputStream** i metode **writeUTF( )** i **readUTF( )**, možete mešati znakovne nizove i ostale tipove podataka, jer će znakovni nizovi biti ispravno uskladišteni kao Unicode i lako će se rekonstruisati pomoću ulaznog toka **DataInputStream**.

Metoda **writeDouble( )** stavlja broj tipa **double** u tok, a rekonstruiše ga odgovarajuća metoda **readDouble( )** – slične metode za čitanje i upisivanje postoje i za druge tipove. Da bi metode za čitanje radile kako treba, morate znati tačnu poziciju podatka u toku, pošto bi sačuvan podatak tipa **double** mogao da se pročita i kao jednostavan niz bajtova, kao tip **char** i sl. Zbog toga morate da imate fiksan format za podatke u datoteci ili se u njoj moraju čuvati dodatne informacije koje ćete analizirati da biste otkrili gde se nalaze podaci. Imajte u vidu da serijalizacija objekata ili XML (koji će biti opisani u nastavku poglavlja) mogu omogućiti lakši način skladištenja i rekonstrukcije složenih struktura podataka.

---

[3] Jezik XML takođe omogućuje prenos informacija nezavisno od platforme. Da bi se mogao koristiti, na svim platformama ne mora postojati Javina virtuelna mašina. XML će biti predstavljen u nastavku poglavlja.

Vežba 15: (4) Pročitajte HTML dokumentaciju klasa **DataOutputStream** i **DataInputStream** na Webu. Na osnovu programa **CuvanjeIPronalazenjePodataka.java**, napišite program koji skladišti i zatim rekonstruiše sve moguće tipove s kojima rade klase **DataOutputStream** i **DataInputStream**. Dokažite da se svi tipovi ispravno skladište i rekonstruišu.

Čitanje i upisivanje datoteka s nasumičnim pristupom

Korišćenje klase **RandomAccessFile** je kao kombinovanje tokova **DataInputStream** i **DataOutputStream** pošto realizuje jednake interfejse. Osim toga, za nesekvencijalno kretanje po datoteci i promenu nasumično raspoređenih vrednosti na raspolaganju je metoda **seek( )**.

Preduslov za korišćenje klase **RandomAccessFile** jeste da znate (možete da izračunate) položaj svih entiteta u datoteci, jer ćete jedino tako moći ispravno s njima da radite. **RandomAccessFile** ima specifične metode za čitanje i pisanje prostih tipova i UTF-8 znakovnih nizova. Evo jednog primera:

```
//: io/KoriscenjeRandomAccessFile.java
   import java.io.*;

public class KoriscenjeRandomAccessFile {
  static String datoteka = "rtest.dat";
  static void display() throws IOException {
    RandomAccessFile rf = new RandomAccessFile(datoteka, "r");
    for(int i = 0; i < 7; i++)
      System.out.println(
        "Vrednost " + i + ": " + rf.readDouble());
    System.out.println(rf.readUTF());
    rf.close();
  }
  public static void main(String[] args)
  throws IOException {
    RandomAccessFile rf = new RandomAccessFile(datoteka, "rw");
    for(int i = 0; i < 7; i++)
      rf.writeDouble(i*1.414);
    rf.writeUTF("Kraj datoteke");
    rf.close();
    display();
    rf = new RandomAccessFile(datoteka, "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    display();
  }
} /* Ispis:
Vrednost 0: 0.0
Vrednost 1: 1.414
Vrednost 2: 2.828
Vrednost 3: 4.242
```

```
Vrednost 4: 5.656
Vrednost 5: 7.069999999999999
Vrednost 6: 8.484
Kraj datoteke
Vrednost 0: 0.0
Vrednost 1: 1.414
Vrednost 2: 2.828
Vrednost 3: 4.242
Vrednost 4: 5.656
Vrednost 5: 47.0001
Vrednost 6: 8.484
Kraj datoteke
*///:~
```

Metoda **display( )** otvara datoteku i prikazuje njenih sedam elemenata kao vrednosti tipa **double**. U metodi **main( )**, datoteka se pravi, otvara i menja. Pošto svaki broj tipa **double** (uvek) zauzima osam bajtova, da biste metodom **seek( )** pronašli šestu vrednost po redu, morate izračunati 5*8 i dobićete poziciju njenog početnog bajta u datoteci.

Kao što je ranije pomenuto, klasa **RandomAccessFile** je potpuno izdvojena od ostatka hijerahije U/I tokova, osim što realizuje interfejse **DataInput** i **DataOutput**. Ona ne podržava dekorisanje, pa je zbog toga ne možete kombinovati ni s jednim aspektom potklasa **InputStream** i **OutputStream**. Morate pretpostaviti da je klasa **RandomAccessFile** ispravno baferisana, pošto tu funkciju ne možete da joj dodate.

Jedina opcija koja vam stoji na raspolaganju jeste drugi argument konstruktora; datoteku s nasumičnim pristupom možete da otvorite radi čitanja (**r**) ili čitanja i upisivanja (**rw**).

Imajte u vidu da umesto klase **RandomAccessFile** možete koristiti **nio** datoteke preslikane u memoriju.

Vežba 16: (2) Pročitajte HTML dokumentaciju klase **RandomAccessFile** na Webu. Na osnovu programa **KoriscenjeRandomAccessFile.java**, napišite program koji smešta i zatim rekonstruiše sve moguće tipove s kojima radi klasa **RandomAccessFile**. Dokažite da se svi tipovi ispravno skladište i rekonstruišu.

Cevovodi

Klase **PipedInputStream**, **PipedOutputStream**, **PipedReader** i **PipedWriter** pomenute su u ovom poglavlju samo ukratko. Nemojte na osnovu toga zaključiti da nisu korisne. Njihova korisnost nije očigledna dok ne počnete da isprobavate višenitni rad, jer se cevovodi upotrebljavaju za komunikaciju između niti. O ovome će, uz primer, biti reči u poglavlju *Paralelno izvršavanje*.

Uslužne klase za čitanje i pisanje

U programiranju veoma često treba učitati datoteku u memoriju, izmeniti je i zatim je ponovo iz memorije upisati u datoteku. Jedan od nedostataka Javine biblioteke za U/I jeste to što se mora pisati znatna količina koda da bi se obavile te uobičajene operacije – u njoj nema jednostavnih pomoćnih funkcija koje bi to radile umesto nas. Da stvari budu

još gore, zbog dekoratora je prilično teško zapamtiti kako se otvaraju datoteke. Dakle, bilo bi dobro dodati biblioteci pomoćne klase koje će te elementarne operacije raditi umesto nas. Java SE5 je klasi **PrintWriter** dodala prigodni konstruktor koji olakšava otvaranje tekstualne datoteke za upisivanje. Međutim, ostalo je još mnogo drugih poslova koji se često ponavljaju, pa bi valjalo napisati kôd za njih jednom zauvek.

Sledi klasa **TextFile** koju smo već u prethodnim primerima iz ove knjige koristili da pojednostavimo upisivanje i čitanje datoteka. Ona sadrži statične metode za čitanje i upisivanje tekstualnih datoteka u obliku jednog znakovnog niza, a možete napraviti i **TextFile** objekat koji redove te datoteke čuva u listi **ArrayList** (pa tokom rada sa sadržajem datoteke imate na raspolaganju svu njenu funkcionalnost):

```
//: net/mindview/util/TextFile.java
// Statične funkcije za upisivanje i čitanje tekstualnih datoteka u obliku
// jednog znakovnog niza, i tretiranje datoteke kao jedne ArrayList liste.
// x jedne ArrayList liste.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
  // Učitaj datoteku kao jedan znakovni niz:
  public static String read(String imeDatoteke) {
    StringBuilder sb = new StringBuilder();
    try {
      BufferedReader ulaz = new BufferedReader(
new FileReader(
        new File(imeDatoteke).getAbsoluteFile()));
      try {
        String s;
        while((s = ulaz.readLine()) != null) {
          sb.append(s);
          sb.append("\n");
        }
      } finally {
        ulaz.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return sb.toString();
  }
  // Upiši celu datoteku jednim pozivom metode:
  public static void write(String imeDatoteke, String tekst) {
    try {
      PrintWriter out = new PrintWriter(
        new File(imeDatoteke).getAbsoluteFile());
      try {
        out.print(tekst);
      } finally {
```

```
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Učitaj datoteku, izdeljenu na reči pomoću bilo kojeg regularnog izraza:
  public TextFile(String imeDatoteke, String podela) {
    super(Arrays.asList(read(imeDatoteke).split(podela)));
    // Zbog podele metodom split()pomoću regularnog izraza, na
    // prvoj poziciji često ostaje prazan znakovni niz:
    if(get(0).equals("")) remove(0);
  }
  // Obično se čita red po red:
  public TextFile(String imeDatoteke) {
    this(imeDatoteke, "\n");
  }
  public void write(String imeDatoteke) {
    try {
      PrintWriter out = new PrintWriter(
        new File(imeDatoteke).getAbsoluteFile());
      try {
        for(String stavka : this)
          out.println(stavka);
      } finally {
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Jednostavna provera:
  public static void main(String[] args) {
    String datoteka = read("TextFile.java");
    write("test.txt", datoteka);
    TextFile tekst = new TextFile("test.txt");
    tekst.write("test2.txt");
    // Podela na jedinstvene reči u uredenoj listi:
    TreeSet<String> recci = new TreeSet<String>(
      new TextFile("TextFile.java", "\\W+"));
    // Prikaži reči s velikim početnim slovom:
    System.out.println(recci.headSet("a"));
  }
} /* Ispis:
[0, ArrayList, Arrays, Break, BufferedReader, BufferedWriter, Clean, File,
FileReader, FileWriter, IOException, Jednostavna, Obično, Output, Prikaži,
PrintWriter, Read, Regular, RuntimeException, Static, String, StringBuilder,
System, TextFile, Tools, TreeSet, W, Write]
*///:~
```

Metoda **read( )** objektu tipa **StringBuilder** dodaje svaki red i zatim znak za prelazak u novi red, jer se oni uklanjaju tokom čitanja. Potom ona kao svoj rezultat vraća znakovni niz koji sadrži celu datoteku. Metoda **write( )** otvara tekstualni znakovni niz i upisuje ga u datoteku.

Vodite računa o tome da svaki program koji otvara neku datoteku čuva njen poziv metode **close( )** unutar bloka **finally**, da bi datoteka bila zajemčeno zatvorena.

Konstruktor pretvara datoteku u znakovni niz pomoću metode **read( )**, zatim poziva **String.split( )** da bi taj rezultat podelio na redove (uz znakove za prelazak u novi red kao graničnike; ako ovu klasu često upotrebljavate, mogli biste da prepravite taj konstruktor tako da bude delotvorniji). Nažalost, ne postoji odgovarajuća metoda „spajanja" redova u znakovni niz, pa nestatičnom metodom **write( )** moramo da ispisujemo red po red ručno.

Pošto je ova klasa napisana zato da bi do krajnosti pojednostavila postupak čitanja i pisanja datoteke, svi **IOException** izuzeci pretvoreni su u **RuntimeException** izuzetke, kako korisnik ne bi morao da upotrebljava **try-catch** blokove. Međutim, možda ćete morati da napišete drugu verziju koja **IOException** izuzetke prosleđuje pozivaocu.

U metodi **main( )** obavlja se elementarna provera ispravnosti rada spomenutih metoda.

Nije trebalo mnogo koda da bi se napisala ova uslužna klasa, a ona će nam uštedeti mnogo vremena i olakšati život, u šta ćete se uveriti u nekima od narednih primera u ovom poglavlju.

Problem s čitanjem tekstualnih datoteka može se rešiti i na drugi način, klasom **java.util.Scanner** uvedenom u Javu SE5. Međutim, ona služi samo za čitanje datoteka, ne i za pisanje u njih. Ta alatka (koja *nije* stavljena u paket **java.io**) prvenstveno je namenjena za pravljenje analizatora programskih jezika (engl. *scanners*) ili „malih jezika".

Vežba 17: (4) Pomoću klase **TextFile** i mape **Map<Character,Integer>** napišite program koji prebrojava znakove u datoteci. (Dakle, ako u nekoj datoteci ima 12 primeraka slova a, 12 treba da sadrži objekat tipa **Integer** pridružen objektu tipa **Character** koji sadrži a u mapi).

Vežba 18: (1) Izmenite program **TextFile.java** tako da izuzetke **IOExceptions** prosleđuje pozivaocu.

Čitanje binarnih datoteka

Slično klasi **TextFile.java**, i ova klasa pojednostavljuje postupak čitanja binarne datoteke:

```
//: net/mindview/util/BinarnaDatoteka.java
// Uslužna klasa za čitanje datoteke u binarnom obliku.
package net.mindview.util;
import java.io.*;

public class BinarnaDatoteka {
  public static byte[ ] read(File bDatoteka) throws IOException{
    BufferedInputStream bD = new BufferedInputStream(
      new FileInputStream(bDatoteka));
    try {
```

```
      byte[ ] data = new byte[bD.available()];
      bD.read(data);
      return data;
    } finally {
        bD.close();
    }
  }
  public static byte[ ]
  read(String bDatoteka) throws IOException {
    return read(new File(bDatoteka).getAbsoluteFile());
  }
} ///:~
```

Jedna preklopljena metoda prima argument tipa **File**; druga prima argument tipa **String**, što je ime te datoteke. Obe vraćaju rezultujući niz tipa **byte**.

Metodom **available( )** pravi se niz odgovarajuće veličine, a baš ova verzija preklopljene metode **read( )** popunjava taj niz.

Vežba 19: (2) Pomoću klase **BinarnaDatoteka** i mape **Map<Byte,Integer>** napišite program koji prebrojava različite bajtove u datoteci.

Vežba 20: (4) Pomoću metode **Direktorijum.prolazak( )** i klase **BinarnaDatoteka**, dokažite da sve **.class** datoteke u stablu direktorijuma počinju heksadecimalnim kodovima znakova **CAFEBABE**.

Standardni U/I tokovi

Pojam *standardni U/I tok* odnosi se na Unixov koncept koji je u ovom ili onom obliku reprodukovan u Windowsu i mnogim drugim operativnim sistemima, a označava jedinstven tok informacija koji koristi program. Svi ulazni podaci programa stižu iz *standardnog ulaznog toka*, svi izlazni podaci šalju se na *standardni izlazni tok*, a sve poruke o greškama šalju se na *standardni tok za greške*. Standardni ulazno/izlazni tokovi su značajni zato što omogućuju lako povezivanje programa, a standardni izlazni tok jednog programa može da postane standardni ulazni tok drugog programa. To je snažna alatka.

Čitanje standardnog ulaznog toka

Oslanjajući se na standardan U/I model, Java obezbeđuje tokove **System.in**, **System.out** i **System.err**. U celoj knjizi smo upisivali u standardan izlazni tok pomoću objekta **System.out** koji je već omotan kao objekat klase **PrintStream**. **System.err** na sličan način biva objekat klase **PrintStream**, a **System.in** je tipa **InputStream**, bez omotavanja. To znači sledeće: za razliku od objekata **System.out** i **System.err** koje možete da koristite odmah, objekat **System.in** mora da se omota da biste nešto čitali iz njega.

Ulaz, po pravilu, čitate red po red, koristeći metodu **readLine( )**, pa **System.in** treba da omotate u **BufferedReader**. Da biste to uradili, morate da konvertujete **System.in** u **Reader** pomoću klase **InputStreamReader**. Evo primera koji samo ponavlja svaki red koji unesete:

```
//: ui/Eho.java
// Kako se čita standardni ulazni tok.
// {RunByHand}
import java.io.*;

public class Eho {
  public static void main(String[ ] args)
  throws IOException {
    BufferedReader stdin = new BufferedReader(
          new InputStreamReader(System.in));
    String s;
    while((s = stdin.readLine()) != null  && s.length() != 0)
      System.out.println(s);
    // Program završava prazan red ili Ctrl – Z
  }
} ///:~
```

Razlog za specifikaciju izuzetka u metodi **main( )** jeste to što metoda **readLine( )** može da generiše izuzetak **IOException**. Vodite računa o tome da **System.in** obično treba da bude baferisan, kao i većina tokova.

Vežba 21: (1) Napišite program koji prima standardan ulaz, pretvara sva primljena slova u velika, a zatim rezultate šalje na standardan izlaz. Preusmerite u ovaj program sadržaj neke datoteke. (Postupak preusmeravanja se menja u zavisnosti od operativnog sistema.)

Omotavanje toka System.out u PrintWriter

Tok **System.out** je tipa **PrintStream** koji nasleđuje klasu **OutputStream**. Klasa **PrintWriter** ima konstruktor čiji je argument objekat klase **OutputStream**. Stoga tim konstruktorom možete da pretvorite **System.out** u **PrintWriter**:

```
//: ui/OmotavanjeSystemOut.java
// Omotavanje System.out u PrintWriter.
import java.io.*;

public class OmotavanjeSystemOut {
  public static void main(String[] args) {
    PrintWriter out = new PrintWriter(System.out, true);
    out.println("Zdravo, svete");
  }
} /* Ispis:
Zdravo, svete
*///:~
```

Važno je da se koristi verzija konstruktora za klasu **PrintWriter** koja ima dva argumenta i da drugi argument bude **true**, da bi se omogućilo automatsko pražnjenje izlaznih bafera, inače se izlaz neće prikazivati red po red.

Preusmeravanje standardnog ulaza/izlaza

Javina klasa **System** omogućuje preusmeravanje standardnih ulaznih tokova, izlaznih tokova i tokova za greške, pozivanjem jednostavnih statičkih metoda:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Preusmeravanje izlaza je naročito korisno ako odjednom počnete da ispisujete veliku količinu izlaznih podataka na konzolu, pa se oni kreću brže nego što možete da ih pročitate.[4] Preusmeravanje ulaza je korisno za konzolne programe pomoću kojih želite više puta da testirate određenu sekvencu ulaznih podataka. Evo jednostavnog primera koji prikazuje korišćenje ovih metoda:

```
//: ui/Preusmeravanje.java
// Prikazuje standardno U/I preusmeravanje.
import java.io.*;

class Preusmeravanje {
  public static void main(String[] args)
  throws IOException {
    PrintStream konzola = System.out;
    BufferedInputStream in = new BufferedInputStream(
        new FileInputStream("Preusmeravanje.java"));
    PrintStream out = new PrintStream(
        new BufferedOutputStream(
          new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);
    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    out.close(); // Ne zaboravite ovo!
    System.setOut(konzola);
  }
} ///:~
```

Ovaj program povezuje datoteku na standardni ulaz i preusmerava standardni izlaz i standardni tok za greške u drugu datoteku. Obratite pažnju na to da se na početku programa skladišti referenca prvobitnog **System.out** objekta i da se na kraju standardni izlaz vraća tom objektu.

U/I preusmeravanje radi s tokovima bajtova, a ne s tokovima znakova, pa se koriste klase **InputStream** i **OutputStream** umesto klasa **Reader** i **Writer**.

---

[4] U poglavlju *Grafička korisnička okruženja* predstavljeno je bolje rešenje tog problema – grafički program s poljem za tekst.

Upravljanje procesima

Često ćete biti u prilici da iz Jave pokrećete druge programe operativnog sistema i da upravljate ulazom i izlazom tih programa. Javina biblioteka ima klase za takve operacije.

Uobičajen zadatak je pokretanje programa i slanje rezultujućeg izlaza na konzolu. U ovom odeljku napisaćemo uslužnu klasu koja će pojednostaviti taj zadatak.

U ovoj uslužnoj klasi mogu nastati dve vrste grešaka: uobičajene greške koje prouzrokuju izuzetke – za njih ćemo samo ponovo generisati **RuntimeException** – i greške pri izvršavanju samog procesa. Te greške ćemo prijavljivati posebnim izuzetkom:

```
//: net/mindview/util/IzuzetakOSIzvrsenja.java
package net.mindview.util;

public class IzuzetakOSIzvrsenja extends RuntimeException {
  public IzuzetakOSIzvrsenja(String razlog) { super(razlog); }
} ///:~
```

Da biste pokrenuli program, metodi **OSIzvrsenje.komanda( )** prosledite komandni znakovni niz to jest komandu koju biste upisali da pokrećete program s konzole. Ta komanda se prosleđuje konstruktoru klase **java.lang.ProcessBuilder** (kojoj je potrebno da ga dobije u obliku sekvence **String** objekata), i tako nastaje rezultujući objekat tipa **ProcessBuilder**:

```
//: net/mindview/util/OSIzvrsenje.java
// Izvršava komandu operativnog sistema i izlaz šalje na konzolu.
package net.mindview.util;
import java.io.*;

public class OSIzvrsenje {
  public static void komanda(String komanda) {
    boolean greska = false;
    try {
      Process proces =
        new ProcessBuilder(komanda.split(" ")).pocetak();
      BufferedReader rezultati = new BufferedReader(
        new InputStreamReader(proces.getInputStream()));
      String s;
      while((s = rezultati.readLine())!= null)
        System.out.println(s);
      BufferedReader greske = new BufferedReader(
        new InputStreamReader(proces.getErrorStream()));
      // Ako ima problema, prijavi greške i pozivajućem
      // procesu vrati vrednost različitu od nule:
      while((s = greske.readLine())!= null) {
        System.greska.println(s);
        greska = true;
      }
    } catch(Exception e) {
      // Zbog Windowsa 2000, koji generiše
      // izuzetak za podrazumevani sadržaj komandne linije:
```

```
        if(!komanda.startsWith("CMD /C"))
          komanda("CMD /C " + komanda);
        else
          throw new RuntimeException(e);
      }
      if(greska)
        throw new IzuzetakOSIzvrsenja("Greške tokom izvršavanja " +
          komanda);
    }
} ///:~
```

Metodu **getInputStream( )** pozivate da bi hvatala standardni izlaz programa tokom izvršavanja. To se čini zato što objekat tipa **InputStream** možemo čitati.

Rezultati programa stižu red po red, pa ih čitamo metodom **readLine( )**. Kod nas se redovi samo ispisuju, ali biste mogli da ih hvatate i vraćate iz metode **komanda( )**.

Greške u programu šalju se na standardni tok za greške i hvataju pozivanjem metode **getErrorStream( )**. Ako ima grešaka, one se ispisuju i generiše se izuzetak **IzuzetakOSIzvrsenja** da bi pozivajući program rešio problem.

Evo primera kako se koristi **OSIzvrsenje**:

```
//: ui/PrimerOSIzvrsenja.java
// Prikazuje standardno preusmeravanje U/I.
import net.mindview.util.*;

public class PrimerOSIzvrsenja {
  public static void main(String[] args) {
    OSIzvrsenje.komanda("javap PrimerOSIzvrsenja");
  }
} /* Ispis:
Compiled from "PrimerOSIzvrsenja.java"
public class PrimerOSIzvrsenja extends java.lang.Object{
    public PrimerOSIzvrsenja();
    public static void main(java.lang.String[]);
}
*///:~
```

Ovde je za povratno prevođenje programa upotrebljen dekompilator **javap** (koji se isporučuje uz JDK).

Vežba 22: (5) Izmenite program **OSIzvrsenje.java** tako da rezultate izvršavanja programa vraća u obliku liste znakovnih nizova, umesto da ih ispisuje u standardni izlazni tok. Pokažite kako se koristi nova verzija te uslužne klase.

Nove U/I klase

Javina „nova" U/I biblioteka, uvedena u JDK 1.4 u paketima **java.nio.***, ima samo jedan cilj: brzinu. U stvari, „stari" U/I paketi su ponovo realizovani pomoću **nio** klasa da bi se iskoristilo to ubrzanje, pa od njega imate koristi čak i kada u svojim programima ne koristite eksplicitno **nio** klase. Ubrzanje se oseća i pri U/I operacijama sa datotekama, što je tema ovog poglavlja, i pri mrežnom ulazu/izlazu, obrađenom u knjizi *Thinking in Enterprise Java*.

Ubrzanje je posledica korišćenja *kanala* i *bafera* – struktura koje su bliže načinu na koji sâm operativni sistem obavlja U/I operacije. Po analogiji s rudnikom uglja, kanal bi bio rudnik koji sadrži žilu uglja (podatke), a bafer su kola koja se šalju u rudnik. Kola se vraćaju puna uglja, i on se vadi iz njih, a ne neposredno iz rudnika. Dakle, s kanalom nemate posla neposredno; radite s baferom i njega šaljete u kanal. Kanal uzima podatke iz bafera ili ih stavlja u bafer.

Jedina vrsta bafera za komunikaciju neposredno s kanalom jeste **ByteBuffer** – tj. bafer koji čuva sirove bajtove. Ako pročitate HTML dokumentaciju klase **java.nio.ByteBuffer**, videćete da je prilično elementarna: njenom konstruktoru treba saopštiti koliko memorijskog prostora da rezerviše, i postoje metode za umetanje i vađenje podataka, bilo u sirovom obliku bajtova bilo u obliku prostih tipova. Ali nema načina da umetnete ili izvadite objekat, pa čak ni znakovni niz. Sve je zadržano na prilično niskom nivou, baš zato što se u većini operativnih sistema time dobija delotvornije preslikavanje.

Tri klase „starog" U/I izmenjene su tako da proizvode **FileChannel**: **FileInputStream**, **FileOutputStream**, i, kako za čitanje tako i za pisanje, **RandomAccessFile**. Obratite pažnju na to da se radi o tokovima za rukovanje bajtovima, što je u skladu s niskim nivoom **nio** klasa. Znakovno orijentisane klase **Reader** i **Writer** ne proizvode kanale, ali klasa **java.nio.channels.Channels** ima uslužne metode koje od kanala proizvode objekte tipova **Reader** i **Writer**.

Evo jednostavnog primera u kojem sve tri vrste tokova proizvode kanale u koje se može pisati, čitati/pisati odnosno čitati:

```
//: ui/DajKanal.java
// Pravljenje kanala od tokova
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class DajKanal {
  private static final int VELBAF = 1024;
  public static void main(String[] args) throws Exception {
    // Ispiši datotekU:
    FileChannel fc =
      new FileOutputStream("podaci.txt").getChannel();
    fc.write(ByteBuffer.wrap("Nekakav tekst ".getBytes()));
    fc.close();
    // Dodaj na kraj datoteke:
    fc =
      new RandomAccessFile("podaci.txt", "rw").getChannel();
    fc.position(fc.size()); // Pređi na kraj
    fc.write(ByteBuffer.wrap("Još malo".getBytes()));
    fc.close();
    // Čitaj datoteku:
    fc = new FileInputStream("podaci.txt").getChannel();
    ByteBuffer baf = ByteBuffer.allocate(VELBAF);
    fc.read(baf);
    baf.flip();
    while(baf.hasRemaining())
```

```
      System.out.print((char)baf.get());
    }
  } /* Ispis:
  Nekakav tekst Još malo
  *///:~
```

Metoda **getChannel( )** za svaku prikazanu klasu tokova pravi objekat tipa **FileChannel**. Kanal je prilično jednostavan: može mu se predati objekat tipa **ByteBuffer** radi čitanja ili pisanja, i delove datoteke možete zaključati, odnosno ostvariti isključivo pravo pristupanja (to će biti opisano u nastavku).

Jedan od načina smeštanja bajtova u **ByteBuffer** jeste da ih neposredno umetnete u njega nekom od put metoda, i tako upišete jedan ili više bajtova ili vrednosti prostih tipova. Međutim, iz programa vidite da metodom **wrap( )** možete postojeći niz tipa **byte** da omotate objektom tipa **ByteBuffer**. Kada to uradite, pripadajući niz se ne kopira, nego koristi kao skladište za generisani **ByteBuffer**. Kažemo da je taj **ByteBuffer** poduprt nizom.

Datoteku **podaci.txt** ponovo otvara klasa **RandomAccessFile**. Vodite računa o tome da **FileChannel** možete premeštati po datoteci; ovde sam ga premestio na kraj, tako da se novi upisi dodaju na kraj postojećih (engl. *append*).

Kada primenjujete pristup samo za čitanje, memorijski prostor za **ByteBuffer** morate eksplicitno dodeliti statičnom metodom **allocate( )**. Svrha **nio** klasa je brzo premeštanje velikih količina podataka, pa nije nevažno koju veličinu bafera **ByteBuffer** zadate – zapravo, ovde će vam 1 K verovatno mnogo manje značiti nego inače (moraćete da eksperimentišete s tekućom aplikacijom da biste odredili najbolju veličinu bafera).

U potrazi za još većom brzinom, mogli biste metodom **allocateDirect( )** umesto metodom **allocate( )** napraviti direktan bafer koji je još tešnje spregnut sa operativnim sistemom. Međutim, režijski troškovi takve dodele su veći, a njena realizacija se menja u zavisnosti od operativnog sistema, pa ćete opet morati da eksperimentišete s tekućom aplikacijom da biste odredili donose li vam direktni baferi ikakvo ubrzanje.

Nakon što pozovete **read( )** da biste kanalu **FileChannel** saopštili da upiše bajtove u **ByteBuffer**, morate pozvati metodu **flip( )** za taj bafer kako biste mu saopštili da se pripremi za vađenje bajtova iz njega. (To jeste malo nezgrapno, ali ne zaboravite da se radi o veoma niskom nivou i da se tako postiže maksimalna brzina). A kada bismo taj bafer upotrebili za još neku operaciju **read( )**, pre svakog učitavanja sadržaja u bafer morali bismo da pozovemo metodu **clear( )** kao pripremu. To vidite u ovom jednostavnom programu za kopiranje datoteka:

```
//: ui/KopiranjeKanalom.java
// Kopiranje datoteke pomoću kanala i bafera
// {Args: KopiranjeKanalom.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class KopiranjeKanalom {
  private static final int VELBAF = 1024;
  public static void main(String[] args) throws Exception {
```

```
      if(args.length != 2) {
        System.out.println("argumenti: datizvor datodrediste");
        System.exit(1);
      }
      FileChannel
        ulaz = new FileInputStream(args[0]).getChannel(),
        izlaz = new FileOutputStream(args[1]).getChannel();
      ByteBuffer bafer = ByteBuffer.allocate(VELBAF);
      while(ulaz.read(bafer) != -1) {
        bafer.flip(); // Pripremi za pisanje
        izlaz.write(bafer);
        bafer.clear();  // Pripremi za čitanje
      }
    }
} ///:~
```

Jedan **FileChannel** otvorili smo za čitanje, a drugi za pisanje. Objektu tipa **ByteBuffer** dodeljuje se memorija, i kada metoda **FileChannel.read( )** vrati -1 (nema sumnje, ta vrednost je zaostatak iz Unixa i C-a), to znači da smo došli do kraja ulaza. Nakon svakog poziva metode **read( )** koja u bafer smešta podatke, **flip( )** priprema bafer da bi metoda **write( )** mogla da vadi podatke iz njega. Podaci ostaju u baferu i nakon čitanja metodom **write( )**, i tek metoda **clear( )** resetuje sve interne pokazivače tako da bafer postaje spreman da od metode **read( )** primi nove podatke.

Prethodni program nije idealan za obavljanje te vrste operacija. Postoje posebne metode **transferTo( )** i **transferFrom( )** za neposredno povezivanje jednog kanala s drugim:

```
//: ui/TransferTo.java
// Povezivanje kanala metodom transferTo()
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("argumenti: datizvor datodrediste");
      System.exit(1);
    }
    FileChannel
      ulaz = new FileInputStream(args).getChannel(),
      izlaz = new FileOutputStream(args).getChannel();
    ulaz.transferTo(0, ulaz.size(),izlaz);
    // Ili:
    // izlaz.transferFrom(ulaz, 0, ulaz.size());
  }
} ///:~
```

Ovako nešto retko će vam trebati, ali dobro je to znati.

Konverzija podataka

Ako se osvrnete na program **DajKanal.java**, videćete da prilikom ispisivanja datoteke podatke vadimo bajt po bajt, pa svaki **byte** zasebno pretvaramo u **char**. To je primitivno – pročitajte dokumentaciju klase **java.nio.CharBuffer** i videćete da se u njoj za metodu **toString( )** kaže: „Vraća znakovni niz koji sadrži znakove bafera". Pošto se **ByteBuffer** nakon obrade metodom **asCharBuffer( )** može smatrati baferom **CharBuffer**, zašto ne bismo upotrebili njega? Kao što vidite iz prvog reda ispisa rezultata sledećeg programa, to ipak ne radi kako smo očekivali:

```
//: ui/BaferUTekst.java
// Konverzija teksta u ByteBuffer i iz njega
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BaferUTekst {
  private static final int VELBAF = 1024;
  public static void main(String[] args) throws Exception {
    FileChannel fc =
      new FileOutputStream("podaci2.txt").getChannel();
    fc.write(ByteBuffer.wrap("Neki tekst".getBytes()));
    fc.close();
    fc = new FileInputStream("podaci2.txt").getChannel();
    ByteBuffer baf = ByteBuffer.allocate(VELBAF);
    fc.read(baf);
    baf.flip();
    // Ne radi:
    System.out.println(baf.asCharBuffer());
    // Dekodiraj po podrazumevanoj šemi kodiranja ovog sistema:
    baf.rewind();
    String kodiranje = System.getProperty("file.encoding");
    System.out.println("Dekodirano po šemi " + kodiranje + ": "
      + Charset.forName(kodiranje).decode(baf));
    // Ili da kodiramo po šemi koja se može odštampati:
    fc = new FileOutputStream("podaci2.txt").getChannel();
    fc.write(ByteBuffer.wrap("Neki tekst".getBytes("UTF-16BE")));
    fc.close();
    // Sada ponovo pokušajte da čitate:
    fc = new FileInputStream("podaci2.txt").getChannel();
    baf.clear();
    fc.read(baf);
    baf.flip();
    System.out.println(baf.asCharBuffer());
    // Pisaćemo kroz CharBuffer:
    fc = new FileOutputStream("podaci2.txt").getChannel();
    baf = ByteBuffer.allocate(24); // Više nego što je potrebno
    baf.asCharBuffer().put("Neki tekst");
    fc.write(baf);
```

```
      fc.close();
      // čitanje i prikazivanje:
      fc = new FileInputStream("podaci2.txt").getChannel();
      baf.clear();
      fc.read(baf);
      baf.flip();
      System.out.println(baf.asCharBuffer());
    }
} /* Ispis:
????
Dekodirano po šemi Cp1252: Neki tekst
Neki tekst
Neki tekst
*///:-
```

Bafer sadrži sirove bajtove, i da bismo njih pretvorili u znakove, moramo ih *kodirati* prilikom stavljanja unutra (da bi nešto značili kada ih izvadimo iz bafera) ili *dekodirati* prilikom vađenja iz bafera. To se može postići klasom **java.nio.charset.Charset** koja obuhvata alatke za kodiranje po raznim šemama:

```
//: ui/DostupneSemeKodiranja.java
// Prikazuje šeme kodiranja i njihove alijase
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class DostupneSemeKodiranja {
  public static void main(String[] args){
    SortedMap<String,Charset> semeKodiranja =
      Charset.availableCharsets();
    Iterator<String> it = semeKodiranja.keySet().iterator();
    while(it.hasNext()) {
      String imeSemeKodiranja = it.next();
      printnb(imeSemeKodiranja);
      Iterator alijasi =
        semeKodiranja.get(imeSemeKodiranja).aliases().iterator();
      if(alijasi.hasNext())
        printnb(": ");
      while(alijasi.hasNext()) {
        printnb(alijasi.next());
        if(alijasi.hasNext())
          printnb(", ");
      }
      print();
    }
  }
} /* Ispis:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0, big5hkscs, Big5_HKSCS
```

```
EUC-JP: eućis, x-eućp, csEUCPkdFmtjapanese, eućp,
Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp, euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-1987, euc_kr,
ks_c_5601-1987, euckr, csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn, euccn, x-EUC-CN
GBK: windows-936, CP936
...
*///:~
```

Vratimo se na **BaferUTekst.java**. Ako bafer „premotate" metodom **rewind( )** (da biste se vratili na početak podataka) i zatim za dekodiranje podataka metodom **decode( )** upotrebite podrazumevanu šemu kodiranja te platforme, dobićete **CharBuffer** koji se lepo ispisuje na konzoli. Za pronalaženje podrazumevane šeme kodiranja pozovite metodu **System.getProperty("file.encoding")** koja proizvodi znakovni niz s imenom te šeme kodiranja. Prosledite to metodi **Charset.forName( )** i dobićete objekat tipa **Charset** kojim se taj znakovni niz može dekodirati.

Druga mogućnost je da (metodom **encode( )**) kodirate po šemi kodiranja koja će dati rezultat koji se može štampati nakon čitanja datoteke, kao što vidite u trećem delu programa **BaferUTekst.java**. Tu je za pisanje teksta u datoteku upotrebljena šema kodiranja UTF-16BE, pa nakon čitanja samo morate da je konvertujete u **CharBuffer** koji će dati očekivani tekst.

Najzad, vidite šta se dešava ako pišete u **ByteBuffer** preko **CharBuffer** bafera (o tome će još biti reči). Baferu **ByteBuffer** dodelili smo 24 bajta. Pošto svaki znak (**char**) zahteva dva bajta, to je dovoljno za 12 znakova, ali „Neki tekst" ih ima samo 10. Preostali bajtovi s nulama ipak se vide u prikazu **CharBuffer** bafera koji proizvodi njegova metoda **toString( )**, kao što vidite iz ispisa rezultata.

Vežba 23: (6) Napravite i ispitajte uslužnu metodu koja štampa sadržaj **CharBuffer** bafera sve do mesta od kojega se njegovi znaci više ne mogu odštampati.

Pribavljanje prostih tipova

Iako **ByteBuffer** samo čuva bajtove, on obuhvata metode koje od tih bajtova proizvode sve proste tipove. U narednom primeru prikazano je umetanje i vađenje različitih vrednosti pomoću tih metoda:

```
//: ui/DajPodatke.java
// Tumačenje bajtova iz ByteBuffera na različite načine
import java.nio.*;
import static net.mindview.util.Print.*;

public class DajPodatke {
  private static final int VELBAF = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(VELBAF);
    // ByteBuffer se automatski puni nulama čim mu se dodeli memorija:
    int i = 0;
    while(i++ < bb.limit())
```

```
        if(bb.get() != 0)
          print("nije nula");
      print("i = " + i);
      bb.rewind();
      // Upiši i pročitaj niz tipa char:
      bb.asCharBuffer().put("Zdravo!");
      char c;
      while((c = bb.getChar()) != 0)
        printnb(c + " ");
      print();
      bb.rewind();
      // Upiši i pročitaj broj tipa short:
      bb.asShortBuffer().put((short)471142);
      print(bb.getShort());
      bb.rewind();
      // Upiši i pročitaj broj tipa int:
      bb.asIntBuffer().put(99471142);
      print(bb.getInt());
      bb.rewind();
      // Upiši i pročitaj broj tipa long:
      bb.asLongBuffer().put(99471142);
      print(bb.getLong());
      bb.rewind();
      // Upiši i pročitaj broj tipa float:
      bb.asFloatBuffer().put(99471142);
      print(bb.getFloat());
      bb.rewind();
      // Upiši i pročitaj broj tipa double:
      bb.asDoubleBuffer().put(99471142);
      print(bb.getDouble());
      bb.rewind();
    }
} /* Ispis:
i = 1025
Z d r a v o !
12390
99471142
99471142
9.9471144E7
9.9471142E7
*///:~
```

Nakon dođele memorije objektu tipa **ByteBuffer**, proverili smo da li su njegove vrednosti automatski postale nula – i jesu. Pročitane su sve 1024 vrednosti (do kraja bafera koji daje metoda **limit**( )), i sve su bile nula.

Vrednosti prostih tipova najlakše je umetnuti u **ByteBuffer** pomoću odgovarajućih „prikaza" tog bafera koje daju metode **asCharBuffer**( ), **asShortBuffer**( ) itd., i zatim treba upotrebiti metodu **put**( ) tog prikaza. Upravo taj postupak smo primenili za sve proste tipove podataka. Među svim tim metodama, jedino je čudna metoda **put**( ) za **ShortBuffer** koja zahteva eksplicitnu konverziju tipa (a ona odseca i menja rezultujuću vrednost). Nijedan od ostalih prikaza bafera u svojoj metodi **put**( ) ne zahteva konverziju tipa.

Baferi prikaza

„Bafer prikaza" omogućuje da **ByteBuffer** u pozadini posmatrate kroz prozor određenog prostog tipa. **ByteBuffer** je i dalje jedino skladište koje „podupire" taj prikaz, pa se sve izmene koje napravite u prikazu odražavaju u izmenama podataka u **ByteBuffer** baferu. Kao što ste videli u prethodnom primeru, na raspolaganju su vam pomoćne metode za umetanje prostih tipova u **ByteBuffer**. Prikaz omogućuje i čitanje vrednosti prostih tipova iz **ByteBuffer** bafera, bilo jednu po jednu (što **ByteBuffer** dozvoljava) bilo u grupama (kao nizove). Evo primera kako se pomoću klase **IntBuffer** radi s celim brojevima (**int**) u **ByteBuffer** baferu:

```
//: ui/IntBufferPrimer.java
// Rad s celim brojevima u ByteBufferu pomoću IntBuffera
   import java.nio.*;

public class IntBufferPrimer {
  private static final int VELBAF = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(VELBAF);
    IntBuffer ib = bb.asIntBuffer();
    // Uskladišti niz celih brojeva:
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    // Čitanje i pisanje sa/u apsolutne lokacije:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // Zadavanje novog kraja bafera pre "premotavanja".
    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
      System.out.println(i);
    }
  }
} /* Ispis:
99
11
42
47
1811
143
811
1016
*///:~
```

Preklopljena metoda **put( )** prvo se koristi za skladištenje niza celih brojeva. Naredni pozivi metoda **get( )** i **put( )** neposredno pristupaju lokacijama celih brojeva u pripadajućem **ByteBuffer**u. Imajte u vidu da je pristupanje apsolutnim lokacijama dostupno i za proste tipove, preko neposrednog obraćanja **ByteBuffer**u.

Kada se pripadajući **ByteBuffer** preko bafera prikaza popuni celim brojevima ili nekim drugim prostim tipom vrednosti, onda se **ByteBuffer** može upisati neposredno u kanal. Jednako lako je i čitanje iz kanala i upotreba bafera prikaza za konverziju svega u

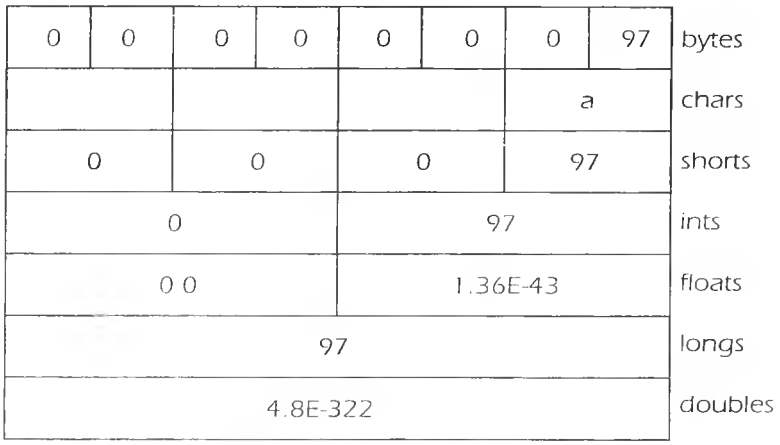
određeni tip proste vrednosti. U narednom primeru se ista sekvenca bajtova redom tumači kao broj tipa **short**, **int**, **float**, **long** i **double**, tako što se proizvode odgovarajući baferi prikaza za isti **ByteBuffer**:

```
//: ui/BaferiPrikaza.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class BaferiPrikaza {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    printnb("ByteBuffer ");
    while(bb.hasRemaining())
      printnb(bb.position()+ " -> " + bb.get() + ", ");
    print();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    printnb("CharBuffer ");
    while(cb.hasRemaining())
      printnb(cb.position() + " -> " + cb.get() + ", ");
    print();
    FloatBuffer fb =
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    printnb("FloatBuffer ");
    while(fb.hasRemaining())
      printnb(fb.position()+ " -> " + fb.get() + ", ");
    print();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    printnb("IntBuffer ");
    while(ib.hasRemaining())
      printnb(ib.position()+ " -> " + ib.get() + ", ");
    print();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    printnb("LongBuffer ");
    while(lb.hasRemaining())
      printnb(lb.position()+ " -> " + lb.get() + ", ");
    print();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    printnb("ShortBuffer ");
    while(sb.hasRemaining())
      printnb(sb.position()+ " -> " + sb.get() + ", ");
    print();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    printnb("DoubleBuffer ");
```

```
    while(db.hasRemaining())
      printnb(db.position()+ " -> " + db.get() + ", ");
  }
} /* Ispis:
ByteBuffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0, 6 -> 0, 7 -> 97,
CharBuffer 0 ->  , 1 ->  , 2 ->  , 3 -> a,
FloatBuffer 0 -> 0.0, 1 -> 1.36E-43,
IntBuffer 0 -> 0, 1 -> 97,
LongBuffer 0 -> 97,
ShortBuffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
DoubleBuffer 0 -> 4.8E-322,
*///:-
```

ByteBuffer je proizveden omotavanjem osmobajtnog niza tipa **byte** koji se zatim prikazuje putem bafera prikaza svih prostih tipova. U narednom dijagramu prikazana su različita tumačenja istih bitova kada se čitaju iz različitih vrsta bafera:

Ovo odgovara ispisu rezultata programa.

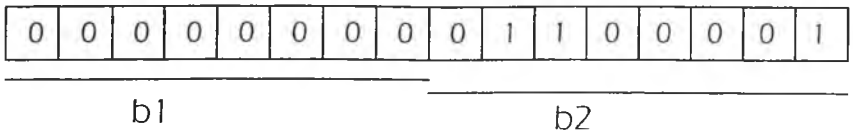


Vežba 24: (1) Izmenite program **IntBufferPrimer.java** tako da se koriste brojevi tipa double.

Endiani

Podaci se na raznim računarima skladište na različite načine. Na „big endian" platformama, najznačajniji bajt se smešta na najnižu memorijsku adresu, dok se na „little endian" platformama najznačajniji bajt smešta na najvišu memorijsku adresu. Kada skladištite vrednost veću od jednog bajta, kao što su **int**, **float** itd., morate uzeti u obzir redosled bajtova. **ByteBuffer** skladišti podatke u „big endian" obliku, a tako se podaci uvek šalju i preko mreže. Redosled smeštanja bajtova podataka u **ByteBuffer**u možete promeniti metodom **order**( ), uz argument **ByteOrder.BIG_ENDIAN** ili **ByteOrder.LITTLE_ENDIAN**.

Pogledajmo **ByteBuffer** koji sadrži ova dva bajta:



Ako ove bajtove pročitate (protumačite) kao broj tipa **short** (**ByteBuffer.asShortBuffer**( )), dobićete broj 97 (00000000 01100001), ali ukoliko pređete na redosled bajtova little endian, dobićete broj 24832 (01100001 00000000).

U narednom primeru pokazano je kako se redosled bajtova u znakovima menja u zavisnosti od parametra endian:

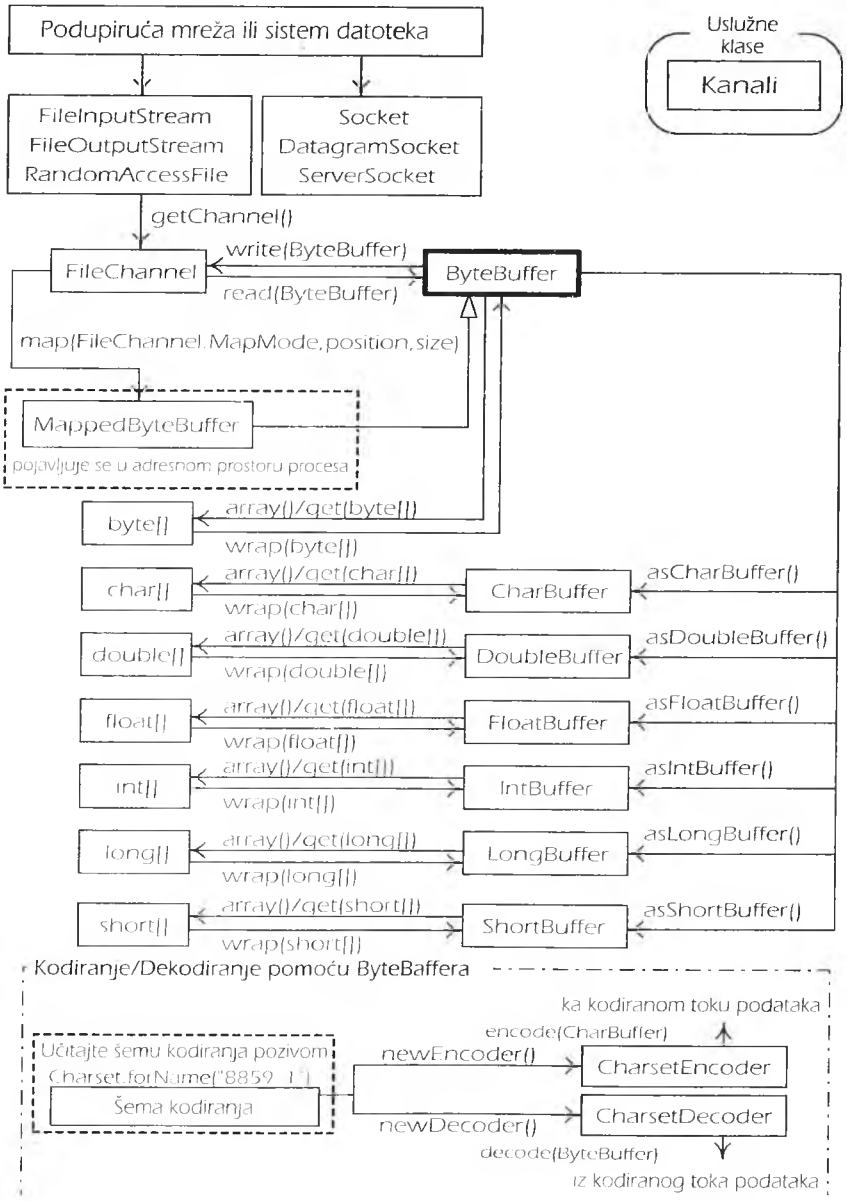
```
//: ui/Endiani.java
// Razlike između endiana i skladištenje podataka.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endiani {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte [12]);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
  }
} /* Ispis:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:~
```

ByteBuffer je dobio dovoljno prostora da sve bajtove čuva u nizu znakova (**charArray**) kao spoljnom baferu, tako da se može pozvati metoda **array**( ) da bi prikazala sve bajtove. Metoda **array**( ) je neobavezna i možete je pozvati za bafere poduprte nizom; u protivnom, dobićete **UnsupportedOperationException**.

Pomoću prikaza **CharBuffer** umeće se **charArray** u **ByteBuffer**. Nakon prikazivanja svih bajtova uverićete se da je podrazumevani redosled jednak kasnijem eksplicitno pozvanom poretku big endian, dok redosled little endian zamenjuje bajtove.

Rad s podacima pomoću bafera

U narednom dijagramu ilustrovani su odnosi između **nio** klasa, pa možete videti kako da premeštate i konvertujete podatke. Na primer, ako niz tipa **byte** želite da upišete u datoteku, onda treba da ga omotate metodom **ByteBuffer.wrap( )**, metodom **getChannel( )** otvorite kanal u toku **FileOutputStream**, a zatim iz tog **ByteBuffer**a podatke upišete u **FileChannel**.



Vodite računa o tome da je **ByteBuffer** jedini način za umetanje podataka u kanal i vađenje podataka iz kanala, i da od **ByteBuffer**a jedino možete napraviti samostalni bafer nekog od prostih tipova, sami ili nekom od „as" metoda. Dakle, bafer nekog od prostih tipova ne možete pretvoriti u **ByteBuffer**. Međutim, pošto podatke prostih tipova možete umetati u **ByteBuffer** i vaditi ih iz njega preko bafera prikaza, to i nije neko ograničenje.

Detaljno o baferima

Bafer se sastoji od podataka i četiri indeksa za pristupanje tim podacima i delotvoran rad s njima: *marker*, *položaj*, *kraj* i *kapacitet*. Postoje metode za postavljanje vrednosti tih indeksa, za njihovo resetovanje i za ispitivanje njihovih vrednosti.

capacity( )	Vraća kapacitet bafera.
clear( )	Briše sadržaj bafera, indeksu *položaja* zadaje vrednost nula, a indeksu *kraja* vrednost indeksa *kapacitet*. Ovom metodom čistite postojeći bafer.
flip( )	Indeksu *kraja* zadaje vrednost indeksa *položaja*, a indeksu *položaja* vrednost nula. Ovom metodom se bafer priprema za čitanje nakon upisivanja podataka.
limit( )	Vraća vrednost indeksa *kraj*.
limit(int kraj)	Zadaje vrednost indeksa *kraj*.
mark( )	Indeksu *marker* zadaje vrednost indeksa *položaja*.
position( )	Vraća vrednost indeksa *položaja*.
position(int pol)	Zadaje vrednost indeksa *položaja*.
remaining( )	Vraća (*kraj* – *položaj*).
hasRemaining( )	Vraća **true** ako u baferu ima elemenata između mesta na koja pokazuju indeksi *položaja* i *kraja*.

Ove indekse ažuriraju metode koje umeću i vade podatke iz bafera. Tako indeksi odražavaju izmene sprovedene u baferu.

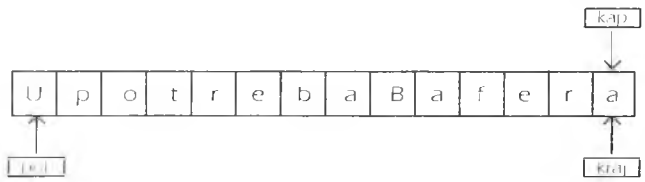
U narednom primeru upotrebljen je veoma jednostavan algoritam (zamena susednih znakova) da bi se ispreturali i ponovo vratili na mesta znakovi u **CharBuffer**u:

```
//: ui/UpotrebaBafera.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UpotrebaBafera {
  private static void simetricnoPreturanje(CharBuffer bafer){
    while(bafer.hasRemaining()) {
      bafer.mark();
      char c1 = bafer.get();
      char c2 = bafer.get();
      bafer.reset();
      bafer.put(c2).put(c1);
    }
```

```
  }
  public static void main(String[] args) {
    char[] podaci = "UpotrebaBafera".toCharArray();
    ByteBuffer bb = ByteBuffer.allocate(podaci.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(podaci);
    print(cb.rewind());
    simetricnoPreturanje(cb);
    print(cb.rewind());
    simetricnoPreturanje(cb);
    print(cb.rewind());
  }
} /* Ispis:
UpotrebaBafera
pUtoerabaBefar
UpotrebaBafera
*///:~
```

Iako se **CharBuffer** može napraviti neposredno metodom **wrap( )** od niza tipa **char**, umesto toga smo memoriju dodelili **ByteBuffer**u, a **CharBuffer** je napravljen kao prikaz toga **ByteBuffer**a. Time se ističe da je cilj uvek koristiti **ByteBuffer**, pošto on radi neposredno s kanalom.

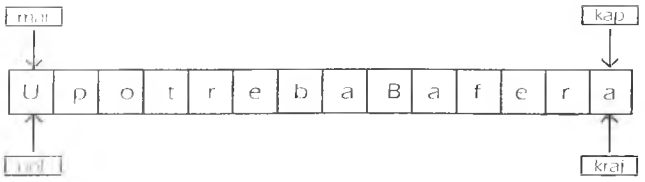
Evo kako izgleda bafer na ulazu u metodu **simetricnoPreturanje( )**:



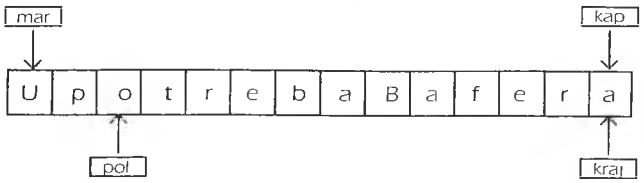
Indeks *položaja* pokazuje na prvi element bafera, a indeksi *kapacitet* i *kraj* na poslednji.

U metodi *simetricnoPreturanje( )*, petlja **while** iterira dok indeks *položaja* ne dođe do vrednosti *kraj*. Indeks (tekućeg) *položaja* bafera se menja kada se pozovu relativne funkcije **get( )** ili **put( )** (pozivi bez argumenata). Možete pozivati i apsolutne metode **get( )** i **put( )**, sa indeksom *položaja* elementa kao argumentom koji određuje mesto na kojem se odigrava vađenje (metodom **get( )**) odnosno umetanje (metodom **put( )**). Te (apsolutne) metode ne menjaju vrednost indeksa *položaja* bafera.

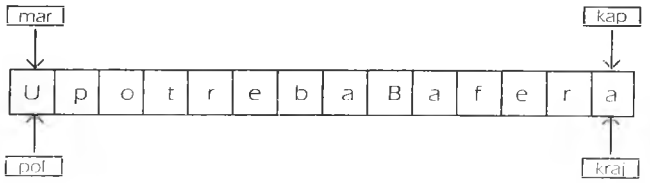
Kada kontrola uđe u petlju **while**, vrednost indeksa **marker** biva zadata pozivom metode **mark( )**. Tada je stanje bafera:



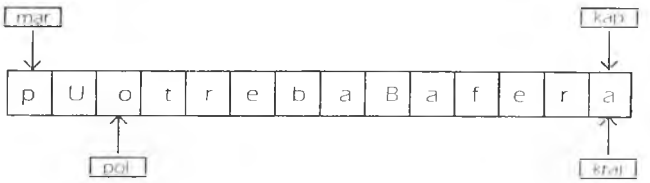
Dva poziva relativne metode **get( )** smeštaju vrednost prva dva znaka u promenljive **c1** i **c2**. Nakon ta dva poziva, bafer izgleda ovako:



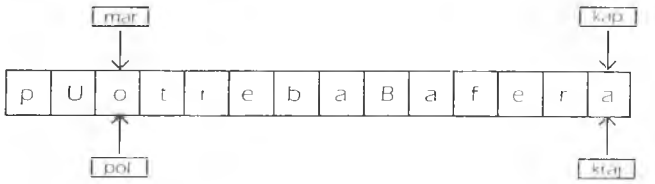
Da bismo zamenili mesta ta dva znaka, vrednost promenljive **c2** moramo upisati na mesto *položaj* = 0 , a vrednost promenljive **c1** na mesto *položaj* = 1. Za to možemo upotrebiti apsolutnu metodu **put( )** ili indeksu *položaja* zadati vrednost indeksa *marker*, što upravo i radi metoda **reset( )**:



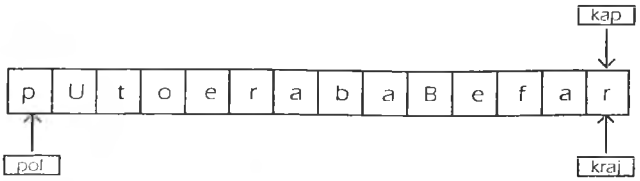
Dva poziva metode **put( )** upisuju **c2** i zatim **c1**:



Tokom sledeće iteracije petlje, indeksu *marker* se zadaje tekuća vrednost indeksa *položaja*:



Postupak se nastavlja do prolaska kroz ceo bafer. Na kraju petlje **while**, indeks *položaja* pokazuje na kraj bafera. Kada ispišete sadržaj bafera, ispisuju se samo znakovi između *položaja* i *kraja*. Dakle, da biste ispisali ceo sadržaj bafera, indeksu *položaja* morate metodom **rewind( )** zadati vrednost početka bafera. Evo stanja bafera nakon poziva metode **rewind( )** (vrednost indeksa *marker* postaje nedefinisana):



Kada se ponovo pozove metoda **simetricnoPreturanje( )**, **CharBuffer** se podvrgava istom postupku i vraća u svoje prvobitno stanje.

Datoteke preslikane u memoriju

Datoteke preslikane u memoriju omogućuju pravljenje i menjanje datoteka koje su prevelike da bi cele bile smeštene u memoriju. Kada je datoteka delom preslikana u memoriju, možete se pretvarati kako je cela u memoriji i da joj možete pristupati tretirajući je kao običan veliki niz. Time se znatno pojednostavljuje kôd koji se mora napisati za menjanje takve datoteke. Sledi jednostavan primer:

```
//: ui/VelikePreslikavaneDatoteke.java
// Pravljenje veoma velike datoteke pomoću preslikavanja.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class VelikePreslikavaneDatoteke {
  static int duzina = 0x8FFFFFF; // 128 MB
  public static void main(String[] args) throws Exception {
    MappedByteBuffer izlaz =
      new RandomAccessFile("test.dat", "rw").getChannel()
      .map(FileChannel.MapMode.READ_WRITE, 0, duzina);
    for(int i = 0; i < duzina; i++)
      izlaz.put((byte)'x');
    print("Pisanje završeno");
    for(int i = duzina/2; i < duzina/2 + 6; i++)
      printnb((char)izlaz.get(i));
  }
} ///:~
```

Da bismo obradili i čitanje i pisanje, počinjemo tako što pravimo (objekat tipa) **RandomAccessFile**, pribavimo kanal za tu datoteku, i zatim pozovemo metodu **map( )** koja proizvodi objekat tipa **MappedByteBuffer**, što je posebna vrsta direktnog bafera. Obratite pažnju na to da možete zadati početnu tačku i dužinu oblasti u datoteci koju želite da preslikate; to znači da imate mogućnost da preslikavate manje oblasti velikih datoteka.

MappedByteBuffer je izvedena iz klase **ByteBuffer**, pa ima sve njene metode. Ovde su prikazane samo veoma jednostavne primene metoda **put( )** i **get( )**, ali na raspolaganju su vam i metode kao **asCharBuffer( )** itd.

Datoteka napravljena u prethodnom programu dugačka je 128 MB, pa verovatno prevazilazi maksimalnu veličinu pojedinačne datoteke u memoriji koju OS vašeg računara dozvoljava. Izgleda kao da je cela datoteka odjedanput dostupna zato što se u memoriju smešta tek jedan njen deo, a ostatak se izmešta napolje. Na taj način možete menjati sadržaj veoma velikih datoteka (do 2 GB). Imajte u vidu da su, radi optimizovanja performansi, za preslikavanje datoteka u memoriju upotrebljene uslužne metode pripadnog operativnog sistema.

Performanse

Iako su performanse „starog" U/I sistema zasnovanog na tokovima poboljšane realizacijom pomoću **nio** klasa, obično se još mnogo brže pristupa datotekama preslikanim u memoriju. U narednom programu upoređene su njihove performanse (na veoma jednostavan način):

```
//: ui/UIPreslikavanih.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class UIPreslikanih {
  private static int brojUpisa = 4000000;
  private static int brojUpisaUBafer = 200000;
  private abstract static class Tester {
    private String ime;
    public Tester(String ime) { this.ime = ime; }
    public void runTest() {
      System.out.print(ime + ": ");
      try {
        long pocetak = System.nanoTime();
        test();
        double trajanje = System.nanoTime() - pocetak;
        System.out.format("%.2f\n", trajanje/1.0e9);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
    public abstract void test() throws IOException;
  }
  private static Tester[] ispitivanja = {
    new Tester("Upisivanje u tok") {
      public void test() throws IOException {
        DataOutputStream dos = new DataOutputStream(
          new BufferedOutputStream(
            new FileOutputStream(new File("temp.tmp"))));
        for(int i = 0; i < brojUpisa; i++)
          dos.writeInt(i);
        dos.close();
      }
    },
```

```
new Tester("Upisivanje u preslikanu") {
  public void test() throws IOException {
    FileChannel fc =
      new RandomAccessFile("temp.tmp", "rw")
      .getChannel();
    IntBuffer ib = fc.map(
      FileChannel.MapMode.READ_WRITE, 0, fc.size())
      .asIntBuffer();
    for(int i = 0; i < brojUpisa; i++)
      ib.put(i);
    fc.close();
  }
},
new Tester("Čitanje toka") {
  public void test() throws IOException {
    DataInputStream dis = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("temp.tmp")));
    for(int i = 0; i < brojUpisa; i++)
      dis.readInt();
    dis.close();
  }
},
new Tester("Čitanje preslikane") {
  public void test() throws IOException {
    FileChannel fc = new FileInputStream(
      new File("temp.tmp")).getChannel();
    IntBuffer ib = fc.map(
      FileChannel.MapMode.READ_ONLY, 0, fc.size())
      .asIntBuffer();
    while(ib.hasRemaining())
      ib.get();
    fc.close();
  }
},
new Tester("Čitanje iz toka/pisanje u tok") {
  public void test() throws IOException {
    RandomAccessFile raf = new RandomAccessFile(
      new File("temp.tmp"), "rw");
    raf.writeInt(1);
    for(int i = 0; i < brojUpisaUBafer; i++) {
      raf.seek(raf.length() - 4);
      raf.writeInt(raf.readInt());
    }
    raf.close();
  }
},
new Tester("Čitanje iz preslikane/pisanje u preslikanu") {
  public void test() throws IOException {
    FileChannel fc = new RandomAccessFile(
      new File("temp.tmp"), "rw").getChannel();
```

```
        IntBuffer ib = fc.map(
          FileChannel.MapMode.READ_WRITE, 0, fc.size())
          .asIntBuffer();
        ib.put(0);
        for(int i = 1; i < brojUpisaUBafer; i++)
          ib.put(ib.get(i - 1));
        fc.close();
      }
    }
  };
  public static void main(String[] args) {
    for(Tester test : ispitivanja)
      test.runTest();
  }
} /* Ispis: (90% podudaranja)
Upisivanje u tok: 0.56
Upisivanje u preslikanu: 0.12
Čitanje toka: 0.80
Čitanje preslikane: 0.07
Čitanje iz toka/pisanje u tok: 5.32
Čitanje iz preslikane/pisanje u preslikanu: 0.02
*///:~
```

Kao što ste videli u prethodnim primerima iz ove knjige, metoda **runTest( )** je upotrebljena u projektnom obrascu *Template Method* (Šablonska metoda) za pravljenje strukture za ispitivanje raznih realizacija metode **test( )** definisanih u anonimnim unutrašnjim potklasama. Svaka od tih potklasa obavlja jednu vrstu ispitivanja, pa te metode **test( )** predstavljaju i prototip za obavljanje raznih U/I operacija.

Premda izgleda kao da se za pisanje u preslikanu datoteku upotrebljava tok **FileOutputStream**, sav izlaz preslikanih datoteka mora upotrebljavati klasu **RandomAccessFile**, baš kao čitanje/pisanje u prethodnom kodu.

Obratite pažnju na to da **test( )** metode obuhvataju i vreme za inicijalizaciju raznih U/I objekata. Priprema preslikanih datoteka ume da bude skupa, ali je sveukupni dobitak značajan kada se uporedi sa stanjem pri korišćenju U/I sistema zasnovanih na toku.

Vežba 25: (6) Napravite eksperiment tako što ćete u primerima iz ovog poglavlja naredbu **ByteBuffer.allocate( )** promeniti u **ByteBuffer.allocateDirect( )**. Pokažite razlike u performansama, i proverite da li se primetno menja trajanje pokretanja programa.

Vežba 26: (3) Izmenite program **strings/JGrep.java** tako da koristi Java **nio** datoteke preslikane u memoriju po delovima.

Zaključavanje datoteka

Zaključavanje datoteka omogućuje sinhronizovanje pristupa datoteci kao deljenom resursu. Međutim, dve niti koje se takmiče za istu datoteku mogu biti u različitim JVM-ovima ili jedna može biti Java nit, a druga nit operativnog sistema. Zaključavanje datoteka vide ostali procesi operativnog sistema zato što se zaključavanje Java datoteka direktno preslikava u uslužnu metodu operativnog sistema za zaključavanje.

Sledi jednostavan primer zaključavanja datoteka.

```
//: ui/ZakljucavanjeDatoteka.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class ZakljucavanjeDatoteka {
  public static void main(String[] args) throws Exception {
    FileOutputStream fos= new FileOutputStream("datoteka.txt");
    FileLock fl = fos.getChannel().tryLock();
    if(fl != null) {
      System.out.println("Zaključana datoteka");
      TimeUnit.MILLISECONDS.sleep(100);
      fl.release();
      System.out.println("Otključana datoteka");
    }
    fos.close();
  }
} /* Ispis:
Zaključana datoteka
Otključana datoteka
*///:~
```

Bravu (objekat tipa **FileLock**) cele datoteke dobijate pozivom metoda **tryLock( )** ili **lock( )** za objekat tipa **FileChannel**. (**SocketChannel**, **DatagramChannel** i **ServerSocketChannel** ne treba zaključavati, pošto su to po svojoj prirodi entiteti jednog procesa; po pravilu se mrežna utičnica ne deli između dva procesa.) **tryLock( )** je neblokirajuća metoda. Ona će pokušati da prigrabi bravu za sebe, ali ako u tome ne uspe (kada neki drugi proces već ima tu bravu, a ona nije deljena), vratiće se bez rezultata. Metoda **lock( )** blokira sve dok ne pribavi bravu (engl. *lock*), ili dok nit koja je pozvala **lock( )** ne bude prekinuta, ili dok se ne zatvori kanal za koji je metoda **lock( )** pozvana. Brava se oslobađa (prepušta drugima) metodom **FileLock.release( )**.

Može se zaključati i deo (oblast) datoteke, i to pozivom:

```
tryLock(long položaj, long veličina, boolean deljena)
```

ili

```
lock(long položaj, long veličina, boolean deljena)
```

koji zaključava oblast (**veličina** – **položaj**). Trećim argumentom se zadaje da li je brava deljena (engl. *shared lock*).

Ako se metoda za zaključavanje pozove bez argumenata, njen učinak se ne menja s promenom veličine datoteke. Ukoliko se brava pribavi za oblast od indeksa **položaj** do indeksa **položaj**+**veličina**, a datoteka se poveća preko granice **položaj**+**veličina**, onda taj povećani deo datoteke neće biti zaključan. Pozivi metoda za zaključavanje bez argumenata, zaključavaju celu datoteku, čak i kada se ona poveća.

Podršku za isključive (engl. *exclusive lock*) ili deljene brave mora dati operativni sistem. Ukoliko operativni sistem ne podržava deljene brave, a primi zahtev da napravi jednu takvu, biće upotrebljena isključiva brava umesto deljene. Vrstu brave (deljena ili isključiva) možete saznati kad pozovete metodu **FileLock.isShared( )**.

Zaključavanje delova preslikane datoteke

Kao što je već bilo rečeno, obično se preslikavaju veoma velike datoteke. Ponekad treba zaključati delove takve velike datoteke, kako bi ostali procesi mogli da menjaju njene nezaključane delove. Primera radi, tako nešto činimo s bazom podataka da bi bila dostupna većem broju korisnika istovremeno.

Evo primera s dve niti, od kojih svaka zaključava određeni deo datoteke:

```
//: ui/ZakljucavanjePreslikanihDatoteka.java
// Zaključavanje delova preslikanih datoteka.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ZakljucavanjePreslikanihDatoteka {
  static final int DUZINA = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer izlaz =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, DUZINA);
    for(int i = 0; i < DUZINA; i++)
      izlaz.put((byte)'x');
    new ZakljucajIPromeni(izlaz, 0, 0 + DUZINA/3);
    new ZakljucajIPromeni(izlaz, DUZINA/2, DUZINA/2 + DUZINA/4);
  }
  private static class ZakljucajIPromeni extends Thread {
    private ByteBuffer baf;
    private int pocetak, kraj;
    ZakljucajIPromeni(ByteBuffer mbb, int pocetak, int kraj) {
      this.pocetak = pocetak;
      this.kraj = kraj;
      mbb.limit(kraj);
      mbb.position(pocetak);
      baf = mbb.slice();
      pocetak();
    }
    public void run() {
      try {
        // Isključiva brava bez preklapanja:
        FileLock fl = fc.lock(pocetak, kraj, false);
        System.out.println("Zakljucano: "+ pocetak +" do "+ kraj);
        // Obavi izmene:
        while(baf.position() < baf.limit() - 1)
```

```
            baf.put((byte)(baf.get() + 1));
          fl.release();
          System.out.println("Otkljucano: "+ pocetak +" do "+ kraj);
        } catch(IOException e) {
          throw new RuntimeException(e);
        }
      }
    }
} ///:~
```

Klasa niti **ZakljucajIPromeni** priprema oblast bafera i metodom **slice( )** pravi odsečak (engl. *slice*) koji će biti izmenjen, a u metodi **run( )** pribavlja se brava za kanal datoteke (ne možete pribaviti bravu bafera, nego samo kanala). Metoda **lock( )** poziva se veoma slično pribavljanju zaključane niti objekta – sada imate „kritičnu sekciju" sa isključivim pristupom tom delu datoteke.[5]

Brave se otključavaju automatski kada JVM izađe ili kada se zatvori kanal za koji je brava pribavljena, ali za otključavanje objekta tipa **FileLock** možete i eksplicitno da pozovete metodu **release( )** kao što je urađeno u prethodnom programu.

Komprimovanje

Javina U/I biblioteka sadrži klase koje podržavaju čitanje i upisivanje tokova u komprimovanom formatu. Te klase treba omotati oko postojećih U/I klasa.

Ove klase nisu izvedene iz klasa **Reader** i **Writer**, već su deo hijerarhija klasa **InputStream** i **OutputStream**. Razlog je to što biblioteka za komprimovanje radi s bajtovima, a ne sa znacima. Ipak, ponekad ćete biti primorani da kombinujete ove dve vrste tokova. (Zapamtite da za jednostavnu konverziju između ova dva tipa možete koristiti klase **InputStreamReader** i **OutputStreamReader**.)

Klasa za komprimovanje	Funkcija
CheckedInputStream	Metoda **GetCheckSum( )** vraća kontrolni zbir za bilo koji objekat klase **InputStream** (ne samo dekomprimovanje).
CheckedOutputStream	Metoda **GetCheckSum( )** vraća kontrolni zbir za bilo koji objekat klase **OutputStream** (ne samo komprimovanje).
DeflaterOutputStream	Osnovna klasa za kompresiju.
ZipOutputStream	Potklasa klase **DeflaterOutputStream** koja komprimuje podatke u formatu Zip.
GZIPOutputStream	Potklasa klase DeflaterOutputStream koja komprimuje podatke u formatu GZIP.
InflaterInputStream	Osnovna klasa za dekomprimovanje.
ZipInputStream	Potklasa klase **InflaterInputStream** koja dekomprimuje podatke snimljene u formatu Zip.
GZIPInputStream	Potklasa klase **InflaterInputStream** koja dekomprimuje podatke snimljene u formatu GZIP.

[5] Više informacija o nitima naći ćete u poglavlju *Paralelno izvršavanje*.

Iako postoje mnogi algoritmi za komprimovanje, Zip i GZIP se verovatno najčešće koriste. Zbog toga s komprimovanim podacima možete lako da radite pomoću brojnih alatki za čitanje i upisivanje u pomenutim formatima.

Jednostavno komprimovanje u formatu GZIP

Interfejs GZIP je jednostavan i stoga je verovatno podesniji za slučajeve kada treba komprimovati jedan tok podataka (a ne kontejner međusobno nesličnih podataka). Evo primera komprimovanja jedne datoteke:

```
//: ui/GZIPKomprimovanje.java
//  {Args: GZIPKomprimovanje.java}
import java.io.*;
import java.util.zip.*;

public class GZIPKomprimovanje {
  public static void main(String[] args)
  throws IOException {
    if (args.length == 0)  {
      System.out.println(
          "Upotreba: \nGZIPKomprimovanje datoteka\n" +
           "\tUpotrebljava GZIP komprimovanje da bi komprimovao " +
           "datoteku u test.gz");
      System.exit(1);
    }
    BufferedReader in = new BufferedReader(
        new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
        new GZIPOutputStream(
          new FileOutputStream("test.gz")));
    System.out.println("Upis u datoteku");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Citanje datoteke");
    BufferedReader in2 = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(
            new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /*  (Pokrenite da biste videli rezultate)  *///:~
```

Klase za komprimovanje prilično se jednostavno koriste – treba samo da omotate izlazni tok u klase **GZIPOutputStream** ili **ZipOutputStream**, a ulazni tok u klase **GZIPInputStream** ili **ZipInputStream**. Sve ostalo svodi se na uobičajeno čitanje i upisivanje.

Ovo je primer kombinovanja znakovno orijentisanih tokova s binarno orijentisanim tokovima: objekat **in** koristi klase iz hijerarhije **Reader,** dok konstruktor klase **GZIPOutputStream** prihvata samo objekat klase **OutputStream**, ali ne i klase **Writer**. Kada se datoteka otvori, **GZIPInputStream** se pretvara u tip **Reader**.

Komprimovanje većeg broja datoteka u formatu Zip

Biblioteka koja podržava format Zip mnogo je obimnija. Pomoću nje možete lako da komprimujete više datoteka, a postoji čak i posebna klasa koja olakšava postupak čitanja Zip datoteke. Ova biblioteka koristi standardni format Zip, pa bez problema radi sa svim alatkama koje se mogu preuzeti sa Interneta. Sledeći primer liči na prethodni, ali može da radi s proizvoljnim brojem argumenata s komandne linije. Osim toga, prikazuje i korišćenje **Checksum** klasa za izračunavanje i proveru kontrolnog zbira datoteke. Postoje dve vrste kontrolnih zbirova: **Adler32** (brži) i **CRC32** (sporiji, ali nešto tačniji):

```
//: ui/ZipKomprimovanje.java
// Koristi format Zip za komprimovanje proizvoljnog
// broja datoteka koji se zadaje na komandnoj liniji.
// {Args: ZipKomprimovanje.java}
import java.io.*;
import java.util.*;
import java.util.zip.*;
import static net.mindview.util.Print.*;

public class ZipKomprimovanje {
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(f, new Adler32());
    ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out = new BufferedOutputStream(zos);
    zos.setComment("Proba pakovanja u formatu Zip");
    // Ipak, nema odgovarajuće metode getComment().
    for(String arg : args) {
      print("Upis u datoteku " + arg);
      BufferedReader in =
          new BufferedReader(
          new FileReader(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // Kontrolni zbir je važeći tek po zatvaranju datoteke!
    print("Kontrolni zbir: " + csum.getChecksum().getValue());
```

Podršku za isključive (engl. *exclusive lock*) ili deljene brave mora dati operativni sistem. Ukoliko operativni sistem ne podržava deljene brave, a primi zahtev da napravi jednu takvu, biće upotrebljena isključiva brava umesto deljene. Vrstu brave (deljena ili isključiva) možete saznati kad pozovete metodu **FileLock.isShared( )**.

Zaključavanje delova preslikane datoteke

Kao što je već bilo rečeno, obično se preslikavaju veoma velike datoteke. Ponekad treba zaključati delove takve velike datoteke, kako bi ostali procesi mogli da menjaju njene nezaključane delove. Primera radi, tako nešto činimo s bazom podataka da bi bila dostupna većem broju korisnika istovremeno.

Evo primera s dve niti, od kojih svaka zaključava određeni deo datoteke:

```
//: ui/ZakljucavanjePreslikanihDatoteka.java
// Zaključavanje delova preslikanih datoteka.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ZakljucavanjePreslikanihDatoteka {
  static final int DUZINA = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer izlaz =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, DUZINA);
    for(int i = 0; i < DUZINA; i++)
      izlaz.put((byte)'x');
    new ZakljucajIPromeni(izlaz, 0, 0 + DUZINA/3);
    new ZakljucajIPromeni(izlaz, DUZINA/2, DUZINA/2 + DUZINA/4);
  }
  private static class ZakljucajIPromeni extends Thread {
    private ByteBuffer baf;
    private int pocetak, kraj;
    ZakljucajIPromeni(ByteBuffer mbb, int pocetak, int kraj) {
      this.pocetak = pocetak;
      this.kraj = kraj;
      mbb.limit(kraj);
      mbb.position(pocetak);
      baf = mbb.slice();
      pocetak();
    }
    public void run() {
      try {
        // Isključiva brava bez preklapanja:
        FileLock fl = fc.lock(pocetak, kraj, false);
        System.out.println("Zakljucano: "+ pocetak +" do "+ kraj);
        // Obavi izmene:
        while(baf.position() < baf.limit() - 1)
```

```
            baf.put((byte)(baf.get() + 1));
          fl.release();
          System.out.println("Otkljucano: "+ pocetak +" do "+ kraj);
        } catch(IOException e) {
          throw new RuntimeException(e);
        }
      }
    }
  } ///:~
```

Klasa niti **ZakljucajIPromeni** priprema oblast bafera i metodom **slice( )** pravi odsečak (engl. *slice*) koji će biti izmenjen, a u metodi **run( )** pribavlja se brava za kanal datoteke (ne možete pribaviti bravu bafera, nego samo kanala). Metoda **lock( )** poziva se veoma slično pribavljanju zaključane niti objekta – sada imate „kritičnu sekciju" sa isključivim pristupom tom delu datoteke.[5]

Brave se otključavaju automatski kada JVM izade ili kada se zatvori kanal za koji je brava pribavljena, ali za otključavanje objekta tipa **FileLock** možete i eksplicitno da pozovete metodu **release( )** kao što je urađeno u prethodnom programu.

Komprimovanje

Javina U/I biblioteka sadrži klase koje podržavaju čitanje i upisivanje tokova u komprimovanom formatu. Te klase treba omotati oko postojećih U/I klasa.

Ove klase nisu izvedene iz klasa **Reader** i **Writer**, već su deo hijerarhija klasa **InputStream** i **OutputStream**. Razlog je to što biblioteka za komprimovanje radi s bajtovima, a ne sa znacima. Ipak, ponekad ćete biti primorani da kombinujete ove dve vrste tokova. (Zapamtite da za jednostavnu konverziju između ova dva tipa možete koristiti klase **InputStreamReader** i **OutputStreamReader**.)

Klasa za komprimovanje	Funkcija
CheckedInputStream	Metoda **GetCheckSum( )** vraća kontrolni zbir za bilo koji objekat klase **InputStream** (ne samo dekomprimovanje).
CheckedOutputStream	Metoda **GetCheckSum( )** vraća kontrolni zbir za bilo koji objekat klase **OutputStream** (ne samo komprimovanje).
DeflaterOutputStream	Osnovna klasa za kompresiju.
ZipOutputStream	Potklasa klase **DeflaterOutputStream** koja komprimuje podatke u formatu Zip.
GZIPOutputStream	Potklasa klase DeflaterOutputStream koja komprimuje podatke u formatu GZIP.
InflaterInputStream	Osnovna klasa za dekomprimovanje.
ZipInputStream	Potklasa klase **InflaterInputStream** koja dekomprimuje podatke snimljene u formatu Zip.
GZIPInputStream	Potklasa klase **InflaterInputStream** koja dekomprimuje podatke snimljene u formatu GZIP.

[5] Više informacija o nitima naći ćete u poglavlju *Paralelno izvršavanje*.

Iako postoje mnogi algoritmi za komprimovanje, Zip i GZIP se verovatno najčešće koriste. Zbog toga s komprimovanim podacima možete lako da radite pomoću brojnih alatki za čitanje i upisivanje u pomenutim formatima.

Jednostavno komprimovanje u formatu GZIP

Interfejs GZIP je jednostavan i stoga je verovatno podesniji za slučajeve kada treba komprimovati jedan tok podataka (a ne kontejner međusobno nesličnih podataka). Evo primera komprimovanja jedne datoteke:

```
//: ui/GZIPKomprimovanje.java
//  {Args: GZIPKomprimovanje.java}
import java.io.*;
import java.util.zip.*;

public class GZIPKomprimovanje {
  public static void main(String[] args)
  throws IOException {
    if (args.length == 0)  {
      System.out.println(
          "Upotreba: \nGZIPKomprimovanje datoteka\n" +
           "\tUpotrebljava GZIP komprimovanje da bi komprimovao " +
           "datoteku u test.gz");
      System.exit(1);
    }
    BufferedReader in = new BufferedReader(
        new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
        new GZIPOutputStream(
          new FileOutputStream("test.gz")));
    System.out.println("Upis u datoteku");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Citanje datoteke");
    BufferedReader in2 = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(
            new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /*  (Pokrenite da biste videli rezultate)  *///:~
```

Klase za komprimovanje prilično se jednostavno koriste – treba samo da omotate izlazni tok u klase **GZIPOutputStream** ili **ZipOutputStream**, a ulazni tok u klase **GZIPInputStream** ili **ZipInputStream**. Sve ostalo svodi se na uobičajeno čitanje i upisivanje.

Ovo je primer kombinovanja znakovno orijentisanih tokova s binarno orijentisanim tokovima: objekat **in** koristi klase iz hijerarhije **Reader**, dok konstruktor klase **GZIPOutputStream** prihvata samo objekat klase **OutputStream**, ali ne i klase **Writer**. Kada se datoteka otvori, **GZIPInputStream** se pretvara u tip **Reader**.

Komprimovanje većeg broja datoteka u formatu Zip

Biblioteka koja podržava format Zip mnogo je obimnija. Pomoću nje možete lako da komprimujete više datoteka, a postoji čak i posebna klasa koja olakšava postupak čitanja Zip datoteke. Ova biblioteka koristi standardni format Zip, pa bez problema radi sa svim alatkama koje se mogu preuzeti sa Interneta. Sledeći primer liči na prethodni, ali može da radi s proizvoljnim brojem argumenata s komandne linije. Osim toga, prikazuje i korišćenje **Checksum** klasa za izračunavanje i proveru kontrolnog zbira datoteke. Postoje dve vrste kontrolnih zbirova: **Adler32** (brži) i **CRC32** (sporiji, ali nešto tačniji):

```
//: ui/ZipKomprimovanje.java
// Koristi format Zip za komprimovanje proizvoljnog
// broja datoteka koji se zadaje na komandnoj liniji.
// {Args: ZipKomprimovanje.java}
import java.io.*;
import java.util.*;
import java.util.zip.*;
import static net.mindview.util.Print.*;

public class ZipKomprimovanje {
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(f, new Adler32());
    ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out = new BufferedOutputStream(zos);
    zos.setComment("Proba pakovanja u formatu Zip");
    // Ipak, nema odgovarajuće metode getComment().
    for(String arg : args) {
      print("Upis u datoteku " + arg);
      BufferedReader in =
          new BufferedReader(
          new FileReader(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // Kontrolni zbir je važeći tek po zatvaranju datoteke!
    print("Kontrolni zbir: " + csum.getChecksum().getValue());
```

```
    // Sada raspakujemo datoteke:
    print("Čitanje datoteke ");
    FileInputStream fi = new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(fi, new Adler32());
    ZipInputStream in2 = new ZipInputStream(csumi);
    BufferedInputStream bis = new BufferedInputStream(in2);
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      print("Čitanje datoteke " + ze);
      int x;
      while((x = bis.read()) != -1)
        System.out.write(x);
    }
    if(args.length == 1)
    print("Kontrolni zbir: " + csumi.getChecksum().getValue());
    bis.close();
    // Drugi način za otvaranje i čitanje zip datoteka:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      print("Datoteka: " + ze2);
      // ... i raspakivanje podataka isto kao prethodno
    }
    /*  if(args.length == 1)  */
  }
} /*  (Pokrenite da biste videli rezultate)  *///:~
```

Metodi **putNextEntry( )** morate da prosledite objekat klase **ZipEntry** za svaku datoteku koja se dodaje u arhivu. Takav objekat sadrži obiman interfejs koji omogućuje čitanje i zadavanje svih podataka o datom elementu Zip datoteke: ime, komprimovana i nekomprimovana veličina, datum, kontrolni zbir, dodatna polja podataka, komentar, metoda kompresije i da li se radi o stavki direktorijuma. Međutim, Javina biblioteka ne podržava uvođenje lozinke iako je format Zip omogućuje. Pored toga, mada klase **CheckedInputStream** i **CheckedOutputStream** podržavaju oba tipa kontrolnih zbirova (**Adler32** i **CRC32**), klasa **ZipEntry** podržava samo interfejs za kontrolni zbir tipa CRC. To ograničenje nameće format Zip, ali zbog toga se ne može koristiti brži algoritam **Adler32**.

Klasa **ZipInputStream** ima metodu **getNextEntry( )** za raspakivanje arhive. Ta metoda vraća sledeći objekat klase **ZipEntry** ako on postoji. Postoji i kraća varijanta – pročitajte datoteku pomoću klase **ZipFile** koja ima metodu **entries( )**, a ona vraća objekat tipa **Enumeration** u kojem su nabrojani objekti klase **ZipEntries**.

Da biste pročitali kontrolni zbir, morate na neki način da obezbedite pristup odgovarajućem objektu klase **Checksum**. Pri tom se zadržava referenca na objekte tipa **CheckedOutputStream** i **CheckedInputStream**, ali možete da se oslonite i na referencu na objekat tipa **Checksum**.

Zbunjujuća metoda u Zip tokovima je **setComment( )**. Kao što je pokazano u gornjem primeru, prilikom upisivanja u datoteku možete da stavite komentar, ali ne postoji način

da ga pronađete u klasi **ZipInputStream**. Komentari su potpuno podržani samo u klasi **ZipEntry**, i to pojedinačno, po svakom unosu u datoteku.

Naravno, pri korišćenju klasa za komprimovanje **GZIP** ili **Zip** niste ograničeni na datoteke – možete da komprimujete bilo šta, uključujući i podatke s mrežnog priključka.

Java arhive (JAR)

Format Zip se koristi i u datotekama formata JAR (Java ARhiva) koje objedinjuju grupe datoteka u jednu komprimovanu datoteku, slično Zip arhivama. Kao i sve ostalo u Javi, JAR datoteke mogu da se koriste na raznim platformama. U Java arhive možete da smestite audio datoteke, slike, ali i datoteke s klasama.

JAR arhive su naročito korisne pri radu s Internetom. Pre njih, čitač Weba je morao da šalje nekoliko zahteva Web serveru da bi preuzeo sve datoteke jednog apleta. Osim toga, nijedna od tih datoteka nije bila komprimovana. Pakovanjem svih datoteka određenog apleta u jednu JAR arhivu, omogućuje se njihovo istovremeno preuzimanje samo jednim zahtevom serveru, a prenos je brži zbog kompresije. Svakom elementu JAR datoteke može se radi bezbednosti pridružiti i digitalan potpis.

JAR arhiva se sastoji od zbirke datoteka u formatu Zip i manifest datoteke, što je, u stvari, opis arhive. (Manifest datoteku možete da napravite sami ili će umesto vas to učiniti program **jar**.) Više informacija o manifest datotekama pronaći ćete u dokumentaciji razvojnog paketa.

Uslužni program **jar** koji se dobija uz razvojni paket za Javu kompanije Sun, automatski komprimuje datoteke koje odaberete. Poziva se s komandne linije:

```
jar [opcije] odredište  [manifest]  ulaznedatoteke
```

Opcije su samo zbirka slova (nisu neophodne crtice ili neki drugi pomoćni znaci). Korisnici Unixa/Linuxa primetiće sličnost s alatkom **tar**. Opcije su:

c	Pravi novu praznu arhivu.
t	Lista sadržaj.
x	Dekomprimuje sve datoteke iz arhive.
x datoteka	Dekomprimuje navedenu datoteku.
f	Kaže „Sada ću ti dostaviti ime datoteke". Ako je ne koristite, **jar** pretpostavlja da će ulazni podaci stići iz standardnog ulaznog toka ili, ako pravi datoteku, da izlazne podatke treba da pošalje na standardni izlazni tok.
m	Kaže da će prvi argument biti ime manifest datoteke koju je napravio korisnik.
v	Detaljno ispisuje informacije tokom rada.
o	Samo snima datoteke, a ne komprimuje ih. (Koristi se za pravljenje JAR datoteke koju možete da stavite u putanju klasa.)
M	Ne pravi automatski manifest datoteku.

Ako spisak datoteka koje treba arhivirati sadrži poddirektorijum, taj poddirektorijum se automatski arhivira sa svim svojim poddirektorijumima itd. Čuvaju se i informacije o putanji.

Evo nekih tipičnih načina za korišćenje programa **jar**. Sledeća komanda pravi JAR arhivu **mojaJarDatoteka.jar** koja sadrži sve datoteke s klasama u tekućem direktorijumu, kao i automatski generisanu manifest datoteku:

```
jar cf mojaJarDatoteka.jar *.class
```

Sledeća komanda radi isto što i prethodni primer, ali uključuje i korisnički napravljenu manifest datoteku **mojManifest.mf**:

```
jar cmf mojaJarDatoteka mojManifest.mf *.class
```

Naredna komanda daje sadržaj arhive **mojaJarDatoteka.jar** u obliku liste datoteka:

```
jar tf mojaJarDatoteka.jar
```

Detaljnije ispisuje informacije o datotekama u arhivi **mojaJarDatoteka.jar**:

```
jar tvf mojaJarDatoteka.jar
```

Pod pretpostavkom da su **audio**, **klase** i **slike** poddirektorijumi, na ovaj način se svi objedinjuju u datoteku **mojaAplikacija.jar**. Tokom izvršavanja programa **jar**, ispisuju se dodatne informacije zbog opcije **v**:

```
jar cvf mojaAplikacija.jar audio klase slike
```

Ako napravite JAR datoteku korišćenjem opcije **0** (nula), moći ćete da je stavite u putanju klasa:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Nakon toga Java može da traži datoteke s klasama u bibliotekama **lib1.jar** i **lib2.jar**.

Alatka **jar** nije toliko korisna kao uslužni program **zip**. Na primer, ne možete da dodajete ili ažurirate datoteke u postojećoj JAR arhivi, tj. ovakve arhive morate da pravite „od nule". Takođe, datoteke ne možete da premeštate u JAR arhivu i da ih nakon premeštanja brišete. Međutim, JAR datoteku, koja je napravljena u jednoj platformi, moći će da pročita alatka **jar** na bilo kojoj drugoj platformi, što je problem koji ponekad muči uslužne programe **zip**.

Kao što ćete videti u poglavlju *Grafička korisnička okruženja*, JAR arhive se koriste i za pakovanje zrna Jave.

Serijalizovanje objekata

Kada napravite objekat, on postoji onoliko dugo koliko vam je potreban, ali ni pod kojim okolnostima ne može postojati kada program prestane da se izvršava. Koliko god da to isprva ima smisla, ima situacija u kojima bi bilo izvanredno korisno kada bi objekat mogao da postoji i sačuva svoje informacije čak i kada se program ne izvršava. Tada bi objekat postojao i kada se program pokrene sledeći put, i sadržao bi iste informacije koje je imao i kada se program izvršavao prethodni put. Naravno, to možete postići upisivanjem informacija u datoteku ili bazu podataka, ali u duhu politike da sve treba da bude objekat,

bilo bi veoma pogodno deklarisati da je određeni objekat „trajan", a da se Java pri tom automatski pobrine za sve pojedinosti.

Serijalizovanje objekata u Javi omogućuje pretvaranje svakog objekta koji realizuje interfejs **Serializable** u niz bajtova iz kojeg se taj objekat kasnije može potpuno rekonstruisati. To važi čak i u mrežama, što znači da mehanizam serijalizovanja automatski kompenzuje razlike između operativnih sistema. Dakle, možete da napravite neki objekat na računaru koji radi pod Windowsom, serijalizujete ga i pošaljete preko mreže na Unix računar gde će on biti pravilno rekonstruisan. Ne morate da brinete o predstavljanju objekata na različitim računarima, redosledu bajtova niti o drugim detaljima.

Samo po sebi, serijalizovanje objekata je zanimljivo zato što omogućuje realizovanje *lake trajnosti* (engl. *lightweight persistence*). Setite se da trajnost znači kako životni vek objekta nije određen izvršavanjem programa, već objekat živi i *između* izvršavanja programa. Efekat trajnosti možete da postignete upisivanjem serijalizovanog objekta na disk i njegovim rekonstruisanjem pri ponovnom pozivanju programa. Epitet laka potiče otud što ne možete da definišete objekat pomoću neke rezervisane reči za trajnost i da prepustite sistemu da brine o detaljima (iako će i to možda jednog dana biti moguće). Umesto toga, morate eksplicitno da serijalizujete i deserijalizujete objekat u programu. Ukoliko vam treba ozbiljniji mehanizam trajnosti, razmotrite alatku Hibernate (*http://hibernate.sourceforge.net*). Više informacija o njoj potražite u knjizi *Thinking in Enterprise Java*, koja se može preuzeti sa adrese *www.MindView.net*.

Serijalizovanje objekata je dodato u jezik radi podrške dvema važnim funkcijama. Javino *daljinsko pozivanje metoda* (engl. *remote method invocation, RMI*) omogućuje objektima na drugim računarima da se ponašaju kao da se nalaze na vašem računaru. Pri slanju poruka udaljenim objektima, serijalizovanje je neophodno da bi se preneli argumenti i povratne vrednosti. Daljinsko pozivanje metoda objašnjeno je u knjizi *Thinking in Enterprise Java*.

Serijalizovanje objekata je neophodno i za zrna Jave, o čemu će biti reči u poglavlju *Grafička korisnička okruženja*. Kada se koristi zrno, informacije o njegovom stanju po pravilu se podešavaju već tokom projektovanja. One se moraju sačuvati i rekonstruisati kasnije, kada se program pokrene. Taj zadatak obavlja serijalizovanje objekata.

Serijalizovanje objekta je prilično jednostavno, pod uslovom da on realizuje interfejs **Serializable** (ovaj interfejs je samo indikator i nema metode). Kada je serijalizovanje dodato u jezik, mnoge klase standardne biblioteke promenjene su da bi ga podržale, uključujući i omotače za proste tipove, sve kontejnerske klase i mnoge druge. Mogu se serijalizovati čak i objekti klase **Class**.

Da bi se objekat serijalizovao, treba napraviti neku vrstu izlaznog toka i omotati ga unutar objekta klase **ObjectOutputStream**. Nakon toga treba samo pozvati metodu **writeObject( )** i objekat će biti serijalizovan i poslat u izlazni tok. (Serijalizovanje objekata je binarno orijentisano i zbog toga koristi hijerarhije klasa **InputStream** i **OutputStream**.) Za obrnut postupak, objekat klase **InputStream** se omotava unutar objekta tipa **ObjectInputStream** i poziva se metoda **readObject( )**. Kao i obično, dobija se referenca na objekat klase **Object**, pa je neophodna konverzija naniže u ispravan tip.

Posebno pametno rešenje primenjeno u serijalizovanju objekata jeste to što se pored čuvanja slike objekta prate i sve reference koje on sadrži, a čuvaju se i *ti* objekti, prate se

njihove reference itd. Ovaj princip se ponekad naziva „mreža objekata" povezana s jednim objektom, i ona sadrži niz referenci na objekte i njihove članove. Kada biste morali da održavate sopstvenu šemu za serijalizovanje, bilo bi preteško napisati kôd koji bi pratio sve veze. Izgleda da serijalizovanje objekata u Javi omogućuje da se to postigne bez po muke, verovatno korišćenjem optimizovanog algoritma koji krstari po mreži objekata. U sledećem primeru testira se mehanizam serijalizovanja pravljenjem „crva" od povezanih objekata, pri čemu svaki pokazuje na sledeći segment crva i sadrži niz referenci na objekte drugačije klase, nazvane **Podatak**:

```
//: ui/Crv.java
// Prikazuje serijalizaciju objekata.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*

class Podatak implements Serializable {
  private int n;
  public Podatak(int n) { this.n = n; }
  public String toString() {return Integer.toString(n);}
}

public class Crv implements Serializable {
  private static Random rand = new Random(47);
  }
  private Podatak[] d = {
    new Podatak(rand.nextInt(10)),
    new Podatak(rand.nextInt(10)),
    new Podatak(rand.nextInt(10))
  };
  private Crv sledeci;
  private char c;
  // Vrednost i == broj segmenata
  public Crv(int i, char x) {
    print(" Konstruktor crva: " + i);
    c = x;
    if(--i > 0)
      sledeci = new Crv(i, (char)(x + 1));
  }
  public Crv() {
    print("Podrazumevani konstruktor");
  }
  public String toString() {
    StringBuilder rezultat = new StringBuilder(":");
    rezultat.append(c);
    rezultat.append("(");
    for(Podatak pod : d);
        rezultat.append(pod);
    rezultat.append(")");
    if(sledeci!= null)
```

```
        rezultat.append(sledeci);
    return rezultat.toString( );
  }
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Crv w = new Crv(6, 'a');
    print("w = " + w);
    ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("crv.out"));
    out.writeObject("Pamcenje crva\n");
    out.writeObject(w);
    out.close(); // Prelazi u sledeci red
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("crv.out"));
    String s = (String)in.readObject();
    Crv w2 = (Crv)in.readObject();
    print(s + "w2 = " + w2);
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out2 = new ObjectOutputStream(bout);
    out2.writeObject("Pamcenje crva\n");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 = new ObjectInputStream(
        new ByteArrayInputStream(bout.toByteArray()));
    s = (String)in2.readObject();
    Crv w3 = (Crv)in2.readObject();
    print(s + "w3 = " + w3);
  }
} /* Ispis:
Konstruktor crva: 6
Konstruktor crva: 5
Konstruktor crva: 4
Konstruktor crva: 3
Konstruktor crva: 2
Konstruktor crva: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Pamcenje crva
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Pamcenje crva
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

Da bi bilo zanimljivije, niz objekata klase **Podatak** unutar crva inicijalizuje se slučajnim brojevima. (Na taj način nećete posumnjati da prevodilac zadržava kakve metainformacije.) Svaki segment klase **Crv** označava se objektom tipa **char** koji se automatski generiše tokom rekurzivnog pravljenja ulančane liste crva. Kada napravite objekat klase **Crv**, saopštavate konstruktoru koliko treba da bude dugačak. Da bi napravio referencu na **sledeci**, on poziva konstruktora klase **Crv** s dužinom koja je manja za jedan itd. Poslednjoj referenci **sledeci** ostaje vrednost **null** koja ukazuje na to da je crv završen.

Svrha svega bila je da se napravi nešto što je dovoljno složeno da se ne može lako serijalizovati ručno. Postupak serijalizovanja, međutim, prilično je jednostavan. Kada se napravi **ObjectOutputStream** od nekog drugog toka, serijalizuje ga metoda **writeObject( )**. Obratite pažnju i na poziv metode **writeObject( )** za objekat tipa **String**. Pomoću istih metoda kao za **DataOutputStream** možete da upisujete i sve proste tipove (oni koriste isti interfejs).

Postoje dva zasebna odeljka koda koji izgledaju slično. Prvi upisuje i čita datoteku, a drugi čita i upisuje u niz bajtova. Serijalizovanje omogućuje čitanje ili upisivanje objekta u bilo koji **DataInputStream** ili **DataOutputStream**, uključujući i mrežu, kao što je navedeno u knjizi *Thinking in Enterprise Java.*

Primećujete li da deserijalizovan objekat zaista sadrži sve veze koje su postojale u originalnom objektu?

U postupku deserijalizovanja objekta interfejsa **Serializable** ne poziva se nijedan konstruktor, čak ni podrazumevani. Ceo objekat se rekonstruiše na osnovu podataka iz ulaznog toka.

Vežba 27: (1) Napravite klasu koja realizuje interfejs **Serializable** i sadrži referencu na objekat neke druge klase koja se može serijalizovati. Napravite instancu svoje klase, serijalizujte je tako da bude snimljena na disk, zatim je rekonstruišite i proverite da li je postupak ispravno sproveden.

Pronalaženje klase

Možda se pitate šta je potrebno da bi se objekat rekonstruisao iz serijalizovanog stanja. Na primer, pretpostavimo da ste serijalizovali objekat i poslali ga nekom drugom računaru kao datoteku ili preko mreže. Da li bi program na drugom računaru mogao da rekonstruiše objekat samo na osnovu sadržaja datoteke?

Najbolji način da se odgovori na ovo pitanje jeste, kao i obično, da se proba. Sledeća datoteka treba da se smesti u poddirektorijum ovoga poglavlja:

```
//: ui/Zeleni.java
// Klasa koja se može serijalizovati.
import java.io.*;
public class Zeleni implements Serializable {} ///:~
```

Datoteka koja pravi i serijalizuje objekat klase **Zeleni** nalazi se u istom direktorijumu:

```
//: ui/ZamrzavanjeZelenog.java
// Pravi serijalizovanu izlaznu datoteku.
import java.io.*;

public class ZamrzavanjeZelenog {
  public static void main(String[] args) throws Exception {
    ObjectOutput izlaz = new ObjectOutputStream(
        new FileOutputStream("dosije.X"));
    Zeleni zorcon = new Zeleni();
```

```
    izlaz.writeObject(zorcon);
  }
} ///:~
```

Umesto da hvata i obrađuje izuzetke, ovaj program ih iz metode **main( )** prosleđuje na konzolu.

Kada prevedete i pokrenete program, on će dobijenu datoteku **dosije.X** smestiti u direktorijum **ui**. U njegov poddirektorijum **dosijex** stavite sledeći listing:

```
//: ui/dosijex/MaliZeleni.java
// Pokušaj da se serijalizovana datoteka rekonstruiše
// bez klase objekata uskladištenih u toj datoteci.
// {RunByHand}
import java.io.*;

public class MaliZeleni {
  public static void main(String args[]) throws Exception {
    ObjectInputStream ulaz = new ObjectInputStream(
        new FileInputStream (new File("..", "dosije.X")));
    Object misterija = ulaz.readObject();
    System.out.println(misterija.getClass());
  }
} /* Ispis:
class Zeleni
///:~
```

Čak i otvaranje datoteke i učitavanje objekta **misterija** zahteva objekat klase **Class** za klasu **Zeleni**; Javina virtuelna mašina (JVM) neće moći da nađe datoteku **Zeleni.class**, osim ako se ona nekim slučajem ne nađe u putanji klase, što nije slučaj u ovom primeru, pa će generisati izuzetak **ClassNotFoundException**. (Kao i obično, svaki dokaz o postojanju vanzemaljaca nestaje pre nego što se može proveriti njegova verodostojnost!) Javina virtuelna mašina mora da pronađe odgovarajuću datoteku **.class**.

Upravljanje serijalizovanjem

Kao što vidite, podrazumevani mehanizam serijalizovanja lako se koristi. Međutim, šta ako imate specijalne potrebe? Možda posebno brinete o bezbednosti i zato ne želite da serijalizujete delove objekta, ili možda nema smisla da se neki podobjekat serijalizuje ako će morati ponovo da se pravi kada se objekat bude rekonstruisao.

Postupak serijalizovanja možete da kontrolišete realizovanjem interfejsa **Externalizable** umesto interfejsa **Serializable**. Interfejs **Externalizable** proširuje interfejs **Serializable** i dodaje mu dve metode, **writeExternal( )** i **readExternal( )**, koje se automatski pozivaju tokom serijalizovanja i deserijalizovanja objekta, da bi omogućile izvršavanje specijalnih operacija.

Sledeći primer prikazuje jednostavne realizacije metoda interfejsa **Externalizable**. Klase **Blip1** i **Blip2** su skoro iste, a malu razliku između njih primetićete ako proučite kōd:

```
//: ui/Blipovi.java
// Jednostavno korišćenje interfejsa Externalizable i jedna zamka.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
  public Blip1() {
    print("Konstruktor za Blip1 ");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    print("Konstruktor za Blip2 ");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip2.readExternal");
  }
}

public class Blipovi {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Pravljenje objekata:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Blips.out"));
    print("Snimanje objekata:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    // Sada ih vraćamo:
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Blips.out"));
    print("Rekonstruisanje b1:");
    b1 = (Blip1)in.readObject();
    // UPS! Generiše izuzetak:
```

```
    //! print("Rekonstruisanje b2:");
    //! b2 = (Blip2)in.readObject();
  }
} /* Ispis:
Pravljenje objekata:
Konstruktor za Blip1
Konstruktor za Blip2
Snimanje objekata:
Blip1.writeExternal
Blip2.writeExternal
Rekonstruisanje b1:
Konstruktor za Blip1
Blip1.readExternal
*///:~
```

Objekat **Blip2** nije rekonstruisan zato što bi pokušaj da se to uradi generisao izuzetak. Vidite li razliku između klasa **Blip1** i **Blip2**? Konstruktor za **Blip1** je javan, dok konstruktor za **Blip2** nije, i to prouzrokuje izuzetak prilikom rekonstruisanja. Konstruktor za klasu **Blip2** pretvorite u javni i uklonite komentare iza **//!** da biste videli ispravne rezultate.

Kada se objekat **b1** rekonstruiše, poziva se podrazumevani konstruktor za **Blip1**. To se razlikuje od postupka rekonstruisanja objekta interfejsa **Serializable**, u kome se objekat potpuno rekonstruiše na osnovu sačuvanih podataka, bez pozivanja konstruktora. Uz objekat interfejsa **Externalizable** podrazumevani konstruktor se ponaša na uobičajen način (uključujući i inicijalizovanje tokom definisanja polja), a *zatim* se poziva metoda **readExternal**( ). O ovome morate voditi računa, posebno o činjenici da se uvek pozivaju podrazumevani konstruktori, kako biste postigli ispravno ponašanje objekata interfejsa **Externalizable**.

Evo primera koji pokazuje šta morate da uradite da biste sačuvali i rekonstruisali objekat tipa **Externalizable**:

```
//: ui/Blip3.java
// Rekonstruisanje objekta externalizable.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // Nema inicijalizacije
  public Blip3() {
    print("Konstruktor za Blip3");
    // s, i nisu inicijalizovani
  }
  public Blip3(String x, int a) {
    print("Blip3(String x, int a)");
    s = x;
    i = a;
    // s i i se inicijalizuju samo u nepodrazumevanom konstruktoru.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
```

```
    throws IOException {
      print("Blip3.writeExternal");
      // Ovo morate da uradite:
      out.writeObject(s);
      out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
      print("Blip3.readExternal");
      // Ovo morate da uradite:
      s = (String)in.readObject();
      i =in.readInt();
    }
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
      print("Pravljenje objekata:");
      Blip3 b3 = new Blip3("tekst " , 47);
      print(b3);
      ObjectOutputStream o = new ObjectOutputStream(
          new FileOutputStream("Blip3.out"));
      print("Snimanje objekta:");
      o.writeObject(b3);
      o.close();
      // Sada ga vraćamo:
      ObjectInputStream in = new ObjectInputStream(
          new FileInputStream("Blip3.out"));
      print("Rekonstruisanje b3:");
      b3 = (Blip3)in.readObject();
      print(b3);
    }
} /* Ispis:
Pravljenje objekata:
Blip3(String x, int a)
tekst 47
Snimanje objekta:
Blip3.writeExternal
Rekonstruisanje b3:
Konstruktor za Blip3
Blip3.readExternal
tekst 47
*///:~
```

Polja **s** i **i** inicijalizuju se tek u drugom konstruktoru, a ne u podrazumevanom. Ako ih ne inicijalizujete u metodi **readExternal( )**, **s** će imati vrednost **null**, a **i** vrednost nula, pošto se memorijska lokacija za nove objekte popunjava nulama. Ako dva reda koda posle rečenice „Ovo morate da uradite" pretvorite u komentare i pokrenete program, videćete da, nakon rekonstruisanja objekta, **s** ima vrednost **null**, a **i** ima vrednost nula.

Ako izvodite nešto iz objekta tipa **Externalizable**, obično ćete pozivati verzije metoda **writeExternal( )** i **readExternal( )** osnovne klase da biste obezbedili pravilno snimanje i rekonstruisanje komponenata osnovne klase.

Da bi sve radilo kako treba, znači da, pored upisivanja važnih podataka iz objekta tokom izvršavanja metode **writeExternal( )** (ne postoji podrazumevano ponašanje kojim se upisuju bilo koji članovi objekta interfejsa **Externalizable**), morate i da rekonstruišete te podatke u metodi **readExternal( )**. To isprva može da zbuni zato što bi ponašanje podrazumevanog konstruktora objekta **Externalizable** moglo stvoriti pogrešan utisak da se neka vrsta pamćenja i rekonstruisanja dešava automatski. To se *ne* dešava.

Vežba 28: (2) U programu **Blipovi.java** kopirajte datoteku i preimenujte je u **ProveraBlipova.java**. Preimenujte i klasu **Blip2** u **ProveraBlipova** (prepravite je u javnu, a ispred klase **Blipovi** uklonite oznaku **public**). Uklonite oznake **//!** iz datoteke i izvršite novi program. Potom postavite znakove za komentar ispred podrazumevanog konstruktora za klasu **ProveraBlipova**. Izvršite program i objasnite zašto radi. Obratite pažnju na to da nakon prevođenja morate da izvršite program sa „**java Blipovi**", zato što se metoda **main( )** i dalje nalazi u klasi **Blipovi**.

Vežba 29: (2) U programu **Blip3.java**, pretvorite u komentar po dva reda iza rečenica „Ovo morate da uradite:" i pokrenite program. Objasnite rezultat i zašto se on razlikuje od slučaja kada su ta dva reda u programu.

Rezervisana reč transient

Ako upravljate serijalizovanjem, možda nećete želeti da Javin mehanizam serijalizovanja automatski snima i rekonstruiše neki podobjekat. To je često slučaj kada se u podobjektu čuvaju poverljive informacije koje ne želite da serijalizujete, npr. lozinka. Čak i ako je takva informacija u objektu privatna, nakon serijalizovanja neko bi mogao da je pročita iz datoteke ili da je presretne prilikom prenosa preko mreže.

Jedan način za sprečavanje serijalizovanja osetljivih delova objekta jeste da se klasa realizuje kao **Externalizable**, kao što je upravo pokazano. Na taj način ništa se ne serijalizuje automatski, to jest možete eksplicitno da serijalizujete samo neophodne delove unutar metode **writeExternal( )**.

Međutim, ako radite sa objektom tipa **Serializable**, serijalizacija se obavlja automatski. Da biste to kontrolisali, serijalizovanje bilo kog elementa možete da isključite rezervisanom rečju **transient**, koja kaže: „Nemoj da se trudiš da ovo rekonstruišeš – ja ću se postarati za to."

Na primer, posmatrajmo objekat **Prijavljivanje** koji čuva informacije o određenom prijavljivanju na sistem. Pretpostavimo da nakon provere ispravnosti prijavljivanja želite da snimite podatke, ali bez lozinke. To je najlakše uraditi tako što se implementira (realizuje) interfejs **Serializable** a polje **lozinka** označi kao **transient**. Evo kako to izgleda:

```
//: ui/Prijavljivanje.java
// Prikazuje korišćenje rezervisane reci transient.
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Prijavljivanje implements Serializable {
  private Date datum = new Date();
```

```
  private String korisnickoIme;
  private transient String lozinka;
  public Prijavljivanje(String ime, String loz) {
    korisnickoIme = ime;
    lozinka = loz;
  }
  public String toString() {
    return "informacije o prijavljivanju: \n   " + "korisničko ime: " + kori-
snickoIme +
      "\n   datum: " + datum + "\n   lozinka: " + loz;
  }
  public static void main(String[] args)  throws Exception {
    Prijavljivanje a = new Prijavljivanje("Hulk", "mojMaliPoni");
    print( "prijavljivanje a = " + a);
      ObjectOutputStream o = new ObjectOutputStream(
          new FileOutputStream("Prijavljivanje.out"));
    o.writeObject(a);
    o.close();
    TimeUnit.SECONDS.sleep(1);  // Kašnjenje
    // Sada ih vraćamo:
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Prijavljivanje.out"));
    print("Rekonstruisanje objekta " + new Date());
    a = (Prijavljivanje)in.readObject();
    print( "prijavljivanje a = " + a);
  }
} /* Ispis: (primer)
prijavljivanje a = informacije o prijavljivanju
   korisnicko ime: Hulk
   datum: Sat Nov 19 15:03:26 MST 2005
   lozinka: mojMaliPoni
Rekonstruisanje objekta Sat Nov 19 15:03:26 MST 2005
prijavljivanje a = informacije o prijavljivanju
   korisnicko ime: Hulk
   datum: Sat Nov 19 15:03:26 MST 2005
   lozinka: null
* ///:~
```

Vidi se da su polja **datum** i **korisnickoIme** obična (a ne **transient**) i zato se automatski serijalizuju. Međutim, polje **lozinka** je označeno kao **transient** i zato se ne snima na disk, niti mehanizam serijalizovanja pokušava da ga rekonstruiše. Kada se objekat rekonstruiše, polje **lozinka** ima vrednost **null**. Obratite pažnju na to da dok **toString**( ) sastavlja **String** objekat preklopljenim operatorom +, referenca jednaka **null** automatski se konvertuje u znakovni niz null.

Možda ste uočili i to da je polje **datum** snimljeno i rekonstruisano s diska, odnosno da nije pravljeno ponovo.

Pošto objekti interfejsa **Externalizable** podrazumevano ne čuvaju nijedno polje, rezervisana reč **transient** koristi se isključivo za objekte interfejsa **Serializable**.

Alternativa za interfejs Externalizable

Ako vam se primena interfejsa **Externalizable** iz nekog razloga ne sviđa, isprobajte drugačiji pristup. Možete da realizujete interfejs **Serializable** i da mu *dodate* (obratite pažnju na to da sam rekao „dodate", a ne „redefinišete" ili „implementirate") metode **writeObject( )** i **readObject( )** koje će se automatski pozivati prilikom serijalizovanja i deserijalizovanja objekta. To jest, ako obezbedite te dve metode, one će se koristiti umesto podrazumevanih metoda za serijalizaciju.

Metode moraju da imaju baš ovakve potpise:

```
private void  writeObject(ObjectOutputStream tok)
    throws IOException;

private void  readObject(ObjectInputStream tok)
    throws IOException, ClassNotFoundException
```

S tačke gledišta programiranja, ovo je prilično čudno. Prvo možete pomisliti da te metode, pošto nisu deo osnovne klase ili interfejsa **Serializable**, treba da budu definisane u sopstvenim interfejsima. Obratite pažnju na to da su one definisane kao privatne, što znači da treba da ih pozivaju samo drugi članovi iste klase. Međutim, njih ne pozivaju drugi članovi klase, već metode **writeObject( )** i **readObject( )** klasa **ObjectOutputStream** i **ObjectInputStream**. (Izuzetno se trudim da se ne upuštam u iscrpljujuću raspravu o istim imenima metoda. Objašnjenje: samo bih vas zbunio.) Možda se pitate kako objekti klasa **ObjectOutputStream** i **ObjectInputStream** imaju pistup privatnim metodama vaše klase. Možemo samo pretpostaviti da je to deo magičnog postupka serijalizovanja.[6]

U svakom slučaju, sve što se definiše u interfejsu podrazumevano je javno, pa ako metode **writeObject( )** i **readObject( )** moraju da budu privatne, one ne mogu biti deo interfejsa. Pošto morate tačno da se pridržavate potpisa, učinak je isti kao da realizujete interfejs.

Izgleda kao da se posle poziva metode **ObjectOutputStream.writeObject( )** objekat interfejsa **Serializable** koji joj prosleđujete ispituje (bez sumnje, korišćenjem refleksije) kako bi se ustanovilo da li realizuje sopstvenu metodu **writeObject( )**. Ako je tako, preskače se uobičajeni postupak serijalizacije i poziva namenska metoda **writeObject( )**. Isto važi i za metodu **readObject( )**.

Postoji još jedna začkoljica. Unutar metode **writeObject( )** koju pišete možete da izvedete podrazumevanu akciju upisivanja objekta pozivom metode **defaultWriteObject( )**. Slično, unutar metode **readObject( )** možete da pozovete metodu **defaultReadObject( )**. Evo jednostavnog primera koji pokazuje kako možete da upravljate čuvanjem i rekonstruisanjem objekta koji realizuje interfejs **Serializable**:

```
//: ui/KontrolaSerijalizovanja.java
// Upravljanje serijalizovanjem dodavanjem sopstvenih
// metoda writeObject() i readObject().
import java.io.*;

public class KontrolaSerijalizovanja implements Serializable {
```

[6] U odeljku „Interfejsi i podaci o tipu" na kraju poglavlja *Podaci o tipu* pokazano je kako je moguće pristupiti privatnim metodama kada je pozivalac izvan njihove klase.

```
  private String a;
  private transient String b;
  public KontrolaSerijalizovanja(String aa, String bb) {
    a = "Nije transient: " + aa;
    b = "transient: " + bb;
  }
  public String toString() { return a + "\n" + b; }
  private void writeObject(ObjectOutputStream tok)
      throws IOException {
    tok.defaultWriteObject();
    tok.writeObject(b);
  }
  private void readObject(ObjectInputStream tok)
      throws IOException, ClassNotFoundException {
    tok.defaultReadObject();
    b = (String)tok.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    KontrolaSerijalizovanja sc = new KontrolaSerijalizovanja("Test1",
       "Test2");
    System.out.println("Pre:\n" + sc);
    ByteArrayOutputStream buf = new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    o.writeObject(sc);
    // Sada ga vraćamo:
    ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(buf.toByteArray()));
    KontrolaSerijalizovanja sc2 = (KontrolaSerijalizovanja)
       in.readObject();
    System.out.println("Posle:\n" + sc2);
  }
} /* Ispis:
Pre:
Nije transient: Test1
transient: Test2
Posle:
Nije transient: Test1
transient: Test2
*///:~
```

U ovom primeru jedno polje tipa **String** je obično, a drugo je označeno kao **transient** kako bi se dokazalo da metoda **defaultWriteObject( )** snima i rekonstruiše polje koje nije **transient**, a da se **transient** polje snima i rekonstruiše eksplicitno. Polja se inicijalizuju u konstruktoru, a ne prilikom definisanja, kako bi se dokazalo da se ne inicijalizuju nekim automatskim mehanizmom tokom deserijalizovanja.

Ako nameravate da koristite podrazumevani mehanizam za upis delova objekta koji nisu označeni sa **transient**, morate da pozovete metodu **defaultWriteObject( )** kao prvu operaciju u metodi **writeObject( )**, odnosno metodu **defaultReadObject( )** kao prvu operaciju u metodi **readObject( )**. To su čudni pozivi metoda. Izgledalo bi, na primer, kao da

pozivate metodu **defaultWriteObject( )** za objekat klase **ObjectOutputStream** i ne prosleđujete nikakve argumente, ali se on ipak nekako snalazi – dobija referencu na vaš objekat i zna kako da upiše sve delove koji nisu označeni sa **transient**. Da čoveka podiđu žmarci.

Za čuvanje i rekonstruisanje polja označenih sa **transient** koristi se razumljiviji kôd. Ipak, razmislite o tome šta se dešava. U funkciji **main( )** pravi se objekat **KontrolaSerijalizovanja** koji se potom serijalizuje u tok tipa **ObjectOutputStream**. (Obratite pažnju na to da se u ovom slučaju umesto datoteke koristi memorijski blok, kao i za **ObjectOutputStream**.) Serijalizacija se dešava u redu:

```
o.writeObject(sc);
```

Metoda **writeObject( )** mora da ispituje objekat **sc** da bi ustanovila ima li sopstvenu metodu **writeObject( )**. (Ne proveravanjem interfejsa, jer on ne postoji, niti tipa klase, već stvarnim traženjem metode pomoću refleksije.) Ako pronađe tu metodu, iskoristiće je. Sličan pristup se koristi i u metodi **readObject( )**. Možda je ovo bio jedini način za rešavanje problema, ali je ipak čudan.

Zadavanje verzije

Možda ćete hteti da promenite verziju klase koja se može serijalizovati (na primer, objekti originalne klase mogu biti čuvani u bazi podataka). To je podržano u Javi, ali ćete to raditi samo u specijalnim slučajevima, a pri tom morate bolje razumeti problem, što ovde nećemo razmatrati. HTML dokumentacija razvojnog paketa za Javu koja se može preuzeti s lokacije *http://java.sun.com*, detaljno obrađuje ovu temu.

U ovoj dokumentaciji uočićete mnoge komentare koji počinju sa:

***Upozorenje**: Serijalizovani objekti ove klase neće biti kompatibilni s budućim verzijama Swinga. Trenutna podrška serijalizaciji pogodna je za kratkotrajno čuvanje metoda koje se pozivaju daljinski u raznim aplikacijama...*

Ovo upozorenje stoji zato što je mehanizam dodeljivanja verzije previše jednostavan da bi pouzdano radio u svim situacijama, naročito sa zrnima Jave. Radi se na ispravci projekta, o čemu upravo i govori ovo upozorenje.

Korišćenje trajnosti

Tehnika serijalizovanja je prilično privlačna za čuvanje stanja nekog programa da bi se on kasnije mogao lako vratiti nazad. Međutim, pre toga, mora se odgovoriti na neka pitanja. Šta će se desiti ako se serijalizuju dva objekta, a svaki od njih sadrži referencu na treći objekat? Kada rekonstruišete ta dva objekta iz serijalizovanog stanja, da li će se treći objekat pojaviti samo jednom? Šta će se desiti ako se dva objekta serijalizuju u zasebne datoteke, a potom deserijalizuju u različitim delovima koda?

Evo primera koji prikazuje ovaj problem:

```
//: ui/MojSvet.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
class Kucica implements Serializable {}
class Zivotinja implements Serializable {
  private String ime;
  private Kucica omiljenaKucica;
  Zivotinja(String im, Kucica k) {
    ime = im;
    omiljenaKucica = k;
  }
  public String toString() {
    return ime + "[" + super.toString() +
      "], " + omiljenaKucica + "\n";
  }
}

public class MojSvet {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    Kucica kucica = new Kucica();
    List<Zivotinja> zivotinje = new ArrayList<Zivotinja>();
    zivotinje.add(new Zivotinja("pas Lesi", kucica));
    zivotinje.add(new Zivotinja("mrmot Bobi", kucica));
    zivotinje.add(new Zivotinja("mačka Cuca", kucica));
    print("životinje: " + zivotinje);
    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 = new ObjectOutputStream(buf1);
    o1.writeObject(zivotinje);
    o1.writeObject(zivotinje); // Upisivanje drugog skupa
    // Upisivanje u drugačiji tok:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 = new ObjectOutputStream(buf2);
    o2.writeObject(zivotinje);
    // Sada ih vraćamo:
    ObjectInputStream in1 = new ObjectInputStream(
        new ByteArrayInputStream(buf1.toByteArray()));
    ObjectInputStream in2 = new ObjectInputStream(
        new ByteArrayInputStream(buf2.toByteArray()));
    List
       zivotinje1 = (List)in1.readObject(),
       zivotinje2 = (List)in1.readObject(),
       zivotinje3 = (List)in2.readObject();
    print("životinje1: " + zivotinje1);
    print("životinje2: " + zivotinje2);
    print("životinje3: " + zivotinje3);
  }
} /* Ispis: (primer)
životinje: [pas Lesi[Zivotinja@1cc76c], Kucica@1cc769
, mrmot Bobi[Zivotinja@1cc76d], Kucica@1cc769
, mačka Cuca[Zivotinja@1cc76e], Kucica@1cc769
]
```

```
životinje1: [pas Lesi[Zivotinje@1cca0c], Kucica@1cca16
, mrmot Bobi[Zivotinja@1cca17], Kucica@1cca16
, mačka Cuca[Zivotinja@1cca1b], Kucica@1cca16
]
životinje2: [pas Lesi[Zivotinje@1cca0c], Kucica@1cca16
, mrmot Bobi[Zivotinja@1cca17], Kucica@1cca16
, mačka Cuca[Zivotinja@1cca1b], Kucica@1cca16
]
životinje3: [pas Lesi[Zivotinje@1cca52], Kucica@1cca5c
, mrmot Bobi[Zivotinja@1cca5d], Kucica@1cca5c
, mačka Cuca[Zivotinja@1cca61], Kucica@1cca5c
]
*///:~
```

Ovde je zanimljivo to da serijalizaciju objekata možete koristiti uz niz bajtova, kao način za temeljno kopiranje svih objekata koji se mogu serijalizovati. (Temeljno kopiranje znači da se kopira čitava mreža objekata, a ne samo osnovni objekat i njegove reference.) Kopiranje objekata je detaljno obrađeno u mrežnim dodacima ove knjige.

Objekti klase **Zivotinja** sadrže polja tipa **Kucica**. U funkciji **main**( ), pravi se lista tih životinja i dvaput se serijalizuje u jedan, a zatim i u drugi tok. Kada se ti tokovi deserijalizuju i ispišu, videćete rezultate prikazane za jedno izvršavanje (objekti će pri svakom pokretanju biti na različitim lokacijama u memoriji).

Naravno, očekuje se da deserijalizovani objekti imaju drugačije adrese od originalnih. Međutim, u objektima **zivotinje1** i **zivotinje2** pojavljuju se iste adrese, uključujući i reference na objekat klase **Kucica** koji oba objekta sadrže. S druge strane, prilikom rekonstruisanja objekta **zivotinje3**, sistem ne zna da su objekti u drugom toku isti kao objekti u prvom toku, pa pravi potpuno različitu mrežu objekata.

Kad serijalizujete sve objekte u jedan tok, moći ćete da rekonstruišete mrežu objekata koju ste napisali, bez slučajnog dupliranja objekata. Naravno, možete da promenite stanje objekata u periodu između pisanja prvog i poslednjeg, ali za to sami odgovarate: u trenutku kada ih serijalizujete, objekti će biti upisani u stanju u kome se nalaze (i s postojećim vezama do drugih objekata).

Ukoliko želite da zabeležite stanje sistema, najbezbednije je da se to obavi kao „nedeljiva" operacija. Ako serijalizujete nešto, pa počnete da radite nešto drugo, zatim opet serijalizujete itd., nećete moći bezbedno da zabeležite stanje sistema. Umesto toga, smestite sve objekte koji odslikavaju stanje sistema u isti kontejner koji ćete snimiti u jednoj operaciji. Na taj način moći ćete da ga rekonstruišete pozivom samo jedne metode.

Sledeći primer je sistem za projektovanje pomoću računara (CAD) koji ilustruje ovaj princip. Pored toga, primer obrađuje i statička polja: u dokumentaciji ćete videti da se objekat klase **Class** može serijalizovati, pa bi statička polja trebalo lako da se čuvaju tako što bi se jednostavno serijalizovali objekti te klase. U svakom slučaju, to zasad *izgleda* kao pametan pristup.

```
//: ui/PamcenjeCADStanja.java
// Pamćenje stanja nazovi-CAD sistema.
import java.io.*;
import java.util.*;
```

```
abstract class Oblik implements Serializable {
  public static final int CRVENA = 1, PLAVA = 2, ZELENA = 3;
  private int xPos, yPos, dimenzija;
  private static Random rand = new Random(47);
  private static int brojac = 0;
  public abstract void postaviBoju(int novaBoja);
  public abstract int citajBoju();
  public Oblik(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimenzija = dim;
  }
  public String toString() {
    return getClass() +
      " boja[" + citajBoju() + "] xPos[" + xPos +
      "] yPos[" + yPos + "] dim[" + dimenzija + "]\n";
  }
  public static Oblik slucajnoGenerisanje() {
    int xVal = rand.nextInt(100);
    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(brojac++ % 3) {
      default:
      case 0: return new Krug(xVal, yVal, dim);
      case 1: return new Kvadrat(xVal, yVal, dim);
      case 2: return new Linija(xVal, yVal, dim);
    }
  }
}

class Krug extends Oblik {
  private static int boja = CRVENA;
  public Krug(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void postaviBoju(int novaBoja) { boja = novaBoja; }
  public int citajBoju() { return boja;  }
}

class Kvadrat extends Oblik {
  private static int boja;
  public Kvadrat(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    boja = CRVENA;
  }
  public void postaviBoju(int novaBoja) { boja = novaBoja; }
  public int citajBoju() { return boja; }
}

class Linija extends Oblik {
```

```
  private static int boja = CRVENA;
  public static void
  serijalizujStaticke(ObjectOutputStream os)
      throws IOException { os.writeInt(boja); }
  public static void
  deserijalizujStaticke(ObjectInputStream os)
      throws IOException { boja = os.readInt(); }
  public Linija(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void postaviBoju(int novaBoja) { boja = novaBoja; }
  public int citajBoju() { return boja; }
}

public class PamcenjeCADStanja {
  public static void main(String[] args) throws Exception {
    List<Class<? extends Oblik>> tipoviOblika =
      new ArrayList<Class<? extends Oblik>>( );
    // Dodavanje referenci na objekte klase:
    tipoviOblika.add(Krug.class);
    tipoviOblika.add(Kvadrat.class);
    tipoviOblika.add(Linija.class);
    List<Oblik> oblici = new ArrayList<Oblik>( );
    // Pravljenje nekih oblika:
    for(int i = 0; i < 10; i++)
        oblici.add(Oblik.slucajnoGenerisanje());
    // Postavljanje svih statičkih boja na ZELENA:
    for(int i = 0; i < 10; i++)
        ((Oblik)oblici.get(i)).postaviBoju(Oblik.ZELENA);
    // Snimanje vektora stanja:
    ObjectOutputStream out = new ObjectOutputStream(
          new FileOutputStream("CADStanje.out"));
    out.writeObject(tipoviOblika);
    Linija.serijalizujStaticke (out);
    out.writeObject(oblici);
    // Prikazivanje oblika:
    System.out.println(oblici);
  }
} /* Ispis:
[class Krug boja xPos yPos dim
, class Kvadrat boja, xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
, class Kvadrat boja xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
, class Kvadrat boja xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
]
///:~
```

Klasa **Oblik** realizuje interfejs **Serializable**, pa se sve što se izvodi iz klase **Oblik** automatski može serijalizovati. Svaki objekat klase **Oblik** sadrži podatke, a svaka klasa izvedena iz klase **Oblik** sadrži statičko polje koje određuje boju svih figura izvedenog tipa. (Stavljanje statičkog polja u osnovnu klasu dalo bi samo jedno polje, pošto se statička polja ne dupliraju u izvedenim klasama.) Metode osnovne klase se mogu redefinisati da bi se zadala boja različitih tipova oblika (za statičke metode se ne primenjuje dinamičko povezivanje, tj. radi se o običnim metodama.) Metoda **slucajnoGenerisanje( )**, kad god se pozove, pravi različit objekat tipa **Oblik** tako što nasumično bira tip oblika.

Klase **Krug** i **Kvadrat** jednostavna su proširenja klase **Oblik**; razlikuju se samo po tome što se u klasi **Krug** boja inicijalizuje tokom definisanja, a u klasi **Kvadrat** – u konstruktoru. Odložićemo razmatranje klase **Linija**.

U funkciji **main( )** koristi se jedna lista tipa **ArrayList** za čuvanje objekata tipa **Class** i još jedna za čuvanje oblika.

Rekonstrukcija objekata je prilično jednostavna:

```
//: ui/RekonstruisanjeCADStanja.java
// Rekonstruisanje stanja nazovi-CAD sistema.
//  {RunFirst: PamcenjeCADStanja}
import java.io.*;
import java.util.*;

public class RekonstruisanjeCADStanja {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("CADStanje.out"));
    // Čitanje po redosledu upisivanja:
    List<Class<? extends Oblik>> tipoviOblika =
      (List<Class<? extends Oblik>>)in.readObject();
    Line.deserijalizujStaticke(in);
    List<Oblik> oblici = (List<Oblik>)in.readObject();
    System.out.println(oblici);
  }
} /* Ispis:
[class Krug boja xPos yPos dim
, class Kvadrat boja, xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
, class Kvadrat boja xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
, class Kvadrat boja xPos yPos dim
, class Linija boja xPos yPos dim
, class Krug boja xPos yPos dim
]
*///:~
```

Vidi se da se vrednosti **xPos**, **yPos** i **dim** uspešno snimaju i rekonstruišu, ali nešto nije u redu sa rekonstruisanjem statičkih informacija. Sve tri ulaze, ali ne izlaze nepromenjene. Krugovi imaju vrednost 1 (**CRVENA**, po definiciji), a kvadrati imaju vrednost 0 (setite se da su inicijalizovani u konstruktoru). Izgleda kao da se statička polja uopšte nisu serijalizovala! To je tačno: iako klasa **Class** može da se serijalizuje, ona ne radi ono što se očekuje. Znači, ako želite da serijalizujete statičke podatke, to morate da uradite sami.

U tu svrhu služe statičke metode **serijalizujStaticke( )** i **deserijalizujStaticke( )** klase **Linija**. Vidite da se one eksplicitno pozivaju u postupku snimanja i rekonstruisanja. (Obratite pažnju na to da mora biti održan redosled upisivanja i čitanja iz datoteke za serijalizaciju.) Dakle, da bi ovi programi radili ispravno, morate da:

1. Dodate metode **serijalizujStaticke( )** i **deserijalizujStaticke( )** u sve klase oblika.
2. Uklonite listu **tipoviOblika** i kôd koji je povezan s tom listom.
3. U oblike dodate pozive novih statičkih metoda za serijalizovanje i deserijalizovanje.

Bezbednost je još jedna tema o kojoj bi valjalo razmišljati, pošto se serijalizovanjem čuvaju i privatni podaci. Ako imate problem s bezbednošću, ta polja treba da označite kao **transient**. Međutim, tada morate da smislite bezbedan način za čuvanje poverljivih informacija tako da prilikom rekonstruisanja možete da vratite privatne promenljive u prvobitno stanje.

Vežba 30: (1) Popravite program **CADStanje.java** kako je opisano u tekstu.

XML

Važno ograničenje serijalizovanja objekata jeste njihova isključiva upućenost na Javu: takve objekte mogu da deserijalizuju samo Java programi. Konverzijom podataka u format XML dobilo bi se interoperabilnije rešenje koje bi mogle da koriste razne platforme i jezici.

Zbog raširenosti XML-a postoji zbunjujuće velik broj opcija za programiranje koje ga proizvode; među njima su i biblioteke **javax.xml.*** koje se isporučuju uz razvojni paket Jave. Odlučio sam da upotrebim biblioteku otvorenog izvornog koda XOM (preuzimanje i dokumentacija na *www.xom.nu*) – autor je Elliotte Rusty Harold. Izgleda mi da je to najjednostavniji način da se XML proizvodi i menja u Javi. Pored toga, XOM proizvodi veoma ispravan XML kôd.

Primera radi, pretpostavimo da imate objekte tipa **Osoba** i u njima imena i prezimena koja želite da serijalizujete u formatu XML. Naredna klasa **Osoba** ima metodu **getXML( )** koja pomoću XOM-a proizvodi podatke o klasi **Osoba** konvertovane u XML **Element**, i konstruktor koji prima **Element** i vadi odgovarajuće podatke o klasi **Osoba** (obratite pažnju na to da su XML primeri u sopstvenom poddirektorijumu):

```
//: xml/Osoba.java
// Upotreba biblioteke XOM za pisanje i čitanje XML-a
// {Zahteva: nu.xom.Node; Morate instalirati
// biblioteku XOM sa http://www.xom.nu }
```

```
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Osoba {
  private String im, prez;
  public Osoba(String im, String prez) {
    this.im = im;
    this.prez = prez;
  }
  // Napravi XML Element od ovog objekta Osoba:
  public Element getXML() {
    Element osoba = new Element("osoba");
    Element ime = new Element("im");
    ime.appendChild(im);
    Element prezime = new Element("prez");
    prezime.appendChild(prez);
    osoba.appendChild(ime);
    osoba.appendChild(prezime);
    return osoba;
  }
  // Konstruktor za rekonstrukciju Osobe iz XML Elementa:
  public Osoba(Element osoba) {
    im= osoba.getFirstChildElement("im").getValue();
    prez = osoba.getFirstChildElement("prez").getValue();
  }
  public String toString() { return im + " " + prez; }
  // Neka bude čitljivo za ljude:
  public static void
  format(OutputStream os, Document doc) throws Exception {
    Serializer serializer= new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
  }
  public static void main(String[] args) throws Exception {
    List<Osoba> ljudi = Arrays.asList(
      new Osoba("Dr. Bunsen", "Honeydew"),
      new Osoba("Gonzo", "The Great"),
      new Osoba("Phillip J.", "Fry"));
    System.out.println(ljudi);
    Element koren = new Element("ljudi");
    for(Osoba p : ljudi)
      koren.appendChild(p.getXML());
    Document doc = new Document(koren);
    format(System.out, doc);
    format(new BufferedOutputStream(new FileOutputStream(
      "Ljudi.xml")), doc);
  }
```

```
} /* Ispis:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<ljudi>
    <osoba>
        <im>Dr. Bunsen</im>
        <prez>Honeydew</prez>
    </osoba>
    <osoba>
        <im>Gonzo</im>
        <prez>The Great</prez>
    </osoba>
    <osoba>
        <im>Phillip J.</im>
        <prez>Fry</prez>
    </osoba>
</ljudi>
*///:~
```

XOM metode su jasne same po sebi, a mogu se naći i u XOM dokumentaciji.

XOM sadrži i klasu **Serializer** koja je u metodi **format**( ) upotrebljena za pretvaranje XML-a u čitljiviji oblik. Ukoliko samo pozovete **toXML**( ), dobićete sve bačeno na gomilu, pa je **Serializer** podesna alatka.

Jednostavno je i deserijalizovanje objekata tipa **Osoba** iz XML datoteka:

```
//: xml/LJudi.java
// {Zahteva: nu.xom.Node; Morate instalirati
// biblioteku XOM sa http://www.xom.nu }
// {RunFirst: Osoba}
import nu.xom.*;
import java.util.*;

public class Ljudi extends ArrayList<Osoba> {
  public Ljudi(String imeDatoteke) throws Exception  {
    Document doc = new Builder().build(imeDatoteke);
    Elements elementi =
      doc.getRootElement().getChildElements();
    for(int i = 0; i < elementi.size(); i++)
      add(new Osoba(elementi.get(i)));
  }
  public static void main(String[] args) throws Exception {
    Ljudi p = new Ljudi("Ljudi.xml");
    System.out.println(p);
  }
} /* Ispis:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*///:~
```

Konstruktor klase **Ljudi** otvara i čita datoteku XOM-ovom metodom **Builder.build**( ), a metoda **getChildElements**( ) proizvodi listu **Elements** (to nije standardna Javina lista, nego objekat koji ima samo metode **size**( ) i **get**( ) – Harold nije želeo da natera korisnike

da upotrebljavaju Javu SE5, ali je ipak želeo da ima kontejner koji omogućuje bezbedan rad s tipovima). Svaki **Element** u toj listi predstavlja objekat tipa **Osoba**, pa se predaje drugom konstruktoru klase **Osoba**. Vodite računa o tome da morate unapred znati strukturu svoje XML datoteke, ali baš to je često slučaj u ovoj vrsti problema. Ako ta struktura ne odgovara onome što očekujete, XOM će generisati izuzetak. Svakako da možete pisati i složeniji kôd koji će ispitati XML dokument umesto da o njemu išta pretpostavlja, u slučajevima kada imate manje konkretne informacije o dolaznoj XML strukturi.

Da bi ovi primeri mogli da se prevedu, moraćete da stavite JAR datoteke iz XOM distribucije u svoju putanju klasa.

Ovo je bio samo kratak uvod u XML programiranje na Javi i s bibliotekom XOM; više informacija potražite na adresi *www.xom.nu*.

Vežba 31: (2) Programima **Osoba.java** i **Ljudi.java** dodajte odgovarajuće podatke o adresi.

Vežba 32: (4) Koristeći **Map<String,Integer>** i uslužnu klasu **net.mindview. util.TextFile**, napišite program koji prebrojava reči u datoteci (kao drugi argument u konstruktoru klase **TextFile** upotrebite **"\\W+"**). Rezultate sačuvajte u obliku XML datoteke.

Preferences

Interfejs za programiranje aplikacija (API) *Preferences* mnogo je bliži konceptu trajnosti nego serijalizovanju objekata, zato što informacije automatski skladišti i vadi iz skladišta. Međutim, može se primenjivati samo na određene male skupove podataka – na proste tipove i znakovne nizove (objekte tipa **String**), a svaki uskladišten znakovni niz ne sme biti duži od 8 K (nije baš da je sićušno, ali nije za ozbiljne poslove). Kao što mu ime govori, API Preferences služi za skladištenje i vađenje parametara koje je izabrao korisnik i postavki konfiguracije programa.

Poželjni parametri i postavke smeštaju se u skupove ključ – vrednost (kao mape), uskladištene u hijerarhiji čvorova. Iako se u hijerarhiji čvorova mogu napraviti složene strukture, obično se pravi samo jedan čvor nazvan po klasi o kojoj se informacije skladište u njemu. Evo jednostavnog primera:

```
//: ui/PrimerZaPreferences.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PrimerZaPreferences {
  public static void main(String[] args) throws Exception {
    Preferences prefs = Preferences
      .userNodeForPackage(PrimerZaPreferences.class);
    prefs.put("Mesto", "Oz");
    prefs.put("Obuća", "Crvene papučice");
    prefs.putInt("Drugara", 4);
    prefs.putBoolean("Ima li veštica?", true);
    int brojUpotreba = prefs.getInt("UsageCount", 0);
    brojUpotreba++;
    prefs.putInt("UsageCount", brojUpotreba);
    for(String kljuc : prefs.keys())
      print(kljuc + ": "+ prefs.get(kljuc, null));
```

```
    // Uvek morate zadati podrazumevanu vrednost:
    print("Koliko drugara ima Doroti? " +
      prefs.getInt("Drugara", 0));
  }
} /* Ispis: (primer)
Mesto: Oz
Obuća: Crvene papučice
Drugara: 4
Ima li veštica?: true
UsageCount: 53
Koliko drugara ima Doroti? 4
*///:~
```

Ovde je upotrebljena metoda **userNodeForPackage( )**, a mogao sam odabrati i **systemNodeForPackage( )**; izbor je donekle proizvoljan, ali ideja je da se „user" upotrebljava za poželjne parametre i postavke pojedinačnih korisnika, a „system" za konfiguraciju cele instalacije. Pošto je metoda **main( )** statična, za identifikaciju čvora upotrebljena je klasa **PrimerZaPreferences.class**, ali unutar nestatične metode obično se koristi **getClass( )**. Kao identifikator čvora ne morate da koristite tekuću klasu, ali je to uobičajena praksa.

Nakon što napravite čvor, on vam je na raspolaganju za upisivanje ili čitanje podataka. U prethodnom primeru u čvor su učitane različite vrste stavki, a zatim je pozvana metoda **keys( )**. Njen rezultat su ključevi (engl. *keys*) u obliku znakovnog niza (**String[]**), što poznavaoci istoimene metode iz biblioteke kolekcija ne bi očekivali. Obratite pažnju na drugi argument metode **get( )**. To je podrazumevana vrednost koja se vraća kao rezultat ukoliko se vrednost za dati ključ ne pronađe. Tokom iteracija kroz skup ključeva uvek znate da neka stavka postoji, pa je korišćenje **null** kao podrazumevane vrednosti bezbedno. Uobičajeno biste pribavljali neki imenovani ključ, kao u:

```
prefs.getInt("Drugara", 0));
```

U normalnom slučaju treba da zadate razumnu podrazumevanu vrednost. Zapravo, ovo je jedan od tipičnih idioma:

```
int brojUpotreba = prefs.getInt("UsageCount", 0);
brojUpotreba++;
prefs.putInt("UsageCount", brojUpotreba);
```

Na taj način, **UsageCount** će biti nula kada program pokrenete prvi put, ali će u kasnijim pokretanjima biti različit od nule.

Kada pokrenete **PrimerZaPreferences.java**, videćete da se **UsageCount** zaista povećava za jedan svaki put kada pokrenete program. Ali gde se taj broj čuva? Nema lokalne datoteke koja bi se pojavila nakon prvog pokretanja programa. API Preferences za obavljanje tog svog posla koristi odgovarajuće sistemske resurse, a oni se menjaju u zavisnosti od operativnog sistema. U Windowsu se za to upotrebljava registar (pošto je on ionako hijerarhija čvorova s parovima ključ – vrednost). Poenta je sledeća: informacije se automatski skladište, a mi ne moramo da brinemo o tome kako se to obavlja u bilo kojem pojedinačnom sistemu.

API Preferences ume više nego što je ovde prikazano. Dalje pojedinosti o tome potražite u dokumentaciji razvojnog paketa JDK, koja je prilično razumljiva.

Vežba 33: (2) Napišite program koji prikazuje tekuću vrednost direktorijuma nazvanog „osnovni direktorijum" i traži od vas njegovu novu vrednost. Za skladištenje te vrednosti upotrebite API Preferences.

Sažetak

Javina biblioteka U/I tokova zadovoljava osnovne zahteve: omogućuje čitanje sa konzole i upisivanje u nju, u datoteke, u memorijske blokove, čak i preko Interneta. Pomoću nasleđivanja možete da napravite nove vrste ulaznih i izlaznih objekata. Možete čak i da izmenite način na koji tokovi rade sa objektima tako što ćete redefinisati metodu **toString( )** koja se automatski poziva kada objekat prosledite metodi čiji je argument tipa **String** (Javina ograničena automatska konverzija tipa).

U dokumentaciji i projektu biblioteke U/I tokova, neka pitanja su ostala nedorečena. Na primer, bilo bi lepo kada biste mogli da generišete izuzetak pri pokušaju pisanja preko postojeće datoteke tokom njenog otvaranja; neki programerski sistemi omogućuju da zadate otvaranje izlazne datoteke, ali samo ako ona već ne postoji. Izgleda da u Javi treba koristiti objekat klase **File** kako bi se odredilo da li datoteka postoji, zato što će se ta datoteka uvek menjati ako je otvarate kao **FileOutputStream** ili **FileWriter**.

Biblioteka U/I tokova izaziva oprečna osećanja; ipak, radi dosta toga i prenosiva je. Ukoliko ne razumete projektni obrazac *Decorator*, njen projekat nije intuitivan pa je potreban dodatni napor da se ona objasni i razume. Istovremeno, biblioteka nije zaokružena: na primer, ne bi trebalo da ja moram da pišem uslužne klase kao što je **TextFile** (nova (SE5) Javina klasa **PrintWriter** predstavlja korak u pravom smeru, ali to je tek delimično rešenje). Java SE5 je donela veliko poboljšanje: najzad je dodato formatiranje izlaza kakvo oduvek podržavaju gotovo svi ostali jezici.

Kada *shvatite* dekoratorski projektni obrazac i počnete da upotrebljavate ovu biblioteku tamo gde je potrebna njena prilagodljivost, uvidećete da je korisna i tada vam nekoliko dodatnih redova koda više neće predstavljati problem.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na Web lokaciji *www.MindView.net*.

Nabrojani tipovi

Rezervisana reč ***enum*** *služi za pravljenje novog tipa od ograničenog skupa imenovanih vrednosti; zbog nje se te vrednosti tretiraju kao punopravne komponente programa. Ispostavlja se da je to veoma korisno.*[1]

NABROJANE TIPOVE UKRATKO SMO PREDSTAVILI NA KRAJU POGLAVLJA *INICIJALIZACIJA I ČIŠĆENJE*. Međutim, pošto sada razumete i neke komplikovanije oblasti u Javi, možemo detaljnije da razmotrimo nabrojane tipove (engl. *enumerated types*) u Javi SE5. Videćete da s nabrojanim tipovima može da se uradi i ponešto zanimljivo, ali ovo poglavlje bi trebalo da vam pruži bolji uvid i u druge mogućnosti jezika koje ste dosad upoznali, kao što su generički tipovi i refleksija. Naučićete i još nekoliko projektnih obrazaca.

Osnovne mogućnosti nabrojanih tipova

Kao što je bilo rečeno u poglavlju *Inicijalizacija i čišćenje*, kroz listu **enum** konstanti možete iterirati metodom **values**( ) tog nabrojanog tipa. Metoda **values**( ) proizvodi niz **enum** konstanti u poretku njihovog deklarisanja; dobijeni niz možete (na primer) upotrebiti u petlji foreach.

Kada napravite nabrojan tip, prevodilac proizvodi njemu pridruženu klasu. Ona automatski nasleđuje klasu **java.lang.Enum**, što joj daje određene mogućnosti koje ću prikazati u narednom primeru:

```
//: nabrojani/KlasaEnum.java
// Mogućnosti klase Enum
import static net.mindview.util.Print.*;

enum Odmor { MORE, PLANINA, BANJA }

public class KlasaEnum {
  public static void main(String[] args) {
    for(Odmor o : Odmor.values()) {
      print(o + " rednibroj: " + o.ordinal());
      printnb(o.compareTo(Odmor.PLANINA) + " ");
      printnb(o.equals(Odmor.PLANINA) + " ");
      print(o == Odmor.PLANINA);
      print(o.getDeclaringClass());
      print(o.name());
      print("----------------------");
    }
    // Pravi nabrojani tip od znakovnog niza:
    for(String o : "BANJA PLANINA MORE".split(" ")) {
      Odmor odm = Enum.valueOf(Odmor.class, o);
      print(odm);
    }
```

[1] Pri pisanju ovog poglavlja mnogo mi je pomogao Joshua Bloch.

```
  }
} /* Ispis:
MORE rednibroj: 0
-1 false false
class Odmor
MORE
----------------------
PLANINA rednibroj: 1
0 true true
class Odmor
PLANINA
----------------------
BANJA rednibroj: 2
1 false false
class Odmor
BANJA
----------------------
BANJA
PLANINA
MORE
*///:~
```

Metoda **ordinal( )** proizvodi ceo broj koji pokazuje redosled deklarisanja svake **enum** instance, počev od nule. **enum** instance uvek možete bezbedno porediti operatorom ==, i metode **equals( )** i **hashCode( )** bivaju automatski napravljene. Klasa **Enum** realizuje interfejs **Comparable**, stoga ima svoju metodu **compareTo( )**, a realizuje i interfejs **Serializable**.

Pozivom metode **getDeclaringClass( )** za **enum** instancu, dobićete obuhvatajuću **enum** klasu.

Metoda **name( )** daje ime tačno onako kako je bilo deklarisano, a isto to dobijate i metodom **toString( )**. **valueOf( )** je statični član klase **Enum** i proizvodi **enum** instancu koja odgovara znakovnom nizu koji joj je prosleđen ili baca izuzetak ukoliko takav niz ne može da pronađe.

Uvoz statičnih članova u nabrojani tip

Evo jedne varijacije programa **Pljeskavica.java** iz poglavlja *Inicijalizacija i čišćenje*:

```
//: nabrojani/LJutina.java
package nabrojani;

public enum Ljutina {
  NE, BLAGO, SREDNJE, MNOGO, PALI
} ///:~

//: nabrojani/Pljeskavica.java
package nabrojani;
import static enumerated.LJutina.*;

public class Pljeskavica {
```

```
  LJutina degree;
  public Pljeskavica(LJutina stepen) { this.stepen = stepen;}
  public String toString() { return "Pljeskavica je "+ stepen;}
  public static void main(String[] args) {
    System.out.println(new Pljeskavica(NE));
    System.out.println(new Pljeskavica(SREDNJE));
    System.out.println(new Pljeskavica(MNOGO));
  }
} /* Ispis:
Pljeskavica je NE
Pljeskavica je SREDNJE
Pljeskavica je MNOGO
*///:~
```

static import uvodi sve identifikatore **enum** instanci u lokalni prostor imena, pa oni ne moraju biti potpuni (kvalifikovani). Da li je to dobro ili je bolje biti izričit i navesti potpuna imena svih **enum** instanci? To verovatno zavisi od složenosti koda. Prevodilac sigurno neće dopustiti da se upotrebi pogrešan tip, pa samo treba da se postarate da kôd bude razumljiv čitaocu. U mnogim situacijama će i nepotpuna imena biti dovoljno razumljiva, ali se to može prosuditi samo od slučaja do slučaja.

Vodite računa o tome da ovu tehniku nije moguće koristiti ako je **enum** definisan u istoj datoteci ili podrazumevanom paketu. (Kao da je u kompaniji Sun bilo različitih mišljenja o tome treba li to dozvoliti.)

Dodavanje metoda nabrojanom tipu

Iz nabrojanog tipa ne možete izvoditi nove potklase, inače se **enum** može tretirati kao obična klasa. To znači da mu možete dodavati metode. **enum** čak može imati svoju metodu **main**( ).

Videli ste da podrazumevana metoda **toString**( ) navodi samo ime **enum** instance. Ukoliko želite da date drugačiji opis određenog nabrojanog tipa, možete napraviti konstruktor koji će hvatati dodatne informacije i metode koje će davati obimniji opis, kao u sledećem primeru:

```
//: nabrojani/VesticaIzOza.java
// Veštice iz Oza.
import static net.mindview.util.Print.*;

public enum VesticaIzOza {
  // Prvo morate definisati instance, pre metoda:
  ZAPAD("Gospođica Galč, poznata i kao zla veštica sa zapada"),
  SEVER("Glinda, dobra veštica severa"),
  ISTOK("Zla veštica istoka, nosilac crvenih " +
    "papučica, koju je smrvila Dorotina kuća"),
  JUG("Verovatno dobra, ali je nema");
  private String opis;
  // Konstruktoru se mora pristupati paketno ili privatno:
  private VesticaIzOza(String opis) {
```

```
    this.opis = opis;
  }
  public String getDescription() { return opis; }
  public static void main(String[] args) {
    for(VesticaIzOza vestica : VesticaIzOza.values())
      print(vestica + ": " + vestica.getDescription());
  }
} /* Ispis:
WEST: Gospođica Galč, poznata i kao zla veštica sa zapada
NORTH: Glinda, dobra veštica sa severa
EAST: Zla veštica sa istoka, nosilac crvenih papučica, koju je smrvila Doro-
tina kuća
SOUTH: Verovatno dobra, ali je nema
*///:~
```

Ukoliko nameravate da definišete metode, sekvencu **enum** instanci morate završiti znakom tačka i zarez. Takođe, Java nas prisiljava da u nabrojanom tipu (**enum**) najpre definišemo instance. Izazvaćete grešku u vreme prevođenja ako pokušate da ih definišete nakon neke od metoda ili polja.

Konstruktor i metode imaju isti oblik kao u uobičajenim klasama, zato što ovo i *jeste* obična klasa uz nekoliko ograničenja. Dakle, s nabrajanjima možete raditi gotovo sve što poželite (iako ćete se verovatno truditi da ostanu prilično jednostavna).

Mada je u prethodnom primeru konstruktor privatan, ne bi bilo velike razlike i da ste primenili neki drugi pristup – konstruktor se može upotrebiti samo za pravljenje **enum** instanci koje ste deklarisali unutar definicije nabrojanog tipa; prevodilac neće dozvoliti da ga upotrebite za pravljenje novih instanci nakon završetka te definicije.

Redefinisanje **enum** metoda

Sledi opis drugog načina proizvodnje drugačijih znakovnih nizova za nabrojane tipove. U ovom slučaju, imena instanci su dobra, ali želimo da ih preformatiramo radi prikazivanja. Redefinisanje metode **toString**( ) za **enum** isto je kao redefinisanje za uobičajenu klasu:

```
//: nabrojani/SvemirskiBrod.java
public enum SvemirskiBrod {
  IZVIDJAC, TERETNI, TRANSPORTNI, KRSTARICA, BOJNIBROD, MATICNI;
  public String toString() {
    String id = name();
    String mala = id.substring(1).toLowerCase();
    return id.charAt(0) + mala;
  }
  public static void main(String[] args) {
    for(SvemirskiBrod s : values()) {
      System.out.println(s);
    }
  }
} /* Ispis:
Izvidjac
Teretni
```

```
Transportni
Krstarica
Bojnibrod
Maticni
*///:~
```

Metoda **toString( )** pribavlja ime klase **SvemirskiBrod** pozivanjem metode **name( )**, a njen rezultat menja tako da je samo prvo slovo veliko.

Nabrojani tipovi u naredbama **switch**

Jedno veoma podesno svojstvo nabrojanih tipova jeste način na koji se oni mogu upotrebiti u naredbama **switch**. **switch** obično radi samo s celobrojnim vrednostima, ali pošto nabrojani tipovi imaju utvrđen celobrojni poredak, a redni broj instance daje metoda **ordinal( )** (izgleda da nešto slično radi prevodilac), nabrojane tipove možemo upotrebljavati u naredbama **switch**.

Iako ime **enum** instance obično morate upotpuniti njenim tipom, to ne morate da radite u naredbi **case**. U sledećem primeru **enum** je upotrebljen za pravljenje male mašine stanja:

```
//: nabrojani/Semafor.java
// Nabrojani tipovi u naredbama switch.
import static net.mindview.util.Print.*;

// Definicija nabrojanog tipa:
enum Signal { ZELENO, ZUTO, CRVENO, }

public class Semafor {
  Signal svetlo = Signal.CRVENO;
  public void change() {
    switch(svetlo) {
      // Obratite pažnju na to da u naredbi case
      // ne morate da pišete Signal.CRVENO:
      case CRVENO: svetlo = Signal.ZELENO;
                   break;
      case ZELENO: svetlo = Signal.ZUTO;
                   break;
      case ZUTO:   svetlo = Signal.CRVENO;
                   break;
    }
  }
  public String toString() {
    return "Na semaforu je " + svetlo;
  }
  public static void main(String[] args) {
    Semafor s = new Semafor();
    for(int i = 0; i < 7; i++) {
      print(s);
      s.change();
    }
  }
```

```
} /* Ispis:
Na semaforu je CRVENO
Na semaforu je ZELENO
Na semaforu je ZUTO
Na semaforu je CRVENO
Na semaforu je ZELENO
Na semaforu je ZUTO
Na semaforu je CRVENO
*///:~
```

Prevodilac se ne žali da unutar naredbe **switch** nema naredbe **default**, ali ne zato što je uočio da za svaku instancu klase **Signal** imate naredbe **case**. Neće se žaliti ni ako jednu od naredaba **case** pretvorite u komentar. Znači, sami morate paziti na to da definišete sve moguće slučajeve. S druge strane, ukoliko iz naredbe **case** pozivate **return**, *prevodilac će se žaliti* ako nemate **default** – čak i ako ste definisali sve moguće vrednosti tog nabrojanog tipa.

Vežba 1: (2) Izmenite **Semafor.java** pomoću uvoza statičnih članova tako da ne morate navoditi potpuna imena **enum** instanci.

Misterija metode **values()**

Već sam rekao da prevodilac pravi sve **enum** klase i da one nasleđuju klasu **Enum**. Međutim, iz dokumentacije klase **Enum** vidi se da ona nema metodu **values( )**, iako smo mi tu metodu koristili. Ima li još „skrivenih" metoda? Napisaćemo mali program koji će to otkriti pomoću refleksije:

```
//: nabrojani/Refleksija.java
// Analiza nabrojanih tipova pomoću refleksije.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Istrazi { OVDE, TAMO }

public class Refleksija {
  public static Set<String> analiziraj(Class<?> enumKlasa) {
    print("----- Analiza klase " + enumKlasa + " -----");
    print("Interfejsi:");
    for(Type t : enumKlasa.getGenericInterfaces())
      print(t);
    print("Natklasa: " + enumKlasa.getSuperclass());
    print("Metode: ");
    Set<String> metode = new TreeSet<String>();
    for(Method m : enumKlasa.getMethods())
      metode.add(m.getName());
    print(metode);
    return metode;
  }
```

```
  public static void main(String[] args) {
    Set<String> metodeIstrazi = analiziraj(Istrazi.class);
    Set<String> enumMetode = analiziraj(Enum.class);
    print("Istrazi.containsAll(Enum)? " +
      metodeIstrazi.containsAll(enumMetode));
    printnb("Istrazi.removeAll(Enum): ");
    metodeIstrazi.removeAll(enumMetode);
    print(metodeIstrazi);
    // Prevedi unazad kôd za enum:
    OSIzvrsenje.komanda("javap Istrazi");
  }
} /* Ispis:
----- Analiza klase class Istrazi -----
Interfejsi:
Natklasa: class java.lang.Enum
Metode:
[compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify, noti-
fyAll, ordinal, toString, valueOf, values, wait]
----- Analiza klase class java.lang.Enum -----
Interfejsi:
java.lang.Comparable<E>
interface java.io.Serializable
Natklasa: class java.lang.Object
Metode:
[compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify, noti-
fyAll, ordinal, toString, valueOf, wait]
Istrazi.containsAll(Enum)? true
Istrazi.removeAll(Enum): [values]
Compiled from "Refleksija.java"
final class Istrazi extends java.lang.Enum{
    public static final Istrazi OVDE;
    public static final Istrazi TAMO;
    public static final Istrazi[] values();
    public static Istrazi valueOf(java.lang.String);
    static {};
}
*///:~
```

Dakle, odgovor je da je **values( )** statična metoda koju je dodao prevodilac. I metoda **valueOf( )** je dodata klasi **Istrazi** dok se pravio **enum**. To pomalo zbunjuje, zato što postoji i metoda **valueOf( )** koja je deo klase **Enum**, ali ona ima dva argumenta, a dodata metoda samo jedan. Metoda interfejsa **Set** ovde je upotrebljena samo za pronalaženje imena metoda, ne i njihovih potpisa, pa posle poziva **Istrazi.removeAll(Enum)** preostaje samo niz **[values]**.

Iz rezultata vidite da je prevodilac klasu **Istrazi** označio sa **final**, pa od nje ne možete izvoditi potklase. Tu je i statična odredba za inicijalizaciju koja se može redefinisati (pokazaću to kasnije).

Zbog brisanja (opisanog u poglavlju *Generički tipovi*), dekompajler nema punu informaciju o nabrojanom tipu (**Enum**), pa prikazuje da je natklasa od **Istrazi** sirovi **Enum**, umesto stvarne **Enum<Istrazi>**.

Pošto je **values( )** statična metoda koju je u definiciju **enum** umetnuo prevodilac, ako neki od **enum** tipova svedete naviše na **Enum**, metoda **values( )** neće biti dostupna. Međutim, imajte u vidu da u klasi **Class** postoji metoda **getEnumConstants( )**, pa čak i ako metoda **values( )** nije deo interfejsa od **Enum**, **enum** instance ipak možete dobiti pomoću **Class** objekta:

```
//: nabrojani/SvodjenjeEnumaNavise.java
// Kada se nabrojan tip svede naviše,
// metoda values() postaje nedostupna

enum Pretrazivanje { OVAMO, ONAMO }

public class SvodjenjeEnumaNavise {
  public static void main(String[] args) {
    Pretrazivanje[] vrdns = Pretrazivanje.values();
    Enum e = Pretrazivanje.OVAMO; // Svođenje naviše
    // e.values(); // Enum nema metodu values()
    for(Enum en : e.getClass().getEnumConstants())
      System.out.println(en);
  }
} /* Ispis:
OVAMO
ONAMO
*///:~
```

Pošto metoda **getEnumConstants( )** pripada klasi **Class**, možete je pozvati i za klase koje nemaju nabrojane tipove:

```
//: nabrojani/NijeEnum.java

public class NijeEnum {
  public static void main(String[] args) {
    Class<Integer> intKlasa = Integer.class;
    try {
      for(Object en : intKlasa.getEnumConstants())
        System.out.println(en);
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Ispis:
java.lang.NullPointerException
*///:~
```

Metoda vraća **null**, pa ćete dobiti izuzetak kada pokušate da upotrebite njen rezultat.

Realizuje, ne nasleđuje

Rekao sam da su svi nabrojani tipovi izvedeni iz klase **java.lang.Enum**. Pošto Java ne podržava višestruko nasleđivanje, to znači da nabrojan tip ne možete napraviti preko nasleđivanja:

```
enum NijeMoguce extends Pet { ... // Ne radi
```

Međutim, *moguće* je napraviti nabrojan tip koji realizuje jedan ili više interfejsa:

```
//: nabrojani/crtaci/EnumRealizacija.java
// Nabrojani tip može da realizuje interfejs
package nabrojani.crtaci;
import java.util.*;
import net.mindview.util.*;

enum LikIzCrtaca
implements Generator<LikIzCrtaca> {
  FUCA, GRDA, TIBA, BLESA, SKOKA, LUDA, BOBA;
  private Random slucajan = new Random(47);
  public LikIzCrtaca next() {
    return values()[slucajan.nextInt(values().length)];
  }
}

public class EnumRealizacija {
  public static <T> void printNext(Generator<T> gsb) {
    System.out.print(gsb.next() + ", ");
  }
  public static void main(String[] args) {
    // Izaberite bilo koju instancu:
    LikIzCrtaca lic = LikIzCrtaca.BOBA;
    for(int i = 0; i < 10; i++)
      printNext(lic);
  }
} /* Ispis:
BOBA, TIBA, BOBA, GRDA, LUDA, TIBA, FUCA, LUDA, LUDA, FUCA,
*///:~
```

Rezultat je malo neobičan, pošto morate imati instancu nabrojanog tipa da biste za nju mogli pozvati metodu. Međutim, **LikIzCrtaca** sada može da prihvati svaka metoda koja prima **Generator**, na primer, metoda **printNext( )**.

Vežba 2: (2) Umesto da realizujete interfejs, napravite **next( )** statičnom metodom. Koje su prednosti i mane tog pristupa?

Nasumičan izbor

U mnogim primerima u ovom poglavlju neophodan je nasumičan izbor **enum** instanci, kao što ste videli u metodi **LikIzCrtaca.next( )**. To možemo uopštiti pomoću generičkih tipova i rezultat smestiti u zajedničku biblioteku:

```
//: net/mindview/util/NabrojaniTipovi.java
package net.mindview.util;
import java.util.*;

public class NabrojaniTipovi {
  private static Random slucajan = new Random(47);
  public static <T extends Enum<T>> T random(Class<T> ec) {
    return random(ec.getEnumConstants());
  }
  public static <T> T random(T[] vrednosti) {
    return vrednosti[slucajan.nextInt(vrednosti.length)];
  }
} ///:~
```

Prilično neobična sintaksa **<T extends Enum<T>>** opisuje **T** kao instancu nekog nabrojanog tipa. Objekat te klase činimo dostupnim tako što prosleđujemo **Class<T>**, i zato se može napraviti niz **enum** instanci. Preklopljena metoda **random( )** treba da zna samo to da dobija **T[]**, pošto ne mora da obavlja **Enum** operacije; ona samo nasumično bira neki element niza. Povratni tip je tačno taj nabrojan tip.

Evo jednostavne provere metode **random( )**:

```
//: nabrojani/ProveraMetodeRandom.java
import net.mindview.util.*;

enum Aktivnost { SEDENJE, LEZANJE, STAJANJE, POSKAKIVANJE,
  TRCANJE, IZMICANJE, SKAKANJE, PADANJE, LETENJE }

public class ProveraMetodeRandom {
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++)
      System.out.print(NabrojaniTipovi.random(Aktivnost.class) + " ");
  }
} /* Ispis:
STAJANJE LETENJE TRCANJE STAJANJE TRCANJE STAJANJE LEZANJE IZMICANJE SEDENJE
TRCANJE POSKAKIVANJE POSKAKIVANJE POSKAKIVANJE TRCANJE STAJANJE LEZANJE
PADANJE TRCANJE LETENJE LEZANJE
*///:~
```

Iako su **NabrojaniTipovi** mala klasa, videćete da njenom zaslugom u ovom poglavlju izbegavamo prilično količinu dupliranja. Dupliranje često dovodi do grešaka, pa je uklanjanje dupliranja koristan posao.

Upotreba interfejsa za organizovanje

To što je nemoguće nasleđivati nabrojan tip ume ponekad da zasmeta. Nasleđivanje nabrojanog tipa potrebno je zbog povećavanja broja elemenata prvobitnog nabrojanog tipa i zbog pravljenja potkategorija pomoću podtipova.

Kategorizaciju možete ostvariti grupisanjem elemenata unutar interfejsa i pravljenjem nabrojanog tipa na osnovu tog interfejsa. Na primer, pretpostavimo da imate različite klase hrane koje biste hteli da napravite kao nabrojane tipove, ali biste ipak hteli da svaka od njih bude neki tip izveden iz klase **Hrana**. Evo kako to izgleda:

```
//: nabrojani/jelovnik/Hrana.java
// Potkategorizacija nabrojanih tipova unutar interfejsa.
package nabrojani.jelovnik;

public interface Hrana {
  enum Predjelo implements Hrana {
    SALATA, SUPA, PROLECNE_ROLNICE;
  }
  enum GlavnoJelo implements Hrana {
    LAZANJE, PLJESKAVICA, PAD_THAI,
    MAHUNARKE, PILAV, SARMA;
  }
  enum Desert implements Hrana {
    TIRAMISU, SLADOLED, SVARCVALD_TORTA,
    VOCE, KARAMEL_KREM;
  }
  enum Kafa implements Hrana {
    CRNA_KAFA, KAFA_BEZ_KOFEINA, ESPRESSO,
    KAFA_S_MLEKOM, CAPPUCCINO, CAJ, BILJNI_CAJ;
  }
} ///:~
```

Pošto nabrojan tip može imati samo podtipove napravljene realizacijom interfejsa, svaki ugnežđeni **enum** realizuje obuhvatajući interfejs **Hrana**. Sada možemo reći da je „svako jelo neki tip klase **Hrana**", kao što vidite ovde:

```
//: nabrojani/jelovnik/VrstaHrane.java
package nabrojani.jelovnik;
import static nabrojani.jelovnik.Hrana.*;

public class VrstaHrane {
  public static void main(String[] args) {
    Hrana hrana = Predjelo.SALATA;
    hrana = GlavnoJelo.LAZANJE;
    hrana = Desert.SLADOLED;
    hrana = Kafa.CAPPUCCINO;
  }
} ///:~
```

Svođenje naviše na klasu **Hrana** funkcioniše za svaki **enum** tip koji realizuje interfejs **Hrana**, pa su svi oni tipovi klase **Hrana**.

Međutim, za rad sa skupom tipova interfejs nije toliko upotrebljiv kao **enum**. Ukoliko želite da imate „nabrojan tip drugih nabrojanih tipova", možete napraviti obuhvatajući **enum** s po jednom instancom za svaki **enum** u klasi **Hrana**:

```
//: nabrojani/jelovnik/VrstaJela.java
package nabrojani.jelovnik;
import net.mindview.util.*;

public enum VrstaJela {
  PREDJELO(Hrana.Predjelo.class),
  GLAVNOJELO(Hrana.GlavnoJelo.class),
  DESERT(Hrana.Desert.class),
  KAFA(Hrana.Kafa.class);
  private Hrana[] vrednosti;
  private VrstaJela(Class<? extends Hrana> vrsta) {
    vrednosti = vrsta.getEnumConstants();
  }
  public Hrana randomSelection() {
    return NabrojaniTipovi.random(vrednosti);
  }
} ///:~
```

Svaki od gorenavedenih nabrojanih tipova prima odgovarajući objekat tipa **Class** kao argument konstruktora od kojega metodom **getEnum-Constants( )** može da izvuče i uskladišti sve **enum** instance. Te instance kasnije koristi metoda **randomSelection( )**, pa sada možemo da napravimo nasumično generisan obrok tako što ćemo izabrati po jednu stavku klase **Hrana** iz svakog tipa **VrstaJela**:

```
//: nabrojani/jelovnik/Obrok.java
package nabrojani.jelovnik;

public class Obrok {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(VrstaJela pojedino_jelo : VrstaJela.vrednosti()) {
        Hrana hrana = pojedino_jelo.randomSelection();
        System.out.println(hrana);
      }
      System.out.println("---");
    }
  }
} /* Ispis:
PROLECNE_ROLNICE
SARMA
VOCE
KAFA_BEZ_KOFEINA
---
SUPA
```

```
SARMA
VOCE
CAJ
---
SALATA
PLJESKAVICA
VOCE
CAJ
---
SALATA
PLJESKAVICA
KARAMEL_KREM
KAFA_S_MLEKOM
---
SUPA
PLJESKAVICA
TIRAMISU
ESPRESSO
---
*///:~
```

U ovom slučaju, svrha pravljenja nabrojanog tipa nabrojanih tipova jeste iteracija kroz sve tipove **VrstaJela**. Kasnije, u primeru **AutomatZaMaloprodaju.java**, videćete drugačiji pristup kategorizaciji određen drugačijim ograničenjima.

Drugačiji, kompaktniji pristup problemu kategorizacije jeste ugnežđivanje nabrojanih tipova unutar drugih nabrojanih tipova, što se radi ovako:

```
//: nabrojani/KategorijeVrednosnihPapira.java
// Sažetija potkategorizacija nabrojanih tipova.
import net.mindview.util.*;

enum KategorijeVrednosnihPapira {
  DEONICE(VrednosniPapiri.Deonica.class), OBVEZNICE(Vrednosni-
     Papiri.Obveznica.class);
  VrednosniPapiri[] vrednosti;
  KategorijeVrednosnihPapira(Class<? extends VrednosniPapiri> vrsta) {
    vrednosti = vrsta.getEnumConstants();
  }
  interface VrednosniPapiri {
    enum Deonica implements VrednosniPapiri { KRATKOROCNE, DUGOROCNE,
       JEMSTVENE }
    enum Obveznica implements VrednosniPapiri { GRADSKE, RAZNE }
  }
  public VrednosniPapiri randomSelection() {
    return NabrojaniTipovi.random(vrednosti);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      KategorijeVrednosnihPapira kategorija =
        NabrojaniTipovi.random(KategorijeVrednosnihPapira.class);
      System.out.println(kategorija + ": " +
```

```
          kategorija.randomSelection());
      }
    }
} /* Ispis:
OBVEZNICE: GRADSKE
OBVEZNICE: GRADSKE
DEONICE: JEMSTVENE
DEONICE: JEMSTVENE
OBVEZNICE: RAZNE
DEONICE: KRATKOROCNE
DEONICE: DUGOROCNE
OBVEZNICE: GRADSKEDEONICE: DUGOROCNE
OBVEZNICE: RAZNE
*///:~
```

Interfejs **VrednosniPapiri** neophodan je za objedinjavanje obuhvaćenih nabrojanih tipova u jedan zajednički tip. Zatim se oni kategorizuju u zasebne nabrojane tipove unutar **KategorijeVrednosnihPapira**.

Ukoliko primer **Hrana** obradimo na ovaj način, rezultat će izgledati ovako:

```
//: nabrojani/jelovnik/Obrok2.java
package nabrojani.jelovnik;
import net.mindview.util.*;

public enum Obrok2 {
  PREDJELO(Hrana.Predjelo.class),
  GLAVNOJELO(Hrana.GlavnoJelo.class),
  DESERT(Hrana.Desert.class),
  KAFA(Hrana.Kafa.class);
  private Hrana[] vrednosti;
  private Obrok2(Class<? extends Hrana> vrsta) {
    vrednosti = vrsta.getEnumConstants();
  }
  public interface Hrana {
    enum Predjelo implements Hrana {
      SALATA, SUPA, PROLECNE_ROLNICE;
    }
    enum GlavnoJelo implements Hrana {
      LAZANJE, PLJESKAVICA, PAD_THAI,
      MAHUNARKE, PILAV, SARMA;
    }
    enum Desert implements Hrana {
      TIRAMISU, SLADOLED, SVARCVALD_TORTA,
      VOCE, KARAMEL_KREM;
    }
    enum Kafa implements Hrana {
      CRNA_KAFA, KAFA_BEZ_KOFEINA, ESPRESSO,
      KAFA_S_MLEKOM, CAPPUCCINO, CAJ, BILJNI_CAJ;
    }
  }
```

```
  public Hrana randomSelection() {
    return NabrojaniTipovi.random(vrednosti);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Obrok2 obrok : Obrok2.values()) {
        Hrana hrana = obrok.randomSelection();
        System.out.println(hrana);
      }
      System.out.println("---");
    }
  }
} /* Ispis je isti kao u primeru Obrok.java *///:~
```

Na kraju smo dobili praktično isti kôd, ali reorganizovan. U nekim slučajevima, time se dobija jasnija struktura.

Vežba 3: (1) Dodajte nov tip **VrstaJela** u **VrstaJela.java** i pokažite da to radi u programu **Obrok.java**.

Vežba 4: (1) Ponovite prethodnu vežbu za **Obrok2.java**.

Vežba 5: (4) Izmenite program **kontrola/SamoglasniciISuglasnici.java** (iz četvrtog poglavlja) tako da koristi tri nabrojana tipa: **SAMOGLASNIK, PONEKAD_SAMOGLASNIK** i **SUGLASNIK**. Konstruktor nabrojanog tipa treba da prima sva slova koja čine odgovarajuću kategoriju. Uputstvo: upotrebite argumente promenljive dužine i ne zaboravite da oni automatski prave niz.

Vežba 6: (3) Daje li ugnežđivanje tipova **Predjelo, GlavnoJelo, Desert** i **Kafa** unutar interfejsa **Hrana** ikakve prednosti nad slučajem kada bismo ih deklarisali kao samostalne nabrojane tipove koji slučajno svi realizuju interfejs **Hrana**?

Skup **EnumSet** umesto indikatora

Skup (**Set**) jedna je vrsta kolekcije koja ne može sadržati više primeraka istog tipa objekta. Naravno, i za nabrojan tip je neophodno da svi njegovi članovi budu jedinstveni, pa se utoliko ponaša kao skup, ali pošto mu ne možete dodavati elemente niti ih uklanjati iz njega, kao skup nije preterano upotrebljiv. **EnumSet** je dodat u Javu SE5 kako bi zajedno s nabrojanim tipovima napravio zamenu za nekadašnje celobrojne „bit indikatore" (engl. *bit flags*). Takvi indikatori se upotrebljavaju za čuvanje neke vrste informacija tipa uključeno/isključeno, ali morate da radite s bitovima umesto s konceptima, pa je dobijeni kôd često nerazumljiv.

EnumSet radi brzo, pošto mora da se takmiči s bit indikatorima (njegove operacije su obično mnogo brže nego one u skupu **HashSet**). Interno, **EnumSet** se (ako je moguće) predstavlja jednim brojem tipa **long** koji se tretira kao bit-vektor, pa je izuzetno brz i delotvoran. Prednost je to što sada imamo mnogo izražajniji način pokazivanja postojanja ili nepostojanja binarnih informacija, a pri tom ne moramo da brinemo zbog performansi.

Elementi skupa **EnumSet** moraju poticati iz istog nabrojanog tipa. U narednom primeru imamo **enum** svih mesta u zgradi na kojima je ugrađen alarmni senzor:

```
//: nabrojani/AlarmneTacke.java
package nabrojani;
public enum AlarmneTacke {
  STEPENISTE1, STEPENISTE2, HODNIK, KANCELARIJA1, KANCELARIJA2,
  KANCELARIJA3, KANCELARIJA4, KUPATILO, SALA, KUHINJA
} ///:~
```

Taj **EnumSet** se može upotrebiti za praćenje statusa alarma:

```
//: nabrojani/SkupEnumSet.java
// Operacije na skupu tipa EnumSet
package nabrojani;
import java.util.*;
import static nabrojani.AlarmneTacke.*;
import static net.mindview.util.Print.*;

public class SkupEnumSet {
  public static void main(String[] args) {
    EnumSet<AlarmneTacke> tacke =
      EnumSet.noneOf(AlarmneTacke.class); // Prazan skup
    tacke.add(KUPATILO);
    print(tacke);
    tacke.addAll(EnumSet.of(STEPENISTE1, STEPENISTE2, KUHINJA));
    print(tacke);
    tacke = EnumSet.allOf(AlarmneTacke.class);
    tacke.removeAll(EnumSet.of(STEPENISTE1, STEPENISTE2, KUHINJA));
    print(tacke);
    tacke.removeAll(EnumSet.range(KANCELARIJA1, KANCELARIJA4));
    print(tacke);
    tacke = EnumSet.complementOf(tacke);
    print(tacke);
  }
} /* Ispis:
[KUPATILO]
[STEPENISTE1, STEPENISTE2, KUPATILO, KUHINJA]
[HODNIK, KANCELARIJA1, KANCELARIJA2, KANCELARIJA3, KANCELARIJA4,
KUPATILO, SALA]
[HODNIK, KUPATILO, SALA]
[STEPENISTE1, STEPENISTE2, KANCELARIJA1, KANCELARIJA2, KANCELARIJA3,
KANCELARIJA4, KUHINJA]
*///:~
```

Statičan uvoz pojednostavljuje korišćenje **enum** konstanti. Imena metoda su prilično jasna sama po sebi, a detalje možete pročitati u HTML dokumentaciji na Webu. Kada to uradite, videćete nešto zanimljivo – metoda **of**( ) je preklopljena i za argumente promenljive dužine i za pojedinačne metode koje primaju od dva do pet eksplicitnih argumenata. To je pokazatelj brige za performanse skupa **EnumSet**, pošto bi problem rešila

već i jedna metoda **of( )** sa argumentima promenljive dužine, ali bi to bilo nešto manje efikasno nego kada su argumenti zadati eksplicitno. Dakle, ako metodu **of( )** pozovete sa od dva od pet argumenata, dobićete eksplicitne (nešto brže) metode, ali ukoliko je pozovete s jednim ili s više od pet argumenata, dobićete vararg verziju od **of( )**. Obratite pažnju na sledeće: ukoliko je pozovete s jednim argumentom, prevodilac neće praviti vararg niz, pa pozivanje te verzije s jednim argumentom neće povećati režijske troškove.

EnumSet se pravi povrh podupirućeg broja tipa **long**, **long** ima 64 bita, a svaka **enum** instanca zahteva jedan bit za informaciju uključeno/isključeno ili postojanje/nepostojanje. To znači da **EnumSet** za jedan **enum** od najviše 64 elementa troši samo jedan broj tipa **long**. Šta se dešava ako u nabrojanom tipu imate više od 64 elementa?

```
//: nabrojani/VelikiEnumSet.java
import java.util.*;

public class VelikiEnumSet {
  enum Veliki { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
    A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
    A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
    A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
    A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
    A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
    A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
  public static void main(String[] args) {
    EnumSet<Veliki> bigEnumSet = EnumSet.allOf(Veliki.class);
    System.out.println(bigEnumSet);
  }
} /* Ispis:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16,
A17, A18, A19, A20, A21, A22, A23, A24, A25, A26, A27, A28, A29, A30, A31,
A32, A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43, A44, A45, A46,
A47, A48, A49, A50, A51, A52, A53, A54, A55, A56, A57, A58, A59, A60, A61,
A62, A63, A64, A65, A66, A67, A68, A69, A70, A71, A72, A73, A74, A75]
*///:~
```

EnumSet očigledno nema problema s nabrojanim tipom koji ima više od 64 elementa, pa možemo pretpostaviti da drugi **long** dodaje po potrebi.

Vežba 7: (3) Pronadite izvorni kod za **EnumSet** i objasnite kako radi.

Korišćenje mape **EnumMap**

EnumMap je specijalizovana mapa čiji ključevi moraju poticati iz istog nabrojanog tipa. Zbog ograničenja kojima je podvrgnut **enum**, **EnumMap** se interno može realizovati kao niz. Zato je izuzetno brza, pa je slobodno možete koristiti za pretraživanja čiju osnovu čine nabrojani tipovi.

Za ključeve u nabrojanom tipu možete pozivati samo metodu **put( )**, a osim toga, sa ovom mapom radite kao sa svakom drugom.

U narednom primeru prikazana je upotreba projektnog obrasca *Command* (Komanda). Taj obrazac počinje interfejsom koji (najčešće) sadrži samo jednu metodu; onda se prave razne realizacije u kojima se ta metoda različito ponaša. Vi instalirate objekte tipa Command, a program ih poziva po potrebi:

```
//: nabrojani/MapeEnumMap.java
// Osnove rada s mapama EnumMap.
package nabrojani;
import java.util.*;
import static nabrojani.AlarmneTacke.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class MapeEnumMap {
  public static void main(String[] args) {
    EnumMap<AlarmneTacke,Command> em =
      new EnumMap<AlarmneTacke,Command>(AlarmneTacke.class);
    em.put(KUHINJA, new Command() {
      public void action() { print("Požar u kuhinji!"); }
    });
    em.put(KUPATILO, new Command() {
      public void action() { print("Uzbuna u kupatilu!"); }
    });
    for(Map.Entry<AlarmneTacke,Command> e : em.entrySet()) {
      printnb(e.getKey() + ": ");
      e.getValue().action();
    }
    try { // Ako se ne pronađe vrednost za određeni ključ:
      em.get(SALA).action();
    } catch(Exception e) {
      print(e);
    }
  }
} /* Ispis:
KUPATILO: Uzbuna u kupatilu!
KUHINJA: Požar u kuhinji!
java.lang.NullPointerException
*///:~
```

Kao u skupu **EnumSet**, redosled elemenata u mapi **EnumMap** određen je redosledom njihovog definisanja u odgovarajućem nabrojanom tipu.

Poslednji deo metode **main**( ) pokazuje da u mapi za svaki nabrojani tip uvek postoji stavka ključ, ali je njena vrednost **null** ukoliko za taj ključ niste pozvali **put**( ).

Jedna od prednosti mape **EnumMap** nad *metodama koje se menjaju u zavisnosti od konstante (nabrojanog tipa)* jeste to što je u **EnumMap** dozvoljeno menjanje objekata vrednost, dok metode koje se menjaju u zavisnosti od konstante bivaju fiksirane u vreme prevođenja.

Kao što ćete videti u nastavku poglavlja, mapama **EnumMap** možemo obavljati *višekratno otkrivanje tipa* u situacijama gde više nabrojanih tipova ulazi u međusobnu interakciju.

Metode koje se menjaju u zavisnosti od konstante

Javini nabrojani tipovi imaju veoma zanimljivu osobinu koja omogućuje da svakoj **enum** instanci definisanjem njenih metoda date različito ponašanje. To ćete postići definisanjem jedne ili više apstraktnih metoda u sklopu nabrojanog tipa, a zatim definisanjem tih metoda za svaku **enum** instancu. Na primer:

```
//: nabrojani/MetodeKojeSeMenjajuUZavisnostiOdKonstante.java
import java.util.*;
import java.text.*;

public enum MetodeKojeSeMenjajuUZavisnostiOdKonstante {
  DATE_TIME {
    String getInfo() {
      return
        DateFormat.getDateInstance().format(new Date());
    }
  },
  CLASSPATH {
    String getInfo() {
      return System.getenv("CLASSPATH");
    }
  },
  VERSION {
    String getInfo() {
      return System.getProperty("java.version");
    }
  };
  abstract String getInfo();
  public static void main(String[] args) {
    for(MetodeKojeSeMenjajuUZavisnostiOdKonstante mksmuzok : values())
      System.out.println(mksmuzok.getInfo());
  }
} /* (Pokrenite da biste videli ispis rezultata) *///:~
```

Metode možete pronaći i pozvati preko njima pridružene **enum** instance. To se često naziva *kôd upravljan tabelom*. (Obratite pažnju na sličnost s prethodno spomenutim obrascem Command.)

U objektno orijentisanom programiranju, različita ponašanja se povezuju s različitim klasama. Pošto svaka instanca nabrojanog tipa može imati sopstveno ponašanje ostvareno metodama koje se menjaju u zavisnosti od konstante, stiče se utisak da je svaka instanca zaseban tip. U gornjem primeru, svaka **enum** instanca se tretira kao „osnovni tip" **MetodeKojeSeMenjajuUZavisnostiOdKonstante**, a polimorfno ponašanje dobijamo pozivom metode **getInfo**( ).

Međutim, ta sličnost ne ide predaleko. **enum** instance ne možete tretirati kao tipove klasa:

```
//: nabrojani/NisuKlase.java
// {Pokretanje: javap -c KaoKlase}
import static net.mindview.util.Print.*;

enum KaoKlase {
  NAMIGNUTI { void ponasanje() { print("Ponasanje1"); } },
  TREPNUTI { void ponasanje() { print("Ponasanje2"); } },
  KLIMNUTI { void ponasanje() { print("Ponasanje3"); } };
  abstract void ponasanje();
}

public class NisuKlase {
  // void f1(KaoKlase.NAMIGNUTI instance) {} // Ne može
} /* Ispis:
Compiled from "NisuKlase.java"
abstract class KaoKlase extends java.lang.Enum{
public static final KaoKlase NAMIGNUTI;

public static final KaoKlase TREPNUTI;

public static final KaoKlase KLIMNUTI;
...
*///:~
```

U metodi **f1( )**, vidite da prevodilac ne dozvoljava upotrebu **enum** instance kao tipa klase, što ima smisla ako razmotrite kôd koji on generiše – svaki **enum** element je statična finalna instanca tipa **KaoKlase**.

Takođe, zbog svoje statičnosti, **enum** instance unutrašnjih nabrojanih tipova ne ponašaju se kao obične unutrašnje klase; mi nemamo pristupa nestatičnim poljima ili metodama u spoljnoj klasi.

Zanimljiviji primer je perionica automobila. Svakom korisniku se daje meni opcija za pranje. Svaka opcija obavlja drugu aktivnost i može joj se pridružiti metoda koja se menja u zavisnosti od konstante (nabrojanog tipa). Po jedan **EnumSet** čuva opcije koje je svaki korisnik odabrao:

```
//: nabrojani/PerionicaAutomobila.java
import java.util.*;
import static net.mindview.util.Print.*;

public class PerionicaAutomobila {
  public enum Ciklus {
    DONJIPOSTROJ {
      void dejstvo() { print("Prskanje donjeg postroja"); }
    },
    PRANJETOCKOVA {
```

```
      void dejstvo() { print("Pranje točkova"); }
    },
    PRETPRANJE {
      void dejstvo() { print("Omekšavanje nečistoće"); }
    },
    PRANJE {
      void dejstvo() { print("Pranje"); }
    },
    VOSKIRANJE {
      void dejstvo() { print("Nanošenje vrućeg voska"); }
    },
    ISPIRANJE {
      void dejstvo() { print("Ispiranje"); }
    },
    SUSENJE {
      void dejstvo() { print("Sušenje ventilatorom"); }
    };
    abstract void dejstvo();
  }
  EnumSet<Ciklus> ciklusi =
    EnumSet.of(Ciklus.PRANJE, Ciklus.ISPIRANJE);
  public void add(Ciklus ciklus) { ciklusi.add(ciklus); }
  public void operiKola() {
    for(Ciklus c : ciklusi)
      c.dejstvo();
  }
  public String toString() { return ciklusi.toString(); }
  public static void main(String[] args) {
    PerionicaAutomobila pranje = new PerionicaAutomobila();
    print(pranje);
    pranje.operiKola();
    // Redosled dodavanja nije važan:
    pranje.add(Ciklus.SUSENJE);
    pranje.add(Ciklus.SUSENJE); // Duplikati se zanemaruju
    pranje.add(Ciklus.ISPIRANJE);
    pranje.add(Ciklus.VOSKIRANJE);
    print(pranje);
    pranje.operiKola();
  }
} /* Ispis:
[PRANJE, ISPIRANJE]
Pranje
Ispiranje
[PRANJE, VOSKIRANJE, ISPIRANJE, SUSENJE]
Pranje
Nanošenje vrućeg voska
Ispiranje
Sušenje ventilatorom
*///:~
```

Sintaksa za definisanje metode koja se menja u zavisnosti od konstante (nabrojanog tipa) u osnovi je kao ona za anonimnu unutrašnju klasu, ali sažetija.

U ovom primeru pokazana su i neka obeležja skupa **EnumSet**. Pošto se radi o skupu, on čuva samo po jedan primerak svake stavke, pa se zanemaruju duplikati poziva metode **add( )** sa istim argumentom (to ima smisla, pošto stanje bita samo jednom možete da promenite u „uključeno"). Sem toga, nije važan redosled dodavanja **enum** instanci – redosled ispisivanja rezultata biva određen redosledom deklarisanja u nabrojanom tipu.

Da li je moguće redefinisati metode koje se menjaju u zavisnosti od konstante, umesto što realizujemo apstraktnu metodu? Da, kao što se vidi u ovom primeru:

```
//: nabrojani/RedefinisanjeMetUZavOdK.java
import static net.mindview.util.Print.*;

public enum RedefinisanjeMetUZavOdK {
  NAVRTKA, ZAVRTANJ,
  PODLOSKA {
    void f() { print("Redefinisana metoda"); }
  };
  void f() { print("podrazumevano ponašanje"); }
  public static void main(String[] args) {
    for(RedefinisanjeMetUZavOdK rmuzok : values()) {
      printnb(rmuzok + ": ");
      rmuzok.f();
    }
  }
} /* Ispis:
NAVRTKA: podrazumevano ponašanje
ZAVRTANJ: podrazumevano ponašanje
PODLOSKA: Redefinisana metoda
*///:~
```

Iako nabrojani tipovi ne dozvoljavaju pisanje određenih vrsta koda, po pravilu bi trebalo da eksperimentišete s njima kao da su klase.

Stvaranje lanca odgovornosti pomoću nabrojanih tipova

U projektnom obrascu *Chain of Responsibility* (Lanac odgovornosti), definiše se više različitih načina rešavanja određenog problema, a zatim se oni ulančavaju. Kada se pojavi neki zahtev, on se prosleđuje duž lanca sve dok jedno od rešenja ne uzmogne da ga obradi.

Jednostavan Chain of Responsibility lako je realizovati pomoću metoda koje se menjaju u zavisnosti od konstante. Zamislite model pošte u kojoj se pokušava obrađivanje svake pošiljke na najopštiji mogući način i tako sve dok se pošiljka ne proglasi za neuručljivu. Svaki pokušaj možemo smatrati jednim projektnim obrascem *Strategy*, a celokupnu listu za Chain of Responsibility.

Počećemo opisom pošiljke. Sva njena važna obeležja mogu biti izražena nabrojanim tipovima. Pošto ćemo objekte tipa **Posiljka** generisati nasumično, najlakše ćemo smanjiti verovatnoću da će (na primer) određena pošiljka dobiti **DA** za **OpstaDostava** ukoliko

napravimo više ne-**DA** instanci, pa će vam definicije nabrojanih tipova spočetka izgledati malo čudno.

U klasi **Posiljka** videćete metodu **nasumicnaPosiljka( )** koja pravi nasumične primerke probnih pošiljki. Metoda **generator( )** proizvodi objekat koji realizuje interfejs **Iterable** i metodom **nasumicnaPosiljka( )** proizvodi više pošiljki, po jednu za svaki poziv metode **next( )** putem iteratora. Takva konstrukcija omogućuje jednostavno pravljenje foreach petlje pozivom metode **Posiljka.generator( )**:

```
//: nabrojani/Posta.java
// Modelovanje pošte.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
class Posiljka {
Tipovi NE su tu da bi se smanjila verovatnoća nasumičnog izbora:
  enum OpstaDostava {DA,NE1,NE2,NE3,NE4,NE5}
  enum MozeSeSkenirati {NEMOZESESKENIRATI,DA1,DA2,DA3,DA4}
  enum Citljivo {NECITLJIVO,DA1,DA2,DA3,DA4}
  enum Adresa {NETACNA,OK1,OK2,OK3,OK4,OK5,OK6}
  enum PovratnaAdresa {NEDOSTAJE,OK1,OK2,OK3,OK4,OK5}
  OpstaDostava opstaDostava;
  MozeSeSkenirati mozeSeSkenirati;
  Citljivo citljivo;
  Adresa adresa;
  PovratnaAdresa povratnaAdresa;
  static long brojac = 0;
  long id = brojac++;
  public String toString() { return "Pošiljka " + id; }
  public String details() {
    return toString() +
      ", Opšta dostava: " + opstaDostava +
      ", Adresa se moze skenirati: " + MozeSeSkenirati +
      ", Čitljivost adrese: " + Citljivo +
      ", Adresa Adresa: " + adresa +
      ", Povratna adresa: " + povratnaAdresa;
  }
  // Generisanje probne Pošiljke:
  public static Posiljka nasumicnaPosiljka() {
    Posiljka p = new Posiljka();
    p.opstaDostava= NabrojaniTipovi.random(OpstaDostava.class);
    p.MozeSeSkenirati = NabrojaniTipovi.random(MozeSeSkenirati.class);
    p.Citljivo = NabrojaniTipovi.random(Citljivo.class);
    p.adresa = NabrojaniTipovi.random(Adresa.class);
    p.povratnaAdresa = NabrojaniTipovi.random(PovratnaAdresa.class);
    return p;
  }
  public static Iterable<Posiljka> generator(final int broj) {
    return new Iterable<Posiljka>() {
      int n = broj;
      public Iterator<Posiljka> iterator() {
```

```
        return new Iterator<Posiljka>() {
          public boolean hasNext() { return n-- > 0; }
          public Posiljka next() { return nasumicnaPosiljka(); }
          public void remove() { // Nije realizovano
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class Posta {
  enum PrZaObraduPoste {
    OPSTA_DOSTAVA {
      boolean obrada(Posiljka p) {
        switch(p.opstaDostava) {
          case DA:
            print(p + " : koristi opštu dostavu");
            return true;
          default: return false;
        }
      }
    },
    MASINSKO_SKENIRANJE {
      boolean obrada(Posiljka p) {
        switch(p.MozeSeSkenirati) {
          case NEMOZESESKENIRATI: return false;
          default:
            switch(p.adresa) {
              case NETACNA: return false;
              default:
                print(p + " : isporuči automatski");
                return true;
            }
        }
      }
    },
    VIZUELNI_PREGLED {
      boolean obrada(Posiljka p) {
        switch(p.Citljivo) {
          case NECITLJIVO: return false;
          default:
            switch(p.adresa) {
              case NETACNA: return false;
              default:
                print(p + " : isporuči obično");
                return true;
            }
        }
```

```
      }
    },
    VRATI_POSILJAOCU {
      boolean obrada(Posiljka p) {
        switch(p.povratnaAdresa) {
          case NEDOSTAJE: return false;
          default:
            print(p + " : vrati pošiljaocu");
            return true;
        }
      }
    };
    abstract boolean obrada(Posiljka p);
  }
  static void obrada(Posiljka p) {
    for(PrZaObraduPoste przaobradu : PrZaObraduPoste.values())
      if(przaobradu.obrada(p))
        return;
    print(p + " se ne može uručiti");
  }
  public static void main(String[] args) {
    for(Posiljka posiljka : Posiljka.generator(10)) {
      print(posiljka.details());
      obrada(posiljka);
      print("*****");
    }
  }
} /* Ispis:
Pošiljka 0, Opšta dostava:  NE2, Adresa se moze skenirati: NEMOZESESKENIRATI,
Adresa čitljiva: DA3, Adresa Adresa: OK1, Povratna adresa: OK1
Pošiljka 0 : isporuči obično
*****
Pošiljka 1, Opšta dostava:  NE5, Adresa se moze skenirati: DA3, Adresa
čitljiva: NECITLJIVO, Adresa Adresa: OK5, Povratna adresa: OK1
Pošiljka 1 : isporuči automatski
*****
Pošiljka 2, Opšta dostava:  DA, Adresa se moze skenirati: DA3, Adresa
čitljiva: DA1, Adresa Adresa: OK1, Povratna adresa: OK5
Pošiljka 2 : koristi opštu dostavu
*****
Pošiljka 3, Opšta dostava:  NE4, Adresa se moze skenirati: DA3, Adresa
čitljiva: DA1, Adresa Adresa: NETACNA, Povratna adresa: OK4
Pošiljka 3  : vrati pošiljaocu
*****
Pošiljka 4, Opšta dostava:  NE4, Adresa se moze skenirati: NEMOZESESKENIRATI,
Adresa čitljiva: DA1, Adresa Adresa: NETACNA, Povratna adresa: OK2
Pošiljka 4  : vrati pošiljaocu
*****
Pošiljka 5, Opšta dostava:  NE3, Adresa se moze skenirati: DA1, Adresa
čitljiva: NECITLJIVO, Adresa Adresa: OK4, Povratna adresa: OK2
```

```
Pošiljka 5 : isporuči automatski
*****
Pošiljka 6, Opšta dostava:  DA, Adresa se moze skenirati: DA4, Adresa
čitljiva: NECITLJIVO, Adresa Adresa: OK4, Povratna adresa: OK4
Pošiljka 6 : koristi opštu dostavu
*****
Pošiljka 7, Opšta dostava:  DA, Adresa se moze skenirati: DA3, Adresa
čitljiva: DA4, Adresa Adresa: OK2, Povratna adresa: NEDOSTAJE
Pošiljka 7 : koristi opštu dostavu
*****
Pošiljka 8, Opšta dostava:  NE3, Adresa se moze skenirati: DA1, Adresa
čitljiva: DA3, Adresa Adresa: NETACNA, Povratna adresa: NEDOSTAJE
Pošiljka 8 se ne može uručiti
*****
Pošiljka 9, Opšta dostava:  NE1, Adresa se moze skenirati: NEMOZESESKENIRATI,
Adresa čitljiva: DA2, Adresa Adresa: OK1, Povratna adresa: OK4
Pošiljka 9 : isporuči obično
*****
*///:~
```

Chain of Responsibility je izražen u nabrojanom tipu **PrZaObraduPoste**, a redosled **enum** definicija određuje redosled isprobavanja strategija na svakoj pošiljci. Isprobava se svaka strategija redom dok jedna ne uspe ili dok sve ne propadnu – u tom slučaju pošiljka se ne može uručiti.

Vežba 8: (6) Izmenite program **Posta.java** tako da pošiljke može da prosleđuje (s jedne na drugu adresu).

Vežba 9: (5) Izmenite klasu **Posta** tako da koristi **EnumMap**.

Projekat:[2] Na specijalizovanim jezicima kao što je Prolog ovakvi problemi se rešavaju *ulančavanjem unazad* (engl. *backward chaining*). Uz **Posta.java** kao nadahnuće, proučite takve jezike i napišite program koji omogućava lako dodavanje novih „pravila" sistemu.

Mašine stanja s nabrojanim tipovima

Nabrojani tipovi mogu biti idealni za pravljenje *mašina stanja*. Mašina stanja se može nalaziti u jednom od konačnog broja određenih stanja. Mašina obično prelazi iz jednog stanja u drugo na osnovu ulaza, ali postoje i *prelazna stanja*; mašina izlazi iz njih čim obavi njihove zadatke.

U svakom stanju dozvoljeni su određeni ulazi. Razni ulazi menjaju stanje mašine u različita nova stanja. Pošto nabrojani tipovi ograničavaju skup mogućih slučajeva, veoma su podesni za nabrajanje različitih stanja i ulaza.

Svakom stanju obično je pridružen i neki izlaz.

Dobar primer mašine stanja je maloprodajni automat. Prvo u nabrojanom tipu definišemo razne ulaze:

[2] Projekti su predlozi koji se mogu koristiti (recimo) za seminarske radove. Vodič s rešenjima ne sadrži rešenja projekata.

```
//: nabrojani/Ulaz.java
package nabrojani;
import java.util.*;

public enum Ulaz {
  PETOPARAC(5), DESETOPARAC(10), FRTALJ(25), DINAR(100),
  ZUBNAPASTA(200), CIPS(75), SOK(100), SAPUN(50),
  ODUSTANI_OD_TRANSAKCIJE {
    public int iznos(() { // Onemogućiti
      throw new RuntimeException("PREKINI.iznos()");
    }
  },
  STOP { // Ovo mora biti poslednja instanca.
    public int iznos() { // Onemogućiti
      throw new RuntimeException("UGASI.iznos()");
    }
  };
  int vrednost; // U parama
  Ulaz(int vrednost) { this.vrednost = vrednost; }
  Ulaz() {}
  int iznos() { return vrednost; }; // U parama
  static Random slucajan = new Random(47);
  public static Ulaz randomSelection() {
    return values()[slucajan.nextInt(values().length - 1)];    // STOP ne
      // treba obuhvatiti:
  }
} ///:~
```

Vodite računa o tome da je dvama ulazima pridružen određeni iznos, pa je u interfejsu definisana metoda **iznos**( ). Međutim, **iznos**( ) nije prikladno zvati za ostala dva tipa ulaza, pa oni generišu izuzetak ako je pozovete za njih. Iako je ovo neobično (definisati metodu u interfejsu, a potom generisati izuzetak ukoliko je pozovete za neke realizacije), to moramo da radimo zbog ograničenja koja nameće rezervisana reč **enum**.

AutomatZaMaloprodaju će na ove ulaze reagovati tako što će ih najpre kategorizovati pomoću nabrojanog tipa **Kategorija**, pa će te kategorije upotrebiti u naredbi **switch**. Iz ovog primera vidite kako se nabrojani tipovi mogu upotrebiti da bi kôd bio jasniji i lakše se održavao:

```
//: nabrojani/AutomatZaMaloprodaju.java
// {Argumenti: UlazUAutomatZaMaloprodaju.txt}
package nabrojani;
import java.util.*;
import net.mindview.util.*;
import static nabrojani.Ulaz.*;
import static net.mindview.util.Print.*;

enum Kategorija {
  NOVAC(PETOPARAC, DESETOPARAC, FRTALJ, DINAR),
  IZBOR_ARTIKLA(ZUBNAPASTA, CIPS, SOK, SAPUN),
  PREKINI_TRANSAKCIJU(ODUSTANI_OD_TRANSAKCIJE),
```

```
    UGASI(STOP);
    private Ulaz[] vrednosti;
    Kategorija(Ulaz... tipovi) { vrednosti = tipovi; }
    private static EnumMap<Ulaz,Kategorija> kategorije =
      new EnumMap<Ulaz,Kategorija>(Ulaz.class);
    static {
      for(Kategorija c : Kategorija.class.getEnumConstants())
        for(Ulaz type : c.vrednosti)
          kategorije.put(type, c);
    }
    public static Kategorija kategorizuj(Ulaz ulaz) {
      return kategorije.get(ulaz);
    }
  }

  public class AutomatZaMaloprodaju {
    private static Stanje stanje = Stanje.MIRUJE;
    private static int iznos = 0;
    private static Ulaz izbor = null;
    enum TrajanjeStanja { PROLAZNO } // Nabrojani tip
  // Kao oznaka
    enum Stanje {
      MIRUJE {
        void next(Ulaz ulaz) {
          switch(Kategorija.kategorizuj(ulaz)) {
            case NOVAC:
              iznos += ulaz.iznos();
              stanje = PRIMANJE_NOVCA;
              break;
            case UGASI:
              stanje = TERMINAL;
            default:
          }
        }
      },
      PRIMANJE_NOVCA {
        void next(Ulaz ulaz) {
          switch(Kategorija.kategorizuj(ulaz)) {
            case NOVAC:
              iznos += ulaz.iznos();
              break;
            case IZBOR_ARTIKLA:
              izbor = ulaz;
              if(iznos < izbor.iznos())
                print("Nedovoljno novca za " + izbor);
              else stanje = OBRACUN;
              break;
            case PREKINI_TRANSAKCIJU:
              stanje = VRACANJE_KUSURA;
              break;
```

```
          case UGASI:
            stanje = TERMINAL;
          default:
        }
      }
    },
    OBRACUN(TrajanjeStanja.PROLAZNO) {
      void next() {
        print("uzmite kupljeno: " + izbor);
        iznos -= izbor.iznos();
        stanje = VRACANJE_KUSURA;
      }
    },
    VRACANJE_KUSURA(TrajanjeStanja.PROLAZNO) {
      void next() {
        if(iznos > 0) {
          print("Vaš kusur: " + iznos);
          iznos = 0;
        }
        stanje = MIRUJE;
      }
    },
    TERMINAL { void output() { print("Zaustavljen"); } };
    private boolean stanjeProlazno = false;
    Stanje() {}
    Stanje(TrajanjeStanja trans) { stanjeProlazno = true; }
    void next(Ulaz ulaz) {
      throw new RuntimeException("Pozivajte " +
        "next(Ulaz ulaz) samo za neprolazna stanja");
    }
    void next() {
      throw new RuntimeException("Metodu next() pozivajte za " +
        "stanja TrajanjeStanja.PROLAZNO");
    }
    void output() { print(iznos); }
  }
  static void run(Generator<Ulaz> gen) {
    while(stanje != Stanje.TERMINAL) {
      stanje.next(gen.next());
      while(stanje.stanjeProlazno)
        stanje.next();
      stanje.output();
    }
  }
  public static void main(String[] args) {
    Generator<Ulaz> gen = new RandomInputGenerator();
    if(args.length == 1)
      gen = new FileInputGenerator(args);
    run(gen);
  }
```

```
}

// Jednostavna provera ispravnosti:
class RandomInputGenerator implements Generator<Ulaz> {
  public Ulaz next() { return Ulaz.randomSelection(); }
}

// Napravi ulaze od dat. znakovnih nizova razdvojenih znakom ';':
class FileInputGenerator implements Generator<Ulaz> {
  private Iterator<String> ulaz;
  public FileInputGenerator(String imeDatoteke) {
    ulaz = new TextFile(imeDatoteke, ";").iterator();
  }
  public Ulaz next() {
    if(!ulaz.hasNext())
      return null;
    return Enum.valueOf(Ulaz.class, ulaz.next().trim());
  }
} /* Ispis:
25
50
75
uzmite kupljeno: CIPS
0
100
200
uzmite kupljeno: ZUBNAPASTA
0
25
35
Vaš kusur: 35
0
25
35
Nedovoljno novca za SOK
35
60
70
75
Nedovoljno novca za SOK
75
Vaš kusur: 75
0
Zaustavljen
*///:~
```

Pošto se **enum** instanca najčešće bira pomoću naredbe **switch** (obratite pažnju na dodatni trud uložen u to da se naredba **switch** može lako upotrebljavati s nabrojanim tipovima), jedno od najčešćih pitanja u vezi sa upotrebom nabrojanih tipova jeste: „Po kojem

osnovu da biram?" U navedenom primeru je najlakše krenuti unazad od klase **AutomatZaMaloprodaju**. U svakom stanju treba izabrati jednu od osnovnih kategorija ulaznog dejstva: primanje novca, izbor artikla, odustajanje od transakcije, i isključenje mašine. Međutim, unutar tih kategorija imate različite novčane apoene koje kupac može ubaciti u mašinu i različite artikle koje može izabrati. Nabrojani tip **Kategorija** grupiše različite tipove ulaza tako da metoda **kategorizuj( )** može unutar naredbe **switch** da proizvede odgovarajuću kategoriju. Ta metoda delotvorno i bezbedno obavlja pretraživanje pomoću mape **EnumMap**.

Ako proučite klasu **AutomatZaMaloprodaju**, videćete da se sva stanja razlikuju i da drugačije reaguju na ulaze. Obratite pažnju i na dva prelazna stanja; u metodi **run( )** mašina čeka na neki **Ulaz** i ne prestaje da menja stanja dok ne izađe iz prelaznog stanja.

AutomatZaMaloprodaju možemo ispitati na dva načina, pomoću dva različita objekta tipa **Generator**. **RandomInputGenerator** stalno proizvodi nove ulaze, sve sem ulaza **UGASI**. Ukoliko dovoljno dugo izvršavate takav program, donekle ćete proveriti ispravnost, tj. videćete da li je mašina sklona da odluta u loše stanje. **FileInputGenerator** prima datoteku koja u tekstualnom obliku opisuje ulaze, pretvara ih u instance nabrojanog tipa i pravi objekte tipa **Ulaz**. Evo tekstualne datoteke koja proizvodi prethodno prikazane rezultate:

```
//:! nabrojani/UlazUAutomatZaMaloprodaju.txt
FRTALJ; FRTALJ; FRTALJ; CIPS;
DINAR; DINAR; ZUBNAPASTA;
FRTALJ; DESETOPARAC; ODUSTANI_OD_TRANSAKCIJE;
FRTALJ; DESETOPARAC; SOK;
FRTALJ; DESETOPARAC; PETOPARAC; SOK;
ODUSTANI_OD_TRANSAKCIJE;
STOP;
///:-
```

Jedno od ograničenja ovog dizajna jeste to što polja klase **AutomatZaMaloprodaju** kojima pristupaju instance nabrojanog tipa **Stanje** *moraju* biti statična, što znači da možete imati samo jednu instancu klase **AutomatZaMaloprodaju**. To vas neće mnogo zabrinuti ako pomislite na stvarnu (ugrađenu Java) realizaciju, pošto ćete verovatno imati samo jednu aplikaciju po automatu.

Vežba 10: (7) Izmenite (samo) klasu **AutomatZaMaloprodaju** koristeći **EnumMap**, tako da program može imati više instanci klase **AutomatZaMaloprodaju**.

Vežba 11: (7) U pravom automatu za maloprodaju trebalo bi da se mogu lako dodavati i menjati vrste artikala koji se prodaju, pa su ograničenja koja nabrojan tip nameće na **Ulaz** nepraktična (pošto **enum** definiše konačan i nepromenljiv skup tipova). Izmenite program **AutomatZaMaloprodaju.java** tako da artikle koji se prodaju predstavlja klasa umesto što su deo **Ulaza**, i pomoću tekstualne datoteke inicijalizujte **ArrayList** tih objekata. (Upotrebite **net.mindview.util.TextFile**.)

Projekat: Projektujte međunarodni automat za maloprodaju (koristeći „internacionalizaciju"), tako da se ista mašina može lako prilagoditi za različite zemlje.

Višekratno otkrivanje tipa

Kada više tipova ulazi u međusobnu interakciju, program se lako zapetlja. Na primer, razmotrite sistem koji raščlanjuje i izračunava matematičke izraze. Hteli biste da kažete **Broj.plus(Broj)**, **Broj.pomnozi(Broj)** itd., gde je **Broj** osnovna klasa neke porodice numeričkih objekata. Ali kada kažete **a.plus(b)**, a ne znate tačan tip ni od **a** ni od **b**, kako će oni ući u interakciju?

Odgovor počinje nečim o čemu verovatno nikada niste razmišljali: Java obavlja samo *jednokratno otkrivanje tipa* (engl. *single dispatching*). Drugim rečima, ukoliko obavljate operaciju s više objekata nepoznatih tipova, Java može da pozove mehanizam dinamičkog povezivanja samo za jedan od njih. Time se ne bi rešio prethodno opisani problem, pa neke tipove moramo sami (ručno) da otkrijemo i tako napravimo sopstveno dinamičko povezivanje.

Rešenje se naziva *višekratno otkrivanje tipa* (engl. *multiple dispatching*). (U ovom slučaju biće samo dva otkrivanja – to je *dvokratno otkrivanje tipa.*) Polimorfizam je moguć samo preko poziva metoda, pa ako hoćete dvokratno otkrivanje tipa, morate imati dva poziva metode: prvi da otkrijete prvi nepoznati tip, i drugi da otkrijete drugi nepoznati tip. Pri višekratnom otkrivanju tipa, morate imati po jedan virtuelni poziv za svaki od nepoznatih tipova – ukoliko imate posla s dve različite hijerarhije tipova koji su u interakciji, u svakoj hijerarhiji vam treba po jedan virtuelni poziv. Po pravilu, treba napraviti konfiguraciju tako da jedan poziv metode proizvodi više virtuelnih poziva metoda i time opslužuje više tipova. To ćete postići pomoću više metoda: za svako otkrivanje tipa treba vam jedan poziv metode. U narednom primeru (koji realizuje igru „papir, kamen, makaze" koja se izvorno zove *RoShamBo*) metode su nazvane **takmicenje( )** i **izrc( )** i obe su članovi istog tipa. One proizvode jedan od tri moguća ishoda:[3]

```
//: nabrojani/Ishod.java
package nabrojani;
public enum Ishod { POBEDA, PORAZ, NERESENO } ///:~

//: nabrojani/RoShamBo1.java
// Primer višekratnog otkrivanja tipa.
package nabrojani;
import java.util.*;
import static nabrojani.Ishod.*;

interface Stavka {
  Ishod takmicenje(Stavka st);
  Ishod izrc(Papir p);
  Ishod izrc(Makaze m);
  Ishod izrc(Kamen k);
}

class Papir implements Stavka {
```

[3] Ovaj primer je više godina stajao napisan na C++-u i Javi (u knjizi *Thinking in Patterns*) na adresi *www.MindView.net* a potom je, bez spominjanja autora, naveden u knjizi drugih pisaca.

```
  public Ishod takmicenje(Stavka st) { return st.izrc(this); }
  public Ishod izrc(Papir p) { return NERESENO; }
  public Ishod izrc(Makaze m) { return POBEDA; }
  public Ishod izrc(Kamen k) { return PORAZ; }
  public String toString() { return "Papir"; }
}

class Makaze implements Stavka {
  public Ishod takmicenje(Stavka st) { return st.izrc(this); }
  public Ishod izrc(Papir p) { return PORAZ; }
  public Ishod izrc(Makaze m) { return NERESENO; }
  public Ishod izrc(Kamen k) { return POBEDA; }
  public String toString() { return "Makaze"; }
}

class Kamen implements Stavka {
  public Ishod takmicenje(Stavka st) { return st.izrc(this); }
  public Ishod izrc(Papir p) { return POBEDA; }
  public Ishod izrc(Makaze m) { return PORAZ; }
  public Ishod izrc(Kamen k) { return NERESENO; }
  public String toString() { return "Kamen"; }
}

public class RoShamBo1 {
  static final int VELICINA = 20;
  private static Random slucajan = new Random(47);
  public static Stavka newItem() {
    switch(slucajan.nextInt(3)) {
      default:
      case 0: return new Makaze();
      case 1: return new Papir();
      case 2: return new Kamen();
    }
  }
  public static void match(Stavka a, Stavka b) {
    System.out.println(
      a + " u odnosu na " + b + ": " +  a.takmicenje(b));
  }
  public static void main(String[] args) {
    for(int i = 0; i < VELICINA; i++)
      match(newItem(), newItem());
  }
} /* Ispis:
Kamen u odnosu na Kamen: NERESENO
Papir u odnosu na Kamen: POBEDA
Papir u odnosu na Kamen: POBEDA
Papir u odnosu na Kamen: POBEDA
Makaze u odnosu na Papir: POBEDA
Makaze u odnosu na Makaze: NERESENO
Makaze u odnosu na Papir: POBEDA
Kamen u odnosu na Papir: PORAZ
```

```
Papir u odnosu na Papir: NERESENO
Kamen u odnosu na Papir: PORAZ
Papir u odnosu na Makaze: PORAZ
Papir u odnosu na Makaze: PORAZ
Kamen u odnosu na Makaze: POBEDA
Kamen u odnosu na Papir: PORAZ
Papir u odnosu na Kamen: POBEDA
Makaze u odnosu na Papir: POBEDA
Papir u odnosu na Makaze: PORAZ
Papir u odnosu na Makaze: PORAZ
Papir u odnosu na Makaze: PORAZ
Papir u odnosu na Makaze: PORAZ
*///:~
```

Stavka je interfejs za tipove koji će biti višekratno otkrivani. **RoShamBo1.match( )** prima dva objekta tipa **Stavka** i postupak dvokratnog otkrivanja tipa otpočinje pozivom funkcije **Stavka.takmicenje( )**. Virtuelni mehanizam određuje tip od **a**, pa se budi unutar funkcije **takmicenje( )** konkretnog tipa od **a**. Funkcija **takmicenje( )** obavlja drugo otkrivanje tipa pozivom metode **izrc( )** za preostali tip. Prosleđivanjem sebe (**this**) kao argumenta metode **izrc( )** proizvodi poziv preklopljene funkcije **izrc( )**, pa ostaje sačuvana informacija o tipu dobijena u prvom otkrivanju tipa. Kada se dovrši drugo otkrivanje tipa, znamo tačan tip oba objekta tipa **Stavka**.

Programiranje višekratnog otkrivanja tipa zahteva veliku ceremoniju, ali ne gubite iz vida dobijenu sintaksičku eleganciju samog poziva – umesto da pišete nezgrapan kôd za utvrđivanje tipa jednog ili više objekata tokom nekog poziva, jednostavno kažete: „Vas dvoje! Ne zanima me koji su vaši tipovi, samo uđite propisno u međusobnu interakciju!" Međutim, neka vam ta vrsta elegancije postane važna pre nego što krenete na višekratno otkrivanje tipa.

Otkrivanje tipa pomoću nabrojanih tipova

Prosto prevođenje programa **RoShamBo1.java** u rešenje čiju osnovu čine nabrojani tipovi nije jednostavno, jer **enum** instance nisu tipovi, pa preklopljene metode **izrc( )** neće raditi – **enum** instance ne možete upotrebljavati kao tipove u argumentu. Međutim, realizacija višekratnog otkrivanja tipa pomoću nabrojanih tipova može se obaviti na više različitih načina.

U jednom načinu upotrebljava se konstruktor za inicijalizaciju svake **enum** instance celim „redom" ishoda; zajedno uzev, time se dobija svojevrsna tabela za pronalaženje:

```
//: nabrojani/RoShamBo2.java
// Izbor konstante jednog nabrojanog tipa
// korišćenjem drugog u naredbi switch.
package nabrojani;
import static nabrojani.Ishod.*;

public enum RoShamBo2 implements Takmicar<RoShamBo2> {
  PAPIR(NERESENO, PORAZ, POBEDA),
  MAKAZE(POBEDA, NERESENO, PORAZ),
```

```
    KAMEN(PORAZ, POBEDA, NERESENO);
    private Ishod vPAPIR, vMAKAZE, vKAMEN;
    RoShamBo2(Ishod papir,Ishod makaze,Ishod kamen) {
      this.vPAPIR = papir;
      this.vMAKAZE = makaze;
      this.vKAMEN = kamen;
    }
    public Ishod takmicenje(RoShamBo2 it) {
      switch(it) {
        default:
        case PAPIR: return vPAPIR;
        case MAKAZE: return vMAKAZE;
        case KAMEN: return vKAMEN;
      }
    }
    public static void main(String[] args) {
      RoShamBo.igraj(RoShamBo2.class, 20);
    }
} /* Ispis:
KAMEN u odnosu na KAMEN: NERESENO
MAKAZE u odnosu na KAMEN: PORAZ
MAKAZE u odnosu na KAMEN: PORAZ
MAKAZE u odnosu na KAMEN: PORAZ
PAPIR u odnosu na MAKAZE: PORAZ
PAPIR u odnosu na PAPIR: NERESENO
PAPIR u odnosu na MAKAZE: PORAZ
KAMEN u odnosu na MAKAZE: POBEDA
MAKAZE u odnosu na MAKAZE: NERESENO
KAMEN u odnosu na MAKAZE: POBEDA
MAKAZE u odnosu na PAPIR: POBEDA
MAKAZE u odnosu na PAPIR: POBEDA
KAMEN u odnosu na PAPIR: PORAZ
KAMEN u odnosu na MAKAZE: POBEDA
MAKAZE u odnosu na KAMEN: PORAZ
PAPIR u odnosu na MAKAZE: PORAZ
MAKAZE u odnosu na PAPIR: POBEDA
MAKAZE u odnosu na PAPIR: POBEDA
MAKAZE u odnosu na PAPIR: POBEDA
MAKAZE u odnosu na PAPIR: POBEDA
*///:~
```

Nakon što metoda **takmicenje( )** otkrije oba tipa, jedino dejstvo je vraćanje rezultujućeg objekta tipa **Ishod**. Međutim, mogli biste pozvati i neku drugu metodu, čak (na primer) preko objekta tipa *Command* dodeljenom u konstruktoru.

RoShamBo2.java je mnogo manji i jednostavniji od prvobitnog primera, te ga je lakše i pratiti. Vodite računa o tome da i dalje koristimo dva otkrivanja tipa da bismo utvrdili tip oba objekta. U programu **RoShamBo1.java** oba otkrivanja su obavljena preko virtuelnih poziva metoda, ali ovde je virtuelni poziv metode upotrebljen samo u prvom otkrivanju tipa. U drugom otkrivanju tipa upotrebljen je **switch**, ali sve je bezbedno zato što **enum** ograničava broj izbora u naredbi **switch**.

Kod koji upravlja nabrojanim tipom tako je raščlanjen da ga možemo koristiti u drugim primerima. Prvo, interfejs **Takmicar** definiše tip koji se takmiči s drugim **Takmicar**om:

```
//: nabrojani/Takmicar.java
// Izbor konstante jednog nabrojanog tipa
// korišćenjem drugog u naredbi switch.
package nabrojani;

public interface Takmicar<T extends Takmicar<T>> {
  Ishod takmicenje(T takmicar);
} ///:~
```

Zatim definišemo dve statične metode (statične zato da ne bismo morali eksplicitno zadati parametarski tip). Prvo **match( )** poziva **takmicenje( )** za jedan objekat tipa **Takmicar** protiv drugog, pa vidite kako je u ovom slučaju dovoljno da parametarski tip bude **Takmicar<T>**. Ali u metodi **igraj( )**, parametarski tip mora biti **Enum<T>** jer se koristi u metodi **NabrojaniTipovi.random( )**, i **Takmicar<T>**, zato što se prosleđuje metodi **match( )**:

```
//: nabrojani/RoShamBo.java
// Zajedničke alatke za RoShamBo primere.
package nabrojani;
import net.mindview.util.*;

public class RoShamBo {
  public static <T extends Takmicar<T>>
  void match(T a, T b) {
    System.out.println(
      a + " u odnosu na " + b + ": " +  a.takmicenje(b));
  }
  public static <T extends Enum<T> & Takmicar<T>>
  void igraj(Class<T> rsbKlasa, int velicina) {
    for(int i = 0; i < velicina; i++)
      match(
        NabrojaniTipovi.random(rsbKlasa),
          NabrojaniTipovi.random(rsbKlasa));
  }
} ///:~
```

Metoda **igraj( )** nema povratnu vrednost koja obuhvata parametar tipa T, pa izgleda kao da bismo mogli upotrebiti džokerske argumente unutar tipa **Class<T>** umesto što koristimo opis vodećih parametara. Međutim, džokerski argumenti se ne mogu protezati na više od jednog osnovnog tipa, pa moramo da koristimo gornji izraz.

Korišćenje metoda koje se menjaju u zavisnosti od konstante nabrojanog tipa

Pošto metode koje se menjaju u zavisnosti od konstante nabrojanog tipa omogućuju pravljenje različitih realizacija metoda za svaku **enum** instancu, možda vam izgleda da su

one savršeno rešenje za višekratno otkrivanje tipa. Ali iako na taj način mogu dobiti drugačija ponašanja, **enum** instance nisu tipovi, pa ih ne možete koristiti kao argumente tipova u potpisima metoda. U ovom primeru upotrebljavate naredbu **switch**, i to je najviše što se može uraditi:

```
//: nabrojani/RoShamBo3.java
// Korišćenje metoda koje se menjaju u zavisnosti od konstante
// (nabrojanog tipa).
package nabrojani;
import static nabrojani.Ishod.*;

public enum RoShamBo3 implements Takmicar<RoShamBo3> {
  PAPIR {
    public Ishod takmicenje(RoShamBo3 it) {
      switch(it) {
        default: // Da se prevodilac ne bi bunio
        case PAPIR: return NERESENO;
        case MAKAZE: return PORAZ;
        case KAMEN: return POBEDA;
      }
    }
  },
  MAKAZE {
    public Ishod takmicenje(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPIR: return POBEDA;
        case MAKAZE: return NERESENO;
        case KAMEN: return PORAZ;
      }
    }
  },
  KAMEN {
    public Ishod takmicenje(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPIR: return PORAZ;
        case MAKAZE: return POBEDA;
        case KAMEN: return NERESENO;
      }
    }
  };
  public abstract Ishod takmicenje(RoShamBo3 it);
  public static void main(String[] args) {
    RoShamBo.igraj(RoShamBo3.class, 20);
  }
} /* Isti ispis kao u RoShamBo2.java *///:~
```

Iako ovo radi, a nije ni glupo, rešenje u programu **RoShamBo2.java** kao da zahteva manje koda za dodavanje novog tipa, te izgleda jednostavnije.

Međutim, **RoShamBo3.java** možemo pojednostaviti i sažeti:

```
//: nabrojani/RoShamBo4.java
package nabrojani;

public enum RoShamBo4 implements Takmicar<RoShamBo4> {
  KAMEN {
    public Ishod takmicenje(RoShamBo4 protivnik) {
      return takmicenje(MAKAZE, protivnik);
    }
  },
  MAKAZE {
    public Ishod takmicenje(RoShamBo4 protivnik) {
      return takmicenje(PAPIR, protivnik);
    }
  },
  PAPIR {
    public Ishod takmicenje(RoShamBo4 protivnik) {
      return takmicenje(KAMEN, protivnik);
    }
  };
  Ishod takmicenje(RoShamBo4 gubitnik, RoShamBo4 protivnik) {
    return ((protivnik == this) ? Ishod.NERESENO
        : ((protivnik == gubitnik) ? Ishod.POBEDA
                                   : Ishod.PORAZ));
  }
  public static void main(String[] args) {
    RoShamBo.igraj(RoShamBo4.class, 20);
  }
} /* Isti ispis kao u RoShamBo2.java *///:~
```

Ovde se drugo otkrivanje tipa obavlja verzijom metode **takmicenje**( ) s dva argumenta; metoda obavlja niz poređenja i stoga radi slično naredbi **switch**. Program je manji, ali i manje jasan. U velikom sistemu ta nejasnost može imati strašne posledice.

Otkrivanje tipa pomoću mapa **EnumMap**

„Pravo" dvokratno otkrivanje tipa možemo obaviti pomoću klase **EnumMap** koja veoma efikasno radi s nabrojanim tipovima. Pošto je naš cilj pravljenje izbora na osnovu dva nepoznata tipa, za dvokratno otkrivanje tipa možemo upotrebiti **EnumMap** čiji su članovi drugi objekti tipa **EnumMap**:

```
//: nabrojani/RoShamBo5.java
// Višekratno otkrivanje tipa pomoću mape EnumMap
// čiji su članovi drugi objekti tipa EnumMap.
package nabrojani;
import java.util.*;
```

```
import static nabrojani.Ishod.*;

enum RoShamBo5 implements Takmicar<RoShamBo5> {
  PAPIR, MAKAZE, KAMEN;
  static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Ishod>>
    tabela = new EnumMap<RoShamBo5,
      EnumMap<RoShamBo5,Ishod>>(RoShamBo5.class);
  static {
    for(RoShamBo5 it : RoShamBo5.vrednosti())
      tabela.put(it,
        new EnumMap<RoShamBo5,Ishod>(RoShamBo5.class));
    inicReda(PAPIR, NERESENO, PORAZ, POBEDA);
    inicReda(MAKAZE, POBEDA, NERESENO, PORAZ);
    inicReda(KAMEN, PORAZ, POBEDA, NERESENO);
  }
  static void inicReda(RoShamBo5 it,
    Ishod vPAPIR, Ishod vMAKAZE, Ishod vKAMEN) {
    EnumMap<RoShamBo5,Ishod> red =
      RoShamBo5.tabela.get(it);
    red.put(RoShamBo5.PAPIR, vPAPIR);
    red.put(RoShamBo5.MAKAZE, vMAKAZE);
    red.put(RoShamBo5.KAMEN, vKAMEN);
  }
  public Ishod takmicenje(RoShamBo5 it) {
    return tabela.get(this).get(it);
  }
  public static void main(String[] args) {
    RoShamBo.igraj(RoShamBo5.class, 20);
  }
} /* Isti ispis kao u RoShamBo2.java *///:~
```

EnumMap se inicijalizuje u statičnoj odredbi; struktura poziva metode **inicReda( )** nalik je na tabelu. Obratite pažnju na metodu **takmicenje( )**, gde se oba otkrivanja tipa obavljaju u istoj naredbi.

Korišćenje 2-D niza

Rešenje možemo još pojednostaviti kada uočimo da svaka **enum** instanca ima fiksnu vrednost (dobijenu na osnovu „rednog broja" njenog deklarisanja) i da metoda **ordinal( )** proizvodi tu vrednost. Najmanje i najjednostavnije rešenje (a možda i najbrže, mada ne treba zaboraviti da **EnumMap** upotrebljava interni niz) daje dvodimenzionalni niz koji takmičare preslikava u ishode:

```
//: nabrojani/RoShamBo6.java
// Nabrojani tipovi i "tabele" umesto
// višekratnog otkrivanja tipa.
package nabrojani;
import static nabrojani.Ishod.*;

enum RoShamBo6 implements Takmicar<RoShamBo6> {
```

```
    PAPIR, MAKAZE, KAMEN;
    private static Ishod[][] tabela = {
      { NERESENO, PORAZ, POBEDA }, // PAPIR
      { POBEDA, NERESENO, PORAZ }, // MAKAZE
      { PORAZ, POBEDA, NERESENO }, // KAMEN
    };
    public Ishod takmicenje(RoShamBo6 drugi) {
      return tabela[this.ordinal()][drugi.ordinal()];
    }
    public static void main(String[] args) {
      RoShamBo.igraj(RoShamBo6.class, 20);
    }
  } ///:~
```

Tabela ima tačno onaj poredak kao pozivi metode **inicReda( )** u prethodnom primeru.

Ovaj kratak kôd deluje mnogo privlačnije od prethodnih primera, delom zato što izgleda mnogo razumljiviji i izmenljiviji, ali i zato što bi se reklo da je jednostavniji. Međutim, nije baš onoliko „bezbedan" kao prethodni primeri, zato što se upotrebljava niz. Kada je niz veći, lakše je pogrešiti u veličini, i ako ispitivanje ne obuhvati sve mogućnosti, nešto bi moglo da procuri.

Sva ova rešenja predstavljaju različite vrste tabela, ali ih vredi istražiti da bismo pronašli najprikladniju. Vodite računa o sledećem: iako je poslednje rešenje najkompaktnije, ono je i prilično neprilagodljivo, zato što može da proizvede samo konstantan izlaz za date konstantne ulaze. Međutim, ništa vas ne sprečava da pomoću takve **tabele** napravite funkcijski objekat. Pri određenim vrstama problema, koncept „koda upravljanog tabelom" ume veoma dobro da posluži.

Sažetak

Iako nabrojani tipovi sami po sebi nisu previše složeni, ovo poglavlje sam pomerio u drugi deo knjige zbog onoga što se može uraditi kombinacijom nabrojanih tipova i polimorfizma, generičkih tipova i refleksije.

Mada znatno sofisticiraniji nego u jezicima C i C++, nabrojani tipovi i dalje predstavljaju „malu" mogućnost, nešto bez čega je jezik opstajao (pomalo nezgrapno) mnogo godina. Ovo poglavlje pokazuje kako može biti važan doprinos „male" mogućnosti – katkada nam daje baš onu polugu kojom problem možemo rešiti elegantno i jasno, a u celoj knjizi sam isticao koliko je elegancija važna. Jasnoća može biti činilac koji određuje da li je rešenje uspešno ili nije dobro zato što ostali ne mogu da ga razumeju.

Što se tiče jasnoće, Java 1.0 je proizvela zbrku zbog korišćenja termina „enumeration" umesto uobičajenog i prihvaćenog termina „iterator" za objekat koji bira svaki element neke sekvence. U nekim jezicima čak se i nabrojani tipovi nazivaju „enumerators"! Ta greška je ispravljena u Javi, ali interfejs **Enumeration** nisu smeli samo tako da izbace, pa se još viđa u starim (a ponekad i u novim!) programima, biblioteci i dokumentaciji.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Anotacije

Anotacije (ili metapodaci) formalizovan su način dodavanja informacija kodu, što olakšava kasniju upotrebu tih podataka.[1]

ANOTACIJE SU DELOM POSLEDICA OPŠTEG TRENDA KOMBINOVANJA METAPODATAKA I datoteka izvornog koda, što je bolje nego da te metapodatke čuvamo u spoljnim dokumentima. Unete su u Javu zato što nešto slično postoji i u drugim jezicima, C#-u, na primer.

Anotacije su jedna od osnovnih jezičkih promena u Javi SE5. U njih se stavljaju informacije koje ne mogu biti izražene Javom, a potrebne su za potpuno opisivanje programa. Dakle, anotacije omogućuju skladištenje dodatnih informacija o programu u formatu koji ispituje i proverava prevodilac. Anotacije se mogu upotrebiti za generisanje opisnih datoteka ili čak definicija novih klasa, čime olakšavaju teret pisanja „šablonskog" koda. Pomoću anotacija, te metapodatke možete zadržati u Javinom izvornom kodu, i uz to imate čistiji kôd, mogućnost provere tipova u vreme prevođenja i pomoć odgovarajućeg API-ja u pravljenju alatki za obradu sopstvenih anotacija. Iako Java SE5 ima nekoliko unapred definisanih tipova metapodataka, po pravilu samo od vas zavisi kakve anotacije ćete dodavati i šta ćete s njima raditi.

Sintaksa anotacija je prilično jednostavna i svodi se uglavnom na dodavanje simbola @ u jezik. Java SE5 sadrži tri ugrađene anotacije opšte namene, definisane u paketu **java.lang**:

- **@Override** pokazuje da definicija neke metode redefiniše metodu osnovne klase. Prevodilac će generisati grešku ukoliko slučajno pogrešno napišete ime te metode ili joj date netačan potpis.[2]
- **@Deprecated**, zbog koje prevodilac generiše upozorenje ukoliko taj element bude upotrebljen.
- **@SuppressWarnings**, za isključivanje neprikladnih upozorenja prevodioca. U ranijim izdanjima Jave SE5 dozvoljavala se ova anotacija, ali nije bila podržavana (nego ignorisana).

Za pravljenje novih anotacija služe četiri druga tipa anotacija koje ću opisati u ovom poglavlju.

Kad god pišete opisne klase ili interfejse s mnogo ponavljanja, taj posao možete da automatizujete i pojednostavite pomoću anotacija. Primera radi, dobar deo dodatnog posla u *Enterprise JavaBeans*, EJBs, više se ne mora raditi otkad u EJB3.0 postoje anotacije.

Anotacije mogu da zamene postojeće sisteme kao što je XDoclet – alatka nezavisnog proizvođača za doclete (videti dodatak na adresi *http://MindView.net/Books/BetterJava*) projektovana baš za pravljenje docleta u stilu anotacija. Za razliku od docleta, anotacije su

---

[1] Jeremy Meyer je došao u Crested Butte i dve sedmice radio sa mnom na ovom poglavlju. Njegova pomoć je neprocenjiva.

[2] Nema sumnje da je nadahnuće za ovo bila slična mogućnost jezika C#. U C#-u ona je rezervisana reč čiju upotrebu nameće prevodilac, a u Javi je anotacija. Drugim rečima, kada na C#-u redefinišete metodu, morate upotrebiti rezervisanu reč **override**, dok na Javi upotreba anotacije **@Override** nije obavezna.

prave jezičke konstrukcije, imaju strukturu, a njihovi tipovi se proveravaju u vreme prevođenja. Kôd je elegantniji i lakše se održava ukoliko su sve informacije zadržane u izvornom kodu, a ne u komentarima. Korišćenjem i nasleđivanjem API-ja i alatki za anotacije, ili pomoću spoljnih biblioteka za rad s bajtkodom koje ćete upoznati u ovom poglavlju, možete obaviti dalekosežnu proveru i obradu izvornog koda i bajtkoda.

Osnovna sintaksa

U donjem primeru, metoda **probnoIzvrsenje( )** ima anotaciju **@Test**. Sama po sebi ona ne radi ništa, ali će prevodilac proveriti imate li definiciju anotacije **@Test** u putanji automatskog prevođenja i pakovanja (builda). Kao što ćete videti u nastavku poglavlja, pomoću refleksije možete napraviti alatku koja će tu metodu izvršavati.

```
//: anotacije/MozeSeTestirati.java
package anotacije;
import net.mindview.atunit.*;

public class MozeSeTestirati {
  public void izvrsi() {
    System.out.println("Izvršavam..");
  }
  @Test void probnoIzvrsenje() { izvrsi(); }
} ///:~
```

Anotirane metode se ne razlikuju od ostalih metoda. Anotaciju **@Test** iz prethodnog primera, možete koristiti u kombinaciji sa svim modifikatorima kao što su **public** ili **static** ili **void**. Sintaksički, anotacije se upotrebljavaju veoma slično modifikatorima.

Definisanje anotacije

Sledi definicija gornje anotacije. Videćete da definicije anotacija mogo liče na definicije interfejsa. Zapravo, prevodilac od njih pravi datoteke klasa kao od svakog drugog Java interfejsa:

```
//: net/mindview/atunit/Test.java
// Oznaka @Test.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} ///:~
```

Sem simbola @, definicija oznake **@Test** liči na prazan interfejs. Definicija anotacije zahteva i *metaanotacije* **@Target** i **@Retention**. **@Target** definiše gde se anotacija može primeniti (recimo, ispred metode ili polja). **@Retention** definiše da li će anotacija biti dostupna u izvornom kodu (**SOURCE**), u datotekama klasa (**CLASS**) ili u vreme izvršavanja (**RUNTIME**).

Anotacije najčešće sadrže *elemente* za zadavanje vrednosti u anotacijama korisnika. Program ili alatka mogu upotrebljavati te parametre prilikom obrade anotacije korisnika. Elementi liče na metode interfejsa, sem što im možete definisati podrazumevane vrednosti.

Anotaciju bez elemenata, kao što je **@Test**, nazivamo *marker anotacija.*

Sledi jednostavna anotacija koja prati slučajeve upotrebe u projektu. Programeri anotiraju svaku metodu ili skup metoda koje zadovoljavaju zahteve određenog slučaja upotrebe. Kad prebroji realizovane slučajeve upotrebe, rukovodilac projekta može steći uvid u napredovanje projekta, a programeri koji održavaju projekat lako pronalaze slučajeve upotrebe kada treba da ažuriraju poslovna pravila u sistemu ili da otkriju i otklone greške u njima.

```
//: anotacije/SlucajUpotrebe.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SlucajUpotrebe {
  public int id();
  public String opis() default "nema opisa";
} ///:~
```

Obratite pažnju na to da **id** i **opis** liče na deklaracije metoda. Pošto prevodilac proverava tip od **id**, to je pouzdan način povezivanja baze podataka za praćenje s dokumentom slučaja upotrebe i sa izvornim kodom. Element **opis** ima podrazumevanu vrednost koju će procesor anotacija upotrebiti ukoliko neka druga vrednost ne bude zadata prilikom anotiranja metode.

U sledećoj klasi tri metode su anotirane kao slučajevi upotrebe:

```
//: anotacije/UsluzneMetodeZaLozinke.java
import java.util.*;

public class UsluzneMetodeZaLozinke {
  @SlucajUpotrebe(id = 47, opis =
  "Lozinka mora sadržati barem jednu brojku")
  public boolean proveriLozinku(String lozinka) {
    return (lozinka.matches("\\w*\\d\\w*"));
  }
  @SlucajUpotrebe(id = 48)
  public String sifrirajLozinku(String lozinka) {
   return new StringBuilder(lozinka).reverse().toString();
  }
  @SlucajUpotrebe(id = 49, opis =
  "Nova lozinka ne može biti jednaka prethodnim")
  public boolean proveraNeponavljanjaLozinke(
    List<String> prethodneLozinke, String lozinka) {
    return !prethodneLozinke.contains(lozinka);
  }
} ///:~
```

Vrednosti elemenata anotacije izražene su kao parovi ime–vrednost u zagradama iza deklaracije anotacije **@SlucajUpotrebe**. Anotaciji metode **sifrirajLozinku( )** ovde nije bila prosleđena vrednost za element **opis**, pa će se podrazumevana vrednost definisana u **@interface SlucajUpotrebe** pojaviti kada ta klasa prođe kroz procesor anotacija.

Ovakav sistem bi mogao „skicirati" strukturu vašeg sistema, s tim što bi mu trebalo dodati još neke funkcije.

Metaanotacije

U jeziku Java trenutno postoje samo tri (prethodno opisane) standardne anotacije i četiri definisane metaanotacije. Ovo su metaanotacije za anotiranje anotacija:

@Target	Gde se anotacija može primeniti. Mogući **ElementType** argumenti jesu: **CONSTRUCTOR**: deklaracija konstruktora **FIELD**: deklaracija polja (uključujući i **enum** konstante) **LOCAL_VARIABLE**: deklaracija lokalnih promenljivih **METHOD**: deklaracija metoda **PACKAGE**: deklaracija paketa **PARAMETER**: deklaracija parametara **TYPE**: deklaracija klasa, interfejsa (uključujući i tip anotacije) ili nabrojanih tipova
@Retention	Koliko dugo se čuvaju informacije anotacije. Mogući **RetentionPolicy** argumenti jesu: **SOURCE**: anotaciju prevodilac izbacuje. **CLASS**: prevodilac ostavlja anotaciju u datoteci klase, ali je VM može izbaciti. **RUNTIME**: VM ostavlja anotaciju u vreme izvršavanja, pa ona može biti pročitana pomoću refleksije
@Documented	Uvrsti anotaciju u Javadoc dokumente.
@Inherited	Dozvoli potklasama nasleđivanje roditeljske anotacije

Uglavnom ćete sami definisati svoje anotacije i pisati sopstvene procesore koji će ih obrađivati.

Pisanje procesora anotacija

Bez alatki koje će ih čitati, anotacije teško da su korisnije od komentara. Važan deo postupka korišćenja anotacija jeste pisanje i korišćenje *procesora anotacija*. Kao pomoć za pravljenje tih alatki, u Javi SE5 koriste se proširenja API-ja za refleksiju. Postoji i spoljna alatka **apt** za analizu (raščlanjivanje) Java izvornog koda sa anotacijama.

Sledi veoma jednostavan procesor anotacija koji čita anotiranu klasu **UsluzneMetode-ZaLozinke** i refleksijom traži oznake **@SlucajUpotrebe**. Za datu listu **id** vrednosti, ispisaće pronađene slučajeve upotrebe i prijaviti one koji nedostaju:

```
//: anotacije/PracenjeSlucajevaUpotrebe.java
import java.lang.reflect.*;
import java.util.*;
```

```
public class PracenjeSlucajevaUpotrebe {
  public static void
  pratiSlucajeveUpotrebe(List<Integer> slucajeviUpotrebe, Class<?> cl) {
    for(Method m : cl.getDeclaredMethods()) {
      SlucajUpotrebe su = m.getAnnotation(SlucajUpotrebe.class);
      if(su != null) {
        System.out.println("Pronađen slučaj upotrebe:" + su.id() +
          " " + su.opis());
        slucajeviUpotrebe.remove(new Integer(su.id()));
      }
    }
    for(int i : slucajeviUpotrebe) {
      System.out.println("Upozorenje: nedostaje slučaj upotrebe-" + i);
    }
  }
  public static void main(String[] args) {
    List<Integer> slucajeviUpotrebe = new ArrayList<Integer>();
    Collections.addAll(slucajeviUpotrebe, 47, 48, 49, 50);
    pratiSlucajeveUpotrebe(slucajeviUpotrebe, UsluzneMetodeZa
      Lozinke.class);
  }
} /* Ispis:
Pronađen slučaj upotrebe:47 Lozinka mora sadržati barem jednu brojku
Pronađen slučaj upotrebe:48 nema opisa
Pronađen slučaj upotrebe:49 Nova lozinka ne može biti jednaka prethodnim
Upozorenje: nedostaje slučaj upotrebe-50
*///:~
```

U prethodnom programu koriste se reflektivna metoda **getDeclaredMethods**( ) i metoda **getAnnotation**( ) koja potiče iz interfejsa **AnnotatedElement** (koji realizuju klase kao što su **Class**, **Method** i **Field**). Ta metoda vraća objekat anotacije specificiranog tipa, u ovom slučaju tipa **SlucajUpotrebe**. Ukoliko nema anotacija tog odredenog tipa za anotiranu metodu, vraća se **null**. Vrednosti elemenata se saznaju kada se pozovu metode **id**( ) i **opis**( ). Ne zaboravite da u anotaciji metode **sifrirajLozinku**( ) nije bilo opisa, pa gornji procesor pronalazi podrazumevanu vrednost „**nema opisa**" kada za tu anotaciju pozove metodu **opis**( ).

Elementi anotacija

Oznaka **@SlucajUpotrebe** definisana u programu **SlucajUpotrebe.java** sadrži **int** element **id** i **String** element **opis**. Ovo su dozvoljeni tipovi elemenata anotacija:

- Svi prosti tipovi (**int**, **float**, **boolean** itd.)
- **String**
- **Class**
- Nabrojani tipovi
- **Anotacije**
- Nizovi svih prethodnih tipova

Prevodilac će prijaviti grešku ako pokušate da upotrebite neki drugi tip. Vodite računa o tome da ne smete koristiti omotačke klase, ali zbog automatskog pakovanja to zapravo i nije ograničenje. Dozvoljeni su i elementi koji su sami anotacije. Kao što ćete videti malo kasnije, ugnežđene anotacije mogu veoma dobro da posluže.

Ograničenja podrazumevanih vrednosti

Prevodilac je veoma izbirljiv kada se radi o podrazumevanim vrednostima elemenata. Vrednost svakog elementa mora biti specificirana. To znači da elementi moraju imati podrazumevane vrednosti ili vrednosti zadate u klasi koja upotrebljava anotaciju.

Postoji još jedno ograničenje: elementi neprostih tipova ne smeju poprimiti vrednost **null**. To važi i za deklaracije u izvornom kodu i za definicije podrazumevanih vrednosti u interfejsu anotacije. Ovo otežava pisanje procesora koji deluje na osnovu postojanja ili nepostojanja određenog elementa, pošto je svaki element efektivno prisutan u svakoj deklaraciji anotacije. Prevazići ćete ovu poteškoću tako što ćete tražiti specifične vrednosti, kao što su prazni znakovni nizovi ili negativne vrednosti:

```
//: anotacije/SimulacijaNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulacijaNull {
  public int id() default -1;
  public String opis() default "";
} ///:~
```

Ovaj idiom je tipičan za definicije anotacija.

Generisanje spoljnih datoteka

Anotacije su naročito korisne u radu sa strukturama za razvoj koda za koje je, uz izvorni kôd korisnika, potrebna i neka vrsta dodatnih informacija. Tehnologije kao što su (pre EJB3) bila *Enterprise* zrna Jave, zahtevaju brojne interfejse i deskriptore realizacije, što je „šablonski" kôd, jednak za svako zrno. Za Web servise, biblioteke namenskih oznaka i alatke za preslikavanje objekata na relacione baze podataka, kao što su Toplink i Hibernate, često su potrebni XML deskriptori koji su spoljni u odnosu na kôd. Nakon definisanja Java klase, programer mora da istrpi dosadno ponovno specificiranje informacija kao što su ime, paket itd. – a one već postoje u originalnoj klasi. Kada koristite spoljnu deskriptor datoteku, imate dva zasebna izvora informacija o klasi, što obično prouzrokuje probleme sa sinhronizacijom. Zbog toga programeri koji rade na projektu, pored toga kako se piše Java program, moraju znati i kako da izmene deskriptor.

Pretpostavimo da pišete osnovne funkcije za preslikavanje objekata na relacione baze podataka; te funkcije automatizuju pravljenje tabele baze podataka za skladištenje zrna Jave. Mogli biste upotrebiti XML datoteku deskriptora koja specificira ime klase, njene članove i informacije o njenom preslikavanju u bazi podataka. Međutim, pomoću

anotacija, sve te informacije zadržali biste u izvornoj datoteci zrna Jave. Za to bi vam bile potrebne anotacije koje definišu ime tabele baze podataka pridružene tom zrnu, kolone i SQL tipove za preslikavanje svojstava zrna.

Sledi anotacija zrna koja procesoru anotacija kazuje da napravi tabelu baze podataka:

```
//: anotacije/bazapodataka/TabelaBP.java
package anotacije.bazapodataka;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Važi samo za klase
@Retention(RetentionPolicy.RUNTIME)
public @interface TabelaBP {
  public String ime() default "";
} ///:~
```

Svaki **ElementType** koji specificirate u anotaciji **@Target** jeste ograničenje koje prevodiocu kazuje da se ta anotacija može primeniti samo na taj određeni tip. Za nabrojani tip **ElementType** možete specificirati jednu vrednost ili listu proizvoljnih kombinacija vrednosti razdvojenih zarezima. Ako anotaciju želite da primenite na svaki **ElementType**, možete potpuno da izostavite anotaciju **@Target**, mada to nije uobičajeno.

Obratite pažnju na to da **@TabelaBP** ima element **ime( )**, tako da anotacija može dati ime tabele baze podataka koju će procesor napraviti.

Evo anotacija za polja ovog zrna Jave:

```
//: anotacije/bazapodataka/Ogranicenja.java
package anotacije.bazapodataka;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ogranicenja {
  boolean primarniKLJuc() default false;
  boolean dozvoliNull() default true;
  boolean jedinstveno() default false;
} ///:~

//: anotacije/bazapodataka/SQLString.java
package anotacije.bazapodataka;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
  int vrednost() default 0;
  String ime() default "";
  Ogranicenja ogranicenja() default @Ogranicenja;
} ///:~
```

```
//: anotacije/bazapodataka/SQLInteger.java
package anotacije.bazapodataka;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
  String ime() default "";
  Ogranicenja ogranicenja() default @Ogranicenja;
} ///:~
```

Anotacija **@Ogranicenja** omogućuje procesoru da izdvoji metapodatke o tabeli baze podataka. Oni predstavljaju tek mali podskup ograničenja koje baze podataka po pravilu nameću, ali dovoljan da steknete opštu sliku. Elementima **primarniKljuc( )**, **dozvoliNull( )** i **jedinstveno( )** date su smislene podrazumevane vrednosti, tako da korisnik anotacije u većini slučajeva ne mora mnogo da kuca.

Ostala dva interfejsa (**@interface**) definišu SQL tipove. I opet, da bi ova struktura bila upotrebljivija, treba da definišete anotaciju za svaki dodatni SQL tip. Ovde će dva tipa biti dovoljna.

Svaki od tih tipova ima element **ime( )** i element **ogranicenja( )**. Taj poslednji u ugnežđenoj anotaciji sadrži informacije o ograničenjima baze podataka za tip kolone. Obratite pažnju na to da je **@Ogranicenja** podrazumevana vrednost elementa **ogranicenja( )**. Pošto nakon ovog tipa anotacije nema u zagradama specificiranih vrednosti elemenata, podrazumevana vrednost **ogranicenja( )** zapravo je anotacija **@Ogranicenja** sa sopstvenim skupom podrazumevanih vrednosti. Da biste ugnežđenoj anotaciji **@Ogranicenja** sa jedinstvenošću postavljenom na podrazumevanu vrednost **true**, njen element možete definisati ovako:

```
//: anotacije/bazapodataka/Jedinstvenost.java
// Primer ugnežđenih anotacija
package anotacije.bazapodataka;

public @interface Jedinstvenost {
  Ogranicenja ogranicenja()
    default @Ogranicenja(jedinstveno=true);
} ///:~
```

Sledi jednostavno zrno u kojem su upotrebljene prethodne anotacije:

```
//: anotacije/bazapodataka/Clan.java
package anotacije.bazapodataka;

@TabelaBP(ime = "CLAN")
public class Clan {
  @SQLString(30) String licnoIme;
  @SQLString(50) String prezIme;
  @SQLInteger Integer starost;
  @SQLString(value = 30,
```

```
    ogranicenja = @Ogranicenja(primarniKljuc = true))
    String identifikator;
    static int brojClanova;
    public String dajIdentifikator() { return identifikator; }
    public String dajLicnoIme() { return licnoIme; }
    public String dajPrezIme() { return prezIme; }
    public String toString() { return identifikator; }
    public Integer dajStarost() { return starost; }
} ///:~
```

Anotaciji **@TabelaBP** klase data je vrednost CLAN koja će biti upotrebljena kao ime tabele. Svojstva zrna **licnoIme** i **prezIme** anotirana su pomoću **@SQLString**ova i imaju vrednosti 30 odnosno 50. Te anotacije su zanimljive iz dva razloga: prvo, koriste podrazumevanu vrednost ugnežđene anotacije **@Ogranicenja**, i drugo, koriste kraći oblik pisanja. Ako anotacija ima samo jedan element i vi mu date ime **value**, ne morate da pišete parove ime–vrednost; dovoljno je da specificirate samo vrednost (engl. *value*) u zagradama. To važi za sve dozvoljene tipove elemenata. Naravno, tada elementu morate dati ime „vrednost", ali u gornjem slučaju ipak dobijate semantički smislenu specifikaciju anotacije koja se lako čita:

```
@SQLString(30)
```

Procesor će tu vrednost upotrebiti za zadavanje širine SQL kolone koju će napraviti.

Koliko god da je sintaksa podrazumevanih vrednosti elegantna, ona brzo postaje složena. Pogledajmo anotaciju čiji je deo polje **identifikator**. To je anotacija **@SQLString** koja mora biti i primarni ključ baze podataka, pa tip elementa **primarniKljuc** mora biti specificiran u ugnežđenoj anotaciji **@Ogranicenja**. Tu stvari postaju zapetljane. Za tu ugnežđenu anotaciju morate upotrebiti opširnu sintaksu ime–vrednost, u kojoj ponovo specificirate ime elementa *i* ime interfejsa (**@interface**). Ali pošto posebno imenovani element **value** više nije jedini element čija se vrednost zadaje, ne možemo upotrebiti kraći oblik sintakse. Kao što vidite, rezultat ne izgleda baš lepo.

Drugačija rešenja

U ovom slučaju ima i drugih načina da se naprave anotacije. Na primer, možemo napraviti samo jednu klasu anotacije **@KolonaTabele** u čijem **enum** elementu definišemo vrednosti kao što su STRING, INTEGER, FLOAT itd. Onda ne bismo morali imati interfejs (**@interface**) za svaki SQL tip, ali bismo izgubili mogućnost da kvalifikujemo tipove pomoću dodatnih elemenata kao što su *velicina* ili *preciznost*, pa je šteta verovatno veća od koristi.

Stvarni SQL tip možemo opisati **String** elementom, npr. VARCHAR(30) ili INTEGER. Time ne bismo izgubili mogućnost da kvalifikujemo tipove, ali bi preslikavanje Java tipa u SQL tip bilo zaključano u kodu, a to nije dobro. Ne želimo da zbog promene baze podataka moramo ponovo da prevodimo klase; bilo bi elegantnije saopštiti procesoru anotacija da koristimo drugu „varijantu" SQL-a i prepustiti njemu da tokom obrade anotacija sam o tome vodi brigu.

Treće rešenje bilo bi korišćenje dva tipa anotacija, **@Ogranicenja** i relevantnog SQL tipa (recimo, **@SQLInteger**) za anotiranje željenog polja. To jeste malo zapetljano, ali prevodilac dozvoljava proizvoljan broj različitih anotacija za svako odredište (engl. *target*). Kada koristite više anotacija, istu anotaciju ne smete upotrebiti dvaput.

Anotacije ne podržavaju nasleđivanje

Rezervisanu reč **extends** ne smete koristiti uz **@interface**. To je šteta, pošto bi elegantno rešenje bilo da definišemo anotaciju **@KolonaTabele** (kao što sam predložio gore) uz ugnežđenu anotaciju tipa **@SQLTip**. Tako bismo od klase **@SQLTip** mogli da nasledimo sve SQL tipove, kao što su **@SQLInteger** i **@SQLString**. Sintaksa bi bila elegantnija, a manje bi se kucalo. Nema nagoveštaja da će u budućim izdanjima anotacije podržati nasleđivanje, pa izgleda kao da su gornji primeri ono najviše što se može uraditi u datim okolnostima.

Realizacija procesora

Sledi primer procesora anotacija koji učitava datoteku klase, traži u njoj anotacije baze podataka i generiše SQL komandu za pravljenje takve baze podataka:

```
//: anotacije/bazapodataka/TvoracTabele.java
// Reflektivni procesor anotacija.
// {Argumenti: anotacije.bazapodataka.Clan}
package anotacije.bazapodataka;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TvoracTabele {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
      System.out.println("argumenti: anotirane klase");
      System.exit(0);
    }
    for(String imeKlase : args) {
      Class<?> cl = Class.forName(imeKlase);
      TabelaBP tabelaBP = cl.getAnnotation(TabelaBP.class);
      if(tabelaBP == null) {
        System.out.println(
          "U klasi " + imeKlase + " nema anotacija TabelaBP");
        continue;
      }
      String imeTabele = tabelaBP.ime();
      // Ako je ime prazno, upotrebi ime Class:
      if(imeTabele.length() < 1)
        imeTabele = cl.getName().toUpperCase();
      List<String> defKolona = new ArrayList<String>();
```

```
      for(Field polje : cl.getDeclaredFields()) {
        String imeKolone = null;
        Annotation[] anot = polje.getDeclaredAnnotations();
        if(anot.length < 1)
          continue; // Nije kolona tabele baze podataka
        if(anot[0] instanceof SQLInteger) {
          SQLInteger sInt = (SQLInteger) anot[0];
          // Ako ime nije specificirano, upotrebi ime polja
          if(sInt.ime().length() < 1)
            imeKolone = polje.getName().toUpperCase();
          else
            imeKolone = sInt.ime();
          defKolona.add(imeKolone + " INT" +
            dajOgranicenja(sInt.ogranicenja()));
        }
        if(anot[0] instanceof SQLString) {
          SQLString sString = (SQLString) anot[0];
          // Ako ime nije specificirano, upotrebi ime polja.
          if(sString.ime().length() < 1)
            imeKolone = polje.getName().toUpperCase();
          else
            imeKolone = sString.ime();
          defKolona.add(imeKolone + " VARCHAR(" +
            sString.vrednost() + ")" +
            dajOgranicenja(sString.ogranicenja()));
        }
        StringBuilder komandaZaPravljenje = new StringBuilder(
          "NAPRAVI TABELU " + imeTabele + "(");
        for(String defKolone : defKolona)
          komandaZaPravljenje.append("\n    " + defKolone + ",");
        // Ukloni prateći zarez
        String napraviTabelu = komandaZaPravljenje.substring(
          0, komandaZaPravljenje.length() - 1) + ");";
        System.out.println("SQL za pravljenje tabele za klasu " +
          imeKlase + " glasi:\n" + napraviTabelu);
      }
    }
  }
  private static String dajOgranicenja(Ogranicenja ogr) {
    String ogranicenja = "";
    if(!ogr.dozvoliNull())
      ogranicenja += " NIJE NULL";
    if(ogr.primarniKLJuc())
      ogranicenja += " PRIMARNI KLJUC";
    if(ogr.jedinstveno())
      ogranicenja += " JEDINSTVENO";
    return ogranicenja;
  }
```

```
} /* Ispis:
SQL za pravljenje tabele za klasu anotacije.bazapodataka.Clan glasi:
NAPRAVI TABELU CLAN(
    LICNOIME VARCHAR(30));
SQL za pravljenje tabele za klasu anotacije.bazapodataka.Clan glasi:
NAPRAVI TABELU CLAN(
    LICNOIME VARCHAR(30),
    PREZIME VARCHAR(50));
SQL za pravljenje tabele za klasu anotacije.bazapodataka.Clan glasi:
NAPRAVI TABELU CLAN(
    LICNOIME VARCHAR(30),
    PREZIME VARCHAR(50),
    STAROST INT);
SQL za pravljenje tabele za klasu anotacije.bazapodataka.Clan glasi:
NAPRAVI TABELU CLAN(
    LICNOIME VARCHAR(30),
    PREZIME VARCHAR(50),
    STAROST INT,
    IDENTIFIKATOR VARCHAR(30) PRIMARNI KLJUC);
*///:-
```

Metoda **main**( ) prolazi kroza sva imena klasa na komandnoj liniji. Svaku klasu redom učitava metoda **forName**( ) i naredbom **getAnnotation**(**TabelaBP.class**) proverava ima li anotaciju **@TabelaBP**. Ukoliko ima, ime te tabele biva pronađeno i uskladišteno. Zatim se sva polja te klase učitavaju i proveravaju metodom **getDeclaredAnnotations**( ). Ona vraća niz svih anotacija definisanih za datu metodu. Operatorom **instanceof** utvrđuje se da li su te anotacije tipova **@SQLInteger** odnosno **@SQLString**, i u svakom od tih slučajeva pravi se relevantni **String** odlomak sa imenom kolone tabele. Pošto se interfejsi anotacija ne mogu nasleđivati, korišćenje metode **getDeclaredAnnotations**( ) jedini je način da se postigne približno polimorfno ponašanje.

Ugnežđenu anotaciju **@Ogranicenja** prosleđujemo metodi **dajOgranicenja**( ) koja gradi **String** što sadrži ta SQL ograničenja.

Vredi spomenuti da je goreopisana tehnika pomalo naivan način definisanja preslikavanja objekata na relacione baze podataka. Ako promenite ime tabele, moraćete iznova da prevodite Java kôd, pošto anotacija tipa **@TabelaBP** prima ime tabele kao parametar. To nije naročito poželjno ponašanje. Već je mnogo dostupnih struktura (engl. *framework*) za preslikavanje objekata na relacione baze podataka, a u sve više njih koriste se anotacije.

Vežba 1: (2) Realizujte više SQL tipova u primeru s bazom podataka.

Projekat:[1] Izmenite primer s bazom podataka tako da se pomoću JDBC-a povezuje i radi s pravom bazom podataka.

Projekat: Izmenite primer s bazom podataka tako da umesto pisanja SQL koda pravi odgovarajuće XML datoteke.

[1] Projekti su predlozi koji se mogu koristiti (recimo) za seminarske radove. Vodič s rešenjima ne sadrži rešenja projekata.

apt za obradu anotacija

Alatka za obradu anotacija **apt** (engl. *annotation processing tool*) prva je Sunova verzija alatke za pomoć u obradi anotacija. Pošto predstavlja prvi pokušaj, malo je primitivna, ali ipak može da posluži.

Kao i **javac**, **apt** obrađuje izvorne datoteke, a ne prevedene Java datoteke. Kada završi obradu izvornih datoteka, **apt** ih podrazumevano prevodi. To je korisno ako automatski pravite nove izvorne datoteke u sklopu postupka prevođenja i pakovanja (*build*). U stvari, **apt** u istom prolazu traži anotacije u novonapravljenim izvornim datotekama i prevodi ih.

Kada procesor anotacija napravi novu izvornu datoteku, anotacije u njoj traži u sledećoj *rundi* obrade (kako se to naziva u dokumentaciji). Alatka nastavlja obradu rundu za rundom dok ne prestane pravljenje novih izvornih datoteka. Zatim prevodi sve izvorne datoteke.

Za svaku anotaciju koju napišete potreban je sopstveni procesor, ali alatka **apt** može da grupiše više procesora. Njoj možete zadati više klasa za obradu, što je mnogo lakše nego da sami iterirate kroz klase **File**. Možete dodati i osluškivače koji će vas obavestiti kada se svaka runda obrade anotacija završi.

U vreme pisanja ove knjige, **apt** nije dostupan kao Ant zadatak (videti dodatak na adresi *http://MindView.net/Books/BetterJava*). Dok ne postane dostupan, očigledno je da se može pokrenuti iz Anta kao spoljni zadatak. Da biste preveli procesore anotacija iz ovog odeljka, u svojoj putanji klasa morate imati **tools.jar**; ta biblioteka sadrži i interfejse **com.sun.mirror.***.

apt radi tako što za svaku anotaciju koju pronađe upotrebi **AnnotationProcessorFactory** za pravljenje odgovarajuće vrste procesora anotacija. Kada pokrećete **apt**, zadajte proizvođačku klasu ili putanju klasa gde **apt** može naći proizvođačke klase koje su mu potrebne. Ako to ne uradite, **apt** će se otisnuti na tajnoviti postupak *otkrivanja*, čije pojedinosti možete naći u odeljku *Developing an Annotation Processor (Razvoj procesora anotacija)* Sunove dokumentacije.

Javinu sposobnost refleksije ne možete koristiti u procesoru anotacija za rad sa alatkom **apt**, pošto radite sa izvornim kodom a ne s prevedenim klasama.[4] API **mirror**[5] rešava taj problem tako što omogućuje da vidite metode, polja i tipove u neprevedenom izvornom kodu.

Sledeću anotaciju možete upotrebiti da izdvojite javne metode iz klase i pretvorite ih u interfejs:

```
//: anotacije/IzdvojiInterfejs.java
// Obrada anotacija pomoću APT-a.
package anotacije;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface IzdvojiInterfejs {
  public String vrednost();
} ///:~
```

[4] Međutim, nestandardna opcija **-XclassesAsDecls** omogućuje da radite sa anotacijama koje se nalaze u prevednim klasama.

[5] Projektanti Jave time sugerišu da refleksiju (odraz) nalazimo u ogledalu (engl. *mirror*).

Metaanotacija **RetentionPolicy** ima vrednost **SOURCE** zato što ovu anotaciju nema smisla čuvati u datoteci klase nakon što iz nje izdvojimo interfejs. Sledeća klasa ima javnu metodu koja može postati deo korisnog interfejsa:

```
//: anotacije/Mnozac.java
// Obrada anotacija pomoću APT-a.
package anotacije;

@IzdvojiInterfejs("IfMnozaca")
public class Mnozac {
  public int pomnozi(int x, int y) {
    int ukupno = 0;
    for(int i = 0; i < x; i++)
      ukupno = add(ukupno, y);
    return ukupno;
  }
  private int add(int x, int y) { return x + y; }
  public static void main(String[] args) {
    Mnozac m = new Mnozac();
    System.out.println("11*16 = " + m.pomnozi(11, 16));
  }
} /* Ispis:
11*16 = 176
*///:~
```

Klasa **Mnozac** (koja radi samo s pozitivnim celim brojevima) ima metodu **pomnozi( )** koja više puta poziva privatnu metodu **add( )** da bi obavila množenje. Metoda **add( )** nije javna, te stoga nije ni deo interfejsa. Anotaciji je data vrednost **IfMnozaca**, što je ime interfejsa koji treba napraviti.

Sada nam treba procesor za izdvajanje:

```
//: anotacije/ProcesorZaIzdvajanjeInterfejsa.java
// Obrada anotacija pomoću APT-a.
// {Exec: apt -factory
// anotacije.ProizvođačProcesoraZaIzdvajanjeInterfejsa
// Mnozac.java -s ../anotacije}
package anotacije;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
import java.util.*;

public class ProcesorZaIzdvajanjeInterfejsa
  implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;
  private ArrayList<MethodDeclaration> metodeInterfejsa =
    new ArrayList<MethodDeclaration>();
  public ProcesorZaIzdvajanjeInterfejsa(
    AnnotationProcessorEnvironment env) { this.env = env; }
```

```
  public void process() {
    for(TypeDeclaration deklTipa :
      env.getSpecifiedTypeDeclarations()) {
      IzdvojiInterfejs anot =
        deklTipa.getAnnotation(IzdvojiInterfejs.class);
      if(anot == null)
        break;
      for(MethodDeclaration m : deklTipa.getMethods())
        if(m.getModifiers().contains(Modifier.PUBLIC) &&
           !(m.getModifiers().contains(Modifier.STATIC)))
          metodeInterfejsa.add(m);
      if(metodeInterfejsa.size() > 0) {
        try {
          PrintWriter stampac =
            env.getFiler().createSourceFile(anot.vrednost());
          stampac.println("paket " +
            deklTipa.getPackage().getQualifiedName() +";");
          stampac.println("javni interfejs " +
            anot.vrednost() + " {");
          for(MethodDeclaration m : metodeInterfejsa) {
            stampac.print("  javni ");
            stampac.print(m.getReturnType() + " ");
            stampac.print(m.getSimpleName() + " (");
            int i = 0;
            for(ParameterDeclaration parm :
              m.getParameters()) {
              stampac.print(parm.getType() + " " +
                parm.getSimpleName());
              if(++i < m.getParameters().size())
                stampac.print(", ");
            }
            stampac.println(");");
          }
          stampac.println("}");
          stampac.close();
        } catch(IOException ioe) {
          throw new RuntimeException(ioe);
        }
      }
    }
  }
} ///:~
```

Sve se radi u metodi **process( )**. Pomoću klase **MethodDeclaration** i njene metode **getModifiers( )** identifikujemo javne metode (i zanemarujemo one koje su statične) obrađivane klase. Ukoliko ih pronađemo, skladištimo ih u objektu tipa **ArrayList** i upotrebljavamo za pravljenje metoda novog interfejsa u datoteci **.java**.

Obratite pažnju na to da objekat tipa **AnnotationProcessorEnvironment** prosleđujemo konstruktoru. Tom objektu možete slati upite za sve tipove (definicije klasa) koje alatka **apt** obrađuje i pomoću njega možete dobiti objekat tipa **Messager** i objekat tipa

Filer. **Messager** služi za slanje poruka korisniku, npr. grešaka koje su nastale tokom obrade i njihovih mesta u izvornom kodu. **Filer** je neka vrsta **PrintWriter**a za pravljenje novih datoteka. Umesto običnog **PrintWriter**a najčešće se koristi objekat tipa **Filer** zato što omogućuje alatki **apt** da prati nove datoteke koje napravite i, ako treba, pronalazi u njima anotacije i prevodi ih.

Videćete i da metoda **createSourceFile( )** otvara običan izlazni tok s tačnim imenom odgovarajuće Java klase ili interfejsa. Nema podrške za pravljenje Javinih jezičkih konstrukcija, pa Javin izvorni kôd morate da generišete pomalo primitivnim metodama **print( )** i **println( )**. To znači da morate paziti na to da zatvorite sve zagrade koje ste otvorili i da vaš kôd mora biti sintaksički ispravan.

Metodu **process( )** poziva alatka **apt** kojoj treba proizvodna metoda da bi napravila odgovarajući procesor:

```
//: anotacije/ProizvodjacProcesoraZaIzdvajanjeInterfejsa.java
// Obrada anotacija pomoću APT-a.
package anotacije;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class ProizvodjacProcesoraZaIzdvajanjeInterfejsa
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new ProcesorZaIzdvajanjeInterfejsa(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return
     Collections.singleton("anotacije.IzdvojiInterfejs");
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
} ///:~
```

U interfejsu **AnnotationProcessorFactory** samo su tri metode. Kao što vidite, procesor pravi metoda **getProcessorFor( )** koja prima **Set** (skup) deklaracija tipova (Java klasa koje alatka **apt** obrađuje) i objekat tipa **AnnotationProcessorEnvironment** koji smo već propuštali kroz procesor. Ostale dve metode, **supportedAnnotationTypes( )** i **supportedOptions( )**, postoje da biste mogli proveriti imate li procesore za sve anotacije koje je **apt** pronašao i da li podržavate sve opcije specificirane na komandnoj liniji. Naročito je važna metoda **getProcessorFor( )** zato što će vas **apt**, ako u kolekciji **String** ne vratite puno ime klase (tipa) anotacije, upozoriti da nema relevantnog procesora i izaći, a da ništa ne uradi.

Procesor i proizvodna metoda su u paketu **anotacije**, pa je za gornju strukturu direktorijuma komandna linija ugrađena u komentar **Exec** na početku programa **ProcesorZaIzdvajanjeInterfejsa.java**. Time se alatki **apt** kazuje da upotrebi goredefinisanu

proizvodnu metodu i da obradi datoteku **Mnozac.java**. Opcija **-s** zadaje da se sve nove datoteke moraju napraviti u direktorijumu **anotacije**. Ovako izgleda generisana datoteka **IfMnozaca.java**, što ste mogli pogoditi pročitavši naredbe **println( )** u gornjem procesoru:

```
package anotacije;
public interface IfMnozaca {
 public int pomnozi (int x, int y);
}
```

I ovu datoteku će prevesti **apt**, pa će i datoteka **IfMnozaca.class** biti u istom direktorijumu.

Vežba 2: (3) Programu za izdvajanje interfejsa dodajte podršku za deljenje.

Upotreba obrasca *Visitor* sa alatkom **apt**

Obrada anotacija ume da postane teška. U gornjem primeru imamo relativno jednostavan procesor anotacija koji tumači samo jednu anotaciju, pa ipak smo morali prilično da ga usložnimo. Da se složenost ne bi preterano povećavala s dodavanjem više anotacija i više procesora, API **mirror** ima klase koje podržavaju projektni obrazac *Visitor* (Posetilac). Visitor je jedan od klasičnih projektnih obrazaca iz knjige *Design Patterns* (Projektni obrasci) koju su napisali Gamma i dr, a opširnije objašnjenje nalazi se i u knjizi *Thinking in Patterns*.

Visitor prolazi kroz strukturu podataka ili kolekciju objekata i obavlja određenu operaciju na svakome od njih. Struktura podataka ne mora biti uređena, a operacija koju obavljate na svakom objektu mora biti specifična za njegov tip. Time se operacije razdvajaju od samih objekata, što znači da možete dodavati nove operacije, a da ne dodajete metode u definicije klasa.

Visitor je podesan za obradu anotacija zato što Java klasu možete zamisliti kao kolekciju objekata tipova kao što su **TypeDeclaration**, **FieldDeclaration**, **MethodDeclaration** itd. Kada alatku **apt** upotrebite sa obrascem *Visitor*, dajete Visitor klasu koja ima metodu za obradu svakog tipa deklaracije koja će biti posećena. Dakle, možete realizovati odgovarajuće ponašanje anotacije za metode, klase, polja itd.

Evo nam opet SQL generatora tabela, ali ćemo ovoga puta upotrebiti proizvodnu metodu i procesor koji koristi projektni obrazac *Visitor*:

```
//: anotacije/bazapodataka/ProizvodjacProcesoraZaPravljenjeTabela.java
// Primer s bazom podataka napravljen po projektnom obrascu Visitor.
// {Exec: apt -factory
// anotacije.bazapodataka.ProizvođačProcesoraZaPravljenjeTabela
// bazapodataka/Clan.java -s bazapodataka}
package anotacije.bazapodataka;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.util.*;
import java.util.*;
import static com.sun.mirror.util.DeclarationVisitors.*;
```

```
public class ProizvodjacProcesoraZaPravljenjeTabela
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new ProcesorZaPravljenjeTabela(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return Arrays.asList(
      "anotacije.bazapodataka.TabelaBP",
      "anotacije.bazapodataka.Ogranicenja",
      "anotacije.bazapodataka.SQLString",
      "anotacije.bazapodataka.SQLInteger");
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
  private static class ProcesorZaPravljenjeTabela
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private String sql = "";
    public ProcesorZaPravljenjeTabela(
      AnnotationProcessorEnvironment env) {
      this.env = env;
    }
    public void process() {
      for(TypeDeclaration deklTipa :
        env.getSpecifiedTypeDeclarations()) {
        deklTipa.accept(getDeclarationScanner(
          new PosetilacZaPravljenjeTabela(), NO_OP));
        sql = sql.substring(0, sql.length() - 1) + ");";
        System.out.println("SQL za pravljenje glasi:\n" + sql);
        sql = "";
      }
    }
    private class PosetilacZaPravljenjeTabela
      extends SimpleDeclarationVisitor {
      public void visitClassDeclaration(
        ClassDeclaration d) {
        TabelaBP tabelaBP = d.getAnnotation(TabelaBP.class);
        if(tabelaBP != null) {
          sql += "NAPRAVI TABELU ";
          sql += (tabelaBP.ime().length() < 1)
            ? d.getSimpleName().toUpperCase()
            : tabelaBP.ime();
          sql += " (";
        }
      }
```

```
    public void visitFieldDeclaration(
      FieldDeclaration d) {
      String imeKolone = "";
      if(d.getAnnotation(SQLInteger.class) != null) {
        SQLInteger sInt = d.getAnnotation(
            SQLInteger.class);
        // Ako ime nije specificirano, upotrebi ime polja.
        if(sInt.ime().length() < 1)
          imeKolone = d.getSimpleName().toUpperCase();
        else
          imeKolone = sInt.ime();
        sql += "\n    " + imeKolone + " INT" +
          dajOgranicenja(sInt.ogranicenja()) + ",";
      }
      if(d.getAnnotation(SQLString.class) != null) {
        SQLString sString = d.getAnnotation(
            SQLString.class);
        // Ako ime nije specificirano, upotrebi ime polja.
        if(sString.ime().length() < 1)
          imeKolone = d.getSimpleName().toUpperCase();
        else
          imeKolone = sString.ime();
        sql += "\n    " + imeKolone + " VARCHAR(" +
          sString.vrednost() + ")" +
          dajOgranicenja(sString.ogranicenja()) + ",";
      }
    }
    private String dajOgranicenja(Ogranicenja ogr) {
      String ogranicenja = "";
      if(!ogr.dozvoliNull())
        ogranicenja += " NIJE NULL";
      if(ogr.primarniKLJuc())
        ogranicenja += " PRIMARNI KLJUC";
      if(ogr.jedinstveno())
        ogranicenja += " JEDINSTVENO";
    }       return ogranicenja;
  }
 }
} ///:~
```

Ispis rezultata je isti kao u prethodnom primeru **TabelaBP**.

U ovom primeru su procesor i posetilac unutrašnje klase. Obratite pažnju na to da metoda **process( )** samo dodaje klasu posetioca i inicijalizuje SQL znakovni niz.

Oba parametra metode **getDeclarationScanner( )** jesu posetioci; prvi biva upotrebljen pre posete svakoj deklaraciji, a drugi posle. Ovom procesoru je potreban samo posetilac od pre posete, pa se **NO_OP** ne daje kao drugi parametar. To je statično polje interfejsa **DeclarationVisitor**, što je objekat tipa **DeclarationVisitor** koji ne radi ništa.

PosetilacZaPravljenjeTabela nasleđuje klasu **SimpleDeclarationVisitor** i redefiniše metode **visitClassDeclaration( )** i **visitFieldDeclaration( )**. **SimpleDeclarationVisitor**

je adapter koji realizuje sve metode interfejsa **DeclarationVisitor**, pa možete da se usredsredite samo na one koje su vam potrebne. U metodi **visitClassDeclaration( )** proverava se da li u objektu **ClassDeclaration** postoji anotacija **TabelaBP**, i ako postoji, inicijalizuje se prvi deo SQL znakovnog niza za pravljenje. U metodi **visitFieldDeclaration( )** ispituje se da li anotacije polja postoje u deklaraciji polja. Informacije se izdvajaju kao u prvobitnom primeru, navedenom u prethodnom delu poglavlja.

Možda vam ovo izgleda kao komplikovaniji način rada, ali njime dobijamo skalabilnije rešenje. Za složeniji procesor anotacija, pisanje samostalnog procesora na način prethodnog primera ubrzo bi postalo previše komplikovano.

Vežba 3: (2) Programu **ProizvodjacProcesoraZaPravljenjeTabela.java** dodajte podršku za više SQL tipova.

Jedinično testiranje pomoću anotacija

Jedinično testiranje (engl. *unit testing*) je praksa pravljenja jednog ili više testova za svaku metodu klase, kako bi se pravilno ispitalo da li se delovi klase ispravno ponašaju. Najkorišćenija alatka za jedinično testiranje u Javi je *JUnit*; u vreme pisanja ove knjige, JUnit su ažurirali u JUnit verziju 4 da bi se obuhvatile anotacije.[6] Jedan od glavnih problema s verzijama JUnita pre anotacija bile su brojne „ceremonije" neophodne u pripremi i izvršavanju JUnit ispitivanja. S vremenom se to svelo, ali će anotacije testiranje još približiti „najjednostavnijem sistemu za jedinično testiranje koji se može zamisliti".

U verzijama JUnita pre anotacija, morali smo da pišemo zasebnu klasu za skladištenje jediničnih testova. Uz anotacije, jedinične testove možemo da uključimo u klasu koja se testira i time na minimum svedemo vreme i trud pri jediničnom testiranju. Taj pristup ima i veliku dodatnu prednost što se na taj način mogu testirati i privatne metode, jednako lako kao javne.

Pošto je struktura za testiranje u ovom primeru napravljena pomoću anotacija, nazvao sam je **@Unit**. Za najjednostavnije testiranje (koje ćete verovatno najčešće koristiti) dovoljno je staviti anotaciju **@Test** uz metode koje treba testirati. Ispitne metode ove strukture imaju opciju da vrate **boolean** koji pokazuje uspeh ili neuspeh ukoliko im se ne zadaju ulazni argumenti. Tim ispitnim metodama možete dati proizvoljna imena. Takođe, **@Unit** ispitnim metodama možete pristupati kako god želite, i privatno.

Da biste mogli da koristite **@Unit**, treba da uvezete paket **net.mindview.atunit**,[7] odgovarajuće metode i polja označite sa **@Unit** (o tome ćete više saznati u narednim primerima) i date *build* sistemu da pokrene **@Unit** za dobijenu klasu. Sledi jednostavan primer:

```
//: anotacije/AtUnitPrimer1.java
package anotacije;
import net.mindview.atunit.*;
import net.mindview.util.*;
```

[6] Spočetka sam pomišljao na to da napišem „bolji JUnit" na osnovu ovde prikazanog dizajna. Međutim, izgleda da JUnit4 obuhvata mnoge ovde prikazane ideje, pa mi je lakše ovako.

[7] Ta biblioteka je deo mrežnog koda ove knjige i dostupna je na adresi *www.MindView.net*.

```
public class AtUnitPrimer1 {
  public String metodaJedan() {
    return "Ovo je metodaJedan";
  }
  public int metodaDva() {
    System.out.println("Ovo je metodaDva");
    return 2;
  }
  @Test boolean metodaJedanTest() {
    return metodaJedan().equals("Ovo je metodaJedan");
  }
  @Test boolean m2() { return metodaDva() == 2; }
  @Test private boolean m3() { return true; }
  // Prikazuje rezultat u slučaju grešaka:
  @Test boolean neuspesanTest() { return false; }
  @Test boolean josjedanNeuspeh() { return false; }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit AtUnitPrimer1");
  }
} /* Ispis:
anotacije.AtUnitPrimer1
  . metodaJedanTest
  . m2 Ovo je metodaDva

  . m3
  . neuspesanTest (failed)
  . josjedanNeuspeh (failed)
(5 tests)

>>> 2 FAILURES <<<
  anotacije.AtUnitPrimer1: neuspesanTest
  anotacije.AtUnitPrimer1: josjedanNeuspeh
*///:~
```

Klase koje će biti jedinično testirane (engl. ***@Unit tested*** ili *unit tested*) moraju biti smeštene u pakete.

Anotacija **@Test** ispred metoda **metodaJedanTest( )**, **m2( )**, **m3( )**, **neuspesanTest( )** i **josjedanNeuspeh( )** kazuje strukturi **@Unit** da te metode pokrene kao jedinične testove. Struktura se stara da one ne prime ulazne argumente i da vrate **boolean** ili **void**. Kada budete sami pisali jedinične testove, samo utvrdite da li je test uspeo ili ne, tj. da li je vratio **true** ili **false** (za metode koje vraćaju **boolean**).

Ako poznajete JUnit, primetićete i da **@Unit** daje informativnije rezultate – prikazuje se tekući test (onaj koji se trenutno izvršava) pa je njegov rezultat korisniji, a na kraju ispisuje klase i testove koji su prouzrokovali neuspeh.

Ukoliko vam to ne odgovara, ispitne metode ne morate da ugrađujete u svoje klase. Neugrađene testove je najlakše napraviti nasleđivanjem:

```
//: anotacije/AtUnitSpoljniTest.java
// Pravljenje neugrađenih testova.
package anotacije;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitSpoljniTest extends AtUnitPrimer1 {
  @Test boolean _metodaJedan() {
    return metodaJedan().equals("Ovo je metodaJedan");
  }
  @Test boolean _metodaDva() { return metodaDva() == 2; }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
     "java net.mindview.atunit.AtUnit AtUnitSpoljniTest");
  }
} /* Ispis:
anotacije.AtUnitSpoljniTest
  . _metodaJedan
  . _metodaDva Ovo je metodaDva

OK (2 tests)
*///:~
```

Iz ovog primera vidi se i vrednost slobodnog davanja imena (za razliku od obaveznog **test** na početku imena svih testova u JUnitu). Ovde je **@Test** metodi koja neposredno testira drugu metodu dato ime te metode, uz potcrtu (_) na početku. (Ne tvrdim da je to idealno rešenje, nego samo pokazujem jednu od mogućnosti.)

Za pravljenje neugrađenih testova možete upotrebiti i kompoziciju:

```
//: anotacije/AtUnitKompozicija.java
// Pravljenje neugrađenih testova.
package anotacije;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitKompozicija {
  AtUnitPrimer1 testObjekat = new AtUnitPrimer1();
  @Test boolean _metodaJedan() {
    return
      testObjekat.metodaJedan().equals("Ovo je metodaJedan");
  }
  @Test boolean _metodaDva() {
    return testObjekat.metodaDva() == 2;
  }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
```

```
      "java net.mindview.atunit.AtUnit AtUnitKompozicija");
  }
} /* Ispis:
anotacije.AtUnitKompozicija
  . _metodaJedan
  . _metodaDva Ovo je metodaDva

OK (2 tests)
*///:~
```

Za svaki test pravi se po jedan nov član **testObjekat**, pošto se po jedan objekat tipa **AtUnitKompozicija** pravi za svaki test.

Nema posebnih metoda za proveru uspešnosti testiranja kao u JUnitu, ali drugi oblik metode **@Test** omogućuje da vratite tip **void** (ili **boolean**, ako i u ovom slučaju želite da vraćate **true** ili **false**). Za testiranje uspešnosti možete koristiti Javine naredbe **assert** koje treba uključiti indikatorom **-ea** na **java** komandnoj liniji, ali ih **@Unit** uključuje automatski. Kao pokazatelj neuspeha možete upotrebiti čak i izuzetak. Jedan od ciljeva prilikom projektovanja alatke **@Unit** bio je da se količina dodatne sintakse svede na najmanju meru, pa su za prijavljivanje grešaka potrebni samo Javine naredbe **assert** i izuzeci. Da je test neuspešan, pokazuje **assert** koji vrati negativan rezultat ili izuzetak koji potiče iz ispitne metode, ali **@Unit** ni u tom slučaju ne prestaje da radi, nego nastavlja ispitivanje dok ne obavi sve testove. Evo primera:

```
//: anotacije/AtUnitPrimer2.java
// Uz testiranje alatkom @Test možemo upotrebljavati
// naredbe assert i izuzetke.
package anotacije;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitPrimer2 {
  public String metodaJedan() {
    return "Ovo je metodaJedan";
  }
  public int metodaDva() {
    System.out.println("Ovo je metodaDva");
    return 2;
  }
  @Test void assertPrimer() {
    assert metodaJedan().equals("Ovo je metodaJedan");
  }
  @Test void assertPrimerNeuspeha() {
    assert 1 == 2: "Kojeg li iznenađenja!";
  }
  @Test void primerIzuzetka() throws IOException {
    new FileInputStream("nijedatoteka.txt"); // Baca
  }
  @Test boolean assertIReturn() {
```

```
    // Provera uspešnosti uz poruku:
    assert metodaDva() == 2: "metodaDva mora dati 2";
    return metodaJedan().equals("Ovo je metodaJedan");
  }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit AtUnitPrimer2");
  }
} /* Ispis:
anotacije.AtUnitPrimer2
  . assertPrimer
  . assertPrimerNeuspeha java.lang.AssertionError: Kojeg li iznenađenja!
(failed)
  . primerIzuzetka java.io.FileNotFoundException: nijedatoteka.txt
(The system cannot find the file specified)
(failed)
  . assertIReturn Ovo je metodaDva

(4 tests)

>>> 2 FAILURES <<<
  anotacije.AtUnitPrimer2: assertPrimerNeuspeha
  anotacije.AtUnitPrimer2: primerIzuzetka
*///:~
```

U narednom primeru, jednostavnim neugrađenim testovima čija se uspešnost proverava naredbama **assert**, ispitujemo klasu **java.util.HashSet**:

```
//: anotacije/HashSetTest.java
package anotacije;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
  HashSet<String> testObjekat = new HashSet<String>();
  @Test void inicijalizacija() {
    assert testObjekat.isEmpty();
  }
  @Test void _contains() {
    testObjekat.add("jedan");
    assert testObjekat.contains("jedan");
  }
  @Test void _remove() {
    testObjekat.add("jedan");
    testObjekat.remove("jedan");
    assert testObjekat.isEmpty();
  }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit HashSetTest");
```

```
  }
} /* Ispis:
anotacije.HashSetTest
  . inicijalizacija
  . _remove
  . _contains
OK (3 tests)
*///:~
```

Ukoliko nema drugih ograničenja, izgleda da je način s nasleđivanjem jednostavniji.

Vežba 4: (3) Proverite da li se nov **testObjekat** pravi pre svakog testa.

Vežba 5: (1) Izmenite gornji primer korišćenjem pristupa s nasleđivanjem.

Vežba 6: (1) Testirajte **LinkedList** na način prikazan u programu **HashSetTest.java**.

Vežba 7: (1) Izmenite prethodni primer korišćenjem pristupa s nasleđivanjem.

Za svako jedinično testiranje, **@Unit** podrazumevanim konstruktorom pravi objekat klase koju testira. Poziva se test za taj objekat, koji se zatim odbacuje da bi se izbegli uticaji na druga jedinična testiranja. Za pravljenje objekata koristi se podrazumevani konstruktor. Ukoliko nemate podrazumevani konstruktor ili vam je potrebna sofisticiranija konstrukcija objekata, napravite statičnu metodu za građenje objekata i prikačite za nju anotaciju **@TestObjectCreate**, na ovaj način:

```
//: anotacije/AtUnitPrimer3.java
package anotacije;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitPrimer3 {
  private int n;
  public AtUnitPrimer3(int n) { this.n = n; }
  public int getN() { return n; }
  public String metodaJedan() {
    return "Ovo je metodaJedan";
  }
  public int metodaDva() {
    System.out.println("Ovo je metodaDva");
    return 2;
  }
  @TestObjectCreate static AtUnitPrimer3 napravi() {
    return new AtUnitPrimer3(47);
  }
  @Test boolean inicijalizacija() { return n == 47; }
  @Test boolean metodaJedanTest() {
    return metodaJedan().equals("Ovo je metodaJedan");
  }
  @Test boolean m2() { return metodaDva() == 2; }
  public static void main(String[] args) throws Exception {
```

```
      OSIzvršenje.komanda(
        "java net.mindview.atunit.AtUnit AtUnitPrimer3");
    }
} /* Ispis:
anotacije.AtUnitPrimer3
  . inicijalizacija
  . metodaJedanTest
  . m2 Ovo je metodaDva

OK (3 tests)
*///:~
```

Metoda **@TestObjectCreate** mora biti statična i vratiti objekat tipa koji testirate – program **@Unit** će se postarati da to bude ispunjeno.

Katkada će vam zatrebati dodatna polja za podršku jediničnom testiranju. Anotaciju **@TestProperty** možete upotrebiti za označavanje polja koja se upotrebljavaju samo za jedinično testiranje (da biste mogli da ih uklonite pre isporuke programa klijentu). U narednom primeru čitamo vrednosti iz znakovnog niza raščlanjenog metodom **String.split( )**. Taj ulaz se upotrebljava za proizvodnju test objekata:

```
//: anotacije/AtUnitPrimer4.java
package anotacije;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitPrimer4 {
  static String teorija = "All brontosauruses " +
    "are thin at one end, much MUCH thicker in the " +
    "middle, and then thin again at the far end.";
  private String rec;
// Inicijalizacija generatora slučajnih brojeva koja zavisi od vremena
  private Random slucajan = new Random();
  public AtUnitPrimer4(String rec) { this.rec = rec; }
  public String dajRec() { return rec; }
  public String ispreturajRec() {
    List<Character> znakovi = new ArrayList<Character>();
    for(Character z : rec.toCharArray())
      znakovi.add(z);
    Collections.shuffle(znakovi, slucajan);
    StringBuilder rezultat = new StringBuilder();
    for(char zn : znakovi)
      rezultat.append(zn);
    return rezultat.toString();
  }
  @TestProperty static List<String> ulaz =
    Arrays.asList(teorija.split(" "));
  @TestProperty
    static Iterator<String> reci = ulaz.iterator();
```

```
  @TestObjectCreate static AtUnitPrimer4 napravi() {
    if(reci.hasNext())
      return new AtUnitPrimer4(reci.next());
    else
      return null;
  }
  @Test boolean reci() {
    print("'" + dajRec() + "'");
    return dajRec().equals("are");
  }
  @Test boolean ispreturaj1() {
    // Prelazimo na određeno seme (generatora slučajnih
    // brojeva) da bismo mogli da proveravamo rezultate:
    slucajan = new Random(47);
    print("'" + dajRec() + "'");
    String ispreturano = ispreturajRec();
    print(ispreturano);
    return ispreturano.equals("lAl");
  }
  @Test boolean ispreturaj2() {
    slucajan = new Random(74);
    print("'" + dajRec() + "'");
    String ispreturano = ispreturajRec();
    print(ispreturano);
    return ispreturano.equals("tsaeborornussu");
  }
  public static void main(String[] args) throws Exception {
    System.out.println("kreće");
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit AtUnitPrimer4");
  }
} /* Ispis:
kreće
anotacije.AtUnitPrimer4
  . ispreturaj1 'All'
lAl

  . ispreturaj2 'brontosauruses'
tsaeborornussu

  . reci 'are'

OK (3 tests)
*///:~
```

@TestProperty možete upotrebiti i za označavanje metoda koje se tokom testiranja koriste, ali same nisu testovi.

Vodite računa o tome da uspešnost ovog programa zavisi od redosleda izvršavanja testova, što po pravilu nije dobro.

Ukoliko tokom inicijalizacije objekata za testiranje uradite nešto što kasnije treba počistiti (engl. *clean up*), možete dodati statičnu metodu označenu sa **@TestObject-Cleanup** koja obavlja čišćenje kada završite testiranje objekta. U sledećem primeru, **@TestObjectCreate** otvara datoteku radi pravljenja svakog objekta za testiranje, pa se ta datoteka mora zatvoriti pre odbacivanja samog objekta testiranja:

```
//: anotacije/AtUnitPrimer5.java
package anotacije;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitPrimer5 {
  private String tekst;
  public AtUnitPrimer5(String tekst) { this.tekst = tekst; }
  public String toString() { return tekst; }
  @TestProperty static PrintWriter output;
  @TestProperty static int brojac;
  @TestObjectCreate static AtUnitPrimer5 napravi() {
    String id = Integer.toString(brojac++);
    try {
      izlaz = new PrintWriter("Test" + id + ".txt");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return new AtUnitPrimer5(id);
  }
  @TestObjectCleanup static void
  cleanup(AtUnitPrimer5 tobj) {
    System.out.println("Pokrećem metodu cleanup");
    izlaz.close();
  }
  @Test boolean test1() {
    izlaz.print("test1");
    return true;
  }
  @Test boolean test2() {
    izlaz.print("test2");
    return true;
  }
  @Test boolean test3() {
    izlaz.print("test3");
    return true;
  }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit AtUnitPrimer5");
  }
```

```
} /* Ispis:
anotacije.AtUnitPrimer5
  . test1
Pokrećem cleanup
  . test2
Pokrećem cleanup
  . test3
Pokrećem cleanup
OK (3 tests)
*///:~
```

Iz ispisa rezultata vidite da se metoda za čišćenje automatski pokreće nakon svakog testiranja.

Testiranje generičkih tipova alatkom @Unit

Generički tipovi predstavljaju poseban problem, zato što ne možete „testirati generički" (uopšteno). Testira se određeni parametar tipa ili skup parametara. Rešenje je jednostavno: izvedite klasu za testiranje iz specificirane verzije generičke klase.

Ovo je jednostavna realizacija steka:

```
//: anotacije/StekL.java
// Stek napravljen od ulančane liste.
package anotacije;
import java.util.*;

public class StekL<T> {
  private LinkedList<T> list = new LinkedList<T>();
  public void push(T v) { list.addFirst(v); }
  public T top() { return list.ggetFirst(); }
  public T pop() { return list.removeFirst(); }
} ///:~
```

Izvešćemo klasu za testiranje iz **StekL**<**String**> da bismo testirali **String** verziju:

```
//: anotacije/StekLStringTest.java
// Primena alatke @Unit na generičke tipove.
package anotacije;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class StekLStringTest extends StekL<String> {
  @Test void _push() {
    push("jedan");
    assert top().equals("jedan");
    push("dva");
    assert top().equals("dva");
  }
  @Test void _pop() {
    push("jedan");
```

```
    push("dva");
    assert pop().equals("dva");
    assert pop().equals("jedan");
  }
  @Test void _top() {
    push("A");
    push("B");
    assert top().equals("B");
    assert top().equals("B");
  }
  public static void main(String[] args) throws Exception {
    OSIzvršenje.komanda(
      "java net.mindview.atunit.AtUnit StekLStringTest");
  }
} /* Ispis:
anotacije.StekLStringTest
  . _push
  . _pop
  . _top
OK (3 tests)
*///:~
```

Jedini potencijalni nedostatak nasleđivanja jeste to što gubimo mogućnost pristupanja privatnim metodama klase koja se testira. Ukoliko to predstavlja problem, dodajte toj metodi modifikator **protected** ili napišite neprivatnu **@TestProperty** metodu koja poziva tu privatnu metodu (potom će **@TestProperty** metoda alatkom **AtUnitRemover** biti izbačena iz koda za isporuku, i to ću prikazati u nastavku poglavlja).

Vežba 8: (2) Napravite privatnu metodu i dodajte joj neprivatnu **@TestProperty** metodu, kao što je prethodno opisano. Pozovite tu metodu u svom kodu za testiranje.

Vežba 9: (2) Napišite osnovne **@Unit** testove za **HashMap**.

Vežba 10: (2) Izaberite neki primer iz drugih delova knjige i dodajte mu **@Unit** testove.

„Svite" nisu potrebne

Jedna od velikih prednosti alatke **@Unit** nad JUnitom jeste to što „svite" nisu potrebne. JUnitu morate saopštiti šta želite da testirate, i zato se testovi grupišu u „svite" da bi JUnit mogao da ih pronađe i pokrene.

@Unit jednostavno traži datoteke klasa koje sadrže odgovarajuće anotacije i zatim izvršava **@Test** metode. Jedan od mojih glavnih ciljeva prilikom projektovanja sistema za testiranje **@Unit** bio je da on bude neverovatno jasan, kako bi ljudi mogli početi da ga koriste jednostavnim dodavanjem **@Test** metoda, bez ikakvog specijalnog koda ili predznanja kakvo je neophodno za JUnit i mnoge druge strukture za testiranje. Pisanje testova je dovoljno teško bez ikakvih dodatnih prepreka, pa pomoću @Unit može postati mnogo lakše. Zbog toga su veće šanse da ćete zaista početi da pišete testove.

Realizacija interfejsa @Unit

Prvo moramo da definišemo sve tipove anotacija. To su obične oznake bez polja. Oznaka **@Test** je bila definisana na početku poglavlja, a ovde su ostale anotacije:

```
//: net/mindview/atunit/TestObjectCreate.java
// @Unit oznaka @TestObjectCreate.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~

//: net/mindview/atunit/TestObjectCleanup.java
// @Unit oznaka @TestObjectCleanup.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~

//: net/mindview/atunit/TestProperty.java
// @Unit oznaka @TestProperty.
package net.mindview.atunit;
import java.lang.annotation.*;

// I polja i metode mogu biti označeni sa @TestProperty:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} ///:~
```

Svi testovi imaju **RUNTIME** kao vrednost metaanotacije **@Retention**, zato što sistem **@Unit** testove mora da pronalazi u prevedenom kodu.

Izdvojićemo anotacije pomoću refleksije da bismo realizovali sistem koji izvršava testiranje. Na osnovu tih informacija program odlučuje kako da napravi test objekte i kako da ih testira. Zbog anotacija, to se postiže iznenađujuće malim i jednostavnim programom:

```
//: net/mindview/atunit/AtUnit.java
// Struktura za jedinično testiranje čiju osnovu čine anotacije.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class AtUnit implements ProcessFiles.Strategy {
  static Class<?> testClass;
  static List<String> failedTests= new ArrayList<String>();
  static long testsRun = 0;
  static long failures = 0;
  public static void main(String[] args) throws Exception {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true); // Dozvoli naredbe assert
    new ProcessFiles(new AtUnit(), "class").start(args);
    if(failures == 0)
      print("OK (" + testsRun + " tests)");
    else {
      print("(" + testsRun + " tests)");
      print("\n>>> " + failures + " FAILURE" +
        (failures > 1 ? "S" : "") + " <<<");
      for(String failed : failedTests)
        print("  " + failed);
    }
  }
  public void process(File cFile) {
    try {
      String cName = ClassNameFinder.thisClass(
        BinaryFile.read(cFile));
        return; // Zanemari klase izvan paketa
        if(!cName.contains("."))
      testClass = Class.forName(cName);
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    TestMethods testMethods = new TestMethods();
    Method creator = null;
    Method cleanup = null;
    for(Method m : testClass.getDeclaredMethods()) {
      testMethods.addIfTestMethod(m);
      if(creator == null)
        creator = checkForCreatorMethod(m);
      if(cleanup == null)
        cleanup = checkForCleanupMethod(m);
    }
    if(testMethods.size() > 0) {
      if(creator == null)
        try {
          if(!Modifier.isPublic(testClass
              .getDeclaredConstructor().getModifiers())) {
            print("Error: " + testClass +
              " default constructor must be public");
            System.exit(1);
          }
        } catch(NoSuchMethodException e) {
```

```
        // Napravljen podrazumevani konstruktor; OK
      }
    print(testClass.getName());
  }
  for(Method m : testMethods) {
    printnb("  . " + m.getName() + " ");
    try {
      Object testObjekat = createTestObject(creator);
      boolean success = false;
      try {
        if(m.getReturnType().equals(boolean.class))
          success = (Boolean)m.invoke(testObjekat);
        else {
          m.invoke(testObjekat);
          success = true; // Ako sve naredbe assert uspeju
        }
      } catch(InvocationTargetException e) {
        // Stvarni izuzetak je unutar e:
        print(e.getCause());
      }
      print(success ? "" : "(failed)");
      testsRun++;
      if(!success) {
        failures++;
        failedTests.add(testClass.getName() +
          ": " + m.getName());
      }
      if(cleanup != null)
        cleanup.invoke(testObjekat, testObjekat);
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}
static class TestMethods extends ArrayList<Method> {
  void addIfTestMethod(Method m) {
    if(m.getAnnotation(Test.class) == null)
      return;
    if(!(m.getReturnType().equals(boolean.class) ||
        m.getReturnType().equals(void.class)))
      throw new RuntimeException("@Test method" +
        " must return boolean or void");
    m.setAccessible(true); // Ukoliko je privatna itd.
    add(m);
  }
}
private static Method checkForCreatorMethod(Method m) {
  if(m.getAnnotation(TestObjectCreate.class) == null)
    return null;
  if(!m.getReturnType().equals(testClass))
```

```
        throw new RuntimeException("@TestObjectCreate " +
          "must return instance of Class to be tested");
      if((m.getModifiers() &
          java.lang.reflect.Modifier.STATIC) < 1)
        throw new RuntimeException("@TestObjectCreate " +
          "must be static.");
      m.setAccessible(true);
      return m;
    }
    private static Method checkForCleanupMethod(Method m) {
      if(m.getAnnotation(TestObjectCleanup.class) == null)
        return null;
      if(!m.getReturnType().equals(void.class))
        throw new RuntimeException("@TestObjectCleanup " +
          "must return void");
      if((m.getModifiers() &
          java.lang.reflect.Modifier.STATIC) < 1)
        throw new RuntimeException("@TestObjectCleanup " +
          "must be static.");
      if(m.getParameterTypes().length == 0 ||
         m.getParameterTypes() != testClass)
        throw new RuntimeException("@TestObjectCleanup " +
          "must take an argument of the tested type.");
      m.setAccessible(true);
      return m;
    }
    private static Object createTestObject(Method creator) {
      if(creator != null) {
        try {
          return creator.invoke(testClass);
        } catch(Exception e) {
          throw new RuntimeException("Couldn't run " +
            "@TestObject (creator) method.");
        }
      } else { // Upotrebi podrazumevani konstruktor:
        try {
          return testClass.newInstance();
        } catch(Exception e) {
          throw new RuntimeException("Couldn't create a " +
            "test object. Try using a @TestObject method.");
        }
      }
    }
} ///:~
```

AtUnit.java upotrebljava alatku **ProcessFiles** iz paketa **net.mindview.util**. Klasa **AtUnit** realizuje interfejs **ProcessFiles.Strategy** koji obuhvata metodu **process**( ). Na taj način se instanca od **AtUnit** može proslediti **ProcessFiles** konstruktoru. Drugi argument konstruktora saopštava alatci **ProcessFiles** da traži datoteke s nastavkom imena **class**.

Ako ne zadate argument s komandne linije, program će pretražiti stablo tekućeg direktorijuma. Možete zadati i više argumenata, bilo class datoteka (s nastavkom **.class** ili bez njega) bilo direktorijuma. Pošto **@Unit** automatski pronalazi klase i metode koji se mogu testirati, nije potreban mehanizam „svita".[8]

Jedan od problema koje **AtUnit.java** mora da reši kada pronađe class datoteku jeste taj što se puno ime klase (koje obuhvata i ime njenog paketa) ne može saznati iz imena same class datoteke. Zato se class datoteka mora analizirati, što nije lako, ali ni nemoguće.[9] Dakle, nakon pronalaska **.class** datoteke ona se prvo otvara, njen binarni sadržaj učitava i predaje metodi **ClassNameFinder.thisClass( )**. Potom prelazimo u domen „inženjeringa bajtkoda", pošto zapravo analiziramo sadržaj class datoteke:

```
//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
  public static String thisClass(byte[] classBytes) {
    Map<Integer,Integer> offsetTable =
      new HashMap<Integer,Integer>();
    Map<Integer,String> classNameTable =
      new HashMap<Integer,String>();
    try {
      DataInputStream data = new DataInputStream(
        new ByteArrayInputStream(classBytes));
      int magic = data.readInt();  // 0xcafebabe
      int minorVersion = data.readShort();
      int majorVersion = data.readShort();
      int constant_pool_count = data.readShort();
      int[] constant_pool = new int[constant_pool_count];
      for(int i = 1; i < constant_pool_count; i++) {
        int tag = data.read();
        int tableSize;
        switch(tag) {
          case 1: // UTF
            int length = data.readShort();
            char[] bytes = new char[length];
            for(int k = 0; k < bytes.length; k++)
              bytes[k] = (char)data.read();
            String imeKlase = new String(bytes);
            classNameTable.put(i, imeKlase);
            break;
```

[8] Nije jasno zašto podrazumevani konstruktor klase koja se testira mora biti javan, ali ako nije, poziv metode **newInstance( )** prouzrokuje zamrzavanje (ne generiše izuzetak).

[9] Jeremy Meyeru i meni trebalo je skoro ceo dan da to smislimo.

```
            case 5: // LONG
            case 6: // DOUBLE
              data.readLong(); // odbaciti 8 bajtova
              i++; // Potreban je poseban skok
              break;
            case 7: // CLASS
              int offset = data.readShort();
              offsetTable.put(i, offset);
              break;
            case 8: // STRING
              data.readShort(); // odbaciti 2 bajta
              break;
            case 3:  // INTEGER
            case 4:  // FLOAT
            case 9:  // FIELD_REF
            case 10: // METHOD_REF
            case 11: // INTERFACE_METHOD_REF
            case 12: // NAME_AND_TYPE
              data.readInt(); // odbaciti 4 bajta;
              break;
            default:
              throw new RuntimeException("Bad tag " + tag);
          }
        }
        short access_flags = data.readShort();
        int this_class = data.readShort();
        int super_class = data.readShort();
        return classNameTable.get(
          offsetTable.get(this_class)).replace('/', '.');
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
    // Prikaz:
    public static void main(String[] args) throws Exception {
      if(args.length > 0) {
        for(String arg : args)
          print(thisClass(BinaryFile.read(new File(arg))));
      } else
        // Prolazak kroz celo stablo:
        for(File klass : Directory.walk(".", ".*\\.class"))
          print(thisClass(BinaryFile.read(klass)));
    }
} ///:~
```

Iako ovde ne možemo ulaziti u sve pojedinosti, svaka class datoteka ima određeni format, a ja sam parčićima podataka izvadenih iz toka **ByteArrayInputStream** pokušao da dam smislena imena. Veličinu svakog parčeta možete saznati na osnovu dužine učitanog ulaznog toka. Primera radi, prva 32 bita svake class datoteke uvek zauzima „magični broj"

0xcafebabe,[10] a sledeća dva **short**a su informacija o verziji. Skladište (engl. *pool*) konstanti sadrži konstante za program i zato je promenljive veličine; sledeći **short** kazuje koliki je, da bi mu mogao biti dodeljen niz odgovarajuće veličine. Svaka stavka skladišta konstanti može biti vrednost fiksne ili promenljive veličine, pa moramo da ispitamo oznaku kojom stavka počinje da bismo shvatili šta s njom da radimo – to je naredba **switch**. Ovde nismo pokušavali da tačno analiziramo sve podatke u class datoteci, nego samo da prođemo kroz sve važne parčiće i da ih uskladištimo, pa se nemojte čuditi što se dobar deo podataka odbacuje. Informacije o klasama skladište se u tabeli **classNameTable** i tabeli **offsetTable**. Nakon učitavanja skladišta konstanti može se pronaći informacija **this_class**, što je indeks tabele **offsetTable** koja proizvodi indeks za **classNameTable** koja proizvodi ime klase.

Vratimo se programu **AtUnit.java**. Metoda **process( )** sada ima ime klase i može u njoj potražiti **.**, što znači da je deo nekog paketa. Klase izvan paketa se zanemaruju. Ukoliko je klasa u paketu, standardni učitavač klasa učitava klasu metodom **Class.forName( )**. Sada se u klasi mogu potražiti **@Unit** anotacije.

Tražimo samo tri stvari: **@Test** metode koje su uskladištene u listi **TestMethods** i metode označene sa **@TestObjectCreate** ili **@TestObjectCleanup**. Njih pronalazimo tako što pozivamo pridružene metode koje traže anotacije.

Ako se pronađe neka **@Test** metoda, ime odgovarajuće klase se štampa da bi posmatrač znao šta se događa i zatim se izvršava svaki test. To znači da se štampa ime metode, zatim poziva **createTestObject( )** koja će pozvati metodu **@TestObjectCreate** ako takva postoji ili će u protivnom pozvati podrazumevani konstruktor. Nakon pravljenja test objekta poziva se metoda koja ga testira. Ako test vrati vrednost tipa **boolean**, taj rezultat se pamti. U suprotnom, pretpostavljamo da je test uspešan ukoliko se nije pojavio izuzetak (što bi se desilo u slučaju neuspešne naredbe **assert** *ili* bilo koje druge vrste izuzetka). Ukoliko izuzetak bude generisan, štampaju se pripadajuće informacije da bi korisnik video uzrok. U svim slučajevima neuspeha (engl. *failure*) povećava se sadržaj brojača neuspeha, a ime neuspešne klase i metode dodaju se listi **failedTests** da bi bile prijavljene na kraju testiranja.

Vežba 11: (5) Alatki **@Unit** dodajte anotaciju **@TestNote**; ona označava prateće napomene (engl. *notes*) koje se ispisuju tokom testiranja.

Uklanjanje koda za testiranje

Mada u mnogim projektima neće smetati ako kôd za testiranje ostavite u programu koji se isporučuje (naročito ako sve ispitne metode označite kao privatne, što možete ako želite), u nekim slučajevima moraćete taj kôd da izbacite kako ga ne biste otkrili kupcu ili da bi verzija koja se isporučuje bila mala.

Za to je potreban inženjering bajtkoda sofisticiraniji od onog koji se može lako napisati. Međutim, biblioteka otvorenog koda Javassist[11] svodi inženjering bajtkoda u domen mogućeg. Naredni program prima opcioni indikator **-r** kao svoj prvi argument; ako zadate taj

[10] O značenju tih bajtova postoje razne legende, ali pošto su Javu pravili ljudi zaludeni računarima, verovatno su pri tom u kafiću maštali o nekoj ženi.

[11] Zahvaljujem dr Shigeruu Chibai što je napravio ovu biblioteku i za svu njegovu pomoć u pisanju programa **AtUnitRemover.java**.

indikator, on će ukloniti **@Test** anotacije, a ako to ne učinite, on će ih prikazati. I ovde se za prolazak kroz datoteke i direktorijume koje odaberete upotrebljava **ProcessFiles**:

```
//: net/mindview/atunit/AtUnitRemover.java
// Prikazuje @Unit anotacije u prevednim datotekama klasa. Ako je
// prvi argument "-r", @Unit anotacije će biti uklonjene.
// {Args: ..}
// {Neophodno za rad: javassist.bytecode.ClassFile;
// Morate instalirati biblioteku Javassist sa
// http://sourceforge.net/projects/jboss/ }
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitRemover
implements ProcessFiles.Strategy {
  private static boolean ukloni = false;
  public static void main(String[] args) throws Exception {
    if(args.length > 0 && args.equals("-r")) {
      ukloni = true;
      String[] brojarg = new String[args.length - 1];
      System.arraycopy(args, 1, brojarg, 0, brojarg.length);
      args = brojarg;
    }
    new ProcessFiles(
      new AtUnitRemover(), "class").start(args);
  }
  public void process(File cDatka) {
    boolean modifikovana = false;
    try {
      String cIme = ClassNameFinder.thisClass(
        BinaryFile.read(cDatka));
      if(!cIme.contains("."))
        return; // Klase izvan paketa se zanemaruju
      ClassPool cPool = ClassPool.getDefault();
      CtClass ctClass = cPool.get(cIme);
      for(CtMethod metoda : ctClass.getDeclaredMethods()) {
        MethodInfo mi = metoda.getMethodInfo();
        AnnotationsAttribute attr = (AnnotationsAttribute)
          mi.getAttribute(AnnotationsAttribute.visibleTag);
        if(attr == null) continue;
        for(Annotation ann : attr.getAnnotations()) {
          if(ann.getTypeName()
```

```
                .startsWith("net.mindview.atunit")) {
              print(ctClass.getName() + " Method: "
                + mi.getName() + " " + ann);
              if(ukloni) {
                ctClass.removeMethod(metoda);
                modifikovana = true;
              }
            }
          }
        }
        // Ova verzija ne uklanja polja (videti tekst).
        if(modifikovana)
          ctClass.toBytecode(new DataOutputStream(
            new FileOutputStream(cDatka)));
        ctClass.detach();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  } ///:~
```

ClassPool je neka vrsta slike svih klasa u sistemu koji modifikujete. On jemči doslednost obrade svih modifikovanih klasa. Svaku **CtClass** moramo izvaditi iz skladišta **ClassPool**, slično kao što učitavač klasa (engl. *class loader*) i metoda **Class.forName( )** učitavaju klase u JVM.

CtClass sadrži bajtkodove objekta klase i omogućuje proizvodnju informacija o toj klasi i rad s njenim kodom. Ovde sam iz svakog **CtMethod** objekta pozvao metodu **getDeclaredMethods( )** (kao Javin mehanizam refleksije) i dobio **MethodInfo**. U njima se traže anotacije. Ukoliko metoda ima anotaciju iz paketa **net.mindview.atunit**, ta metoda biva uklonjena.

Ako je klasa bila modifikovana, originalna klasa će biti zamenjena novom.

U vreme pisanja ove knjige, funkcije za „uklanjanje" su tek bile dodate u Javassist,[12] i otkrili smo da je uklanjanje **@TestProperty** polja ispalo složenije od uklanjanja metoda. Polja nije dovoljno prosto ukloniti, pošto u vezi s njima može biti statičnih operacija inicijalizacije. Zato gornja verzija koda uklanja samo **@Unit** metode. Redovno na Web lokaciji biblioteke Javassist tražite nove verzije; trebalo bi da i uklanjanje polja jednom bude moguće. U međuvremenu, imajte u vidu da spoljna metoda za testiranje prikazana u programu **AtUnitSpoljniTest.java** omogućuje uklanjanje svih testova tako što se obriše class datoteka koju pravi kôd za testiranje.

Sažetak

Anotacije su dobrodošao dodatak Javi. One su strukturiran način dodavanja metapodataka kodu, zajemčeno bezbedan u pogledu tipova, a pri tom kôd ne čini nečitljivim i zapetljanim. One mogu pomoći da se smanji dosadno pisanje deskriptora primene i drugih

[12] Dr Shigeru Chiba je veoma ljubazno, na naš zahtev, biblioteci dodao **CtClass.removeMethod( )**.

generisanih datoteka. Činjenica da je Javadoc oznaka **@deprecated** zamenjena anotacijom **@Deprecated** samo je jedan od pokazatelja koliko su anotacije prikladnije od komentara za opisivanje informacija o klasama.

Java SE5 ima tek šaku ugrađenih anotacija. To znači da ćete sami pisati anotacije i njima pridruženu logiku, ukoliko ne pronađete neku biblioteku s gotovim anotacijama. Alatkom **apt** možete u jednom koraku prevesti novogenerisane datoteke i olakšati proces automatskog prevođenja i pakovanja (*builda*), ali trenutno u API-ju **mirror** ima malo više od osnovnih funkcija za identifikovanje elemenata definicija Java klasa. Kao što ste videli, za inženjering bajtkoda može se upotrebiti Javassist ili kôd koji ćete eventualno sami napisati.

Situacija će se svakako poboljšati u ovom pogledu i proizvođači API-ja i struktura počeće da isporučuju anotacije kao deo svojih kompleta alatki. Kao što ste mogli da zaključite nakon upoznavanja sa sistemom **@Unit**, veoma je verovatno da će anotacije znatno promeniti naš doživljaj programiranja na Javi.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Paralelno izvršavanje

Dosad ste učili o sekvencijalnom programiranju. *U programu se sve odvija korak po korak.*

SEKVENCIJALNIM PROGRAMIRANJEM MOŽEMO REŠITI VELIKI PODSKUP PROGRAMERSKIH zadataka. Međutim, neke probleme je podesnije ili čak neophodno rešavati paralelnim izvršavanjem više delova programa, tako da izgleda da se ti delovi izvršavaju istovremeno ili to zaista i jeste tako, ukoliko računar ima više procesora ili procesor ima više jezgara.

Paralelnim programiranjem može se značajno ubrzati izvršavanje programa, dobiti lakši model za projektovanje određenih vrsta programa, ili i jedno i drugo. Međutim, izveštiti se u teoriji i tehnikama paralelnog programiranja predstavlja viši stepen sposobnosti od svega što ste dosad saznali u ovoj knjizi. To je tema za srednji ili napredni nivo stručnosti. Ovo dugačko poglavlje može poslužiti samo kao uvod, pa nikako nemojte smatrati da je temeljno ovladavanje znanjem iz ovog poglavlja dovoljno za pisanje dobrih paralelnih programa.

Kao što ćete videti, pravi problem pri paralelnom izvršavanju nastaje kada zadaci koji se paralelno izvršavaju počnu međusobno da se ometaju. To se može desiti na tako suptilan i slučajan način da je verovatno pošteno reći kako je paralelno izvršavanje „u teoriji determinističko, ali u praksi stohastičko". Drugim rečima, ukoliko se pažljivo radi i pregleda kôd, moguće je napisati programe za paralelno izvršavanje koji rade ispravno. Međutim, u praksi je mnogo lakše pisati programe za paralelno izvršavanje koji samo naizgled rade, a u određenim okolnostima zakazuju. Te okolnosti se ne moraju dogoditi ili se mogu događati toliko retko da ih tokom testiranja uopšte ne vidite. Zapravo, možda ne možete da napišete kôd za testiranje koji generiše okolnosti u kojima vaš program za paralelno izvršavanje otkazuje. Otkazivanja se često dešavaju tek povremeno i zato se o njima sazna tek iz žalbi korisnika. To je jedan od najjačih razloga za proučavanje paralelnog izvršavanja: ako ga preskočite, verovatno će vam se osvetiti.

Dakle, paralelno izvršavanje je prepuno opasnosti, i ako vas to plaši, tako i treba. Iako je Java SE5 donela znatna poboljšanja u paralelnom izvršavanju, i dalje nema zaštite koja bi vam ukazala na grešku, kao što su provera u vreme prevođenja ili izuzeci koji se proveravaju. S paralelnim izvršavanjem morate sami da se nosite, i pouzdan višenitni kôd možete da pišete na Javi samo ako ste istovremeno sumnjičavi i agresivni.

Ima mišljenja da je paralelno izvršavanje preteška tema za knjigu čiji je stručni nivo početni. Ti ljudi smatraju da je paralelno izvršavanje zasebna tema koja se može obraditi nezavisno, a nekoliko slučajeva koji se pojavljuju u svakodnevnom programiranju (kao što su grafička korisnička okruženja) mogu biti obrađeni posebnim idiomima. Zašto načinjati toliko složenu temu ako se to može izbeći?

Eh, kada bi to bila istina. Nažalost, ne biramo mi kada će se u našim Java programima pojaviti niti. To što sami niste otpočeli nit ne znači da možete izbeći pisanje višenitnog koda. Na primer, na Javi se najčešće pišu Web sistemi, a osnovna klasa Web biblioteke, servlet, po prirodi je višenitna – i mora da bude pošto Web serveri često imaju više procesora, a paralelno izvršavanje je idealan način upotrebe tih procesora. Koliko god da servlet izgleda jednostavno, morate razumeti paralelno izvršavanje da biste servlete ispravno koristili.

Isto važi i za programiranje grafičkih korisničkih okruženja, kao što ćete videti u sledećem poglavlju. Iako biblioteke Swing i SWT imaju mehanizme za bezbednost niti, teško je shvatiti kako da ih ispravno upotrebljavate ako ne razumete paralelno izvršavanje.

Java je višenitni jezik i problemi s paralelnim izvršavanjem postoje, znali vi to ili ne. Zato postoji mnogo Java programa koji rade ili slučajnim sticajem okolnosti ili tek veći deo vremena, i povremeno misteriozno zakazuju zbog neotkrivenih grešaka u paralelnom izvršavanju. Katkada su ta zakazivanja dobroćudna, ali umeju da prouzrokuju gubljenje vrednih podataka, i ako barem niste čuli za probleme s paralelnim izvršavanjem, mislićete da je problem u svemu pre nego u vašem softveru. Ta vrsta problema može izroniti ili se pojačati kada se program premesti na višeprocesorski računar. Nakon što proučite ovo poglavlje, trebalo bi da znate kako prividno ispravni programi mogu zbog paralelnog izvršavanja da ispolje neispravno ponašanje.

Kada počnete da se bavite paralelnim programiranjem, kao da ste otišli u stranu zemlju da učite njen jezik ili barem potpuno nov skup jezičkih pojmova. Naučiti paralelno programiranje jednako je zahtevno kao naučiti objektno orijentisano programiranje. Ako se potrudite, shvatićete osnovni mehanizam, ali za pravo ovladavanje tim predmetom po pravilu je potrebno dobro zagrejati stolicu. Iz ovog poglavlja steći ćete osnovno znanje o paralelnom izvršavanju, pa ćete razumeti pojmove i moći ćete da pišete umereno složene višenitne programe. Budite svesni da je veoma lako steći neosnovano veliko poverenje u svoje sposobnosti za paralelno programiranje. Moraćete da pročitate knjige posvećene isključivo toj temi ukoliko se odlučite za pisanje iole složenijih višenitnih programa.

Svi aspekti paralelnog izvršavanja

Paralelno programiranje je teško shvatljivo zato što se njime istovremeno mora rešiti više problema. Uz to, postoji više načina realizacije paralelnog rada, a da nam nije jasno koji problem se rešava kojim pristupom. (Često su nejasne i granice između njih.) Zato morate da shvatite sve probleme i specijalne slučajeve da biste efikasno upotrebljavali paralelno izvršavanje.

Probleme koje rešavamo paralelnim izvršavanjem grubo delimo na „brzinu izvršavanja“ i „upravljivost projekta“.

Brže izvršavanje

Spočetka problem zvuči jednostavno: ako želite da se program izvršava brže, podelite ga na niti koje izvršavaju zasebni procesori. Paralelno izvršavanje je osnovna alatka višeprocesorskog programiranja. Pošto Murov zakon sve ređe deluje (barem za obična integrisana kola), sredstvo za povećanje brzine postaju procesori s više jezgara, a ne brža integrisana kola. Da biste naterali programe da rade brže, moraćete naučiti da iskoristite te dodatne procesore, i to je jedna od stvari koje će vam paralelno izvršavanje omogućiti.

Ako imate računar s više procesora, njima se može dodeliti više zadataka (na istovremeno izvršavanje), što ume znatno da poboljša protok. Tako često rade snažni višeprocesorski Web serveri; oni procesorima raspoređuju veliki broj korisničkih zahteva u programu koji dodeljuje jednu nit po zahtevu.

Međutim, paralelno izvršavanje često poboljšava performanse programa koji se izvršavaju na *jednom* procesoru.

Ovo zvuči neverovatno ili barem neočekivano. Razmislite: paralelni program koji se izvršava na jednom procesoru trebalo bi da ima *veće* režijske troškove (slabije performanse) nego u slučaju da se svi delovi programa izvršavaju sekvencijalno, jer se mora platiti dodatno *prebacivanje konteksta* (prelazak s jednog zadatka na drugi). Pri površnom razmatranju, reklo bi se da je jeftinije sve delove programa izvršavati kao jedan zadatak i uštedeti troškove prebacivanja konteksta.

Ali uzmite u obzir *blokiranje*. Ukoliko se jedan zadatak u vašem programu ne izvršava zbog neke okolnosti izvan kontrole programa (obično je to U/I), kažemo da taj zadatak ili nit *blokira* izvršavanje programa. Ako nema paralelnog izvršavanja, ceo program se zaustavlja dok se spoljne okolnosti ne promene. S druge strane, ako je program napisan za paralelno izvršavanje, ostali zadaci u programu mogu se izvršavati dok je jedan zadatak blokiran, pa program u celini nastavlja rad. Zapravo, sa stanovišta performansi, nema smisla pisati program za paralelno izvršavanje na jednoprocesorskom računaru ukoliko nijedan od njegovih zadataka ne može da zablokira izvršavanje programa.

Performanse jednoprocesorskih računara često poboljšavamo *programima kojima upravljaju događaji*. Više niti se koristi baš zato da bi se napravilo korisničko okruženje koje brzo reaguje. Ukoliko program obavlja neke dugotrajne operacije i stoga zanemaruje ono što unosi korisnik, on sporo reaguje na sve što korisnik radi. Dobar primer za to je dugme „izlaz" – ne želite da u svakom delu koda unutar celog programa ispitujete da li je korisnik pritisnuo to dugme. Time se dobija nezgrapan kôd, bez ikakvog jemstva da programer u nekom delu programa neće zaboraviti da proveri šta radi korisnik. Ako nema paralelnog izvršavanja, korisničko okruženje koje brzo reaguje ostvaruje se samo kada svi zadaci periodično proveravaju korisnikov unos. Određenu brzinu reagovanja program jemči tako što pravi zasebnu nit izvršavanja koja reaguje na korisnikov unos, iako će ta nit gotovo neprestano biti blokirana.

Program treba da nastavi izvršavanje svojih operacija i istovremeno da vrati kontrolu korisničkom okruženju kako bi mogao da reaguje na radnje korisnika. Ali obična metoda ne može da se izvršava i da istovremeno vrati kontrolu nad procesorom ostatku programa. To zvuči nemoguće – kao da procesor treba da bude na dva mesta u isto vreme, ali upravo to i jeste iluzija koju pruža rad s više niti. (U slučaju višeprocesorskih sistema, to i nije iluzija.)

Veoma jednostavan način da se na nivou operativnog sistema realizuje paralelno izvršavanje jeste upotreba *procesa*. Proces je svaki samostalan program koji se izvršava i ima vlastiti adresni prostor. *Višeprogramski* (engl. *multitasking*) operativni sistem može da izvršava više procesa (programa) istovremeno, a da pri tom izgleda kao da se svaki izvršava sam za sebe; to se postiže raspodelom procesorskog vremena na sve tekuće procese. Procesi su veoma privlačni zato što ih operativni sistem najčešće izoluje jedne od drugih da se ne bi uzajamno ometali, pa je programiranje s njima relativno lako. Za razliku od toga, sistemi za paralelno izvršavanje – kao što je Javin – dele resurse (memoriju i U/I) među nitima, pa je osnovna poteškoća pisanja višenitnih programa koordinacija upotrebe tih resursa u različitim zadacima vođenim nitima, kako bi im u svakom trenutku pristupao samo jedan zadatak.

Evo jednostavnog primera procesa u operativnom sistemu. Dok pišem knjigu, redovno pravim više redundantnih rezervnih kopija tekućeg stanja knjige. Jednu kopiju napravim u lokalnom direktorijumu, drugu na fleš uređaju, treću na Zip disku i četvrtu na udaljenoj FTP lokaciji. Za automatizaciju tog postupka napisao sam mali program (na Pythonu, ali koncepti su isti) koji komprimuje knjigu u datoteku čije ime sadrži broj verzije i zatim kopira. Spočetka sam sve kopije pravio sekvencijalno (čekao da se prethodno kopiranje završi pa da otpočnem sledeće). Potom sam shvatio da operacije kopiranja ne traju jednako, jer se U/I brzine medijuma razlikuju. Pošto sam imao višeprogramski operativni sistem, svaku operaciju kopiranja mogao sam da započnem kao zaseban proces i pustim ih da se izvršavaju paralelno – tako se ubrzavalo izvršavanje celog programa. Dok je jedan proces blokiran, drugi mogu da rade.

To je idealan primer paralelnog izvršavanja. Svaki zadatak se izvršava kao proces u sopstvenom adresnom prostoru, pa zadaci ne mogu da utiču jedni na druge. Još je važnije to što zadaci nemaju *potrebe* da komuniciraju, pošto su potpuno nezavisni. Operativni sistem se stara o svim pojedinostima koje obezbeđuju ispravno kopiranje datoteka. Stoga nema rizika i dobijamo brži program, zapravo besplatno.

Neki zagovaraju procese kao jedini razuman pristup paralelnom izvršavanju,[1] ali je nažalost broj procesa po pravilu ograničen, a njihovi režijski troškovi veliki. Zato se primenom procesa ne mogu rešiti svi zadaci iz spektra paralelnog izvršavanja.

Neki programski jezici međusobno izoluju zadatke koji se izvršavaju paralelno. Obično ih nazivamo *funkcijski jezici*, jer u njima poziv funkcije ne prouzrokuje sporedne uticaje (te ne može ni da utiče na druge funkcije) i stoga se može izvršavati kao zaseban zadatak. Jedan od takvih jezika je *Erlang*, a on obuhvata mehanizam sefa za komunikaciju jednog zadatka s drugim. Ukoliko shvatite da u delu vašeg programa mora intenzivno da se koristi paralelno izvršavanje, a pri pisanju tog dela nailazite na velike probleme, razmislite o ideji da taj deo programa napišete na namenskom jeziku za paralelno izvršavanje kao što je Erlang.

U Javi je pristup tradicionalniji – podrška za višenitni rad dodata je na gotov sekvencijalni jezik.[2] Umesto da se „love" spoljni procesi u višeprogramskom operativnom sistemu, niti prave zadatke *unutar* istog procesa koji predstavlja program koji se izvršava. Jedna od prednosti koje se pri tome postižu jeste nezavisnost od operativnog sistema, što je bio važan cilj prilikom projektovanja Jave. Na primer, pre verzije OS X operativnog sistema Macintosh (prilično važna publika za prve verzije Jave) nisu podržavale višeprogramski rad. Da višenitni rad nije dodat u Javu, Java programi za paralelno izvršavanje ne bi mogli da se izvršavaju na Macintoshu i njemu sličnim platformama, čime bi se narušio zahtev „napiši jednom/izvršavaj svuda".[3]

---

[1] Primera radi, Eric Raymond navodi jake razloge za to u svojoj knjizi *The Art of UNIX Programming* (Addison-Wesley, 2004).

[2] Neki smatraju da je svaki pokušaj da se paralelno izvršavanje prihefta uz sekvencijalni jezik osuđen na neuspeh, pa vi sami zaključite šta je istina.

[3] Ovaj zahtev nikada nije bio potpuno ispunjen i Sun ga više ne ističe tako glasno. Da ironija bude veća, jedan od razloga što koncept „napiši jednom/izvršavaj posvuda" nije potpuno uspeo možda je posledica problema u sistemu niti – koji će možda biti rešeni u Javi SE5.

Poboljšan dizajn koda

Program koji izvršava više zadataka na računaru s jednim procesorom ipak u svakom trenutku radi samo jedan posao, pa teorijski mora biti moguće da se isti program napiše bez svih tih zadataka. Međutim, paralelno programiranje daje važnu organizacionu prednost: dizajn programa se može znatno pojednostaviti. Neke vrste zadataka, kao što je simulacija, teško je rešiti bez paralelnog programiranja.

Većina ljudi je videla barem jedan oblik simulacije, bilo kao računarsku igricu ili računarski generisanu animaciju u filmovima. Simulacije po pravilu obuhvataju mnogo elemenata u interakciji, svaki sa „svojom pameću". Mada se svako može uveriti da u raćunaru s jednim procesorom svaki element simulacije izvršava taj jedini procesor, sa stanovišta programiranja mnogo je lakše praviti se da svaki element simulacije ima sopstveni procesor i da predstavlja nezavisan zadatak.

Potpuna simulacija često sadrži veoma veliki broj zadataka, zato što svaki element simulacije može da dejstvuje nezavisno – to važi i za vrata i stene, ne samo za duhove i čarobnjake. Višenitni sistemi često imaju relativno mali broj dostupnih niti, katkada reda veličine nekoliko desetina ili stotina. Taj broj može da se menja nezavisno od kontrole programa – može zavisiti od platforme, ili u slučaju Jave, od verzije JVM-a. Kad koristite Javu, obično možete pretpostaviti da broj dostupnih niti neće biti dovoljan da svakom elementu velike simulacije date sopstvenu nit.

Ovaj problem se obično rešava *kooperativnim višenitnim programiranjem* (engl. *cooperative multithreading*). Java upotrebljava niti *prednapredno* (engl. *preemptive*), što znači da mehanizam raspoređivanja svim nitima dodeljuje procesorsko vreme, periodično prekida svaku nit i prebacuje kontekst na drugu nit, tako da svaka nit dobija umerenu količinu vremena za izvršavanje svog zadatka. U kooperativnom sistemu, svaki zadatak dobrovoljno predaje kontrolu sistemu, zbog čega programer u svaki zadatak mora da umetne neku vrstu naredbe za prepuštanje kontrole. Prednost kooperativnog sistema je dvostruka: prebacivanje konteksta je obično mnogo jeftinije nego u preduprednom sistemu, i teorijski nema ograničenja broja nezavisnih zadataka koji se mogu izvršavati istovremeno. Kada simulacija ima mnogo elemenata, ovo može da bude idealno rešenje. Imajte u vidu da neki kooperativni sistemi ne raspoređuju zadatke po procesorima, što je veliko ograničenje.

S druge strane, paralelno izvršavanje je veoma koristan model – zato što se ono upravo i dešava – u radu sa savremenim sistemima koji razmenjuju *poruke* i mogu obuhvatati mnogo nezavisnih računara razbacanih u mreži. U tom slučaju, svi procesi se odvijaju nezavisno jedan od drugog i nema mogućnosti čak ni za deljenje resursa. Međutim, i dalje morate sinhronizovati prenos informacija između procesa, da ceo sistem za razmenu poruka ne bi izgubio informaciju ili je usvojio u neodgovarajuće vreme. Čak i ako nemate nameru da u neposrednoj budućnosti mnogo upotrebljavate paralelno programiranje, dobro je upoznati ga kako biste mogli da shvatite arhitekture sistema za razmenu poruka koji postaju dominantni način pravljenja distribuiranih sistema.

Paralelno programiranje nameće troškove, među kojima i troškove zbog složenosti, ali njih po pravilu opravdavaju poboljšanja u dizajnu programa, uravnoteženom korišćenju resursa i lakoći upotrebe. Uopšte uzev, niti omogućuju pravljenje labavije povezane strukture programa, inače bi delovi programa morali stalno da obraćaju pažnju na zadatke koje obavljaju niti.

Osnove višenitnog programiranja

Paralelno programiranje omogućuje deljenje programa na zasebne delove koji se izvršavaju nezavisno. U višenitnom programu, svaki od tih zasebnih zadataka (nazivaju ih i *podzadaci*) izvršava jedna *nit izvršavanja* (engl. *thread of execution*). *Nit* je pojedinačan sekvencijalni tok kontrole unutar nekog procesa. Proces može da sadrži više zadataka koji se paralelno izvršavaju, ali program se piše kao da svaki zadatak ima CPU samo za sebe. Postoji pozadinski mehanizam koji svaki čas prebacuje procesor s jednog zadatka na drugi, ali vi o tome po pravilu ne treba da brinete.

Model niti je olakšica za programiranje koja pojednostavljuje istovremeni rad s nekoliko operacija u istom programu. Niti omogućavaju da procesor ne bude trajno zauzet jednim procesom, nego da svakoj niti posveti malo vremena.[4] Svaka nit misli da koristi procesor sama – u stvari, procesorsko vreme je podeljeno. Izuzetak od ovog pravila je izvršavanje programa na računaru s više procesora. Ali, jedna od velikih prednosti korišćenja niti leži baš u tome da ne morate misliti o hardveru na kome će se program izvršavati, pa u kodu ne treba praviti varijante za jedan ili više procesora. Stoga su niti sredstvo za pravljenje prenosivih programa koji se sami prilagođuju platformi – ako se program izvršava presporo, lako ćete ga ubrzati dodavanjem procesora u računar. Upotreba višeprogramskog (engl. *multitasking*) operativnog sistema i istovremeno izvršavanje više niti unutar jednog programa (engl. *multithreading*) predstavlja najbolji način da iskoristite višeprocesorski računar.

Definisanje zadataka

Nit izvršava jedan zadatak, pa moramo imati načina da taj zadatak opišemo. Tome služi interfejs **Runnable**. Zadatak ćete definisati kada realizujete **Runnable** i napišete metodu **run( )** koja obavlja ono što zadatak treba da uradi.

Primera radi, naredni zadatak **Lansiranje** prikazuje odbrojavanje pre lansiranja:

```
//: paralelno/Lansiranje.java
// Primer interfejsa Runnable.

public class Lansiranje implements Runnable {
  protected int odbrojavanje = 10; // Podrazumevana
  private static int brojZadatka = 0;
  private final int id = brojZadatka++;
  public Lansiranje() {}
  public Lansiranje(int odbrojavanje) {
    this.odbrojavanje = odbrojavanje;
  }
  public String status() {
    return "#" + id + "(" +
      (odbrojavanje > 0 ? odbrojavanje : "Lansiranje!") + "), ";
```

[4] Ovo važi kada sistem deli procesorsko vreme (tako radi Windows, na primer). Solaris radi po FIFO modelu paralelnog izvršavanja: ukoliko se ne probudi nit s višim prioritetom, tekuća nit se izvršava sve dok se ne zablokira ili se ne završi. To znači da se ostale niti istog prioriteta ne izvršavaju sve dok im tekuća nit ne prepusti procesor.

```
  }
  public void run() {
    while(odbrojavanje-- > 0) {
      System.out.print(status());
      Thread.yield();
    }
  }
} ///:~
```

Instance zadatka se razlikuju po pripadajućem identifikatoru **id**. Uz njega je rezervisana reč **final** zato što ne očekujemo da će se promeniti nakon što bude inicijalizovan.

Metoda **run( )** obično ima neku vrstu petlje koja se izvršava sve dok zadatak ne postane nepotreban, pa morate da utvrdite uslov za izlazak iz nje. Jedna mogućnost bi bila da jednostavno izađete (vratite se) iz metode **run( )**. Metodu **run( )** često pišu u obliku beskonačne petlje, što znači da će se ona izvršavati u nedogled ako je nešto ne prekine. (U nastavku poglavlja saznaćete kako da bezbedno okončate zadatak.)

Poziv statičke metode **Thread.yield( )** unutar metode **run( )** predstavlja sugestiju *mehanizmu za raspodelu procesorskog vremena nitima*, engl. *thread schedular* (koji prebacuje procesor s jedne niti na drugu) koja kazuje: „Obavio sam važne delove svog ciklusa i sada je vreme za prebacivanje na drugi zadatak". Metoda je neobavezna (opciona), a ovde sam je upotrebio zato što u navedenim primerima daje zanimljivije rezultate: ona povećava verovatnoću da će doći do prebacivanja s jednog zadatka na drugi.

U narednom primeru, zadatkovu metodu **run( )** ne izvršava zasebna nit; nju jednostavno poziva metoda **main( )**. Zapravo, to *jeste* zasebna nit: ona koja se uvek dodeljuje metodi **main( )**:

```
//: paralelno/GlavnaNit.java

public class GlavnaNit {
  public static void main(String[] args) {
    Lansiranje lansiraj = new Lansiranje();
    lansiraj.run();
  }
} /* Ispis:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
#0(Lansiranje!),
*///:~
```

Svaka klasa izvedena iz interfejsa **Runnable** mora da ima metodu **run( )**, ali to nije ništa osobito – time se ne dobija naročita sposobnost višenitnog rada. Nju ćete postići eksplicitnim dodeljivanjem zadatka nekoj niti.

Klasa **Thread**

Runnable objekat se pretvara u zadatak koji se izvršava najčešće tako što objekat prosledite konstruktoru klase **Thread**. U sledećem primeru ćemo pomoću klase **Thread** pokrenuti izvršavanje objekta tipa **Lansiranje**:

```
//: paralelno/OsnoveNiti.java
// Najjednostavnija upotreba klase Thread.

public class OsnoveNiti {
  public static void main(String[] args) {
    Thread t = new Thread(new Lansiranje());
    t.start();
    System.out.println("Čekamo na Lansiranje");
  }
} /* Ispis: (90% podudaranja)
Čekamo na Lansiranje
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
#0(Lansiranje!),
*///:~
```

Konstruktoru klase **Thread** treba samo objekat tipa **Runnable**. Nit se inicijalizuje pozivom metode **start( )** objekta tipa **Thread**, a zatim se pozivom metode **run( )** objekta tipa **Runnable** pokreće zadatak u novoj niti. Mada izgleda kao da **start( )** poziva neku metodu koja se izvršava dugo, iz ispisa rezultata možete videti – poruka „Čekamo na Lansiranje" ispisuje se pre dovršenja odbrojavanja – da se metoda **start( )** vraća brzo. U stvari, pozvana je metoda **Lansiranje.run( )** koja se još nije završila, ali pošto se **Lansiranje.run( )** izvršava u drugoj niti, u metodi **main( )** i dalje možemo da izvršavamo ostale operacije. (Ta sposobnost nije ograničena na nit metode **main( )** – svaka nit može da pokrene druge niti.) Dakle, program izvršava dve metode istovremeno – **main( )** i **Lansiranje.run( )**. Kôd metode **run( )** izvršava se istovremeno sa ostalim nitima programa.

Lako je dodati još niti za izvršavanje drugih zadataka. U narednom primeru vidite sve zadatke u uzajamno usklađenom izvršavanju:[5]

```
//: paralelno/JosOsnovaNiti.java
// Dodavanje niti.

public class JosOsnovaNiti {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new Thread(new Lansiranje()).start();
    System.out.println("Čekamo na Lansiranje");
  }
} /* Ispis: (primer)
Čekamo na Lansiranje
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7),
#1(7), #2(7), #3(7), #4(7), #0(6), #1(6), #2(6), #3(6), #4(6), #0(5), #1(5),
#2(5), #3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3), #2(3),
#3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1),
#4(1), #0(Lansiranje!), #1(Lansiranje!), #2(Lansiranje!), #3(Lansiranje!),
#4(Lansiranje!),
*///:~
```

[5] U ovom slučaju, jedna nit – **main( )** – pravi sve niti za **Lansiranje**. Međutim, ukoliko imate više niti koje prave niti za **Lansiranje**, više objekata tipa **Lansiranje** može imati isti **id**. Razlog za to saznaćete u nastavku poglavlja.

Ispis rezultata pokazuje da je izvršavanje raznih zadataka izmešano kako se procesor prebacuje s jedne niti na drugu. To prebacivanje automatski kontroliše mehanizam za raspodelu vremena nitima. Ako računar ima više procesora, mehanizam za raspodelu će te niti (za korisnika nevidljivo) raspodeliti procesorima.[6]

Rezultati jednog izvršavanja ovog programa neće biti isti kao rezultati drugog, pošto mehanizam za raspodelu procesorskog vremena nitima nije deterministički. U stvari, različite verzije JDK prave velike razlike u rezultatima ovog jednostavnog programa. Na primer, jedna od ranijih JDK nije prebacivala procesor baš često, pa se dešavalo da nit 1 završi svoju petlju, zatim nit 2 prođe kroz sve svoje petlje itd. To je bilo isto kao poziv rutine koja sekvencijalno izvršava jednu petlju za drugom, sem što je pokretanje niti skuplje. Kasnije su JDK bolje delile procesorsko vreme, pa su sve niti bivale redovnije opsluživane. Sun po pravilu nije ni spominjao takve promene u ponašanju JDK, pa ne možete računati na bilo kakvo dosledno ponašanje višenitnog mehanizma. Najbolje će biti da prilikom pisanja višenitnog koda ne pretpostavljate ništa o ponašanju niti.

Kada funkcija **main( )** napravi niti (objekte klase **Thread**), ona nigde ne pamti njihove reference. Obični objekti bi bili proglašeni za smeće, ali to ne važi za niti. Svaka nit se „registruje" tako da negde postoji njena referenca i skupljač smeća ne može da je ukloni dok zadatak ne izađe iz svoje metode **run( )** i ne umre. Iz ispisa rezultata vidite da se zadaci zaista izvršavaju sve do svog završetka. Dakle, svaka nit pravi zasebnu nit izvršavanja koja postoji i nakon završetka metode **start( )**.

Vežba 1: (2) Realizujte interfejs **Runnable**. U metodi **run( )** ispišite neku poruku i zatim pozovite **yield( )**. Ponovite to triput i zatim izađite (vratite se) iz metode **run( )**. U konstruktor stavite poruku koja se ispisuje prilikom pokretanja, a odgovarajuću poruku ispišite i kada se zadatak izvrši. Napravite više takvih zadataka i izvršite ih pomoću niti.

Vežba 2: (2) Po uzoru na program **genericki/Fibonacci.java**, napišite zadatak koji proizvodi sekvencu od **n** Fibonaccijevih brojeva, pri čemu se **n** prosleđuje konstruktoru zadatka. Napravite više takvih zadataka i izvršite ih pomoću niti.

Upotreba izvršilaca (interfejsa **Executor**)

Java SE5 u paketu **java.util.concurrent** sadrži klasu *Executors*, tj. izvršioce koji pojednostavljuju paralelno programiranje tako što iniciraju izvršavanje niti (objekata tipa **Thread**) i upravljaju njime. Izvršioci obezbeđuju sloj indirekcije između klijenta i izvršavanja zadatka; umesto da klijent izvršava zadatak neposredno, njega izvršava posrednički objekat, tj. izvršilac (engl. *executor*). Klasa **Executors** omogućuje da upravljamo izvršavanjem asinhronih zadataka, a da ne moramo eksplicitno da upravljamo životnim ciklusom niti. Upravo su izvršioci preporučeni način pokretanja zadataka u Javi SE5/6.

Umesto da u programu **JosOsnovaNiti.java** niti pravimo eksplicitno, možemo pustiti izvršioca da se stara o svemu. Objekat tipa **Lansiranje** ume da izvrši određeni zadatak; kao i projektni obrazac *Command* (Komanda), on eksponira samo jednu metodu za izvršavanje. Interfejs **ExecutorService** (koji proširuje interfejs **Executor** tako što dodaje metode za ceo životni ciklus usluge, što znači da zna npr. i da se ugasi – engl. *shutdown*) ume da napravi kontekst podesan za izvršavanje **Runnable** objekata. U narednom

[6] U prvim verzijama Jave nije bilo tako.

primeru, klasa **CachedThreadPool** pravi po jednu nit za svaki zadatak. Obratite pažnju na to da se objekat tipa **ExecutorService** pravi statičkom metodom klase **Executors**; ona određuje koja će vrsta izvršioca biti napravljena:

```
//: paralelno/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Lansiranje());
    exec.shutdown();
  }
} /* Ispis: (primer)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8),
#0(6), #1(7), #2(7), #3(7), #4(7), #0(5), #1(6), #2(6), #3(6), #4(6), #0(4),
#1(5), #2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2), #1(3),
#2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Lansiranje!),
#1(1), #2(1), #3(1), #4(1), #1(Lansiranje!), #2(Lansiranje!),
#3(Lansiranje!), #4(Lansiranje!),
*///:~
```

Veoma često, samo jednim izvršiocem možete da napravite sve zadatke u sistemu i da svima njima upravljate.

Poziv metode **shutdown( )** sprečava dodeljivanje novih zadataka tom izvršiocu. Tekuća nit – u ovom slučaju, ona koja izvršava **main( )** – nastaviće da izvršava sve zadatke prijavljene pre poziva metode **shutdown( )**. Program će prekinuti izvršavanje čim se završe svi zadaci u tom izvršiocu.

CachedThreadPool iz prethodnog primera lako možete zameniti nekom drugom vrstom izvršioca. Za izvršavanje prijavljenih zadataka, **FixedThreadPool** koristi ograničen skup niti:

```
//: paralelno/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
  public static void main(String[] args) {
    // Argument konstruktora je broj niti:
    ExecutorService exec = Executors.newFixedThreadPool(5);
    for(int i = 0; i < 5; i++)
      exec.execute(new Lansiranje());
    exec.shutdown();
  }
} /* Ispis: (primer)
```

```
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8),
#0(6), #1(7), #2(7), #3(7), #4(7), #0(5), #1(6), #2(6), #3(6), #4(6), #0(4),
#1(5), #2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2), #1(3),
#2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Lansiranje!),
#1(1), #2(1), #3(1), #4(1), #1(Lansiranje!), #2(Lansiranje!), #3(Lansiranje!),
#4(Lansiranje!),
*///:~
```

Za **FixedThreadPool** se skupo dodeljivanje niti obavlja samo jednom – na početku – čime se broj niti ograničava. Štedi se vreme, zato što se niti ne prave stalno za svaki zadatak. Pored toga, u sistemu vođenom događajima, rukovaoci događajem (engl. *event handlers*) kojima zatrebaju niti mogu biti odmah opsluženi tako što jednostavno uzimaju niti iz grupe (engl. *pool*). Ne možete preterati s korišćenjem resursa zato što **FixedThreadPool** upotrebljava ograničen broj objekata tipa **Thread**.

Imajte u vidu da se postojeće niti u svim grupama automatski ponovo upotrebljavaju čim to postane moguće.

Iako ću ja u ovoj knjizi upotrebljavati neograničene grupe niti (**CachedThreadPool**), razmislite o korišćenju ograničenih grupa (**FixedThreadPool**) u kodu koji se isporučuje kupcima. **CachedThreadPool** obično pravi onoliko niti koliko mu je potrebno tokom izvršavanja programa, a zatim prestaje da pravi nove niti zato što reciklira stare; stoga je najbolje da on bude izabran kao prvi izvršilac. Na **FixedThreadPool** treba da pređete tek ako prethodni pristup napravi probleme.

SingleThreadExecutor je kao **FixedThreadPool** čija je veličina ograničena na jednu nit.[1] To je podesno za kontinualne (dugotrajne) zadatke, kao što je zadatak koji osluškuje priključke ulaznih utičnica. Prikladno je i za kratke zadatke koje želite da izvršite u niti – recimo, male zadatke koji ažuriraju lokalni ili udaljeni dnevnik (engl. *log*) ili za nit koja raspoređuje događaje.

Kada se izvršiocu tipa **SingleThreadExecutor** prijavi više zadataka, oni se smeštaju u red za čekanje i svaki zadatak se poziva tek kada prethodni potpuno završi rad, a svi koriste istu nit. U narednom primeru, videćete da svaki zadatak, po redosledu prijavljivanja, biva dovršen pre otpočinjanja sledećeg. Dakle, **SingleThreadExecutor** serijalizuje zadatke koji mu se prijave i održava sopstveni (skriveni) red zadataka koji čekaju na izvršenje.

```
//: paralelno/SingleThreadExecutor.java
import java.util.concurrent.*;

public class SingleThreadExecutor {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    for(int i = 0; i < 5; i++)
      exec.execute(new Lansiranje());
    exec.shutdown();
```

[1] Jemči i nešto što ostali izvršioci ne mogu – ne može se desiti da dva zadatka pozove na paralelno izvršavanje. Time se menjaju zahtevi za zaključavanje zadataka (o kojima ću govoriti u nastavku poglavlja).

```
    }
  } /* Ispis:
  #0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
  #0(Lansiranje!), #1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2),
  #1(1), #1(Lansiranje!), #2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3),
  #2(2), #2(1), #2(Lansiranje!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4),
  #3(3), #3(2), #3(1), #3(Lansiranje!), #4(9), #4(8), #4(7), #4(6), #4(5),
  #4(4), #4(3), #4(2), #4(1), #4(Lansiranje!),
  *///:~
```

Kao drugi primer, pretpostavimo da više niti izvršava zadatke koji koriste sistem datoteka. Za te zadatke možete upotrebiti izvršilac tipa **SingleThreadExecutor** koji jemči da se u svakom trenutku u svakoj niti izvršava samo po jedan zadatak. U tom slučaju ne morate da se bavite sinhronizovanjem deljenog resursa (a ni sistem datoteka nećete zabrljati u međuvremenu). Ponekad je bolje rešenje sinhronizovanje resursa (o čemu ćete više saznati u nastavku poglavlja), ali **SingleThreadExecutor** omogućuje da preskočite pravilnu koordinaciju kada želite samo da napravite neki prototip. Serijalizacijom zadataka možete da uklonite potrebu za serijalizacijom objekata.

Vežba 3: (1) Ponovite vežbu 1 s raznim vrstama izvršilaca predstavljenim u ovom odeljku.

Vežba 4: (1) Ponovite vežbu 2 s raznim vrstama izvršilaca predstavljenim u ovom odeljku.

Dobijanje povratnih vrednosti od zadataka

Zadatak koji realizuje interfejs **Runnable** može da obavi posao, ali ne može da vrati rezultat. Ukoliko želite da zadatak vrati neki rezultat kada završi s radom, realizujte interfejs **Callable**, a ne **Runnable**. **Callable** je novina u Javi SE5; to je generički interfejs čiji parametar tipa predstavlja povratnu vrednost metode **call**( ) – a ne **run**( ). Morate ga pozvati metodom **submit**( ) interfejsa **ExecutorService**. Sledi jednostavan primer:

```
//: paralelno/PrimerZaCallable.java
import java.util.concurrent.*;
import java.util.*;

class ZadatakSRezultatom implements Callable<String> {
  private int id;
  public ZadatakSRezultatom(int id) {
    this.id = id;
  }
  public String call() {
    return "rezultat ZadatkaSRezultatom " + id;
  }
}

public class PrimerZaCallable {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    ArrayList<Future<String>> rezultati =
```

```
      new ArrayList<Future<String>>();
    for(int i = 0; i < 10; i++)
      rezultati.add(exec.submit(new ZadatakSRezultatom(i)));
    for(Future<String> fs : rezultati)
      try {
        // get() blokira do dovršenja:
        System.out.println(fs.get());
      } catch(InterruptedException e) {
        System.out.println(e);
        return;
      } catch(ExecutionException e) {
        System.out.println(e);
      } finally {
        exec.shutdown();
      }
  }
} /* Ispis:
rezultat ZadatkaSRezultatom 0
rezultat ZadatkaSRezultatom 1
rezultat ZadatkaSRezultatom 2
rezultat ZadatkaSRezultatom 3
rezultat ZadatkaSRezultatom 4
rezultat ZadatkaSRezultatom 5
rezultat ZadatkaSRezultatom 6
rezultat ZadatkaSRezultatom 7
rezultat ZadatkaSRezultatom 8
rezultat ZadatkaSRezultatom 9
*///:~
```

Metoda **submit**( ) proizvodi objekat tipa **Future**, parametrizovan za određeni tip rezultata koji vraća taj objekat tipa **Callable**. Objekat tipa **Future** možete ispitivati metodom **isDone**( ) kako biste videli da li je završio rad. Kada zadatak završi s radom i ima rezultat, pozovite metodu **get**( ) da biste ga saznali. Ukoliko **get**( ) pozovete bez prethodne provere završenosti metodom **isDone**( ), **get**( ) će se blokirati sve dok rezultat ne bude spreman. **get**( ) možete pozvati i uz vremensko odlaganje (engl. *timeout*), ili pozovite **isDone**( ) kako biste videli da li je zadatak završio s radom pre nego što pozovete **get**( ) da pribavi njegov rezultat.

Preklopljena metoda **Executors.callable**( ) prima objekat koji realizuje interfejs **Runnable** i proizvodi objekat tipa **Callable**. **ExecutorService** ima „pozivajuće" metode koje izvršavaju kolekcije **Callable** objekata.

Vežba 5: (2) Izmenite vežbu 2 tako da zadatak realizuje interfejs **Callable** i sabira sve Fibonaccijeve brojeve. Napravite nekoliko zadataka i prikažite rezultate.

Spavanje

Metodom **sleep**( ) prekidate (blokirate) izvršavanje zadatka na određeno vreme. Ako u klasi **Lansiranje** zamenite poziv metode **yield**( ) pozivom metode **sleep**( ), dobićete ovo:

```
//: paralelno/ZadatakKojiSpava.java
// Prekidam izvršavanje zadatka
// na određeno vreme metodom sleep().
import java.util.concurrent.*;

public class ZadatakKojiSpava extends Lansiranje {
  public void run() {
    try {
      while(odbrojavanje-- > 0) {
        System.out.print(status());
        // Stari način:
        // Thread.sleep(100);
        // Način Jave SE5/6:
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      System.err.println("Prekinut");
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new ZadatakKojiSpava());
    exec.shutdown();
  }
} /* Ispis:
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7),
#1(7), #2(7), #3(7), #4(7), #0(6), #1(6), #2(6), #3(6), #4(6), #0(5), #1(5),
#2(5), #3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3), #2(3),
#3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1),
#4(1), #0(Lansiranje!), #1(Lansiranje!), #2(Lansiranje!), #3(Lansiranje!),
#4(Lansiranje!),
*///:~
```

Poziv metode **sleep**( ) može da baci izuzetak **InterruptedException** koji se hvata u metodi **run**( ). Pošto se izuzeci ne prostiru preko granica niti do metode **main**( ), morate lokalno da obradite sve izuzetke koji nastanu unutar zadatka.

Java SE5 je donela i eksplicitniju verziju metode **sleep**( ) kao deo klase **TimeUnit**, što je prikazano u prethodnom primeru. Ona poboljšava čitljivost tako što omogućuje zadavanje vremenskih jedinica za metodu **sleep**( ). **TimeUnit** se može upotrebiti i za konverzije, što ćete videti u nastavku poglavlja.

U zavisnosti od platforme, možda ćete videti da se zadaci izvršavaju po „savršenom redosledu" – nulti do četvrtog, pa opet nulti. To ima smisla zato što nakon svake naredbe **print** svaki zadatak ide na spavanje (blokira se), a to mehanizmu za raspodelu procesorskog vremena nitima daje priliku da procesor prebaci na drugu nit koja izvršava neki drugi zadatak. To sekvencijalno ponašanje omogućuje pripadni mehanizam višenitnog rada koji se menja u zavisnosti od operativnog sistema, pa na to ne možete da računate. Ukoliko morate da kontrolišete redosled izvršavanja zadataka, najbolje ćete proći ako

upotrebite kontrole za sinhronizovanje (opisane u nastavku) ili, u nekim slučajevima, ukoliko uopšte ne budete upotrebljavali niti, nego sami napišete rutine koje jedna drugoj prepuštaju kontrolu po zadatom redosledu.

Vežba 6: (2) Napravite zadatak koji spava tokom nasumično odabranog vremena dugačkog između 1 i 10 sekundi, a zatim prikazuje to vreme spavanja i prekida rad. Napravite i pokrenite više takvih zadataka. (Neka se njihov broj zadaje na komandnoj liniji.)

Prioritet

Prioritet niti pokazuje koliko je ona važna mehanizmu za raspodelu procesorskog vremena. Iako je redosled kojim CPU izvršava skup niti neodređen, mehanizam za raspodelu procesorskog vremena daje prednost niti najvišeg prioriteta koja čeka na izvršavanje. To ne znači da se niti nižeg prioriteta uopšte ne izvršavaju. (Dakle, prioriteti ne mogu da prouzrokuju zastoj u izvršavanju). Mehanizam za raspodelu procesorskog vremena ređe pokreće niti nižeg prioriteta i to je sva razlika.

Gotovo uvek bi sve niti trebalo da se izvršavaju s podrazumevanim prioritetom. Ručno menjanje prioriteta najčešće nije opravdano.

Sledi primer s različitim nivoima prioriteta. Prioritet postojeće niti čitamo metodom **getPriority**( ), a zadajemo (kad god poželimo) metodom **setPriority**( ).

```
//: paralelno/ProstiPrioriteti.java
// Primer korišćenja prioriteta niti.
import java.util.concurrent.*;

public class ProstiPrioriteti implements Runnable {
  private int odbrojavanje = 5;
  private volatile double d; // Bez optimizacije
  private int prioritet;
  public ProstiPrioriteti(int prioritet) {
    this.prioritet = prioritet;
  }
  public String toString() {
    return Thread.currentThread() + ": " + odbrojavanje;
  }
  public void run() {
    Thread.currentThread().setPriority(prioritet);
    while(true) {
      // Skupa operacija koja se može prekinuti:
      for(int i = 1; i < 100000; i++) {
        d += (Math.PI + Math.E) / (double)i;
        if(i % 1000 == 0)
          Thread.yield();
      }
      System.out.println(this);
      if(--odbrojavanje == 0) return;
    }
  }
  public static void main(String[] args) {
```

```
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(
        new ProstiPrioriteti(Thread.MIN_PRIORITY));
    exec.execute(
        new ProstiPrioriteti(Thread.MAX_PRIORITY));
    exec.shutdown();
  }
} /* Ispis: (70% podudaranja)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

Metoda **toString( )** je preklopljena tako da poziva **Thread.toString( )** koja ispisuje ime niti, njen nivo prioriteta i grupu kojoj nit pripada. Ime niti možete i sami zadati preko konstruktora; ovde se ono automatski generiše u obliku **pool-1-thread-1**, **pool-1--thread-2** itd. Preklopljena metoda **toString( )** prikazuje i iznos odbrojavanja za zadatak. Vodite računa o tome da metodom **Thread.currentThread( )** iz tog zadatka možete dobiti referencu niti (objekta tipa **Thread**) koja izvršava zadatak.

Iz rezultata vidite da je nivo prioriteta poslednje niti najviši i da su sve ostale niti na najnižem nivou. Obratite pažnju na to da se prioritet postavlja na početku izvršavanja metode **run( )**; nema smisla postavljati ga u konstruktoru, pošto **Executor** (izvršilac) u tom trenutku još nije otpočeo zadatak.

U metodi **run( )** 100 000 puta se obavlja prilično skupo izračunavanje u formatu pokretnog zareza koje obuhvata sabiranje i deljenje tipa **double**. Promenljiva **d** je dobila modifikator **volatile** da prevodilac ne bi sprovodio optimizaciju. Ne bismo videli uticaj zadavanja nivoa prioriteta da nema tog izračunavanja. (Probajte: pretvorite u komentar petlju **for** koja sadrži **double** izračunavanja.) Uz izračunavanje, vidite da je mehanizam za raspodelu procesorskog vremena češće pozivao nit najvećeg prioriteta (**MAX_PRIORITY**). (Tako se ponašao barem na Windows XP računaru.) Iako je i ispisivanje na konzoli skupa operacija, ono ne omogućuje da vidimo uticaj različitih nivoa prioriteta, pošto se ono ne prekida (u protivnom bi se konzolni ispis pokvario pri prebacivanju s jedne niti na drugu), dok se matematička izračunavanja mogu prekinuti. Izračunavanje traje dovoljno dugo da mehanizam za raspodelu procesorskog vremena uskače, zamenjuje zadatke i pri tom pazi na prioritete, tako da se niti visokog prioriteta izvode češće. Metoda **yield( )** se redovno poziva da bi se izazvalo prebacivanje konteksta.

JDK ima 10 nivoa prioriteta, ali se oni ne preslikavaju dobro u svim operativnim sistemima. Primera radi, Windows ima 7 nivoa prioriteta koji nisu fiksni, pa je preslikavanje

(onih 10 na ovih 7) neodređeno. Sunov Solaris ima 2^{31} nivoa. Jedini pristup koji je prenosiv na sve platforme jeste držati se **MAX_PRIORITY, NORM_PRIORITY** i **MIN_PRIORITY** prilikom podešavanja nivoa prioriteta.

Prepuštanje

Ukoliko znate da ste uradili ono što je trebalo tokom jednog prolaska kroz petlju u metodi **run( )**, mehanizmu za raspodelu procesorskog vremena nitima možete saopštiti da je tekuća nit uradila dovoljno i predložiti mu da procesor dodeli nekom drugom zadatku. Taj predlog (to i *jeste* samo predlog, pošto nema jemstva da će ga realizacija poslušati) ima oblik metode **yield( )**. Kada pozovete **yield( )**, predlažete prelazak na izvršavanje drugih niti *istog prioriteta.*

Lansiranje.java metodom **yield( )** dobro raspodeljuje obradu na razne zadatke programa **Lansiranje**. U metodi **Lansiranje. run( )** pozive **Thread.yield( )** pretvorite u komentare, pa ćete videti razliku. Međutim, za ozbiljnu kontrolu ili podešavanje aplikacije po pravilu se ne možete osloniti na **yield( )**. Šta više, često se **yield( )** pogrešno upotrebljava.

Servisne niti

Servisne niti (engl. *daemon threads*) obezbeđuju neke usluge namenjene programu u celini i izvršavaju se u pozadini tokom rada programa, ali ne spadaju u njegove suštinske delove. Zato se program završava čim završe s radom sve obične niti. Važi i obratno: ako još ima običnih niti koje se izvršavaju, program nastavlja rad – na primer, ako postoji nit koja izvršava **main( )**.

```
//: paralelno/JednostavneServisneNiti.java
// Servisne niti ne sprečavaju da se program okonča.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class JednostavneServisneNiti implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("sleep() prekinuta");
    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < 10; i++) {
      Thread servisnanit = new Thread(new JednostavneServisneNiti());
      servisnanit.setDaemon(true); // Mora biti pozvana pre metode start()
      servisnanit.start();
    }
    print("Sve servisne niti pokrenute");
```

```
      TimeUnit.MILLISECONDS.sleep(175);
    }
  } /* Ispis: (primer)
  Sve servisne niti pokrenute
  Thread[Thread-0,5,main] JednostavneServisneNiti@530daa
  Thread[Thread-1,5,main] JednostavneServisneNiti@a62fc3
  Thread[Thread-2,5,main] JednostavneServisneNiti@89ae9e
  Thread[Thread-3,5,main] JednostavneServisneNiti@1270b73
  Thread[Thread-4,5,main] JednostavneServisneNiti@60aeb0
  Thread[Thread-5,5,main] JednostavneServisneNiti@16caf43
  Thread[Thread-6,5,main] JednostavneServisneNiti@66848c
  Thread[Thread-7,5,main] JednostavneServisneNiti@8813f2
  Thread[Thread-8,5,main] JednostavneServisneNiti@1d58aae
  Thread[Thread-9,5,main] JednostavneServisneNiti@83cc67
  ...
  *///:~
```

Pre pokretanja metodom **start**( ), nit morate metodom **setDaemon**( ) najpre pretvoriti u servisnu (naterati je da se izvršava u pozadini).

Program nema šta da radi kada **main**( ) završi svoj posao, jer se tada ne izvršava ništa drugo sem servisnih niti. Stavio sam **main**( ) nakratko na spavanje da biste videli posledice pokretanja svih servisnih niti. Bez toga biste videli samo neke rezultate pravljenja niti koje se izvršavaju u pozadini. (Isprobajte pozive metode **sleep**( ) raznih dužina da biste videli to ponašanje.)

Program **JednostavneServisneNiti.java** pravi eksplicitne **Thread** objekte kako bi mogao da ih pretvori u servisne postavljanjem odgovarajućeg indikatora. Atribute niti (da li je servisna, njen prioritet, ime) napravljenih pomoću izvršioca možete prilagoditi pisanjem namenske klase **ThreadFactory**:

```
//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    Thread t = new Thread(r);
    t.setDaemon(true);
    return t;
  }
} ///:~
```

Jedina razlika od obične **ThreadFactory** (proizvodne niti) jeste to što ova zadaje vrednost **true** za status servisne niti. Nov objekat tipa **DaemonThreadFactory** sada možemo proslediti kao argument metodi **Executors.newCachedThreadPool**( ):

```
//: paralelno/DaemonThreadFactory.java
// Pravljenje servisnih niti pomoću ThreadFactory.
import java.util.concurrent.*;
import net.mindview.util.*;
```

```
import static net.mindview.util.Print.*;

public class DaemonThreadFactory implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("Prekinuto");
    }
  }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool(
      new DaemonThreadFactory());
    for(int i = 0; i < 10; i++)
      exec.execute(new DaemonThreadFactory());
    print("Sve servisne niti pokrenute");
    TimeUnit.MILLISECONDS.sleep(500); // Neka radi neko vreme
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Svaka od statičkih **ExecutorService** metoda za pravljenje preklopljena je tako da prima objekat tipa **ThreadFactory** pomoću kojega će praviti nove niti.

Možemo to podići za jedan stepen više i napraviti uslužnog izvršioca **DaemonThreadPoolExecutor** za pravljenje servisnih niti:

```
//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor
extends ThreadPoolExecutor {
  public DaemonThreadPoolExecutor() {
    super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
      new SynchronousQueue<Runnable>(),
      new DaemonThreadFactory());
  }
} ///:~
```

Argumente za poziv konstruktora osnovne klase pronašao sam u izvornom kodu klase **Executors.java**.

Pozivanjem metode **isDaemon( )** saznaćete da li je data nit servisna (da li se izvršava u pozadini). Ako se neka nit izvršava u pozadini, to automatski važi i za sve niti koje ona napravi, kao što se vidi iz sledećeg primera:

```
//: paralelno/ServisneNiti.java
// Servisna nit rada druge servisne niti.
```

```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ServisnaNit implements Runnable {
  private Thread[] t = new Thread[10];
  public void run() {
    for(int i = 0; i < t.length; i++) {
      t[i] = new Thread(new MajkaServisnihNiti());
      t[i].start();
      printnb("MajkaServisnihNiti " + i + " pokrenuta, ");
    }
    for(int i = 0; i < t.length; i++)
      printnb("t[" + i + "].isDaemon() = " +
        t[i].isDaemon() + ", ");
    while(true)
      Thread.yield();
  }
}

class MajkaServisnihNiti implements Runnable {
  public void run() {
    while(true)
      Thread.yield();
  }
}

public class ServisneNiti {
  public static void main(String[] args) throws Exception {
    Thread d = new Thread(new ServisnaNit());
    d.setDaemon(true);
    d.start();
    printnb("d.isDaemon() = " + d.isDaemon() + ", ");
    // Pusti servisne niti da dovrše svoje
    // procese pokretanja:
    TimeUnit.SECONDS.sleep(1);
  }
} /* Ispis: (primer)
d.isDaemon() = true, MajkaServisnihNiti 0 pokrenuta, MajkaServisnihNiti 1
pokrenuta, MajkaServisnihNiti 2 pokrenuta, MajkaServisnihNiti 3
pokrenuta, MajkaServisnihNiti 4 pokrenuta, MajkaServisnihNiti 5
pokrenuta, MajkaServisnihNiti 6 pokrenuta, MajkaServisnihNiti 7
pokrenuta, MajkaServisnihNiti 8 pokrenuta, MajkaServisnihNiti 9
pokrenuta, t[0].isDaemon() = true,
t[1].isDaemon() = true, t[2].isDaemon() = true,
t[3].isDaemon() = true, t[4].isDaemon() = true,
t[5].isDaemon() = true, t[6].isDaemon() = true,
t[7].isDaemon() = true, t[8].isDaemon() = true,
t[9].isDaemon() = true,
*///:~
```

ServisnaNit uključuje svoj indikator servisne niti i zatim pravi gomilu drugih niti kako bi pokazala da su i one servisne, iako *nisu* eksplicitno pretvorene u servisne. Zatim **ServisnaNit** ide u beskonačnu petlju koja poziva metodu **yield( )** da bi predala kontrolu drugim procesima.

Imajte u vidu činjenicu da servisne niti ne izvršavaju odredbe **finally** prilikom okončanja svojih metoda **run( )**:

```
//: paralelno/ServisneNitiNeIzvrsavajuOdredbeFinally.java
// Servisne niti ne izvršavaju odredbe finally
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ServisnaNitA implements Runnable {
  public void run() {
    try {
      print("Kreće ServisnaNitA");
      TimeUnit.SECONDS.sleep(1);
    } catch(InterruptedException e) {
      print("Izlazak pomoću InterruptedException");
    } finally {
      print("Ovo treba stalno da se izvršava?");
    }
  }
}

public class ServisneNitiNeIzvrsavajuOdredbeFinally {
  public static void main(String[] args) throws Exception {
    Thread t = new Thread(new ServisnaNitA());
    t.setDaemon(true);
    t.start();
  }
} /* Ispis:
Kreće ServisnaNitA
*///:~
```

Kada pokrenete ovaj program, videćete da se odredba **finally** ne izvršava, ali ako poziv metode **setDaemon( )** pretvorite u komentar, videćete da se odredba **finally** *izvršava.*

Takvo ponašanje je ispravno, iako ga možda ne očekujete na osnovu prethodnih obećanja u vezi sa odredbom **finally**. Servisne niti bivaju naglo prekinute kada prestane izvršavanje poslednje neservisne niti. Dakle, čim **main( )** izađe, JVM odmah prekida rad svih servisnih niti, bez ikakvih formalnosti koje ste možda očekivali. Pošto se servisne niti ne mogu lepo prekinuti, one su retko kada podesne. Po pravilu su neservisni izvršioci bolji, pošto se svi zadaci koje izvršilac kontroliše mogu prekinuti odjednom. Kao što ćete videti u nastavku poglavlja, u tom slučaju se gašenje odvija pravilno.

Vežba 7: (2) Eksperimentišite s različitim vremenima spavanja u programu **ServisneNiti.java** da biste videli šta se dešava.

Vežba 8: (1) Izmenite program **JosOsnovaNiti.java** tako da sve niti budu servisne i proverite da li program prekida rad čim metoda **main**( ) dobije priliku da izađe.

Vežba 9: (3) Izmenite program **ProstiPrioriteti.java** tako da namenski proizvođač niti (klasa **ThreadFactory**) postavlja prioritete niti.

Varijante programiranja

U dosadašnjim primerima su sve klase zadataka realizovale interfejs **Runnable**. U veoma jednostavnim slučajevima moguć je i drugačiji pristup, nasleđivanje neposredno od klase **Thread**, što se radi ovako:

```
//: paralelno/JednostavnaNit.java
// Neposredno nasleđivanje od klase Thread.

public class JednostavnaNit extends Thread {
  private int odbrojavanje = 5;
  private static int brojacNiti = 0;
  public JednostavnaNit() {
    // Sačuvaj ime niti:
    super(Integer.toString(++brojacNiti));
    start();
  }
  public String toString() {
    return "#" + getName() + "(" + odbrojavanje + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--odbrojavanje == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new JednostavnaNit();
  }
} /* Ispis:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3), #2(2), #2(1), #3(5),
#3(4), #3(3), #3(2), #3(1), #4(5), #4(4), #4(3), #4(2), #4(1), #5(5), #5(4),
#5(3), #5(2), #5(1),
*///:~
```

Nitima (objektima klase **Thread**) dajemo imena tako što pozivamo odgovarajuće **Thread** konstruktore. To ime ispisuje **toString**( ) nakon što ga dobije od metode **getName**( ).

Možda ćete sresti i idiom objekta tipa **Runnable** koji sam sobom upravlja:

```
//: paralelno/Samoupravno.java
// Objekat tipa Runnable koji sadrži sopstvenu nit koja ga izvršava.
```

```
public class Samoupravno implements Runnable {
  private int odbrojavanje = 5;
  private Thread t = new Thread(this);
  public Samoupravno() { t.start(); }
  public String toString() {
    return Thread.currentThread().getName() +
      "(" + odbrojavanje + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--odbrojavanje == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new Samoupravno();
  }
} /* Ispis:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-0(1), Thread-1(5),
Thread-1(4), Thread-1(3), Thread-1(2), Thread-1(1), Thread-2(5), Thread-2(4),
Thread-2(3), Thread-2(2), Thread-2(1), Thread-3(5), Thread-3(4), Thread-3(3),
Thread-3(2), Thread-3(1), Thread-4(5), Thread-4(4), Thread-4(3), Thread-4(2),
Thread-4(1),
*///:-
```

Ovo se ne razlikuje mnogo od nasleđivanja od klase **Thread**, sem što je sintaksa nešto neobičnija. Međutim, realizacija interfejsa omogućuje nasleđivanje od druge klase, dok nasleđivanje od klase **Thread** to ne omogućuje.

Obratite pažnju na to da je metoda **start**( ) pozvana iz konstruktora. Primer je krajnje jednostavan i zato verovatno bezbedan; ipak, morate biti svesni da pokretanje niti iz konstruktora ume da bude veoma problematično, pošto bi moglo početi izvršavanje nekog drugog zadatka pre nego što konstruktor završi svoj posao, što znači da taj zadatak može da pristupi objektu koji je u nestabilnom stanju. I zbog toga je korišćenje izvršilaca bolje od eksplicitnog pravljenja niti (objekata tipa **Thread**).

Ponekad kôd za pravljenje niti treba sakriti u klasi pomoću unutrašnje klase, kao što ću sada uraditi:

```
//: paralelno/VarijanteNiti.java
// Pravljenje niti pomoću unutrašnjih klasa.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Upotrebiću imenovanu unutrašnju klasu:
class UnutrasnjaNit1 {
  private int odbrojavanje = 5;
  private Unutrasnja unutrasnja;
  private class Unutrasnja extends Thread {
```

```
    Unutrasnja(String ime) {
      super(ime);
      start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--odbrojavanje == 0) return;
          sleep(10);
        }
      } catch(InterruptedException e) {
        print("prekinuto");
      }
    }
    public String toString() {
      return getName() + ": " + odbrojavanje;
    }
  }
  public UnutrasnjaNit1(String ime) {
    unutrasnja = new Unutrasnja(ime);
  }
}

// Upotrebiću anonimnu unutrašnju klasu:
class UnutrasnjaNit2 {
  private int odbrojavanje = 5;
  private Thread t;
  public UnutrasnjaNit2(String ime) {
    t = new Thread(ime) {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--odbrojavanje == 0) return;
            sleep(10);
          }
        } catch(InterruptedException e) {
          print("sleep() prekinuta");
        }
      }
      public String toString() {
        return getName() + ": " + odbrojavanje;
      }
    };
    t.start();
  }
}
```

```
// Upotrebiću imenovanu realizaciju interfejsa Runnable:
class UnutrasnjaRunnable1 {
  private int odbrojavanje = 5;
  private Unutrasnja unutrasnja;
  private class Unutrasnja implements Runnable {
    Thread t;
    Unutrasnja(String ime) {
      t = new Thread(this, ime);
      t.start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--odbrojavanje == 0) return;
          TimeUnit.MILLISECONDS.sleep(10);
        }
      } catch(InterruptedException e) {
        print("sleep() prekinuta");
      }
    }
    public String toString() {
      return t.getName() + ": " + odbrojavanje;
    }
  }
  public UnutrasnjaRunnable1(String ime) {
    unutrasnja = new Unutrasnja(ime);
  }
}

// Upotrebiću anonimnu realizaciju interfejsa Runnable:
class UnutrasnjaRunnable2 {
  private int odbrojavanje = 5;
  private Thread t;
  public UnutrasnjaRunnable2(String ime) {
    t = new Thread(new Runnable() {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--odbrojavanje == 0) return;
            TimeUnit.MILLISECONDS.sleep(10);
          }
        } catch(InterruptedException e) {
          print("sleep() prekinuta");
        }
      }
      public String toString() {
        return Thread.currentThread().getName() +
          ": " + odbrojavanje;
      }
```

```
    }, ime);
    t.start();
  }
}

// Zasebna metoda koja će neki kôd izvršavati kao zadatak:
class MetodaNit {
  private int odbrojavanje = 5;
  private Thread t;
  private String ime;
  public MetodaNit(String ime) { this.ime = ime; }
  public void runTask() {
    if(t == null) {
      t = new Thread(ime) {
        public void run() {
          try {
            while(true) {
              print(this);
              if(--odbrojavanje == 0) return;
              sleep(10);
            }
          } catch(InterruptedException e) {
            print("sleep() prekinuta");
          }
        }
        public String toString() {
          return getName() + ": " + odbrojavanje;
        }
      };
      t.start();
    }
  }
}

public class VarijanteNiti {
  public static void main(String[] args) {
    new UnutrasnjaNit1("UnutrasnjaNit1");
    new UnutrasnjaNit2("UnutrasnjaNit2");
    new UnutrasnjaRunnable1("UnutrasnjaRunnable1");
    new UnutrasnjaRunnable2("UnutrasnjaRunnable2");
    new MetodaNit("MetodaNit").runTask();
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

UnutrasnjaNit1 pravi imenovanu unutrašnju klasu koja nasleđuje **Thread** i u konstruktoru pravi instancu te unutrašnje klase. To ima smisla ako unutrašnja klasa ima posebne sposobnosti (nove metode) potrebne u drugim metodama. Međutim, nit obično pravimo samo zbog sposobnosti klase **Thread**, pa nije neophodno da pravimo imenovanu unutrašnju klasu. **UnutrasnjaNit2** pokazuje drugačiji pristup: u konstruktoru se pravi anonimna unutrašnja potklasa od **Thread** i svodi naviše na **Thread** referencu **t**. Ukoliko

ostalim metodama te klase zatreba pristup objektu na koji upućuje **t**, mogu to učiniti preko interfejsa **Thread** i bez poznavanja stvarnog tipa tog objekta.

Treća i četvrta klasa u primeru ponavljaju prve dve klase, ali koriste interfejse **Runnable**, a ne klasu **Thread**.

Klasa **MetodaNit** pokazuje pravljenje niti unutar metode. Metodu pozivate kada ste spremni da pokrenete tu nit, i metoda vraća svoj rezultat nakon što nit otpočne s radom. Ukoliko nit obavlja samo pomoćne, a ne i osnovne operacije klase, to je verovatno korisniji i prikladniji pristup nego što je pokretanje niti unutar konstruktora klase.

Vežba 10: (4) Izmenite vežbu 5 po uzoru na primer s klasom **MetodaNit**, tako da metoda **runTask( )** prima kao argument broj Fibonaccijevih brojeva koje treba sabrati, i svaki put kada pozovete **runTask( )**, ona vraća objekat tipa **Future** koji proizvodi poziv metode **submit( )**.

Terminologija

Kao što se vidi iz prethodnog odeljka, paralelni programi se u Javi mogu pisati na više načina i to programera može da zbuni. Često problem nastaje zbog terminologije kojom je opisan program za paralelno izvršavanje, naročito tamo gde su u pitanju niti.

Dosad je trebalo da uvidite kako postoji razlika između zadatka i niti koja taj zadatak izvršava; ta razlika je naročito jasna u Javinim bibliotekama, pošto nad klasom **Thread** zapravo nemamo nikakvu kontrolu (a još je jasnija kada se radi o izvršiocima koji umesto nas prave niti i upravljaju njima). Programer piše zadatak i na neki način mu dodeljuje nit koja će ga izvršavati.

U Javi, nit (klasa **Thread**) sama po sebi ne radi ništa. Ona izvršava zadatak koji joj damo. Pa ipak, u literaturi o višenitnom programiranju uvek se kaže „nit radi ovo ili ono". Dobija se utisak da nit *jeste* zadatak. Kada sam upoznao Javine niti, utisak je bio toliko jak da sam video jasnu relaciju JE, koja mi je govorila kako zadatak očigledno treba da izvedem iz klase **Thread**. Dodajte tome loše odabrano ime interfejsa **Runnable** (koji se može izvršavati); po mom mišljenju, trebalo je da se zove **Task** (zadatak). Ukoliko interfejs očigledno nije ništa više od generičkog kapsuliranja svojih metoda, onda je imenovanje po kalupu „može da uradi to i to" odgovarajuće, ali ako treba da izrazi viši pojam – kao što je **Zadatak** – onda mu treba dati ime tog pojma.

Problem potiče od mešanja nivoa apstrakcije. Na nivou koncepta, želimo da napravimo zadatak koji se izvršava nezavisno od drugih zadataka, pa bi trebalo da je moguće definisati zadatak i reći „radi", bez petljanja s detaljima. Ali fizički, pravljenje niti je skupo, pa moramo da ih štedimo i upravljamo njima. Zato *sa stanovišta realizacije* ima smisla odvojiti zadatke od niti. Sem toga, Javine niti su napravljene na osnovu nekakvih p-niti koje potiču još iz C-a, a u tom jeziku programer je opkoljen detaljima i mora poznavati do u sitnicu sve što se događa. Delimično je takav pristup prešao i u realizaciju Jave, pa da bismo ostali na visokom nivou apstrakcije, moramo biti disciplinovani pri pisanju programa. (Potrudiću se da u ovom poglavlju sam pokažem tu disciplinu.)

Da bi razmatranje bilo jasnije, terminom zadatak opisivaću ono što se radi, a terminom nit – mehanizam koji izvršava zadatak. Dakle, ako sistem razmatramo na konceptualnom nivou, možemo upotrebiti termin zadatak, a da uopšte ne spominjemo mehanizam izvršavanja.

Kako se pridružiti postojećoj niti

Jedna nit može pozvati metodu **join( )** za neku drugu nit; u tom slučaju, prva nit čeka da druga završi rad pa tek onda nastavlja izvršavanje. Ukoliko nit pozove **n.join( )** za drugu nit **t** kao argument, pozivajuća nit će biti zaustavljena sve dok odredišna nit **t** ne završi s radom – dok **t.isAlive( )** ne poprimi vrednost **false**.

Metodu **join( )** možete pozvati i s vremenskim argumentom (izraženim u milisekundama ili milisekundama i nanosekundama); u tom slučaju se izvršavanje metode **join( )** završava ukoliko se odredišna nit u tom vremenu ne završi.

Pošto poziv metode **join( )** može biti prekinut pozivom metode **interrupt( )** za pozivajuću nit, obavezna je upotreba odredbe **try-catch**.

Sve te operacije se izvršavaju u sledećem primeru:

```
//: paralelno/Joining.java
// Objašnjenje metode join().
import static net.mindview.util.Print.*;

class Spavac extends Thread {
  private int trajanje;
  public Spavac(String ime, int vremeSpavanja) {
    super(ime);
    trajanje = vremeSpavanja;
    start();
  }
  public void run() {
    try {
      sleep(trajanje);
    } catch(InterruptedException e) {
      print(getName() + " je bila prekinuta. " +
        "isInterrupted(): " + isInterrupted());
      return;
    }
    print(getName() + " je probuđena");
  }
}

class Joiner extends Thread {
  private Spavac spavac;
  public Joiner(String ime, Spavac spavac) {
    super(ime);
    this.spavac = spavac;
    start();
  }
  public void run() {
   try {
      spavac.join();
    } catch(InterruptedException e) {
      print("Prekinuto");
    }
```

```
      print(getName() + ": čekanje završeno");
    }
  }

  public class Joining {
    public static void main(String[] args) {
      Spavac
        pospana = new Spavac("Pospana", 1500),
        gundjalo = new Spavac("Gundjalo", 1500);
      Joiner
        omamljena = new Joiner("Omamljena", pospana),
        doktor = new Joiner("Doktor", gundjalo);
      gundjalo.interrupt();
    }
  } /* Ispis:
  Gundjalo je bila prekinuta. isInterrupted(): false
  Doktor: čekanje završeno
  Pospana je probuđena
  Omamljena: čekanje završeno
  *///:~
```

Spavac je nit koja odlazi na spavanje za vreme specificirano u njenom konstruktoru. Unutar metode **run**( ), metoda **sleep**( ) može završiti s radom kada to vreme istekne, ali može takođe biti i prekinuta u izvršavanju. Prekid se prijavljuje unutar odredbe **catch**, kao i rezultat metode **isInterrupted**( ). Kada neka druga nit pozove **interrupt**( ) za ovu nit, postavlja se indikator da je nit prekinuta u izvršavanju. Međutim, taj indikator biva obrisan prilikom hvatanja izuzetka, pa će unutar odredbe **catch** ispitivanje vrednosti te metode uvek dati rezultat **false**. Taj indikator se koristi u drugim situacijama u kojima nit može da ispita svoje prekinuto stanje, a u kojima se ne generiše izuzetak.

Joiner je zadatak koji čeka da se **Spavac** probudi; čekanje se postiže pozivom metode **join**( ) za objekat tipa **Spavac**. U metodi **main**( ) svaki **Spavac** ima po jedan objekat tipa **Joiner**, i iz rezultata vidite da **Joiner** završava svoje izvršavanje zajedno s tim objektom tipa **Spavac**, bez obzira na to da li je **Spavac** bio prekinut ili je normalno okončao svoje izvršavanje.

Imajte u vidu da Java SE5 biblioteke **java.util.concurrent** sadrže alatke. Među njima je **CyclicBarrier** (opisaću je u nastavku poglavlja); ona može biti podesnija od metode **join**( ) koja je bila deo prvobitne biblioteke za višenitno programiranje.

Korisničko okruženje koje brzo reaguje

Već sam rekao da je jedan od motiva za višenitno programiranje potreba da se dobiju korisnička okruženja koja brzo reaguju. Iako sve do poglavlja *Grafička korisnička okruženja* nećemo imati posla s *grafičkim* okruženjima, naredni primer je jednostavna skica konzolnog korisničkog okruženja. Primer ima dve verzije: jednu koja stalno nešto računa i zato ne stiže da pročita ulaz s konzole, i drugu u kojoj je izračunavanje stavljeno u zaseban zadatak, pa pored njega program stiže da osluškuje *i* ulaz s konzole.

```
//: paralelno/BrzoKorisnickoOkruzenje.java
// Brzina reagovanja korisničkog okruženja.
// {RunByHand}

class SporoKorisnickoOkruzenje {
  private volatile double d = 1;
  public SporoKorisnickoOkruzenje() throws Exception {
    while(d > 0)
      d = d + (Math.PI + Math.E) / d;
    System.in.read(); // Ovo se nikada ne izvršava
  }
}

public class BrzoKorisnickoOkruzenje extends Thread {
  private static volatile double d = 1;
  public BrzoKorisnickoOkruzenje() {
    setDaemon(true);
    start();
  }
  public void run() {
    while(true) {
      d = d + (Math.PI + Math.E) / d;
    }
  }
  public static void main(String[] args) throws Exception {
    //! new SporoKorisnickoOkruzenje(); // Ovaj proces moramo ubiti
    new BrzoKorisnickoOkruzenje();
    System.in.read();
    System.out.println(d); // Pokazuje stepen napredovanja
  }
} ///:~
```

SporoKorisnickoOkruzenje obavlja izračunavanje unutar beskonačne petlje **while**, te očigledno ne može da dopre do reda u kojem se čita ulaz s konzole (uslov **while** čini da se prevodilac prevari i oceni da je taj red dostižan). Ukoliko red u kojem se pravi novo **SporoKorisnickoOkruzenje** pretvorite u naredbu koja se izvršava, moraćete taj proces ručno da ubijete da biste izašli iz programa.

Program će brzo reagovati ukoliko izračunavanje stavite unutar metode **run( )** i tako omogućite da ono bude predupređeno. Kada pritisnete taster Enter, videćete da se izračunavanje odvija u pozadini dok program osluškuje ulaz korisnika.

Grupe niti

Grupa niti sadrži kolekciju niti. Vrednost grupa niti sažeo je Joshua Bloch,[8] projektant softvera koji je (između ostalog), dok je radio za Sun, popravio i znatno poboljšao biblioteku Java kolekcija u JDK 1.2:

„Grupe niti je najbolje smatrati neuspešnim eksperimentom i zaboraviti da postoje."

[8] *Efikasno programiranje na Javi*, autor Joshua Bloch (Mikro knjiga, 2004), s. 211.

Ukoliko ste (kao ja) potrošili vreme i energiju pokušavajući da shvatite vrednost grupa niti, možda se pitate zašto Sun o toj temi nikada nije dao zvanično saopštenje. Isto važi i za brojne druge promene sprovedene u Javi tokom godina. Izgleda da bismo mogli primeniti *Teoriju rastućeg angažovanja*[9] – njen tvorac je Joseph Stiglitz (dobitnik Nobelove nagrade).

„Cenu nastavljanja grešaka plaćaju drugi, dok cenu priznavanja grešaka plaćamo sami."

Hvatanje izuzetaka

Izuzetak koji je pobegao iz niti ne možete uhvatiti – takva je njihova priroda. Kada izuzetak izađe iz metode **run( )** zadatka, prodreće do konzole ukoliko ne preduzmete posebne mere za hvatanje takvih zabludelih grešaka. Pre Jave SE5, za hvatanje takvih izuzetaka koristili smo grupe niti, ali sada problem možemo rešiti izvršiocima, te o grupama niti više ne moramo *ništa* da znamo (sem onoliko koliko je potrebno da bismo razumeli stari kôd; pojedinosti o grupama niti pročitajte u knjizi *Thinking in Java, 2nd Edition*, koju možete preuzeti sa adrese *www.MindView.net*).

Sledeći zadatak uvek baca izuzetak koji izlazi iz njegove metode **run( )**, a **main( )** pokazuje šta se dešava kada ga pokrenete:

```
//: paralelno/NitIzuzetka.java
// {ThrowsException}
import java.util.concurrent.*;

public class NitIzuzetka implements Runnable {
  public void run() {
    throw new RuntimeException();
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new NitIzuzetka());
  }
} ///:~
```

Rezultat programa je (nakon uklanjanja nekih odrednica da bi stalo):

```
java.lang.RuntimeException
        at NitIzuzetka.run(NitIzuzetka.java:7)
        at ThreadPoolExecutor$Worker.runTask(Unknown Source)
        at ThreadPoolExecutor$Worker.run(Unknown Source)
        at java.lang.Thread.run(Unknown Source)
```

Ništa ne pomaže ako telo metode **main( )** stavimo u blok **try-catch**:

```
//: paralelno/NaivnaObradaIzuzetka.java
// {ThrowsException}
import java.util.concurrent.*;
```

[9] I na nekoliko drugih mesta u celoj Javi. Zašto da se ograničimo samo na Javu? Bio sam konsultant u više projekata gde se pokazalo da je ta tvrdnja tačna.

```
public class NaivnaObradaIzuzetka {
  public static void main(String[] args) {
    try {
      ExecutorService exec =
        Executors.newCachedThreadPool();
      exec.execute(new NitIzuzetka());
    } catch(RuntimeException ue) {
      // Ova naredba se NEĆE izvršiti!
      System.out.println("Izuzetak je obrađen!");
    }
  }
} ///:~
```

Rezultat je isti kao u prethodnom primeru: izuzetak koji nije bio uhvaćen.

Problem ćemo rešiti promenom načina na koji izvršilac (**Executor**) pravi niti. Java SE5 ima nov interfejs po imenu **Thread.UncaughtExceptionHandler**; pomoću njega svakoj niti možete da pridružite njen program za obradu. Neposredno pre nego što bi nit umrla zato što njen izuzetak nije bio uhvaćen, automatski se poziva **Thread.UncaughtExceptionHandler.uncaughtException( )**. Da bismo mogli da ga upotrebimo, napravićemo nov tip interfejsa **ThreadFactory** koji svakoj niti koju napravi pridružuje **Thread.UncaughtExceptionHandler**. Tu proizvodnu metodu prosledićemo metodi klase **Executors** koja pravi nov **ExecutorService**:

```
//: paralelno/HvatanjeNeuhvacenihIzuzetaka.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
  public void run() {
    Thread t = Thread.currentThread();
    System.out.println("izvršava nit " + t);
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    throw new RuntimeException();
  }
}

class MojBlokZaObraduNeuhvacenihIzuzetaka implements
Thread.UncaughtExceptionHandler {
  public void neuhvacenIzuzetak(Thread t, Throwable e) {
    System.out.println("uhvatio " + e);
  }
}

class HandlerThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    System.out.println(this + " pravi novu nit");
    Thread t = new Thread(r);
    System.out.println("napravio " + t);
```

```
    t.setUncaughtExceptionHandler(
      new MojBlokZaObraduNeuhvacenihIzuzetaka());
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    return t;
  }
}

public class HvatanjeNeuhvacenihIzuzetaka {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool(
      new HandlerThreadFactory());
    exec.execute(new ExceptionThread2());
  }
} /* Ispis: (90% podudaranja)
HandlerThreadFactory@de6ced pravi novu nit
napravio Thread[Thread-0,5,main]
eh = MojBlokZaObraduNeuhvacenihIzuzetaka@1fb8ee3
izvršava nit Thread[Thread-0,5,main]
eh = MojBlokZaObraduNeuhvacenihIzuzetaka@1fb8ee3
uhvatio java.lang.RuntimeException
*///:~
```

Ispisao sam i dodatne oznake kako biste mogli proveriti da se nitima koje proizvodi Factory daje novi **UncaughtExceptionHandler**. Možete videti da neuhvaćene izuzetke sada hvata metoda **neuhvacenIzuzetak**( ).

Gornji primer omogućuje da blok za obradu (engl. *handler*) prilagođavate svakom pojedinom slučaju. Ukoliko nameravate da isti blok za obradu izuzetaka koristite svuda, još je jednostavnije da napravite *podrazumevani* blok za obradu neuhvaćenih izuzetaka koji postavlja određeno statičko polje u klasi **Thread**:

```
//: paralelno/PodraBlokZaObraduNeuhvacenihIzuzetaka.java
import java.util.concurrent.*;

public class PodraBlokZaObraduNeuhvacenihIzuzetaka {
  public static void main(String[] args) {
    Thread.setDefaultUncaughtExceptionHandler(
      new MojBlokZaObraduNeuhvacenihIzuzetaka());
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new NitIzuzetka());
  }
} /* Ispis:
uhvatio java.lang.RuntimeException
*///:~
```

Ovaj program se poziva samo u slučaju da ne postoji blok za obradu neuhvaćenih izuzetaka prilagoden svakoj niti. Sistem proverava postoji li verzija za konkretnu nit i ako je ne pronađe, proverava da li ta grupa niti ima sopstvenu specijalizovanu metodu **neuhvacenIzuzetak**( ); ako nema, poziva se **defaultUncaughtExceptionHandler**.

Deljenje resursa

Program s jednom niti možete da zamislite kao usamljenu jedinku koja se kreće kroz prostor vašeg problema i rešava jedan po jedan zadatak. Pošto postoji samo jedna jedinka, uopšte ne morate da vodite računa o situaciji kada dve jedinke pokušavaju da iskoriste isti resurs istovremeno, poput dvoje koji pokušavaju da se parkiraju na istom mestu, ili da prođu kroz vrata istovremeno ili čak da samo govore istovremeno.

Ako za rešavanje problema upotrebite više niti, ne javlja se problem usamljenih jedinki, ali postoji mogućnost da dve ili više niti pokušaju da upotrebe isti ograničen resurs u istom trenutku, tj. da se uzajamno ometu. Sukobljavanje zbog nekog resursa mora se sprečiti ili će dve niti istovremeno pokušati da pristupe istom bankovnom nalogu, odštampaju nešto na istom štampaču, okrenu isti ventil itd.

Nepravilno pristupanje resursima

Razmotrite sledeći primer u kojem jedan zadatak generiše parne brojeve, a drugi zadaci ih troše. Jedini posao zadataka potrošača jeste da provere validnost parnih brojeva.

Prvo ćemo definisati potrošački zadatak **ProveraParnosti**, pošto ćemo ga koristiti i u narednim primerima. Da bismo zadatak **ProveraParnosti** odvojili od raznih tipova generatora s kojima ćemo eksperimentisati, napravićemo apstraktnu klasu **GeneratorCelobroj** koja sadrži minimum neophodnih metoda za koje **ProveraParnosti** mora da zna: metodu **next( )** i metodu za otkazivanje pravljenja objekta. Ta klasa ne realizuje interfejs **Generator**, zato što mora da proizvede ceo broj (**int**), a generičke klase ne primaju proste tipove kao parametre.

```
//: paralelno/GeneratorCelobroj.java

public abstract class GeneratorCelobroj {
  private volatile boolean cancelled = false;
  public abstract int next();
  // Dopustiću otkazivanje ovoga:
  public void cancel() { cancelled = true; }
  public boolean isCancelled() { return cancelled; }
} ///:~
```

GeneratorCelobroj ima metodu **cancel( )** za promenu stanja **boolean** indikatora **cancelled** i metodu **isCancelled( )** koja utvrđuje da li je pravljenje objekta otkazano. Pošto je indikator **cancelled** tipa **boolean**, on je *atomski* (nedeljiv), što znači da takvo polje ne možete pročitati u nekom međustanju usred jednostavnih operacija kao što su dodeljivanje i vraćanje vrednosti, jer se one ne mogu prekidati. Indikatoru **cancelled** dodat je i modifikator **volatile**, da bi se obezbedila *vidljivost*. O atomskim indikatorima i vidljivosti saznaćete više u nastavku poglavlja.

Sledeća klasa **ProveraParnosti** može da testira svaki **GeneratorCelobroj**:

```
//: paralelno/ProveraParnosti.java
import java.util.concurrent.*;
```

```
public class ProveraParnosti implements Runnable {
  private GeneratorCelobroj generator;
  private final int id;
  public ProveraParnosti(GeneratorCelobroj g, int ident) {
    generator = g;
    id = ident;
  }
  public void run() {
    while(!generator.isCancelled()) {
      int vre = generator.next();
      if(vre % 2 != 0) {
        System.out.println(vre + " nije paran!");
        generator.cancel(); // Otkazuje sve ProvereParnosti
      }
    }
  }
  // Testiranje svih tipova GeneratorCelobroj:
  public static void test(GeneratorCelobroj gp, int broj) {
    System.out.println("Pritisnite Control-C ako želite da izađete
       iz programa");
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < broj; i++)
      exec.execute(new ProveraParnosti(gp, i));
    exec.shutdown();
  }
  // Podrazumevana vrednost broja:
  public static void test(GeneratorCelobroj gp) {
    test(gp, 10);
  }
} ///:~
```

Obratite pažnju na to da, u ovom primeru, klasa čije se pravljenje može otkazati ne realizuje interfejs **Runnable**. Zato svi zadaci klase **ProveraParnosti** proveravaju da li je otkazano pravljenje objekta tipa **GeneratorCelobroj** od kojeg zavise, kao što vidite u metodi **run( )**. Na taj način, zadaci koji dele zajednički resurs (**GeneratorCelobroj**) osluškuju signal tog resursa koji im kazuje kada da okončaju svoj rad. Time se izbegava tzv. uslov za trku (engl. *race condition*), gde se dva ili više zadataka trkaju da reaguju na ispunjenje nekog uslova i zato se sudare ili na drugi način proizvedu nedosledne rezultate. Pažljivo razmislite o svim mogućim načinima na koje paralelni program može da zakaže; morate napraviti zaštitu od svakog. Na primer, zadatak ne sme da zavisi od drugog zadatka, zato što Java ne jemči redosled gašenja zadataka. U prethodnom primeru zadatak je zavisio od objekta koji nije zadatak; time je izbegnut potencijalni uslov za trku.

Metoda **test( )** priprema i obavlja testiranje svih tipova **GeneratorCelobroj** tako što pokreće više **ProveraParnosti** za isti **GeneratorCelobroj**. Ukoliko **GeneratorCelobroj** nije dao željeni rezultat, metoda **test( )** će to prijaviti i izaći; u protivnom, morate pritisnuti kombinaciju tastera Control-C da biste je okončali.

Zadaci **ProvereParnosti** stalno čitaju i ispituju brojeve koje je dao njima pridruženi **GeneratorCelobroj**. Ukoliko **generator.isCancelled( )** ima vrednost **true**, metoda **run( )** vraća svoj rezultat i izlazi, što izvršiocu u metodi **ProveraParnosti.test( )** kazuje da je zadatak obavljen. Svaki zadatak **ProvereParnosti** može da pozove metodu **cancel( )** za svoj pridruženi **GeneratorCelobroj**, čime prouzrokuje da sve druge **ProvereParnosti** tog objekta tipa **GeneratorCelobroj** normalno okončaju svoj rad. U nastavku poglavlja videćete da Java ima i opštije mehanizme za okončavanje rada niti.

Prvi **GeneratorCelobroj** koji ćemo razmotriti ima metodu **next( )** koja proizvodi seriju parnih brojeva:

```
//: paralelno/GeneratorParnih.java
// Kada se niti sudare.

public class GeneratorParnih extends GeneratorCelobroj {
  private int tekuciParniBroj = 0;
  public int next() {
    ++tekuciParniBroj; // Ovo je opasno!
    ++tekuciParniBroj;
    return tekuciParniBroj;
  }
  public static void main(String[] args) {
    ProveraParnosti.test(new GeneratorParnih());
  }
} /* Ispis: (primer)
Pritisnite Control-C ako želite da izađete iz programa
89476993 nije paran!
89476993 nije paran!
*///:~
```

Može se desiti da jedan zadatak pozove **next( )** nakon što je drugi zadatak obavio prvo povećanje broja **tekuciParniBroj** za 1, ali ne i drugo (na onom mestu u programu gde stoji komentar „Ovo je opasno!"). Time je broj dospeo u nepravilno stanje. Da bih vam dokazao da se to može dogoditi, **ProveraParnosti.test( )** pravi grupu objekata tipa **ProveraParnosti** koji stalno čitaju izlaz **GeneratoraParnih** i ispituju parnost svakog od tih brojeva. Kada se pronađe neki koji nije paran, prijavljuje se greška i program se gasi.

Taj program pre ili kasnije mora da otkaže, zato što zadaci **ProvereParnosti** mogu da pristupe informacijama u **GeneratoruParnih** dok je on u nepravilnom stanju. Međutim, problem ne mora biti otkriven dok **GeneratorParnih** ne izvrši mnogo ciklusa, čiji se broj menja u zavisnosti od operativnog sistema i pojedinosti realizacije. Ako želite da vidite otkazivanje što pre, između prvog i drugog povećanja za 1 stavite poziv metode **yield( )**. I to je deo problema s višenitnim programima – umeju da daju privid ispravnosti čak i kada greška postoji, ukoliko je verovatnoća otkazivanja veoma mala.

Važno je uočiti da se operacija povećanja za 1 interno sastoji od više koraka, i da višenitni mehanizam može zaustaviti (prekinuti) zadatak usred tih koraka – drugim rečima, u Javi povećanje za 1 nije atomska operacija. Dakle, čak se ni povećanje za 1 ne može obaviti bezbedno ako zadatak nije zaštićen.

Razrešavanje takmičenja za deljene resurse

Prethodni primer pokazuje osnovni problem s nitima. Nikad se ne zna kada će koja nit proraditi. Zamislite da sedite za stolom s viljuškom u ruci i da upravo nameravate da nabodete poslednje parče hrane u tanjiru; u trenutku kad viljuška dodirne to parče, ono iznenada nestane (pošto je vaša nit prekinuta, a pokrenuta druga nit koja je uletela i ukrala to parče). Sada razumete o kakvom problemu se radi. Da bi paralelno izvršavanje funkcionisalo, barem tokom bitnih razdoblja morate sprečiti da više zadataka istovremeno pristupi istom resursu.

Takvi sudari jednostavno se sprečavaju zaključavanjem (engl. *lock*) resursa dok ga jedna nit koristi. Prva nit koja pristupi nekom resursu prethodno ga zaključa, čime se sprečava da mu druge niti pristupaju sve dok ne bude otključan; tada ga druga nit zaključava, koristi ga itd. Ako prednje sedište u kolima predstavlja ograničen resurs, dete koje poviče: „Prvi!" i sedne na njega, zaključava ga.

Da bi se rešio problem sudaranja niti, gotovo sve šeme paralelnog rada *serijalizuju pristup deljenim resursima*. To znači da je, u svakom trenutku, pristup deljenom resursu dozvoljen samo jednom zadatku. To se obično postiže stavljanjem koda u odredbu koja, u svakom trenutku, dozvoljava samo po jednom zadatku da prolazi kroz taj deo koda. Pošto ta odredba proizvodi *uzajamno isključivanje* (engl. *mutual exclusion*), takav mehanizam se obično naziva *uzajamno isključiva brava* (engl. *mutex*).

Uzmimo kao primer vaše kupatilo; više osoba (zadataka koji se izvršavaju u nitima) može poželeti da samostalno koristi kupatilo (deljeni resurs). Kada pođe u kupatilo, osoba kuca na vrata kako bi utvrdila da li je slobodno. Ako jeste, osoba uđe i zaključa vrata. Nadalje je svima drugima koji požele da koriste to kupatilo korišćenje „blokirano", pa moraju da čekaju ispred vrata dok se kupatilo ne oslobodi.

Analogija donekle popušta kada se kupatilo oslobodi i dođe vreme da se prepusti drugom zadatku na korišćenje. Nema reda za čekanje i mi ne možemo znati ko će sledeći dobiti resurs na korišćenje, zato što mehanizam za raspodelu procesorskog vremena nitima nije u tom smislu deterministički. Za razliku od toga, kao da grupa blokiranih zadataka kruži ispred kupatila, i kada zadatak koji ga je zaključao otključa i izađe, ulazi onaj koji se u tom trenutku zatekao najbliže vratima. Već sam rekao da mehanizmu za raspodelu procesorskog vremena nitima pomoću metoda **yield( )** i **setPriority( )** predlažete kome da dodeli resurs, ali ti predlozi ne moraju imati mnogo uticaja, pošto to zavisi od platforme i realizacije JVM-a.

Da bi se sprečili sukobi zbog resursa, Java ima ugrađenu podršku u vidu rezervisane reči **synchronized**. Kada zadatak treba da izvrši deo koda zaštićen rezervisanom rečju **synchronized**, on najpre utvrdi da je resurs otključan, zaključa ga, izvrši kôd i onda otključa.

Deljeni resurs je najčešće deo memorije u obliku objekta, ali može biti i datoteka, U/I priključak ili štampač. Da biste upravljali pristupom deljenom resursu, najpre ga stavite u neki objekat. Potom sinhronizujte sve metode koje koriste taj objekat, tj. dodajte im modifikator **synchronized**. Ukoliko se neki zadatak upravo izvršava pozivom jedne od sinhronizovanih metoda, ulazak u *sve* sinhronizovane metode tog objekta blokiran je za *sve* ostale zadatke dokle god se prvi zadatak ne vrati iz svog poziva.

Već ste naučili da podaci klasa moraju biti privatni (imati modifikator **private**) u kodu koji se isporučuje kupcima; toj memoriji treba da pristupate samo preko metoda. Sukobe ćete sprečiti tako što ćete deklarisati da su te metode sinhronizovane, što se radi ovako:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Svi objekti automatski dobijaju sopstvenu bravu (koristi se i ime *monitor*). Kada pozovete bilo koju sinhronizovanu metodu, objekat se zaključava i nijedna druga sinhronizovana metoda tog objekta ne može biti pozvana dok se prva ne završi i ne otključa tu bravu. Za prethodne metode važi sledeće: ako jedan zadatak pozove **f( )** za neki objekat, drugi zadaci ne mogu pozivati ni **f( )** ni **g( )** za taj objekat dokle god **f( )** ne završi rad i ne otključa bravu. Dakle, postoji samo jedna brava koju dele sve sinhronizovane metode određenog objekta, i ona sprečava da više zadataka istovremeno upisuje u memoriju objekta.

Vodite računa o tome da je za paralelno izvršavanje izuzetno važno da polja budu privatna; ako nisu, rezervisana reč **synchronized** ne može sprečiti druge zadatke da polju pristupaju neposredno i tako izazivaju sukobe.

Isti zadatak može više puta zaključati objekat. To se dešava kada jedna metoda za isti objekat pozove drugu metodu, a druga za isti objekat pozove treću itd. JVM prati broj zaključavanja objekta. Broj zaključavanja otključanog objekta je nula. Kada zadatak prvi put zaključa taj objekat, broj zaključavanja se poveća za jedan. Svaki put kada taj zadatak ponovo zaključa taj objekat, broj zaključavanja se poveća za jedan. Naravno, samo zadatak koji je prvi zaključao taj objekat može da ga zaključa više puta. Broj zaključavanja se smanjuje za jedan kad god taj zadatak izađe iz neke od sinhronizovanih metoda; kada broj zaključavanja padne na nulu, objekat je slobodan i mogu ga koristiti drugi zadaci.

I *svaka klasa* automatski dobija svoju bravu (koja je deo njenog objekta klase **Class**), tako da sinhronizovane statičke metode (označene kao **synchronized** i **static**) mogu jedna drugu sprečiti da istovremeno pristupe statičkim podacima u okviru pojedine klase.

Kada sinhronizovati? Primenite *Brajanovo pravilo sinhronizacije*:[10]

Sinhronizaciju morate upotrebiti ukoliko pišete u promenljivu koju bi zatim mogla da pročita neka druga nit ili čitate promenljivu koju je prethodno mogla upisati neka druga nit. Nadalje, i čitanje i pisanje mora biti sinhronizovano pomoću iste brave (monitora).

Ako u klasi imate više metoda koje rade s bitnim podacima, morate sinhronizovati sve relevantne metode. Ukoliko sinhronizujete samo jednu od tih metoda, ostale mogu zanemariti bravu na objektu i biti nesmetano pozvane. Ovo je važno: *sve* metode koje pristupaju kritičnom resursu moraju biti sinhronizovane ili program neće raditi kako treba.

Sinhronizovanje **GeneratoraParnih**

Ako klasi **GeneratorParnih.java** dodamo rezervisanu reč **synchronized**, sprečićemo neželjeno pristupanje toj niti:

[10] Po Brianu Goetzu, koautoru *Java Concurrency in Practice*; ostali autori su Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea (Addison-Wesley, 2006).

```
//: paralelno/SinhronizovanGeneratorParnih.java
// Pojednostavljivanje uzajamno isključivih brava
// pomoću rezervisane reči synchronized.
// {RunByHand}

public class
SinhronizovanGeneratorParnih extends GeneratorCelobroj {
  private int tekuciParniBroj = 0;
  public synchronized int next() {
    ++tekuciParniBroj;
    Thread.yield(); // Brže izaziva zakazivanje
    ++tekuciParniBroj;
    return tekuciParniBroj;
  }
  public static void main(String[] args) {
    ProveraParnosti.test(new SinhronizovanGeneratorParnih());
  }
} ///:~
```

Poziv metode **Thread.yield**( ) umetnut je između dva povećanja za 1 da bi se povećala verovatnoća prebacivanja konteksta dok je **tekuciParniBroj** neparan. Pošto uzajamno isključiva brava sprečava da više zadataka u isto vreme bude u kritičnom delu programa, do otkazivanja neće doći, ali poziv metode **yield**( ) pomaže da se otkazivanje ranije desi ukoliko već može da nastane.

Metodu **next**( ) zaključava prvi zadatak koji uđe u nju, i svi drugi zadaci koji pokušaju da je zaključaju bivaju blokirani sve dok je prvi zadatak ne otključa. Tada mehanizam za raspodelu procesorskog vremena odabire jedan od ostalih zadataka koji čekaju da se brava otključa. Na taj način, u svakom trenutku samo po jedan zadatak može proći kroz kôd koji čuva uzajamno isključiva brava (mutex).

Vežba 11: (3) Napravite klasu koja sadrži dva polja podataka i metodu koja radi s tim poljima u postupku koji traje više koraka, tako da tokom izvršavanja te metode polja budu u nepravilnom stanju (u odnosu na definiciju koju ćete sami dati). Dodajte metode koje čitaju ta polja i napravite više niti za pozivanje raznih metoda i pokažite da su podaci vidljivi i kada su nepravilni. Rešite problem pomoću rezervisane reči **synchronized**.

Upotreba eksplicitnih brava (**Lock** objekata)

Biblioteka Java SE5 **java.util.concurrent** sadrži i eksplicitan mehanizam uzajamno isključive brave (mutexa) definisan u paketu **java.util.concurrent.locks**. Objekat tipa **Lock** morate eksplicitno praviti, zaključavati i otključavati; zato je njegov kôd manje elegantan od ugrađenih oblika. Međutim, prilagodljiviji je za rešavanje određenih vrsta problema. Evo kako izgleda **SinhronizovanGeneratorParnih.java** u verziji sa eksplicitnim bravama (objektima tipa **Lock**):

```
//: paralelno/MutexGeneratorParnih.java
// Sprečavanje sukoba niti pomoću uzajamno isključivih brava (mutexa).
// {RunByHand}
import java.util.concurrent.locks.*;
```

```
public class MutexGeneratorParnih extends GeneratorCelobroj {
  private int tekuciParniBroj = 0;
  private Lock brava = new ReentrantLock();
  public int next() {
    brava.lock();
    try {
      ++tekuciParniBroj;
      Thread.yield(); // Brže izaziva otkazivanje
      ++tekuciParniBroj;
      return tekuciParniBroj;
    } finally {
      brava.unlock();
    }
  }
  public static void main(String[] args) {
    ProveraParnosti.test(new MutexGeneratorParnih());
  }
} ///:~
```

MutexGeneratorParnih dodaje mutex **brava** i metodama **lock**( ) i **unlock**( ) stvara kritični deo unutar metode **next**( ). Prilikom upotrebe **Lock** objekata morate da internalizujete ovde prikazani idiom: neposredno iza poziva metode **lock**( ) morate staviti blok **try-finally**, uz **unlock**( ) u odredbi **finally** – to je *jedini* način da brava uvek bude otključana. Naredba **return** mora biti unutar odredbe **try** da otključavanje metodom **unlock**( ) ne bi došlo prerano i izložilo podatke drugim zadacima.

Iako blok **try-finally** zahteva više koda nego rezervisana reč **synchronized**, on predstavlja i jednu od prednosti eksplicitnih **Lock** objekata. Kada program otkaže uz rezervisanu reč **synchronized**, baca se izuzetak, ali programer nema priliku da počisti i tako sistem vrati u dobro stanje. Ukoliko upotrebljavamo eksplicitne **Lock** objekte, imamo odredbu **finally** unutar koje možemo obaviti sve čišćenje potrebno da bismo sistem vratili u odgovarajuće stanje.

Uopšte uzev, uz **synchronized** se piše manje koda, pa je manja prilika da programer pogreši. Stoga se eksplicitni **Lock** objekti obično koriste samo za rešavanje specijalnih problema. Na primer, uz rezervisanu reč **synchronized** svakako ćete uspeti da zaključate zadatak. Uz to, ne možete pokušavati da zaključate zadatak pa odustati – morate upotrebiti biblioteku **concurrent**:

```
//: paralelno/PokusajZakljucavanja.java
// Brave u biblioteci concurrent dozvoljavaju
// da odustanete od pokušaja zaključavanja.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class PokusajZakljucavanja {
  private ReentrantLock brava = new ReentrantLock();
  public void neodredjenoVreme() {
    boolean zakljucano = brava.tryLock();
```

```
    try {
      System.out.println("tryLock(): " + zakljucano);
    } finally {
      if(zakljucano)
        brava.unlock();
    }
  }
  public void odredjenoVreme() {
    boolean zakljucano = false;
    try {
      zakljucano = brava.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    try {
      System.out.println("tryLock(2, TimeUnit.SECONDS): " +
        zakljucano);
    } finally {
      if(zakljucano)
        brava.unlock();
    }
  }
  public static void main(String[] args) {
    final PokusajZakljucavanja pz = new PokusajZakljucavanja();
    pz.neodredjenoVreme(); // True -- brava je otključana
    pz.odredjenoVreme();   // True -- brava je otključana
    // Sada napravi zaseban zadatak koji će pokušati da zaključa:
    new Thread() {
      { setDaemon(true); }
      public void run() {
        pz.brava.lock();
        System.out.println("zaključano");
      }
    }.start();
    Thread.yield(); // Daj priliku drugom zadatku
    pz.neodredjenoVreme(); // False -- prethodni zadatak je već zaključao
    pz.odredjenoVreme();   // False -- prethodni zadatak je već zaključao
  }
} /* Ispis:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
zaključano
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*///:~
```

Objekat tipa **ReentrantLock** omogućuje da pokušate zaključati bravu i da u tome ne uspete; ako je neki drugi zadatak bravu već zaključao, možete nešto drugo da radite umesto da samo čekate dok se brava ne oslobodi, kao u metodi **neodredjenoVreme**( ).

Pokušaj zaključavanja u metodi **odredjenoVreme( )** traje najviše 2 sekunde dok se ne proglasi neuspešnim (vremenska jedinica je zadata pomoću klase **TimeUnit** iz Jave SE5). U metodi **main( )**, zasebna nit se pravi kao anonimna klasa koja zaključava bravu, da bi metode **neodredjenoVreme( )** i **odredjenoVreme( )** imale oko čega da se takmiče.

Eksplicitni objekti tipa **Lock** omogućuju finiju kontrolu nad zaključavanjem i otključavanjem od ugrađenih brava **synchronized**. Oni služe za realizaciju specijalizovanih struktura za sinhronizaciju, kao što je *povezano zaključavanje* (engl. *hand-over-hand locking* ili *lock coupling*) koje se upotrebljava za prolaz kroz ulančanu listu – kôd za prolaz mora zaključati bravu sledećeg čvora pre nego što otključa bravu tekućeg čvora.

Atomske operacije i trenutna vidljivost

U raspravama o višenitnom programiranju na Javi često se ponavlja netačna priča da „atomske operacije ne moraju biti sinhronizovane". *Atomska* je svaka operacija koju mehanizam za raspodelu procesorskog vremena nitima ne može da prekine; ako takva operacija otpočne, prebacivanje konteksta će postati moguće tek kada se ona završi. Osloniti se na neprekidnost operacije je škakljivo i opasno – ne bi trebalo da se usudite da umesto sinhronizacije koristite neprekidnost ukoliko niste stručnjak za paralelno programiranje ili ukoliko vam takav stručnjak ne pomaže. Ako mislite da ste dovoljno pametni da se igrate s vatrom, podvrgnite se ovom testu:

Gecov test[11]: Ako umete da napišete JVM visokih performansi za savremeni mikroprocesor, onda ste kvalifikovani da *razmislite* o tome možete li da izbegnete sinhronizaciju.[12]

Korisno je *znati* da neprekidne operacije postoje i da su one, uz druge napredne tehnike, bile upotrebljene za realizaciju dela pametnijih komponenata biblioteke **java.util.concurrent**. Ali oduprite se nagonu da ih sami koristite (ponovo pročitajte Brajanovo pravilo za sinhronizaciju).

Atomske su „jednostavne operacije" na prostim tipovima (sem tipova **long** i **double**). Čitanje i pisanje prostih promenljivih (sem onih tipa **long** ili **double**) iz memorije i u memoriju zajemčeno su neprekidne (atomske) operacije. Međutim, čitanje i pisanje 64-bitnih vrednosti (promenljivih tipa **long** ili **double**) JVM sme da podeli na dve zasebne 32-bitne operacije; time se povećava verovatnoća da prebacivanje konteksta nastane usred čitanja ili pisanja, pa bi razni zadaci videli pogrešne rezultate (to se ponekad naziva *cepanje reči*, pošto se može desiti da vidite vrednost, tj. rezultat operacije, nakon što je izračunat samo jedan njen deo). Neprekidnost (jednostavnih dodeljivanja i vraćanja rezultata) ostvaruje se ukoliko ispred definicije **long** ili **double** promenljivih stavite modifikator **volatile** (koji pre Jave SE5 nije radio ispravno, ali sada radi). Razne JVM smeju da jemče i nešto više, ali ne bi trebalo da koristite sposobnosti koje se menjaju u zavisnosti od platforme.

Dakle, mehanizam za raspodelu procesorskog vremena nitima ne može da prekine atomske operacije. Izuzetno stručni programeri to koriste za pisanje *koda bez zaključavanja* koji se ne mora sinhronizovati. Ali čak je i ta tvrdnja preterano pojednostavljena.

---

[11] Po prethodno pomenutom Brajanu Gecu (Brian Goetz), stručnjaku za paralelno programiranje koji mi je pomogao da napišem ovo poglavlje na osnovu njegovih samo delimično drskih komentara.

[12] Iz ovog testa neposredno sledi: „Ako neko tvrdi da je paralelno programiranje lako i jednostavno, postarajte se da ta osoba ne donosi važne odluke o projektu na kojem radite. Ukoliko to nikako ne možete da izbegnete, imate problem."

Katkada izgleda da bi atomska operacija trebalo da bude bezbedna, ali ona to nije. Većina čitalaca ove knjige sigurno neće moći da položi spomenuti Gecov test, te uopšte nisu kvalifikovani *ni da pokušaju* da sinhronizaciju zamene atomskim operacijama. Pokušaj uklanjanja sinhronizacije obično je znak prerane optimizacije i prouzrokuje grdne probleme, a dobitak je mali ili nikakav.

Na višeprocesorskim sistemima (koji se sada prave u obliku procesora s više jezgara – to je više procesora u jednom integrisanom kolu), mnogo veći problem od neprekidnosti operacija je *vidljivost*. (Taj problem ne postoji u jednoprocesorskim sistemima.) Izmene koje napravi jedan zadatak, čak i kada su atomske, odnosno neprekidne, ne moraju biti vidljive drugim zadacima (jer su privremeno smeštene u lokalni keš procesora, na primer), pa razni zadaci dobijaju različite uvide u stanje aplikacije. S druge strane, mehanizam sinhronizacije obezbeđuje da izmene koje na višeprocesorskom sistemu napravi jedan zadatak budu vidljive celoj aplikaciji. Bez sinhronizacije se *ne može* znati kada će izmene postati vidljive.

Vidljivost u celoj aplikaciji obezbeđuje i rezervisana reč **volatile**. Ako deklarišete da je neko polje **volatile**, to znači da će svi zadaci koji ga čitaju videti izmenu čim se upisivanje u njega završi. To važi čak i za lokalnu keš memoriju, zato što se polja označena s **volatile** odmah prepisuju u glavnu memoriju, a sva čitanja se obavljaju iz glavne memorije.

Nadam se da shvatate kako su neprekidnost i trenutna vidljivost različiti pojmovi. Atomska operacija polja koje nema modifikator **volatile** ne mora da bude odmah prepisana u glavnu memoriju, pa ni drugi zadaci koji čitaju to polje ne moraju odmah da vide novu vrednost. Svako polje kojem pristupa više zadataka treba da ima modifikator **volatile**; u protivnom, polju treba pristupati samo preko sinhronizacije. I sinhronizacija prouzrokuje prepisivanje u glavnu memoriju, pa polje koje je potpuno zaštićeno sinhronizovanim metodama ili blokovima ne mora da ima modifikator **volatile**.

Rezultati svih upisivanja koja zadatak obavi odmah su vidljivi tom zadatku, pa polju ne morate da dodajete modifikator **volatile** ukoliko ga čita samo njegov zadatak.

volatile ne funkcioniše kada vrednost polja zavisi od njegove prethodne vrednosti (takvo je povećanje vrednosti brojača za 1), niti funkcioniše za vrednosti koje su ograničene vrednostima drugih polja, kao što su **lower** (donja) i **upper** (gornja) granica objekta tipa **Range** koje moraju da zadovolje ograničenje **lower** <= **upper**.

Upotreba modifikatora **volatile** umesto rezervisane reči **synchronized** bezbedna je najčešće jedino u slučaju da klasa ima samo jedno promenljivo polje. Ponavljam, najbolje je izabrati rezervisanu reč **synchronized** – to je najbezbedniji pristup, a sve ostalo je rizično.

Koje operacije su atomske? Dodeljivanje i vraćanje vrednosti polja obično su atomske operacije. Međutim, u C++-u bi atomske *mogle* biti čak i ove operacije:

```
i++;    // Možda je atomska u C++-u
i += 2; // Možda je atomska u C++-u
```

Ali u C++-u to zavisi od prevodioca i procesora (zato piše *možda*). Na C++-u se ne može pisati prenosivi (međuplatformski) kôd koji koristi neprekidnost operacija, pošto C++ nema dosledan *model memorije* – ali Java ima (u Javi SE5).[13]

[13] Ovo će biti popravljeno u sledećem standardu C++-a.

U Javi gornje operacije sigurno *nisu* atomske, kao što vidite iz JVM instrukcija koje proizvode sledeće metode:

```
//: paralelno/Neprekidnost.java
// {Pokretanje: javap -c Neprekidnost}

public class Neprekidnost {
  int i;
  void f1() { i++; }
  void f2() { i += 3; }
} /* Ispis: (primer)
...
void f1();
  Code:
   0:        aload_0
   1:        dup
   2:        getfield        #2; //Polje i:I
   5:        iconst_1
   6:        iadd
   7:        putfield        #2; //Polje i:I
   10:        return

void f2();
  Code:
   0:        aload_0
   1:        dup
   2:        getfield        #2; //Polje i:I
   5:        iconst_3
   6:        iadd
   7:        putfield        #2; //Polje i:I
   10:        return
*///:-
```

Svaka instrukcija proizvodi jedan get i jedan put između kojih ima drugih instrukcija. Dakle, između čitanja (get) i upisivanja (put) neki drugi zadatak može da izmeni vrednost polja; stoga te operacije nisu atomske.

Ako slepo primenjujete neprekidnost, mogli biste pomisliti da je takva i metoda **dajVrednost( )** u sledećem programu:

```
//: paralelno/TestNeprekidnosti.java
import java.util.concurrent.*;

public class TestNeprekidnosti implements Runnable {
  private int i = 0;
  public int dajVrednost() { return i; }
  private synchronized void povecanjeZaDva() { i++; i++; }
  public void run() {
    while(true)
      povecanjeZaDva();
  }
```

```
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    TestNeprekidnosti at = new TestNeprekidnosti();
    exec.execute(at);
    while(true) {
      int vre = at.dajVrednost();
      if(vre % 2 != 0) {
        System.out.println(vre);
        System.exit(0);
      }
    }
  }
} /* Ispis: (primer)
191583767
*///:~
```

Međutim, program je pronašao neparan broj i prekinuo rad. Premda je **return i** zaista atomska operacija, zbog nepostojanja sinhronizacije **i** može biti pročitan dok je objekat u nestabilnom međustanju (između jednog i drugog `i++`). Sem toga, biće i problema s vidljivošću, pošto ispred **i** nema modifikatora **volatile**. Obe metode, i **dajVrednost( )** *i* **povecanjeZaDva( )** moraju biti sinhronizovane. Samo su stručnjaci za paralelno programiranje kvalifikovani da u ovakvim situacijama pokušaju optimizaciju; ponavljam, trebalo bi da primenjujete Brajanovo pravilo za sinhronizaciju.

Kao drugi primer, razmotrite nešto još jednostavnije: klasu koja proizvodi serijske brojeve.[14] Metoda **sledeciSerijskiBroj( )** mora pozivaocu da vrati jedinstven broj svaki put kada bude pozvana:

```
//: paralelno/GeneratorSerijskihBrojeva.java

public class GeneratorSerijskihBrojeva {
  private static volatile int serijskiBroj = 0;
  public static int sledeciSerijskiBroj() {
    return serijskiBroj++; // Nije bezbedno za višenitno izvršavanje
  }
} ///:~
```

GeneratorSerijskihBrojeva je otprilike najjednostavnija klasa koja se može zamisliti, i ako ste dosad koristili C++ ili neki drugi jezik u kojem se programira na niskom nivou, možda očekujete da je povećanje za 1 (engl. *increment*) atomska operacija, pošto se operacija povećanja za 1 u C++-u često može realizovati kao jedna mikroprocesorska instrukcija (iako ni to ne na način pouzdan na svim platformama). Međutim, već sam rekao da u Javi povećanje za 1 *nije* atomska operacija jer obuhvata jedno čitanje i jedno upisivanje, pa čak i u tako jednostavnoj operaciji ima prostora za probleme s nitima. Kao što ćete videti, ovde problem nije trenutna vidljivost rezultata; pravi problem je to što **sledeciSerijskiBroj( )** pristupa deljenoj promenljivoj vrednosti bez sinhronizacije.

[14] Ideja iz knjige *Efikasno programiranje na Javi* Joshue Blocha (Mikro knjiga, 2004), s. 168.

Polje **serijskiBroj** ima modifikator **volatile** zato što svaka nit može da ima lokalni stek i da u njemu održava kopije nekih promenljivih. Ako ispred definicije promenljive stavite modifikator **volatile**, time kazujete prevodiocu da ne sprovodi optimizacije kojima bi se uklonila čitanja i upisivanja koja polje održavaju u sinhronizaciji s lokalnim podacima u nitima. Zato se čitanja i upisivanja obavljaju neposredno u glavnoj memoriji, a ne u kešu. Modifikator **volatile** ograničava i prevodiočevo preraspoređivanje pristupa tokom optimizacije. Međutim, **volatile** ne utiče na činjenicu da povećanje za 1 nije atomska operacija.

U suštini, definišite da je polje **volatile** ukoliko mu više zadataka može istovremeno pristupiti, a barem jedno od tih pristupanja je upisivanje. Primera radi, polje koje se upotrebljava kao indikator za zaustavljanje zadatka mora biti deklarisano kao **volatile**; u protivnom bi indikator mogao biti keširan u nekom registru procesora, i ta vrednost se ne bi izmenila kada vi iz nekog drugog zadatka izmenite vrednost indikatora, pa prvi zadatak ne bi znao kada treba da se zaustavi.

Za ispitivanje **GeneratoraSerijskihBrojeva** treba nam skup kojem neće ponestati memorije ni u slučaju da pronalaženje problema potraje. Ovde prikazani **KruzniSkup** ponovo upotrebljava memoriju u kojoj su bili uskladišteni celi brojevi, uz pretpostavku da će verovatnoća sukoba sa zamenjenim brojevima biti minimalna kada se s kraja skupa vratimo na njegov početak. Metode **add( )** i **contains( )** sinhronizovane su da bi se sprečilo sudaranje niti:

```
//: paralelno/ProveraSerijskihBrojeva.java
// Operacije prestaju da budu bezbedne
// kada izvršavanje postane višenitno.
// {Args: 4}
import java.util.concurrent.*;

// Ponovo upotrebljava isto parče memorije da je ne bi ponestalo:
class KruzniSkup {
  private int[] niz;
  private int duzina;
  private int indeks = 0;
  public KruzniSkup(int velicina) {
    niz = new int[velicina];
    duzina = velicina;
    // Inicijalizacija brojem koji GeneratorSerijskihBrojeva
    // ne proizvodi:
    for(int i = 0; i < velicina; i++)
      niz[i] = -1;
  }
  public synchronized void add(int i) {
    niz[indeks] = i;
    // Vrati indeks na početak niza i počni da prepisuješ
    // nove elemente preko starih:
    indeks = ++indeks % duzina;
  }
```

```
  public synchronized boolean contains(int vre) {
    for(int i = 0; i < duzina; i++)
      if(niz[i] == vre) return true;
    return false;
  }
}

public class ProveraSerijskihBrojeva {
  private static final int VELICINA = 10;
  private static KruzniSkup serijskibr =
    new KruzniSkup(1000);
  private static ExecutorService exec =
    Executors.newCachedThreadPool();
  static class ProveraSerijskih implements Runnable {
    public void run() {
      while(true) {
        int serijski =
          GeneratorSerijskihBrojeva.sledeciSerijskiBroj();
        if(serijskibr.contains(serijski)) {
          System.out.println("Duplikat: " + serijski);
          System.exit(0);
        }
        serijskibr.add(serijski);
      }
    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < VELICINA; i++)
      exec.execute(new ProveraSerijskih());
    // Ako argument postoji, prestani nakon n sekundi:
    if(args.length > 0) {
      TimeUnit.SECONDS.sleep(new Integer(args[0]));
      System.out.println("Nijedan duplikat nije pronađen");
      System.exit(0);
    }
  }
} /* Ispis: (primer)
Duplikat: 8468656
*///:~
```

ProveraSerijskihBrojeva sadrži statički **KruzniSkup** koji obuhvata sve proizvedene serijske brojeve i ugneždenu klasu **ProveraSerijskih** koja proverava da li je svaki od tih serijskih brojeva jedinstven. Kada napravite više zadataka koji se takmiče za dobijanje serijskih brojeva, videćete da će jedan od njih pre ili kasnije dobiti postojeći serijski broj (duplikat), samo ga dovoljno dugo pustite da radi. Problem ćete rešiti sinhronizovanjem metode **sledeciSerijskiBroj( )**. (Dodajte rezervisanu reč **synchronized** ispred nje.)

Ako išta, onda bi čitanje i dodeljivanje prostih tipova trebalo da budu atomske operacije. Međutim, kao što vidite iz programa **TestNeprekidnosti.java**, i dalje je lako pozvati

atomsku operaciju koja pristupa objektu dok je on u nekom nestabilnom međustanju. Donositi bilo kakve zaključke o tome škakljivo je i opasno. Najrazboritije je prosto pridržavati se Brajanovog pravila za sinhronizaciju.

Vežba 12: (3) Popravite **TestNeprekidnosti.java** pomoću rezervisane reči **synchronized**. Umete li da pokažete kako je program sada ispravan?

Vežba 13: (1) Popravite program **ProveraSerijskihBrojeva.java** pomoću rezervisane reči **synchronized**. Umete li da pokažete kako je program sada ispravan?

Atomske klase

Java SE5 je donela i posebne atomske klase promenljivih kao što su **AtomicInteger**, **AtomicLong**, **AtomicReference** itd., koje obezbeđuju atomsko uslovno ažuriranje oblika:

```
boolean uporediIAžuriraj(očekivanaVrednost, novaVrednost);
```

One su namenjene za fino podešavanje pri kome se koriste atomske operacije (neprekidnosti) dostupne na nekim od savremenih procesora, pa o njima po pravilu ne bi trebalo da razbijate glavu. Ponekad mogu da posluže i za obično programiranje, ali opet tamo gde se radi o podešavanju performansi. Na primer, izmenićemo **TestNeprekidnosti.java** tako da upotrebljava klasu **AtomicInteger**:

```
//: paralelno/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
  private AtomicInteger i = new AtomicInteger(0);
  public int dajVrednost() { return i.get(); }
  private void povecanjeZaDva() { i.addAndGet(2); }
  public void run() {
    while(true)
      povecanjeZaDva();
  }
  public static void main(String[] args) {
    new Timer().schedule(new TimerTask() {
      public void run() {
        System.err.println("Prekidam");
        System.exit(0);
      }
    }, 5000); // Završi rad nakon 5 sekundi
    ExecutorService exec = Executors.newCachedThreadPool();
    AtomicIntegerTest ait = new AtomicIntegerTest();
    exec.execute(ait);
    while(true) {
      int vre = ait.dajVrednost();
      if(vre % 2 != 0) {
```

```
          System.out.println(vre);
          System.exit(0);
        }
      }
    }
  } ///:~
```

Upotrebom klase **AtomicInteger** izbegli smo korišćenje rezervisane reči **synchronized**. Pošto ovaj program ne otkazuje, dodat je **Timer** koji automatski prekida rad nakon 5 sekundi.

Evo kako izgleda **MutexGeneratorParnih.java** izmenjen tako da upotrebljava klasu **AtomicInteger**:

```
//: paralelno/AtomskiGeneratorParnih.java
// Atomske klase se povremeno koriste i u običnim programima.
// {RunByHand}
import java.util.concurrent.atomic.*;

public class AtomskiGeneratorParnih extends GeneratorCelobroj {
  private AtomicInteger tekuciParniBroj =
    new AtomicInteger(0);
  public int next() {
    return tekuciParniBroj.addAndGet(2);
  }
  public static void main(String[] args) {
    ProveraParnosti.test(new AtomskiGeneratorParnih());
  }
} ///:~
```

I opet smo korišćenjem klase **AtomicInteger** izbegli sve ostale oblike sinhronizacije.

Treba naglasiti da klase iz paketa **Atomic** služe za izgradnju klasa u biblioteci **java.util.concurrent** pa ih u svojim programima upotrebljavajte samo u posebnim okolnostima, a čak i tada samo ukoliko možete da izbegnete sve druge moguće probleme. Obično je bezbednije raditi s bravama i zaključavanjem (pomoću rezervisane reči **synchronized** ili eksplicitnih objekata tipa **Lock**).

Vežba 14: (4) Pokažite da **java.util.Timer** daje velike brojeve tako što ćete napisati program koji generiše mnoge **Timer** objekte što obavljaju neke jednostavne zadatke kad istekne vreme.

Kritični odeljci

Ponekad treba izolovati samo deo koda unutar metode **run**( ), a ne celu metodu. Deo koji treba izolovati naziva se kritičan (engl. *critical*), a rezervisana reč **synchronized** se u kritičnom delu programa koristi drugačije. U sledećem primeru reč **synchronized** zaključava bravu određenog objekta da bi sinhronizovala obuhvaćeni blok koda:

```
synchronized(brava) {
  // Ovom kodu može da pristupi samo jedna nit u jednom trenutku
}
```

Obuhvaćeni blok koda naziva se *sinhronizovani blok*; pre ulaska u sinhronizovani blok, morate da zaključate objekat **brava**. Ako je neka druga nit već zaključala taj objekat, u taj blok nije moguće ući sve dok ne bude otključan.

U sledećem primeru upoređena su oba pristupa sinhronizaciji. Pokazano je da se vreme tokom kojega je objekat dostupan drugim zadacima znatno produžava kada se koristi sinhronizovani blok umesto sinhronizovanja cele metode. Pored toga, pokazano je kako se nezaštićena klasa može upotrebiti u višenitnoj situaciji ukoliko je kontroliše i štiti druga klasa:

```
//: paralelno/KriticanDeo.java
// Sinhronizovanje blokova umesto celih metoda. Pokazuje i zaštitu klase
// nebezbedne u višenitnom izvršavanju pomoću druge koja je bezbedna.
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Par { // Nije bezbedna u višenitnom izvršavanju
  private int x, y;
  public Par(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Par() { this(0, 0); }
  public int dajX() { return x; }
  public int dajY() { return y; }
  public void povecajXzaJedan() { x++; }
  public void povecajYzaJedan() { y++; }
  public String toString() {
    return "x: " + x + ", y: " + y;
  }
  public class IzuzetakBrojeviUParuNisuJednaki
  extends RuntimeException {
    public IzuzetakBrojeviUParuNisuJednaki() {
      super("Brojevi u paru nisu jednaki: " + Par.this);
    }
  }
  // Proizvoljnih, ali nepromenljivih vrednosti --
  // obe promenljive moraju biti jednake:
  public void checkState() {
    if(x != y)
      throw new IzuzetakBrojeviUParuNisuJednaki();
  }
}

// Zaštita Para unutar klase bezbedne u višenitnom izvršavanju:
abstract class MenadzerPara {
```

```
  AtomicInteger brojacProvera = new AtomicInteger(0);
  protected Par p = new Par();
  private List<Par> skladiste =
    Collections.synchronizedList(new ArrayList<Par>());
  public synchronized Par dajPar() {
    // Napraviću kopiju da bi original bio bezbedan:
    return new Par(p.dajX(), p.dajY());
  }
  // Pretpostaviću da je ova operacija dugačka
  protected void sacuvaj(Par p) {
    skladiste.add(p);
    try {
      TimeUnit.MILLISECONDS.sleep(50);
    } catch(InterruptedException ignore) {}
  }
  public abstract void povecajZaJedan();
}

// Sinhronizacija cele metode:
class MenadzerPara1 extends MenadzerPara {
  public synchronized void povecajZaJedan() {
    p.povecajXzaJedan();
    p.povecajYzaJedan();
    sacuvaj(dajPar());
  }
}

// Zaštitiću samo kritičan deo:
class MenadzerPara2 extends MenadzerPara {
  public void povecajZaJedan() {
    Par privr;
    synchronized(this) {
      p.povecajXzaJedan();
      p.povecajYzaJedan();
      privr = dajPar();
    }
    sacuvaj(privr);
  }
}

class RukovalacPara implements Runnable {
  private MenadzerPara mp;
  public RukovalacPara(MenadzerPara mp) {
    this.mp = mp;
  }
  public void run() {
    while(true)
      mp.povecajZaJedan();
  }
```

```
    public String toString() {
      return "Par: " + mp.dajPar() +
        " brojacProvera = " + mp.brojacProvera.get();
    }
  }

  class ProveraPara implements Runnable {
    private MenadzerPara mp;
    public ProveraPara(MenadzerPara mp) {
      this.mp = mp;
    }
    public void run() {
      while(true) {
        mp.brojacProvera.incrementAndGet();
        mp.dajPar().checkState();
      }
    }
  }

  public class KriticanDeo {
    // Poređenje dva različita pristupa:
    static void
    pristupiTestiranju(MenadzerPara menp1, MenadzerPara menp2) {
      ExecutorService exec = Executors.newCachedThreadPool();
      RukovalacPara
        mp1 = new RukovalacPara(menp1),
        mp2 = new RukovalacPara(menp2);
      ProveraPara
        proverapara1 = new ProveraPara(menp1),
        proverapara2 = new ProveraPara(menp2);
      exec.execute(mp1);
      exec.execute(mp2);
      exec.execute(proverapara1);
      exec.execute(proverapara2);
      try {
        TimeUnit.MILLISECONDS.sleep(500);
      } catch(InterruptedException e) {
        System.out.println("Spavanje prekinuto");
      }
      System.out.println("mp1: " + mp1 + "\nmp2: " + mp2);
      System.exit(0);
    }
    public static void main(String[] args) {
      MenadzerPara
        menp1 = new MenadzerPara1(),
        menp2 = new MenadzerPara2();
      pristupiTestiranju(menp1, menp2);
    }
```

```
} /* Ispis: (primer)
mp1: Par: x: 15, y: 15 brojacProvera = 272565
mp2: Par: x: 16, y: 16 brojacProvera = 3956974
*///:~
```

Kao što rekoh, **Par** nije bezbedan u višenitnom izvršavanju zato što njegova invarijanta (priznajem, sasvim proizvoljno) zahteva da obe promenljive zadrže istu vrednost. Sem toga, već ste u prethodnom delu poglavlja videli da operacije povećanja za jedan nisu bezbedne u višenitnom izvršavanju, a pošto nijedna od metoda nije sinhronizovana, ne možete očekivati da će objekat tipa **Par** ostati nepokvaren u višenitnom programu.

Recimo da vam je neko dao klasu **Par** koja nije bezbedna u višenitnom izvršavanju, a vi morate da je upotrebite u višenitnom okruženju. To ćete postići pravljenjem klase **MenadzerPara** koja sadrži objekat tipa **Par** i kontroliše pristupanje njemu. Vodite računa o tome da su jedine javne metode **dajPar( )**, koja je sinhronizovana, i apstraktna metoda **povecajZaJedan( )**. Metoda **povecajZaJedan( )** biće sinhronizovana na mestu svoje realizacije.

Struktura klase **MenadzerPara**, u kojoj funkcije realizovane u osnovnoj klasi upotrebljavaju jednu ili više apstraktnih metoda definisanih u izvedenim klasama, naziva se projektni obrazac *Template Method*.[15] Projektni obrasci omogućuju kapsuliranje promena u kodu; deo koji se ovde menja je metoda **povecajZaJedan( )**. U klasi **MenadzerPara1** sinhronizovana je cela metoda **povecajZaJedan( )**, dok je u klasi **MenadzerPara2** pomoću sinhronizovanog bloka sinhronizovan samo deo metode **povecajZaJedan( )**. Obratite pažnju na to da rezervisana reč **synchronized** ne spada u potpis metode i stoga se može dodati prilikom preklapanja.

Metoda **sacuvaj( )** dodaje objekat tipa **Par** sinhronizovanoj listi tipa **ArrayList**, pa je ta operacija bezbedna u višenitnom izvršavanju. Stoga nju ne moram da štitim, pa sam je smestio izvan sinhronizovanog bloka u klasi **MenadzerPara2**.

RukovalacPara je napravljen da bi pozivanjem metode **povecajZaJedan( )** iz zadatka testirao dve različite realizacije apstraktne klase **MenadzerPara**, dok se **ProveraPara** obavlja iz drugog zadatka. Svaki put kada je **ProveraPara** uspešna, ona **brojacProvera** poveća za 1 i tako prati učestalost prilika da obavi testiranje. U metodi **main( )**, prave se dva objekta tipa **RukovalacPara** koji neko vreme rade pa se potom prikazuju rezultati svakog od njih.

Iako će se izlazni rezultati verovatno mnogo razlikovati od jednog do drugog izvršavanja, videćete da metoda **MenadzerPara1.povecajZaJedan( )** ne dozvoljava klasi **ProveraPara** pristup ni približno onoliko često kao metoda **MenadzerPara2.povecajZaJedan( )** koja ima sinhronizovan blok i zato duže radi bez zaključavanja. Obično je baš to razlog za upotrebu sinhronizovanog bloka umesto sinhronizacije cele metode: omogućiti drugim zadacima češći i duži pristup (koliko je to bezbedno).

Za zaštitu kritičnih delova programa možete upotrebiti i eksplicitne objekte tipa **Lock**:

```
//: paralelno/EksplicitanKriticanDeo.java
// Zaštita kritičnih delova programa eksplicitnim objektima tipa Lock.
package concurrency;
import java.util.concurrent.locks.*;
```

[15] Videti knjigu *Design Patterns*; autor je Gamma i dr. (Addison-Wesley, 1995).

```
// Sinhronizacija cele metode:
class EksplicitanMenadzerPara1 extends MenadzerPara {
  private Lock brava = new ReentrantLock();
  public synchronized void povecajZaJedan() {
    brava.lock();
    try {
      p.povecajXzaJedan();
      p.povecajYzaJedan();
      sacuvaj(dajPar());
    } finally {
      brava.unlock();
    }
  }
}

// Kritičan deo:
class EksplicitanMenadzerPara2 extends MenadzerPara {
  private Lock brava = new ReentrantLock();
  public void povecajZaJedan() {
    Par privr;
    brava.lock();
    try {
      p.povecajXzaJedan();
      p.povecajYzaJedan();
      privr = dajPar();
    } finally {
      brava.unlock();
    }
    sacuvaj(privr);
  }
}

public class EksplicitanKriticanDeo {
  public static void main(String[] args) throws Exception {
    MenadzerPara
      menp1 = new EksplicitanMenadzerPara1(),
      menp2 = new EksplicitanMenadzerPara2();
    KriticanDeo.pristupiTestiranju(menp1, menp2);
  }
} /* Ispis: (primer)
mp1: Par: x: 15, y: 15 brojacProvera = 174035
mp2: Par: x: 16, y: 16 brojacProvera = 2608588
*///:~
```

Ponovo smo upotrebili najveći deo programa **KriticanDeo.java** i napravili nove tipove od **MenadzerPara** koji upotrebljavaju eksplicitne objekte tipa **Lock**. **EksplicitanMenadzerPara2** prikazuje pravljenje kritičnog dela pomoću objekta tipa **Lock**; poziv metode **sacuvaj**( ) nalazi se izvan kritičnog dela.

Sinhronizacija s drugim objektima

Sinhronizovanom bloku mora biti dat objekat s kojim će se sinhronizovati, i obično je za to najpametnije upotrebiti tekući objekat za koji se metoda poziva: **synchronized(this)**; upravo to je urađeno u klasi **MenadzerPara2**. Na taj način, ostale sinhronizovane metode i kritični delovi objekta ne mogu biti pozvani nakon zaključavanja sinhronizovanog bloka. Dakle, uticaj sinhronizovanog bloka na kritičan deo objekta, kada se sinhronizuje na **this**, svodi se samo na smanjenje opsega sinhronizacije.

Katkada se sinhronizacija mora obaviti s nekim drugim objektom, ali se u tom slučaju svi relevantni zadaci moraju sinhronizovati sa istim objektom. U narednom primeru pokazano je da dva zadatka mogu ući u objekat čije su metode sinhronizovane s različitim bravama:

```
//: paralelno/ObjekatSinhronizacije.java
// Sinhronizacija s drugim objektom.
import static net.mindview.util.Print.*;

class DvojnaSinhronizacija {
  private Object objekatSinhronizacije = new Object();
  public synchronized void f() {
    for(int i = 0; i < 5; i++) {
      print("f()");
      Thread.yield();
    }
  }
  public void g() {
    synchronized(objekatSinhronizacije) {
      for(int i = 0; i < 5; i++) {
        print("g()");
        Thread.yield();
      }
    }
  }
}

public class ObjekatSinhronizacije {
  public static void main(String[] args) {
    final DvojnaSinhronizacija ds = new DvojnaSinhronizacija();
    new Thread() {
      public void run() {
        ds.f();
      }
    }.start();
    ds.g();
  }
} /* Ispis: (primer)
g()
f()
g()
```

```
f()
g()
f()
g()
f()
g()
f()
*///:~
```

DvojnaSinhronizacija.f( ) sinhronizuje na objekat **this** (sinhronizacijom cele metode), a **g( )** ima sinhronizovan blok koji sinhronizuje na **objekatSinhronizacije**. Dakle, te dve sinhronizacije su nezavisne. To je dokazano u metodi **main( )** pravljenjem niti (objekta tipa **Thread**) koja poziva **f( )**. Glavna nit – metode **main( )** – upotrebljena je za poziv **g( )**. Iz rezultata vidite da se obe metode izvršavaju istovremeno, dakle sinhronizacija jedne ne blokira onu drugu.

Vežba 15: (1) Napravite klasu s tri metode koje sadrže kritične delove, a sve su sinhronizovane sa istim objektom. Napravite više zadataka kako biste pokazali da se u svakom trenutku može izvršavati samo jedan od njih. Potom modifikujte metode tako da se sinhronizuju s različitim objektima i pokažite da se sve tri metode mogu izvršavati istovremeno.

Vežba 16: (1) Izmenite vežbu 15 tako da se upotrebljavaju eksplicitni **Lock** objekti.

Lokalno skladište niti

Drugi način da sprečite sudare zadataka zbog deljenih resursa jeste da izbegnete deljenje promenljivih. *Lokalno skladište niti* je mehanizam koji za svaku nit koja pristupa objektu automatski pravi novo skladište za istu promenljivu. Dakle, ako pet niti pristupa objektu koji ima promenljivu **x**, lokalno skladište niti generiše pet različitih skladišta za **x**. U sušti-ni, ta skladišta omogućuju da svakoj niti pridružite njeno stanje.

Pravljenjem i upravljanjem lokalnim skladištem niti bavi se klasa **java.lang.ThreadLocal**, kao što ćete videti u narednom primeru:

```
//: paralelno/ThreadLocalSpremnikPromenljivih.java
// Svaka nit automatski dobija svoje skladište.
import java.util.concurrent.*;
import java.util.*;

class Pristup implements Runnable {
  private final int id;
  public Pristup(int idn) { id = idn; }
  public void run() {
    while(!Thread.currentThread().isInterrupted()) {
      ThreadLocalSpremnikPromenljivih.povecajZaJedan();
      System.out.println(this);
      Thread.yield();
    }
  }
```

```
  public String toString() {
    return "#" + id + ": " +
      ThreadLocalSpremnikPromenljivih.get();
  }
}

public class ThreadLocalSpremnikPromenljivih {
  private static ThreadLocal<Integer> vrednost =
    new ThreadLocal<Integer>() {
      private Random slucajan = new Random(47);
      protected synchronized Integer initialValue() {
        return slucajan.nextInt(10000);
      }
    };
  public static void povecajZaJedan() {
    vrednost.set(vrednost.get() + 1);
  }
  public static int get() { return vrednost.get(); }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Pristup(i));
    TimeUnit.SECONDS.sleep(3);  // Radi neko vreme
    exec.shutdownNow();         // Završiće se svi Pristupi
  }
} /* Ispis: (primer)
#0: 9259
#1: 556
#2: 6694
#3: 1862
#4: 962
#0: 9260
#1: 557
#2: 6695
#3: 1863
#4: 963
...
*///:~
```

Objekti tipa **ThreadLocal** obično se skladište kao statička polja. Kada napravite objekat tipa **ThreadLocal**, sadržaju tog objekta možete pristupiti samo metodama **get( )** i **set( )**. Metoda **get( )** vraća kopiju objekta pridruženu toj niti, dok **set( )** umeće svoj argument u objekat uskladišten za tu nit i vraća stari objekat koji je bio u tom skladištu. To pokazuju metode **povecajZaJedan( )** i **get( )** programa **ThreadLocalSpremnikPromenljivih**. Obratite pažnju na to da metode **povecajZaJedan( )** i **get( )** nisu sinhronizovane, zato što **ThreadLocal** jemči da neće doći do uslova za trku.

Kada pokrenete ovaj program, videćete dokaz da se svakoj niti dodeljuje njeno sopstveno skladište, pošto svaka od njih održava sopstveni brojač iako postoji samo jedan objekat **ThreadLocalSpremnikPromenljivih**.

Gašenje zadataka

U nekima od prethodnih primera, metode **cancel( )** i **isCancelled( )** smeštene su u klasu vidljivu svim zadacima. Zadaci ispituju vrednost metode **isCancelled( )** da bi znali kada da se ugase. To je razborit pristup problemu. Međutim, u nekim situacijama zadatak mora biti ugašen naglije. U ovom odeljku saznaćete sve o takvom gašenju i problemima zbog njega.

Prvo, razmotrimo primer koji ne samo da pokazuje problem gašenja, nego je i dodatan primer deljenja resursa.

Ukrasna bašta

U ovoj simulaciji, baštenska komisija bi htela da zna koliko ljudi svakog dana uđe u baštu kroz neku od njenih kapija. Svaka kapija ima blokirajuću vrtešku ili neku drugu vrstu brojača, i nakon povećanja brojača blokirajuće vrteške za 1, povećava se i zajednički brojač koji predstavlja ukupan broj posetilaca bašte.

```
//: paralelno/UkrasnaBasta.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Broj {
  private int broj = 0;
  private Random slucajan = new Random(47);
  // Ako uklonite rezervisanu reč synchronized,
  // prebrojavanje će zakazati:
  public synchronized int povecajZaJedan() {
    int privr = broji;
    if(slucajan.nextBoolean()) // Pola vremena prepusti drugima
      Thread.yield();
    return (broj = ++privr);
  }
  public synchronized int value() { return broj; }
}

class Ulaz implements Runnable {
  private static Broj broj = new Broj();
  private static List<Ulaz> ulazi =
    new ArrayList<Ulaz>();
  private int broj = 0;
  // Sinhronizacija nije potrebna za čitanje:
  private final int id;
  private static volatile boolean cancelled = false;
  // Atomska operacija na volatile polju:
  public static void cancel() { cancelled = true; }
  public Ulaz(int id) {
    this.id = id;
```

```
    // Drži ovaj zadatak u listi. Takođe sprečava
    // da mrtvi zadaci budu pokupljeni kao smeće:
    ulazi.add(this);
  }
  public void run() {
    while(!cancelled) {
      synchronized(this) {
        ++broj;
      }
      print(this + " Ukupno: " + broj.povecajZaJedan());
      try {
        TimeUnit.MILLISECONDS.sleep(100);
      } catch(InterruptedException e) {
        print("spavanje prekinuto");
      }
    }
    print("Zaustavljam " + this);
  }
  public synchronized int dajVrednost() { return broj; }
  public String toString() {
    return "Ulaz " + id + ": " + dajVrednost();
  }
  public static int dajUkupanBroj () {
    return broj.value();
  }
  public static int saberiUlaze() {
    int zbir = 0;
    for(Ulaz entrance : ulazi)
      zbir += entrance.dajVrednost();
    return zbir;
  }
}

public class UkrasnaBasta {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Ulaz(i));
    // Radi neko vreme, zatim stani i prikupi podatke:
    TimeUnit.SECONDS.sleep(3);
    Ulaz.cancel();
    exec.shutdown();
    if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
      print("Neki zadaci nisu ugašeni!");
    print("Ukupno: " + Ulaz.dajUkupanBroj ());
    print("Zbir ulaza: " + Ulaz.saberiUlaze());
  }
} /* Ispis: (primer)
Ulaz 0: 1 Ukupno: 1
Ulaz 2: 1 Ukupno: 3
```

```
Ulaz 1: 1 Ukupno: 2
Ulaz 4: 1 Ukupno: 5
Ulaz 3: 1 Ukupno: 4
Ulaz 2: 2 Ukupno: 6
Ulaz 4: 2 Ukupno: 7
Ulaz 0: 2 Ukupno: 8
...
Ulaz 3: 29 Ukupno: 143
Ulaz 0: 29 Ukupno: 144
Ulaz 4: 29 Ukupno: 145
Ulaz 2: 30 Ukupno: 147
Ulaz 1: 30 Ukupno: 146
Ulaz 0: 30 Ukupno: 149
Ulaz 3: 30 Ukupno: 148
Ulaz 4: 30 Ukupno: 150
Zaustavljam Ulaz 2: 30
Zaustavljam Ulaz 1: 30
Zaustavljam Ulaz 0: 30
Zaustavljam Ulaz 3: 30
Zaustavljam Ulaz 4: 30
Ukupno: 150
Zbir ulaza: 150
*///:~
```

Jedan objekat tipa **Broj** pamti ukupan broj posetilaca bašte i čuva se kao statičko polje klase **Ulaz**. Metode **Broj.povecajZaJedan( )** i **Broj.value( )** sinhronizovane su da bi se kontrolisalo pristupanje polju **broj**. Metoda **povecajZaJedan( )** upotrebljava objekat tipa **Random** da bi metodom **yield( )** drugim zadacima prepustila približno pola vremena, između upisivanja **broja** u **privr** i povećanja za 1 i upisivanja promenljive **privr** nazad u **broj**. Ako rezervisanu reč **synchronized** izbacite iz definicije metode **povecajZaJedan( )**, program otkazuje zato što više zadataka istovremeno pristupa i menja **broj**. (Poziv metode **yield( )** to samo ubrzava.)

Svaki (zadatak) **Ulaz** čuva svoj lokalni **broj**; on pokazuje koliko je posetilaca ušlo kroz taj ulaz. Time se može kontrolisati da li objekat **broj** pamti tačan broj posetilaca. **Ulaz.run( )** povećava **broj** i objekat **broj** i zatim spava 100 milisekundi.

Pošto je **Ulaz.cancelled** indikator tipa **boolean** s modifikatorom **volatile** nad kojim se obavljaju samo operacije čitanja i dodeljivanja (i koji se nikada ne čita u kombinaciji s drugim poljima), možemo se izvući a da ga ne sinhronizujemo. Ukoliko ste iole nesigurni u takvoj situaciji, bolje upotrebite **synchronized**.

Ovaj program se prilično mnogo trudi da sve pogasi na stabilan način. Tako sam ga napisao da bih vam pokazao koliko pažljivo morate gasiti višenitni program i to koliko je vredna metoda **interrupt( )** o kojoj će uskoro biti više reči.

Nakon 3 sekunde, **main( )** šalje statičku poruku **cancel( )** na **Ulaz**, zatim poziva **shutdown( )** za objekat **exec** (izvršilac), a onda poziva metodu **awaitTermination( )** za taj **exec**. **ExecutorService.awaitTermination( )** čeka da se svi zadaci završe i vraća **true** ako se svi završe pre isteka zadatog vremena čekanja; u protivnom vraća **false**, što pokazuje da svi zadaci nisu završeni. Iako ovo prouzrokuje da svi zadaci izadu iz svojih metoda **run( )**

i stoga završe svoje izvršavanje, objekti tipa **Ulaz** i dalje su validni zato što je u konstruktoru svaki objekat **Ulaz** uskladišten u statičkoj listi **ulazi** tipa **List<Ulaz>**. Zato **saberiUlaze( )** i dalje radi s validnim objektima **Ulaz**.

Dok ovaj program bude radio, prikazivaće ukupan broj posetilaca i broj posetilaca na svakom ulazu koji prođu kroz blokirajuću vrtešku. Ako uklonite deklaraciju **synchronized** ispred metode **Broj.povecajZaJedan( )**, videćete da ukupan broj posetilaca nije jednak očekivanom. Broj posetilaca koji se prebroji na svim blokirajućim vrteškama neće biti jednak iznosu promenljive **broj**. Sve ispravno radi dok postoji uzajamno isključiva brava (mutex) koja sinhronizuje pristupanje objektu **Broj**. Imajte u vidu da metoda **Broj.povecajZaJedan( )** pomoću promenljive **privr** i metode **yield( )** namerno povećava mogućnost otkazivanja. U pravim višenitnim problemima, mogućnost otkazivanja može biti statistički mala, pa ćete lakše poverovati u zabludu da vaš program radi ispravno. Kao u gornjem primeru, verovatno će postojati i skriveni problemi kojih niste svesni, pa budite izuzetno pažljivi kada pregledate programe za paralelno izvršavanje.

Vežba 17: (2) Napravite Gajgerov brojač koji može imati proizvoljan broj daljinskih senzora.

Gašenje blokiranih zadataka

U prethodnom primeru, metoda **Ulaz.run( )** u svojoj petlji obuhvata i poziv metode **sleep( )**. Znamo da će se metoda **sleep( )** kad-tad probuditi i zadatak će doći na vrh petlje, gde nakon provere indikatora **cancelled** može da izađe iz nje. Međutim, spavanje prouzrokovano metodom **sleep( )** samo je jedna od situacija u kojima je izvršavanje zadatka blokirano, a ponekad se i zadatak koji je blokiran mora ugasiti.

Stanja niti

Nit može biti u jednom od četiri stanja:

1. *Nova*: Nit je u ovom stanju privremeno, dok je prave. Ona dodeljuje potrebne sistemske resurse i obavlja inicijalizaciju. Od tog trenutka postaje podobna za primanje procesorskog vremena. Potom mehanizam za raspodelu vremena menja stanje niti u spremna za izvršavanje ili blokirana.
2. *Spremna za izvršavanje*: Znači da se nit može pokrenuti kada mehanizam podele procesorskog vremena bude imao slobodnih procesorskih ciklusa za nju. Možda se ta nit izvršava, a možda i ne, ali ništa ne može sprečiti njeno izvršavanje ukoliko joj mehanizam za raspodelu vremena dodeli procesor. Drugim rečima, nit nije ni mrtva ni blokirana.
3. *Blokirana*: Nit bi mogla da se izvršava, ali je nešto sprečava. Dok je nit blokirana, mehanizam za raspodelu vremena je preskače i ne dodeljuje joj procesor. Sve dok ta nit ponovo ne postane spremna za izvršavanje, neće biti pokretana.
4. *Mrtva*: Mrtva ili *ugašena* (engl. *terminated*) nit više nije podobna za raspoređivanje i procesorsko vreme joj se ne dodeljuje. Njen zadatak je završen i ona se više ne može pokrenuti. Zadatak normalno umire kada se vrati iz svoje metode **run( )**, ali i njegova nit može biti prekinuta, kao što ćete uskoro videti.

Prelazak u stanje blokiranosti

Zadatak može postati blokiran iz jednog od sledećih razloga:

- Uspavali ste taj zadatak pozivom metode **sleep(milisekundi)** i on se neće izvršavati tokom zadatog vremena.
- Zaustavili ste izvršavanje niti metodom **wait( )**. Nit neće postati spremna za izvršavanje sve dok ne dobije poruku **notify( )** ili **notifyAll( )** (ili ekvivalentnu poruku **signal( )** odnosno **signalAll( )** za alatke iz Java SE5 biblioteke **java.util.concurrent**). Objasniću to u jednom od kasnijih odeljaka.
- Nit čeka da se završi neka ulazno/izlazna operacija.
- Nit pokušava da pozove neku sinhronizovanu metodu drugog objekta koji je zaključan.

U starim programima još se mogu videti pozivi metoda **suspend( )** i **resume( )** za blokiranje odnosno deblokiranje niti, ali su one zastarele u savremenoj Javi (pošto su umele da izazovu uzajamnu blokadu), pa ih u ovoj knjizi neću opisivati. Zastarela je i metoda **stop( )** zato što ne otključava brave koje je nit zaključala, pa ako su objekti u nedoslednom stanju („oštećeni"), ostali zadaci ih u tom stanju mogu videti i modifikovati. Iz toga se mogu izroditi suptilni problemi koji se teško otkrivaju.

Razmotrićemo sada ovaj problem: ponekad treba ugasiti zadatak koji je blokiran. Ako ne možete sačekati da on dođe do mesta u kodu gde može da proveri neko stanje i sam odluči da se ugasi, morate ga primorati da izađe iz stanja blokiranosti.

Prekidanje izvršavanja

Kao što možete zamisliti, mnogo je teže izaći iz metode **Runnable.run( )** pre njenog kraja nego sačekati da ona dospe do mesta gde ispituje vrednost indikatora „otkaži" ili do nekog drugog mesta gde je programer spreman da napusti tu metodu. Kada izađete iz blokiranog zadatka, često treba da počistite resurse. Zato izlazak iz metode **run( )** datog zadatka pre njenog kraja, najviše liči na bacanje izuzetka, pa se u Javinim nitima za tu vrstu prekidanja koriste izuzeci.[16] (Ovo je na samoj ivici nekorektne upotrebe izuzetaka, pošto znači da ih često upotrebljavate za kontrolu toka programa.) Da biste se prilikom ovakvog gašenja zadatka vratili na neko od poznatih dobrih stanja, morate pažljivo razmotriti putanje izvršavanja koda i napisati odredbu **catch** tako da pravilno sve počisti.

Klasa **Thread** sadrži metodu **interrupt( )** koja služi za gašenje blokiranih zadataka. Ona niti daje status prekinute. Nit kojoj je postavljen status prekinuta baca izuzetak **InterruptedException** ukoliko je već blokirana ili pokušava da izvrši operaciju koja prouzrokuje blokiranje. Status prekinuta biće resetovan kada se baci izuzetak ili kada zadatak pozove metodu **Thread.interrupted( )**. Kao što ćete videti, metoda **Thread.interrupted( )** pruža drugi način izlaska iz petlje **run( )**, bez bacanja izuzetka.

---

[16] Međutim, izuzeci se nikada ne generišu asinhrono. Stoga nema opasnosti da nešto prekine instrukciju/poziv metode. Ukoliko prilikom upotrebe mutexa (za razliku od rezervisane reči **synchronized**) budete upotrebljavali idiom **try-finally**, te uzajamno isključive brave (mutexi) biće automatski otključane kada se baci izuzetak.

Da biste pozvali **interrupt( )**, morate imati nit, tj. objekat tipa **Thread**. Možda ste uočili da nova biblioteka **concurrent** (za paralelno izvršavanje) kao da izbegava neposredan rad s nitima; ona sve pokušava da ostvari izvršiocima. Ako pozovete **shutdownNow( )** za nekog izvršioca, on će poslati poziv **interrupt( )** svim nitima koje je pokrenuo. To ima smisla, pošto najčešće hoćemo odjednom da ugasimo sve zadatke određenog izvršioca, kada završimo deo projekta ili ceo program. Međutim, ima slučajeva kada bismo da ugasimo samo jedan zadatak. Ako radite sa izvršiocem, kontekst zadatka koji ste pokrenuli možete da uhvatite tako što ćete pozvati metodu **submit( )** umesto metode **execute( )**. **submit( )** vraća generički objekat **Future<?>** s nespecificiranim parametrom, zato što za taj objekat nikada nećete zvati **get( )** – svrha posedovanja ovakvog objekta tipa **Future** jeste da za njega možete pozvati **cancel( )** i tako ga upotrebiti za prekidanje određenog zadatka. Ukoliko metodi **cancel( )** prosledite **true**, ona ima dozvolu da pozove **interrupt( )** za tu nit da bi je prekinula; dakle, metoda **cancel( )** predstavlja jedan od načina prekidanja niti koju je pokrenuo izvršilac.

U narednom primeru videćete osnove upotrebe metode **interrupt( )** u izvršiocu:

```
//: paralelno/Prekidam.java
// Prekidam blokirane niti.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class BlokiranoSpavanjem implements Runnable {
  public void run() {
    try {
      TimeUnit.SECONDS.sleep(100);
    } catch(InterruptedException e) {
      print("InterruptedException");
    }
    print("Izlazim iz niti BlokiranoSpavanjem.run()");
  }
}

class BlokiranoUIOperacijom implements Runnable {
  private UlazniTok ul;
  public BlokiranoUIOperacijom(UlazniTok ut) { ul = ut; }
  public void run() {
    try {
      print("Čekamo na read():");
      ul.read();
    } catch(IOException e) {
      if(Thread.currentThread().isInterrupted()) {
        print("Prekinuto iz blokirane U/I operacije");
      } else {
        throw new RuntimeException(e);
      }
    }
```

```
      print("Izlazim iz niti BlokiranoUIOperacijom.run()");
    }
  }

  class SinhronizovanoBlokiranje implements Runnable {
    public synchronized void f() {
      while(true) // Nikada ne otključava bravu
        Thread.yield();
    }
    public SinhronizovanoBlokiranje() {
      new Thread() {
        public void run() {
          f(); // Ova nit zaključava bravu
        }
      }.start();
    }
    public void run() {
      print("Pokušavam da pozovem f()");
      f();
      print("Izlazim iz niti SinhronizovanoBlokiranje.run()");
    }
  }

  public class Prekidam {
    private static ExecutorService exec =
      Executors.newCachedThreadPool();
    static void test(Runnable r) throws InterruptedException{
      Future<?> f = exec.submit(r);
      TimeUnit.MILLISECONDS.sleep(100);
      print("Prekidam " + r.getClass().getName());
      f.cancel(true); // Prekida u slučaju izvršavanja
      print("Prekid poslat objektu " + r.getClass().getName());
    }
    public static void main(String[] args) throws Exception {
      test(new BlokiranoSpavanjem());
      test(new BlokiranoUIOperacijom(System.ul));
      test(new SinhronizovanoBlokiranje());
      TimeUnit.SECONDS.sleep(3);
      print("Prekidam uz System.exit(0)");
      System.exit(0); // ... pošto su poslednja 2 prekida zakazala
    }
  } /* Ispis: (95% podudaranja)
  Prekidam BlokiranoSpavanjem
  InterruptedException
  Izlazim iz niti BlokiranoSpavanjem.run()
  Prekid poslat objektu BlokiranoSpavanjem
  Čekamo na read():
  Prekidam BlokiranoUIOperacijom
  Prekid poslat objektu BlokiranoUIOperacijom
```

```
Pokušavam da pozovem f()
Prekidam SinhronizovanoBlokiranje
Prekid poslat objektu SinhronizovanoBlokiranje
Prekidam uz System.exit(0)
*///:~
```

Svaki zadatak predstavlja različitu vrstu blokiranja. **BlokiranoSpavanjem** je primer blokiranja koje se može prekinuti, dok **BlokiranoUIOperacijom** i **SinhronizovanoBlokiranje** nije moguće prekinuti.[17] Prethodni program dokazuje da se ulazno/izlazne operacije i čekanje na sinhronizovanu bravu ne mogu prekinuti, ali ste to mogli predvideti i nakon pregledanja koda, pošto za ulazno/izlazne operacije i čekanje na sinhronizovane metode nije potreban blok za obradu izuzetka **InterruptedException**.

Prve dve klase su jednostavne: metoda **run( )** poziva **sleep( )** u prvoj klasi i **read( )** u drugoj. Međutim, da bismo pokazali **SinhronizovanoBlokiranje**, najpre moramo zaključati bravu. To postižemo u konstruktoru, tako što se napravi instanca anonimne klase **Thread** koja zaključava objekat pozivom metode **f( )** (nit mora biti neka druga a ne ona koja izvršava **run( )** za **SinhronizovanoBlokiranje**, zato što jedna nit može više puta zaključati objekat). Pošto se **f( )** nikada ne vraća, ta brava se nikada ne otključava. **SinhronizovanoBlokiranje.run( )** pokušava da pozove **f( )** i blokirano je dok čeka da brava bude otključana.

Iz rezultata vidite da poziv metode **sleep( )** možete prekinuti (kao i svaki drugi poziv koji zahteva hvatanje izuzetka **InterruptedException**). Međutim, ne možete prekinuti zadatak koji pokušava da zaključa sinhronizovanu bravu, niti objekat koji pokušava da obavi ulazno/izlaznu operaciju. To je pomalo uznemirujuće, naročito ako pišete zadatak koji treba da obavlja ulazno/izlazne operacije, pošto to znači da one mogu da zaključaju višenitni program. To je razlog za brigu, pogotovo ako se radi o programima za Web.

Dosta složeno, ali ponekad efikasno rešenje ovog problema jeste zatvaranje pripadnog resursa na kojem je zadatak blokiran:

```
//: paralelno/ZatvoriResurs.java
// Prekidam blokirani zadatak zatvaranjem pripadnog resursa.
// {RunByHand}
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class ZatvoriResurs {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    ServerSocket server = new ServerSocket(8080);
    UlazniTok ulazIzUticnice =
      new Socket("localhost", 8080).getInputStream();
    exec.execute(new BlokiranoUIOperacijom(ulazIzUticnice));
```

[17] Neka izdanja JDK podržavaju i **InterruptedIOException**. Međutim, taj izuzetak je tek delimično realizovan, i to samo na nekim platformama. Bacanje tog izuzetka prouzrokuje da U/I objekat postane neupotrebljiv. Malo je verovatno da će buduća izdanja podržavati taj izuzetak.

```
    exec.execute(new BlokiranoUIOperacijom(System.in));
    TimeUnit.MILLISECONDS.sleep(100);
    print("Gasim sve niti");
    exec.shutdownNow();
    TimeUnit.SECONDS.sleep(1);
    print("Zatvaram " + ulazIzUticnice.getClass().getName());
    ulazIzUticnice.close(); // Oslobađa blokiranu nit
    TimeUnit.SECONDS.sleep(1);
    print("Zatvaram " + System.in.getClass().getName());
    System.in.close(); // Oslobađa blokiranu nit
  }
} /* Ispis: (85% podudaranja)
Čekamo na read():
Čekamo na read():
Gasim sve niti
Zatvaram java.net.SocketInputStream
Prekinuto iz blokirane U/I operacije
Izlazim iz niti BlokiranoUIOperacijom.run()
Zatvaram java.io.BufferedInputStream
Izlazim iz niti BlokiranoUIOperacijom.run()
*///:~
```

Nakon poziva metode **shutdownNow( )**, odlaganjima pre poziva metode **close( )** za dva ulazna toka naglašava se da se zadaci debłokiraju posle zatvaranja odgovarajućih resursa. Možda ste primetili da se **interrupt( )** pojavljuje kada zatvaramo **Socket**, ali ne i kada zatvaramo **System.in**.

Srećom, civilizovaniji prekid ulazno/izlaznih operacija omogućuju **nio** klase predstavljene u poglavlju o ulazno/izlaznim operacijama. Blokirani **nio** kanali automatski reaguju na prekide:

```
//: paralelno/NIOPrekid.java
// Prekidam blokiran NIO kanal.
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlokiran implements Runnable {
  private final KanalUticnice ku;
  public NIOBlokiran(KanalUticnice ku) { this.ku = ku; }
  public void run() {
    try {
      print("Čekamo na read() u " + this);
      ku.read(ByteBuffer.allocate(1));
    } catch(ClosedByInterruptException e) {
      print("ClosedByInterruptException");
    } catch(AsynchronousCloseException e) {
      print("AsynchronousCloseException");
```

```
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
      print("Izlazim iz niti NIOBlokiran.run() " + this);
    }
  }

  public class NIOPrekid {
    public static void main(String[] args) throws Exception {
      ExecutorService exec = Executors.newCachedThreadPool();
      ServerSocket server = new ServerSocket(8080);
      InetSocketAddress isa =
        new InetSocketAddress("localhost", 8080);
      KanalUticnice ku1 = KanalUticnice.open(isa);
      KanalUticnice ku2 = KanalUticnice.open(isa);
      Future<?> f = exec.submit(new NIOBlokiran(ku1));
      exec.execute(new NIOBlokiran(ku2));
      exec.shutdown();
      TimeUnit.SECONDS.sleep(1);
      // Napravi prekid otkazivanjem operacije:
      f.cancel(true);
      TimeUnit.SECONDS.sleep(1);
      // Deblokiraj zatvaranjem kanala:
    }    ku2.close();
  } /* Ispis: (primer)
  Čekamo na read() u NIOBlokiran@7a84e4
  Čekamo na read() u NIOBlokiran@15c7850
  ClosedByInterruptException
  Izlazim iz niti NIOBlokiran.run() NIOBlokiran@15c7850
  AsynchronousCloseException
  Izlazim iz niti NIOBlokiran.run() NIOBlokiran@7a84e4
  *///:~
```

Kao što vidite, deblokiranje možete izvršiti i zatvaranjem pripadnog kanala, mada bi to retko trebalo da bude neophodno. Vodite računa o tome da biste pokretanjem oba zadatka metodom **execute**( ) i pozivom metode **e.shutdownNow**( ) lako sve pogasili; hvatanje objekta tipa **Future** u gornjem primeru bilo je potrebno samo da bi se izuzetak poslao jednoj niti, a ne i onoj drugoj.[18]

Vežba 18: (2) Napravite klasu bez zadataka, ali s metodom koja poziva **sleep**( ) za dugačko razdoblje. Napravite zadatak koji poziva tu metodu u klasi bez zadataka. U metodi **main**( ) pokrenite zadatak, a zatim pozovite **interrupt**( ) da biste ga ugasili. Treba da postignete da se zadatak ugasi bezbedno.

Vežba 19: (4) Izmenite program **UkrasnaBasta.java** tako da upotrebite metodu **interrupt**( ).

Vežba 20: (1) Izmenite program **CachedThreadPool.java** tako da svi zadaci prime **interrupt**( ) pre nego što budu završeni.

[18] Pri istraživanju grade za ovaj odeljak pomogao mi je Ervin Varga.

Blokiranost koju izaziva uzajamno isključiva brava (mutex)

Kao što ste videli u programu **Prekidanje.java**, ako pokušate da pozovete sinhronizovanu metodu za zaključani objekat, pozivajući zadatak će biti zaustavljen (blokiran) dok se objekat ne otključa. U sledećem primeru videćete kako jedan zadatak može više puta zaključati istu uzajamno isključivu bravu (mutex):

```
//: paralelno/VisePutaZakljucanaBrava.java
// Nit može više puta da zaključa istu bravu.
import static net.mindview.util.Print.*;

public class VisePutaZakljucanaBrava {
  public synchronized void f1(int broj) {
    if(broj-- > 0) {
      print("f1() poziva f2() uz broj " + broj);
      f2(broj);
    }
  }
  public synchronized void f2(int broj) {
    if(broj-- > 0) {
      print("f2() poziva f1() uz broj " + broj);
      f1(broj);
    }
  }
  public static void main(String[] args) throws Exception {
    final VisePutaZakljucanaBrava visePutaZakljucanaBrava
    = new VisePutaZakljucanaBrava();
    new Thread() {
      public void run() {
        visePutaZakljucanaBrava.f1(10);
      }
    }.start();
  }
} /* Ispis:
f1() poziva f2() uz broj 9
f2() poziva f1() uz broj 8
f1() poziva f2() uz broj 7
f2() poziva f1() uz broj 6
f1() poziva f2() uz broj 5
f2() poziva f1() uz broj 4
f1() poziva f2() uz broj 3
f2() poziva f1() uz broj 2
f1() poziva f2() uz broj 1
f2() poziva f1() uz broj 0
*///:~
```

U metodi **main( )** se pravi nit koja poziva **f1( )**, a zatim se **f1( )** i **f2( )** uzajamno pozivaju dok **broj** ne postane jednak nuli. Pošto je zadatak već zaključao bravu objekta **visePutaZakljucanaBrava** unutar prvog poziva metode **f1( )**, isti zadatak je ponovo zaključava u

pozivu metode **f2( )** itd. To ima smisla zato što bi zadatak trebalo da može da poziva sinhronizovane metode unutar istog objekta; taj zadatak je već zaključao objekat (bravu).

Kao što je već bilo rečeno za ulazno/izlazne operacije koje nije moguće prekinuti, kad god zadatak može biti tako blokiran da ga se ne može prekinuti, postoji mogućnost da će to zaključati ceo program. U biblioteke za paralelno izvršavanje Jave SE5 dodata je i mogućnost prekidanja zadataka blokiranih na bravama (objektima) tipa **ReentrantLock**, za razliku od zadataka blokiranih na sinhronizovanim metodama ili kritičnim delovima:

```
//: paralelno/Prekidam2.java
// Prekidam zadatak blokiran pomoću objekta tipa ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlokiranMutex {
  private Lock brava = new ReentrantLock();
  public BlokiranMutex() {
    // Odmah ćemo ga zaključati, da bismo pokazali prekidanje
    // zadatka blokiranog na bravi ReentrantLock:
    brava.lock();
  }
  public void f() {
    try {
      // Drugim zadacima ovo nikada neće biti dostupno
      brava.lockInterruptibly(); // Poseban poziv
      print("zaključana brava u f()");
    } catch(InterruptedException e) {
      print("Prekinuto u pokušaju zaključavanja brave u f()");
    }
  }
}

class Blokirana2 implements Runnable {
  BlokiranMutex blokiran = new BlokiranMutex();
  public void run() {
    print("Čekamo na f() u klasi BlokiranMutex");
    blokiran.f();
    print("Izašao iz blokiranog poziva");
  }
}

public class Prekidam2 {
  public static void main(String[] args) throws Exception {
    Thread n = new Thread(new Blokirana2());
    n.start();
    TimeUnit.SECONDS.sleep(1);
    System.out.println("Pozivam n.interrupt()");
    n.interrupt();
  }
```

```
} /* Ispis:
Čekamo na f() u klasi BlokiranMutex
Pozivam n.interrupt()
Prekinuto u pokušaju zaključavanja brave u f()
Izašao iz blokiranog poziva
*///:~
```

Klasa **BlokiranMutex** ima konstruktor koji zaključa bravu (objekat tipa **Lock**) samog objekta i nikada je ne otključa. Zbog toga ćete, ukoliko pokušate da pozovete **f( )** iz nekog drugog zadatka (različitog od onoga koji je napravio objekat **BlokiranMutex**), uvek biti blokirani pošto je ta uzajamno isključiva brava stalno zaključana. U klasi **Blokirana2**, metodu **run( )** zaustaviće poziv metode **blokirana.f( )**. Kada pokrenete program, videćete da, za razliku od poziva ulazno/izlaznih operacija, **interrupt( )** može da izađe iz poziva koji je blokirala uzajamno isključiva brava (mutex).[19]

Provera postoji li prekid

Kada za neku nit pozovete prekid metodom **interrupt( )**, prekid će nastati jedino u slučaju da zadatak uđe u operaciju koja blokira ili je već unutar nje (sem, kao što ste videli, u slučaju ulazno/izlaznih operacija koje nije moguće prekinuti, ili blokiranih sinhronizovanih metoda – tada ste bespomoćni). Ali šta ako kod takav blokirajući poziv pravi samo u zavisnosti od uslova u kojima se izvršava? Ukoliko možete da izađete samo tako što ćete baciti izuzetak na blokirajući poziv, nećete uvek moći da izađete iz petlje metode **run( )**. Dakle, ako pozovete **interrupt( )** da zaustavi neki zadatak, on mora imati još neki način da izađe, za slučaj da petlja metode **run( )** ne napravi nijedan blokirajući poziv.

Tu mogućnost pruža *status prekinutosti* (engl. *interrupted status*) koji se postavlja pozivom metode **interrupt( )**. Status prekinutosti ispitujete metodom **interrupted( )**. Ona vam kazuje ne samo da li je metoda **interrupt( )** bila pozvana, nego i briše status prekinutosti. Kada se izbriše status prekinutosti, obezbeđuje se da osnovna struktura (engl. *framework*) ne obaveštava dvaput o tome da je neki zadatak prekinut. Bićete obavešteni preko izuzetka **InterruptedException** ili poziva metode **Thread.interrupted( )** koji vraća **true**. Ukoliko želite ponovo da proverite da li je nastao prekid, sačuvajte rezultat poziva metode **Thread.interrupted( )**.

U narednom primeru prikazan je tipičan idiom. Ukoliko je status prekinutosti postavljen, koristite taj idiom za izlazak iz metode **run( )** i u blokiranom i u neblokiranom stanju:

```
//: paralelno/IdiomPrekidanja.java
// Opšti idiom za prekidanje zadatka.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class TrebaPocistiti {
  private final int id;
```

[19] Iako je to malo verovatno, imajte u vidu da bi se poziv metode **t.interrupt( )** zapravo mogao desiti pre poziva metode **blokirana.f( )**.

```
  public TrebaPocistiti(int ident) {
    id = ident;
    print("TrebaPocistiti " + id);
  }
  public void cleanup() {
    print("Čistim " + id);
  }
}

class Blokirano3 implements Runnable {
  private volatile double d = 0.0;
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // tačka1
        TrebaPocistiti n1 = new TrebaPocistiti(1);
        // Blok try-finally počnite neposredno iza definicije n1,
        // da bi bilo zajemčeno pravilno čišćenje n1:
        try {
          print("Spavam");
          TimeUnit.SECONDS.sleep(1);
          // tačka2
          TrebaPocistiti n2 = new TrebaPocistiti(2);
          // Zajemčeno pravilno čišćenje n2:
          try {
            print("Računam");
            // Dugotrajna operacija koja ne blokira:
            for(int i = 1; i < 2500000; i++)
              d = d + (Math.PI + Math.E) / d;
            print("Završena dugotrajna operacija ");
          } finally {
            n2.cleanup();
          }
        } finally {
          n1.cleanup();
        }
      }
      print("Izlazim iz niti ispitivanjem uslova od while()");
    } catch(InterruptedException e) {
      print("Izlazim iz niti pomoću izuzetka InterruptedException");
    }
  }
}

public class IdiomPrekidanja {
  public static void main(String[] args) throws Exception {
    if(args.length != 1) {
      print("pokretanje: java IdiomPrekidanja kašnjenje_u_ms");
      System.exit(1);
    }
```

```
    Thread n = new Thread(new Blokirano3());
    n.start();
    TimeUnit.MILLISECONDS.sleep(new Integer(args));
    n.interrupt();
  }
} /* Ispis: (primer)
TrebaPocistiti 1
Spavam
TrebaPocistiti 2
Računam
Završena dugotrajna operacija
Čistim 2
Čistim 1
TrebaPocistiti 1
Spavam
Čistim 1
Izlazim iz niti pomoću izuzetka InterruptedException
*///:~
```

Klasa **TrebaPocistiti** naglašava potrebu za pravilnim čišćenjem resursa ukoliko petlju napustite putem izuzetka. Vodite računa o tome da sve resurse klase **TrebaPocistiti** napravljene u metodi **Blokirano3.run( )** moraju neposredno da slede odredbe **try-finally** koje jemče da će metoda **cleanup( )** uvek biti pozvana.

Na komandnoj liniji, programu morate dati argument – vreme kašnjenja u milisekundama – pre nego što pozove **interrupt( )**. Korišćenjem različitih kašnjenja možete izaći iz metode **Blokirano3.run( )** na različitim tačkama petlje: u pozivu metode **sleep( )** koja blokira ili u matematičkom proračunu koji ne blokira. Ukoliko **interrupt( )** bude pozvana nakon komentara „tačka2" (tokom operacije koja ne blokira), videćete da se prvo dovršava petlja, zatim se uništavaju svi lokalni objekti i najzad se izlazi iz petlje preko naredbe **while** na njenom vrhu. Međutim, ako se **interrupt( )** pozove između „tačke1" i „tačke2" (nakon naredbe **while**, ali pre ili u toku operacije **sleep( )** koja blokira), zadatak izlazi pomoću izuzetka **InterruptedException** čim se prvi put pokuša operacija koja blokira. U tom slučaju, čiste se samo objekti klase **TrebaPocistiti** napravljeni do trenutka bacanja izuzetka, a sve ostalo čišćenje možete obaviti unutar odredbe **catch**.

Klasa koja odgovara na **interrupt( )** mora da ustanovi strategiju kojom obezbeđuje svoj ostanak u konsistentnom (neoštećenom) stanju. Po pravilu, to znači da iza pravljenja svih objekata koji zahtevaju čišćenje moraju slediti odredbe **try-finally**, da bi se čišćenje obavilo bez obzira na način izlaska iz petlje **run( )**. Takav kôd može dobro da radi, ali nažalost – zato što u Javi nema automatskih poziva destruktora – zavisi od toga da li će programer klijent dobro napisati odredbe **try-finally**.

Međusobna saradnja zadataka

Kao što ste videli, kada pomoću niti istovremeno izvršavate više zadataka, sprečićete jedan zadatak da pokvari resurse drugog ako bravom (mutexom) sinhronizujete njihovo ponašanje. Drugim rečima, ako se dva zadatka sudaraju zbog deljenog resursa (najčešće memorije), za pristupanje tom resursu upotrebite uzajamno isključivu bravu (mutex).

Kada je taj problem rešen, sledeći korak je naučiti kako da zadaci sarađuju jedan s drugim, da bi više zadataka moglo zajednički da reši problem. Sada se ne postavlja pitanje kako da se zadaci međusobno ne ometaju, nego kako da rade usaglašeno, pošto se takvi problemi moraju rešavati postepeno, utvrđenim redosledom. Kao u građevinarstvu: najpre se mora iskopati rupa za temelje, ali se armatura može položiti paralelno sa pravljenjem betonskih blokova, i oba ta zadatka moraju biti dovršena da bi se nalili temelji. Cevi moraju biti položene pre nalivanja betonske ploče, ploča mora biti nalivena pre nego što počne podizanje zidova itd. Neki od ovih zadataka mogu se izvršavati paralelno, ali određeni koraci ne mogu se izvesti ukoliko svi zadaci nisu završeni.

U međusobnoj saradnji zadataka glavno pitanje je primopredaja signala. Za tu primopredaju upotrebljavamo zajednički temelj: uzajamno isključivu bravu (mutex) koji u ovom slučaju jemči da će na određeni signal odgovoriti samo jedan zadatak. Time se izbegavaju svi eventualni uslovi za trku. Pored mutexa, dodaćemo način da se zadatak zaustavi dok se ne promeni neko spoljno stanje (npr. „Cevi su položene"), koje pokazuje da je vreme da se taj zadatak pokrene. U ovom odeljku, razmotrićemo primopredaju signala između zadataka, koju bezbedno realizuju metode **wait( )** i **notifyAll( )** klase **Object**. Biblioteka za paralelno izvršavanje Jave SE5 ima i klasu **Condition**, s metodama **await( )** i **signal( )**. Videćemo koji problemi mogu nastati i kako se rešavaju.

wait() i notifyAll()

Metoda **wait( )** se obično koristi dok čekate na promenu uslova čije ispunjenje određuju sile kojima ne upravlja tekuća metoda. Često neki drugi zadatak menja taj uslov. Ne želite da program stoji besposleno u niti, stalno ispitujući taj uslov; to se naziva zauzetost čekanjem (engl. *busy waiting*) i obično predstavlja loš način korišćenja procesorskog vremena. Stoga **wait( )** omogućava da uspavate tu nit dok čeka da se svet promeni. Ona se budi i ispituje promene tek kad naiđe na metode **notify( )** ili **notifyAll( )** koje nagoveštavaju da se možda desilo nešto važno. Tako je dobijen još jedan način sinhronizacije niti.

Važno je shvatiti da ni **sleep( )** niti **yield( )** *ne otključavaju* objekat kada ih pozovete. S druge strane, kada zadatak iz neke metode pozove metodu **wait( )**, ona otključava objekat pre nego što zaustavi izvršavanje niti. Pošto **wait( )** otključava bravu čim je pozovete, to znači da drugi zadaci mogu da pribave tu bravu, pa tokom izvršavanja metode **wait( )** možete da pozivate druge sinhronizovane metode tog (sada otključanog) objekta. To je neophodno, pošto obično baš te druge metode prouzrokuju promenu koju zaustavljena nit čeka da bi se probudila. Dakle, kada pozovete **wait( )**, saopštavate: „Zasad sam uradio sve što sam mogao, pa ću na ovom mestu da pričekam, ali hoću da omogućim drugim sinhronizovanim operacijama da se izvršavaju ako mogu."

Postoje dva oblika metode **wait( )**. Prvi prima argument u milisekundama, čije je značenje jednako onom u metodi **sleep( )**, dakle zaustavlja rad u zadatom periodu vremena. Za razliku od primene metode **sleep( )**, kada se koristi **wait(pauza)**, dešava se sledeće:

1. Otključava se objekat tokom čekanja izvršavanja metode **wait( )**.
2. Iz metode **wait( )** možete da izađete i zbog obaveštenja koje šalju metode **notify( )** ili **notifyAll( )**, ili će se izvršavanje samo nastaviti po isteku zadatog vremena čekanja.

Drugi, češće upotrebljavani oblik metode **wait( )** ne prima nikakve argumente, što znači da će metoda čekati sve dok ta nit ne primi poruku **notify( )** ili **notifyAll( )**; taj oblik metode **wait( )** *ne* završava zadatak automatski posle isteka nekog vremena.

Jedna od posebnih osobina metoda **wait( )**, **notify( )** i **notifyAll( )** jeste to što su one delovi osnovne klase **Object**, a ne klase **Thread**. Mada to na prvi pogled izgleda čudno – da nešto što služi isključivo za niti bude pripadnik univerzalne osnovne klase – ali je neizbežno, jer te metode rade s bravama koje su takođe deo svakog objekta. Zato **wait( )** možete da smestite u svaku sinhronizovanu metodu, bez obzira na to da li u toj konkretnoj klasi postoji ili ne postoji više niti, tj. da li je izvedena iz klase **Thread** ili realizuje interfejs **Runnable**. U stvari, **wait( )**, **notify( )** i **notifyAll( )** možete da pozovete jedino iz sinhronizovanih metoda ili blokova (**sleep( )** možete pozivati i iz nesinhronizovanih metoda, pošto ona ne radi s bravama). Ako bilo koju od tih metoda pozovete iz metode koja nije sinhronizovana, program će se ispravno prevesti, ali ćete tokom izvršavanja dobiti izuzetak tipa **IllegalMonitorStateException** uz pomalo nerazumljivu poruku „current thread not owner" (tekuća nit trenutno nije vlasnik brave objekta s kojim hoće da radi). Poruka znači da zadatak koji poziva **wait( )**, **notify( )** ili **notifyAll( )** mora zaključati bravu objekta pre nego što pozove jednu od tih metoda.

Metode **wait( )**, **notify( )** ili **notifyAll( )** možete da pozivate samo za svoje brave. Nema petljanja s tuđim bravama, ali od drugog objekta možete zatražiti da obavi operaciju koja radi s njegovom bravom. Prethodno morate da pribavite bravu tog objekta. Primera radi, ako objektu **x** želite da pošaljete obaveštenje **notifyAll( )**, to možete uraditi unutar sinhronizovanog bloka koji pribavlja bravu od **x**, tj. zaključava **x**:

```
synchronized(x) {
  x.notifyAll();
}
```

Razmotrićemo jednostavan primer. **VoskOMatik.java** ima dva procesa: nanošenje voska na **Kola** i poliranje. Poliranja nema dok se ne završi nanošenje voska, a zadatak nanošenja mora da sačeka dok se zadatak poliranja završi – tek onda se može naneti sledeći sloj voska. I **VoskaIma** i **VoskaNema** rade sa istim objektom tipa **Kola** koji metodama **wait( )** i **notifyAll( )** zaustavlja i ponovo pokreće zadatke dok oni čekaju da se promeni određeni uslov:

```
//: paralelno/voskomatik/VoskOMatik.java
// Osnove međusobne saradnje zadataka.
package paralelno.voskomatik;
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Kola {
  private boolean voskaIma = false;
  public synchronized void voskirano() {
    voskaIma = true; // Spremno za poliranje
    notifyAll();
  }
```

```
  public synchronized void polirano() {
    voskaIma = false; // Spremno za nanošenje sledećeg sloja voska
    notifyAll();
  }
  public synchronized void cekajNaVoskiranje()
  throws InterruptedException {
    while(voskaIma == false)
      wait();
  }
  public synchronized void cekajNaPoliranje()
  throws InterruptedException {
    while(voskaIma == true)
      wait();
  }
}

class VoskaIma implements Runnable {
  private Kola kola;
  public VoskaIma(Kola k) { kola = k; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Voska ima! ");
        TimeUnit.MILLISECONDS.sleep(200);
        kola.voskirano();
        kola.cekajNaPoliranje();
      }
    } catch(InterruptedException e) {
      print("Izlazim iz niti pomoću prekida");
    }
    print("Završavam zadatak Voska Ima");
  }
}

class VoskaNema implements Runnable {
  private Kola kola;
  public VoskaNema(Kola k) { kola = k; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        kola.cekajNaVoskiranje();
        printnb("Voska nema! ");
        TimeUnit.MILLISECONDS.sleep(200);
        kola.polirano();
      }
    } catch(InterruptedException e) {
      print("Izlazim iz niti pomoću prekida");
    }
    print("Završavam zadatak Voska Nema");
  }
```

```
}

public class VoskOMatik {
  public static void main(String[] args) throws Exception {
    Kola kola = new Kola();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new VoskaNema(kola));
    exec.execute(new VoskaIma(kola));
    TimeUnit.SECONDS.sleep(5); // Radi neko vreme...
    exec.shutdownNow(); // Prekini sve zadatke
  }
} /* Ispis: (95% podudaranja)
Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska
ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima!
Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska
nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Izlazim iz niti
putem prekida
Završavam zadatak Voska Ima
Izlazim iz niti pomoću prekida
Završavam zadatak Voska Nema
*///:~
```

Kola imaju samo jednu promenljivu tipa **boolean** pod imenom **voskaIma** koja pokazuje stanje procesa nanošenja voska i poliranja.

U metodi **cekajNaVoskiranje**( ) proverava se indikator **voskaIma**, i ako je njegova vrednost **false**, pozivajući zadatak se zaustavlja tako što se poziva metoda **wait**( ). Važno je da se to dešava u sinhronizovanoj metodi, gde je zadatak pribavio bravu. Kada pozovete **wait**( ), nit se zaustavlja, a brava otključava. Neophodno je da brava bude otključana zato što drugi zadaci moraju imati mogućnost da je pribave (zaključaju) da bi stanje tog objekta moglo biti bezbedno promenjeno (na primer, da bi vrednost indikatora **voskaIma** promenili u **true**, što moramo da uradimo da bi zaustavljeni zadatak uopšte dobio priliku da nastavi rad). U ovom primeru, kada drugi zadatak pozove **voskirano**( ) da bi pokazao kako je vreme da se nešto uradi, brava mora biti pribavljena (zaključana) da bi se vrednost indikatora **voskaIma** promenila u **true**. Zatim **voskirano**( ) poziva metodu **notifyAll**( ) koja budi zadatak zaustavljen pozivom metode **wait**( ). Da bi se zadatak probudio iz stanja čekanja koje prouzrokuje metoda **wait**( ), najpre mora ponovo da zaključa bravu koju je otključao kada je ušao u taj **wait**( ). Zadatak se neće probuditi dok ta brava ne postane dostupna, tj. otključana.[20]

[20] Na nekim platformama postoji i treći način izlaska iz metode **wait**( ): to je takozvano *slučajno buđenje*. U suštini, slučajno buđenje prouzrokuje nit koja prerano prestane da blokira (dok čeka na promenljivu ili semafor uslova), a da je nisu probudile metode **notify**( ) ili **notifyAll**( ) (ili njihovi ekvivalenti za objekte nove klase **Condition**). Nit se jednostavno probudi, naizgled sama od sebe. Slučajno buđenje postoji zato što realizacija POSIX niti (ili njima ekvivalentnih) na nekim platformama nije onoliko jednostavna koliko bi Sun hteo da bude. No, posao pisanja pthread biblioteke za te platforme mnogo je lakši ako se proglasi da slučajna buđenja „ne smetaju toliko".

VoskaIma.run( ) predstavlja prvi korak u postupku voskiranja kola, pa je njegova operacija poziv metode **sleep( )** kojim simulira vreme potrebno za voskiranje. Zatim kazuje kolima da je voskiranje završeno i poziva **cekajNaPoliranje( )** koja zaustavlja taj zadatak metodom **wait( )** dok zadatak **VoskaNema** ne pozove **polirano( )** za kola, promeni stanje i pozove **notifyAll( )**. S druge strane, **VoskaNema.run( )** odmah prelazi na **cekajNaVoskiranje( )** i stoga biva zaustavljena sve dok **VoskaIma** ne nanese vosak i ne pozove **voskirano( )**. Kada pokrenete ovaj program, videćete kako se ovaj dvostepeni postupak ponavlja dok zadaci jedan drugom predaju kontrolu. Posle pet sekundi, obe niti prekida **interrupt( )**; kada metodu **shutdownNow( )** pozovete za **ExecutorService**, on poziva **interrupt( )** za sve zadatke koje kontroliše.

U prethodnom primeru naglašeno je da poziv metode **wait( )** morate staviti u petlju **while** koja proverava odgovarajuće uslove. To je važno zato što:

- Možete imati više zadataka koji iz istog razloga čekaju na istu bravu, a prvi zadatak koji se probudi može promeniti tu situaciju (čak i ako to sami ne uradite, to može svako ko je izveo svoju klasu iz vaše). U tom slučaju, taj zadatak treba ponovo zaustaviti dok se njegov odgovarajući uslov ne promeni.
- Kada se zadatak probudi iz svog čekanja – metode **wait( )** – možda su drugi zadaci promenili situaciju tako da ovaj zadatak ne može da u tom trenutku obavi svoju operaciju ili nije zainteresovan za to. I opet, zadatak treba ponovo zaustaviti novim pozivom metode **wait( )**.
- Moguće je i da zadaci čekaju na bravu tog objekta *iz različitih razloga*, a tada *morate* upotrebiti **notifyAll( )**. U tom slučaju, *morate* proveriti da li je zadatak probuđen iz pravog razloga, i ako nije, ponovo pozovite **wait( )**.

Dakle, neophodno je da proveravate ispunjenost odgovarajućeg uslova i da se vraćate u **wait( )** ako taj uslov nije ispunjen. To se obično piše pomoću petlje **while**.

Vežba 21: (2) Napravite dve klase koje realizuju interfejs **Runnable**, jednu s metodom **run( )** koja na svom početku poziva **wait( )**. Druga klasa treba da uhvati referencu prvog **Runnable** objekta. Njena metoda **run( )** treba da pozove **notifyAll( )** za prvi zadatak nakon nekoliko sekundi, da bi prvi zadatak mogao da prikaže poruku. Ispitajte klase pomoću izvršioca (**Executor**).

Vežba 22: (4) Napravite primer *zauzetosti čekanjem*. Jedan zadatak neko vreme spava i zatim postavlja vrednost određenog indikatora na **true**. Drugi zadatak posmatra taj indikator iz petlje **while** (to je ta zauzetost čekanjem) i kada indikator poprimi vrednost **true**, zadaje mu vrednost **false** i prijavljuje promenu konzoli. Izmerite vreme koje program gubi na zauzetost čekanjem i napravite drugu verziju programa koja umesto zauzetosti čekanjem koristi **wait( )**.

Propušteni signali

Kada je rad dve niti usklađen metodama **notify( )/wait( )** ili **notifyAll( )/wait( )**, može se propustiti signal. Pretpostavimo da je **N1** nit koja obaveštava nit **N2** i da su obe realizovane na sledeći (pogrešan) način:

```
N1:
synchronized(deljeniMonitor) {
  <priprema uslova za N2>
  deljeniMonitor.notify();
}

N2:
while(nekiUslov) {
  // Tačka 1
  synchronized(deljeniMonitor) {
    deljeniMonitor.wait();
  }
}
```

<priprema uslova za N2> treba da spreči **N2** da zove **wait( )**, ako to **N2** već nije učinila.

Pretpostavimo da **N2** izračuna **nekiUslov** i dobije rezultat **true**. U **Tački1**, mehanizam za raspodelu procesorskog vremena može prebaciti na nit **N1**. **N1** izvršava svoju pripremu i zatim poziva **notify( )**. Kada **N2** nastavi da se izvršava, za nju je prekasno da shvati kako se uslov u međuvremenu promenio, i ona slepo ulazi u **wait( )**. Poruka metode **notify( )** biće propuštena i **N2** će čekati beskonačno dugo na signal koji je već bio poslat; dobili smo uzajamnu blokadu (engl. *deadlock*).

Rešenje je da se spreči uslov za trku primenom promenljive **nekiUslov**. Ovo je ispravan način realizacije **N2**:

```
synchronized(deljeniMonitor) {
  while(nekiUslov)
    deljeniMonitor.wait();
}
```

U ovoj realizaciji važi sledeće: ako nit **N1** prva dobije priliku da se izvršava, kada se kontrola vrati niti **N2** ona će shvatiti da se uslov promenio i *neće* ući u **wait( )**. U protivnom, ukoliko nit **N2** prva dobije priliku da se izvršava, ući će u **wait( )** i kasnije će je probuditi **N1**. Dakle, signal ne može biti propušten.

notify() u odnosu na notifyAll()

Strogo uzev, više zadataka može čekati – metodom **wait( )** – na isti objekat tipa **Kola**, pa je bezbednije pozvati **notifyAll( )** umesto **notify( )**. Međutim, struktura gornjeg programa je takva da će samo jedan zadatak biti u stanju čekanja – **wait( )** – pa smemo da upotrebimo **notify( )** umesto **notifyAll( )**.

Korišćenje **notify( )** umesto **notifyAll( )** predstavlja optimizaciju. Od svih mogućih zadataka koji čekaju na određenu bravu, **notify( )** će probuditi samo jedan, pa ako pokušate da upotrebite **notify( )**, morate biti sigurni da će se probuditi pravi zadatak. Pored toga, svi zadaci moraju čekati na isti uslov da biste mogli da upotrebite **notify( )**, jer ako razni zadaci čekaju na različite uslove, ne možete znati da će se probuditi onaj pravi. Ako upotrebite **notify( )**, samo jedan zadatak mora imati koristi od promene uslova. Najzad, ova ograničenja moraju uvek važiti za sve moguće potklase. Ukoliko bilo koje od ovih pravila ne može biti zadovoljeno, morate koristiti **notifyAll( )**, a ne **notify( )**.

U raspravama o višenitnom izvršavanju u Javi često se ponavlja zbunjujuća tvrdnja da **notifyAll( )** budi „sve zadatke koji čekaju". Da li to znači da svaki poziv metode **notifyAll( )** budi sve zadatke koji su pozvali **wait( )**, na bilo kojem mestu u programu? U narednom primeru, kôd zadatka **Zadatak2** pokazuje da ta tvrdnja nije istinita – zapravo, kada se pozove **notifyAll( )** za određenu bravu, bude se samo zadaci koji čekaju *na tu bravu*:

```
//: paralelno/NotifyIliNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blokator {
  synchronized void cekamPoziv() {
    try {
      while(!Thread.interrupted()) {
        wait();
        System.out.print(Thread.currentThread() + " ");
      }
    } catch(InterruptedException e) {
      // Ovo je prihvatljiv način izlaska
    }
  }
  synchronized void probudi() { notify(); }
  synchronized void probudiSve() { notifyAll(); }
}

class Zadatak implements Runnable {
  static Blokator blokator = new Blokator();
  public void run() { blokator.cekamPoziv(); }
}

class Zadatak2 implements Runnable {
  // Zaseban objekat tipa Blokator:
  static Blokator blokator = new Blokator();
  public void run() { blokator.cekamPoziv(); }
}

public class NotifyIliNotifyAll {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Zadatak());
    exec.execute(new Zadatak2());
    Timer meracVremena = new Timer();
    meracVremena.scheduleAtFixedRate(new TimerTask() {
      boolean probudi = true;
      public void run() {
        if(probudi) {
          System.out.print("\nnotify() ");
          Zadatak.blokator.probudi();
          probudi = false;
```

```
          } else {
            System.out.print("\nnotifyAll() ");
            Zadatak.blokator.probudiSve();
            probudi = true;
          }
        }
      }, 400, 400); // Pokreni na svake 0,4 sekunde
      TimeUnit.SECONDS.sleep(5); // Neka radi neko vreme...
      meracVremena.cancel();
      System.out.println("\nMerač vremena otkazan");
      TimeUnit.MILLISECONDS.sleep(500);
      System.out.print("Zadatak2.blokator.probudiSve() ");
      Zadatak2.blokator.probudiSve();
      TimeUnit.MILLISECONDS.sleep(500);
      System.out.println("\nGasim");
      exec.shutdownNow(); // Prekini sve zadatke
    }
} /* Ispis: (primer)
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool
-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool
-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool
-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool
-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool
-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool
-1-thread-5,5,main]
Merač vremena otkazan
Zadatak2.blokator.probudiSve() Thread[pool-1-thread-6,5,main]
Gasim
*///:-
```

Zadatak i **Zadatak2** imaju sopstvene **Blokator** objekte, pa svaki objekat tipa **Zadatak.blokator** blokira svaki objekat tipa **Zadatak**, i svaki objekat tipa **Zadatak2.blokator** blokira svaki objekat tipa **Zadatak2** . U metodi **main( )**, objekat tipa **java.util.Timer** biva

podešen tako da svoju metodu **run( )** izvršava svake 4/10 sekunde, i ta metoda **run( )** naizmenično poziva **notify( )** i **notifyAll( )** za **Zadatak.blokator** metodama „probudi".

Iz rezultata uočavate sledeće: iako objekat **Zadatak2** postoji i blokirao ga je objekat **Zadatak2.blokator**, nijedan od poziva metoda **notify( )** ili **notifyAll( )** za **Zadatak.blokator** ne prouzrokuje buđenje objekta **Zadatak2**. Isto tako, na kraju metode **main( )**, poziva se **cancel( )** za **meracVremena**, i mada je taj merač vremena otkazan, prvih pet zadataka se i dalje izvršava i još su blokirani u svojim pozivima metode **Zadatak.blokator.cekamPoziv( )**. Rezultat poziva metode **Zadatak2.blokator.probudiSve( )** *nema veze* ni sa jednim od zadataka koji čekaju na bravu objekta **Zadatak.blokator**.

To ima smisla i kada razmotrite metode **probudi( )** i **probudiSve( )** u **Blokator**u. Te metode su sinhronizovane, što znači da same pribavljaju (zaključavaju) svoju bravu, pa kada pozivaju **notify( )** ili **notifyAll( )**, logično je da ih pozivaju samo za tu bravu – i time bude samo zadatke koji čekaju na tu bravu.

Blokator.cekamPoziv( ) toliko je jednostavan da sam mogao da napišem samo **for(;;)** umesto **while(!Thread.interrupted( ))** i u ovom slučaju postignem isti rezultat; to je zato što u ovom primeru nema razlike između napuštanja petlje pomoću izuzetka i napuštanja preko provere indikatora **interrupted( )** – u oba slučaja izvršava se isti kôd. Međutim, u ovom primeru se forme radi proverava **interrupted( )**, pošto postoje dva različita načina za napuštanje petlje. Ukoliko kasnije odlučite da dodate još koda u petlju, rizikujete da napravite grešku ako ne uzmete u obzir obe putanja izlaska iz petlje.

Vežba 23: (7) Pokažite da **VoskOMatik.java** dobro radi i kada upotrebite **notify( )** umesto **notifyAll( )**.

Proizvođači i potrošači

Zamislite restoran s jednim kuvarom i jednim konobarom. Konobar mora da sačeka dok kuvar pripremi jelo. Kuvar obaveštava konobara kada je jelo zgotovljeno. Tada konobar uzima jelo, poslužuje ga pa se vraća i čeka. To je primer saradnje zadataka: kuvar predstavlja *proizvođača*, a konobar *potrošača*. Zadaci moraju da razmenjuju signale tokom proizvodnje i potrošnje jela, i sistem mora da se uredno ugasi. Ovo je ta priča oblikovana kao kôd:

```
//: paralelno/Restoran.java
// Saradnja zadataka kao saradnja proizvođača i potrošača.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Jelo {
  private final int brojNarudzbe;
  public Jelo(int brojNarudzbe) { this.brojNarudzbe = brojNarudzbe; }
  public String toString() { return "Jelo " + brojNarudzbe; }
}

class Konobar implements Runnable {
  private Restoran restoran;
  public Konobar(Restoran r) { restoran = r; }
```

```
    public void run() {
      try {
        while(!Thread.interrupted()) {
          synchronized(this) {
            while(restoran.jelo == null)
              wait(); // ... dok kuvar ne pripremi jelo
          }
          print("Konobar je uzeo " + restoran.jelo);
          synchronized(restoran.kuvar) {
            restoran.jelo = null;
            restoran.kuvar.notifyAll(); // Spreman za sledeće
          }
        }
      } catch(InterruptedException e) {
        print("Konobar prekinut");
      }
    }
}

class Kuvar implements Runnable {
  private Restoran restoran;
  private int broj = 0;
  public Kuvar(Restoran r) { restoran = r; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          while(restoran.jelo != null)
            wait(); // ... da jelo bude posluženo
        }
        if(++broj == 10) {
          print("Ponestalo hrane, zatvaramo");
          restoran.exec.shutdownNow();
        }
        printnb("Evo narudzbe! ");
        synchronized(restoran.konobar) {
          restoran.jelo = new Jelo(broj);
          restoran.konobar.notifyAll();
        }
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      print("Kuvar prekinut");
    }
  }
}

public class Restoran {
  Jelo jelo;
```

```
  ExecutorService exec = Executors.newCachedThreadPool();
  Konobar konobar = new Konobar(this);
  Kuvar kuvar = new Kuvar(this);
  public Restoran() {
    exec.execute(kuvar);
    exec.execute(konobar);
  }
  public static void main(String[] args) {
    new Restoran();
  }
} /* Ispis:
Evo narudzbe! Konobar je uzeo Jelo 1
Evo narudzbe! Konobar je uzeo Jelo 2
Evo narudzbe! Konobar je uzeo Jelo 3
Evo narudzbe! Konobar je uzeo Jelo 4
Evo narudzbe! Konobar je uzeo Jelo 5
Evo narudzbe! Konobar je uzeo Jelo 6
Evo narudzbe! Konobar je uzeo Jelo 7
Evo narudzbe! Konobar je uzeo Jelo 8
Evo narudzbe! Konobar je uzeo Jelo 9
Ponestalo hrane, zatvaramo
Konobar prekinut
Evo narudzbe! Kuvar prekinut
*///:~
```

Restoran je žiža i za **Konobar**a i za **Kuvar**a. Oba moraju znati za koji **Restoran** rade, pošto moraju staviti jelo, **restoran.jelo**, u restoranski „izlog s jelima", odnosno, moraju izvaditi jelo iz njega. U metodi **run**( ), **Konobar** prelazi u režim čekanja – **wait**( ) – i zaustavlja taj zadatak dok ga ne probudi poruka **notifyAll**( ) od **Kuvar**a. Pošto je ovaj program veoma jednostavan, znamo da samo jedan zadatak čeka na **Konobar**ovu bravu: to je sam zadatak **Konobar**. Zato sam umesto **notifyAll**( ) mogao pozvati **notify**( ). Medutim, u složenijim situacijama, na bravu odredenog objekta može čekati više zadataka, pa tada ne znamo koji od njih treba probuditi. Zato je bezbednije pozvati **notifyAll**( ); ona budi sve zadatke koji čekaju na određenu bravu. Potom svaki od tih zadataka mora odlučiti da li je to obaveštenje relevantno za njega.

Nakon što **Kuvar** proizvede **Jelo** i o tome obavesti (engl. *notify*) **Konobar**a, **Kuvar** čeka dok **Konobar** ne uzme jelo i o tome obavesti **Kuvar**a, a on zatim može da proizvede sledeće **Jelo**.

Obratite pažnju na to da je **wait**( ) omotana u naredbu **while**( ) koja ispituje ono isto na šta se čeka. To isprva izgleda malo čudno – ako čekate (spavajući) na narudžbu i budete probudeni, to mora značiti da postoji nova narudžba, zar ne? Već sam rekao da je problem to što u paralelnoj aplikaciji neki drugi zadatak može da uleti i ugrabi narudžbu dok se **Konobar** budi. Jedini bezbedan način jeste da za **wait**( ) *uvek* upotrebljavate sledeći idiom (uz pravilnu sinhronizaciju, naravno, i programiranje koje sprečava propuštanje signala):

```
while(uslovNijeIspunjen)
  wait();
```

Time se jemči da će uslov biti ispunjen pre nego što izađete iz petlje čekanja, i ako ste obavešteni o nečemu što nema veze sa uslovom – što se može desiti kada je poruka **notifyAll( )** – ili se uslov promeni pre nego što potpuno izađete iz petlje čekanja, svakako ćete se vratiti u čekanje.

Vodite računa o tome da poziv **notifyAll( )** mora najpre zaključati bravu **konobar**a. Poziv metode **wait( )** iz metode **Konobar.run( )** automatski otključava bravu, pa je to moguće. Pošto brava mora biti zaključana da bi **notifyAll( )** mogla biti pozvana, dva zadatka koji pokušavaju da pozovu **notifyAll( )** za isti objekat zajemčeno neće zasmetati jedan drugom.

Obe metode **run( )** uredno se gase tako što je cela metoda **run( )** umetnuta u blok **try**. Odredba **catch** završava se neposredno pre završne vitičaste zagrade metode **run( )**, pa ako zadatak primi **InterruptedException**, završiće se odmah čim uhvati izuzetak.

U klasi **Kuvar**, obratite pažnju na to da bismo nakon poziva metode **shutdownNow( )** običnom naredbom **return** *mogli* da se jednostavno vratimo iz metode **run( )**, i to najčešće i treba da uradite. Međutim, ovako je malo zanimljivije. Prisetite se da **shutdownNow( )** šalje **interrupt( )** svim zadacima koje je **ExecutorService** pokrenuo. Ali u slučaju klase **Kuvar**, zadatak se ne gasi neposredno po prijemu poruke **interrupt( )**, pošto prekid može da generiše izuzetak **InterruptedException** jedino kada zadatak pokuša da uđe u (neprekidnu) operaciju koja blokira. Zato se najpre prikazuje „Evo narudzbe!" i zatim baca izuzetak **InterruptedException** kada **Kuvar** pokuša da pozove **sleep( )**. Ako uklonite poziv metode **sleep( )**, zadatak će doći na vrh petlje **run( )** i izaći zbog ispitivanja vrednosti metode **Thread.interrupted( )**, a da ne baci izuzetak.

U prethodnom primeru, zadatak na samo jedno mesto može da smesti objekat koji će drugi zadatak kasnije upotrebiti. Međutim, u tipičnoj realizaciji proizvođača i potrošača, za skladištenje objekata koji se proizvode i troše primenjuje se princip FIFO (*first-in, first-out* – prvi izlazi onaj koji je prvi ušao). Više reči o takvim redovima za čekanje biće u nastavku poglavlja.

Vežba 24: (1) Metodama **wait( )** i **notifyAll( )** rešite problem jednog proizvođača i jednog potrošača. Proizvođač ne sme da prepuni primaočev bafer, što se može desiti ukoliko je proizvođač brži od potrošača. Ako je potrošač brži od proizvođača, onda isti podatak treba da pročita samo jednom. Ne smete ništa pretpostaviti o relativnim brzinama proizvođača i potrošača.

Vežba 25: (1) U klasi **Kuvar** programa **Restoran.java**, vratite se naredbom **return** iz metode **run( )** nakon poziva metode **shutdownNow( )** i posmatrajte razliku u ponašanju.

Vežba 26: (8) Programu **Restoran.java** dodajte klasu **Pomocnik**. Kada posluži jelo, **Konobar** treba da obavesti **Pomocnik**a da počisti.

Korišćenje eksplicitnih objekata tipa **Lock** i **Condition**

U biblioteci Java SE5 **java.util.concurrent** ima i drugih, eksplicitnih alatki pomoću kojih možemo preraditi program **VoskOMatik.java**. Osnovna klasa koja koristi uzajamno isključivu bravu (mutex) i omogućuje zaustavljanje zadataka nazvana je **Condition**.

Zadatak zaustavljate pozivom metode **await( )** za objekat tipa **Condition**. Kada nastanu spoljne promene stanja koje možda znače da neki zadatak treba da nastavi izvršavanje, taj zadatak obaveštavate porukom **signal( )**, odnosno **signalAll( )** da biste probudili sve zadatke koji su se zaustavili do ispunjenja uslova tj. tog objekta **Condition** (kao i **notifyAll( )**, **signalAll( )** je bezbednije koristiti).

Evo kako izgleda program **VoskOMatik.java** preraden tako da sadrži **Condition** pomoću kojeg se određeni zadatak zaustavlja unutar metoda **cekajNaVoskiranje( )** ili **cekajNaPoliranje( )**:

```
//: paralelno/voskomatik2/VoskOMatik2.java
// Upotreba objekata Lock i Condition.
package paralelno.voskomatik2;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Kola {
  private Lock brava = new ReentrantLock();
  private Condition uslov = brava.newCondition();
  private boolean voskaIma = false;
  public void voskirano() {
    brava.lock();
    try {
      voskaIma = true; // Spremna za poliranje
      uslov.signalAll();
    } finally {
      brava.unlock();
    }
  }
  public void polirano() {
    brava.lock();
    try {
      voskaIma = false; // Spremna za nanošenje sledećeg sloja voska
      uslov.signalAll();
    } finally {
      brava.unlock();
    }
  }
  public void cekajNaVoskiranje() throws InterruptedException {
    brava.lock();
    try {
      while(voskaIma == false)
        uslov.await();
    } finally {
      brava.unlock();
    }
  }
```

```
    public void cekajNaPoliranje() throws InterruptedException{
      brava.lock();
      try {
        while(voskaIma == true)
          uslov.await();
      } finally {
        brava.unlock();
      }
    }
  }

  class VoskaIma implements Runnable {
    private Kola kola;
    public VoskaIma(Kola k) { kola = k; }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          printnb("Voska ima! ");
          TimeUnit.MILLISECONDS.sleep(200);
          kola.voskirano();
          kola.cekajNaPoliranje();
        }
      } catch(InterruptedException e) {
        print("Izlazim iz niti pomoću prekida");
      }
      print("Završavam zadatak Voska Ima");
    }
  }

  class VoskaNema implements Runnable {
    private Kola kola;
    public VoskaNema(Kola k) { kola = k; }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          kola.cekajNaVoskiranje();
          printnb("Voska nema! ");
          TimeUnit.MILLISECONDS.sleep(200);
          kola.polirano();
        }
      } catch(InterruptedException e) {
        print("Izlazim iz niti pomoću prekida");
      }
      print("Završavam zadatak Voska Nema");
    }
  }

  public class VoskOMatik2 {
    public static void main(String[] args) throws Exception {
      Kola kola = new Kola();
```

```
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new VoskaNema(kola));
        exec.execute(new VoskaIma(kola));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
      }
    } /* Ispis: (90% podudaranja)
    Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska
    ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima!
    Voska nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Voska
    nema! Voska ima! Voska nema! Voska ima! Voska nema! Voska ima! Izlazim iz
    niti pomoću prekida
    Završavam zadatak Voska Nema
    Izlazim iz niti pomoću prekida
    Završavam zadatak Voska Ima
    *///:~
```

U konstruktoru **Kola**, jedna brava (objekat tipa **Lock**) proizvodi uslov, tj. objekat tipa **Condition** pomoću kojeg se upravlja komunikacijom između zadataka. Međutim, objekat **Condition** ne sadrži informacije o stanju procesa, pa su potrebne dodatne informacije kojima pokazujete stanje procesa, a to je promenljiva **voskaIma** tipa **boolean**.

Iza svakog poziva metode **lock**( ) mora neposredno slediti odredba **try-finally** koja jemči da će se u svim slučajevima ostvariti otključavanje. Kao u ugrađenim verzijama, zadatak mora zaključati bravu da bi mogao da pozove **await**( ), **signal**( ) ili **signalAll**( ).

Imajte u vidu da je ovo rešenje složenije nego prethodno, a ta složenost u ovom slučaju nije donela ništa dobro. Objekti tipa **Lock** i **Condition** potrebni su samo za teže probleme višenitnog izvršavanja.

Vežba 27: (2) Izmenite **Restoran.java** tako da koristi eksplicitne objekte tipa **Lock** i **Condition**.

Proizvođači, potrošači i redovi za čekanje

Metode **wait**( ) i **notifyAll**( ) rešavaju problem saradnje zadataka na prilično niskom nivou, razmenjujući signale u svakoj interakciji. U mnogo slučajeva, možete se popeti za jedan nivo apstrakcije i probleme saradnje zadataka rešiti pomoću *sinhronizovanog reda za čekanje* koji u svakom trenutku dozvoljava samo po jednom zadatku da u red umetne ili iz njega ukloni neki element. Taj red sadrži interfejs **java.util.concurrent.BlockingQueue** koji ima više standardnih realizacija. Najčešće ćete koristiti **LinkedBlockingQueue**, što je neograničen red za čekanje; **ArrayBlockingQueue** ima nepromenljivu veličinu, pa u njega možete staviti samo nevelik broj elemenata pre nego što se zablokira.

Takav red za čekanje zaustavlja potrošački zadatak koji pokuša da uzme objekat iz praznog reda i nastavlja njegovo izvršavanje kada više elemenata postane dostupno. Blokiranje redova za čekanje rešava veliki broj problema na mnogo jednostavniji i pouzdaniji način od metoda **wait**( ) i **notifyAll**( ).

Sledi jednostavan test koji serijalizuje izvršavanje **Lansiranje** objekata. Potrošač je **IzvršavaLansiranje**, koji vadi svaki objekat **Lansiranje** iz blokirajućeg reda za čekanje (**BlockingQueue**) i izvršava ga neposredno. (Drugim rečima, eksplicitnim pozivom metode **run**( ) koristi sopstvenu nit, umesto da za svaki zadatak pokreće novu.)

```
//: paralelno/TestBlokirajucihRedovaZaCekanje.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class IzvrsavaLansiranje implements Runnable {
  private BlockingQueue<Lansiranje> rakete;
  public IzvrsavaLansiranje(BlockingQueue<Lansiranje> redZaCekanje) {
    rakete = redZaCekanje;
  }
  public void add(Lansiranje lo) {
    try {
      rakete.put(lo);
    } catch(InterruptedException e) {
      print("Prekinuto tokom operacije put()");
    }
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        Lansiranje raketa = rakete.take();
        raketa.run(); // Koristi ovu nit
      }
    } catch(InterruptedException e) {
      print("Buđenje iz metode take()");
    }
    print("Izlazim iz niti IzvrsavaLansiranje");
  }
}

public class TestBlokirajucihRedovaZaCekanje {
  static void getkey() {
    try {
      // Zato što taster Enter proizvodi rezultat koji je
      // različite dužine u Windowsu i u Linuxu:
      new BufferedReader(
        new InputStreamReader(System.in)).readLine();
    } catch(java.io.IOException e) {
      throw new RuntimeException(e);
    }
  }
  static void getkey(String poruka) {
    print(poruka);
    getkey();
  }
```

```
  static void
  test(String prk, BlockingQueue<Lansiranje> redZaCekanje) {
    print(prk);
    IzvrsavaLansiranje izvrsilac = new IzvrsavaLansiranje(redZaCekanje);
    Thread n = new Thread(izvrsilac);
    n.start();
    for(int i = 0; i < 5; i++)
      izvrsilac.add(new Lansiranje(5));
    getkey("Pritisnite 'Enter' (" + prk + ")");
    n.interrupt();
    print("Završen " + prk + " test");
  }
  public static void main(String[] args) {
    test("LinkedBlockingQueue", // Neograničena veličina
      new LinkedBlockingQueue<Lansiranje>());
    test("ArrayBlockingQueue", // Nepromenljiva veličina
      new ArrayBlockingQueue<Lansiranje>(3));
    test("SynchronousQueue", // Veličine 1
      new SynchronousQueue<Lansiranje>());
  }
} ///:~
```

Metoda **main( )** smešta zadatke u **BlockingQueue**, iz kojeg ih vadi metoda **IzvrsavaLansiranje**. Obratite pažnju na to da **IzvrsavaLansiranje** može da zaboravi na sinhronizaciju zato što to rešava **BlockingQueue**.

Vežba 28: (3) Izmenite **TestBlokirajucihRedovaZaCekanje.java** dodavanjem novog zadatka koji stavlja **Lansiranje** u **BlockingQueue**, umesto da se to radi u metodi **main( )**.

Blokirajući redovi za čekanje tosta

Kao primer korišćenja blokirajućih redova za čekanje (**BlockingQueue**), uzmimo mašinu koja ima tri zadatka: pravljenje tosta, mazanje maslaca na tost i mazanje pekmeza na tost namazan maslacem. Između tih postupaka tost može da se nalazi u blokirajućim redovima za čekanje:

```
//: paralelno/TostOMatik.java
// Toster koji upotrebljava redove za čekanje.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Tost {
  public enum Status { SUV, SMASLACEM, SPEKMEZOM }
  private Status status = Status.SUV;
  private final int id;
  public Tost(int idn) { id = idn; }
  public void namaziMaslacem() { status = Status.SMASLACEM; }
  public void namaziPekmezom() { status = Status.SPEKMEZOM; }
  public Status dajStatus() { return status; }
```

```
    public int dajId() { return id; }
    public String toString() {
      return "Tost " + id + ": " + status;
    }
  }

  class RedZaCekanjeTosta extends LinkedBlockingQueue<Tost> {}

  class Toster implements Runnable {
    private RedZaCekanjeTosta redZaCekanjeTosta;
    private int broj = 0;
    private Random slucajan = new Random(47);
    public Toster(RedZaCekanjeTosta rzct) { redZaCekanjeTosta = rzct; }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          TimeUnit.MILLISECONDS.sleep(
            100 + slucajan.nextInt(500));
          // Napravi tost
          Tost n = new Tost(broj++);
          print(n);
          // Umetni u redZaCekanje
          redZaCekanjeTosta.put(n);
        }
      } catch(InterruptedException e) {
        print("Toster prekinut");
      }
      print("Toster isključen");
    }
  }

  // Namaži tost maslacem:
  class MazeMaslacem implements Runnable {
    private RedZaCekanjeTosta redSuvih, redSMaslacem;
    public MazeMaslacem(RedZaCekanjeTosta suv, RedZaCekanjeTosta smaslacem) {
      redSuvih = suv;
      redSMaslacem = smaslacem;
    }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          // Blokira dok ne dobije sledeće parče tosta:
          Tost n = redSuvih.take();
          n.namaziMaslacem();
          print(n);
          redSMaslacem.put(n);
        }
      } catch(InterruptedException e) {
        print("MazeMaslacem prekinut");
      }
```

```
    print("MazeMaslacem isključen");
  }
}

// Namaži pekmezom tost s maslacem:
class MazePekmezom implements Runnable {
  private RedZaCekanjeTosta redSMaslacem, redGotovih;
  public MazePekmezom(RedZaCekanjeTosta smaslacem, RedZaCekanjeTosta gotovi)
{
    redSMaslacem = smaslacem;
    redGotovih = gotovi;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokira dok ne dobije sledeće parče tosta:
        Tost n = redSMaslacem.take();
        n.namaziPekmezom();
        print(n);
        redGotovih.put(n);
      }
    } catch(InterruptedException e) {
      print("MazePekmezom prekinut");
    }
    print("MazePekmezom isključen");
  }
}

// Potroši tost:
class Gost implements Runnable {
  private RedZaCekanjeTosta redGotovih;
  private int brojac = 0;
  public Gost(RedZaCekanjeTosta gotovi) {
    redGotovih = gotovi;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokira se dok ne dobije sledeće parče tosta:
        Tost n = redGotovih.take();
        // Proveri da li tost uredno dolazi,
        // i da li sva parčad bivaju namazana pekmezom:
        if(n.dajId() != brojac++ ||
           n.dajStatus() != Tost.Status.SPEKMEZOM) {
          print(">>>> Greška: " + n);
          System.exit(1);
        } else
          print("Njam! " + n);
      }
    } catch(InterruptedException e) {
```

```
        print("Gost prekinut");
      }
      print("Gost isključen");
    }
  }

  public class TostOMatik {
    public static void main(String[] args) throws Exception {
      RedZaCekanjeTosta redSuvih = new RedZaCekanjeTosta(),
                        redSMaslacem = new RedZaCekanjeTosta(),
                        redGotovih = new RedZaCekanjeTosta();
      ExecutorService exec = Executors.newCachedThreadPool();
      exec.execute(new Toster(redSuvih));
      exec.execute(new MazeMaslacem(redSuvih, redSMaslacem));
      exec.execute(new MazePekmezom(redSMaslacem, redGotovih));
      exec.execute(new Gost(redGotovih));
      TimeUnit.SECONDS.sleep(5);
      exec.shutdownNow();
    }
  } /* (Pokrenite da biste videli rezultat) *///:~
```

Tost je odličan primer vrednosti nabrojanih tipova. Vodite računa o tome da nema eksplicitne sinhronizacije (ne upotrebljavaju se objekti tipa **Lock** niti rezervisana reč **synchronized**), pošto sinhronizaciju implicitno (interno) obavljaju redovi za čekanje i projekat sistema – svako parče **Tost**a u svakom trenutku predmet je obrade samo jednog zadatka. Pošto redovi za čekanje blokiraju izvršavanje, procesi se zaustavljaju i nastavljaju automatski. Nadam se da vidite kako pojednostavljenje koje su doneli blokirajući redovi za čekanje ume da bude veoma veliko. Izbegnute su veze između klasa koje bi postojale uz eksplicitne naredbe **wait( )** i **notifyAll( )**, jer svaka klasa komunicira samo sa svojim blokirajućim redom za čekanje.

Vežba 29: (8) Izmenite **TostOMatik.java** tako da na dve zasebne linije pravi sendviče od tosta namazanog maslacem i marmeladom (jedna za tost namazan maslacem, druga za marmeladu, zatim spojite linije).

Cevi za ulazno/izlazne operacije između zadataka

Često je korisno da zadaci međusobno komuniciraju pomoću ulazno/izlaznih operacija. Biblioteke za višenitno izvršavanje podržavaju ulazno/izlazne operacije između zadataka u obliku *cevi* (engl. *pipes*). U Javinoj biblioteci za ulazno/izlazne operacije njih predstavljaju klase **PipedWriter** (pomoću koje zadatak piše u cev) i **PipedReader** (pomoću koje drugi zadatak čita iz iste cevi). To možete smatrati varijacijom problema proizvođača i potrošača, uz cevovod kao gotovo rešenje. U suštini, cev je blokirajući red za čekanje koji je postojao u verzijama Jave pre uvođenja klase **BlockingQueue**.

Sledi jednostavan primer u kojem dva zadatka komuniciraju pomoću cevi:

```
//: paralelno/UICevovod.java
// Ulazno/izlazna komunikacija zadataka kroz cevi
import java.util.concurrent.*;
```

```
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Predajnik implements Runnable {
  private Random slucajan = new Random(47);
  private PipedWriter izlaz = new PipedWriter();
  public PipedWriter getPipedWriter() { return izlaz; }
  public void run() {
    try {
      while(true)
        for(char znak = 'A'; znak <= 'z'; znak++) {
          izlaz.write(znak);
          TimeUnit.MILLISECONDS.sleep(slucajan.nextInt(500));
        }
    } catch(IOException e) {
      print(e + " Izuzetak tokom pisanja Predajnika");
    } catch(InterruptedException e) {
      print(e + " Prekinuto spavanje Predajnika");
    }
  }
}

class Prijemnik implements Runnable {
  private PipedReader ulaz;
  public Prijemnik(Predajnik predajnik) throws IOException {
    ulaz = new PipedReader(predajnik.getPipedWriter());
  }
  public void run() {
    try {
      while(true) {
        // Blokira proces dok ne dobije znakove:
        printnb("Čitam: " + (char)ulaz.read() + ", ");
      }
    } catch(IOException e) {
      print(e + " Izuzetak tokom čitanja Prijemnika");
    }
  }
}

public class UICevovod {
  public static void main(String[] args) throws Exception {
    Predajnik predajnik = new Predajnik();
    Prijemnik prijemnik = new Prijemnik(predajnik);
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(predajnik);
    exec.execute(prijemnik);
    TimeUnit.SECONDS.sleep(4);
    exec.shutdownNow();
  }
```

```
} /* Ispis: (65% podudaranja)
Čitam: A, Čitam: B, Čitam: C, Čitam: D, Čitam: E, Čitam: F, Čitam: G, Čitam:
H, Čitam: I, Čitam: J, Čitam: K, Čitam: L, Čitam: M, java.lang.InterruptedEx-
ception: sleep interrupted Prekinuto spavanje Predajnika
java.io.InterruptedIOException Izuzetak tokom čitanja Prijemnika
*///:~
```

Predajnik i **Prijemnik** predstavljaju zadatke koji treba međusobno da komuniciraju. **Predajnik** pravi objekat tipa **PipedWriter** što jeste samostalan objekat, ali unutar **Prijemnika** pravljenje **PipedReadera** mora biti pridruženo određenom **PipedWriteru** u konstruktoru. **Predajnik** smešta podatke u **Writer** i odlazi na nasumično dugo spavanje. Međutim, **Prijemnik** nema ni metode **sleep**( ) niti **wait**( ). Ali kada pokrene metodu **read**( ), cev se automatski blokira čim ponestane podataka za čitanje.

Obratite pažnju na to da se **predajnik** i **prijemnik** pokreću u metodi **main**( ), *nakon* što su objekti potpuno konstruisani. Ako potpuno konstruisane objekte ne pokrenete, cev se različito ponaša na raznim platformama. (Vodite računa o tome da su objekti tipa **BlockingQueue** (blokirajući redovi za čekanje) robusniji i lakše se koriste.)

Važna razlika između objekta tipa **PipedReader** i normalnih ulazno/izlaznih operacija vidi se u pozivu metode **shutdownNow**( ) – **PipedReader** se može prekinuti; s druge strane, ako biste poziv **ulaz.read**( ) promenili u **System.ulaz.read**( ), metoda **interrupt**( ) ne bi uspela da izade iz poziva **read**( ).

Vežba 30: (1) Izmenite **UICevovod.java** tako da se umesto cevi koristi **BlockingQueue**.

Uzajamna blokada

Sada znate da objekat može imati sinhronizovane metode ili druge oblike zaključavanja koji sprečavaju zadatke da pristupaju tom objektu dok se ne otključa uzajamno isključiva brava (mutex). Saznali ste i da zadaci mogu postati blokirani. Dakle, moguće je da se jedan zadatak zaglavi čekajući na drugi, koji čeka na treći itd., dok lanac ne dođe do zadatka koji čeka na onaj prvi zadatak – neprekidan lanac zadataka koji čekaju jedan na drugog i nijedan od njih ne može da se pomeri. To se naziva *uzajamna blokada* (engl. *deadlock*).[21]

Ako pokrenete program i on odmah uđe u uzajamnu blokadu, grešku ćete odmah moći da pronađete. Pravi problem je kada program naizgled radi dobro, ali sadrži skrivenu mogućnost uzajamne blokade. U tom slučaju, uopšte ne morate dobiti nikakvu naznaku da mogućnost uzajamne blokade postoji, pa će greška biti latentna u programu dok se neočekivano ne desi kupcu (na način koji će gotovo sigurno biti teško ponoviti). Zato je pažljivo projektovanje programa kako bi se sprečila uzajamna blokada, ključan deo razvoja paralelnih sistema.

Klasičan primer uzajamne blokade jeste problem *večere filozofa*, koji je smislio Edsger Dijkstra. U njegovoj verziji, filozofa ima pet, ali u primeru koji ću ja ovde opisati, broj filozofa je proizvoljan. Ti filozofi deo vremena provode u razmišljanju, a deo u jelu. Dok razmišljaju, ne koriste deljene resurse, ali za jelo imaju na raspolaganju ograničenu

[21] Može nastati i *uzajamna blokada uprkos izvršavanju*, kada zadaci menjaju svoja stanja (ne blokiraju se), ali ne uspevaju da naprave išta korisno.

količinu pribora. U originalnom opisu problema, pribor za jelo su viljuške, i za vađenje špageta iz činije na sredini stola potrebne su dve viljuške, ali meni se čini da je prikladnije da za pribor za jelo proglasimo štapiće (kakvima Istočnjaci jedu pirinač). Jasno, svakom filozofu trebaju dva štapića za jelo.

Zaplet problema je sledeći: budući da su filozofi, veoma su siromašni i imaju samo pet štapića (odnosno štapića ima onoliko koliko ima i filozofa). Štapići su ravnomerno raspoređeni po celom stolu. Kada filozof želi da jede, mora da uzme jedan štapić sa svoje leve strane i jedan s desne. Ako je neki od filozofa s njegove leve ili desne strane već uzeo jedan od željenih štapića, naš filozof mora da sačeka dok potrebni štapić ne postane dostupan.

```
//: paralelno/Stapic.java
// Štapići za večeru filozofa.

public class Stapic {
  private boolean zauzet = false;
  public synchronized
  void take() throws InterruptedException {
    while(zauzet)
      wait();
    zauzet = true;
  }
  public synchronized void pusti() {
    zauzet = false;
    notifyAll();
  }
} ///:~
```

Dva **Filozofa** ne mogu istovremeno uzeti (metodom **take( )**) isti **Stapic**. Sem toga, ako je jedan **Filozof** već uzeo određeni **Stapic**, drugi može metodom **wait( )** da sačeka dok taj **Stapic** ne postane dostupan, a to će se desiti kada trenutni korisnik pozove metodu **pusti( )**.

Kada zadatak **Filozof** pozove metodu **take( )**, on čeka dok indikator **zauzet** ne poprimi vrednost **false** (dok **Filozof** koji ga trenutno drži ne pusti taj **Stapic**). Zatim zadatak daje indikatoru **zauzet** vrednost **true** da bi pokazao da **Stapic** sada drži novi **Filozof**. Kada taj **Filozof** završi jelo pomoću **Stapica**, on poziva metodu **pusti( )** da bi promenio vrednost tog indikatora i metodu **notifyAll( )** kojom obaveštava sve druge **Filozofe** koji možda čekaju na taj **Stapic**.

```
//: paralelno/Filozof.java
// Filozof večera
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Filozof implements Runnable {
  private Stapic levi;
  private Stapic desni;
  private final int id;
```

```
  private final int tezinskiFaktor;
  private Random slucajan = new Random(47);
  private void pause() throws InterruptedException {
    if(tezinskiFaktor == 0) return;
    TimeUnit.MILLISECONDS.sleep(
      slucajan.nextInt(tezinskiFaktor * 250));
  }
  public Filozof(Stapic levi, Stapic desni,
    int ident, int ponder) {
    this.levi = levi;
    this.desni = desni;
    id = ident;
    tezinskiFaktor = ponder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        print(this + " " + "razmišlja");
        pause();
        // Filozof je ogladneo
        print(this + " " + "uzima desni");
        desni.take();
        print(this + " " + "uzima levi");
        levi.take();
        print(this + " " + "jede");
        pause();
        desni.pusti();
        levi.pusti();
      }
    } catch(InterruptedException e) {
      print(this + " " + "izlazi pomoću prekida");
    }
  }
  public String toString() { return "Filozof " + id; }
} ///:~
```

U metodi **Filozof.run**( ), svaki **Filozof** ili misli ili jede. Metoda **pause**( ) prouzrokuje spavanje zadatka tokom nasumično odabranog vremenskog razdoblja ukoliko je **tezinskiFaktor** različit od nule. Dakle, **Filozof** razmišlja tokom nasumično odabranog vremenskog razdoblja, zatim pokušava da uzme desni i levi **Stapic**, jede tokom nasumično odabranog vremenskog razdoblja i zatim ponavlja ceo postupak od početka.

Sada ćemo napraviti verziju programa koja će se završiti uzajamnom blokadom:

```
//: paralelno/UzajamnoBlokiraniFilozofiKojiVeceraju.java
// Pokazuje kako u programu može biti prikrivena uzajamna blokada.
// {Args: 0 5 odmor}
import java.util.concurrent.*;
```

```
public class UzajamnoBlokiraniFilozofiKojiVeceraju {
  public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
      ponder = Integer.parseInt(args);
    int broj = 5;
    if(args.length > 1)
      broj = Integer.parseInt(args);
    ExecutorService exec = Executors.newCachedThreadPool();
    Stapic[] stapici = new Stapic[broj];
    for(int i = 0; i < broj; i++)
      stapici[i] = new Stapic();
    for(int i = 0; i < broj; i++)
      exec.execute(new Filozof(
        stapici[i], stapici[(i+1) % broj], i, ponder));
    if(args.length == 3 && args.equals("odmor"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Pritisnite 'Enter' ako želite da prekinete
      izvršavanje programa");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Primetili ste sledeće: ukoliko **Filozofi** provode malo vremena razmišljajući, kada pokušaju da jedu svi će jedan drugom konkurisati za **Stapice** i uzajamna blokada će nastupiti mnogo brže.

Prvi argument na komandnoj liniji zadaje težinski faktor **ponder** koji utiče na količinu vremena koje svaki **Filozof** provodi u razmišljanju. Ako **Filozofa** ima mnogo ili oni vazdan razmišljaju, možda uopšte nećete videti uzajamnu blokadu, iako je ona i dalje moguća. Nula kao argument na komandnoj liniji čini da program prilično brzo uđe u uzajamnu blokadu.

Obratite pažnju na to da objektima tipa **Stapic** nisu potrebni interni identifikatori; njih identifikuje njihov položaj u nizu **stapici**. Konstruktor svakog objekta tipa **Filozof** dobija referencu na levi i desni **Stapic**. Svaki **Filozof** (sem poslednjeg) u inicijalizaciji biva smešten između sledećeg para **Stapica**. Poslednji **Filozof** dobija nulti **Stapic** kao svoj desni i tako se obišao krug oko stola – naime, poslednji **Filozof** sedi odmah do prvog i obojica dele nulti **Stapic**. Ukoliko sada svi **Filozofi** istovremeno pokušaju da jedu, svaki od njih će morati da čeka dok susedni **Filozof** ne spusti svoj **Stapic**. Program će zato ući u uzajamnu blokadu.

Ukoliko **Filozofi** više vremena provode u razmišljanju nego u jelu, onda je i verovatnoća da će im zatrebati deljeni resursi (**Stapici**) mnogo manja, i vi ćete biti uvereni da program ne može ući u uzajamnu blokadu (ako za **ponder** zadate vrednost različitu od nule ili veliki broj **Filozofa**), iako to nije istina. Ovaj primer je zanimljiv upravo zato što pokazuje da program može naizgled da radi ispravno, a da u stvari krije uzajamnu blokadu.

Da biste rešili problem, morate shvatiti da uzajamna blokada nastaje ako su istovremeno ispunjena četiri uslova:

1. Uzajamna isključivost. Barem jedan resurs koji zadaci upotrebljavaju ne sme biti deljiv. U ovom slučaju, svaki **Stapic** u svakom trenutku može koristiti samo jedan **Filozof**.
2. Barem jedan zadatak mora zauzimati resurs i čekati na resurs koji trenutno koristi drugi zadatak. Drugim rečima, da bi se ostvarila uzajamna blokada, **Filozof** mora držati jedan **Stapic** i čekati na drugi.
3. Resurs ne može biti prekinut zadatku. Zadaci oslobađaju resurse isključivo kao normalan događaj. Naši **Filozofi** su pristojni i ne otimaju **Stapic**e jedan od drugog.
4. Može se javiti kružno čekanje, pri kojem prvi zadatak čeka na resurs koji koristi drugi zadatak, drugi čeka na resurs koji koristi treći zadatak itd., dok jedan od zadataka čeka na resurs koji koristi prvi i time zatvara uzajamnu blokadu. U programu **UzajamnoBlokiraniFilozofiKojiVeceraju.java**, kružno čekanje nastaje zato što svaki **Filozof** najpre pokušava da uzme **Stapic** s desne strane, a zatim onaj sleva.

Pošto *svi* ovi uslovi moraju biti ispunjeni da bi se ostvarila uzajamna blokada, dovoljno je da sprečite samo jedan od njih i izbegli ste uzajamnu blokadu. U ovom programu, najlakši način sprečavanja uzajamne blokade jeste rušenje četvrtog uslova. Taj uslov je ispunjen zato što svaki **Filozof** pokušava da uzme svoje **Stapic**e po određenom redosledu: prvo desni, onda levi. Zato je moguća situacija u kojoj svaki od njih drži svoj desni **Stapic** i čeka da uzme levi, a to je uslov za kružno čekanje. Međutim, kada bi poslednji **Filozof** bio inicijalizovan tako da prvo pokuša da uzme levi štapić, a tek onda desni, taj **Filozof** ne bi mogao da spreči **Filozof**a sa svoje desne strane da uzme štapić između njih. Time bi kružno čekanje bilo sprečeno. To je samo jedno rešenje problema, a rešili biste ga i sprečavanjem ispunjenja bilo kojeg od preostalih uslova. (Više o tome pročitajte u nekoj od knjiga za naprednije višenitno programiranje):

```
//: paralelno/PopravljeniFilozofiKojiVeceraju.java
// Filozofi koji večeraju bez uzajamne blokade.
// {Args: 5 5 odmor}
import java.util.concurrent.*;

public class PopravljeniFilozofiKojiVeceraju {
  public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
      ponder = Integer.parseInt(args[0]);
    int broj = 5;
    if(args.length > 1)
      broj = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Stapic[] stapici = new Stapic[broj];
    for(int i = 0; i < broj; i++)
      stapici[i] = new Stapic();
```

```
    for(int i = 0; i < broj; i++)
      if(i < (broj-1))
        exec.execute(new Filozof(
          stapici[i], stapici[i+1], i, ponder));
      else
        exec.execute(new Filozof(
          stapici[1], stapici[i], i, ponder));
    if(args.length == 3 && args.equals("odmor"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Pritisnite 'Enter' ako želite da prekinete
      izvršavanje programa");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Uzajamnu blokadu smo uklonili tako što poslednji **Filozof** sada uzima i pušta levi **Stapic** pre desnog, i program će raditi dobro.

Java nema podršku za sprečavanje uzajamne blokade; morate je sami izbeći tako što ćete pažljivije projektovati program. To nisu utešne reči za onoga ko pokušava da otkrije i otkloni grešku iz programa koji upada u uzajamnu blokadu.

Vežba 31: (8) Izmenite program **UzajamnoBlokiraniFilozofiKojiVeceraju.java** na sledeći način: kada filozof upotrebi štapiće, neka ih ne spušta na sto nego u zdelu. Kada filozof hoće da jede, uzima prva dva dostupna štapića iz te zdele. Da li je time izbegnuta mogućnost uzajamne blokade? Da li ćete prostim smanjenjem broja dostupnih štapića ponovo uvesti mogućnost uzajamne blokade?

Nove komponente biblioteke

U biblioteci **java.util.concurrent** u Javi SE5 postoji znatan broj novih klasa za rešavanje problema paralelnog izvršavanja. Ako naučite da ih koristite, to će vam pomoći da pišete jednostavnije i robusnije programe za paralelno izvršavanje.

U ovom odeljku prikazaću reprezentativan uzorak primera raznih komponenata, ali nekoliko njih – one koje se ređe sreću i koriste – ovde neće biti razmotrene.

Pošto ove komponente služe za rešavanje raznih problema, ne postoji jasan način da ih organizujemo, pa ću krenuti od jednostavnijih primera ka sve složenijim.

CountDownLatch – brava sa odbrojavanjem

Koristi se za sinhronizaciju jednog ili više zadataka tako što ih primorava da čekaju na dovršenje skupa operacija koje obavljaju drugi zadaci.

Najpre **CountDownLatch** objektu date početnu vrednost. Svaki zadatak koji za taj objekat pozove metodu **await**( ), blokira se dok se ta vrednost ne izjednači s nulom. Ostali zadaci mogu pozivati metodu **countDown**( ) za taj objekat i tako smanjivati tu vrednost;

to odbrojavanje se obično radi kada zadatak obavi svoj posao. **CountDownLatch** se koristi jednokratno; vrednost se ne može resetovati. Kada vam zatreba verzija koja resetuje vrednost, upotrebite **CyclicBarrier**.

Zadaci koji pozovu **countDown( )** ne blokiraju se zbog tog poziva. Blokiran je samo poziv metode **await( )** dok se vrednost ne izjednači s nulom.

Ova klasa se obično koristi tako što se problem podeli na *n* nezavisno rešivih zadataka i napravi objekat tipa **CountDownLatch** kojem se dodeli početna vrednost *n*. Svaki završen zadatak poziva **countDown( )** za taj objekat. Zadaci koji čekaju da se problem reši pozivaju **await( )** za taj objekat, da bi sačekali dok problem bude rešen. Tu tehniku prikazuje sledeći primer:

```
//: paralelno/CountDownLatchPrimer.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Obavlja deo nekog zadatka:
class DeoZadatka implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private static Random slucajan = new Random(47);
  private final CountDownLatch brava;
  DeoZadatka(CountDownLatch brava) {
    this.brava = brava;
  }
  public void run() {
    try {
      radiStaTreba();
      brava.countDown();
    } catch(InterruptedException izz) {
      // Prihvatljiv način čekanja
    }
  }
  public void radiStaTreba() throws InterruptedException {
    TimeUnit.MILLISECONDS.sleep(slucajan.nextInt(2000));
    print(this + "završen");
  }
  public String toString() {
    return String.format("%1$-3d ", id);
  }
}

// Čeka na CountDownLatch:
class ZadatakKojiCeka implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private final CountDownLatch brava;
  ZadatakKojiCeka(CountDownLatch brava) {
    this.brava = brava;
```

```
  }
  public void run() {
    try {
      brava.await();
      print(this + " prošao barijeru brave");
    } catch(InterruptedException izz) {
      print(this + " prekinut");
    }
  }
  public String toString() {
    return String.format("ZadatakKojiCeka %1$-3d ", id);
  }
}

public class CountDownLatchPrimer {
  static final int VELICINA = 100;
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    // Svi moraju deliti isti objekat tipa CountDownLatch:
    CountDownLatch brava = new CountDownLatch(VELICINA);
    for(int i = 0; i < 10; i++)
      exec.execute(new ZadatakKojiCeka(brava));
    for(int i = 0; i < VELICINA; i++)
      exec.execute(new DeoZadatka(brava));
    print("Svi zadaci pokrenuti");
    exec.shutdown(); // Ugasi kada se svi zadaci završe
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

DeoZadatka spava tokom nasumično odabranog vremenskog razdoblja i tako simulira izvršavanje dela zadatka, a **ZadatakKojiCeka** predstavlja deo sistema koji mora da sačeka dok se prvi deo problema ne reši. Svi zadaci rade sa istim objektom tipa **CountDownLatch** definisanim u metodi **main**( ).

Vežba 32: (7) Upotrebite **CountDownLatch** za korelaciju rezultata raznih **Ulaz**a u primeru **UkrasnaBasta.java**. Iz nove verzije primera uklonite nepotreban kôd.

Bezbednost niti iz biblioteke

Obratite pažnju na to da **DeoZadatka** sadrži statičan objekat tipa **Random**, što znači da bi više zadataka moglo istovremeno da pozove **Random.nextInt**( ). Da li je to bezbedno?

Ako problem postoji, u ovom slučaju ga možete rešiti tako što ćete objektu **DeoZadatka** dati sopstveni **Random** objekat, tj. ukloniti modifikator **static**. Ali ostaje opšte pitanje o svim standardnim metodama iz Javine biblioteke: koje su bezbedne u višenitnom radu, a koje nisu?

Nažalost, u dokumentaciji JDK ne postoji odgovor. Ispostavlja se da **Random.nextInt**( ) *jeste* bezbedna u višenitnom izvršavanju, ali što se tiče drugih metoda, moraćete to sami da otkrivate od slučaja do slučaja, bilo pretraživanjem Weba bilo pregledanjem koda Java biblioteke. To i nije baš pohvalno za programski jezik koji je, barem u teoriji, projektovan upravo zato da podrži paralelno izvršavanje.

CyclicBarrier

Klasu **CyclicBarrier** koristite u situacijama kada hoćete da napravite grupu zadataka koji se izvršavaju paralelno, a zatim da sačekate dok se svi oni ne završe pre nego što pređete na sledeći korak (nešto kao **join( )**, recimo). Prilikom prolaska kroz barijeru svi paralelni zadaci su završeni, pa napred možete da krenete sa rezultatima svih njih. To je veoma slično klasi **CountDownLatch**, sem što je objekat tipa **CountDownLatch** za jednokratnu upotrebu, dok se objekat tipa **CyclicBarrier** može koristiti više puta.

Simulacije me fasciniraju od prvih susreta s računarom, a paralelno izvršavanje je ključni faktor koji omogućuje simulacije. Prvi program koji sam napisao[22] bila je simulacija: igra s konjskim trakama napisana u BASIC-u, nazvana (zbog ograničene dužine imena datoteka) HOSRAC.BAS. Sledi objektno orijentisana, višenitna verzija tog programa, u kojoj je upotrebljena klasa **CyclicBarrier**:

```
//: paralelno/TrkaKonja.java
// Upotreba klase CyclicBarrier.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Konj implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private int koraka = 0;
  private static Random slucajan = new Random(47);
  private static CyclicBarrier barijera;
  public Konj(CyclicBarrier b) { barijera = b; }
  public synchronized int dajKorake() { return koraka; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          koraka += slucajan.nextInt(3); // Daje 0, 1 ili 2
        }
        barijera.await();
      }
    } catch(InterruptedException e) {
      // Legitiman način izlaska
    } catch(BrokenBarrierException e) {
      // Želimo da dobijemo obaveštenje o ovome
      throw new RuntimeException(e);
    }
  }
  public String toString() { return "Konj " + id + " "; }
  public String staza() {
    StringBuilder s = new StringBuilder();
```

[22] U prvoj godini srednje škole; u učionici je bio teleprinter ASR-33 s modemom brzine 110 boda za akustičnu vezu s računarom HP-1000.

```
    for(int i = 0; i < dajKorake(); i++)
      s.append("*");
    s.append(id);
    return s.toString();
  }
}

public class TrkaKonja {
  static final int DUZINA_STAZE = 75;
  private List<Konj> konji = new ArrayList<Konj>();
  private ExecutorService exec =
    Executors.newCachedThreadPool();
  private CyclicBarrier barijera;
  public TrkaKonja(int nKonja, final int sacekaj) {
    barijera = new CyclicBarrier(nKonja, new Runnable() {
      public void run() {
        StringBuilder s = new StringBuilder();
        for(int i = 0; i < DUZINA_STAZE; i++)
          s.append("="); // Ograda trkališta
        print(s);
        for(Konj konj : konji)
          print(konj.staza());
        for(Konj konj : konji)
          if(konj.dajKorake() >= DUZINA_STAZE) {
            print(konj + "pobedio!");
            exec.shutdownNow();
            return;
          }
        try {
          TimeUnit.MILLISECONDS.sleep(sacekaj);
        } catch(InterruptedException e) {
          print("prekinuto spavanje koje je akcija barijere");
        }
      }
    });
    for(int i = 0; i < nKonja; i++) {
      Konj konj = new Konj(barijera);
      konji.add(konj);
      exec.execute(konj);
    }
  }
  public static void main(String[] args) {
    int nKonja = 7;
    int sacekaj = 200;
    if(args.length > 0) { // Opcioni argument
      int n = new Integer(args[0]);
      nKonja = n > 0 ? n : nKonja;
    }
    if(args.length > 1) { // Opcioni argument
```

```
        int p = new Integer(args[1]);
        sacekaj = p > -1 ? p : sacekaj;
      }
      new TrkaKonja(nKonja, sacekaj);
    }
  } /* (Pokrenite da biste videli rezultat) *///:~
```

Objektu tipa **CyclicBarrier** može biti pridodata „akcija barijere", a to je klasa koja realizuje interfejs **Runnable** i izvršava se automatski kada se (na početku zadat) broj izjednači s nulom – to je druga razlika između klasa **CyclicBarrier** i **CountdownLatch**. Ovde je akcija barijere anonimna klasa koja se daje konstruktoru objekta tipa **CyclicBarrier**.

Probao sam da postignem da svaki konj odštampa kada je stigao na cilj, ali se u tom slučaju redosled prikazivanja menja u zavisnosti od mehanizma za raspodelu procesorskog vremena nitima (zadacima). Klasa **CyclicBarrier** omogućuje svakom konju da radi sve što treba kako bi došao do cilja, a zatim mora da čeka na barijeri dok ne dođu svi ostali konji. Kada svi konji stignu, objekat tipa **CyclicBarrier** automatski poziva svoju akciju barijere, tj. zadatak koji realizuje interfejs **Runnable** i prikazuje konje redom kojim su stigli, kao i ogradu trkališta.

Kada svi zadaci prođu barijeru, ona je automatski spremna za sledeću rundu.

Ako želite da postignete efekat veoma jednostavne animacije, smanjite prozor konzole tako da se vide samo konji.

DelayQueue

Ovo je neograničen **BlockingQueue** (blokirajući red za čekanje) objekata koji realizuju interfejs **Delayed**, tj. svaki je odložen na neko vreme (engl. *delayed*). Objekat može biti uzet iz reda za čekanje tek kada njegovo vreme odlaganja istekne. Red je tako ureden da je čeonom objektu preostalo vreme odlaganja koje se najviše skratilo od početka izvršavanja. Ako se nijedno vreme odlaganja nije skratilo, onda nema čeonog elementa i metoda **poll( )** vraća **null** (zato u taj red za čekanje ne možete stavljati **null** elemente).

U sledećem primeru **Delayed** objekti su i sami zadaci, a **PotrosacOdlozenihZadataka** iz rada za čekanje uzima najhitniji zadatak (onaj kojem se vreme odlaganja najviše skratilo od početka izvršavanja) i izvršava ga. Dakle, **DelayQueue** je varijacija prioritetnog reda za čekanje.

```
//: paralelno/DelayQueuePrimer.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class OdlozeniZadatak implements Runnable, Delayed {
  private static int brojac = 0;
  private final int id = brojac++;
  private final int delta;
  private final long okidac;
  protected static List<OdlozeniZadatak> sekvenca =
```

```
    new ArrayList<OdlozeniZadatak>();
  public OdlozeniZadatak(int odlaganjeUMilisekundama) {
    delta = odlaganjeUMilisekundama;
    okidac = System.nanoTime() +
      NANOSECONDS.convert(delta, MILLISECONDS);
    sekvenca.add(this);
  }
  public long getDelay(TimeUnit jedinica) {
    return jedinica.convert(
      okidac - System.nanoTime(), NANOSECONDS);
  }
  public int compareTo(Delayed arg) {
    OdlozeniZadatak onaj = (OdlozeniZadatak)arg;
    if(okidac < onaj.okidac) return -1;
    if(okidac > onaj.okidac) return 1;
    return 0;
  }
  public void run() { printnb(this + " "); }
  public String toString() {
    return String.format("[%1$-4d]", delta) +
      " Zadatak " + id;
  }
  public String sazetak() {
    return "(" + id + ":" + delta + ")";
  }
  public static class StrazarNaCilju extends OdlozeniZadatak {
    private ExecutorService exec;
    public StrazarNaCilju(int odlaganje, ExecutorService e) {
      super(odlaganje);
      exec = e;
    }
    public void run() {
      for(OdlozeniZadatak iz : sekvenca) {
        printnb(iz.sazetak() + " ");
      }
      print();
      print(this + " poziva shutdownNow()");
      exec.shutdownNow();
    }
  }
}

class PotrosacOdlozenihZadataka implements Runnable {
  private DelayQueue<OdlozeniZadatak> rzc;
  public PotrosacOdlozenihZadataka(DelayQueue<OdlozeniZadatak> rzc) {
    this.rzc = rzc;
  }
  public void run() {
    try {
      while(!Thread.interrupted())
```

```
        rzc.take().run(); // Pokreni zadatak s tekućom niti
      } catch(InterruptedException e) {
        // Prihvatljiv način izlaska
      }
      print("Završen PotrosacOdlozenihZadataka");
    }
  }

  public class DelayQueuePrimer {
    public static void main(String[] args) {
      Random slucajan = new Random(47);
      ExecutorService exec = Executors.newCachedThreadPool();
      DelayQueue<OdlozeniZadatak> redzacekanje =
        new DelayQueue<OdlozeniZadatak>();
      // Popuni zadacima s nasumičnim odlaganjima:
      for(int i = 0; i < 20; i++)
        redzacekanje.put(new OdlozeniZadatak(slucajan.nextInt(5000)));
      // Zadaj tačku zaustavljanja
      redzacekanje.add(new OdlozeniZadatak.StrazarNaCilju(5000, exec));
      exec.execute(new PotrosacOdlozenihZadataka(redzacekanje));
    }
  } /* Ispis:
  [128 ] Zadatak 11 [200 ] Zadatak 7 [429 ] Zadatak 5 [520 ] Zadatak 18
  [555 ] Zadatak 1 [961 ] Zadatak 4 [998 ] Zadatak 16 [1207] Zadatak 9
  [1693] Zadatak 2 [1809] Zadatak 14 [1861] Zadatak 3 [2278] Zadatak 15
  [3288] Zadatak 10 [3551] Zadatak 12 [4258] Zadatak 0 [4258] Zadatak 19
  [4522] Zadatak 8 [4589] Zadatak 13 [4861] Zadatak 17 [4868] Zadatak 6
  (0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868) (7:200)
  (8:4522) (9:1207) (10:3288) (11:128) (12:3551) (13:4589) (14:1809)
  (15:2278) (16:998) (17:4861) (18:520) (19:4258) (20:5000)
  [5000] Zadatak 20 poziva shutdownNow()
  Završen PotrosacOdlozenihZadataka
  *///:~
```

OdlozeniZadatak sadrži listu **List<OdlozeniZadatak>** nazvanu **sekvenca** koja održava redosled kojim su zadaci pravljeni, pa možemo videti da se red za čekanje zaista uređuje.

Interfejs **Delayed** ima metodu **getDelay( )** koja kazuje koliko vremena je preostalo do isteka odlaganja ili koliko je proteklo od isteka odlaganja. Ta metoda nas prisiljava da upotrebimo klasu **TimeUnit**, zato što je njen argument tog tipa. Ispostavlja se da je ta klasa veoma podesna jer omogućuje laku konverziju vremenskih jedinica bez ikakvih proračuna. Primera radi, iznos **delta** se pamti u milisekundama, a metoda Jave SE5 **System.nanoTime( )** daje vreme u nanosekundama. Iznos **delta** konvertujemo tako što saopštimo u kojim jedinicama jeste i u kojim želimo da bude, i to na ovaj način:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

U metodi **getDelay( )**, željena jedinica se prosleđuje kao argument **jedinica** i pomoću nje vreme proteklo od okidača konvertujemo u jedinice koje zahteva pozivalac, a da čak i ne znamo koje su. (Ovo je jednostavan primer projektnog obrasca *Strategy*, u kojem se deo algoritma prosleđuje kao argument.)

Za uređivanje, interfejs **Delayed** nasleđuje i interfejs **Comparable**, pa moramo realizovati metodu **compareTo( )** tako da proizvodi razumna poređenja. Metode **toString( )** i **sazetak( )** formatiraju izlaz, a ugnežđena klasa **StrazarNaCilju** služi za gašenje svega zato što je stavljena u red za čekanje kao njegov poslednji element.

Imajte u vidu da je **PotrosacOdlozenihZadataka** i sam zadatak, pa ima sopstvenu nit (objekat tipa **Thread**) u kojoj može da izvršava svaki zadatak koji izađe iz reda za čekanje. Pošto se zadaci izvršavaju redom prioriteta reda za čekanje, u ovom primeru nema potrebe da pravimo zasebne niti za izvršavanje odloženih zadataka (objekata tipa **OdlozeniZadatak**).

Iz rezultata vidite da redosled pravljenja zadataka nema uticaja na redosled izvršavanja, nego se zadaci izvršavaju redosledom kojim im ističe odlaganje, kao što smo očekivali.

PriorityBlockingQueue

U suštini, ovo je prioritetni red za čekanje čije se operacije vađenja iz reda blokiraju. U narednom primeru, objekti u prioritetnom redu za čekanje su zadaci koji se iz reda za čekanje pojavljuju po redosledu njihovih prioriteta. **ZadatakSPrioritetom** dobija broj prioriteta iz kojeg sledi taj redosled:

```
//: paralelno/PriorityBlockingQueuePrimer.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class ZadatakSPrioritetom implements
Runnable, Comparable<ZadatakSPrioritetom>  {
  private Random slucajan = new Random(47);
  private static int brojac = 0;
  private final int id = brojac++;
  private final int prioritet;
  protected static List<ZadatakSPrioritetom> sekvenca =
    new ArrayList<ZadatakSPrioritetom>();
  public ZadatakSPrioritetom(int prioritet) {
    this.prioritet = prioritet;
    sekvenca.add(this);
  }
  public int compareTo(ZadatakSPrioritetom arg) {
    return prioritet < arg.prioritet ? 1 :
      (prioritet > arg.prioritet ? -1 : 0);
  }
  public void run() {
    try {
      TimeUnit.MILLISECONDS.sleep(slucajan.nextInt(250));
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska
    }
    print(this);
  }
```

```
    public String toString() {
      return String.format("[%1$-3d]", prioritet) +
        " Zadatak " + id;
    }
    public String sazetak() {
      return "(" + id + ":" + prioritet + ")";
    }
    public static class StrazarNaCilju extends ZadatakSPrioritetom {
      private ExecutorService exec;
      public StrazarNaCilju(ExecutorService e) {
        super(-1); // Najniži prioritet u ovom programu
        exec = e;
      }
      public void run() {
        int broj = 0;
        for(ZadatakSPrioritetom iz : sekvenca) {
          printnb(iz.sazetak());
          if(++broj % 5 == 0)
            print();
        }
        print();
        print(this + " Poziva shutdownNow()");
        exec.shutdownNow();
      }
    }
  }

  class ProizvodjacZadatakaSPrioritetom implements Runnable {
    private Random slucajan = new Random(47);
    private Queue<Runnable> redzacekanje;
    private ExecutorService exec;
    public ProizvodjacZadatakaSPrioritetom(
      Queue<Runnable> rzc, ExecutorService e) {
      redzacekanje = rzc;
      exec = e; // Upotrebljava se za StrazarNaCilju
    }
    public void run() {
      // Neograničen red za čekanje; nikada se ne blokira.
      // Popuniću brzo nasumičnim prioritetima:
      for(int i = 0; i < 20; i++) {
        redzacekanje.add(new ZadatakSPrioritetom(slucajan.nextInt(10)));
        Thread.yield();
      }
      // Ubacujem postepeno poslove najvišeg prioriteta:
      try {
        for(int i = 0; i < 10; i++) {
          TimeUnit.MILLISECONDS.sleep(250);
          redzacekanje.add(new ZadatakSPrioritetom(10));
        }
        // Dodajem poslove, najpre one s najnižim prioritetom:
```

```
      for(int i = 0; i < 10; i++)
        redzacekanje.add(new ZadatakSPrioritetom(i));
      // Oznaka kraja koja zaustavlja sve zadatke:
      redzacekanje.add(new ZadatakSPrioritetom.StrazarNaCilju(exec));
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska
    }
    print("Završen ProizvodjacZadatakaSPrioritetom");
  }
}

class PotrosacZadatakaSPrioritetom implements Runnable {
  private PriorityBlockingQueue<Runnable> rzc;
  public PotrosacZadatakaSPrioritetom(
    PriorityBlockingQueue<Runnable> rzc) {
    this.rzc = rzc;
  }
  public void run() {
    try {
      while(!Thread.interrupted())
        // Za izvršavanje zadatka upotrebi tekuću nit:
        rzc.take().run();
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska
    }
    print("Završen PotrosacZadatakaSPrioritetom");
  }
}

public class PriorityBlockingQueuePrimer {
  public static void main(String[] args) throws Exception {
    Random slucajan = new Random(47);
    ExecutorService exec = Executors.newCachedThreadPool();
    PriorityBlockingQueue<Runnable> redzacekanje =
      new PriorityBlockingQueue<Runnable>();
    exec.execute(new ProizvodjacZadatakaSPrioritetom(redzacekanje,
      exec));
    exec.execute(new PotrosacZadatakaSPrioritetom(redzacekanje));
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Kao u prethodnom primeru, redosled pravljenja objekata tipa **ZadatakSPrioritetom** pamti se u listi **sekvenca**, radi poređenja sa stvarnim redosledom izvršavanja. Metoda **run**( ) spava kratko, nasumično odabrano vreme i štampa informacije o objektu, a **StrazarNaCilju** ima funkciju kao pre i jemči da je poslednji objekat u redu.

ProizvodjacZadatakaSPrioritetom i **PotrosacZadatakaSPrioritetom** međusobno su povezani preko objekta tipa **PriorityBlockingQueue**. Pošto blokirajući red za čekanje obavlja svu potrebnu sinhronizaciju, eksplicitna sinhronizacija nije neophodna – ne morate da mislite ima li red za čekanje ijedan element kada pokušate da ga čitate, pošto će on blokirati čitaoca kada mu ponestane elemenata.

Kontroler staklenika napravljen pomoću **ScheduledExecutor**a

U poglavlju *Unutrašnje klase* predstavljen je primer hipotetičkog staklenika i njegovog upravljačkog sistema koji uključuje, isključuje i podešava opremu. To se može posmatrati kao jedna vrsta problema paralelnog izvršavanja, pri kome je svaki željeni događaj u stakleniku zadatak koji se izvršava u unapred definisano vreme. Klasa **ScheduledThreadPoolExecutor** ima sve što treba za rešavanje ovog problema. Metodama **schedule( )** (koja zadatak izvršava jednom) ili **scheduleAtFixedRate( )** (koja izvršavanje zadatka ponavlja u pravilnim intervalima) pripremate objekte koji realizuju interfejs **Runnable** za izvršavanje u određeno buduće vreme. Uporedite naredni primer i primer iz poglavlja *Unutrašnje klase* da biste videli koliko je sve jednostavnije kada je na raspolaganju gotova alatka kao što je **ScheduledThreadPoolExecutor**:

```
//: paralelno/RasporedjivacStaklenika.java
// Prerada programa unutrasnjeklase/KontrolerStaklenika.java
// u kojoj je upotrebljen ScheduledThreadPoolExecutor.
// {Args: 5000}
import java.util.concurrent.*;
import java.util.*;

public class RasporedjivacStaklenika {
  private volatile boolean svetlo = false;
  private volatile boolean voda = false;
  private String termostat = "Dan";
  public synchronized String dajTermostat() {
    return termostat;
  }
  public synchronized void podesiTermostat(String iznos) {
    termostat = iznos;
  }
  ScheduledThreadPoolExecutor rasporedjivac =
    new ScheduledThreadPoolExecutor(10);
  public void raspored(Runnable dogadjaj, long odlaganje) {
    rasporedjivac.raspored(dogadjaj,odlaganje,TimeUnit.MILLISECONDS);
  }
  public void
  ponavljaj(Runnable dogadjaj, long pocetnoOdlaganje, long period) {
    rasporedjivac.scheduleAtFixedRate(
      dogadjaj, pocetnoOdlaganje, period, TimeUnit.MILLISECONDS);
  }
  class UkljuciSvetlo implements Runnable {
    public void run() {
      // Ovde napišite kôd za upravljanje
      // hardverom koji fizički pali svetlo.
      System.out.println("Uključujem svetlo");
      svetlo = true;
    }
  }
```

```
class IskljuciSvetlo implements Runnable {
  public void run() {
    // Ovde napišite kōd za upravljanje hardverom koji fizički gasi svetlo.
    System.out.println("Gasim svetlo");
    svetlo = false;
  }
}
class UkljuciVodu implements Runnable {
  public void run() {
    // Ovde napišite kōd za upravljanje hardverom.
    System.out.println("Uključujem vodu u stakleniku");
    voda = true;
  }
class IskljuciVodu implements Runnable {  }
  public void run() {
    // Ovde napišite kōd za upravljanje hardverom.
    System.out.println("Isključujem vodu u stakleniku");
    voda = false;
  }
}
class TermostatNoc implements Runnable {
  public void run() {
    // Ovde napišite kōd za upravljanje hardverom.
    System.out.println("Termostat podešen za noć");
    podesiTermostat("Noć");
  }
}
class TermostatDan implements Runnable {
  public void run() {
    // Ovde napišite kōd za upravljanje hardverom.
    System.out.println("Termostat podešen za dan");
    podesiTermostat("Dan");
  }
}
class Zvono implements Runnable {
  public void run() { System.out.println("Zvrc!"); }
}
class Gasi implements Runnable {
  public void run() {
    System.out.println("Gasim");
    rasporedjivac.shutdownNow();
    // Moram da pokrenem zaseban zadatak da ovo obavi,
    // pošto je raspoređivač ugašen:
    new Thread() {
      public void run() {
        for(RadnaTacka d : podaci)
          System.out.println(d);
      }
    }.start();
  }
}
```

```
// Novina: prikupljanje podataka
static class RadnaTacka {
  final Calendar vreme;
  final float temperatura;
  final float vlaznost;
  public RadnaTacka(Calendar d, float privr, float vlag) {
    vreme = d;
    temperatura = privr;
    vlaznost = vlag;
  }
  public String toString() {
    return vreme.getTime() +
      String.format(
        " temperatura: %1$.1f vlaznost: %2$.2f",
        temperatura, vlaznost);
  }
}
private Calendar poslednjiPut = Calendar.getInstance();
{ // Podesi vreme na pola punog sata
  poslednjiPut.set(Calendar.MINUTE, 30);
  poslednjiPut.set(Calendar.SECOND, 00);
}
private float poslednjaTemp = 65.0f;
private int smerTemp = +1;
private float poslednjaVlaznost = 50.0f;
private int smerVlaznosti = +1;
private Random slucajan = new Random(47);
List<RadnaTacka> podaci = Collections.synchronizedList(
  new ArrayList<RadnaTacka>());
class PrikupiPodatke implements Runnable {
  public void run() {
    System.out.println("Prikupljanje podataka");
    synchronized(RasporedjivacStaklenika.this) {
      // Praviću se da je interval duži nego što zapravo jeste:
      poslednjiPut.set(Calendar.MINUTE,
        poslednjiPut.get(Calendar.MINUTE) + 30);
      // Verovatnoća promene smera 1/5:
      if(slucajan.nextInt(5) == 4)
        smerTemp = -smerTemp;
      // Sačuvaj prethodni iznos:
      poslednjaTemp = poslednjaTemp +
        smerTemp * (1.0f + slucajan.nextFloat());
      if(slucajan.nextInt(5) == 4)
        smerVlaznosti = -smerVlaznosti;
      poslednjaVlaznost = poslednjaVlaznost +
        smerVlaznosti * slucajan.nextFloat();
      // Kalendar mora biti kloniran, inače bi sve
      // RadneTačke imale reference na isti poslednjiPut.
      // Za jednostavan objekat kao što je Calendar,
      // metoda clone() je dovoljno dobra.
```

```
          podaci.add(new RadnaTacka((Calendar)poslednjiPut.clone(),
            poslednjaTemp, poslednjaVlaznost));
        }
      }
    }
    public static void main(String[] args) {
      RasporedjivacStaklenika st = new RasporedjivacStaklenika();
      st.raspored(st.new Gasi(), 5000);
      // Ranije klase "Restart" nisu potrebne:
      st.ponavljaj(st.new Zvono(), 0, 1000);
      st.ponavljaj(st.new TermostatNoc(), 0, 2000);
      st.ponavljaj(st.new UkljuciSvetlo(), 0, 200);
      st.ponavljaj(st.new IskljuciSvetlo(), 0, 400);
      st.ponavljaj(st.new UkljuciVodu(), 0, 600);
      st.ponavljaj(st.new IskljuciVodu(), 0, 800);
      st.ponavljaj(st.new TermostatDan(), 0, 1400);
      st.ponavljaj(st.new PrikupiPodatke(), 500, 500);
    }
  } /* (Pokrenite da biste videli rezultat) *///:~
```

Ova verzija reorganizuje kôd i dodaje jednu novinu: prikupljanje merenja temperature i vlažnosti u stakleniku. **RadnaTacka** sadrži i prikazuje jedan skup merenja, dok je **PrikupiPodatke** planiran zadatak koji generiše simulirane podatke i dodaje ih listi **List<RadnaTacka>** u **Stakleniku** nakon svakog svog izvršavanja.

Obratite pažnju na to da se modifikatori **volatile** i **synchronized** upotrebljavaju na odgovarajućim mestima kako bi se sprečilo da zadaci smetaju jedan drugom. Prilikom pravljenja liste koja sadrži **RadneTačke**, sve metode bivaju sinhronizovane uslužnom metodom **synchronizedList( )** iz biblioteke **java.util.Collections**.

Vežba 33: (7) Izmenite **RasporedjivacStaklenika.java** tako da koristi **DelayQueue** umesto **ScheduledExecutor**a.

Semafor

Obična brava (eksplicitna iz **concurrent.locks** ili ugrađena **synchronized**) u svakom trenutku dozvoljava samo jednom zadatku da pristupi svakom resursu. *Semafor brojač* omogućuje da *n* zadataka istovremeno pristupa jednom resursu. Možete smatrati da semafor izdaje dozvole za korišćenje odredenog resursa, mada se zapravo ne upotrebljavaju nikakvi objekti dozvola.

Kao primer, razmotrićemo pojam *grupe objekata* koja upravlja ograničenim brojem objekata tako što dozvoljava da budu pojedinačno izdati na upotrebu i potom vraćeni kada korisnik završi svoj posao s njima. Te funkcije ćemo kapsulirati u generičkoj klasi:

```
//: paralelno/Grupa.java
// Semafor u grupi objekata koji ograničava broj zadataka
// koji istovremeno mogu koristiti resurs.
import java.util.concurrent.*;
import java.util.*;
```

```
public class Grupa<T> {
  private int velicina;
  private List<T> stavke = new ArrayList<T>();
  private volatile boolean[] izdat;
  private Semaphore dostupan;
  public Grupa(Class<T> classObjekat, int velicina) {
    this.velicina = velicina;
    izdat = new boolean[velicina];
    dostupan = new Semaphore(velicina, true);
    // Popuni grupu objektima koji mogu biti izdati:
    for(int i = 0; i < velicina; ++i)
      try {
        // Pretpostavlja da se koristi podrazumevani konstruktor:
        stavke.add(classObjekat.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
  public T izdajObjekatNaKoriscenje() throws InterruptedException {
    dostupan.acquire();
    return uzmiStavku();
  }
  public void primiObjekatNazadUGrupu(T x) {
    if(pustiStavku(x))
      dostupan.release();
  }
  private synchronized T uzmiStavku() {
    for(int i = 0; i < velicina; ++i)
      if(!izdat[i]) {
        izdat[i] = true;
        return stavke.get(i);
      }
    return null; // Semafor ne dozvoljava pristup ovoj tački
  }
  private synchronized boolean pustiStavku(T stavka) {
    int indeks = stavke.indexOf(stavka);
    if(indeks == -1) return false; // Nije u listi
    if(izdat[indeks]) {
      izdat[indeks] = false;
      return true;
    }
    return false; // Nije bio izdat
  }
} ///:~
```

U ovom pojednostavljenom obliku, konstruktor metodom **newInstance**( ) popunjava grupu objektima. Kada vam zatreba nov objekat, pozovete metodu **izdajObjekatNaKoriscenje**( ), a kada vam objekat više nije potreban, prosledite ga metodi **primiObjekatNazadUGrupu**( ).

Niz **izdat** tipa **boolean** prati objekte koji su izdati na korišćenje, a njime upravljaju metode **uzmiStavku( )** i **pustiStavku( )**. Njih pak čuva semafor (objekat tipa **Semaphore**) **dostupan**, tako da u metodi **izdajObjekatNaKoriscenje( )**, **dostupan** blokira poziv ukoliko više nema dostupnih dozvola za korišćenje objekata iz grupe (što znači da u grupi više nema objekata). Metoda **primiObjekatNazadUGrupu( )** vraća dozvolu semaforu ukoliko je objekat koji se vraća validan.

Kao primer upotrebićemo **Debeli**, tip objekta koji je skupo praviti zato što njegovom konstruktoru treba mnogo vremena da obavi svoj posao:

```
//: paralelno/Debeli.java
// Objekti koje je skupo praviti.

public class Debeli {
  private volatile double d; // Spreči optimizaciju
  private static int brojac = 0;
  private final int id = brojac++;
  public Debeli() {
    // Skupa operacija koja se može prekinuti:
    for(int i = 1; i < 10000; i++) {
      d += (Math.PI + Math.E) / (double)i;
    }
  }
  public void operacija() { System.out.println(this); }
  public String toString() { return "Debeli id: " + id; }
} ///:~
```

Okupićemo ove objekte u grupu da bismo ograničili uticaj tog konstruktora. Klasu **Grupa** ćemo testirati pravljenjem zadatka koji objekte tipa **Debeli** uzima na korišćenje, drži ih neko vreme i zatim vraća nazad:

```
//: paralelno/PrimerSemafora.java
// Testiranje klase Grupa
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Zadatak koji uzima resurs iz grupe:
class ZadatakKojiUzimaResurs<T> implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private Grupa<T> grupa;
  public ZadatakKojiUzimaResurs(Grupa<T> grupa) {
    this.grupa = grupa;
  }
  public void run() {
    try {
      T stavka = grupa.izdajObjekatNaKoriscenje();
      print(this + "izdat " + stavka);
      TimeUnit.SECONDS.sleep(1);
```

```
        print(this +"vraćen " + stavka);
        grupa.primiObjekatNazadUGrupu(stavka);
      } catch(InterruptedException e) {
        // Prihvatljiv način izlaska
      }
    }
    public String toString() {
      return "ZadatakKojiUzimaResurs " + id + " ";
    }
  }

  public class PrimerSemafora {
    final static int VELICINA = 25;
    public static void main(String[] args) throws Exception {
      final Grupa<Debeli> grupa =
        new Grupa<Debeli>(Debeli.class, VELICINA);
      ExecutorService exec = Executors.newCachedThreadPool();
      for(int i = 0; i < VELICINA; i++)
        exec.execute(new ZadatakKojiUzimaResurs<Debeli>(grupa));
      print("Napravljeni svi ZadaciKojiUzimajuResurs ");
      List<Debeli> lista = new ArrayList<Debeli>();
      for(int i = 0; i < VELICINA; i++) {
        Debeli f = grupa.izdajObjekatNaKoriscenje();
        printnb(i + ": nit main() izdata na korišćenje ");
        f.operacija();
        lista.add(f);
      }
      Future<?> blokiran = exec.submit(new Runnable() {
        public void run() {
          try {
            // Semafor sprečava daljnje izdavanje,
            // pa se poziv blokira:
            grupa.izdajObjekatNaKoriscenje();
          } catch(InterruptedException e) {
            print("izdajObjekatNaKoriscenje() prekinuta");
          }
        }
      });
      TimeUnit.SECONDS.sleep(2);
      blokiran.cancel(true); // Izađi iz blokiranog poziva
      print("Vraćanje objekata u " + lista);
      for(Debeli f : lista)
        grupa.primiObjekatNazadUGrupu(f);
      for(Debeli f : lista)
        grupa.primiObjekatNazadUGrupu(f); // Drugi primiObjekatNazadUGrupu
                                          // ignorišem
      exec.shutdown();
    }
  } /* (Pokrenite da biste videli rezultat) *///:~
```

U metodi **main( )**, pravi se **Grupa** objekata tipa **Debeli** i skup **ZadatakaKojiUzimaju-Resurs** koji počinje da je upotrebljava. Zatim nit **main( )** počinje da uzima objekte tipa **Debeli** *i da ih ne vraća natrag.* Kada uzme sve objekte iz grupe, **semafor** neće dozvoliti dalje izdavanje objekata na korišćenje. Zato se blokira metoda **run( )** objekta **blokiran**, i nakon dve sekunde poziva se metoda **cancel( )** da izađe iz tog objekta tipa **Future**. Obratite pažnju na to da **Grupa** ignoriše redundantna vraćanja objekata.

U ovom primeru računao sam na to da će se klijent **Grupe** uvek setiti da dobrovoljno vrati objekte u grupu, što je najjednostavnije rešenje kada funkcioniše. Ukoliko ne možete da se oslonite na to, knjiga *Thinking in Patterns* (na adresi *www.MindView.net*) sadrži dalja istraživanja načina upravljanja objektima izdatim na korišćenje iz grupe.

Exchanger

Exchanger (razmenjivač) jeste barijera koja omogućava da se međusobno zamene objekti dvaju zadataka. Kada zadaci uđu u barijeru, imaju svaki po jedan objekat, a kada izađu, imaju objekat koji je prethodno imao onaj drugi zadatak. Ovakvi razmenjivači se obično koriste kada jedan zadatak pravi objekte čija je proizvodnja skupa, a drugi zadatak ih troši; na taj način, može se napraviti više objekata koji se istovremeno troše.

Da bismo upotrebili klasu **Exchanger**, napravićemo zadatke proizvođača i potrošača koji preko generičkih tipova i **Generatora** rade sa svim vrstama objekata, i zatim ćemo ih primeniti na klasu **Debeli**. Klase **RazmenjivacProizvodjac** i **RazmenjivacPotrosac** koriste objekat tipa **List<T>** za razmenu; svaka od njih sadrži **Exchanger** za ovu listu **List<T>**. Kada pozovete metodu **Exchanger.exchange( )**, ona blokira zadatak dok partnerski zadatak ne pozove svoju metodu **exchange( )**. Kada obe metode **exchange( )** završe svoj posao, lista tipa **List<T>** biće zamenjena:

```
//: paralelno/ExchangerPrimer.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class RazmenjivacProizvodjac<T> implements Runnable {
  private Generator<T> generator;
  private Exchanger<List<T>> razmenjivac;
  private List<T> spremnik;
  RazmenjivacProizvodjac(Exchanger<List<T>> rapro,
  Generator<T> gen, List<T> spremnik) {
    razmenjivac = rapro;
    generator = gen;
    this.spremnik = spremnik;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        for(int i = 0; i < ExchangerPrimer.velicina; i++)
          spremnik.add(generator.next());
```

```
        // Zameni pun za prazan:
        spremnik = razmenjivac.exchange(spremnik);
      }
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska.
    }
  }
}

class RazmenjivacPotrosac<T> implements Runnable {
  private Exchanger<List<T>> razmenjivac;
  private List<T> spremnik;
  private volatile T vrednost;
  RazmenjivacPotrosac(Exchanger<List<T>> rapot, List<T> spremnik){
    razmenjivac = rapot;
    this.spremnik = spremnik;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        spremnik = razmenjivac.exchange(spremnik);
        for(T x : spremnik) {
          vrednost = x; // Daj vrednost spoljnom svetu
          spremnik.remove(x); // OK za CopyOnWriteArrayList
        }
      }
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska.
    }
    System.out.println("Konačna vrednost: " + vrednost);
  }
}

public class ExchangerPrimer {
  static int velicina = 10;
  static int odlaganje = 5; // Sekundi
  public static void main(String[] args) throws Exception {
    if(args.length > 0)
      velicina = new Integer(args[0]);
    if(args.length > 1)
      odlaganje = new Integer(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Exchanger<List<Debeli>> xc = new Exchanger<List<Debeli>>();
    List<Debeli>
      listaProizvodjaca = new CopyOnWriteArrayList<Debeli>(),
      listaPotrosaca = new CopyOnWriteArrayList<Debeli>();
    exec.execute(new RazmenjivacProizvodjac<Debeli>(xc,
      BasicGenerator.create(Debeli.class), listaProizvodjaca));
    exec.execute(
      new RazmenjivacPotrosac<Debeli>(xc, listaPotrosaca));
```

```
      TimeUnit.SECONDS.sleep(odlaganje);
      exec.shutdownNow();
    }
  } /* Ispis: (primer)
  Konačna vrednost: Debeli id: 29999
  *///:~
```

U metodi **main( )**, pravi se jedan **Exchanger** koji koriste oba zadatka, i dve liste tipa **CopyOnWriteArrayList** za razmenu. Ova varijanta objekta tipa **List** toleriše pozivanje metode **remove( )** tokom prolaska kroz listu (ne baca **ConcurrentModificationException**). **RazmenjivacProizvodjac** popunjava jednu listu, zatim zamenjuje punu listu praznom koju mu prosleđuje **RazmenjivacPotrosac**. Klasa **Exchanger** čini da popunjavanje jedne liste i trošenje druge mogu da se odvijaju istovremeno.

Vežba 34: (1) Izmenite **ExchangerPrimer.java** tako da umesto klase **Debeli** upotrebite neku svoju.

Simulacija

Jedna od najzanimljivijih i najuzbudljivijih primena paralelnog izvršavanja jeste pravljenje simulacija. Zbog paralelnosti, svaka komponenta simulacije može biti zaseban zadatak, a simulaciju je zato mnogo lakše programirati. Mnoge video igrice i CGI animacije u filmovima su simulacije, a i ranije prikazani programi **TrkaKonja.java** i **RasporedjivacStaklenika.java** mogu se smatrati simulacijama.

Simulacija šalterskog službenika

Ova klasična simulacija predstavlja svaku situaciju u kojoj se objekti pojavljuju nasumično i za čiju je obradu – koju obavlja ograničen broj poslužilaca, potrebna nasumična količina vremena. Možemo napraviti simulaciju za izračunavanje idealnog broja poslužilaca.

U ovom primeru, potrebno je da se svakom klijentu banke posveti određeno vreme, a to je broj vremenskih jedinica koje službenik mora da potroši na klijenta da bi ga uslužio. Količina vremena za pružanje usluge razlikuje se za svakog klijenta i utvrdićemo je nasumično. Sem toga, ne znamo koliko klijenata dolazi u svakom vremenskom intervalu, pa ćemo i to utvrditi nasumično.

```
//: paralelno/SimulacijaSalterskogSluzbenika.java
// Pomoću redova za čekanje i višenitnog izvršavanja.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Za objekte namenjene samo za čitanje nije potrebna sinhronizacija:
class Klijent {
  private final int vremeUsluzivanja;
  public Klijent(int tm) { vremeUsluzivanja = tm; }
```

```
    public int dajVremeUsluzivanja() { return vremeUsluzivanja; }
    public String toString() {
      return "[" + vremeUsluzivanja + "]";
    }
  }

  // Nauči red klijenata da se prikaže:
  class RedKlijenata extends ArrayBlockingQueue<Klijent> {
    public RedKlijenata(int najveciRed) {
      super(najveciRed);
    }
    public String toString() {
      if(this.velicina() == 0)
        return "[Prazan]";
      StringBuilder rezultat = new StringBuilder();
      for(Klijent klijent : this)
        rezultat.append(klijent);
      return rezultat.toString();
    }
  }

  // Nasumično dodavanje klijenata u red:
  class GeneratorKlijenata implements Runnable {
    private RedKlijenata klijenti;
    private static Random slucajan = new Random(47);
    public GeneratorKlijenata(RedKlijenata rp) {
      klijenti = rp;
    }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          TimeUnit.MILLISECONDS.sleep(slucajan.nextInt(300));
          klijenti.put(new Klijent(slucajan.nextInt(1000)));
        }
      } catch(InterruptedException e) {
        System.out.println("GeneratorKlijenata prekinut");
      }
      System.out.println("GeneratorKlijenata završava");
    }
  }

  class Sluzbenik implements Runnable, Comparable<Sluzbenik> {
    private static int brojac = 0;
    private final int id = brojac++;
    // Broj klijenata usluženih u ovoj smeni:
    private int usluzenihKlijenata = 0;
    private RedKlijenata klijenti;
    private boolean redKlijenataSeUsluzuje = true;
    public Sluzbenik(RedKlijenata rp) { klijenti = rp; }
    public void run() {
```

```
      try {
        while(!Thread.interrupted()) {
          Klijent klijent = klijenti.take();
          TimeUnit.MILLISECONDS.sleep(
            klijent.dajVremeUsluzivanja());
          synchronized(this) {
            usluzenihKlijenata++;
            while(!redKlijenataSeUsluzuje)
              wait();
          }
        }
      } catch(InterruptedException e) {
        System.out.println(this + "prekinut");
      }
      System.out.println(this + "završava");
    }
    public synchronized void radiNestoDrugo() {
      usluzenihKlijenata = 0;
      redKlijenataSeUsluzuje = false;
    }
    public synchronized void usluzujRedKlijenata() {
      assert !redKlijenataSeUsluzuje:"već uslužujem: " + this;
      redKlijenataSeUsluzuje = true;
      notifyAll();
    }
    public String toString() { return "Službenik " + id + " "; }
    public String shortString() { return "T" + id; }
    // Upotrebljava prioritetni red za čekanje:
    public synchronized int compareTo(Sluzbenik drugi) {
      return usluzenihKlijenata < drugi.usluzenihKlijenata ? -1 :
        (usluzenihKlijenata == drugi.usluzenihKlijenata ? 0 : 1);
    }
}

class RukovodilacSluzbenika implements Runnable {
    private ExecutorService exec;
    private RedKlijenata klijenti;
    private PriorityQueue<Sluzbenik> radiSluzbenika =
      new PriorityQueue<Sluzbenik>();
    private Queue<Sluzbenik> sluzbenikaKojiRadeNestoDrugo =
      new LinkedList<Sluzbenik>();
    private int periodPrilagodjavanja;
    private static Random slucajan = new Random(47);
    public RukovodilacSluzbenika(ExecutorService e,
      RedKlijenata klijenti, int periodPrilagodjavanja) {
      exec = e;
      this.klijenti = klijenti;
      this.periodPrilagodjavanja = periodPrilagodjavanja;
      // Počni s jednim službenikom:
      Sluzbenik sluzbenik = new Sluzbenik(klijenti);
```

```
    exec.execute(sluzbenik);
    radiSluzbenika.add(sluzbenik);
  }
  public void prilagodiBrojSluzbenika() {
    // Ovo je zapravo upravljački sistem. Menjanjem brojeva otkrićete
    // nestabilne radne tačke upravljačkog mehanizma.
    // Ako je red predugačak, dodaj još jednog službenika:
    if(klijenti.size() / radiSluzbenika.size() > 2) {
        // Ako su službenici na pauzi ili rade nešto drugo,
        // vrati jednog nazad:
        if(sluzbenikaKojiRadeNestoDrugo.size() > 0) {
          Sluzbenik sluzbenik = sluzbenikaKojiRadeNestoDrugo.remove();
          sluzbenik.usluzujRedKlijenata();
          radiSluzbenika.offer(sluzbenik);
          return;
        }
      // U protivnom napravi (zaposli) novog službenika
      Sluzbenik sluzbenik = new Sluzbenik(klijenti);
      exec.execute(sluzbenik);
      radiSluzbenika.add(sluzbenik);
      return;
    }
    // Ako je red dovoljno kratak, ukloni jednog službenika:
    if(radiSluzbenika.size() > 1 &&
      klijenti.size() / radiSluzbenika.size() < 2)
        premestiJednogSluzbenika();
    // Ako reda uopšte nema, treba nam samo jedan službenik:
    if(klijenti.size() == 0)
      while(radiSluzbenika.size() > 1)
        premestiJednogSluzbenika();
  }
  // Daj službeniku drugi posao ili ga pošalji na pauzu:
  private void premestiJednogSluzbenika() {
    Sluzbenik sluzbenik = radiSluzbenika.poll();
    sluzbenik.radiNestoDrugo();
    sluzbenikaKojiRadeNestoDrugo.offer(sluzbenik);
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(periodPrilagodjavanja);
        prilagodiBrojSluzbenika();
        System.out.print(klijenti + " { ");
        for(Sluzbenik sluzbenik : radiSluzbenika)
          System.out.print(sluzbenik.shortString() + " ");
        System.out.println("}");
      }
    } catch(InterruptedException e) {
      System.out.println(this + "prekinut");
    }
```

```
    System.out.println(this + "završava");
  }
  public String toString() { return "RukovodilacSluzbenika "; }
}

public class SimulacijaSalterskogSluzbenika {
  static final int NAJVECI_DOZVOLJENI_RED = 50;
  static final int PERIOD_PRILAGODJAVANJA = 1000;
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    // Ako je red predugačak, klijenti će otići:
    RedKlijenata klijenti =
      new RedKlijenata(NAJVECI_DOZVOLJENI_RED);
    exec.execute(new GeneratorKlijenata(klijenti));
    // Rukovodilac dodaje i uklanja službenike prema potrebi:
    exec.execute(new RukovodilacSluzbenika(
      exec, klijenti, PERIOD_PRILAGODJAVANJA));
    if(args.length > 0) // Opcioni argument
      TimeUnit.SECONDS.sleep(new Integer(args[0]));
    else {
      System.out.println("Pritisnite 'Enter' ako želite da prekinete
      izvršavanje programa");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* Ispis: (primer)
[429][200][207] { T0 T1 }
[861][258][140][322] { T0 T1 }
[575][342][804][826][896][984] { T0 T1 T2 }
[984][810][141][12][689][992][976][368][395][354] { T0 T1
T2 T3 }
Službenik 2 prekinut
Službenik 2 završava
Službenik 1 prekinut
Službenik 1 završava
RukovodilacSluzbenika prekinut
RukovodilacSluzbenika završava
Službenik 3 prekinut
Službenik 3 završava
Službenik 0 prekinut
Službenik 0 završava
GeneratorKlijenata prekinut
GeneratorKlijenata završava
*///:~
```

Objekti tipa **Klijent** veoma su jednostavni, pošto sadrže samo jedno **final int** polje. Pošto se ti objekti nikada ne menjaju, oni su *samo za čitanje* i ne zahtevaju sinhronizaciju niti modifikator **volatile**. Sem toga, svaki zadatak **Službenik** u svakom trenutku uklanja

samo po jedan objekat tipa **Klijent** iz ulaznog reda i uslužuje ga dok ne završi, pa svakom objektu tipa **Klijent** u svakom trenutku ionako pristupa samo po jedan zadatak.

RedKlijenata, kako mu ime kaže, čine klijenti koji čekaju da ih usluži neki objekat tipa **Službenik**. To je samo objekat tipa **ArrayBlockingQueue** s metodom **toString( )** koja ispisuje rezultate u željenom obliku.

Objektu tipa **RedKlijenata** pridružen je jedan **GeneratorKlijenata** koji u nasumičnim vremenskim intervalima dodaje klijente u red.

Službenik uzima klijente iz objekta tipa **RedKlijenata** i obrađuje ih jednog po jednog, pamteći broj klijenata koje je uslužio tokom određene smene. Može mu se poručiti da **radiNestoDrugo( )** kada nema dovoljno klijenata i **usluzujRedKlijenata( )** kada dođe mnogo klijenata. Službenika koji će biti vraćen na opsluživanje ulaznog reda bira metoda **compareTo( )** koja poredi broj usluženih klijenata da bi prioritetni red za čekanje (**PriorityQueue**) mogao automatski da stavi najmanje opterećenog službenika napred.

RukovodilacSluzbenika upravlja svim aktivnostima. On prati sve službenike i nadgleda šta rade klijenti. Jedna od zanimljivosti ove simulacije jeste pokušaj otkrivanja optimalnog broja službenika za dati dotok klijenata. To možete videti u metodi **prilagodiBrojSluzbenika( )** koja je upravljački sistem za dodavanje i uklanjanje službenika na stabilan način. Svi upravljački sistemi moraju da paze na stabilnost; ako na promenu reaguju prebrzo, izgubiće stabilnost, a ako reaguju presporo, sistem će preći u jedan od ekstrema (red klijenata maksimalne dužine, a ne rade svi dostupni službenici ili su zaposleni svi službenici iako klijenata nema).

Vežba 35: (8) Izmenite program **SimulacijaSalterskogSluzbenika.java** tako da predstavlja Web klijente koji šalju zahteve određenom broju servera. Treba utvrditi opterećenje koje ta grupa servera može da savlada.

Simulacija restorana

Jednostavan primer **Restoran.java** prikazan u prethodnom delu poglavlja obogatiću u ovoj simulaciji dodavanjem više komponenata kao što su objekti tipa **Narudzba** i **DeoObroka**; pored toga, ponovo ću upotrebiti klase **jelovnik** iz poglavlja *Nabrojani tipovi*.

Predstaviću i Java SE5 klasu **SynchronousQueue** što je blokirajući red za čekanje nultog internog kapaciteta, pa svaki poziv metode **put( )** mora da čeka na svoj poziv metode **take( )** i obrnuto. Kao da nekome treba da predate neki predmet, a nema stola na koji biste ga mogli staviti – uspećete jedino u slučaju da ta osoba pruži ka vama ruku da uzme predmet. U ovom primeru, **SynchronousQueue** predstavlja prostor neposredno ispred restoranskog gosta, kako bi se naglasilo da u svakom trenutku može biti posluženo samo jedno jelo.

Ostale klase i funkcije u ovom primeru slede iz strukture programa **Restoran.java** ili predstavljaju prilično neposredno preslikavanje operacija pravog restorana:

```
//: paralelno/restoran2/RestoranSRedovima.java
// {Args: 5}
package concurrency.restoran2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
// Ovo se daje konobaru, koji to predaje kuvaru:
class Narudzba { // (Objekat za prenos podataka)
  private static int brojac = 0;
  private final int id = brojac++;
  private final Gost gost;
  private final Konobar konobar;
  private final Hrana hrana;
  public Narudzba(Gost gst, Konobar knbr, Hrana h) {
    gost = gst;
    konobar = knbr;
    hrana = h;
  }
  public Hrana item() { return hrana; }
  public Gost dajGosta() { return gost; }
  public Konobar dajKonobara() { return konobar; }
  public String toString() {
    return "Narudzba: " + id + " stavka: " + hrana +
      " za: " + gost +
      " servirao: " + konobar;
  }
}

// Ovo se vraća od kuvara:
class DeoObroka {
  private final Narudzba narudzba;
  private final Hrana hrana;
  public DeoObroka(Narudzba nar, Hrana h) {
    narudzba = nar;
    hrana = h;
  }
  public Narudzba dajNarudzbu() { return narudzba; }
  public Hrana dajHranu() { return hrana; }
  public String toString() { return hrana.toString(); }
}

class Gost implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private final Konobar konobar;
  // U svakom trenutku može primiti samo jedan deo obroka:
  private SynchronousQueue<DeoObroka> mestoZaStolom =
    new SynchronousQueue<DeoObroka>();
  public Gost(Konobar k) { konobar = k; }
  public void
  posluzi(DeoObroka deo0) throws InterruptedException {
    // Blokira samo ako gost još uvek
    // jede prethodni deo obroka:
    mestoZaStolom.put(deo0);
  }
```

```
  public void run() {
    for(Jelo jelo : Jelo.values()) {
      Hrana hrana = jelo.randomSelection();
      try {
        konobar.primiNarudzbu(this, hrana);
        // Blokira dok se jelo ne posluži:
        print(this + "jede " + mestoZaStolom.take());
      } catch(InterruptedException e) {
        print(this + "čeka na " +
          jelo + " prekinut");
        break;
      }
    }
    print(this + "pojeo, odlazi");
  }
  public String toString() {
    return "Gost " + id + " ";
  }
}

class Konobar implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private final Restoran restoran;
  BlockingQueue<DeoObroka> primljeneNarudzbe =
    new LinkedBlockingQueue<DeoObroka>();
  public Konobar(Restoran rest) { restoran = rest; }
  public void primiNarudzbu(Gost gst, Hrana hrana) {
    try {
      // Ne bi trebalo da blokira zato što se radi o objektu tipa
      // LinkedBlockingQueue koji nema ograničenu veličinu:
      restoran.narudzbe.put(new Narudzba(gst, this, hrana));
    } catch(InterruptedException e) {
      print(this + " primiNarudzbu prekinuta");
    }
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokira dok se jelo ne pripremi
        DeoObroka deoObroka = primljeneNarudzbe.take();
        print(this + "primio " + deoObroka +
          " što poslužuje gostu " +
          deoObroka.dajNarudzbu().dajGosta());
        deoObroka.dajNarudzbu().dajGosta().posluzi(deoObroka);
      }
    } catch(InterruptedException e) {
      print(this + " prekinut");
    }
```

```
    print(this + " nije u smeni");
  }
  public String toString() {
    return "Konobar " + id + " ";
  }
}

class Kuvar implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  private final Restoran restoran;
  private static Random slucajan = new Random(47);
  public Kuvar(Restoran rest) { restoran = rest; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokira dok ne dobije narudžbu:
        Narudzba narudzba = restoran.narudzbe.take();
        Hrana narucenaStavka = narudzba.item();
        // Vreme za pripremu narudzbe:
        TimeUnit.MILLISECONDS.sleep(slucajan.nextInt(500));
        DeoObroka deoObroka = new DeoObroka(narudzba, narucenaStavka);
        narudzba.dajKonobara().primljeneNarudzbe.put(deoObroka);
      }
    } catch(InterruptedException e) {
      print(this + " prekinut");
    }
    print(this + " nije u smeni");
  }
  public String toString() { return "Kuvar " + id + " "; }
}

class Restoran implements Runnable {
  private List<Konobar> konobari =
    new ArrayList<Konobar>();
  private List<Kuvar> kuvari = new ArrayList<Kuvar>();
  private ExecutorService exec;
  private static Random slucajan = new Random(47);
  BlockingQueue<Narudzba>
    narudzbe = new LinkedBlockingQueue<Narudzba>();
  public Restoran(ExecutorService e, int nKonobara,
    int nKuvara) {
    exec = e;
    for(int i = 0; i < nKonobara; i++) {
      Konobar konobar = new Konobar(this);
      konobari.add(konobar);
      exec.execute(konobar);
    }
    for(int i = 0; i < nKuvara; i++) {
      Kuvar kuvar = new Kuvar(this);
      kuvari.add(kuvar);
```

```
        exec.execute(kuvar);
      }
    }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          // Došao je novi gost; dodeli mu Konobara:
          Konobar knbr = konobari.get(
            slucajan.nextInt(konobari.size()));
          Gost k = new Gost(knbr);
          exec.execute(k);
          TimeUnit.MILLISECONDS.sleep(100);
        }
      } catch(InterruptedException e) {
        print("Restoran prekinut");
      }
      print("Restoran se zatvara");
    }
  }

  public class RestoranSRedovima {
    public static void main(String[] args) throws Exception {
      ExecutorService exec = Executors.newCachedThreadPool();
      Restoran restoran = new Restoran(exec, 5, 2);
      exec.execute(restoran);
      if(args.length > 0) // Opcioni argument
        TimeUnit.SECONDS.sleep(new Integer(args));
      else {
        print("Pritisnite 'Enter' ako želite da prekinete izvršavanje
          programa");
        System.in.read();
      }
      exec.shutdownNow();
    }
  } /* Ispis: (primer)
  Konobar 0 primio PROLECNE_ROLNICE što poslužuje gostu Gost 1
  Gost 1 jede PROLECNE_ROLNICE
  Konobar 3 primio PROLECNE_ROLNICE što poslužuje gostu Gost 0
  Gost 0 jede PROLECNE_ROLNICE
  Konobar 0 primio PLJESKAVICU što poslužuje gostu Gost 1
  Gost 1 jede PLJESKAVICU
  Konobar 3 primio PROLECNE_ROLNICE što poslužuje gostu Gost 2
  Gost 2 jede PROLECNE_ROLNICE
  Konobar 1 primio SUPU što poslužuje gostu Gost 3
  Gost 3 jede SUPU
  Konobar 3 primio LOVACKI_PIR što poslužuje gostu Gost 0
  Gost 0 jede LOVACKI_PIR
  Konobar 0 primio VOCE što poslužuje gostu Gost 1
  ...
  *///:~
```

Jedna od veoma važnih stvari na koje u ovom primeru treba obratiti pažnju jeste upravljanje složenošću tako što su za komunikaciju između zadataka upotrebljeni redovi za čekanje. Već sama ta tehnika uveliko pojednostavljuje postupak paralelnog programiranja tako što invertuje kontrolu: zadaci se ne obraćaju jedan drugom neposredno, nego jedni drugima šalju objekte preko redova za čekanje. Zadatak koji je primio objekat obrađuje ga i tretira ga kao poruku, umesto da ga podvrgava poruci. Ukoliko se budete pridržavali tog načina rada, imaćete mnogo više šanse da pravite robusne paralelne sisteme.

Vežba 36: (10) Izmenite **RestoranSRedovima.java** tako da za svaki sto postoji po jedan objekat tipa **Narudzbenica**. Prepravite klasu **narudzba** u **narudzbenica** i dodajte klasu **Sto**, s više **Gost**iju za stolom.

Raspodela posla

Naredni primer je simulacija koja obuhvata više koncepata iz ovog poglavlja. Zamislite robotizovanu montažnu liniju za automobile. Svaka **Kola** se proizvode u nekoliko faza, počev od pravljenja šasije, preko montaže motora, pogonskog sistema i točkova.

```
//: paralelno/ProizvodnjaKola.java
// Složen primer saradnje zadataka.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Kola {
  private final int id;
  private boolean
    motor = false, pogonskiSistem = false, tockovi = false;
  public Kola(int idn)  { id = idn; }
  // Prazan objekat tipa Kola:
  public Kola()  { id = -1; }
  public synchronized int dajId() { return id; }
  public synchronized void dodajMotor() { motor = true; }
  public synchronized void dodajPogonskiSistem() {
    pogonskiSistem = true;
  }
  public synchronized void dodajTockove() { tockovi = true; }
  public synchronized String toString() {
    return "Kola " + id + " [" + " motor: " + motor
      + " pogonskiSistem: " + pogonskiSistem
      + " tockovi: " + tockovi + " ]";
  }
}

class RedKola extends LinkedBlockingQueue<Kola> {}

class PravljenjeSasije implements Runnable {
  private RedKola redKola;
  private int brojac = 0;
```

```
  public PravljenjeSasije(RedKola rk) { redKola = rk; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(500);
        // Napravi šasiju:
        Kola k = new Kola(brojac++);
        print("Objekat tipa PravljenjeSasije napravljen " + k);
        // Umetni u red
        redKola.put(k);
      }
    } catch(InterruptedException e) {
      print("Prekinuto: PravljenjeSasije");
    }
    print("PravljenjeSasije isključeno");
  }
}

class Monter implements Runnable {
  private RedKola redZaSasiju, redZaZavrsnu;
  private Kola kola;
  private CyclicBarrier barijera = new CyclicBarrier(4);
  private GrupaRobota grupaRobota;
  public Monter(RedKola rk, RedKola rzz, GrupaRobota gr){
    redZaSasiju = gr;
    redZaZavrsnu = rzz;
    grupaRobota = gr;
  }
  public Kola kola() { return kola; }
  public CyclicBarrier barijera() { return barijera; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokira dok se šasija ne napravi:
        kola = redZaSasiju.take();
        // Unajmi robote da rade:
        grupaRobota.unajmi(RobotZaMotor.class, this);
        grupaRobota.unajmi(RobotZaPogonskiSistem.class, this);
        grupaRobota.unajmi(RobotZaTockove.class, this);
        barijera.await(); // Dok roboti ne završe
        // Stavi kola u redZaZavrsnu za dalju obradu
        redZaZavrsnu.put(kola);
      }
    } catch(InterruptedException e) {
      print("Izlazim iz niti Monter pomoću prekida");
    } catch(BrokenBarrierException e) {
      // Želim da dobijem obaveštenje o ovome
      throw new RuntimeException(e);
    }
```

```
      print("Monter isključen");
    }
  }

  class Izvestilac implements Runnable {
    private RedKola redKola;
    public Izvestilac(RedKola gr) { redKola = gr; }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          print(redKola.take());
        }
      } catch(InterruptedException e) {
        print("Izlazim iz niti Izvestilac pomoću prekida");
      }
      print("Izvestilac isključen");
    }
  }

  abstract class Robot implements Runnable {
    private GrupaRobota grupa;
    public Robot(GrupaRobota g) { grupa = g; }
    protected Monter monter;
    public Robot dodeliMontera(Monter monter) {
      this.monter = monter;
      return this;
    }
    private boolean angazuj = false;
    public synchronized void angazuj() {
      angazuj = true;
      notifyAll();
    }
    // Ovaj deo metode run() različit je za svakog robota:
    abstract protected void pruziUslugu();
    public void run() {
      try {
        iskljuciSe(); // Čekaj dok ne budeš potreban
        while(!Thread.interrupted()) {
          pruziUslugu();
          monter.barijera().await(); // Sinhronizacija
          // Ovaj posao je završen...
          iskljuciSe();
        }
      } catch(InterruptedException e) {
        print("Izlazim iz niti " + this + " pomoću prekida");
      } catch(BrokenBarrierException e) {
        // Želim da dobijem obaveštenje o ovome
        throw new RuntimeException(e);
      }
```

```
      print(this + " isključen");
    }
    private synchronized void
    iskljuciSe() throws InterruptedException {
      angazuj = false;
      monter = null; // Otkači od objekta Monter
      // Vraćamo se u grupu dostupnih:
      grupa.release(this);
      while(angazuj == false)  // Isključi se
        wait();
    }
    public String toString() { return getClass().getName(); }
}

class RobotZaMotor extends Robot {
  public RobotZaMotor(GrupaRobota grupa) { super(grupa); }
  protected void pruziUslugu() {
    print(this + " instalira motor");
    monter.kola().dodajMotor();
  }
}

class RobotZaPogonskiSistem extends Robot {
  public RobotZaPogonskiSistem(GrupaRobota grupa) { super(grupa); }
  protected void pruziUslugu() {
    print(this + " instalira PogonskiSistem");
    monter.kola().dodajPogonskiSistem();
  }
}

class RobotZaTockove extends Robot {
  public RobotZaTockove(GrupaRobota grupa) { super(grupa); }
  protected void pruziUslugu() {
    print(this + " instalira Točkove");
    monter.kola().dodajTockove();
  }
}

class GrupaRobota {
  // (Nečujno) sprečava identične stavke:
  private Set<Robot> grupa = new HashSet<Robot>();
  public synchronized void add(Robot r) {
    grupa.add(r);
    notifyAll();
  }
  public synchronized void
  unajmi(Class<? extends Robot> tipRobota, Monter d)
  throws InterruptedException {
    for(Robot r : grupa)
      if(r.getClass().equals(tipRobota)) {
```

```
        grupa.remove(r);
        r.dodeliMontera(d);
        r.angazuj(); // Uključi ga da obavi posao
        return;
      }
    wait(); // Nema dostupnih
    unajmi(tipRobota, d); // Rekurzivno pokušaj ponovo
  }
  public synchronized void release(Robot r) { add(r); }
}

public class ProizvodnjaKola {
  public static void main(String[] args) throws Exception {
    RedKola redZaSasiju = new RedKola(),
            redZaZavrsnu = new RedKola();
    ExecutorService exec = Executors.newCachedThreadPool();
    GrupaRobota grupaRobota = new GrupaRobota();
    exec.execute(new RobotZaMotor(grupaRobota));
    exec.execute(new RobotZaPogonskiSistem(grupaRobota));
    exec.execute(new RobotZaTockove(grupaRobota));
    exec.execute(new Monter(
      redZaSasiju, redZaZavrsnu, grupaRobota));
    exec.execute(new Izvestilac(redZaZavrsnu));
    // Pokreni sve proizvodnjom šasije:
    exec.execute(new PravljenjeSasije(redZaSasiju));
    TimeUnit.SECONDS.sleep(7);
    exec.shutdownNow();
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Kola se prenose s mesta na mesto pomoću objekta tipa **RedKola**, što je tip reda **LinkedBlockingQueue**. **PravljenjeSasije** pravi kostur **Kola** i smešta ga u **RedKola**. **Monter** skida **Kola** iz **RedaKola** i unajmljuje **Robote** da na njima rade. Barijera tipa **CyclicBarrier** omogućuje **Monteru** da čeka dok svi unajmljeni **Roboti** ne budu gotovi, a potom stavlja **Kola** u izlazni **RedKola** koji ih prenosi za sledeću operaciju. Potrošač završnog objekta tipa **RedKola** jeste **Izvestilac** koji samo ispisuje sastav **Kola** da bi pokazao da su svi zadaci pravilno izvršeni.

Robotima se upravlja u grupi, i kada nešto treba da se uradi, odgovarajući **Robot** biva unajmljen iz grupe. Nakon dovršetka posla, **Robot** se vraća u grupu.

U metodi **main**( ) prave se svi potrebni objekti i inicijalizuju svi zadaci; poslednje se pokreće **PravljenjeSasije** da bi se pokrenuo proces. (Međutim, ponašanje reda **LinkedBlockingQueue** je takvo da sam mogao njega prvog da pokrenem.) Obratite pažnju na to da ovaj program sledi sve smernice u vezi sa životnim ciklusom objekata i zadataka koje su spomenute u ovom poglavlju, pa je postupak gašenja bezbedan.

Uočili ste da su sve metode **Kola** sinhronizovane. Ispostavlja se da je to *u ovom primeru* redundantno, pošto su **Kola** unutar proizvodnje uvek u nekom od redova za čekanje i na svakim kolima u svakom trenutku može raditi samo jedan zadatak. U suštini, redovi za čekanje nameću serijsko pristupanje objektima tipa **Kola**. Ali upravo to i jeste zamka – mogli

biste reći: „Hajde da optimizujemo performanse tako što klasu **Kola** nećemo sinhronizovati, pošto izgleda da to ovde nije neophodno". Ali kasnije, kada ovaj sistem bude povezan s nekim drugim koji *zahteva* da **Kola** budu sinhronizovana, sve će se raspasti.

Brian Goetz komentariše:

Mnogo je lakše reći „**Kola** mogu biti upotrebljena iz više niti, pa hajde da ih napravimo očigledno bezbednim za višenitno izvršavanje". Evo kako ja to posmatram: na strmim mestima u parkovima postoje zaštitne ograde i natpisi „Zabranjeno naslanjanje na zaštitnu ogradu". Naravno, prava svrha ovog pravila nije da spreči naslanjanje na ogradu, nego da spreči padanje niza strminu. Ali je mnogo lakše pridržavati se pravila „Zabranjeno naslanjanje na ogradu" nego pravila „Zabranjeno padanje niza strminu".

Vežba 37: (2) Izmenite program **ProizvodnjaKola.java** dodavanjem još jedne faze u postupak pravljenja kola, u kojoj se dodaju izduvni sistem, karoserija i branici. Kao i u drugoj fazi, pretpostavite da te procese roboti mogu da obavljaju istovremeno.

Vežba 38: (3) Koristeći pristup iz programa **ProizvodnjaKola.java**, modelujte priču o gradnji kuće koja je data u ovom poglavlju.

Optimizacija performansi

Znatan broj klasa u biblioteci Jave SE5 **java.util.concurrent** postoji radi poboljšanja performansi. Tokom prelistavanja biblioteke **concurrent**, teško ćete razlučiti klase namenjene za redovnu upotrebu (kao što su redovi **BlockingQueue**) od klasa koje su samo za poboljšavanje performansi. U ovom odeljku razmotrićemo neka od tih pitanja i klasa za optimizaciju performansi.

Poređenje tehnologija uzajamno isključivih brava (mutexa)

Pošto Java sada obuhvata i staru rezervisanu reč **synchronized** i nove klase **Lock** i **Atomic** Jave SE5, zanimljivo je uporediti te različite pristupe da bismo bolje shvatili vrednost svakog od njih i saznali gde ih treba upotrebljavati.

Bilo bi previše naivno napraviti po jedan jednostavan test za svaki od tih pristupa, kao u sledećem primeru:

```
//: paralelno/JednostavnoMikropoređenjePerformansi.java
// Opasnosti od mikropoređenja performansi.
import java.util.concurrent.locks.*;

abstract class ImaMetoduPovecajZaJedan {
  protected long brojac = 0;
  public abstract void povecajZaJedan();
}
class TestSinhronizacije extends ImaMetoduPovecajZaJedan {
  public synchronized void povecajZaJedan() { ++brojac; }
}

class TestZakljucavanja extends ImaMetoduPovecajZaJedan {
  private Lock brava = new ReentrantLock();
```

```
  public void povecajZaJedan() {
    brava.lock();
    try {
      ++brojac;
    } finally {
      brava.unlock();
    }
  }
}

public class JednostavnoMikropoređenjePerformansi {
  static long test(ImaMetoduPovecajZaJedan pzj) {
    long start = System.nanoTime();
    for(long i = 0; i < 10000000L; i++)
      pzj.povecajZaJedan();
    return System.nanoTime() - start;
  }
  public static void main(String[] args) {
    long vremeZaSynch = test(new TestSinhronizacije());
    long vremeZaLock = test(new TestZakljucavanja());
    System.out.printf("synchronized: %1$10d\n", vremeZaSynch);
    System.out.printf("Lock:         %1$10d\n", vremeZaLock);
    System.out.printf("Lock/synchronized = %1$.3f",
      (double)vremeZaLock/(double)vremeZaSynch);
  }
} /* Ispis: (75% podudaranja)
synchronized:  244919117
Lock:          939098964
Lock/synchronized = 3.834
*///:~
```

Iz rezultata vidite da je korišćenje poziva metode sinhronizovane pomoću rezervisane reči **synchronized** naizgled mnogo brže od upotrebe objekta tipa **Lock**. Šta se tu desilo?

Ovaj primer pokazuje opasnosti od tzv. mikropoređenja performansi.[23] Taj pojam po pravilu upućuje na izolovano merenje performansi, izvan konteksta. Naravno, ni tvrdnje kao što je „Klasa **Lock** je mnogo brža od rezervisane reči **synchronized**" ne možete izricati bez prethodnog merenja. Ali, kada pišete takva ispitivanja, morate biti svesni toga šta se događa tokom prevođenja i u vreme izvršavanja.

Gornji primer je problematičan iz nekoliko razloga. Prvo i osnovno, stvarnu razliku u performansama videćemo samo u slučaju da za uzajamno isključive brave (mutexe) *postoji takmičenje*, dakle više zadataka mora pokušavati da pristupi delovima koda zaštićenim mutexima. U gornjem primeru, svaki mutex ispitujemo pomoću samo jedne niti u metodi **main( )**, izolovano.

Drugo, možda prevodilac obavlja posebne optimizacije kada ugleda rezervisanu reč **synchronized**, a možda primeti da program ima samo jednu nit. Prevodilac bi čak mogao

[23] Brian Goetz mi je mnogo pomogao da to shvatim. Više informacija o merenju performansi pročitajte u njegovom članku na adresi *www-128.ibm.com/developerworks/library/j-jtp12214.*

da shvati kako se **brojac** povećava fiksan broj puta pa da unapred izračuna rezultat. Ima različitih prevodilaca i sistema za izvršavanje, pa je teško reći šta će se tačno dogoditi, ali moramo sprečiti mogućnost da prevodilac predvidi ishod.

Da bi testiranje bilo validno, program mora biti složeniji. Prvo moramo napraviti više zadataka, i to ne samo onih koji menjaju interne vrednosti, već i onih koji te vrednosti čitaju (inače bi optimizator mogao prepoznati da se te vrednosti nikada ne koriste). Pored toga, izračunavanje mora biti toliko složeno i nepredvidljivo da prevodilac nema šanse da obavi agresivnu optimizaciju. To ćemo postići prethodnim učitavanjem velikog niza pseudoslučajnih celih brojeva (prethodno učitavanje smanjuje uticaj poziva metode **Random.nextInt( )** na glavne petlje) i sabiranjem tih brojeva:

```
//: paralelno/PoredjenjeSinhronizacija.java
// Poređenje performansi pri upotrebi eksplicitnih objekata tipa Lock
// i Atomic i performansi pri korišćenju rezervisane reči synchronized.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

abstract class Akumulator {
  public static long ciklusa = 50000L;
  // Broj Modifikatora i Čitalaca tokom svakog testiranja:
  private static final int N = 4;
  public static ExecutorService exec =
    Executors.newFixedThreadPool(N*2);
  private static CyclicBarrier barijera =
    new CyclicBarrier(N*2 + 1);
  protected volatile int indeks = 0;
  protected volatile long broj = 0;
  protected long trajanje = 0;
  protected String id = "greška";
  protected final static int VELICINA = 100000;
  protected static int[] unapredUcitani = new int[VELICINA];
  static {
    // Učitaj niz slučajnih brojeva:
    Random slucajan = new Random(47);
    for(int i = 0; i < VELICINA; i++)
      unapredUcitani[i] = slucajan.nextInt();
  }
  public abstract void akumuliraj();
  public abstract long citaj();
  private class Modifikator implements Runnable {
    public void run() {
      for(long i = 0; i < ciklusa; i++)
        akumuliraj();
      try {
        barijera.await();
      } catch(Exception e) {
```

```
        throw new RuntimeException(e);
      }
    }
  }
  private class Citalac implements Runnable {
    private volatile long broj;
    public void run() {
      for(long i = 0; i < ciklusa; i++)
        broj = citaj();
      try {
        barijera.await();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  public void merenjeVremena() {
    long start = System.nanoTime();
    for(int i = 0; i < N; i++) {
      exec.execute(new Modifikator());
      exec.execute(new Citalac());
    }
    try {
      barijera.await();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    trajanje = System.nanoTime() - start;
    printf("%-13s: %13d\n", id, trajanje);
  }
  public static void
  izvesti(Akumulator aku1, Akumulator aku2) {
    printf("%-22s: %.2f\n", aku1.id + "/" + aku2.id,
      (double)aku1.trajanje/(double)aku2.trajanje);
  }
}

class OsnZaPor extends Akumulator {
  { id = "OsnZaPor"; }
  public void akumuliraj() {
    broj += unapredUcitani[indeks++];
    if(indeks >= VELICINA) indeks = 0;
  }
  public long citaj() { return broj; }
}

class TestRrSynchronized extends Akumulator {
  { id = "synchronized"; }
  public synchronized void akumuliraj() {
```

```
      broj += unapredUcitani[indeks++];
      if(indeks >= VELICINA) indeks = 0;
    }
    public synchronized long citaj() {
      return broj;
    }
  }

  class TestKlaseLock extends Akumulator {
    { id = "Lock"; }
    private Lock brava = new ReentrantLock();
    public void akumuliraj() {
      brava.lock();
      try {
        broj += unapredUcitani[indeks++];
        if(indeks >= VELICINA) indeks = 0;
      } finally {
        brava.unlock();
      }
    }
    public long citaj() {
      brava.lock();
      try {
        return broj;
      } finally {
        brava.unlock();
      }
    }
  }

  class TestKlaseAtomic extends Akumulator {
    { id = "Atomic"; }
    private AtomicInteger indeks = new AtomicInteger(0);
    private AtomicLong broj = new AtomicLong(0);
    public void akumuliraj() {
      // Pazi! U svakom trenutku sme da radi samo jedan
      // objekat tipa Atomic. Ali ipak ćemo steći
      // neki uvid u performanse:
      int i = indeks.getAndIncrement();
      broj.getAndAdd(unapredUcitani[i]);
      if(++i >= VELICINA)
        indeks.set(0);
    }
    public long citaj() { return broj.get(); }
  }

  public class PoredjenjeSinhronizacija {
    static OsnZaPor osnovaZaPoredjenje = new OsnZaPor();
    static TestRrSynchronized synch = new TestRrSynchronized();
    static TestKlaseLock brava = new TestKlaseLock();
```

```
  static TestKlaseAtomic atomic = new TestKlaseAtomic();
  static void test() {
    print("============================");
    printf("%-12s : %13d\n", "Ciklusa", Akumulator.ciklusa);
    osnovaZaPoredjenje.merenjeVremena();
    synch.merenjeVremena();
    brava.merenjeVremena();
    atomic.merenjeVremena();
    Akumulator.izvesti(synch, osnovaZaPoredjenje);
    Akumulator.izvesti(brava, osnovaZaPoredjenje);
    Akumulator.izvesti(atomic, osnovaZaPoredjenje);
    Akumulator.izvesti(synch, brava);
    Akumulator.izvesti(synch, atomic);
    Akumulator.izvesti(brava, atomic);
  }
  public static void main(String[] args) {
    int iteracija = 5; // Podrazumevani broj
    if(args.length > 0) // Opciono promeni broj iteracija
      iteracija = new Integer(args);
    // Prvi put popunjava grupu niti:
    print("Zagrevanje");
    osnovaZaPoredjenje.merenjeVremena();
    // Sada ovo početno testiranje ne obuhvata troškove
    // prvog pokretanja niti.
    // Napravi više mernih tačaka:
    for(int i = 0; i < iteracija; i++) {
      test();
      Akumulator.ciklusa *= 2;
    }
    Akumulator.exec.shutdown();
  }
} /* Ispis: (primer)
Zagrevanje
OsnZaPor     :      34237033
============================
Ciklusa      :         50000
OsnZaPor     :      20966632
synchronized :      24326555
Lock         :      53669950
Atomic       :      30552487
synchronized/OsnZaPor : 1.16
Lock/OsnZaPor         : 2.56
Atomic/OsnZaPor       : 1.46
synchronized/Lock     : 0.45
synchronized/Atomic   : 0.79
Lock/Atomic           : 1.76
============================
Ciklusa      :        100000
OsnZaPor     :      41512818
synchronized :      43843003
```

```
Lock          :      87430386
Atomic        :      51892350
synchronized/OsnZaPor : 1.06
Lock/OsnZaPor         : 2.11
Atomic/OsnZaPor       : 1.25
synchronized/Lock     : 0.50
synchronized/Atomic   : 0.84
Lock/Atomic           : 1.68
============================
Ciklusa       :        200000
OsnZaPor      :      80176670
synchronized  :    5455046661
Lock          :     177686829
Atomic        :     101789194
synchronized/OsnZaPor : 68.04
Lock/OsnZaPor         : 2.22
Atomic/OsnZaPor       : 1.27
synchronized/Lock     : 30.70
synchronized/Atomic   : 53.59
Lock/Atomic           : 1.75
============================
Ciklusa       :        400000
OsnZaPor      :     160383513
synchronized  :     780052493
Lock          :     362187652
Atomic        :     202030984
synchronized/OsnZaPor : 4.86
Lock/OsnZaPor         : 2.26
Atomic/OsnZaPor       : 1.26
synchronized/Lock     : 2.15
synchronized/Atomic   : 3.86
Lock/Atomic           : 1.79
============================
Ciklusa       :        800000
OsnZaPor      :     322064955
synchronized  :     336155014
Lock          :     704615531
Atomic        :     393231542
synchronized/OsnZaPor : 1.04
Lock/OsnZaPor         : 2.19
Atomic/OsnZaPor       : 1.22
synchronized/Lock     : 0.47
synchronized/Atomic   : 0.85
Lock/Atomic           : 1.79
============================
Ciklusa       :       1600000
OsnZaPor      :     650004120
synchronized  :   52235762925
Lock          :    1419602771
Atomic        :     796950171
```

```
synchronized/OsnZaPor : 80.36
Lock/OsnZaPor         : 2.18
Atomic/OsnZaPor       : 1.23
synchronized/Lock     : 36.80
synchronized/Atomic   : 65.54
Lock/Atomic           : 1.78
============================
Ciklusa       :       3200000
OsnZaPor      :    1285664519
synchronized  :   96336767661
Lock          :    2846988654
Atomic        :    1590545726
synchronized/OsnZaPor : 74.93
Lock/OsnZaPor         : 2.21
Atomic/OsnZaPor       : 1.24
synchronized/Lock     : 33.84
synchronized/Atomic   : 60.57
Lock/Atomic           : 1.79
*///:~
```

U ovom programu upotrebljen je projektni obrazac *Template Method*[24] da bi sav zajednički kôd bio u osnovnoj klasi, izolovan od promenljivog koda u realizacijama metoda **akumuliraj**( ) i **citaj**( ) u izvedenim klasama. U izvedenim klasama **TestRrSynchronized**, **TestKlaseLock** i **TestKlaseAtomic** vidite kako metode **akumuliraj**( ) i **citaj**( ) izražavaju različite načine realizacije uzajamnog isključivanja.

U prethodnom programu, zadaci se izvršavaju pomoću objekta tipa **FixedThreadPool** u pokušaju da se sve niti naprave na početku i kako bi se sprečili eventualni dodatni troškovi tokom testiranja. Za svaki slučaj, početno testiranje se ponavlja i prvi rezultati odbacuju, zato što obuhvataju i početno pravljenje niti.

CyclicBarrier je potrebna zato što obezbeđuje da se svi zadaci završe pre nego što svako testiranje proglasimo završenim.

Odredba **static** je upotrebljena za prethodno učitavanje niza slučajnih brojeva, pre početka testiranja. Na taj način se tokom testiranja ne vide režijski troškovi generisanja slučajnih brojeva.

Metoda **akumuliraj**( ) nakon svakog poziva pomera se na sledeće mesto u nizu **unapredUcitani** (kada dođe do kraja, vraća se na njegov početak) i vrednosti **broj** dodaje još jedan slučajno generisan broj. Više zadataka **Modifikator** i **Citalac** nadmeće se za dobijanje objekta **Akumulator**.

Obratite pažnju na to da sam u **TestuKlaseAtomic** napomenuo kako je situacija previše složena da bismo upotrebili objekte tipa **Atomic** – u suštini, ukoliko ima više **Atomic** objekata, morate da odustanete i koristite konvencionalnije uzajamno isključive brave (mutexe). (U dokumentaciji JDK izričito piše da je korišćenje objekata tipa **Atomic** dobro samo kada je kritično ažuriranje objekta svedeno na jednu promenljivu.) Međutim, test sam ipak ostavio da biste stekli uvid u poboljšanje performansi koje prouzrokuju **Atomic** objekti.

[24] Videti *Thinking in Patterns* na adresi *www.MindView.net*.

U metodi **main( )**, testiranje se ponavlja i možete odlučiti da zatražite više od pet (podrazumevanih) ponavljanja. U svakom ponavljanju, broj ciklusa testiranja se udvostručuje, pa možete videti kako se razne uzajamno isključive brave (mutexi) ponašaju kada se izvršavaju sve duže i duže. Kao što vidite iz izlaza, rezultati su prilično iznenađujući. U prve četiri iteracije izgleda da je rezervisana reč **synchronized** efikasnija od klasa **Lock** i **Atomic**. Ali iznenada neka granica biva prekoračena pa deluje da **synchronized** postaje veoma neefikasna, dok **Lock** i **Atomic** kao da približno održavaju svoju srazmeru prema testu **OsnZaPor** (osnova za poređenje) i stoga postaju mnogo efikasniji od **synchronized**.

Imajte u vidu da ovaj program daje tek naznaku razlike između raznih realizacija uzajamno isključivih brava i da gornji izlaz pokazuje te razlike samo na mom konkretnom računaru pod mojim konkretnim okolnostima. Kao što ćete videti kada budete eksperimentisali s ponašanjem, ono se znatno menja u zavisnosti od broja niti i trajanja izvršavanja programa. Neke optimizacije vrućih tačaka pozivaju se tek nekoliko minuta nakon početka izvršavanja programa, a u slučaju serverskih programa, nakon nekoliko sati.

Uza sve to, prilično je jasno da je klasa **Lock** najčešće znatno efikasnija od rezervisane reči **synchronized**, a izgleda i da se režijski troškovi upotrebe rezervisane reči **synchronized** znatno menjaju, dok oni klase **Lock** ostaju relativno ujednačeni.

Da li to znači da rezervisanu reč **synchronized** uopšte ne treba upotrebljavati? U obzir morate uzeti dva činioca: prvo, u primeru **PoredjenjeSinhronizacija.java** tela mutex metoda su izuzetno mala. Uopšte uzev, to je dobro – uzajamno isključite samo one delove programa koje apsolutno morate. Međutim, uzajamno isključeni delovi u praksi mogu biti veći nego u gornjem primeru, te će i procentualni deo vremena koji se provodi u telu metoda verovatno biti znatno veći nego režijski troškovi od ulaska i izlaska iz uzajamno isključive brave, a to bi moglo da poništi eventualno poboljšanje koje prouzrokuje ubrzanje uzajamno isključive brave. Naravno, istinu ćete saznati tek kada isprobate različite pristupe i vidite njihov uticaj, a to bi trebalo da radite tek prilikom optimizacije performansi.

Drugo, ko pročita kôd u ovom poglavlju videće da rezervisana reč **synchronized** proizvodi mnogo čitljiviji kôd od idioma zaključaj-**try/finally**-otključaj, za koji su potrebne brave tipa **Lock**, i zato sam u ovom poglavlju uglavnom koristio **synchronized**. Na više mesta u ovoj knjizi rekao sam da se kôd mnogo češće čita nego piše – u programiranju je važnije da kôd razumeju drugi ljudi nego računar – te je čitljivost koda ključna. Zato ima smisla početi s rezervisanom reči **synchronized** i preći na objekte tipa **Lock** tek prilikom optimizacije performansi.

Najzad, lepo je kada možete da upotrebite klase **Atomic** u paralelnom programu, ali morate voditi računa o tome da su, kao što smo videli u programu **PoredenjeSinhronizacija.java**, **Atomic** objekti korisni samo u veoma jednostavnim slučajevima, po pravilu samo kada imate jedan **Atomic** objekat koji menjate i kada je taj objekat nezavisan od svih ostalih objekata. Bezbednije je krenuti s tradicionalnijim uzajamno isključivim bravama i pokušati prelazak na **Atomic** objekte kasnije, ukoliko to budete morali da učinite zbog performansi.

Kontejneri bez zaključavanja

U poglavlju *Čuvanje objekata* naglasio sam da su kontejneri osnovna alatka za celo programiranje, a to važi i za paralelno programiranje. Zato su prvi Java kontejneri (kao **Vector** i **Hashtable**) imali mnogo sinhronizovanih metoda koje su prouzrokovale neprihvatljive režijske troškove kada nisu bile korišćene u višenitnim programima. U Javi 1.2 nova kontejnerska biblioteka bila je desinhronizovana, a klasa **Collections** je dobila razne statičke „sinhronizovane" dekorativne metode za sinhronizaciju različitih tipova kontejnera. Iako je to bilo poboljšanje jer je programer mogao da bira da li da koristi sinhronizaciju kontejnera, režijski troškovi su i dalje zavisili od zaključavanja pomoću rezervisane reči **synchronized**. U Javu SE5 dodati su novi kontejneri baš da bi se poboljšale performanse bezbednog višenitnog rada, a za to su upotrebljene pametne tehnike kojima se izbegava zaključavanje.

Opšta strategija u pozadini tih kontejnera bez zaključavanja jeste sledeća: modifikacija kontejnera je dozvoljena istovremeno kada i čitanje njihovog sadržaja, ukoliko čitaoci mogu da vide samo rezultate *završenih* modifikacija. Modifikacija se obavlja na zasebnoj kopiji određenog dela strukture podataka (katkada i cele strukture) i ta kopija je nevidljiva tokom postupka modifikacije. Modifikovana struktura se atomski zamenjuje „glavnom" strukturom podataka tek kada je modifikacija završena i nakon toga čitaoci mogu da je vide.

U listi **CopyOnWriteArrayList**, svako upisivanje prouzrokuje pravljenje kopije celog pripadnog niza. Originalni niz se ne dira, pa se tokom modifikacije kopiranog niza sva čitanja mogu obaviti bezbedno. Kada se modifikacija završi, atomska operacija će zameniti stari niz novim, pa će nova čitanja videti nove informacije. Jedna od prednosti klase **CopyOnWriteArrayList** jeste to što ne baca izuzetak **ConcurrentModificationException** kada tom listom istovremeno prolazi i modifikuje je više iteratora, pa ne morate da pišete specijalni kôd za zaštitu od takvih izuzetaka, kao što ste morali pre.

Klasa **CopyOnWriteArraySet** upotrebljava **CopyOnWriteArrayList** da bi svoje ponašanje postigla bez zaključavanja.

Klase **ConcurrentHashMap** i **ConcurrentLinkedQueue** koriste slične tehnike da bi omogućile paralelno čitanje i upisivanje, ali kopiraju i modifikuju samo delove kontejnera, a ne ceo kontejner. Međutim, čitaoci ne vide modifikacije pre nego što budu završene. **ConcurrentHashMap** ne baca izuzetke **ConcurrentModificationException**.

O performansama

Ukoliko iz kontejnera bez zaključavanja uglavnom čitate, biće to mnogo brže nego da čitate iz njegove sinhronizovane verzije, zato što izbegavate režijske troškove od zaključavanja i otključavanja. To važi i za mali broj upisivanja u kontejner bez zaključavanja, ali bi bilo zanimljivo steći neku predstavu koliko je to „malo". U ovom odeljku daćemo grubu sliku o razlikama u performansama tih kontejnera pod različitim uslovima.

Počeću od generičke strukture za testiranje svih tipova kontejnera, uključujući tu i **Mape**. Generički parametar **C** predstavlja tip kontejnera:

```
//: paralelno/Tester.java
// Osnovna struktura za testiranje performansi paralelnih kontejnera.
import java.util.concurrent.*;
import net.mindview.util.*;

public abstract class Tester<C> {
  static int iteracijaTestiranja = 10;
  static int ciklusaTestiranja = 1000;
  static int velicinaKontejnera = 1000;
  abstract C inicijalizatorKontejnera();
  abstract void pokreniCitaoceIUpisivace();
  C kontejnerZaTestiranje;
  String testId;
  int nCitalaca;
  int nUpisivaca;
  volatile long procitajRezultat = 0;
  volatile long vremeCit = 0;
  volatile long vremeUpis = 0;
  CountDownLatch krajnjaBrava;
  static ExecutorService exec =
    Executors.newCachedThreadPool();
  Integer[] upisiPodatke;
  Tester(String testId, int nCitalaca, int nUpisivaca) {
    this.testId = testId + " " +
      nCitalaca + "č " + nUpisivaca + "p";
    this.nCitalaca = nCitalaca;
    this.nUpisivaca = nUpisivaca;
    upisiPodatke = Generated.array(Integer.class,
      new RandomGenerator.Integer(), velicinaKontejnera);
    for(int i = 0; i < iteracijaTestiranja; i++) {
      izvrsiTestiranje();
      vremeCit = 0;
      vremeUpis = 0;
    }
  }
  void izvrsiTestiranje() {
    krajnjaBrava = new CountDownLatch(nCitalaca + nUpisivaca);
    kontejnerZaTestiranje = inicijalizatorKontejnera();
    pokreniCitaoceIUpisivace();
    try {
      krajnjaBrava.await();
    } catch(InterruptedException izz) {
      System.out.println("krajnjaBrava prekinuta");
    }
```

```
      System.out.printf("%-27s %14d %14d\n",
        testId, vremeCit, vremeUpis);
      if(vremeCit != 0 && vremeUpis != 0)
        System.out.printf("%-27s %14d\n",
          "vremeCit + vremeUpis =", vremeCit + vremeUpis);
    }
    abstract class ZadatakTestiranja implements Runnable {
      abstract void test();
      abstract void upisiRezultate();
      long trajanje;
      public void run() {
        long vremePocetka = System.nanoTime();
        test();
        trajanje = System.nanoTime() - vremePocetka;
        synchronized(Tester.this) {
          upisiRezultate();
        }
        krajnjaBrava.countDown();
      }
    }
    public static void inicMMain(String[] args) {
      if(args.length > 0)
        iteracijaTestiranja = new Integer(args[0]);
      if(args.length > 1)
        ciklusaTestiranja = new Integer(args[1]);
      if(args.length > 2)
        velicinaKontejnera = new Integer(args[2]);
      System.out.printf("%-27s %14s %14s\n",
        "Tip", "Vreme čitanja", "Vreme upis.");
    }
  } ///:-
```

Apstraktna metoda **inicijalizatorKontejnera**( ) vraća inicijalizovan kontejner koji treba testirati i smešta ga u polje **kontejnerZaTestiranje**. Druga apstraktna metoda, **pokreniCitaoceIUpisivace**( ), pokreće zadatke čitanja i upisivanja koji će čitati i modifikovati testirani kontejner. Razni testovi se obavljaju s različitim brojem čitalaca i upisivača da bi se video uticaj takmičenja za bravu (za sinhronizovane kontejnere) i upisivanja (za kontejnere bez zaključavanja).

Konstruktor dobija razne informacije o testiranju (trebalo bi da su vam identifikatori argumenata jasni sami po sebi), zatim poziva metodu **izvrsiTestiranje**( ) **iteracijaTestiranja** broj puta. Metoda **izvrsiTestiranje**( ) pravi objekat tipa **CountDownLatch** (da bi test mogao da zna kada su se svi zadaci završili), inicijalizuje kontejner, poziva metodu pokreniCitaoceIUpisivace( ) i zatim čeka dok se svi oni ne završe.

Osnovu klasa Citalac ili Upisivac čini **ZadatakTestiranja** koji meri trajanje svoje apstraktne metode **test**( ) i zatim unutar sinhronizovanog bloka poziva metodu **upisiRezultate**( ) da bi sačuvao rezultate.

Prethodnu strukturu (u kojoj ste prepoznali projektni obrazac *Template Method*) koristite tako što iz klase **Tester** izvedete konkretni tip kontejnera koji želite da testirate i napravite odgovarajuće klase **Citalac** i **Upisivac**:

```
//: paralelno/PoredjenjaListi.java
// {Args: 1 10 10} (Brza verifikaciona provera tokom builda)
// Grubo poređenje performansi lista bezbednih u višenitnom radu.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class TestiranjeLista extends Tester<List<Integer>> {
  TestiranjeLista(String testId, int nCitalaca, int nUpisivaca) {
    super(testId, nCitalaca, nUpisivaca);
  }
  class Citalac extends ZadatakTestiranja {
    long rezultat = 0;
    void test() {
      for(long i = 0; i < ciklusaTestiranja; i++)
        for(int indeks = 0; indeks < velicinaKontejnera; indeks++)
          rezultat += kontejnerZaTestiranje.get(indeks);
    }
    void upisiRezultate() {
      procitajRezultat += rezultat;
      vremeCit += trajanje;
    }
  }
  class Upisivac extends ZadatakTestiranja {
    void test() {
      for(long i = 0; i < ciklusaTestiranja; i++)
        for(int indeks = 0; indeks < velicinaKontejnera; indeks++)
          kontejnerZaTestiranje.set(indeks, upisiPodatke[indeks]);
    }
    void upisiRezultate() {
      vremeUpis += trajanje;
    }
  }
  void pokreniCitaoceIUpisivace() {
    for(int i = 0; i < nCitalaca; i++)
      exec.execute(new Citalac());
    for(int i = 0; i < nUpisivaca; i++)
      exec.execute(new Upisivac());
  }
}
TestSinhronizovanogObjektaTipaArrayList
class TestSinhronizovanogObjektaTipaArrayList extends TestiranjeLista {
  List<Integer> inicijalizatorKontejnera() {
    return Collections.synchronizedList(
      new ArrayList<Integer>(
        new CountingIntegerList(velicinaKontejnera)));
  }
```

```
  TestSinhronizovanogObjektaTipaArrayList(int nCitalaca, int nUpisivaca)
{
    super("Sinhro. ArrayList", nCitalaca, nUpisivaca);
  }
}

class TestKlaseCopyOnWriteArrayList extends TestiranjeLista {
  List<Integer> inicijalizatorKontejnera() {
    return new CopyOnWriteArrayList<Integer>(
      new CountingIntegerList(velicinaKontejnera));
  }
  TestKlaseCopyOnWriteArrayList(int nCitalaca, int nUpisivaca) {
    super("CopyOnWriteArrayList", nCitalaca, nUpisivaca);
  }
}

public class PoredjenjaLista {
  public static void main(String[] args) {
    Tester.inicMMain(args);
    new TestSinhronizovanogObjektaTipaArrayList(10, 0);
    new TestSinhronizovanogObjektaTipaArrayList(9, 1);
    new TestSinhronizovanogObjektaTipaArrayList(5, 5);
    new TestKlaseCopyOnWriteArrayList(10, 0);
    new TestKlaseCopyOnWriteArrayList(9, 1);
    new TestKlaseCopyOnWriteArrayList(5, 5);
    Tester.exec.shutdown();
  }
} /* Ispis: (primer)
Tip                          Vreme čitanja     Vreme upis.
Sinhro. ArrayList 10c 0u      232158294700              0
Sinhro. ArrayList 9c 1u       198947618203    24918613399
vremeCit + vremeUpis =        223866231602
Sinhro. ArrayList 5c 5u       117367305062   132176613508
vremeCit + vremeUpis =        249543918570
CopyOnWriteArrayList 10c 0u      758386889             0
CopyOnWriteArrayList 9c 1u       741305671     136145237
vremeCit + vremeUpis =           877450908
CopyOnWriteArrayList 5c 5u       212763075    67967464300
vremeCit + vremeUpis =         68180227375
*///:-
```

U programu **TestiranjeLista**, klase **Citalac** i **Upisivac** obavljaju određene radnje u listi **List<Integer>**. U metodi **Citalac.upisiRezultate( )** skladišti se **trajanje** ali i **rezultat**, kako bi se sprečilo da optimizacija izbaci izračunavanja. Zatim se definiše metoda **pokreniCitaoceIUpisivace( )** koja pravi i izvršava određene **Citaoce** i **Upisivace**.

Nakon pravljenja klase **TestiranjeLista**, iz nje moramo da izvedemo novu klasu kako bismo sprečili **inicijalizatorKontejnera( )** da napravi i inicijalizuje konkretne kontejnere za testiranje.

U metodi **main( )** vidite varijacije testiranja s različitim brojevima čitalaca i upisivača. Zbog poziva **Tester.inicMMain(args)**, promenljive testiranja možete promeniti tako što ćete zadati argumente na komandnoj liniji.

Svako testiranje se podrazumevano obavlja 10 puta; to stabilizuje izlazne rezultate, koji se mogu promeniti zbog delatnosti JVM-a kao što su optimizacija „vrućih tačaka" i sakupljanje smeća.[25] Prikazani primer izlaznih rezultata izmenio sam tako da se vidi samo poslednja iteracija svakog testiranja. Iz izlaznih rezultata vidite da sinhronizovana **ArrayList** ima približno iste performanse bez obzira na broj čitalaca i upisivača – čitaoci se nadmeću za brave sa drugim čitaocima, isto kao upisivači. Međutim, **CopyOnWriteArrayList** je mnogo brži kada nema upisivača i još uvek znatno brži s pet upisivača. Izgleda kao da **CopyOnWriteArrayList** možete bezbrižno da koristite koliko god hoćete; troškovi od upisivanja u listu kao da neko vreme ostaju manji nego troškovi od sinhronizacije cele liste. Naravno, oba pristupa morate isprobati u konkretnoj aplikaciji da biste pouzdano utvrdili koji je bolji.

Podsećam da ovo ni približno nije dobro poređenje performansi za dobijanje apsolutnih brojeva. Vi ćete gotovo sigurno dobiti drugačije brojeve. Cilj mi je bio samo da steknete uvid u relativna ponašanja te dve vrste kontejnera.

Pošto skup **CopyOnWriteArraySet** upotrebljava listu **CopyOnWriteArrayList**, njegovo ponašanje će biti slično i ovde ga ne moramo zasebno ispitivati.

Poređenje realizacija Mapa

Istu strukturu možemo da upotrebimo za dobijanje grube slike performansi sinhronizovane klase **HashMap** i klase **ConcurrentHashMap**:

```
//: paralelno/PoredjenjaMapa.java
// {Args: 1 10 10} (Brza verifikaciona provera tokom builda)
// Grubo poređenje performansi mapa bezbednih u višenitnom radu.
import java.util.*;import java.util.concurrent.*;
import net.mindview.util.*;

abstract class TestiranjeMapa
extends Tester<Map<Integer,Integer>> {
  TestiranjeMapa(String testId, int nCitalaca, int nUpisivaca) {
    super(testId, nCitalaca, nUpisivaca);
  }
  class Citalac extends ZadatakTestiranja {
    long rezultat = 0;
    void test() {
      for(long i = 0; i < ciklusaTestiranja; i++)
        for(int indeks = 0; indeks < velicinaKontejnera; indeks++)
          rezultat += kontejnerZaTestiranje.get(indeks);
    }
    void upisiRezultate() {
      procitajRezultat += rezultat;
```

[25] Uvod u poređenje performansi pod uticajem Javinog dinamičkog prevođenja pročitajte u članku *www-128.ibm.com/developerworks/library/j-jtp12214*.

```
      vremeCit += trajanje;
    }
  }
  class Upisivac extends ZadatakTestiranja {
    void test() {
      for(long i = 0; i < ciklusaTestiranja; i++)
        for(int indeks = 0; indeks < velicinaKontejnera; indeks++)
          kontejnerZaTestiranje.put(indeks, upisiPodatke[indeks]);
    }
    void upisiRezultate() {
      vremeUpis += trajanje;
    }
  }
  void pokreniCitaoceIUpisivace() {
    for(int i = 0; i < nCitalaca; i++)
      exec.execute(new Citalac());
    for(int i = 0; i < nUpisivaca; i++)
      exec.execute(new Upisivac());
  }
}

class TestiranjeSinhronizovaneHashMape extends TestiranjeMapa {
  Map<Integer,Integer> inicijalizatorKontejnera() {
    return Collections.synchronizedMap(
      new HashMap<Integer,Integer>(
        MapData.map(
          new CountingGenerator.Integer(),
          new CountingGenerator.Integer(),
          velicinaKontejnera)));
  }
  TestiranjeSinhronizovaneHashMape(int nCitalaca, int nUpisivaca) {
    super("Sinhro. HashMap", nCitalaca, nUpisivaca);
  }
}

class TestiranjeConcurrentHashMape extends TestiranjeMapa {
  Map<Integer,Integer> inicijalizatorKontejnera() {
    return new ConcurrentHashMap<Integer,Integer>(
      MapData.map(
        new CountingGenerator.Integer(),
        new CountingGenerator.Integer(), velicinaKontejnera));
  }
  TestiranjeConcurrentHashMape(int nCitalaca, int nUpisivaca) {
    super("ConcurrentHashMap", nCitalaca, nUpisivaca);
  }
}

public class PoredjenjaMapa {
  public static void main(String[] args) {
    Tester.inicMMain(args);
    new TestiranjeSinhronizovaneHashMape(10, 0);
```

```
      new TestiranjeSinhronizovaneHashMape(9, 1);
      new TestiranjeSinhronizovaneHashMape(5, 5);
      new TestiranjeConcurrentHashMape(10, 0);
      new TestiranjeConcurrentHashMape(9, 1);
      new TestiranjeConcurrentHashMape(5, 5);
      Tester.exec.shutdown();
    }
} /* Ispis: (primer)
Tip                          Vreme čitanja     Vreme upis.
Sinhro. HashMap 10c 0u        306052025049               0
Sinhro. HashMap 9c 1u         428319156207    47697347568
vremeCit + vremeUpis =        476016503775
Sinhro. HashMap 5c 5u         243956877760   244012003202
vremeCit + vremeUpis =        487968880962
ConcurrentHashMap 10c 0u       23352654318               0
ConcurrentHashMap 9c 1u        18833089400     1541853224
vremeCit + vremeUpis =         20374942624
ConcurrentHashMap 5c 5u        12037625732    11850489099
vremeCit + vremeUpis =         23888114831
*///:~
```

Uticaj dodavanja upisivača na **ConcurrentHashMap** manji je čak i od uticaja na **CopyOnWriteArrayList**, ali **ConcurrentHashMap** upotrebljava drugačiju tehniku koja očigledno minimizuje uticaj (troškove) upisivanja.

Optimističko zaključavanje

Pored toga što objekti tipa **Atomic** izvršavaju atomske operacije kao što je **decrementAndGet( )**, neke **Atomic** klase omogućuju i tzv. optimističko zaključavanje. To znači da prilikom izračunavanja ne koristite uzajamno isključivanje, ali da nakon završetka izračunavanja za ažuriranje **Atomic** objekta koristite metodu **compareAndSet( )**. Prosleđujete joj staru i novu vrednost, i ako se stara vrednost ne podudara s vrednošću pronađenom u **Atomic** objektu, operacija se završava neuspehom – to znači da je neki drugi zadatak u međuvremenu izmenio objekat. Ne zaboravite da inače uvek upotrebljavamo mutex (rezervisanu reč **synchronized** ili objekat tipa **Lock**) i tako sprečavamo da više zadataka istovremeno modifikuje objekat, ali ovde se ponašamo optimistički tako što podatke ostavljamo nezaključane i nadamo se da drugi zadaci neće uleteti i modifikovati ih. Naravno, sve to radimo samo zbog performansi – korišćenjem objekta tipa **Atomic** umesto rezervisane reči **synchronized** ili objekta tipa **Lock**, verovatno ćete poboljšati performanse.

Šta se dešava ako operacija **compareAndSet( )** zakaže? Stvari postaju čupave, pa ovu tehniku možete da primenjujete samo na projekte koje možete krojiti kako god potrebe nalažu. Ako **compareAndSet( )** zakaže, morate da odlučite šta da radite; to je veoma važno, pošto u slučaju da je oporavak nemoguć, umesto ove tehnike morate da upotrebite neku od konvencionalnih brava za uzajamno isključivanje (mutexa). Možda možete pokušati da ponovite operaciju i neće smetati ako ona (tek) drugi put uspe. Ili možda možete da zanemarite neuspeh – u nekim simulacijama, gubitak jedne tačke ne znači ništa za celinu, jer je broj tačaka ogroman. (Naravno, model morate poznavati toliko dobro da znate je li prethodna hipoteza istinita.)

Zamislimo fiktivnu simulaciju koja se sastoji od 100 000 „gena" dužine 30; recimo da je to početak nekakvog genetičkog algoritma. Pretpostavimo da je za svaku „evoluciju" genetičkog algoritma potreban veoma skup proračun, pa ste odlučili da upotrebite višeprocesorski računar, rasporedite zadatke među procesorima i tako poboljšate performanse. Sem toga, umesto objekata tipa **Lock** koristite **Atomic** objekte da biste izbegli režijske troškove uzajamnog isključivanja. (Naravno, program ste najpre napisali na najjednostavniji mogući način, pomoću rezervisane reči **synchronized**. Tek kada je takav program proradio, otkrili ste da je prespor i počeli da primenjujete tehnike za poboljšavanje performansi!) Zbog prirode ovog modela, zadatak koji otkrije sudar tokom proračuna, može da ga zanemari i da ne ažurira svoju vrednost. Evo kako to izgleda:

```
//: paralelno/BrzaSimulacija.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class BrzaSimulacija {
  static final int N_ELEMENATA = 100000;
  static final int N_GENA = 30;
  static final int N_EVOLVERA = 50;
  static final AtomicInteger[][] MATRICA =
    new AtomicInteger[N_ELEMENATA][N_GENA];
  static Random slucajan = new Random(47);
  static class Evolver implements Runnable {
    public void run() {
      while(!Thread.interrupted()) {
        // Nasumično izaberi element na kojem češ raditi:
        int element = slucajan.nextInt(N_ELEMENATA);
        for(int i = 0; i < N_GENA; i++) {
          int prethodni = element - 1;
          if(prethodni < 0) prethodni = N_ELEMENATA - 1;
          int sledeci = element + 1;
          if(sledeci >= N_ELEMENATA) sledeci = 0;
          int staravrednost = MATRICA[element][i].get();
          // Nekakav proračun po modelu:
          int novavrednost = staravrednost +
            MATRICA[prethodni][i].get() + MATRICA[sledeci][i].get();
          novavrednost /= 3; // Prosek tri vrednosti
          if(!MATRICA[element][i]
            .compareAndSet(staravrednost, novavrednost)) {
            // Ovde dolazi strategija obrade neuspeha. Mi ćemo
            // samo izvestiti o neuspehu i zanemariti ga; model
            // će kad-tad morati da ga obradi.
            print("Stara vrednost se razlikuje od " + staravrednost);
          }
        }
      }
    }
  }
}
```

```
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENATA; i++)
      for(int j = 0; j < N_GENA; j++)
        MATRICA[i][j] = new AtomicInteger(slucajan.nextInt(1000));
    for(int i = 0; i < N_EVOLVERA; i++)
      exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* (Pokrenite da biste videli rezultat) *///:~
```

Sve elemente smo stavili u niz, zato što smatramo da će to poboljšati performanse. (Tu pretpostavku ćemo testirati u jednoj od vežbi.) Svaki objekat tipa **Evolver** uprosečava svoju vrednost s prethodnom i sa sledećom, i ako ažuriranje ne uspe, on ispisuje tu vrednost i ide dalje. Imajte u vidu da u programu nema uzajamnog isključivanja.

Vežba 39: (6) Da li su pretpostavke u programu **BrzaSimulacija.java** razumne? Izmenite niz tako da sadrži proste cele brojeve umesto objekata tipa **AtomicInteger** i upotrebite **Lock** brave za uzajamno isključivanje (mutexe). Uporedite performanse te dve verzije programa.

ReadWriteLock

ReadWriteLock optimizuje situaciju kada u strukturu podataka upisujete relativno retko, ali iz nje više zadataka često čita. Klasa **ReadWriteLock** omogućuje da do prvog pokušaja upisivanja istovremeno radi više čitalaca. Nakon zaključavanja brave za pisanje, čitaoci ne mogu da rade sve dok se brava za pisanje ponovo ne otključa.

Nemoguće je unapred reći da li će **ReadWriteLock** poboljšati performanse vašeg programa; to zavisi od činilaca kao što su srazmera broja čitanja i broja modifikovanja podataka, trajanje operacije čitanja i upisivanja (brava je složenija, pa kratke operacije ne daju da se vidi poboljšanje), količina takmičenja među nitima i da li se program izvršava na višeprocesorskom računaru. Na kraju krajeva, jedini način da saznate da li **ReadWriteLock** poboljšava vaš program jeste da ga isprobate.

U narednom primeru prikazana je samo najjednostavnija upotreba klase **ReadWriteLock**, tj. zasebnih brava za čitanje i upisivanje:

```
//: paralelno/ListaCitalacaIUpisivaca.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListaCitalacaIUpisivaca<T> {
  private ArrayList<T> zakljucanaLista;
  // Neka redosled bude pošten:
  private ReentrantReadWriteLock brava =
```

```
    new ReentrantReadWriteLock(true);
  public ListaCitalacaIUpisivaca(int velicina, T pocetnaVrednost) {
    zakljucanaLista = new ArrayList<T>(
      Collections.nCopies(velicina, pocetnaVrednost));
  }
  public T set(int indeks, T element) {
    Lock bravaZaUpisivanje = brava.writeLock();
    bravaZaUpisivanje.lock();
    try {
      return zakljucanaLista.set(indeks, element);
    } finally {
      bravaZaUpisivanje.unlock();
    }
  }
  public T get(int indeks) {
    Lock bravaZaCitanje = brava.readLock();
    bravaZaCitanje.lock();
    try {
      // Dokaz da više čitalaca može zaključati
      // (pribaviti) bravu za čitanje:
      if(brava.getReadLockCount() > 1)
        print(brava.getReadLockCount());
      return zakljucanaLista.get(indeks);
    } finally {
      bravaZaCitanje.unlock();
    }
  }
  public static void main(String[] args) throws Exception {
    new TestListeCitalacaIUpisivaca(30, 1);
  }
}

class TestListeCitalacaIUpisivaca {
  ExecutorService exec = Executors.newCachedThreadPool();
  private final static int VELICINA = 100;
  private static Random slucajan = new Random(47);
  private ListaCitalacaIUpisivaca<Integer> lista =
    new ListaCitalacaIUpisivaca<Integer>(VELICINA, 0);
  private class Upisivac implements Runnable {
    public void run() {
      try {
        for(int i = 0; i < 20; i++) { // test od 2 sekunde
          lista.set(i, slucajan.nextInt());
          TimeUnit.MILLISECONDS.sleep(100);
        }
      } catch(InterruptedException e) {
        // Prihvatljiv način izlaska
      }
```

```
        print("Upisivac gotov, gasim");
        exec.shutdownNow();
      }
    }
    private class Citalac implements Runnable {
      public void run() {
        try {
          while(!Thread.interrupted()) {
            for(int i = 0; i < VELICINA; i++) {
              lista.get(i);
              TimeUnit.MILLISECONDS.sleep(1);
            }
          }
        } catch(InterruptedException e) {
          // Prihvatljiv način izlaska
        }
      }
    }
    public TestListeCitalacaIUpisivaca(int citalaca, int upisivaca) {
      for(int i = 0; i < citalaca; i++)
        exec.execute(new Citalac());
      for(int i = 0; i < upisivaca; i++)
        exec.execute(new Upisivac());
    }
  } /* (Pokrenite da biste videli rezultat) *///:~
```

ListaCitalacaIUpisivaca može sadržati fiksan broj objekata proizvoljnog tipa. Njenom konstruktoru morate dati željenu veličinu liste i početnu vrednost kojom listu treba popuniti. Metoda **set( )** zaključava bravu za upisivanje kako bi mogla da pozove pripadnu metodu **ArrayList.set( )**, a metoda **get( )** zaključava bravu za čitanje kako bi mogla da pozove metodu **ArrayList.get( )**. Pored toga, **get( )** proverava da li je više čitalaca pribavilo (zaključalo) bravu za čitanje i ako jeste, prikazuje njihov broj da bi dokazala kako više čitalaca može tu bravu istovremeno da zaključa.

Kao test klase **ListaCitalacaIUpisivaca**, **TestListeCitalacaIUpisivaca** pravi zadatke za čitaoce i upisivače u objekat tipa **ListaCitalacaIUpisivaca<Integer>**. Vodite računa o tome da je broj upisivanja mnogo manji od broja čitanja.

Pročitajte dokumentaciju JDK za klasu **ReentrantReadWriteLock** i videćete da ona ima više drugih metoda i da se spominju nekakva „ravnopravnost" i „strateške odluke". Ta alatka je prilično sofisticirana, pa je treba koristiti samo kada pokušavate da poboljšate performanse. U prvoj verziji programa treba da koristite jednostavnu sinhronizaciju, a **ReadWriteLock** primenjujte samo ako morate

Vežba 40: (6) Na osnovu primera **ListaCitalacaIUpisivaca.java**, napravite objekat tipa MapaCitalacaIUpisivaca pomoću klase HashMap. Ispitajte njegove performanse pomoću prilagodenog programa **PoredjenjaMapa.java**. Kakve su u odnosu na performanse sinhronizovanog objekta tipa **HashMap** odnosno **ConcurrentHashMap**?

Aktivni objekti

Kada proučite ovo poglavlje, možda ćete steći utisak da je u Javi veoma složeno i teško korektno sprovesti višenitni rad. Pored toga, izgleda i pomalo kontraproduktivno – iako zadaci rade paralelno, morate uložiti mnogo truda da realizujete tehnike za sprečavanje tih zadataka kako ne bi smetali jedan drugom.

Ako ste ikada pisali u asembleru, pisanje višenitnih programa izaziva isti osećaj: važna je svaka sitnica, sve morate sami da uradite i nema zaštite u vidu provere koju sprovodi prevodilac.

Da problem nije u samom modelu višenitnog rada? Na kraju krajeva, on je preuzet relativno nepromenjen iz sveta proceduralnog programiranja. Možda postoji drugačiji model paralelnog rada, prikladniji za objektno orijentisano programiranje.

Jedan od alternativnih pristupa nazvan je *aktivni objekti* ili *aktori*.[26] Objekti su proglašeni „aktivnim" zato što svaki objekat održava sopstvenu radnu nit i red za poruke; svi zahtevi koji se odnose na taj objekat ulaze u njegov red za čekanje i izvršavaju se jedan po jedan. Dakle, kod aktivnih objekata *serijalizujemo poruke, a ne metode*, što znači da više ne moramo da se štitimo od problema koji nastaju kada se zadatak prekine usred petlje.

Kada aktivnom objektu pošaljete poruku, ona se transformiše u zadatak koji ulazi u objektov red za čekanje na kasnije izvršavanje. Za realizaciju te šeme podesna je klasa **Future** Jave SE5. Evo jednostavnog primera u kojem dve metode stavljaju pozive u red za čekanje:

```
//: paralelno/PrimerAktivnogObjekta.java
// Može da prosledi samo konstante, nepromenljive, "nevezane
// objekte" ili druge aktivne objekte kao argumente
// asinhronim metodama.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PrimerAktivnogObjekta {
  private ExecutorService izvrsilac =
    Executors.newSingleThreadExecutor();
  private Random slucajan = new Random(47);
  // Ubaciću odlaganje nasumičnog trajanja da bih postigao efekat
  // trajanja proračuna:
  private void sacekaj(int faktor) {
    try {
      TimeUnit.MILLISECONDS.sleep(
        100 + slucajan.nextInt(faktor));
    } catch(InterruptedException e) {
      print("sleep() prekinut");
    }
  }
  public Future<Integer>
  calculateInt(final int x, final int y) {
    return izvrsilac.submit(new Callable<Integer>() {
```

[26] Hvala Allenu Holubu što je našao vremena da mi to objasni.

```
      public Integer call() {
        print("pokrećem " + x + " + " + y);
        sacekaj(500);
        return x + y;
      }
    });
  }
  public Future<Float>
  calculateFloat(final float x, final float y) {
    return izvrsilac.submit(new Callable<Float>() {
      public Float call() {
        print("pokrećem " + x + " + " + y);
        sacekaj(2000);
        return x + y;
      }
    });
  }
  public void shutdown() { izvrsilac.shutdown(); }
  public static void main(String[] args) {
    PrimerAktivnogObjekta p1 = new PrimerAktivnogObjekta();
    // Sprečava izuzetak ConcurrentModificationException:
    List<Future<?>> rezultati =
      new CopyOnWriteArrayList<Future<?>>();
    for(float f = 0.0f; f < 1.0f; f += 0.2f)
      rezultati.add(p1.calculateFloat(f, f));
    for(int i = 0; i < 5; i++)
      rezultati.add(p1.calculateInt(i, i));
    print("Završeni pozivi svih asinhronih metoda");
    while(rezultati.size() > 0) {
      for(Future<?> f : rezultati)
        if(f.isDone()) {
          try {
            print(f.get());
          } catch(Exception e) {
            throw new RuntimeException(e);
          }
          rezultati.remove(f);
        }
    }
    p1.shutdown();
  }
} /* Ispis: (85% podudaranja)
Završeni pozivi svih asinhronih metoda
pokrećem 0.0 + 0.0
pokrećem 0.2 + 0.2
0.0
pokrećem 0.4 + 0.4
0.4
pokrećem 0.6 + 0.6
0.8
```

```
pokrećem 0.8 + 0.8
1.2
pokrećem 0 + 0
1.6
pokrećem 1 + 1
0
pokrećem 2 + 2
2
pokrećem 3 + 3
4
pokrećem 4 + 4
6
8
*///:~
```

„Izvršilac jedne niti" proizveden pozivom metode **Executors.newSingleThreadExecutor( )** održava sopstveni neograničeni blokirajući red za čekanje i ima samo jednu nit koja vadi zadatke iz reda i izvršava ih do završetka. U metodama **calculateInt( )** i **calculateFloat( )** moramo samo da pozovemo **submit( )** da bismo novi objekat tipa **Callable** dali kao odgovor na poziv metode, čime pozive pretvaramo u poruke. Telo metode nalazi se unutar metode **call( )** u anonimnoj unutrašnjoj klasi. Obratite pažnju na to da sve metode aktivnih objekata imaju povratnu vrednost tipa **Future**, s generičkim parametrom koji je stvarni povratni tip te metode. Na taj način se poziv metode gotovo trenutno vraća, a pozivalac pomoću tog objekta tipa **Future** saznaje kada je zadatak završen i dobija stvarnu povratnu vrednost. Time je rešen najsloženiji slučaj, a postupak je još jednostavniji ukoliko poziv nema povratnu vrednost.

U metodi **main( )**, pravi se objekat tipa **List<Future<?>>** za hvatanje **Future** objekata koje vraćaju poruke **calculateFloat( )** i **calculateInt( )** poslate aktivnom objektu. Za svaki **Future**, tu listu ispituje metoda **isDone( )** koja ga uklanja iz liste kada završi rad i obradi rezultate. Imajte u vidu da klasa **CopyOnWriteArrayList** čini da listu ne moramo da kopiramo da bismo izbegli izuzetke **ConcurrentModificationException**.

Da bi se sprečio slučajni međusobni uticaj niti, argumenti prosleđeni pozivu metode aktivnog objekta smeju biti: samo za čitanje ili drugi aktivni objekti ili *nevezani objekti* (moj termin), što su objekti koji nemaju veze s drugim zadacima. (To je teško sprovesti, pošto Java to zasad ne podržava.)

Za aktivne objekte važi sledeće:

1. Svaki objekat ima sopstvenu radnu nit.
2. Svaki objekat zadržava potpunu kontrolu nad svim svojim poljima (što je nešto strože nego kod običnih objekata koji imaju *opciju* kontrole svojih polja).
3. Sva komunikacija između aktivnih objekata odvija se u obliku poruka.
4. Sve poruke između aktivnih objekata ulaze u redove za čekanje.

Rezultati su veoma privlačni. Pošto poruka jednog aktivnog objekta drugom može biti blokirana samo ako se odloži njen ulazak u red za čekanje, i pošto je to odlaganje uvek veoma kratko i ne zavisi od drugih objekata, slanje poruke se zapravo ne može blokirati

(najgore što se može desiti je kratko odlaganje). Pošto sistem aktivnih objekata komunicira samo preko poruka, dva objekta ne mogu biti blokirana dok se nadmeću da pozovu metodu nekog trećeg objekta, a to znači da ne može nastati uzajamna blokada, što je veliki korak napred. Pošto radna nit aktivnog objekta u svakom trenutku izvršava samo po jednu poruku, nema takmičenja za resurse i ne moramo da sinhronizujemo metode. Sinhronizacija i dalje postoji, ali na nivou poruka, zato što se pozivi metoda ubacuju u red za čekanje, pa se u svakom trenutku može izvršavati samo jedan od njih.

Nažalost, prevodilac ne daje neposrednu podršku za aktivne objekte, a ručno programiranje po gornjem obrascu previše je zametno. Međutim, oblast aktivnih objekata i aktora se razvija, kao i oblast *agentski orijentisanog* programiranja koja je još zanimljivija. Agenti su zapravo aktivni objekti, ali agentski sistemi se ne menjaju u zavisnosti od računara niti mreže. Ne bi me iznenadilo da agentski orijentisano programiranje nasledi objektno orijentisano programiranje, zato što se u njemu kombinuju objekti i relativno lako rešenje za paralelno izvršavanje.

Više informacija o aktivnim objektima, aktorima i agentima možete naći na Webu; neke ideje realizovane u aktivnim objektima možete naći naročito u C.A.R. Hoareovoj *teoriji komunicirajućih sekvencijalnih procesa* (*Communicating Sequential Processes*, CSP).

Vežba 41: (6) Programu **PrimerAktivnogObjekta.java** dodajte deo za obradu poruka koji nema povratnu vrednost, a poziva se iz metode **main( )**.

Vežba 42: (7) Izmenite **VoskOMatik.java** tako da realizuje aktivne objekte.
Projekat:[27] Pomoću anotacija i Javassista napravite anotaciju klase **@Active** koja odredišnu klasu transformiše u aktivan objekat.

Sažetak

Trebalo je da iz ovog poglavlja naučite osnove paralelnog programiranja pomoću Javinih niti, kako biste shvatili sledeće:

1. Možete izvršavati više nezavisnih zadataka.
2. Morate uzeti u obzir sve moguće probleme prilikom gašenja tih zadataka.
3. Zadaci se međusobno ometaju prilikom korišćenja deljenih resursa. Osnovna alatka za sprečavanje takvih sudara je uzajamno isključiva brava (mutex).
4. Ako zadatke ne projektujete pažljivo, oni se mogu uzajamno blokirati.

Neophodno je naučiti kada se koristi paralelno izvršavanje, a kada da ga izbegavati. Osnovni razlozi za upotrebu jesu:

- Da biste posao podelili na više zadataka i tako efikasnije koristili računar. Time se postiže i (za programera nevidljiva) raspodela zadataka na više procesora.
- Radi bolje organizacije koda.
- Radi veće pogodnosti za korisnika.

[27] Projekti su predlozi koji se mogu koristiti (recimo) za seminarske radove. Vodič s rešenjima ne sadrži rešenja projekata.

Klasičan primer uravnoteženog korišćenja resursa jeste upotreba procesora dok se čeka na završetak ulazno/izlaznih operacija. Bolja organizacija koda obično se postiže u simulacijama. Klasičan primer pogodnosti za korisnika jeste nadziranje dugmeta za prekid tokom dugačkih preuzimanja s mreže.

Dodatna prednost niti je to što „teška" prebacivanja iz konteksta jednog *procesnog* okruženja u kontekst drugog (reda veličine nekoliko hiljada naredaba) zamenjuju „lakim" prebacivanjem iz konteksta jednog *izvršnog* okruženja u kontekst drugog u okviru istog procesnog okruženja (reda veličine nekoliko stotina naredaba). Pošto sve niti jednog procesa dele isti memorijski prostor, pri lakom prebacivanju iz konteksta zamenjuju se samo promenljive izvršavanja programa i lokalne promenljive. S druge strane, promena procesa – dakle, teška promena konteksta – mora da zameni ceo memorijski prostor.

Glavni nedostaci višenitnog izvršavanja su:

1. Usporenje rada dok niti čekaju na deljene resurse.
2. Za upravljanje nitima troši se dodatno procesorsko vreme.
3. U slučaju lošeg projektovanja, programi postaju neopravdano složeni.
4. Postoji verovatnoća da će biti patoloških pojava kao što su nemogućnost dobijanja resursa, trka za resursima, uzajamna blokada i uzajamna blokada uprkos izvršavanju (više niti izvršava pojedinačne zadatke koje sve zajedno ne mogu da završe).
5. Uslovi za paralelno izvršavanje menjaju se u zavisnosti od platforme. (Pišući primere za ovu knjigu, otkrio sam uslove za trku koji su se brzo pokazali na nekim računarima, a uopšte se nisu pojavili na drugim.) Ukoliko program budete razvijali na jednom od računara na kojem se loši delovi ne vide, doživećete neprijatno iznenađenje kada taj program distribuirate.

Jedna od najvećih poteškoća pri višenitnom izvršavanju nastaje zato što više zadataka deli (istovremeno koristi) neki resurs – kao što je memorija objekta – a programer mora sprečiti da više zadataka istovremeno pokuša da čita i menja taj resurs. To se može postići promišljenom upotrebom dostupnih mehanizama za zaključavanje (npr. rezervisane reči **synchronized**). Te alatke jesu neophodne, ali ih morate dobro upoznati pošto će inače ćutke proizvesti uzajamno blokiranje.

Pored toga, za primenu niti je potrebna izvesna umešnost. Java omogućuje pravljenje proizvoljnog broja objekata potrebnog za rešavanje problema – barem u teoriji. (Na primer, pravljenje miliona objekata za inženjersku analizu metodom konačnih elemenata u Javi nije izvodljivo bez projektnog obrasca *Flyweight*.) Međutim, izgleda da postoji gornja granica broja niti koje se mogu napraviti, pošto od neke granice niti postaju zametne. Tu kritičnu granicu nije lako otkriti. Ona se često menja u zavisnosti od operativnog sistema i JVM-a; može biti manja od sto, a i veća od nekoliko hiljada. Pošto se za rešenje problema često pravi tek nekoliko niti, ovo najčešće i nije neko ograničenje, ali u opštijem projektu može vas naterati da usvojite neku od šema paralelne saradnje.

Bez obzira na to koliko jednostavno višenitno programiranje izgleda u određenom jeziku ili biblioteci, smatrajte ga nekom vrstom magije. Uvek vas nešto može ujesti kada se najmanje nadate. Problem večere filozofa zanimljiv je zato što se može napisati tako da se retko javlja uzajamna blokada i zato vam daje utisak da sve lepo radi.

Višenitni rad bi po pravilu trebalo primenjivati pažljivo i štedljivo. Ukoliko problemi s višenitnim radom postanu veliki i složeni, možda bi trebalo da ih rešite na jeziku kao što je *Erlang*. To je jedan od nekoliko *funkcijskih* jezika specijalizovanih za višenitno programiranje. Na takvom jeziku treba napisati samo one delove programa u kojima je neophodno višenitno izvršavanje – ako ih ima mnogo i ako su dovoljno komplikovani da opravdavaju takav pristup.

Literatura za dalje usavršavanje

Nažalost, o paralelnom programiranju postoji mnogo netačnih informacija – to pokazuje koliko je ono samo po sebi zamršeno i koliko je lako pomisliti da ste ga najzad shvatili. (Govorim iz vlastitog iskustva, pošto sam već više puta pomišljao kako sam ga napokon savladao; i ne sumnjam da me u budućnosti čekaju neprijatna otkrića.) Kada uzmete u ruke bilo kakav nov dokument o paralelnom radu, uvek se morate zapitati koliko njegov autor zapravo zna o tome što piše. Ovo su neke od knjiga za koje smem kazati da su pouzdane:

Java Concurrency in Practice, autori Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea (Addison-Wesley, 2006). U suštini, ovo je „who's who" u svetu višenitnog izvršavanja u Javi.

Concurrent Programming in Java (drugo izdanje), autor Doug Lea (Addison-Wesley, 2000). Mada je knjiga objavljena znatno pre Jave SE5, dobar deo nje Doug je pretočio u nove biblioteke **java.util.concurrent**, što je čini neophodnom za potpuno upoznavanje paralelnog programiranja. Ona prevazilazi paralelnost u Javi i razmatra trenutno stanje u više jezika i tehnologija. Iako može biti teška, zaslužuje da je pročitate više puta (najbolje na po nekoliko meseci, kako biste stigli da usvojite pročitano). Doug je jedan od retkih ljudi koji zaista razumeju paralelnost, pa će vam se trud isplatiti.

The Java Language Specification (treće izdanje), poglavlje 17, autori Gosling, Joy, Steele i Bracha (Addison-Wesley, 2005). Tehnička specifikacija dostupna je i u obliku elektronskog dokumenta, na adresi: *http://java.sun.com/docs/books/jls*.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

Grafička korisnička okruženja

Jedan od osnovnih principa projektovanja glasi: „neka što je jednostavno postane lako, a ono što je teško postane moguće".[1]

PRVOBITNO ZAMIŠLJENA SVRHA BIBLIOTEKE GRAFIČKOG KORISNIČKOG OKRUŽENJA (GKO, engl. *graphical user interface, GUI*) u Javi 1.0 bila je da se programeru omogući da napravi grafičke programe koji izgledaju dobro na svim platformama. Taj cilj nije ostvaren. Umesto toga, u Javi 1.0 postojao je *komplet alatki za apstraktne prozore* (engl. *Abstract Window Toolkit, AWT*) koji je davao grafičko okruženje podjednako osrednjeg izgleda na svim sistemima. Osim toga, ta biblioteka je jako ograničena: dozvoljeno je korišćenje samo četiri fonta, a ne može se pristupiti nijednom naprednijem elementu grafičkog okruženja iz nekog operativnog sistema. Programerski model AWT iz Jave 1.0 istovremeno je bio nezgrapan i nije bio objektno orijentisan. Polaznik jednog od mojih seminara (koji je radio u kompaniji Sun tokom pisanja Jave) objasnio je razlog: prvobitni AWT je smišljen, projektovan i realizovan za mesec dana. To je sigurno čudo produktivnosti, ali i lekcija o tome zašto je projektovanje važno.

Situacija se poboljšala nakon uvođenja modela AWT iz Jave 1.1 koji koristi mnogo jasniji, objektno orijentisan pristup, a podržava i zrna Jave, tj. model programiranja komponenata usmeren ka lakom pravljenju vizuelnih razvojnih okruženja. U Javi 2 (JDK 1.2) dovršena je transformacija starog AWT-a iz Jave 1.0 tako što je sve zamenjeno *Javinim osnovnim klasama* (engl. *Java Foundation Classes, JFC*) čiji se grafički deo zove Swing. To je bogat skup Javinih zrna koja se lako koriste i razumeju, a pomoću njih se u vizuelnim razvojnim alatima (prevlačenjem i otpuštanjem, kao i ručnim programiranjem) može napraviti grafičko okruženje kojim možete da budete zadovoljni. Izgleda da pravilo „treće prepravke" koje važi u industriji softvera (proizvod nije dobar sve dok se triput ne prepravi) važi i za programske jezike.

Ovo poglavlje posvećeno je isključivo modernoj biblioteci Swing iz Jave 2, pri čemu se opravdano pretpostavilo da se grafička okruženja u Javi pišu pomoću Swinga.[2] Ako iz nekog razloga morate da koristite prvobitni, stari AWT (radi podrške starom kodu ili usled ograničenja koja nameće čitač), uvod u tu problematiku možete da pronađete u prvom izdanju ove knjige na adresi *www.MindView.net*. Obratite pažnju na to da Java i dalje sadrži neke AWT komponente i da ih u nekim situacijama morate koristiti.

Imajte u vidu da ovo nije sveobuhvatan pregled komponenata grafičke biblioteke Swing, niti svih metoda opisanih klasa. Grafička biblioteka Swing je ogromna, a iz ovog poglavlja samo treba da naučite osnovne pojmove i da upoznate principe projekta. Ukoliko vam treba više od toga, biblioteka Swing verovatno može da pruži ono što želite ako ste volini da je istražujete.

Ovde ću pretpostavljati da ste s lokacije *java.sun.com* preuzeli i instalirali (besplatnu) dokumentaciju Javine biblioteke u HTML formatu i da ćete pregledati klase iz paketa

---

[1] Varijanta ovoga se zove „princip najmanjeg iznenađenja", što znači da korisnika ne bi trebalo iznenađivati.

[2] IBM je za svoj program za uređivanje teksta Eclipse (*www.Eclipse.org*) napravio novu GUI biblioteku otvorenog izvornog koda koja je alternativa Swingu; bolje je predstavljena u nastavku poglavlja.

javax.swing u toj dokumentaciji kako biste saznali sve detalje o biblioteci Swing i metode iz nje. Pomoć možete potražiti i na Webu, ali najbolje je da krenete od Sunovog udžbenika za Swing na adresi *http://java.sun.com/docs/books/tutorial/uiswing*.

Postoji veliki broj (prilično debelih) knjiga posvećenih isključivo biblioteci Swing i njih ćete sigurno proučavati ako vam zatreba šire znanje, ili ako poželite da promenite podrazumevano ponašanje biblioteke Swing.

Kako budete savladavali biblioteku Swing, otkrivaćete sledeće:

1. Biblioteka Swing je mnogo bolji model za programiranje od onih koje ste verovatno sretali u drugim jezicima i razvojnim okruženjima. Osnovna struktura za tu biblioteku jesu zrna Jave (koja će biti objašnjena pri kraju ovog poglavlja).
2. Alatke za pravljenje grafičkih okruženja (vizuelna okruženja za programiranje) obavezan su vid kompletnog razvojnog okruženja za Javu. Zrna Jave i Swing omogućuju razvojnom okruženju da piše kôd umesto vas, dok vi razmeštate komponente na obrasce pomoću grafičkih alatki. To u velikoj meri ubrzava projektovanje grafičkog okruženja i omogućuje slobodnije eksperimentisanje i isprobavanje dizajna, što posledično dovodi do izbora boljeg okruženja.
3. Jednostavnost i dobar projekat grafičke biblioteke Swing znači da će čak i ako koristite vizuelnu alatku, umesto da ručno pišete kôd, dobijeni kôd biti razumljiv. To rešava veliki problem s korišćenjem ranijih vizuelnih alatki kojima se često generisao nerazumljiv kôd.

Grafička biblioteka Swing sadrži sve komponente koje se očekuju u modernom korisničkom okruženju, od dugmadi sa slikama, preko stabala do tabela. To je velika biblioteka, ali složenost odgovara zadatku koji se obavlja: ako je nešto jednostavno, ne morate da pišete mnogo koda, ali ukoliko pokušate da uradite nešto teže, i kôd postaje srazmerno složeniji.

Dobar deo privlačnosti biblioteke Swing potiče od nečega što bi se moglo nazvati „ortogonalnost korišćenja". Odnosno, kada jednom steknete opšte ideje o ovoj biblioteci, moći ćete da ih primenjujete svuda. Dok sam pisao primere iz ovog poglavlja, najčešće sam već na osnovu imena metode mogao da zaključim šta ona radi, i to zahvaljujući standardu za imenovanje. To je, sasvim sigurno, obeležje dobre biblioteke. Pored toga, komponente se obično mogu dodavati drugim komponentama i sve će raditi ispravno.

Navigacija pomoću tastature je automatska: aplikaciju napisanu pomoću Swinga možete da pokrenete i bez miša, što ne zahteva nikakvo dodatno programiranje. Podrška za pomeranje sadržaja je izvanredna – treba samo da omotate komponentu u objekat klase **JScrollPane** dok je dodajete u obrazac. Za dodavanje komponenata poput priručnih saveta (engl. *tooltips*) obično je dovoljan samo jedan red koda.

Cela biblioteka Swing napisana je na Javi radi prenosivosti.

Swing podržava i prilično radikalnu mogućnost nazvanu „prilagodljiv izgled i ponašanje" (engl. *pluggable look & feel*), što znači da izgled grafičkog okruženja može dinamički da se promeni da bi se prilagodio očekivanjima korisnika na različitim platformama i operativnim sistemima. Moguće je (mada teško) napraviti čak i sopstveni izgled okruženja. Neka takva okruženja možete naći i na Webu.[1]

[1] Meni se najviše sviđa izgled i ponašanje Salveta (engl. *napkin*) autora Kena Arnolda, u kojem prozori izgledaju kao da su skicirani na salveti. Posetite *http://napkinlaf.sourceforge.net*.

Uprkos svim svojim pozitivnim aspektima, Swing nije za svakoga, niti je sve probleme s korisničkim okruženjima rešio onako kako su njihovi projektanti zamislili. Na kraju poglavlja razmotrićemo dve alternative za Swing: IBM-ov SWT, razvijen za editor teksta Eclipse, ali besplatno dostupan kao samostalna GUI biblioteka otvorenog izvornog koda, i Flex kompanije Adobe, alatku za razvoj klijentskih okruženja Web aplikacija u Flashu.

Apleti

Kada se Java tek pojavila, dobar deo halabuke oko tog jezika izazvao je *aplet* – program koji se preuzima sa Interneta da bi se izvršavao u čitaču Weba (unutar tzv. *bazena s peskom*, da ne bi naškodio klijentskom računaru). Predviđalo se da će Java aplet postati sledeća faza u evoluciji Interneta i u mnogima od prvih knjiga o Javi pretpostavljalo se da je čitalac zainteresovan za taj jezik prvenstveno zato što želi da piše aplete.

Iz raznih razloga, ta revolucija se nije odigrala. Dobar deo problema činilo je to što većina računara nema potreban Java softver za izvršavanje apleta, a većina korisnika ne želi da preuzme i instalira paket od 10 MB da bi prikazali nešto što su slučajno pokupili s Weba. Već i spominjanje nečeg sličnog dovoljno je da uplaši mnoge korisnike. Java apleti nikada nisu dostigli kritičnu masu kao sistem za distribuciju aplikacija klijentima, i premda se apleti i dalje povremeno sreću, po pravilu pripadaju prošlosti računarstva.

To ne znači da apleti nisu zanimljiva i vredna tehnologija. Ukoliko možete obezbediti da svi korisnici imaju instaliran JRE (recimo u jednoj kompaniji), onda bi apleti (ili JNLP/Java Web Start koje ćemo opisati u nastavku poglavlja) mogli biti savršeni za distribuciju klijentskih programa i automatsko ažuriranje svih računara i pri tom bi se izbegli troškovi i trud koji obično prate instalaciju i distribuciju novog softvera.

Uvod u tehnologiju apleta naći ćete u mrežnim dodacima ove knjige, na adresi *www.MindView.net*.

Osnove Swinga

Većina Swing aplikacija, tj. grafičkih korisničkih okruženja, gradi se unutar jednostavnog objekta tipa **JFrame**, koji pravi prazan prozor (ili okvir, engl. *frame*) u operativnim sistemima svih korisnika. Naslov prozora se ovako zadaje pomoću konstruktora objekta tipa **JFrame**:

```
//: gui/ZdravoSwing.java
import javax.swing.*;

public class ZdravoSwing {
  public static void main(String[] args) {
    JFrame prozor = new JFrame("Zdravo Swing");
    prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    prozor.setSize(300, 100);
    prozor.setVisible(true);
  }
} ///:~
```

Metoda **setDefaultCloseOperation( )** kazuje objektu tipa **JFrame** šta da radi kada korisnik izvede manevar gašenja. Konstanta **EXIT_ON_CLOSE** saopštava mu da izađe iz programa. Podrazumevano se u tom slučaju ne radi ništa, pa ako ne pozovete metodu **setDefaultCloseOperation( )** ili ne napišete ekvivalentan kôd za vaš prozor, aplikacija se neće završiti.

setSize( ) zadaje veličinu prozora u pikselima.

Obratite pažnju na poslednji red:

```
prozor.setVisible(true);
```

Bez njega, na ekranu ne biste videli ništa.

U prazan prozor (objekat tipa **JFrame**) dodaćemo natpis (objekat tipa **JLabel**):

```
//: gui/ZdravoNatpis.java
import javax.swing.*;
import java.util.concurrent.*;

public class ZdravoNatpis {
  public static void main(String[] args) throws Exception {
    JFrame prozor = new JFrame("Zdravo, Swing");
    JLabel natpis = new JLabel("Jedan natpis");
    prozor.add(natpis);
    prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    prozor.setSize(300, 100);
    prozor.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    natpis.setText("Ej! Ovo se promenilo!");
  }
} ///:~
```

Tekst **JLabel** natpisa menja se nakon jedne sekunde. Iako je to zabavno i u tako jednostavnom programu bezbedno, baš i nije dobro da glavna nit **main( )** piše neposredno u GUI komponente. Swing ima sopstvenu nit za primanje UI događaja i ažuriranje ekrana. Ako sadržaj ekrana počnete da menjate pomoću drugih niti, možete izazvati sudare i uzajamnu blokadu opisane u poglavlju *Paralelno izvršavanje*.

Umesto toga, druge niti – kao prethodno spomenuta nit **main( )** – treba Swingovoj *niti za otpremu događaja* (engl. *event dispatch thread*) da šalju (engl. *submit*) zadatke na izvršavanje.[4] To se postiže predajom zadatka metodi **SwingUtilities.invokeLater( )** koja zadatak smešta u *red za čekanje događaja* (engl. *event queue*), iz koga ih nit za otpremu događaja (u nekom trenutku) vadi i izvršava. U prethodnom primeru to bi izgledalo ovako:

```
//: gui/PodnosenjeZadatkaZaRukovanjeNatpisom.java
import javax.swing.*;
import java.util.concurrent.*;
```

[4] Strogo uzev, nit za otpremu događaja pripada biblioteci AWT.

```
public class PodnosenjeZadatkaZaRukovanjeNatpisom {
  public static void main(String[] args) throws Exception {
    JFrame prozor = new JFrame("Zdravo, Swing");
    final JLabel natpis = new JLabel("Jedan natpis");
    prozor.add(natpis);
    prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    prozor.setSize(300, 100);
    prozor.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        natpis.setText("Ej! Ovo se promenilo!");
      }
    });
  }
} ///:~
```

Sada natpisom više ne rukujemo neposredno, nego šaljemo objekat koji realizuje interfejs **Runnable**, a stvarno rukovanje obavlja nit za otpremu događaja kada u redu za čekanje događaja dođe do tog zadatka. Kada bude izvršavala taj zadatak, ona neće raditi ništa drugo, pa ne može nastati sudar – *ukoliko* se *sav* kôd u vašem programu bude pridržavao tog pristupa, tj. rukovao ekranom pomoću metode **SwingUtilities.invokeLater( )**. To se odnosi i na pokretanje samog programa – **main( )** ne bi trebalo da poziva Swing metode kao u gornjem programu, nego da podnese zadatak redu za otpremu događaja.[5] Dakle, pravilno napisan program izgledao bi otprilike ovako:

```
//: gui/SlanjeSwingPrograma.java
import javax.swing.*;
import java.util.concurrent.*;

public class SlanjeSwingPrograma extends JFrame {
  JLabel natpis;
  public SlanjeSwingPrograma() {
    super("Zdravo, Swing");
    natpis = new JLabel("Jedan natpis");
    add(natpis);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 100);
    setVisible(true);
  }
  static SlanjeSwingPrograma psp;
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { psp = new SlanjeSwingPrograma(); }
    });
```

[5] Ova praksa je novina u Javi SE5, pa je gomila starih programa ne podržava. To ne znači da su njihovi autori bili neznalice, nego da se preporučena praksa stalno menja.

```
      TimeUnit.SECONDS.sleep(1);
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          psp.natpis.setText("Ej! Ovo se promenilo!");
        }
      });
    }
} ///:~
```

Obratite pažnju na to da poziv metode **sleep( )** *nije* unutar konstruktora. Ako ga stavite tamo, prvobitni tekst natpisa („Jedan natpis") uopšte neće biti prikazan, pošto u tom slučaju konstruktor neće završiti dok se ne završi metoda **sleep( )** i novi natpis ne bude umetnut. Ali ako je poziv metode **sleep( )** unutar konstruktora ili bilo koje UI operacije, to znači da tokom spavanja koje prouzrokuje metoda **sleep( )** zadržavate izvršavanje niti za otpremu događaja, što je po pravilu loše.

Vežba 1: (1) Izmenite **ZdravoSwing.java** tako da dokažete da se aplikacija ne gasi bez poziva metode **setDefaultCloseOperation( )**.

Vežba 2: (2) Izmenite **ZdravoNatpis.java** tako da pokažete da je dodavanje natpisa dinamičko – stavite u prozor pseudoslučajan broj natpisa.

Alatka za prikazivanje

Objedinićemo navedene ideje i smanjiti količinu redundantnog koda sledećom alatkom za prikazivanje koju ćemo koristiti u ostalim primerima biblioteke Swing u ostatku poglavlja:

```
//: net/mindview/util/SwingKonzola.java
// Alatka za pokretanje Swing primera,
// bilo apleta bilo objekata tipa JFrame, s konzole.
package net.mindview.util;
import javax.swing.*;

public class SwingKonzola {
  public static void
  run(final JFrame f, final int sirina, final int visina) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        f.setTitle(f.getClass().getSimpleName());
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(sirina, visina);
        f.setVisible(true);
      }
    });
  }
} ///:~
```

Ovu alatku ćete možda poželeti da koristite na više mesta, pa je smeštena u biblioteku **net.mindview.util**. Da bi mogla da je koristi, aplikacija mora biti u objektu tipa **JFrame** (kao svi primeri u ovoj knjizi). Statična metoda **run( )** zadaje naslov prozora jednak Class imenu objekta tipa **JFrame**.

Vežba 3: (3) Izmenite **SlanjeSwingPrograma.java** tako da upotrebljava program **SwingKonzola**.

Pravljenje dugmeta

Pravljenje dugmeta je prilično jednostavno: samo pozovite konstruktor klase **JButton** i prosledite mu natpis koji treba da se pojavi na dugmetu. Kasnije ćete videti da možete uraditi i nešto naprednije, npr. staviti slike na dugmad.

Referenca na dugme se obično čuva unutar klase da bi kasnije moglo ponovo da joj se pristupi.

JButton je Swing komponenta sa sopstvenim prozorčićem koja se automatski iscrtava tokom ažuriranja prikaza. To znači da dugme ili neki drugi kontrolni objekat ne crtate ručno, već ih samo postavljate na obrazac i omogućujete im da se automatski iscrtavaju. Dugme se na obrazac postavlja obično unutar konstruktora:

```
//: gui/Dugme1.java
// Stavljanje dugmadi u Swing aplikaciju.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class Dugme1 extends JFrame {
  private JButton
    d1 = new JButton("Dugme 1"),
    d2 = new JButton("Dugme 2");
  public Dugme1() {
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
  }
  public static void main(String[] args) {
    run(new Dugme1(), 200, 100);
  }
} ///:~
```

Ovde je dodato nešto novo: pre nego što se element postavi u okno sadržaja, dodeljuje mu se nov „raspoređivač" tipa **FlowLayout**. Raspoređivač (engl. *Layout manager*) opisuje način na koji okno implicitno odlučuje o tome gde će postaviti kontrolni objekat na obrascu. Aplet obično koristi raspoređivač tipa **BorderLayout**, ali to nam u ovom slučaju ne odgovara zato što se na taj način (kao što ćete saznati u nastavku poglavlja) svaki kontrolni objekat potpuno prekriva novim objektima u istoj oblasti. Međutim, raspoređivač

tipa **FlowLayout** prouzrokuje ravnomerno ređanje kontrolnih objekata po obrascu, sleva udesno i odozgo naniže.

Vežba 4: (1) Pokažite da se bez poziva metode **setLayout( )** u programu **Dugme1.java** prikazuje samo jedno dugme.

Hvatanje događaja

Ako prevedete i izvršite prethodni program, primetićete da se neće dešavati ništa kada pritiskate dugmad. U tu svrhu morate sami da napišete kôd. Suština programiranja upravljanog događajima, što čini veliki deo programiranja grafičkih korisničkih okruženja, jeste povezivanje događaja s kodom koji odgovara na njih.

Ovo se u grafičkoj biblioteci Swing postiže jasnim razdvajanjem okruženja (grafičkih komponenata) i realizacije (koda koji želite da izvršavate kada se desi događaj). Svaka komponenta može da prijavljuje događaje, i svaki od njih može da prijavljuje zasebno. Dakle, ako niste zainteresovani, na primer, za to da li je pokazivač miša prešao preko dugmeta, nećete osluškivati taj događaj. To je veoma jednostavan i elegantan način za pisanje programa vođenih događajima. Kada shvatite osnovne pojmove, lako ćete koristiti Swing komponente koje nikada niste videli. Zapravo, ovaj model se primenjuje na sve što se može okarakterisati kao zrno Jave (o čemu ćete saznati više u nastavku poglavlja).

Za početak, usredsredićemo se na glavne događaje komponenata. Za dugme tipa **JButton**, „glavni događaj" je njegovo pritiskanje. Da biste objavili svoje zanimanje za događaj pritiskanja dugmeta, pozivate metodu **addActionListener( )** klase **JButton**. Ta metoda očekuje argument koji realizuje interfejs **ActionListener** što sadrži samo jednu metodu **actionPerformed( )**. Dakle, da biste povezali kôd s dugmetom tipa **JButton**, treba samo da realizujete interfejs **ActionListener** u klasi i da prosledite objekat te klase dugmetu metodom **addActionListener( )**. Metoda prosledenog objekta pozivaće se kad god se pritisne dugme (to se obično naziva *povratni poziv*, engl. *callback*).

Šta bi trebalo da bude rezultat pritiskanja tog dugmeta? Želeli bismo da vidimo da se nešto menja na ekranu, pa će biti predstavljena nova Swing komponenta: **JTextField**. To je jednoredno polje za unos teksta, ili, u ovom slučaju, polje u kome program može menjati tekst. Iako postoji više načina da se napravi objekat klase **JTextField**, najjednostavniji način je da se samo saopšti konstruktoru koliko treba da bude široko to polje. Kada se polje za tekst postavi na obrazac, njegov sadržaj možete da menjate metodom **setText( )** (klasa **JTextField** sadrži i mnoge druge metode, ali njih morate da potražite sami u HTML dokumentaciji na adresi *http://java.sun.com*). Evo kako to izgleda:

```
//: gui/Dugme2.java
// Reagovanje na pritiskanja dugmeta.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;
```

```
public class Dugme2 extends JFrame {
  private JButton
    d1 = new JButton("Dugme 1"),
    d2 = new JButton("Dugme 2");
  private JTextField txt = new JTextField(10);
  class PrijemnikDugmeta implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String ime = ((JButton)e.getSource()).getText();
      txt.setText(ime);
    }
  }
  private PrijemnikDugmeta pd = new PrijemnikDugmeta();
  public Dugme2() {
    d1.addActionListener(pd);
    d2.addActionListener(pd);
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Dugme2(), 200, 150);
  }
} ///:~
```

Jednoredno polje za tekst pravi se i smešta na obrazac na isti način kao dugme ili neka druga Swing komponenta. Gornji program se razlikuje po pravljenju pomenute klase **PrijemnikDugmeta** koja realizuje interfejs **ActionListener**. Argument metode **actionPerformed**( ) tipa je **ActionEvent** i sadrži sve informacije o događaju i mestu gde je nastao događaj. U ovom slučaju, želeo sam da opišem dugme koje je pritisnuto: metoda **getSource**( ) vraća objekat gde se desio događaj, a ja sam (eksplicitnom konverzijom) učinio da njegov tip bude **JButton.** Metoda **getText**( ) vraća tekst koji se nalazi na dugmetu, i ja sam ga stavio u polje tipa **JTextField** kako bih dokazao da se kôd zaista poziva kada se dugme pritisne.

U konstruktoru se metoda **addActionListener**( ) koristi za registrovanje objekata klase **PrijemnikDugmeta** kao prijemnika događaja oba dugmeta.

Često je zgodnije da interfejs **ActionListener** bude realizovan kao anonimna unutrašnja klasa, naročito zato što se obično koristi samo jedna instanca svake klase ovog tipa. Program **Dugme2.java** može se promeniti tako da koristi anonimnu unutrašnju klasu na sledeći način:

```
//: gui/Dugme2b.java
// Korišćenje anonimnih unutrašnjih klasa.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;
```

```
public class Dugme2b extends JFrame {
  private JButton
    d1 = new JButton("Dugme 1"),
    d2 = new JButton("Dugme 2");
  private JTextField txt = new JTextField(10);
  private ActionListener pd = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String ime = ((JButton)e.getSource()).getText();
      txt.setText(ime);
    }
  };
  public Dugme2b() {
    b1.addActionListener(pd);
    b2.addActionListener(pd);
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Dugme2b(), 200, 150);
  }
} ///:~
```

U primerima iz ove knjige nadalje će se koristiti anonimne unutrašnje klase (kad god je to moguće).

Vežba 5: (4) Napišite aplikaciju uz korišćenje klase **SwingKonzola**. Neka sadrži polje za tekst i tri dugmeta. Kada se pritisne neko dugme, u polju treba da se pojavi nekakav tekst.

Višeredna polja za tekst

Klasa **JTextArea** slična je klasi **JTextField**, ali može da sadrži više redova teksta i da ima više funkcija. Naročito korisna metoda je **append**( ); pomoću nje možete lako da smestite izlaz programa u polje za tekst. Pošto unutar višerednog polja za tekst možete da se krećete i unazad, Swing program je poboljšanje u poređenju sa onim što je dosad moglo da se postigne pomoću programa s komandne linije koji ispisuju rezultate na standardnom izlaznom uređaju. Kao primer, razmotrimo sledeći program koji popunjava višeredno polje za tekst izlaznim podacima generatora **Countries** iz poglavlja *Detaljno razmatranje kontejnera*:

```
//: gui/TextArea.java
// Korišćenje kontrole JTextArea.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingKonzola.*;
```

```
public class TextArea extends JFrame {
  private JButton
    b = new JButton("Dodaj podatke"),
    c = new JButton("Obriši podatke");
  private JTextArea t = new JTextArea(20, 40);
  private Map<String,String> m =
    new HashMap<String,String>();
  public TextArea() {
    // Iskoristi sve podatke:
    m.putAll(Countries.capitals());
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(Map.Entry me : m.entrySet())
          t.append(me.getKey() + ": "+ me.getValue()+"\n");
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText("");
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(b);
    add(c);
  }
  public static void main(String[] args) {
    run(new TextArea(), 475, 425);
  }
} ///:~
```

U konstruktoru se kontejner tipa **Map** popunjava imenima svih zemalja i njihovih glavnih gradova. Obratite pažnju na to da se prijemnik tipa **ActionListener** za oba dugmeta pravi i dodaje bez definisanja posredne promenljive, pošto se on ne poziva nigde više u programu. Dugme „Dodaj podatke" formatira i dodaje sve podatke, dok dugme „Obriši podatke" koristi metodu **setText( )** za uklanjanje sadržaja iz polja za tekst **JTextArea**.

Kada se polje tipa **JTextArea** dodaje u **JFrame** prozor, treba ga omotati u okno s klizačima (**JScrollPane**) da bi se mogao pomerati sadržaj kada se na ekranu pojavi previše teksta. To je sve što treba da uradite da biste potpuno omogućili pomeranje sadržaja na ekranu. Pošto sam probao to da uradim i pomoću nekih drugih alata za programiranje grafičkih korisničkih okruženja, bio sam oduševljen jednostavnošću i dobrim dizajnom komponenata kao što je **JScrollPane**.

Vežba 6: (7) Pretvorite **znakovninizovi/TestRegularExpression.java** u interaktivan Swing program koji omogućuje umetanje ulaznog niza znakova u jedno višeredno tekstualno polje i regularnog izraza u jedno jednoredno tekstualno polje. Rezultate (primene regularnog izraza na umetnuti znakovni niz) prikažite u drugom višerednom tekstualnom polju.

Vežba 7: (5) Napravite aplikaciju pomoću klase **SwingKonzola** i dodajte sve Swing komponente koje imaju metodu **addActionListener( )**. (Potražite ih u HTML dokumentaciji na adresi *http://java.sun.com*. Savet: upotebite indeks.) Uhvatite njihove događaje i za svaki prikažite odgovarajuću poruku u polju za tekst.

Vežba 8: (6) Gotovo sve komponente biblioteke Swing izvedene su iz klase **Component** koja ima metodu **setCursor( )**. Potražite je u HTML dokumentaciji za Javu. Napišite aplikaciju i promenite pokazivač miša u neki iz klase **Cursor**.

Raspoređivanje elemenata

Način na koji se komponente raspoređuju u Javi verovatno je drugačiji nego u svim ostalim sistemima za grafička okruženja koje ste koristili. Prvo, sve se obavlja programski: ne postoje „resursi" koji upravljaju postavljanjem komponenata. Drugo, načinom na koji se komponente postavljaju na obrazac ne upravlja apsolutno pozicioniranje, već raspoređivač koji odlučuje o položaju komponenata na osnovu redosleda kojim ih dodajete. Veličina, oblik i položaj komponenata znatno se razlikuju od jednog raspoređivača do drugog. Pored toga, raspoređivači se prilagođavaju dimenzijama apleta ili prozora aplikacije, pa ako se promeni dimenzija prozora, u saglasnosti s tim mogu da se promene veličina, oblik i položaj komponenata.

Klase **JApplet**, **JFrame**, **JWindow**, **JDialog**, **JPanel** itd. mogu da sadrže i prikazuju komponente. Klasa **Container** sadrži metodu **setLayout( )** koja omogućuje zadavanje raspoređivača. U ovom odeljku proučićemo različite raspoređivače tako što ćemo postavljati dugmad (jer je to najjednostavnije). Nećemo obrađivati nikakve događaje, pošto ovi primeri samo pokazuju kako se komponente raspoređuju.

Raspoređivač BorderLayout

Prozor podrazumevano koristi raspoređivač tipa **BorderLayout**. Ako ne zadate drugačije, on uzima sve objekte koje mu dodajete i smešta ih u centar, i pri tom ih proširuje sve do ivica.

Ovaj raspoređivač koristi četiri granične oblasti i centralnu oblast. Kada na pano koji koristi raspoređivač **BorderLayout** dodate nešto, možete da upotrebite preklopljenu metodu **add( )** čiji je prvi argument konstantna vrednost. Ta vrednost može da bude nešto od sledećeg:

BorderLayout.NORTH	(vrh)
BorderLayout.SOUTH	(dno)
BorderLayout.EAST	(desno)
BorderLayout.WEST	(levo)
BorderLayout.CENTER	(popunjavanje sredine, do ostalih komponenata ili ivica)

Ako ne zadate oblast za postavljanje objekta, podrazumevano se koristi **CENTER**. Evo jednostavnog primera. Raspoređivač se ne zadaje izričito jer **JFrame** podrazumevano koristi raspoređivač tipa **BorderLayout**:

```
//: gui/BorderLayout1.java
// Prikazuje raspoređivač BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class BorderLayout1 extends JFrame {
  public BorderLayout1() {
    add(BorderLayout.NORTH, new JButton("Sever"));
    add(BorderLayout.SOUTH, new JButton("Jug"));
    add(BorderLayout.EAST, new JButton("Istok"));
    add(BorderLayout.WEST, new JButton("Zapad"));
    add(BorderLayout.CENTER, new JButton("Centar"));
  }
  public static void main(String[] args) {
    run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

Ako se element postavlja u bilo koju oblast osim **CENTER**, sažima se da bi se uklopio u najmanji mogući prostor duž jedne dimenzije, a maksimalno se produžava duž druge dimenzije. Oblast **CENTER**, međutim, proširuje objekat u obe dimenzije da bi zauzeo sredinu.

Raspoređivač FlowLayout

Ovaj raspoređivač samo „reda" komponente na obrazac, sleva udesno, sve dok se ne popuni jedan red, a zatim prelazi u sledeći red i nastavlja redanje.

Evo primera koji bira raspoređivač tipa **FlowLayout** i zatim postavlja dugmad na obrazac. Primetićete da u ovakvom načinu raspoređivanja komponente zadržavaju svoju „prirodnu" veličinu. Dugme je, na primer, taman toliko da se može prikazati znakovni niz koji sadrži.

```
//: gui/FlowLayout1.java
// Prikazuje raspoređivač FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class FlowLayout1 extends JFrame {
  public FlowLayout1() {
    setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      add(new JButton("Dugme " + i));
  }
```

```
  public static void main(String[] args) {
    run(new FlowLayout1(), 300, 300);
  }
} ///:~
```

Raspoređivač tipa **FlowLayout** potpuno smanjuje sve komponente, pa su rezultati pomalo iznenađujući. Na primer, pošto će natpis tipa **JLabel** biti veličine znakovnog niza koji sadrži, pokušaj da se tekst poravna udesno nema efekta ako se koristi raspoređivač **FlowLayout**.

Vodite računa o tome da će raspoređivač prerasporediti komponente ukoliko promenite veličinu prozora.

Raspoređivač GridLayout

GridLayout omogućuje izradu tabele komponenata, a kako ih dodajete, one se smeštaju sleva udesno i odozgo naniže u tabeli. U konstruktoru se zadaje potreban broj redova i kolona i one se prikazuju u jednakim proporcijama.

```
//: gui/GridLayout1.java
// Prikazuje raspoređivač GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class GridLayout1 extends JFrame {
  public GridLayout1() {
    setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      add(new JButton("Dugme " + i));
  }
  public static void main(String[] args) {
    run(new GridLayout1(), 300, 300);
  }
} ///:~
```

U ovom slučaju postoji 21 polje i samo 20 dugmadi. Poslednje polje je ostalo prazno zato što **GridLayout** ne vrši preraspodelu.

Raspoređivač GridBagLayout

GridBagLayout obezbeđuje potpunu kontrolu nad odlučivanjem o rasporedu i formatiranju oblasti u prozoru kada mu se promeni veličina. Međutim, to je istovremeno i najsloženiji raspoređivač koji je veoma teško razumeti. Namenjen je prvenstveno automatskom generisanju koda pomoću alatke za pravljenje grafičkih aplikacija (dobri vizuelni alati koriste **GridBagLayout** umesto apsolutnog pozicioniranja). Ako je program toliko složen da mislite kako treba koristiti raspoređivač tipa **GridBagLayout**, trebalo bi da upotrebite vizuelnu alatku. Ako baš želite da saznate sve detalje o ovome, potražite ih u nekoj knjizi potpuno posvećenoj grafičkoj biblioteci Swing.

Umesto raspoređivača **GridBagLayout** mogli biste da upotrebite **TableLayout** koji nije deo biblioteke Swing ali se može preuzeti na adresi *http://java.sun.com*. Ta komponenta je izgrađena povrh raspoređivača **GridBagLayout** i skriva najveći deo njegove složenosti, pa uveliko pojednostavljuje njegovo korišćenje.

Apsolutno pozicioniranje

Apsolutan položaj grafičkih komponenata moguće je zadati i na sledeći način:

1. Uklonite raspoređivač metodom **Container.setLayout(null)**.
2. U zavisnosti od verzije jezika, za sve komponente pozovite metodu **setBounds( )** ili **reshape( )**, prosleđujući joj okvirni pravougaonik čije su koordinate izražene pikselima. To možete da uradite u konstruktoru ili u metodi **paint( )**, u zavisnosti od toga šta želite da postignete.

Neke alatke za pravljenje grafičkih okruženja često koriste ovakav pristup, ali to obično nije najbolji način za generisanje koda.

Raspoređivač BoxLayout

Pošto korisnici nisu razumevali klasu **GridBagLayout** i teško su je primenjivali, u biblioteku Swing dodata je i klasa **BoxLayout** koja pruža mnoge prednosti raspoređivača **GridBagLayout**, a nije toliko složena. Ovakvo raspoređivanje često možete da koristite pri ručnom razmeštanju elemenata (i ovde, ako program postane previše složen, upotrebite vizuelnu alatku). Raspoređivač **BoxLayout** omogućuje i vertikalno i horizontalno kontrolisanje položaja komponenata, a za kontrolisanje prostora između komponenata koriste se „podupirači" i „lepak". Elementarne primere upotrebe raspoređivača **BoxLayout** potražite u mrežnim dodacima ove knjige, na adresi *www.MindView.net*.

Koji je pristup najbolji?

Biblioteka Swing je moćna i može da uradi mnogo toga uz samo nekoliko redova koda. Primeri u ovoj knjizi prilično su jednostavni, a radi učenja vredi ih napisati ručno. Prilično mnogo se može postići kombinovanjem jednostavnih raspoređivača. Međutim, u određenom trenutku, ručno pisanje koda za grafičko okruženje postaje besmisleno, previše složeno i predstavlja traćenje vremena. Projektanti Jave i Swinga orijentisali su jezik i biblioteke tako da podržavaju alatke za pravljenje grafičkih okruženja i te alatke znatno ubrzavaju programiranje. Pod uslovom da razumete šta se dešava u raspoređivaču i kako da se izborite s događajima (što će biti objašnjeno u nastavku), nije naročito važno da zaista znate sve detalje ručnog raspoređivanja komponenata. Dozvolite odgovarajućoj alatki da to uradi umesto vas (konačno, Java je i projektovana da bi povećala produktivnost programera).

Model događaja grafičke biblioteke Swing

U modelu događaja biblioteke Swing, komponenta može da *izazove događaj* (engl. *fire an event*). Svaka vrsta događaja predstavljena je zasebnom klasom. O događaju se obaveštava jedan prijemnik (engl. *listener*) ili više njih a oni potom reaguju na njega. To znači da izvor

događaja i mesto na kom se on obrađuje mogu da se razlikuju. Komponente grafičke biblioteke Swing obično se koriste bez izmene, ali je neophodno pisati kôd koji se poziva kada komponente izazovu događaj. To je odličan primer razdvajanja interfejsa i realizacije.

Svaki prijemnik događaja je objekat klase koja realizuje određen tip prijemničkog interfejsa. Zbog toga treba samo da napravite objekat prijemnika i da ga prijavite komponenti koja pokreće događaj. Prijavljivanje se obavlja metodom **addXXXListener( )** komponente čiji se događaj obrađuje, pri čemu „**XXX**" predstavlja tip događaja koji se osluškuje. Čitanjem imena metoda tipa „addListener" lako ćete zaključiti koje se vrste događaja mogu obrađivati. Ako pokušate da primate pogrešne događaje, dobićete grešku tokom prevođenja. U nastavku poglavlja videćete da zrna Jave takođe koriste metode tipa „addListener" za određivanje događaja koje zrno može da izazove.

Prema tome, sva logika za obradu događaja smešta se u klasu prijemnika. Kada pravite takvu klasu, jedino ograničenje je da ona mora realizovati odgovarajući interfejs. Prijemnička klasa može biti globalna, ali u takvim slučajevima obično su pogodnije unutrašnje klase. Ne samo da one logički grupišu prijemničke klase unutar korisničkog okruženja ili klasa poslovne logike kojima služe, nego i čuvaju referencu na roditeljski objekat, što predstavlja lep način za prevazilaženje granica oblasti važenja klase i podsistema.

U svim dosadašnjim primerima u ovom poglavlju koristio sam model događaja grafičke biblioteke Swing, a sada ćemo detaljnije razmotriti taj model.

Tipovi događaja i prijemnika

Sve Swing komponente sadrže metode tipa **addXXXListener( )** i **removeXXXListener( )**, pa se odgovarajući tipovi prijemnika mogu dodavati u sve komponente i uklanjati iz njih. Primetićete da oznaka **XXX** u različitim slučajevima određuje i argument metode, na primer: **addMojPrijemnik (MojPrijemnik m)**. U sledećoj tabeli nabrojani su osnovni događaji, prijemnici i metode, kao i osnovne komponente koje metodama **addXXXListener( )** i **removeXXXListener( )** podržavaju pojedine događaje. Uvek treba imati u vidu da se model događaja može proširivati, pa ćete možda naići na druge tipove prijemnika koji nisu navedeni u ovoj tabeli.

Događaj, prijemnički interfejs i metode add- i remove-	Komponenta koja podržava taj događaj
ActionEvent **ActionListener** **addActionListener( )** removeActionListener( )	**JButton**, **JList**, **JTextField**, **JMenuItem** i klase izvedene iz ovih, kao što su **JCheckBoxMenuItem**, **JMenu** i **JRadioButtonMenuItem**
AdjustmentEvent **AdjustmentListener** **addAdjustmentListener( )** **removeAdjustmentListener( )**	**JScrollbar** i sve što napravite, a realizuje interfejs **Adjustable**
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	Klasa ***Component** i klase izvedene iz nje, među kojima su **JButton**, **JCheckBox**, **JComboBox**, **Container**, **JPanel**, **JApplet**, **JScrollPane**, **Window**, **JDialog**, **JFileDialog**, **JFrame**, **JLabel**, **JList**, **JScrollbar**, **JTextArea** i **JTextField**

Događaj, prijemnički interfejs i metode add- i remove-	**Komponenta koja podržava taj događaj (nastavak)**
ContainerEvent	Klasa **Container** i klase izvedene iz nje.
addContainerListener( ) **removeContainerListener( )**	**JScrollPane**, **Window**, **JDialog**, **JFileDialog**, i **JFrame**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	Klasa **Component** i klase izvedene iz nje*.
KeyEvent	Klasa **Component** i klase izvedene iz nje*.
KeyListener **addKeyListener( )** **removeKeyListener**	Klasa **Component** i klase izvedene iz nje*.
MouseEvent (za pomeranje i pritiskanje tastera miša) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	Klasa **Component** i klase izvedene iz nje*.
MouseEvent (za pomeranje i pritiskanje tastera miša) **MouseMotionListener** **addMouseMotionListener** **removeMouseMotionListener( )**	Klasa **Component** i klase izvedene iz nje*.
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	Klasa **Window** i klase izvedene iz nje, među kojima su i **JDialog**, **JFileDialog** i **JFrame**
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox**, **JCheckBoxMenuItem**, **JComboBox**, **JList** i sve što realizuje interfejs **ItemSelectable**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Sve što je izvedeno iz klase **JTextComponent**, uključujući i **JTextArea** i **JTextField**

Vidi se da svaki tip komponente podržava samo određene tipove događaja. Ispostavlja se da je prilično teško tražiti sve događaje koje podržava svaka komponenta. Lakši pristup je da se prilagodi program **PrikaziMetode.java** iz poglavlja *Podaci o tipu* tako da prikazuje sve prijemnike događaja koje podržava bilo koja uneta Swing komponenta.

U poglavlju *Podaci o tipu* objašnjena je *refleksija* a potom je korišćena i za listanje metoda određene klase (ceo spisak metoda ili podskup metoda u čijem se imenu nalazi zadata rezervisana reč). Ovaj postupak je moćan zato što automatski prikazuje *sve* metode

* Događaj **MouseMotionEvent** ne postoji mada izgleda kao da bi morao postojati. Pomeranje miša uz držanje pritisnutog tastera objedinjeno je u događaj **MouseEvent**, pa drugo pojavljivanje klase **MouseEvent** u tabeli nije greška.

klase, a da ne morate da prolazite kroz hijerarhiju nasleđivanja i ispitujete osnovne klase na svakom nivou. Zato ta vredna alatka štedi vreme pri programiranju: pošto su imena najvećeg broja metoda u Javi opisna i dugačka, tražite metode čija imena sadrže određenu reč. Kada pronađete metodu koja bi mogla biti ono što tražite, pročitajte njen opis u dokumentaciji na Webu.

Međutim, do poglavlja *Podaci o tipu* nije se pominjala grafička biblioteka Swing, pa je alatka u tom poglavlju razvijena kao aplikacija koja se izvršava s komandne linije. Evo korisnije verzije za grafičko okruženje koja traži metode tipa „addListener" u Swing komponentama:

```
//: gui/PrikaziMetodeAddListener.java
// Prikazuje metode "addXXXListener" bilo koje Swing klase.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingKonzola.*;

public class PrikaziMetodeAddListener extends JFrame {
  private JTextField ime = new JTextField(25);
  private JTextArea rezultati = new JTextArea(40, 65);
  private static Uzorak addListener =
      Uzorak.compile("(add\\w+?Listener\\(.*?\\))");
  private static Uzorak kvalifikator =
      Uzorak.compile("\\w+\\.");
  class ImeL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String im = ime.getText().trim();
      if(im.length() == 0) {
        rezultati.setText("Ne postoji");
        return;
      }
Class<?> vrsta;
try {
   vrsta = Class.forName("javax.swing." + im)
} catch(ClassNotFoundException ex) {
        rezultati.setText("Ne postoji");
        return;
}
Method[] metode = vrsta.getMethods();
rezultati.setText("");
for (Method m : metode)  {
    Matcher matcher =
        addListener.matcher(m.toString());
    if(matcher.find())
       rezultati.append(kvalifikator.matcher(
           matcher.group(1)).replaceAll("") + "\n");
```

```
        }
      }
    }
    public PrikaziMetodeAddListener() {
      ImeL imeListener = new ImeL();
      ime.addActionListener(imeListener);
      JPanel gornji = new JPanel();
      gornji.add(new JLabel("Ime Swing klase(pritisnite Enter):"));
      gornji.add(ime);
      add(BorderLayout.NORTH, gornji);
      add(new JScrollPane(rezultati));
      // Početni podaci i test:
      ime.setText("JTextArea");
      imeListener.actionPerformed(
          new ActionEvent("", 0 ,""));
    }
    public static void main(String[] args) {
      run(new PrikaziMetodeAddListener(), 500,400);
    }
  } ///:~
```

Glavni prozor ovog programa sadrži polje **ime** tipa **JTextField**; u njega treba da unesete ime Swing klase koju tražite. Rezultati se vade pomoću regularnog izraza i prikazuju u polju tipa **JTextArea**.

Primetićete da nema dugmadi niti drugih komponenata kojima bi se naznačilo kako želite da počnete pretraživanje. Razlog je to što događaje polja tipa **JTextField** nadgleda prijemnik tipa **ActionListener**. Kad god nešto izmenite i pritisnete taster Enter, spisak se trenutno ažurira. Ako polje nije prazno, njegov sadržaj se koristi za pokušaj pronalaženja klase metodom **Class.forName( )**. Ukoliko je ime nepravilno, metoda **Class.forName( )** neće naći klasu, pa generiše izuzetak koji se hvata, a u objektu klase **JTextArea** prikazuje se znakovni niz „Ne postoji". Međutim, ako otkucate ispravno ime (važan je i raspored velikih i malih slova), metoda **Class.forName( )** nalazi klasu i **getMethods( )** vraća niz objekata klase **Method**.

Ovde se koriste dva regularna izraza. Prvi, **addListener**, traži reč add iza koje slede slova pa reč Listener i spisak argumenata u zagradi. Ceo regularan izraz je u običnim zagradama (koje nisu pretvorene u izlazne sekvence), što znači da će biti pristupačan kao „grupa" regularnog izraza kada bude pronađen. Unutar metode **ImeL.ActionPerformed( )**, svaki objekat **Method** prosleđuje se metodi **Uzorak.matcher( )**, čime se pravi objekat tipa **Matcher**. Kada se za taj objekat tipa **Matcher** pozove metoda **find( )**, ona vraća **true** ako pronađe podudaran niz znakova i u tom slučaju pozivom metode **group(1)** možete odabrati prvu podudarnu grupu unutar zagrada. Taj znakovni niz još uvek sadrži kvalifikatore, pa se za njihovo odbacivanje upotrebljava objekat **kvalifikator Uzorak**, kao u **PrikažiMetode.java**.

Na kraju konstruktora, u **ime** se smešta početna vrednost i izvršava se reakcija na događaj (**ActionEvent**) koja ispitivanje snabdeva početnim podacima.

Ovaj program je zgodan za ispitivanje mogućnosti neke Swing komponente. Kada znate koje događaje podržava određena Swing komponenta, ne morate da tražite ništa posebno da bi ona reagovala na taj događaj; samo učinite sledeće:

1. Iz imena klase događaja uklonite reč **Event**. Dodajte reč **Listener** onome što preostane. To je interfejs prijemnika koji morate da realizujete u unutrašnjoj klasi.
2. Realizujete pomenuti interfejs i napišete metode za obradu događaja. Na primer, možda želite da pratite pokrete miša; u tom slučaju treba da napišete kôd za metodu **mouseMoved( )** interfejsa **MouseMotionListener**. (Naravno, morate da realizujete i druge metode tog interfejsa, ali za to često postoji prečica, što ćete uskoro saznati.)
3. Napravite objekat prijemničke klase iz drugog koraka. Prijavite ga svojoj komponenti metodom čije ćete ime dobiti ako ispred imena prijemnika dodate **add**, npr. **addMouseMotionListener**.

Evo nekih prijemničkih interfejsa:

Prijemnički interfejs i adapter	Metode u interfejsu
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(Adjustment Event)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Ovo nije sveobuhvatan spisak, delimično i zbog toga što vam model događaja dozvoljava da pravite sopstvene tipove događaja i prijemnike za njih. Zato ćete često viđati biblioteke u kojima postoje novi događaji, a znanje koje ste stekli čitajući ovo poglavlje omogućiće vam da smislite kako ćete koristiti te događaje.

Korišćenje prijemničkih adaptera radi jednostavnosti

Iz gornje tabele vidi se da neki interfejsi prijemnika sadrže samo jednu metodu. Oni se veoma lako realizuju, pošto se to radi samo kada želite da napišete metodu koju sadrže. Međutim, interfejsi prijemnika ponekad sadrže više metoda koje nije tako prijatno koristiti. Na primer, kada poželite da uhvatite pritisak na taster miša (koji nije već uhvaćen, recimo, nekim dugmetom), morate da napišete metodu **mouseClicked( )**. Ali pošto je **MouseListener** interfejs, morate da realizujete i sve ostale metode, čak i ako vam one u tom slučaju ne koriste. To ume da iznervira.

Da bi se rešio taj problem, neki (ali ne i svi) prijemnički interfejsi koji sadrže više od jedne metode imaju *adaptere*, čija imena vidite u prethodnoj tabeli. Svaki adapter obezbeđuje podrazumevane prazne metode za sve metode interfejsa. U tom slučaju treba samo da izvedete klasu iz adaptera i da redefinišete samo one metode koje moraju da se promene. Na primer, tipičan interfejs **MouseListener** koji ćete koristiti izgleda ovako:

```
class MojPrijemnikMisa extends MouseAdapter {
  public void mouseClicked (MouseEvent e) {
      // Odgovor na pritisak tastera miša…
  }
}
```

Smisao adaptera i jeste da olakša pravljenje prijemničkih klasa.

Ipak, adapteri imaju i jedan nedostatak. Pretpostavimo da ste nasledili klasu **MouseAdapter** na ovaj način:

```
class MojPrijemnikMisa extends MouseAdapter {
  public void MouseClicked (MouseEvent e) {
      // Odgovor na pritiskanje tastera miša...
  }
}
```

Ovo ne radi, a vi ćete izludeti pokušavajući da otkrijete razlog. Sve se lepo prevodi i izvršava, osim što nakon pritiska na taster miša vaša metoda neće biti pozivana. Vidite li šta je problem? Ime metode: navedeno je **MouseClicked( )** umesto **mouseClicked( )**. Malo zanemarivanje veličine slova prouzrokuje dodavanje potpuno nove metode. Međutim, to nije metoda koja se poziva kada se prozor zatvara, pa nećete dobiti željene rezultate. I pored nepogodnosti, interfejs će garantovati da su metode pravilno realizovane.

Izvesno ćete redefinisati određenu metodu ukoliko u gornjem kodu upotrebite ugrađenu anotaciju **@Override**.

Vežba 9: (5) Na osnovu programa **PrikaziMetodeAddListener.java** napišite program sa svim funkcijama programa **podaciotipu.PrikaziMetode.java**.

Praćenje više događaja

Da bismo dokazali da se ovi događaji zaista dešavaju, napisaćemo program koji prati dodatno ponašanje dugmeta **JButton** (a ne samo to da li je pritisnuto ili nije). Ovaj primer pokazuje i kako da sopstveno dugme izvedete iz klase **JButton**.[7]

Klasa **MojeDugme** je unutrašnja klasa u klasi **PratiDogadjaj**, pa **MojeDugme** ima pristup roditeljskom prozoru i može da radi s njegovim poljima za tekst, što je neophodno da bi statusne informacije mogle da se upisuju u polja roditelja. Naravno, ovo rešenje ima ograničenja, pošto objekat te klase može da se koristi samo zajedno sa objektom klase **PratiDogadjaj**. Ovakva vrsta koda ponekad se naziva visoko spregnuta (engl. *highly coupled*):

```
//: gui/PratiDogadjaj.java
// Prikazuje događaje koji se dešavaju.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingKonzola.*;

public class PratiDogadjaj extends JFrame {
  private HashMap<String, JTextField> h =
     new HashMap<String, JTextField>();
  private String[] dogadjaj = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  private MojeDugme
    d1 = new MojeDugme(Color.BLUE, "test1"),
    d2 = new MojeDugme(Color.RED, "test2");
  class MojeDugme extends JButton {
    void prijavi(String polje, String poruka) {
      (h.get(polje)).setText(poruka);
    }
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
        prijavi("focusGained", e.paramString());
      }
      public void focusLost(FocusEvent e) {
        prijavi("focusLost", e.paramString());
      }
    };
    KeyListener kl = new KeyListener() {
      public void keyPressed(KeyEvent e) {
        prijavi("keyPressed", e.paramString());
      }
```

[7] U Javi 1.0/1.1 *nije* bilo moguće izvesti korisne klase iz dugmeta. To je bio samo jedan od mnogih velikih propusta u projektovanju.

```
      public void keyReleased(KeyEvent e) {
        prijavi("keyReleased", e.paramString());
      }
      public void keyTyped(KeyEvent e) {
        prijavi("keyTyped", e.paramString());
      }
    };
    MouseListener ml = new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        prijavi("mouseClicked", e.paramString());
      }
      public void mouseEntered(MouseEvent e) {
        prijavi("mouseEntered", e.paramString());
      }
      public void mouseExited(MouseEvent e) {
        prijavi("mouseExited", e.paramString());
      }
      public void mousePressed(MouseEvent e) {
        prijavi("mousePressed", e.paramString());
      }
      public void mouseReleased(MouseEvent e) {
        prijavi("mouseReleased", e.paramString());
      }
    };
    MouseMotionListener mml = new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        prijavi("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        prijavi("mouseMoved", e.paramString());
      }
    };
    public MojeDugme(Color boja, String natpis) {
      super(natpis);
      setBackground(boja);
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  }
  public PratiDogadjaj() {
    setLayout(new GridLayout(dogadjaj.length + 1, 2));
    for(String dgd : dogadjaj) {
      JTextField t = new JTextField();
      t.setEditable(false);
      add(new JLabel(dgd, JLabel.RIGHT));
      add(t);
```

```
        h.put(dgd, t);
      }
      add(d1);
      add(d2);
    }
    public static void main(String[] args) {
      run(new PratiDogadjaj(), 700, 500);
    }
  } ///:~
```

U konstruktoru klase **MojeDugme** boja dugmeta se zadaje metodom **SetBackground( )**. Prijemnici su instalirani preko poziva metoda.

Klasa **PratiDogadjaj** sadrži mapu tipa **HashMap**. Ona povezuje znakovne nizove koji predstavljaju tip događaja i polja tipa **JTextField** u kojima se nalaze informacije o događaju. Naravno, ova polja su mogla da budu i statička, ali složićete se da se ovako mnogo lakše koriste i menjaju. Poseban je slučaj ako dodajete ili uklanjate nov tip događaja u objektu klase **PratiDogadjaj**: treba samo da dodate ili uklonite znakovni niz iz niza **dogadjaj**; sve ostalo se odvija automatski.

Metodi **prijavi( )** prosleđuje se ime događaja i parametarski znakovni niz iz događaja. Ova metoda koristi **HashMap** mapu **h** iz spoljašnje klase da bi pronašla tekstualno polje povezano s imenom događaja, a zatim smešta parametarski znakovni niz u to polje.

Ovaj primer je koristan pošto možete da vidite šta se u programu stvarno dešava sa događajima.

Vežba 10: (6) Napravite aplikaciju uz pomoć klase **SwingKonzola** s dugmetom tipa **JButton** i poljem tipa **JTextField**. Napišite i povežite odgovarajući prijemnik koji u polju **JTextField** prikazuje sadržaj dugmeta kada je ono u fokusu.

Vežba 11: (4) Izvedite nov tip dugmeta iz klase **JButton**. Kad god se to dugme pritisne, trebalo bi da promeni boju u neku nasumično izabranu. Primer kako da generišete nasumičnu boju potražite u programu **ObojeneKutije.java** u nastavku poglavlja.

Vežba 12: (4) Obradite nov tip događaja u programu **PratiDogadjaj.java** dodavanjem novog koda za obradu događaja. Moraćete sami da smislite tip događaja koji želite da obrađujete.

Primeri Swing komponenata

Sad razumete raspoređivače i model događaja i spremni ste da vidite kako se mogu koristiti komponente grafičke biblioteke Swing. Ovaj odeljak ukratko prikazuje Swing komponente i njihove mogućnosti koje ćete verovatno najčešće koristiti. Primeri nisu previše obimni, pa kôd možete da iskoristite u sopstvenim programima.

Imajte u vidu sledeće:

1. Lako ćete videti kako svaki primer izgleda u praksi ako pregledate HTML strane koje se nalaze u paketu sa izvornim kodom za ovu knjigu (možete ga preuzeti s lokacije *www.MindView.net*).
2. U HTML dokumentaciji na adresi *java.sun.com* opisane su sve klase i metode biblioteke Swing (ovde su prikazane samo neke).

3. Zbog pravila za imenovanje koja se primenjuju na Swing događaje, prilično se lako pogađa kako se piše i registruje prijemnik za odredeni tip događaja. Upotrebite program **PrikaziMetodeAddListener.java** iz ovog poglavlja da biste sebi olakšali traženje određene komponente.
4. Kada stvari postanu komplikovanije – vreme je da koristite vizuelne alatke.

Dugmad

Biblioteka Swing sadrži veliki broj dugmadi različitih tipova. Sva dugmad, polja za potvrdu, radio-dugmad, čak i stavke menija, izvedeni su iz klase **AbstractButton** (kojoj bi, pošto obuhvata i stavke menija, bolje odgovaralo ime „AbstractSelector" ili nešto opštije). Uskoro ćete saznati kako se koriste stavke menija, a sledeći primer prikazuje različite dostupne tipove dugmadi:

```
//: gui/Dugmad.java
// Razna dugmad iz biblioteke Swing.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class Dugmad extends JFrame {
  private JButton jb = new JButton("JButton");
  private BasicArrowButton
    up = new BasicArrowButton(BasicArrowButton.NORTH),
    down = new BasicArrowButton(BasicArrowButton.SOUTH),
    right = new BasicArrowButton(BasicArrowButton.EAST),
    left = new BasicArrowButton(BasicArrowButton.WEST);
  public Dugmad() {
    setLayout(new FlowLayout());
    add(jb);
    add(new JToggleButton("JToggleButton"));
    add(new JCheckBox("JCheckBox"));
    add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Pravci"));
    jp.add(up);
    jp.add(down);
    jp.add(left);
    jp.add(right);
    add(jp);

  public static void main(String[] args) {
    run(new Dugmad(), 350, 200);
  }
} ///:~
```

Program prvo prikazuje klasu **BasicArrowButton** iz biblioteke **javax.swing. plaf.basic**, a zatim razne tipove dugmadi. Kada izvršite program, videćete da dugme za uključivanje i isključivanje zadržava svoj poslednji položaj (uključeno ili isključeno). Ali polja za potvrdu i radio-dugmad ponašaju se jednako – mogu da budu potvrđeni ili nepotvrđeni (izvedeni su iz klase **JToggleButton**).

Grupe dugmadi

Ako želite da radio-dugmad isključuju jedno drugo, morate ih smestiti u „grupu dugmadi". Međutim, kao što prikazuje sledeći primer, svako dugme tipa **AbstractButton** može se dodati grupi tipa **ButtonGroup**.

Da bi se izbeglo često ponavljanje koda, u ovom primeru koristi se refleksija za pravljenje grupa dugmadi različitih tipova. To se vidi u metodi **napraviPano( )** koja pravi grupu dugmadi i objekat klase **JPanel**. Drugi argument metode **napraviPano( )** jeste niz objekata tipa **String**. Za svaki element tog niza na pano se dodaje dugme, a tip dugmadi određuje prvi argument metode:

```
//: gui/GrupeDugmadi.java
// Primenjuje refleksiju za pravljenje grupa
// različitih tipova dugmadi AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingKonzola.*;

public class GrupeDugmadi extends JFrame {
  static String[] ids = {
    "Gaja", "Raja", "Vlaja", "Miki", "Silja", "Pluton",
  };
  static JPanel napraviPano(
    Class<? extends AbstractButton> vrsta, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String naslov = vrsta.getName();
    naslov = naslov.substring(naslov.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(naslov));
    for(String id : ids) {
      AbstractButton ab = new JButton("neuspešno");
      try {
        // Pozivanje dinamičkog konstruktora
        // koji prihvata argument tipa String:
        Constructor ctor = vrsta.getConstructor(String.class);
        // Pravi nov objekat:
        ab = (AbstractButton)ctor.newInstance(id);
      } catch(Exception ex) {
        System.err.println("Ne mogu da napravim" + vrsta);
      }
```

```
      bg.add(ab);
      jp.add(ab);
    }
    return jp;
  }
  public GrupeDugmadi() {
    setLayout(new FlowLayout());
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
  }
  public static void main(String[] args) {
    run(new GrupeDugmadi(), 500, 350);
  }
} ///:~
```

Naslov se pravi na osnovu imena klase, uz izbacivanje svih informacija o putanji. Na početku je dugme **ab** tipa **AbstractButton** u stvari **JButton** sa oznakom „neuspešno", pa ćete lako uočiti eventualne probleme, čak i ako zanemarite poruke o izuzecima. Metoda **getConstructor( )** vraća objekat tipa **Constructor** koji opisuje konstruktor čija lista argumenata odgovara nizu tipova u spisku klasa prosleđenom toj metodi. Nakon toga treba samo da pozovete metodu **newInstance( )** kojoj ćete proslediti niz objekata koji sadrži stvarne argumente; u ovom slučaju, samo objekte tipa **String** iz niza **ids**.

Da biste postigli „isključivo" ponašanje dugmadi, napravite grupu dugmadi i dodajte joj svu dugmad koja tako treba da se ponašaju. Kada izvršite program, videćete da svi tipovi dugmadi osim **JButton** realizuju „isključivo" ponašanje.

Ikonice

Klasu **Icon** možete da koristite unutar natpisa tipa **JLabel** ili bilo koje klase koja se izvodi iz klase **AbstractButton** (uključujući **JButton**, **JCheckBox**, **JRadioButton** i razne vrste **JMenuItem**). Korišćenje ikonica s klasom **JLabel** sasvim je jednostavno (kasnije ćete videti primer). U sledećem primeru istražuju se drugi načini korišćenja ikonica s dugmadima i objektima koji su iz njih izvedeni.

Možete da koristite proizvoljne datoteke u formatu **GIF**, a one koje se upotrebljavaju u ovom primeru naći ćete u paketu sa izvornim kodom, dostupnom na adresi *www.MindView.net*. Da biste otvorili datoteku i prikazali sliku, napravite objekat klase **ImageIcon** i prosledite mu ime datoteke. Nakon toga, moći ćete u programu da koristite dobijeni objekat tipa **Icon**.

```
//: gui/Likovi.java
// Ponašanje ikonica na dugmadima tipa JButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;
```

```
public class Likovi extends JFrame {
  private static Icon[] likovi;
  private JButton jb, jb2 = new JButton("Onemoguci");
  private boolean besan = false;
  public Likovi()  {
    likovi = new Icon[] {
       new ImageIcon(getClass().getResource("Face0.gif")),
       new ImageIcon(getClass().getResource("Face1.gif")),
       new ImageIcon(getClass().getResource("Face2.gif")),
       new ImageIcon(getClass().getResource("Face3.gif")),
       new ImageIcon(getClass().getResource("Face4.gif")),
    };
    jb = new JButton("JButton", likovi[3]),
    setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(besan) {
          jb.setIcon(likovi[0]);
          besan = false;
        } else {
          jb.setIcon(likovi[0]);
          besan = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(likovi[1]);
    jb.setPressedIcon(likovi[2]);
    jb.setDisabledIcon(likovi[4]);
    jb.setToolTipText("Jao!");
    add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Dozvoli");
        } else {
          jb.setEnabled(true);
          jb2.setText("Onemoguci");
        }
      }
    });
    add(jb2);
  }
  public static void main(String[] args) {
    run(new Likovi(), 250, 125);
  }
} ///:~
```

Objekat klase **Icon** može se koristiti u mnogim konstruktorima grafičkih elemenata, ali za dodavanje ili uklanjanje ikonice možete da koristite i metodu **setIcon( )**. Ovaj primer pokazuje i kako dugme tipa **JButton** (ili bilo koji objekat klase **AbstractButton**) može da prikazuje različite vrste ikonica u zavisnosti od događaja: kada se pritisne, isključi ili kada se pokazivač miša zadrži iznad njega bez pritiskanja. Videćete da se tako dobija zanimljivo, animirano dugme.

Priručni saveti

U prethodnom primeru dugmetu je dodat i „priručni savet" (engl. *tool tip*). Gotovo sve klase grafičkih elemenata izvedene su iz klase **JComponent** koja sadrži metodu **setToolTipText(String)**. Zbog toga za doslovno sve što stavite na obrazac, treba samo da napišete (za objekat **jc** bilo koje klase izvedene iz klase **JComponent**):

```
jc.setToolTipText("Moj savet");
```

i kada se pokazivač miša zadrži iznad objekta klase **JComponent** zadato vreme, pored pokazivača će se pojaviti malo polje s vašim tekstom.

Jednoredna polja za tekst

Ovaj primer prikazuje dodatno ponašanje polja za tekst tipa **JTextField**:

```
//: gui/PoljaZaTekst.java
// Polja za tekst i Javini događaji.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class PoljaZaTekst extends JFrame {
  JButton
    d1 = new JButton("Uzmi tekst"),
    d2 = new JButton("Zadaj tekst");
  private JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  private String s = "";
  private DokumentSVelikimSlovima dvs = new DokumentSVelikimSlovima();
  public PoljaZaTek...() {
    t1.setDocument(dvs);
    dvs.addDocumentListener(new T1());
    d1.addActionListener(new B1());
    d2.addActionListener(new B2());
```

```
      t1.addActionListener(new T1A());
      setLayout(new FlowLayout());
      add(d1);
      add(d2);
      add(t1);
      add(t2);
      add(t3);
    }
    class T1 implements DocumentListener {
      public void changedUpdate(DocumentEvent e){}
      public void insertUpdate(DocumentEvent e){
        t2.setText(t1.getText());
        t3.setText("Tekst: "+ t1.getText());
      }
      public void removeUpdate(DocumentEvent e){
        t2.setText(t1.getText());
      }
    }
    class T1A implements ActionListener {
      private int broj = 0;
      public void actionPerformed(ActionEvent e) {
        t3.setText("t1 dogadjaj " + broj++);
      }
    }
    class B1 implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(t1.getSelectedText() == null)
          s = t1.getText();
        else
          s = t1.getSelectedText();
        t1.setEditable(true);
      }
    }
    class B2 implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        dvs.zadajPretvaranje(false);
        t1.setText("Ubaceno dugmetom 2: " + s);
        dvs.zadajPretvaranje(true);
        t1.setEditable(false);
      }
    }
    public static void main(String[] args) {
      run(new PoljaZaTekst(), 375, 200);
    }
}

class DokumentSVelikimSlovima extends PlainDocument {
  boolean pretvaranje = true;
```

```
  public void zadajPretvaranje(boolean indikator) {
    pretvaranje = indikator;
  }
  public void insertString(int offset, String str, AttributeSet attSet)
    throws BadLocationException {
      if(pretvaranje) string = string.toUpperCase();
      super.insertString(offset, string, attributeSet);
  }
} ///:~
```

Polje **t3** tipa **JTextField** služi za izveštavanje o događajima polja **t1** tipa **JTextField**. Videćete da se prijemnik događaja za polje **JTextField** pokreće samo kada pritisnete taster Enter.

S poljem **t1** povezano je nekoliko prijemnika. **T1** je tipa **DocumentListener** i odgovara na sve promene „dokumenta" (u ovom slučaju, sadržaja polja). On automatski kopira sav tekst iz **t1** u **t2**. Pored toga, dokument polja **t1** je tipa **DokumentSVelikimSlovima** koji je izveden iz klase **PlainDocument**, a pretvara sva slova u velika. Automatski se otkrivaju znakovi za brisanje, briše sadržaj i pomera pokazivač, pa sve radi kao što se i očekuje.

Vežba 13: (3) Promenite program **PoljaZaTekst.java** tako da znakovi u **t2** zadržavaju prvobitni raspored malih i velikih slova kako su bila unesena, umesto da se automatski pretvaraju u velika slova.

Ivice

Klasa **JComponent** sadrži metodu **setBorder( )** koja omogućuje postavljanje zanimljivih ivica oko svih vidljivih komponenata. Sledeći primer prikazuje razne ivice – metodom **prikaziIvicu( )** pravi se objekat klase **JPanel** i na njega postavlja ivica. Primenjuje se i tehnika prepoznavanja tipa tokom izvršavanja da bi se ustanovilo ime ivice koja se koristi (uz izbacivanje svih informacija o putanji), a zatim se to ime stavlja u objekat klase **JLabel**, u sredinu panoa.

```
//: gui/Ivice.java
// Razne ivice iz biblioteke Swing.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class Ivice extends JFrame {
  static JPanel prikaziIvicu(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
```

```
    jp.setBorder(b);
    return jp;
  }
  public Ivice() {
    setLayout(new GridLayout(2,4));
    add(prikaziIvicu(new TitledBorder("Naslov")));
    add(prikaziIvicu(new EtchedBorder()));
    add(prikaziIvicu(new LineBorder(Color.BLUE)));
    add(prikaziIvicu(
      new MatteBorder(5,5,30,30,Color.GREEN)));
    add(prikaziIvicu(
      new BevelBorder(BevelBorder.RAISED)));
    add(prikaziIvicu(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    add(prikaziIvicu(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.RED))));
  }
  public static void main(String[] args) {
    run(new Ivice(), 500, 300);
  }
} ///:~
```

Možete da napravite i sopstvene ivice koje ćete staviti oko dugmadi, natpisa ili bilo čega što je izvedeno iz klase **JComponent**.

Mali program za uređivanje teksta

Klasa **JTextPane** obezbeđuje najveći deo podrške za obradu teksta bez mnogo truda. Sledeći primer ilustruje veoma jednostavnu upotrebu, bez mnogih funkcija ove klase:

```
//: gui/TekstualniPano.java
// Klasa JTextPane kao mali program za uređivanje teksta.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import net.mindview.util.*;
import static net.mindview.util.SwingKonzola.*;
public class TekstualniPano extends JFrame {
  private JButton b = new JButton("Dodaj tekst");
  private JTextPane tp = new JTextPane();
  private static Generator sg =
    new RandomGenerator.String(7);
  public TekstualniPano() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() + sg.next() + "\n");
      }
```

```
    });
    add(new JScrollPane(tp));
    add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    run(new TekstualniPano(), 475, 425);
  }
} ///:~
```

Dugme samo dodaje slučajno generisan tekst u polje. Polje tipa **JTextPane** treba da omogući obradu teksta na licu mesta, pa ćete primetiti da ne postoji metoda **append( )**. U ovom slučaju (priznajem, bedno u odnosu na mogućnosti klase **JTextPane**), tekst mora da se pročita, promeni i vrati nazad u pano pomoću metode **setText( )**.

Podrazumevani raspoređivač za **JFrame** jeste **BorderLayout** i on mu dodaje elemente. Pošto je **JTextPane** dodat u pano **JScrollPane** bez zadavanja oblasti, on se postavlja u centar panoa i razvlači prema ivicama. Dugme je dodato nadole (u oblast **SOUTH**), pa će se uklopiti u tu oblast. U ovom slučaju, dugme će se ugnezditi na dnu ekrana.

Obratite pažnju na ugrađene mogućnosti klase **JTextPane** kao što je automatsko prelamanje redova. Postoje i brojne druge funkcije o kojima možete mnogo saznati iz dokumentacije razvojnog programskog paketa JDK.

Vežba 14: (2) Promenite program **TekstualniPano.java** tako da koristi polje tipa **JTextArea** umesto **JTextPane**.

Polja za potvrdu

Polje za potvrdu omogućuje pravljenje opcije koja može da bude potvrđena ili nepotvrđena; sastoji se od malog polja i oznake. Polje obično sadrži malo x (ili nešto drugo što pokazuje da je opcija potvrđena) ili je prazno, u zavisnosti od toga da li je stavka koju označava potvrđena ili nije.

Objekat klase **JCheckBox** obično se pravi pomoću konstruktora čiji je argument natpis. Stanje polja možete da čitate i zadajete, ali možete i da čitate i menjate njegov natpis nakon pravljenja.

Kad god se polje tipa **JCheckBox** potvrdi ili se potvrda ukloni, nastaje događaj koji možete da uhvatite na isti način kao događaj dugmeta, pomoću objekta klase **ActionListener**. U sledećem primeru koristi se tekstualno polje za brojanje potvrđenih polja:

```
//: gui/PoljaZaPotvrdu.java
// Korišćenje klase JCheckBoxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class PoljaZaPotvrdu extends JFrame {
  private JTextArea t = new JTextArea(6, 15);
  private JCheckBox
```

```
    cb1 = new JCheckBox("Polje za potvrdu 1"),
    cb2 = new JCheckBox("Polje za potvrdu 2"),
    cb3 = new JCheckBox("Polje za potvrdu 3");
  public PoljaZaPotvrdu() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        obradi("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        obradi("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        obradi("3", cb3);
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(cb1);
    add(cb2);
    add(cb3);
  }
  private void obradi(String b, JCheckBox cb) {
    if(cb.isSelected())
      t.append("Polje " + b + " je potvrđeno\n");
    else
      t.append("Polje " + b + " nije potvrđeno\n");
  }
  public static void main(String[] args) {
    run(new PoljaZaPotvrdu(), 200, 300);
  }
} ///:~
```

Metoda **obradi**( ) dodaje ime polja i njegovo tekuće stanje u sadržaj višerednog polja za tekst metodom **append**( ), pa će se prikazati zbirni spisak odabranih polja i njihovih stanja.

Vežba 15: (5) Dodajte polje za potvrdu u aplikaciju napravljenu u vežbi 5, uhvatite događaj i umetnite drugačiji tekst u polje.

Radio-dugmad

Ime radio-dugme u programiranju grafičkih korisničkih okruženja izvedeno je asocijacijom na radija iz starinskih automobila koji su imali mehaničku dugmad. Kada bi se jedno dugme pritisnulo, iskakalo bi dugme koje je ranije bilo pritisnuto. Na taj način mogla je da se odabere samo jedna mogućnost.

Da biste napravili povezanu grupu dugmadi tipa **JRadioButton**, treba samo da ih dodate u grupu tipa **ButtonGroup**. (U obrascu može da postoji proizvoljan broj grupa dugmadi.) Jednom dugmetu može da se (drugim argumentom konstruktora) zada početno stanje tako da bude potvrđeno. Ako više radio-dugmadi u grupi uključite konstruktorom, poslednje dugme će isključiti sva prethodna.

Evo jednostavnog primera korišćenja radio-dugmadi. Obratite pažnju na to da se događaji radio-dugmadi obrađuju isto kao i svi drugi događaji:

```
//: gui/RadioDugmad.java
// Korišćenje klase JRadioButtons.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class RadioDugmad extends JFrame {
  private JTextField t = new JTextField(15);
  private ButtonGroup g = new ButtonGroup();
  private JRadioButton
    rb1 = new JRadioButton("jedan", false),
    rb2 = new JRadioButton("dva", false),
    rb3 = new JRadioButton("tri", false);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio-dugme " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public RadioDugmad() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    setLayout(new FlowLayout());
    add(t);
    add(rb1);
    add(rb2);
    add(rb3);
  }
  public static void main(String[] args) {
    run(new RadioDugmad(), 200, 125);
  }
} ///:~
```

Za prikazivanje stanja koristi se tekstualno polje. Menjanje sadržaja tog polja je isključeno, zato što se u njemu prikazuju podaci a ne unose se u njega. Takvo tekstualno polje je alternativa za natpise tipa **JLabel**.

Kombinovane liste (padajuće liste)

Poput grupe radio-dugmadi, padajuća lista omogućava da se korisnik navede da izabere samo jedan element iz grupe mogućnosti. Međutim, lista je kompaktnije rešenje da se to postigne, a elemente liste je lakše menjati tako da se korisnik ne iznenadi. (Radio-dugmad možete menjati dinamički, ali to ume i da smeta.)

Klasa **JComboBox** podrazumevano se *ne* ponaša kao kombinovana lista u Windowsu koja omogućuje da korisnik odabere stavku iz liste *ili* da u nju sam unese neku vrednost, ako nije zadovoljan ponuđenim opcijama. Da biste omogućili korisnicima da unose vrednosti u objekat tipa **JComboBox**, potrebno je da pozovete metodu **setEditable**. Iz liste tipa **JComboBox** korisnik može odabrati jedan i samo jedan element. U sledećem primeru, polje tipa **JComboBox** na početku sadrži određen broj stavki, a zatim se nakon pritiskanja dugmeta u polje dodaju novi elementi.

```
//: gui/PadajuceListe.java
// Korišćenje padajućih lista.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class PadajuceListe extends JFrame {
  private String[] opis = { "Živahan", "Dosadan",
    "Tvrdoglav", "Sjajan", "Uspavan",
    "Zastrašujući", "Ispunjen", "Iskvaren" };
  private JTextField t = new JTextField(15);
  private JComboBox c = new JComboBox();
  private JButton b = new JButton("Dodaj elemente");
  private int broj = 0;
  public PadajuceListe() {
    for(int i = 0; i < 4; i++)
      c.addItem(opis[broj++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(broj < opis.length)
          c.addItem(opis[broj++]);
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("indeks: "+ c.getSelectedIndex() + "   " +
          ((JComboBox)e.getSource()).getSelectedItem());
      }
    });
    setLayout(new FlowLayout());
    add(t);
    add(c);
```

```
    add(b);
  }
  public static void main(String[] args) {
    run(new PadajuceListe(), 200, 175);
  }
} ///:~
```

Tekstualno polje prikazuje „odabrani indeks" (redni broj elementa koji je trenutno izabran u nizu), kao i tekst stavke koja je odabrana u kombinovanoj listi.

Grafičke liste

Polja tipa **JList** prilično se razlikuju od polja tipa **JComboBox**, i to ne samo po izgledu. Za razliku od polja tipa **JComboBox** koje pada kada ga aktivirate, **JList** uvek zauzima tačno određen broj redova na ekranu i ne menja se. Ako želite da vidite elemente liste, samo pozovite metodu **getSelectedValues**( ) koja vraća niz izabranih stavki.

Lista **JList** omogućuje višestruki izbor: ako uz pritisnut taster Ctrl mišem pritisnete više od jedne stavke, stavka koju ste prvu izabrali ostaće istaknuta, a nakon toga možete da izaberete još proizvoljan broj elemenata. Ukoliko izaberete neki element, a zatim uz pritisnut taster Shift mišem pritisnete još jedan element, biće izabrani svi elementi između njih. Da biste uklonili stavku iz izabrane grupe, pritisnite je mišem, istovremeno držeći pritisnut taster Ctrl.

```
//: gui/Lista.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class Lista extends JFrame {
  private String[] ukusi= {
    "Čokolada", "Jagoda", "Vanila", "Pepermint",
    "Moka", "Rum grozdice", "Praline krem", "Pita od blata"
  };
  private DefaultListModel elementiListe=new DefaultListModel();
  private JList lst = new JList(elementiListe);
  private JTextArea t = new JTextArea(ukusi.length,20);
  private JButton b = new JButton("Dodaj element");
  private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(broj < ukusi.length) {
        elementiListe.add(0, ukusi[broj++]);
      } else {
        // Isključi, pošto nema više ukusa
        // koji bi se dodali u listu
```

```
        b.setEnabled(false);
      }
    }
  };
  private ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
          if(e.getValueIsAdjusting()) return;
          t.setText("");
          for (Object stavke : lst.getSelectedValues())
            t.append(stavke[i] + "\n");
        }
    };
  private int broj = 0;
  public Lista() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Pravi ivice za komponente:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Dodavanje prve četiri stavke u listu
    for(int i = 0; i < 4; i++)
      elementiListe.addElement(ukusi[broj++]);
    // Dodavanje stavki u pano sadržaja radi prikaza
    add(t);
    add(lst);
    add(b);
    // Registrovanje slušalaca događaja
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
  }
  public static void main(String[] args) {
    run(new Lista(), 250, 375);
  }
} ///:~
```

Primećujete i da su listama dodate ivice.

Ako želite samo da smestite niz objekata tipa **String** u listu tipa **JList**, postoji mnogo jednostavnije rešenje: niz treba da prosledite konstruktoru klase **JList** i on će automatski napraviti listu. Model liste u gornjem primeru koristi se jedino zato da bi se lista menjala tokom izvršavanja programa.

Liste tipa **JList** ne obezbeđuju automatski direktnu podršku za pomeranje sadržaja po ekranu. Naravno, treba samo da stavite listu u okno tipa **JScrollPane** i njemu prepustite da se brine o detaljima.

Vežba 16: (5) Pojednostavite program **Lista.java** prosleđivanjem niza konstruktoru i uklanjanjem dinamičkog dodavanja elemenata u listu.

Okno s jezičcima

Klasa **JTabbedPane** omogućuje pravljenje okna u kome se duž gornje ivice prikazuju jezičci. Pritiskom na određeni jezičak prikazaćete njegov poddijalog.

```
//: gui/ OknoSaJeziccima 1.java
// Prikazuje okno s jezičcima.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class OknoSaJeziccima1 extends JFrame {
  private String[] ukusi = {
    "Čokolada", "Jagoda", "Vanila", "Pepermint",
    "Moka", "Rum grozđice", "Praline krem", "Pita od blata"
  };
  private JTabbedPane jezicci = new JTabbedPane();
  private JTextField txt = new JTextField(20);
  public OknoSaJeziccima1() {
    int i = 0;
    for(String ukus : ukusi)
      jezicci.addTab(ukusi [i],
        new JButton("Okno s jezičcima" + i++));
    jezicci.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        txt.setText("Izabrani jezičak kartice: " +
          jezicci.getSelectedIndex());
      }
    });
    add(BorderLayout.SOUTH, txt);
    add(jezicci);
  }
  public static void main(String[] args) {
    run(new OknoSaJeziccima1(), 400, 250);
  }
} ///:~
```

Kada izvršite program, videćete da **JTabbedPane** automatski prenosi jezičke u sledeći red ako ih ima previše da bi stali u isti red. To ćete videti kada u toku rada promenite veličinu prozora.

Okviri za poruke

Grafičko okruženje obično sadrži standardan skup okvira za poruke koji omogućuju brzo dostavljanje informacija korisniku ili uzimanje informacija od njega. U grafičkoj biblioteci Swing, polja za poruke realizuje klasa **JOptionPane**. Postoje brojne mogućnosti (od kojih su neke vrlo napredne), ali najčešće ćete koristiti dijalog poruke i dijalog za potvrdu koji se dobijaju pozivom statičkih metoda **JOptionPane.showMessageDialog( )** i **JOptionPane.showConfirmDialog( )**. Sledeći primer prikazuje neke od okvira za poruke koji postoje u klasi **JOptionPane**:

```
//: gui/OkviriZaPoruke.java
// Prikazuje klasu JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class OkviriZaPoruke extends JFrame {
  private JButton[] b = {
    new JButton("Upozorenje"),
    new JButton("Da/Ne"), new JButton("Boja"),
    new JButton("Ulaz"), new JButton("3 vrednosti")
  };
  private JTextField txt = new JTextField(15);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String id = ((JButton)e.getSource()).getText();
      if(id.equals("Upozorenje"))
        JOptionPane.showMessageDialog(null,
          "Imate grešku!", "Hej!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Da/Ne"))
        JOptionPane.showConfirmDialog(null,
          "ili ne", "izaberi da",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Boja")) {
        Object[] opcije = { "Crvena", "Zelena" };
        int izabrana = JOptionPane.showOptionDialog(
          null, "Izaberite boju!", "Upozorenje",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          opcije, opcije[0]);
          if(izabrana != JOptionPane.CLOSED_OPTION)
            txt.setText("Izabrana boja: " + opcije[izabrana]);
      } else if(id.equals("Ulaz")) {
        String vrednost = JOptionPane.showInputDialog(
            "Koliko prstiju vidite?");
        txt.setText(vrednost);
      } else if(id.equals("3 vrednosti")) {
        Object[] izbor = { "Prva", "Druga", "Treca" };
        Object vrednost = JOptionPane.showInputDialog(
          null, "Izaberite jednu", "Ulaz",
          JOptionPane.INFORMATION_MESSAGE,
          null, izbor, izbor[0]);
        if(vrednost != null)
          txt.setText(vrednost.toString());
      }
    }
```

```
  };
  public OkviriZaPoruke() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
      b[i].addActionListener(al);
      add(b[i]);
    }
    add(txt);
  }
  public static void main(String[] args) {
    run(new OkviriZaPoruke(), 200, 200);
  }
} ///:~
```

Da bih napisao samo jedan prijemnik događaja, koristio sam donekle rizičnu proveru natpisa na dugmadima. Pri takvoj proveri često se pogreši u pisanju, obično u rasporedu velikih i malih slova, a takva greška se teško uočava.

Obratite pažnju na to da su rezultati metoda **showOptionDialog( )** i **showInputDialog( )** objekti s vrednošću koju je uneo korisnik.

Vežba 17: (5) Napravite aplikaciju pomoću klase **SwingKonzola**. U HTML dokumentaciji na lokaciji *java.sun.com* pronađite klasu **JPasswordField** i dodajte je u program. Ako korisnik otkuca tačnu lozinku, prikažite poruku o uspehu pomoću objekta klase **JOptionPane**.

Vežba 18: (4) Promenite program **PoljaZaPotvrdu.java** tako da ima zaseban prijemnik tipa **ActionListener** za svako dugme (umesto da traži tekst na dugmetu).

Meniji

Svaka komponenta koja može da sadrži meni, uključujući **JApplet**, **JFrame**, **JDialog** i klase koje su iz njih izvedene, ima metodu **setJMenuBar( )**. Toj metodi se prosleđuje linija menija – objekat klase **JMenuBar** (može da postoji samo jedan objekat klase **JMenuBar** za određenu komponentu). Meniji tipa **JMenu** dodaju se na liniju menija, a objekti klase **JMenuItem** su stavke menija i oni se dodaju u menije. Sa svakim elementom tipa **JMenuItem** može da bude povezan poseban prijemnik događaja, i on se aktivira kada se izabere ta stavka menija.

Kada koristite Javu i biblioteku Swing, morate ručno da definišete sve menije u izvornom kodu. Evo vrlo jednostavnog primera menija:

```
//: gui/JednostavniMeniji.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;
public class JednostavniMeniji extends JFrame {
  private JTextField t = new JTextField(15);
  private ActionListener al = new ActionListener() {
```

```
    public void actionPerformed(ActionEvent e){
      t.setText(((JMenuItem)e.getSource()).getText());
    }
  };
  private JMenu[] meniji = {
    new JMenu("Zeka"), new JMenu("Pera"), new JMenu("Ana")
  };
  private JMenuItem[] stavke = {
    new JMenuItem("Mirko"), new JMenuItem("Joca"),
    new JMenuItem("Rade"),  new JMenuItem("Luka"),
    new JMenuItem("Slavko"), new JMenuItem("Vlada"),
    new JMenuItem("Mile"), new JMenuItem("Ivan"),
    new JMenuItem("Nada")
  };
  public JednostavniMeniji() {
    for(int i = 0; i < stavke.length; i++) {
      stavke[i].addActionListener(al);
      meniji[i % 3].add(stavke[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(JMenu jm : meniji)
      mb.add(jm);
    setJMenuBar(mb);
    setLayout(new FlowLayout());
    add(t);
  }
  public static void main(String[] args) {
   run(new JednostavniMeniji(), 200, 150);
  }
} ///:~
```

Zbog korišćenja operatora modulo u izrazu **i%3** ravnomerno se raspoređuju stavke na tri menija. Sa svakom stavkom mora da bude povezan prijemnik tipa **ActionListener**; ovde se isti prijemnik koristi svuda, ali je obično potreban poseban prijemnik tipa za svaku stavku.

Klasa **JMenuItem** je izvedena iz klase **AbstractButton**, pa se ponaša donekle slično dugmetu. Ona obezbeđuje stavku koja se može staviti u meni. Postoje i tri tipa izvedena iz klase **JMenuItem**: **JMenu** za čuvanje drugih stavki tipa **JMenuItem** (moguće je napraviti kaskadne menije), **JCheckBoxMenuItem** koja sadrži polje za potvrdu što pokazuje da li je stavka menija izabrana i **JRadioButtonMenuItem** koja sadrži radio-dugme.

Kao napredniji primer za pravljenje menija ponovo ćemo razmotriti različite ukuse sladoleda. Ovaj primer prikazuje i kaskadne menije, prečice s tastature, stavke menija s poljima za potvrdu i način za dinamičko menjanje menija:

```
//: gui/Meniji.java
// Podmeniji, polja za potvrdu u meniju, kaskadni
// meniji, prečice i komande za pokretanje.
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class Meniji extends JFrame {
  private String[] ukusi = {
    "Čokolada", "Jagoda","Vanila", "Pepermint",
    "Moka", "Rum grozđice", "Praline krem", "Pita od blata"
  };
  private JTextField t = new JTextField("Bez ukusa", 30);
  private JMenuBar mb1 = new JMenuBar();
  private JMenu
    f = new JMenu("Datoteka"),
    m = new JMenu("Ukusi"),
    s = new JMenu("Bezbednost");
  // Alternativni pristup:
  private JCheckBoxMenuItem[] bezbednost = {
    new JCheckBoxMenuItem("Cuvar"),
    new JCheckBoxMenuItem("Sakrij")
  };
  private JMenuItem[] datoteka = { new JMenuItem("Otvori"), };
  // Druga linija menija, za zamenu:
  private JMenuBar mb2 = new JMenuBar();
  private JMenu fooBar = new JMenu("fooBar");
  private JMenuItem[] ostalo = {
    // Dodavanje prečice za meni vrlo je
    // jednostavno, ali njih mogu da imaju samo
    // stavke JMenuItems u svojim konstruktorima:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Nema prečice:
    new JMenuItem("Baz"),
  };
  private JButton b = new JButton("Kaskadni meniji");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Osvežavanje slike
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem cilj = (JMenuItem)e.getSource();
      String actionCommand = cilj.getActionCommand();
      if(actionCommand.equals("Otvori")) {
        String s = t.getText();
        boolean izabrana = false;
```

```
        for(String  ukus : ukusi)
          if(s.equals(ukusi))
            izabrana = true;
        if(!izabrana)
          t.setText("Prvo izaberite ukus!");
        else
          t.setText("Otvaranje"+ s +". Mmm, mm!");
      }
    }
  }
  class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem cilj = (JMenuItem)e.getSource();
      t.setText(cilj.getText());
    }
  }
  // Možete i da napravite zasebnu klasu
  // za svaku stavku tipa MenuItem.
  // Tada nećete morati da otkrivate o kojem tipu se radi:
  class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Izabran Foo");
    }
  }
  class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Izabran Bar");
    }
  }
  class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Izabran Baz");
    }
  }
  class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      JCheckBoxMenuItem cilj =
        (JCheckBoxMenuItem)e.getSource();
      String actionCommand = cilj.getActionCommand();
      if(actionCommand.equals("Cuvar"))
        t.setText("Čuvaj sladoled! " +
          "Čuva se " + cilj.getState());
      else if(actionCommand.equals("Sakrij"))
        t.setText("Sakrij sladoled! " +
          "Da li je skriven? " + cilj.getState());
    }
  }
  public Meniji() {
    ML ml = new ML();
```

```
    CMIL cmil = new CMIL();
    bezbednost.setActionCommand("Cuvar");
    bezbednost.setMnemonic(KeyEvent.VK_G);
    bezbednost.addItemListener(cmil);
    bezbednost.setActionCommand("Sakrij");
    bezbednost.setMnemonic(KeyEvent.VK_H);
    bezbednost.addItemListener(cmil);
    ostalo.addActionListener(new FooL());
    ostalo.addActionListener(new BarL());
    ostalo.addActionListener(new BazL());
    FL fl = new FL();
    int n = 0;
    for(String ukus : ukusi) {
      JMenuItem mi = new JMenuItem(ukusi);
      mi.addActionListener(fl);
      m.add(mi);
      // Dodavanje razmaka u određenim intervalima:
      if((n++  + 1) % 3 == 0)
        m.addSeparator();
    }
    for(JCheckBoxMenuItem bbdn : bezbednost)
      s.add(bbdn);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < datoteka.length; i++) {
      datoteka[i].addActionListener(fl);
      f.add(datoteka[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    add(t, BorderLayout.CENTER);
    // Podešavanje sistema za kaskadne menije:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    add(b, BorderLayout.NORTH);
    for(JMenuItem ost : ostalo)
      fooBar.add(ost);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
  public static void main(String[] args) {
    run(new Meniji(), 300, 200);
  }
} ///:~
```

U ovom programu smestio sam stavke menija u nizove, a zatim sam svaku stavku niza prosledio metodi **add**( ). Zbog toga je dodavanje ili uklanjanje stavke menija nešto lakše.

Ovaj program ne pravi jednu, već dve linije menija tipa **JMenuBar** da bi pokazao kako se linije menija mogu aktivno menjati dok se program izvršava. Vidite da je linija sastavljena od objekta klase **JMenu**, odnosno da je svaki objekat tipa **JMenu** napravljen od stavki tipa **JMenuItem**, **JCheckBoxItem** ili čak od drugih menija tipa **JMenu** (podmenija). Kada se linija menija sastavi, može se prikazati u tekućem prozoru metodom **setJMenuBar( )**. Obratite pažnju na to da se – nakon pritiska dugmeta – metodom **getJMenuBar( )** proverava koji meni je trenutno prikazan, a zatim se on zamenjuje drugom linijom menija.

Pravopis i raspored malih i velikih slova presudno su važni prilikom ispitivanja natpisa Otvori, ali Java ne ukazuje na grešku ako ne nađe odgovarajuću stavku. Ovakva vrsta poređenja znakovnih nizova čest je izvor grešaka u programiranju.

Potvrđivanje i odjavljivanje stavki menija obavlja se automatski. Kôd kojim se obrađuju stavke tipa **JCheckBoxMenuItem** prikazuje dva načina za određivanje onoga što je potvrđeno: pronalaženje istih znakovnih nizova (kao što je rečeno, to nije naročito bezbedan pristup iako se s vremena na vreme koristi) i pronalaženje istih ciljnih objekata događaja. Kao što je pokazano, metoda **getState( )** može se koristiti za ustanovljavanje stanja. Stanje stavke **JCheckBoxMenuItem** možete da promenite i metodom **setState( )**.

Događaji za menije su donekle nedosledni i mogu da izazovu zabunu: stavke tipa **JMenuItem** koriste prijemnike tipa **ActionListener**, dok stavke tipa **JCheckboxMenuItem** koriste prijemnike tipa **ItemListener**. Objekti tipa **JMenu** mogu da podrže i prijemnik tipa **ActionListener**, ali to obično nije korisno. Po pravilu, prijemnici se povezuju sa svim stavkama **JMenuItem**, **JCheckBoxMenuItem** ili **RadioButtonMenuItem**, ali je u primeru pokazano kako se objekti klasa **ItemListener** i **ActionListener** povezuju s raznim komponentama menija.

Biblioteka Swing podržava prečice s tastature, što znači da možete aktivirati bilo koji objekat klase izvedene iz klase **AbstractButton** (dugme, stavku menija itd.) korišćenjem tastature umesto miša. To je prilično jednostavno: za stavku **JMenuItem** možete da upotrebite preklopljeni konstruktor čiji je drugi argument identifikator tastera. Međutim, većina dugmadi tipa **AbstractButton** nema takve konstruktore, pa je opštiji način za rešavanje problema da se koristi metoda **setMnemonic( )**. U prethodnom primeru dodate su prečice u neke stavke menija; indikatori prečica automatski se pojavljuju na komponentama.

Vidi se i korišćenje metode **setActionCommand( )** kojom se zadaje *komanda akcije*. To je malo neobično zato što je u svim slučajevima „komanda akcije" ista kao natpis na komponenti menija. Zašto se ne upotrebi postojeći natpis umesto novog znakovnog niza? Zbog internacionalizacije. Ako ovaj program prilagodite drugom jeziku, menjaćete samo natpise u meniju, ali ne i kôd (to bi bez sumnje prouzrokovalo nove greške). Da bi se kôd koji traži znakovne nizove povezane s komponentama menija lakše izvršavao, „komanda akcije" je nepromenljiva, dok se natpis menija može menjati. Ceo kôd radi s „komandom akcije", pa na njega ne utiču promene natpisa menija. Obratite pažnju na to da se u ovom programu ne ispituju sve komponente menija, pa se onima koje se ne ispituju ne dodeljuje komanda akcije.

Najveći deo posla odvija se u prijemnicima. Klasa **BL** zamenjuje linije menija (objekata tipa **JMenuBar**). Klasa **ML** primenjuje pristup „otkrij ko je zvonio" tako što ispituje izvor događaja (**ActionEvent**a) i konvertuje ga u **JMenuItem**, a potom ispituje znakovni niz komande akcije kroz nadovezane naredbe **if**.

Prijemnik **FL** je jednostavan, mada obrađuje sve različite ukuse u meniju ukusa. Ovaj pristup je koristan ako je logika dovoljno prosta, ali ćete po pravilu primenjivati pristup upotrebljen za **FooL**, **BarL** i **BazL** u kojem se prijemnici povezuju samo s jednom stavkom menija, pa nije potrebna nikakva dodatna logika za otkrivanje i zna se tačno ko je pozvao prijemnik. Čak i ako se broj klasa koje su napravljene na ovaj način poveća, kôd unutar njih nije dugačak pa je ovakav postupak bezbedniji.

Vidite da kôd menija brzo narasta i postaje neuredan. I u ovom slučaju treba koristiti vizuelnu razvojnu alatku. Dobra alatka te vrste istovremeno će i održavati menije.

Vežba 19: (3) Promenite program **Meniji.java** tako da se u meniju koriste radio-dugmad umesto polja za potvrdu.

Vežba 20: (6) Napišite program koji tekstualnu datoteku deli na reči. Stavite te reči u menije i podmenije kao stavke.

Iskačući meniji

Najjednostavniji način za primenu iskačućeg menija tipa **JPopupMenu** jeste da se napravi unutrašnja klasa koja proširuje klasu **MouseAdapter**, i da se zatim po jedan objekat te unutrašnje klase pridruži svakoj komponenti koja treba da izaziva iskakanje menija:

```
//: gui/Iskakanje.java
// Pravljenje iskačućih menija pomoću biblioteke Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class Iskakanje extends JFrame {
  private JPopupMenu meni = new JPopupMenu();
  private JTextField t = new JTextField(10);
  public Iskakanje() {
    setLayout(new FlowLayout());
    add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText(((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Juce");
    m.addActionListener(al);
    meni.add(m);
    m = new JMenuItem("Danas");
```

```
    m.addActionListener(al);
    meni.add(m);
    m = new JMenuItem("Sutra");
    m.addActionListener(al);
    meni.add(m);
    meni.addSeparator();
    m = new JMenuItem("Nikad");
    m.addActionListener(al);
    meni.add(m);
    PrijemnikMenija pl = new PrijemnikMenija();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class PrijemnikMenija extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      iskociAkoTreba(e);
    }
    public void mouseReleased(MouseEvent e) {
      iskociAkoTreba(e);
    }
    private void iskociAkoTreba(MouseEvent e) {
      if(e.isPopupTrigger()) {
        meni.show(e.getComponent(), e.getX(), e.getY());
      }
    }
  }
  public static void main(String[] args) {
    run(new Iskakanje(), 300, 200);
  }
} ///:~
```

Isti prijemnik tipa **ActionListener** povezuje se sa svim stavkama menija (objektima tipa **JMenuItem**). Prijemnik uzima tekst natpisa menija i umeće ga u tekstualno polje.

Crtanje

U dobroj biblioteci za projektovanje grafičkih okruženja crtanje bi trebalo da bude prilično lako, a u biblioteci Swing tako i jeste. Problem sa svakim primerom crtanja jeste to što su izračunavanja kojima se zadaje gde će se šta crtati obično mnogo složenija od poziva procedura za crtanje. Izračunavanja su često pomešana s pozivima procedura, pa pristup izgleda složeniji nego što stvarno jeste.

Radi jednostavnosti, razmotrimo problem s predstavljanjem podataka na ekranu; ovde će podatke obezbeđivati ugrađena metoda **Math.sin( )**, tj. matematička funkcija sinus. Da bi sve bilo zanimljivije, i da bi se još jednom pokazalo koliko se komponente biblioteke Swing lako koriste, na dno obrasca biće postavljen klizač za dinamičko kontrolisanje broja prikazanih perioda sinusoide. Pored toga, ako promenite veličinu prozora, videćete da se sinusoida prilagođava toj novoj veličini.

Iako se bilo koja komponenta tipa **JComponent** može koristiti kao platno za crtanje, ako želite jednostavnu površinu (pano) za crtanje, obično treba izvesti klasu iz klase **JPanel**. Jedino je potrebno redefinisati metodu **paintComponent( )** koja se poziva kad god se komponenta iscrtava na ekranu. (Obično ne morate da brinete o tome kada će ova metoda biti pozvana, pošto o tome odlučuje biblioteka Swing.) Swing prosleđuje ovoj metodi objekat tipa **Graphics**, a taj objekat zatim možete da koristite za crtanje ili slikanje po površini.

U sledećem primeru se sva logika povezana s crtanjem nalazi u klasi **CrtajSinusoidu**; klasa **Sinusoida** samo podešava program i klizač. Metoda **zadajCikluse( )** omogućuje drugom objektu, u ovom slučaju klizaču, da kontroliše broj ciklusa.

```
//: gui/Sinusoida.java
// Crtanje u biblioteci Swing, korišćenjem klizača JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

class CrtajSinusoidu extends JPanel {
  private static final int FAKTORSKALE = 200;
  private int ciklusi;
  private int tacke;
  private double[] sinusi;
  private int[] tck;
  public CrtajSinusoidu() { zadajCikluse(5); }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maksimalnaSirina = getWidth();
    double hkorak = (double)maksimalnaSirina/(double)tacke;
    int maksimalnaVisina = getHeight();
    tck = new int[tacke];
    for(int i = 0; i < tacke; i++)
      tck[i] =
        (int)(sinusi[i] * maksimalnaVisina/2 * .95 + maksimalnaVisina/
2);
    g.setColor(Color.RED);
    for(int i = 1; i < tacke; i++) {
      int x1 = (int)((i - 1) * hkorak);
      int x2 = (int)(i * hkorak);
      int y1 = tck[i-1];
      int y2 = tck[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
  public void zadajCikluse(int noviCiklusi) {
    ciklusi = noviCiklusi;
    tacke = FAKTORSKALE * ciklusi * 2;
    sinusi = new double[tacke];
    for(int i = 0; i < tacke; i++) {
```

```
        double radijani = (Math.PI/FAKTORSKALE) * i;
        sinusi[i] = Math.sin(radijani);
      }
      repaint();
    }
  }

  public class Sinusoida extends JFrame {
    private CrtajSinusoidu sinusi = new CrtajSinusoidu();
    private JSlider ciklusi = new JSlider(1, 30, 5);
    public Sinusoida() {
      add(sinusi);
      ciklusi.addChangeListener(new ChangeListener(){
        public void stateChanged(ChangeEvent e) {
          sinusi.zadajCikluse(
            ((JSlider)e.getSource()).getValue());
        }
      });
      add(BorderLayout.SOUTH, ciklusi);
    }
    public static void main(String[] args) {
      run(new Sinusoida(), 700, 400);
    }
  } ///:~
```

Svi podaci članovi i nizovi koriste se u izračunavanjima tačaka sinusoide: promenljiva **ciklusi** označava broj potpunih ciklusa sinusoida koji treba da budu prikazani, **tacke** sadrži ukupan broj tačaka, **sinusi** sadrži vrednosti sinusne funkcije, a **tck** sadrži Y koordinate tačaka koje se crtaju na panou. Metoda **zadajCikluse( )** pravi nizove na osnovu broja potrebnih tačaka i brojevima popunjava niz **sinusi**. Pozivanjem metode **repaint( )** iz metode **zadajCikluse( )** prouzrokuje se poziv metode **paintComponent( )** u kojoj se dovršava izračunavanje i ponovno iscrtavanje.

Kada redefinišete metodu **paintComponent( )**, prvo morate da pozovete verziju te metode iz natklase. Nakon toga možete da radite šta god želite; obično se za crtanje i bojenje piksela na panou **JPanel** koriste metode klase **Graphics** koje možete da pronađete u dokumentaciji za biblioteku **java.awt.Graphics** (na adresi *http://java.sun.com*). Primetićete da je ovde najveći deo koda posvećen izračunavanjima; jedine dve metode koje zaista crtaju su **setColor( )** i **drawLine( )**. Verovatno će tako biti i u vašim programima koji prikazuju grafičke podatke: najveći deo vremena provodićete smišljajući šta želite da nacrtate, dok će stvarni postupak crtanja biti sasvim jednostavan.

Dok sam pisao ovaj program, puno vremena sam utrošio na to da postignem da se sinusoida prikaže. Čim sam uspeo, palo mi je na pamet da bi bilo lepo kada bi broj perioda mogao dinamički da se menja. Zbog iskustava iz drugih programskih jezika oklevao sam da to uradim, ali se ispostavilo da je to najlakši deo posla. Napravio sam klizač – objekat klase **JSlider** (argumenti su krajnja leva vrednost, krajnja desna vrednost i početna vrednost klizača, ali ima i drugačijih konstruktora) i ubacio sam ga u klasu **JFrame**. Zatim sam pregledao HTML dokumentaciju i primetio da je jedini prijemnik klizača **addChangeListener**. On dobija informaciju kad god se položaj klizača promeni. Jedina metoda tog

interfejsa je **stateChanged( )**, što se moglo zaključiti po imenu prijemnika. Njen argument je objekat tipa **ChangeEvent** koji omogućuje da se pregleda izvor promene i pročita nova vrednost. Pozivanjem metode **zadajCikluse( )** objekta **sinusi** ponovo se iscrtava pano s novom sinusoidom.

Videćete da se većina problema u biblioteci Swing može rešiti sličnim postupkom, što je obično dosta lako, čak i ako neku komponentu nikada ranije niste koristili.

Ako su vaši problemi složeniji, postoje i naprednije metode za crtanje, uključujući i zrna Jave nezavisnih proizvođača i interfejs za programiranje Java 2D. Ta rešenja prevazilaze okvire ove knjige, ali bi trebalo da ih potražite ako vaš kod za crtanje postane previše problematičan.

Vežba 21: (5) Izmenite program **Sinusoida.java** tako što ćete objekat klase **Sinusoida** pretvoriti u zrno, i to tako što ćete mu dodeliti metode za čitanje i zadavanje svojstava.

Vežba 22: (7) Napravite aplikaciju pomoću klase **SwingKonzola**. Trebalo bi da ima tri klizača, po jedan za crvenu, zelenu i plavu boju u **java.awt.Color**. Ostatak obrasca bi trebalo da bude objekat tipa **JPanel** koji prikazuje boju zadatu s tri klizača. Postavite i polja za tekst čiji se sadržaj ne može menjati i koja prikazuju tekuće RGB vrednosti.

Vežba 23: (8) Na osnovu aplikacije **Sinusoida. java** napravite program koji na ekranu prikazuje rotirajući kvadrat. Neka jedan klizač upravlja brzinom rotacije, a drugi veličinom prozora.

Vežba 24: (7) Sećate li se igračke s dva dugmeta – jednim koje kontroliše vertikalno kretanje nacrtane tačke i drugim koje kontroliše horizontalno kretanje? Napravite takvu igračku koristeći program **Sinusoida. java**. Umesto dugmadi upotrebite klizače. Dodajte dugme kojim se briše cela skica.

Vežba 25: (8) Na osnovu aplikacije **Sinusoida. java** napravite program (aplikaciju koja upotrebljava klasu **SwingKonzola**) koji na ekranu crta animiranu sinusoidu što se pomera u prozoru za prikazivanje kao da je u pitanju osciloskop. Animaciju neka pokreće klasa **java.util.Timer**, a njenu brzinu neka određuje kontrola **javax.swing.JSlider**.

Vežba 26: (5) Izmenite prethodnu vežbu tako da aplikacija ima više panela sa sinusoidama. Broj panela neka se zadaje parametrom na komandnoj liniji.

Vežba 27: (5) Izmenite vežbu 25 tako da animaciju pokreće klasa **javax.swing.Timer**. Obratite pažnju na razliku između ove animacije i one u kojoj se koristi klasa **java.util.Timer**.

Vežba 28: (7) Napravite klasu za kockanje (bez grafičkog korisničkog okruženja). Neka se baca pet kocaka i to više puta. Nacrtajte krivu koja prikazuje zbir palih brojeva i dinamički se ažurira nakon svakog bacanja.

Okviri za dijalog

Okvir za dijalog je prozor koji iskače iz drugog prozora. Njegova svrha je da se izbori sa određenom temom a da ne zatrpa detaljima početni prozor. Dijalozi se često koriste u grafičkim programskim okruženjima.

Da biste napravili okvir za dijalog, treba da nasledite klasu **JDialog**, što je samo jedna vrsta prozora, poput klase **JFrame**. Klasa **JDialog** ima svoj raspoređivač (podrazumevani je **BorderLayout**), a za obradu događaja treba da dodate prijemnike. Evo vrlo jednostavnog primera:

```
//: gui/Dijalozi.java
// Pravljenje i korišćenje okvira za dijalog.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

class MojDijalog extends JDialog {
  public MojDijalog(JFrame roditelj) {
    super(roditelj, "Moj dijalog", true);
    setLayout(new FlowLayout());
    add(new JLabel("Evo mog dijaloga"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dispose(); // Zatvara dijalog
      }
    });
    add(ok);
    setSize(150,125);
  }
}

public class Dijalozi extends JFrame {
  private JButton b1 = new JButton("Okvir za dijalog");
  private MojDijalog dlg = new MojDijalog(null);
  public Dijalozi() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dlg.setVisible();
      }
    });
    add(b1);
  }
  public static void main(String[] args) {
    run(new Dijalozi(), 125, 75);
  }
} ///:~
```

Kada se napravi objekat klase **JDialog**, treba pozvati metodu **setVisible(true)** da bi se dijalog prikazao i aktivirao. Nakon zatvaranja prozora dijaloga, pozovite metodu **dispose( )** da biste oslobodili resurse koje prozor dijaloga (i dalje) koristi.

Sledeći primer je složeniji. Dijalog sadrži tabelu dugmadi tipa **IksOksDugme** (napravljena je pomoću klase **GridLayout**). Ovo dugme oko sebe iscrtava okvir i, u zavisnosti od svog stanja, u sredini ne prikazuje ništa, prikazuje x ili o. Dugme je u početku prazno, a zatim se, u zavisnosti od toga ko je na redu da igra, pretvara u x ili o. Međutim, ono naizmenično menja stanja između x i o i kada ga pritiskate. (Na taj način igra iks-oks se malo komplikuje.) Pored toga, okvir za dijalog može da sadrži proizvoljan broj redova i kolona, što se zadaje promenom brojeva u glavnom prozoru aplikacije.

```
//: gui/IksOks.java
// Prikaz okvira za dijalog
// i izrade komponenata.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class IksOks extends JFrame {
  private JTextField
    redovi = new JTextField("3"),
    kolone = new JTextField("3");
  private enum Stanje { PRAZAN, XX, OO }
  static class IksOksDijalog extends JDialog {
    private Stanje potez = Stanje.XX; // Počinje igrač x
    // s = broj ćelija po širini
    // v = broj ćelija po visini
    IksOksDijalog(int s, int v) {
      setTitle("Igra");
      setLayout(new GridLayout(s, v));
      for(int i = 0; i < s * v; i++)
        add(new IksOksDugme());
      setSize(s * 50, v * 50);
      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    class IksOksDugme extends JPanel {
      private Stanje stanje = Stanje.PRAZAN;
      public IksOksDugme() { addMouseListener(new ML()); }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int
            x1 = 0, y1 = 0;
            x2 = getSize().width - 1;
            y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int siroko = x2/2, visoko = y2/2;
        if(stanje == Stanje.XX) {
          g.drawLine(x1, y1, x1 + siroko, y1 + visoko);
```

```
          g.drawLine(x1, y1 + visoko, x1 + siroko, y1);
        }
        if(stanje == Stanje.OO) {
          g.drawOval(x1, y1, x1 + siroko/2, y1 + visoko/2);
        }
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(stanje == Stanje.PRAZAN) {
            stanje = potez;
            potez =
               (potez == Stanje.XX ? Stanje.OO : Stanje.XX);
          }
          else
            stanje =
              (stanje == Stanje.XX ? Stanje.OO : Stanje.XX);
          repaint();
        }
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JDialog d = new IksOksDijalog(
        new Integer(redovi.getText()),
        new Integer(kolone.getText()));
      d.setVisible(true);
    }
  }
  public IksOks() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Redovi", JLabel.CENTER));
    p.add(redovi);
    p.add(new JLabel("Kolone", JLabel.CENTER));
    p.add(kolone);
    add(p, BorderLayout.NORTH);
    JButton b = new JButton("potez");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    run(new IksOks(), 200, 200);
  }
} ///:~
```

Pošto samo spoljašnji nivo klase može imati statičke članove, unutrašnje klase ne mogu da sadrže statičke podatke ili ugnežđene klase.

Metoda **paintComponent**( ) crta kvadrat oko panoa, kao i oznake x ili o. Taj postupak je prepun dosadnih izračunavanja, ali je jednostavan.

Operaciju pritiskanja miša hvata objekat klase **MouseListener** koji prvo proverava da li je na panou nešto ispisano. Ako nije, ispituje se roditeljski prozor da bi se ustanovilo ko je na potezu i to se koristi za zadavanje stanja objekta **IksOksDugme**. Pomoću mehanizma unutrašnje klase, **IksOksDugme** zatim pristupa svom roditelju i menja igrača koji je na potezu. Ako dugme već prikazuje x ili o, stanje mu se menja. U izračunavanjima se vidi zgodno korišćenje ternarnog operatora uslovljavanja koji je opisan u poglavlju *Operatori*. Nakon promene stanja, objektu **Iks-OksDugme** menja se boja.

Konstruktor klase **IksOksDijalog** je prilično jednostavan: u tabelu tipa **GridLayout** dodaje dugmad po potrebi, a zatim povećava stranu svakog dugmeta za 50 piksela.

Objekat klase **IksOks** pokreće celu aplikaciju tako što pravi tekstualna polja za unošenje broja redova i kolona tabele dugmadi, dugme „potez" i prijemnik tipa **ActionListener**. Kada se pritisne dugme, uzimaju se podaci iz tekstualnih polja, a pošto su to znakovni nizovi, moraju se konvertovati u tip **int** pomoću **Integer** konstruktora koji prima **String** argument.

Dijalozi za rad s datotekama

Neki operativni sistemi imaju veliki broj posebnih ugrađenih dijaloga koji služe za biranje fontova, boje, štampača i sl. Gotovo svi grafički operativni sistemi imaju i dijaloge za otvaranje i snimanje datoteka, pa i Javina klasa **JFileChooser** kapsulira takve dijaloge radi lakšeg korišćenja.

U sledećoj aplikaciji prikazuju se dva oblika dijaloga tipa **JFileChooser**, jedan za otvaranje i drugi za snimanje. Najveći deo koda do sada bi trebalo da vam bude poznat, a sve zanimljive aktivnosti dešavaju se u prijemnicima događaja za dva različita dugmeta:

```
//: gui/TestBiranjaDatoteke.java
// Prikaz dijaloga za rad s datotekama.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class TestBiranjaDatoteke extends JFrame {
  private JTextField
    imeDatoteke = new JTextField(),
    dir = new JTextField();
  private JButton
    otvori = new JButton("Otvori"),
    snimi = new JButton("Snimi");
  public TestBiranjaDatoteke() {
    JPanel p = new JPanel();
    otvori.addActionListener(new OtvoriL());
    p.add(otvori);
    snimi.addActionListener(new SnimiL());
    p.add(snimi);
    add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
```

```
    imeDatoteke.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(imeDatoteke);
    p.add(dir);
    add(p, BorderLayout.NORTH);
  }
  class OtvoriL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Prikazuje dijalog "Otvori":
      int rVrednost = c.showOpenDialog(TestBiranjaDatoteke.this);
      if(rVrednost == JFileChooser.APPROVE_OPTION) {
        imeDatoteke.setText(c.getSelectedFile().getName());
          dir.setText(c.getCurrentDirectory().toString());
      }
      if(rVrednost == JFileChooser.CANCEL_OPTION) {
        imeDatoteke.setText("Pritisnuli ste dugme Cancel");
        dir.setText("");
      }
    }
  }
  class SnimiL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Prikazuje dijalog "Snimi":
      int rVrednost = c.showSaveDialog(TestBiranjaDatoteke.this);
      if(rVrednost == JFileChooser.APPROVE_OPTION) {
        imeDatoteke.setText(c.getSelectedFile().getName());
          dir.setText(c.getCurrentDirectory().toString());
      }
      if(rVrednost == JFileChooser.CANCEL_OPTION) {
        imeDatoteke.setText("Pritisnuli ste dugme Cancel");
        dir.setText("");
      }
    }
  }
  public static void main(String[] args) {
    run(new TestBiranjaDatoteke(), 250, 150);
  }
} ///:~
```

Zapamtite da postoje razne verzije dijaloga **JFileChooser**, uključujući i one koje koriste filtre za sužavanje spiska dozvoljenih imena datoteka.

Metodom **showOpenDialog( )** poziva se dijalog za otvaranje datoteke, a metodom **showSaveDialog( )** dijalog za snimanje datoteke. Iz tih metoda se izlazi tek kada se dijalog zatvori. Objekat tipa **JFileChooser** posle toga i dalje postoji, pa možete da čitate podatke iz njega. Metode **getSelectedFile( )** i **getCurrentDirectory( )** dva su načina za ispitivanje rezultata operacije. Ako te metode vrate **null**, korisnik je u dijalogu pritisnuo dugme za odustajanje (Cancel).

Vežba 29: (3) U HTML dokumentaciji za biblioteku **java.swing** potražite klasu **JColorChooser**. Napišite program koji pravi dugme što, kada se pritisne, prikazuje komandu za biranje boja u obliku dijaloga.

HTML u komponentama biblioteke Swing

Svaka komponenta koja prikazuje tekst može da prikaže i HTML tekst koji će se formatirati u skladu s HTML pravilima. To znači da Swing komponenti možete vrlo lako dodati lepo formatiran tekst. Na primer:

```
//: gui/HTMLDugme.java
// Stavljanje HTML teksta na Swing komponente.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingKonzola.*;

public class HTMLDugme extends JFrame {
  private JButton b = new JButton(
    "<html><b><font size=+2>" +
    "<center>Zdravo!<br><i>Pritisni me!");
  public HTMLDugme() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        add(new JLabel("<html>"+
          "<i><font size=+4>Tras!"));
        // Izaziva ponovno iscrtavanje da bi se prikazao natpis:
        validate();
      }
    });
    setLayout(new FlowLayout());
    add(b);
  }
  public static void main(String[] args) {
    run(new HTMLDugme(), 200, 500);
  }
} ///:~
```

Tekst mora da započne oznakom <html>, a potom možete da upotrebljavate uobičajene HTML oznake. Obratite pažnju na to da nije obavezno korišćenje uobičajenih oznaka za zatvaranje.

Prijemnik tipa **ActionListener** u obrazac dodaje nov natpis tipa **JLabel** koji sadrži i HTML tekst. Međutim, taj natpis se ne dodaje tokom konstruisanja objekta, pa morate da pozovete kontejnersku metodu **validate( )** da biste prouzrokovali ponovno iscrtavanje komponenata (odnosno, prikazivanje novog natpisa).

HTML tekst možete da koristite i u objektima klasa **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** i **JCheckBox**.

Vežba 30: (3) Napišite program u kojem se HTML tekst ispisuje na svim objektima nabrojanim u prethodnom pasusu.

Klizači i linije napredovanja

Klizač (engl. *slider*) koji je već upotrebljen u primeru sinusoide, omogućuje korisniku da unosi podatke tako što ga pomera napred-nazad, što je u nekim situacijama intuitivno (na primer, za kontrolu jačine zvuka). Traka napredovanja (engl. *progress bar*) prikazuje podatke u relativnom odnosu, od nimalo do potpuno. Moj omiljen primer za ove grafičke elemente je da se klizač zakači za traku napredovanja tako da se saglasno pomeranju klizača pomera i traka napredovanja. U narednom primeru videćete i **ProgressMonitor**, što je bogatiji iskačući dijalog :

```
//: gui/Napredak.java
// Korišćenje traka napredovanja, klizača i monitora napredovanja.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class Napredak extends JFrame {
  private JProgressBar pb = new JProgressBar();
  private ProgressMonitor pm = new ProgressMonitor(
    this, "Nadziranje napretka", "Test", 0, 100);
private JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public Napredak() {
    setLayout(new GridLayout(2,1));
    add(pb);
    pm.setProgress(0);
    pm.setMillisToPopup(1000);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Pomeri me"));
    pb.setModel(sb.getModel()); // Zajednički model
    add(sb);
    sb.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        pm.setProgress(sb.getValue());
      }
    });
  }
  public static void main(String[] args) {
    run(new Napredak(), 300, 200);

} ///:~
```

Ključ za povezivanje klizača i trake napredovanja leži u zajedničkom korišćenju modela, u redu:

```
pb.setModel(sb.getModel());
```

Ove dve komponente mogli biste da kontrolišete i pomoću prijemnika, ali je ovo rešenje prostije. **ProgressMonitor** nema model, pa sam morao da upotrebim prijemnik. Verovatno ste primetili da se ovaj **ProgressMonitor** kreće samo unapred i da se zatvara kada dođe do kraja.

Klasa **JProgressBar** je prilično očigledna, ali klasa **JSlider** ima mnoštvo opcija, npr. orijentisanje, veće i manje zareze na klizaču itd. Obratite pažnju na to kako se lako dodaje ivica sa natpisom.

Vežba 31: (8) Napravite „indikator asimptotskog napredovanja" koji se usporava s približavanjem završnoj tački. Dodajte mu nasumično pogrešno ponašanje da bi s vremena na vreme izgledalo kao da počinje da se ubrzava.

Vežba 32: (6) Izmenite program **Napredak.java** tako da za povezivanje klizača i trake napredovanja ne koristi zajedničke modele, već prijemnik.

Biranje izgleda i ponašanja

Jedna od zanimljivih mogućnosti grafičke biblioteke Swing jeste postizanje promenljivog izgleda i ponašanja (engl. *Pluggable Look & Feel*). To omogućuje programu da simulira izgled i ponašanje različitih radnih okruženja. Možete da uradite zanimljive stvari, npr. da dinamički menjate izgled i ponašanje programa. Međutim, obično ćete koristiti samo jednu ili dve mogućnosti: birаćete izgled koji je isti na svim platformama (to je u biblioteci Swing „metal"), ili izgled sistema u kome trenutno radite, da bi program napisan na Javi izgledao kao da je pravljen za taj sistem. Kôd kojim se biraju izgled i ponašanje prilično je jednostavan, ali morate da ga izvršite *pre* nego što počnete da pravite vizuelne komponente. Komponente se *prave* na osnovu tekućeg izgleda i ponašanja, tj. postojeće komponente neće biti promenjene samo zato što ste vi usred programa odlučili da im promenite izgled (takav postupak je složeniji i neuobičajen, a obrađen je u knjigama posvećenim grafičkoj biblioteci Swing).

Zapravo, ako želite da program izgleda isto na svim platformama, ne morate ništa da radite, jer se „metalni" izgled podrazumeva. Ali ako želite da koristite izgled i ponašanje tekućeg radnog okruženja,[8] treba samo da umetnete sledeći kôd, obično na početak metode **main( )**; obavite to pre pravljenja bilo kakvih grafičkih elemenata:

```
try {
   UIManager.setLookAndFeel (
      UIManager.getSystemLookAndFeelClassName());
} catch(Exception e) {
   throw new RuntimeException(e);
}
```

[8] Pitanje je koliko uspešno Swing oponaša razna radna okruženja.

Blok **catch** može da bude prazan zato što će **UIManager** učitati podrazumevan izgled i ponašanje ako pokušaj da zadate drugačiji izgled ne uspe. Međutim, informacija o izuzetku biva veoma korisna pri traženju grešaka, pa u bloku **catch** možete staviti barem naredbu za ispisivanje.

Evo programa koji na osnovu argumenta s komandne linije zadaje izgled i ponašanje, i pokazuje kako neke komponente u tom slučaju izgledaju:

```
//: gui/IzgledIPonasanje.java
// Biranje različitih izgleda i ponašanja.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingKonzola.*;

public class IzgledIPonasanje extends JFrame {
  private String[] izbor =
    "eci peci pec ti si mali zec".split(" ");
  private Component[] uzorci = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(izbor),
    new JList(izbor),
  };
  public IzgledIPonasanje() {
    super("Izgled i ponasanje");
    setLayout(new FlowLayout());
    for(Component komponenta : uzorci)
      add(komponenta);
  }
  private static void greskaKoriscenja() {
    System.out.println(
      "Korišćenje:IzgledIPonasanje [prenosiv|sistemski|motif]");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) greskaKoriscenja();
    if(args[0].equals("prenosiv")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else if(args[0].equals("sistemski")) {
      try {
```

```
          UIManager.setLookAndFeel(UIManager.
            getSystemLookAndFeelClassName());
        } catch(Exception e) {
          e.printStackTrace();
        }
      } else if(args[0].equals("motif")) {
        try {
          UIManager.setLookAndFeel("com.sun.java."+
            "swing.plaf.motif.MotifLookAndFeel");
        } catch(Exception e) {
          e.printStackTrace();
        }
      } else greskaKoriscenja();
      // Obratite pažnju na to da se izgled i ponašanje moraju
      // zadati pre pravljenja bilo kakvih komponenata.
      run(new IzgledIPonasanje(), 300, 300);
    }
  } ///:~
```

Za eksplicitno zadavanje izgleda i ponašanja koristi se **MotifLookAndFeel**. Samo se taj i podrazumevani „metalni" izgled sigurno mogu koristiti na svim platformama; iako postoje komponente koje oponašaju izgled i ponašanje Windowsa i MacOS-a, one se mogu koristiti samo na odgovarajućim platformama (dobijaju se kada pozovete metodu **getSystemLookAndFeelClassName( )** dok radite na određenoj platformi).

Moguće je napraviti i sopstveni paket elemenata koji opisuju izgled i ponašanje, na primer ako pravite strukturu aplikacije za kompaniju koja želi da ta aplikacija posebno izgleda. To je velik posao i uveliko premašuje temu ove knjige (zapravo, otkrićete da se njome ne bave ni mnoge knjige isključivo posvećene biblioteci Swing).

Stabla, tabele i clipboard

Kratak uvod i primere za ove teme naći ćete u mrežnim dodacima ovog poglavlja na adresi *www.MindView.net*.

JNLP i Java Web Start

Aplet može biti *potpisan* da bi se ostvarila bezbednost. To je pokazano u mrežnom dodatku ovog poglavlja na adresi *www.MindView.net*. Potpisani apleti su moćni i mogu da zamene aplikacije, ali moraju da se izvršavaju u čitaču Weba. To znači da čitač koji se izvršava na korisnikovom računaru prouzrokuje dodatne režijske troškove a korisničko okruženje apleta ograničeno je i često zbunjuje. Čitač Weba ima sopstvene menije i palete sa alatkama, koji se prikazuju iznad apleta.[*]

Javin protokol za mrežno pokretanje (*Java Network Launch Protocol, JNLP*) rešava problem a da ne ugrožava prednosti apleta. JNLP aplikacija može se preuzeti i instalirati kao samostalna Java aplikacija na računaru korisnika. Ona se može pokretati s komandne lini-

[*] Ovaj odeljak napisao je Jeremy Meyer.

je, pomoću ikonice na radnoj površini ili programa za upravljanje aplikacijama koji je deo realizacije JNLP-a. Aplikacija se može pokretati čak i s Web lokacije s koje je bila preuzeta.

U vreme izvršavanja, JNLP aplikacija može dinamički da preuzima resurse sa Interneta i da automatski proverava verziju ako je korisnik povezan sa Internetom. To znači da ima sve prednosti apleta, a i prednosti samostalnih aplikacija.

Korisnikov računar mora oprezno da tretira JNLP aplikacije, kao i aplete. Zato se JNLP aplikacije izvršavaju samo u „bazenu s peskom", kao apleti. Ukoliko su u potpisanim JAR datotekama, korisnik može odlučiti da im ukaže poverenje i dopusti korišćenje resursa njegovog računara. (Isto važi i za aplete.) Za razliku od apleta, pomoću usluga JNLP interfejsa za programiranje JNLP aplikacije mogu zatražiti pristup određenim resursima korisnikovog računara i kada su u nepotpisanim JAR datotekama. Tokom izvršavanja programa, od korisnika se traži da odobri te zahteve.

JNLP je specifikacija protokola, a ne njegova realizacija. Zato je za korišćenje neophodna i neka realizacija. Java Web Start (JAWS) Sunova je besplatna zvanična referentna realizacija koja se isporučuje u sklopu Jave SE5. Ukoliko želite da je koristite za razvoj, njena JAR datoteka (**javaws.jar**) mora biti u putanji klasa vašeg računara; najjednostavnije rešenje je dodati datoteku **javaws.jar** putanji klasa –premestiti je iz njene uobičajene putanje Java instalacije na putanju **jre/lib**. Ako JNLP aplikaciju nameravate da instalirate s Web servera, on bi trebalo da prepoznaje MIME tip **application/x-java-jnlp-file**. To je unapred konfigurisano u novijim verzijama servera Tomcat (*http://jakarta.apache.org/tomcat*). Odgovarajuće pojedinosti pročitajte u korisničkim uputstvima za svoj server.

Nije teško napraviti JNLP aplikaciju. Napravite običnu aplikaciju, komprimujte je u JAR arhivu i zatim obezbedite datoteku za pokretanje – jednostavnu XML datoteku koja korisnikovom računaru daje sve informacije potrebne za preuzimanje i instaliranje te aplikacije. Ukoliko ne potpišete JAR arhivu, morate da koristite usluge koje interfejs za programiranje JNLP aplikacija pruža za sve tipove resursa kojima aplikacija treba da pristupi na računaru korisnika.

Sledi varijanta programa **TestBiranjaDatoteka.java** koja za otvaranje birača datoteka upotrebljava JNLP usluge, čime omogućuje da klasa bude distribuirana u obliku JNLP aplikacije u nepotpisanoj JAR arhivi.

```
//: gui/jnlp/JnlpBiranjeDatoteka.java
// Otvaranje datoteka na lokalnom računaru pomoću JNLP-a.
// {Requires: javax.jnlp.FileOpenService;
// javaws.jar mora biti u putanji klasa}
// Ovako ćete napraviti datoteku jnlpbiranjedatoteka.jar:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpbiranjedatoteka.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
```

```
public class JnlpBiranjeDatoteka extends JFrame {
  private JTextField imeDatoteke = new JTextField();
  private JButton
    otvori = new JButton("Otvori"),
    snimi = new JButton("Snimi");
  private JEditorPane ep = new JEditorPane();
  private JScrollPane jsp = new JScrollPane();
  private FileContents sadrzajDatoteke;
  public JnlpBiranjeDatoteka() {
    JPanel p = new JPanel();
    otvori.addActionListener(new OtvoriL());
    p.add(otvori);
    snimi.addActionListener(new SnimiL());
    p.add(snimi);
    jsp.getViewport().add(ep);
    add(jsp, BorderLayout.CENTER);
    add(p, BorderLayout.SOUTH);
    imeDatoteke.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(imeDatoteke);
    add(p, BorderLayout.NORTH);
    ep.setContentType("text");
    snimi.setEnabled(false);
  }
  class OtvoriL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileOpenService fs = null;
      try {
        fs = (FileOpenService)ServiceManager.lookup(
          "javax.jnlp.FileOpenService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          sadrzajDatoteke = fs.openFileDialog(".",
            new String[]{"txt", "*"});
          if(sadrzajDatoteke == null)
            return;
          imeDatoteke.setText(sadrzajDatoteke.getName());
          ep.read(sadrzajDatoteke.getInputStream(), null);
        } catch(Exception exc) {
          throw new RuntimeException(exc);
        }
        snimi.setEnabled(true);
      }
    }
```

```
  }
  class SnimiL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileSaveService fs = null;
      try {
        fs = (FileSaveService)ServiceManager.lookup(
          "javax.jnlp.FileSaveService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          sadrzajDatoteke = fs.saveFileDialog(".",
            new String[]{"txt"},
            new ByteArrayInputStream(
              ep.getText().getBytes()),
            sadrzajDatoteke.getName());
          if(sadrzajDatoteke == null)
            return;
          imeDatoteke.setText(sadrzajDatoteke.getName());
        } catch(Exception exc) {
          throw new RuntimeException(exc);
        }
      }
    }
  }
  public static void main(String[] args) {
    JnlpBiranjeDatoteka bd = new JnlpBiranjeDatoteka();
    bd.setSize(400, 300);
    bd.setVisible(true);
  }
} ///:~
```

Obratite pažnju na to da se klase **FileOpenService** i **FileCloseService** uvoze iz paketa **javax.jnlp** i da se dijalog **JFileChooser** nigde neposredno ne spominje. Obe usluge koje se ovde upotrebljavaju moraju biti zatražene metodom **ServiceManager.lookup( )**, a resursima na računaru korisnika može se pristupiti samo preko objekata koje ta metoda vrati. U ovom slučaju, u sistemu datoteka korisnikovog računara datoteke se čitaju i upisuju pomoću interfejsa **FileContent**, koji obezbeđuje JNLP. Svaki pokušaj neposrednog pristupanja resursima preko, recimo, objekta tipa **File** ili **FileReader**, izazvao bi bacanje izuzetka **SecurityException**, jednako kao da ih pokušate upotrebiti iz nepotpisanog apleta. Ukoliko želite da koristite te klase i da ne budete ograničeni na interfejse JNLP usluga, morate potpisati JAR arhivu.

Potrebnu JAR arhivu pravi komanda **jar** koja je na početku programa **JnlpBiranjeDatoteka.java** pretvorena u komentar. Ovo je datoteka za pokretanje prethodnog primera.

```
//:! gui/jnlp/biranjedatoteka.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
  codebase="file:C:/AAA-TIJ4/code/gui/jnlp"
  href="biranjedatoteka.jnlp">
  <information>
    <title>Primer FileChooser aplikacije</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp aplikacija za biranje datoteka
    </description>
    <description kind="short">
      Pokazuje otvaranje, čitanje i upisivanje u tekstualnu datoteku
    </description>
    <icon href="mindview.gif"/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.3+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlpbiranjedatoteka.jar" download="eager"/>
  </resources>
  <application-desc
    main-class="gui.jnlp.JnlpBiranjeDatoteka"/>
</jnlp>
///:~
```

Navedenu datoteku za pokretanje naći ćete u datoteci izvornog koda ove knjige (može se preuzeti na adresi *www.MindView.net*) pod imenom **biranjedatoteka.jnlp** (bez prvog i poslednjeg reda), u istom direktorijumu kao JAR arhiva. Kao što vidite, radi se o XML datoteci s jednom oznakom <**jnlp**>. Ona ima nekoliko podelemenata, manje-više jasnih po sebi.

Atribut **spec** elementa **jnlp** saopštava klijentskom sistemu koja verzija JNLP-a može da pokrene aplikaciju. Atribut **codebase** pokazuje na URL adresu na kojoj su resursi i datoteka za pokretanje. Ovde pokazuje na direktorijum lokalnog računara, što je dobar način za testiranje ove aplikacije. *Nadam se da shvatate kako tu putanju morate promeniti tako da pokazuje na odgovarajući direktorijum vašeg računara – tek tada će se program učitati.* Atribut **href** specificira ime datoteke koju treba učitati.

Oznaka **information** ima razne podelemente koji pružaju informacije o aplikaciji. Njih čita administrativna konzola Java Web Start ili druga njoj ekvivalentna realizacija JNLP-a koja instalira JNLP aplikaciju i omogućuje korisniku da je pokrene s komandne linije, pravi prečice itd.

Svrha oznake **resources** slična je svrsi oznake <applet> u HTML datoteci. Podelement **j2se** specificira verziju J2SE potrebnu za pokretanje aplikacije, a podelement **jar** JAR datoteku u kojoj je klasa arhivirana. Element **jar** ima atribut **download** čije vrednosti eager odnosno lazy kazuju realizaciji JNLP-a da li pre pokretanja aplikacije mora da učita celu arhivu ili ne mora.

Atribut **application-desc** saopštava realizaciji JNLP-a koja klasa je izvršna klasa (ulazna tačka) JAR arhive.

Drugi koristan podelement oznake **jnlp** jeste oznaka **security** koje u gornjoj datoteci nema. Evo kako izgleda oznaka **security**:

```
<security>
   <all-permissions/>
<security/>
```

Oznaku **security** treba da imaju aplikacije u potpisanim JAR arhivama. U prethodnom primeru nije potrebna zato što se svim lokalnim resursima pristupa pomoću JNLP usluga.

Pojedinosti o ostalim oznakama potražite u specifikaciji na adresi *http://java.sun.com/products/javawebstart/download-spec.html*.

Za pokretanje programa potrebna je stranica za preuzimanje koja sadrži hipertekstualnu vezu sa **.jnlp** datotekom. Evo kako ona izgleda (bez prvog i poslednjeg reda):

```
//:! gui/jnlp/biranjedatoteka.html
<html>
Sledite uputstva iz JnlpBiranjeDatoteka.java za pravljenje
jnlpfilechooser.jar, a zatim:
<a href="biranjedatoteka.jnlp">pritisnite ovde</a>
</html>
///:~
```

Pošto preuzmete aplikaciju, možete da je konfigurišete pomoću administrativne konzole. Ako Java Web Start koristite u Windowsu, prilikom druge upotrebe moći ćete da napravite prečicu do aplikacije. To ponašanje se može konfigurisati.

Ovde sam opisao samo dve JNLP usluge, a u tekućem izdanju ih ima sedam. Svakom od njih obavlja se određeni zadatak, kao što je štampanje ili isecanje i prenošenje na clipboard. Više informacija o tome potražite na adresi *http://java.sun.com*.

Paralelno izvršavanje i Swing

Kada programirate u Swingu, radite s nitima. To ste videli na početku ovog poglavlja kada ste saznali da sve zadatke treba pokretati metodom **SwingUtilities.invokeLater( )** Swingovog raspoređivača. Međutim, činjenica da ne morate eksplicitno praviti niti (objekte klase **Thread**) znači da bi problemi s nitima mogli da vas snađu kada se ne nadate. Vodite računa o tome da nit Swingovog raspoređivača uvek postoji i da sve Swing događaje obrađuje vadeći ih iz reda za čekanje i izvršavajući ih jedan za drugim. Ukoliko ne zaboravite na nit raspoređivača, manje su šanse da će vaša aplikacija upasti u uzajamnu blokadu ili uslov za trku.

U narednom odeljku razmotrićemo oblasti višenitnog izvršavanja na koje treba obratiti pažnju prilikom rada u Swingu.

Dugotrajni zadaci

Jedna od osnovnih grešaka pri programiranju grafičkih korisničkih okruženja jeste upotreba raspoređivača za izvršavanje dugotrajnog zadatka. Sledi jednostavan primer:

```
//: gui/DugotrajanZadatak.java
// Loše projektovan program.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingKonzola.*;

public class DugotrajanZadatak extends JFrame {
  private JButton
    d1 = new JButton("Pokreni dugotrajan zadatak"),
    d2 = new JButton("Ugasi dugotrajan zadatak");
  public DugotrajanZadatak() {
    d1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
          System.out.println("Zadatak prekinut");
          return;
        }
        System.out.println("Zadatak završen");
      }
    });
    d2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        // Da se prekinem?
        Thread.currentThread().interrupt();
      }
    });
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
  }
  public static void main(String[] args) {
    run(new DugotrajanZadatak(), 200, 150);
  }
} ///:~
```

Kada pritisnete **d1**, nit raspoređivača iznenada počinje da izvršava dugotrajan zadatak. Videćete da ni to dugme neće stići da se vrati iz utisnutog položaja, pošto je nit raspoređivača, koja bi inače osvežila sliku na ekranu, zauzeta tokom cele 3 sekunde. A i ništa drugo ne možete da uradite, npr. da pritisnete **d2**, pošto program ne odgovara dok se zadatak dugmeta **d1** ne završi i nit raspoređivača opet ne postane dostupna. Kôd u **d2** je pogrešan pokušaj da se problem reši prekidanjem niti raspoređivača.

Naravno, rešenje je da se dugotrajni procesi izvršavaju u zasebnim nitima. Sada ćemo upotrebiti jednonitni izvršilac (**Executor**, koji pozivajuće zadatke automatski ubacuje u red za čekanje. Odatle ih vadi i izvršava jedan po jedan:

```
//: gui/DugotrZadKojiSeMozePrekinuti.java
// Dugotrajni zadaci u nitima.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingKonzola.*;

class Zadatak implements Runnable {
  private static int brojac = 0;
  private final int id = brojac++;
  public void run() {
    System.out.println(this + " pokrenut");
    try {
      TimeUnit.SECONDS.sleep(3);
    } catch(InterruptedException e) {
      System.out.println(this + " prekinut");
      return;
    }
    System.out.println(this + " završen");
  }
  public String toString() { return "Zadatak " + id; }
  public long id() { return id; }
};

public class DugotrZadKojiSeMozePrekinuti extends JFrame {
  private JButton
    d1 = new JButton("Pokreni dugotrajan zadatak"),
    d2 = new JButton("Ugasi dugotrajan zadatak");
  ExecutorService izvrsilac =
    Executors.newSingleThreadExecutor();
  public DugotrZadKojiSeMozePrekinuti() {
    d1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Zadatak zadatak = new Zadatak();
        izvrsilac.execute(zadatak);
        System.out.println(zadatak + " dodat u red za čekanje");
      }
    });
    d2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        izvrsilac.shutdownNow(); // Prejako
      }
    });
```

```
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
  }
  public static void main(String[] args) {
    run(new DugotrZadKojiSeMozePrekinuti(), 200, 150);
  }
} ///:~
```

Ovo je bolje, ali kada pritisnete dugme **d2**, ono će pozvati metodu **shutdownNow( )** za **ExecutorService** i tako ugasiti taj zadatak. Ukoliko pokušate da dodate još zadataka, dobićete izuzetak. Dakle, pritisak na **d2** čini da program umre. Želeli smo da ugasimo tekući zadatak (i otkažemo zadatke u redu za čekanje), a ne da gasimo ceo program. Treba nam baš mehanizam Jave SE5 **Callable/Future** opisan u poglavlju *Paralelno izvršavanje*. Definisaćemo novu klasu **MenadzerZadataka**; ona sadrži *torku* koja obuhvata objekat tipa **Callable**. **Callable** je zadatak i objekat tipa **Future** koji je rezultat tog zadatka. Torka je potrebna zato što nam omogućuje da pratimo prvobitni zadatak, kako bismo imali i dodatne informacije kojima **Future** ne raspolaže. Evo kako to izgleda:

```
//: net/mindview/util/StavkaZadatka.java
// Objekat tipa Future i Callable koji ga je napravio.
package net.mindview.util;
import java.util.concurrent.*;

public class StavkaZadatka<R,C extends Callable<R>> {
  public final Future<R> buduci;
  public final C zadatak;
  public StavkaZadatka(Future<R> buduci, C zadatak) {
    this.buduci = buduci;
    this.zadatak = zadatak;
  }
} ///:~
```

U biblioteci **java.util.concurrent** zadatak nije podrazumevano dostupan preko klase **Future**, pošto on ne mora ni postojati kada od klase **Future** dobijete rezultat. Ovde zadatak postoji zato što smo ga snimili.

MenadzerZadataka je stavljen u paket **net.mindview.util** da bi bio dostupan kao uslužna klasa opšte namene:

```
//: net/mindview/util/MenadzerZadataka.java
// Upravljanje redom zadataka i njegovo izvršavanje.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class MenadzerZadataka<R,C extends Callable<R>>
extends ArrayList<StavkaZadatka<R,C>> {
```

```
  private ExecutorService exec =
    Executors.newSingleThreadExecutor();
  public void add(C zadatak) {
    add(new StavkaZadatka<R,C>(exec.submit(zadatak),zadatak));
  }
  public List<R> getResults() {
    Iterator<StavkaZadatka<R,C>> stavke = iterator();
    List<R> rezultati = new ArrayList<R>();
    while(stavke.hasNext()) {
      StavkaZadatka<R,C> stavka = stavke.next();
      if(stavka.buduci.isDone()) {
        try {
          rezultati.add(stavka.buduci.get());
        } catch(Exception e) {
          throw new RuntimeException(e);
        }
        stavke.remove();
      }
    }
    return rezultati;
  }
  public List<String> purge() {
    Iterator<StavkaZadatka<R,C>> stavke = iterator();
    List<String> rezultati = new ArrayList<String>();
    while(stavke.hasNext()) {
      StavkaZadatka<R,C> stavka = stavke.next();
      // Ostavi završene zadatke radi izveštavanja o rezultatima:
      if(!stavka.buduci.isDone()) {
        rezultati.add("Otkazujem " + stavka.zadatak);
        stavka.buduci.cancel(true); // Može da prekine
        stavke.remove();
      }
    }
    return rezultati;
  }
} ///:~
```

MenadzerZadataka je lista tipa **ArrayList** čije su stavke tipa **StavkaZadatka**. On sadrži i jednonitni **Executor**, pa kada pozovete metodu **add**( ) i prosledite joj objekat tipa **Callable**, ona šalje taj **Callable**, a rezultujući objekat tipa **Future** skladišti zajedno s prvobitnim zadatkom. Na taj način, ako išta bude trebalo da se radi s tim zadatkom, imaćete referencu na njega. Kao jednostavan primer, u metodi **purge**( ) koristi se metoda **toString**( ) tog zadatka.

To ćemo sada upotrebiti za upravljanje dugotrajnim zadacima u našem primeru:

```
//: gui/DugotrCallableKojiSeMozePrekinuti.java
// Korišćenje Callable objekata za dugotrajne zadatke.
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingKonzola.*;

class CallableZadatak extends Zadatak
implements Callable<String> {
  public String call() {
    run();
    return "Povratna vrednost od " + this;
  }
}

public class
DugotrCallableKojiSeMozePrekinuti extends JFrame {
  private JButton
    d1 = new JButton("Pokreni dugotrajan zadatak"),
    d2 = new JButton("Ugasi dugotrajan zadatak"),
    d3 = new JButton("Daj rezultate");
  private MenadzerZadataka<String,CallableZadatak> menadzer =
    new MenadzerZadataka<String,CallableZadatak>();
  public DugotrCallableKojiSeMozePrekinuti() {
    d1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        CallableZadatak zadatak = new CallableZadatak();
        menadzer.add(zadatak);
        System.out.println(zadatak + " dodat u red za čekanje");
      }
    });
    d2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String rezultat : menadzer.purge())
          System.out.println(rezultat);
      }
    });
    d3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // Primer poziva metode klase Zadatak:
        for(StavkaZadatka<String,CallableZadatak> tt :
            menadzer)
          tt.zadatak.id(); // Nije potrebna eksplicitna konverzija tipa
        for(String rezultat : menadzer.getResults())
          System.out.println(rezultat);
      }
    });
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
    add(d3);
```

```
  }
  public static void main(String[] args) {
    run(new DugotrCallableKojiSeMozePrekinuti(), 200, 150);
  }
} ///:~
```

Kao što vidite, **CallableZadatak** radi isto što i **Zadatak**, sem što vraća i rezultat – u ovom slučaju **String** identifikator zadatka.

Za rešavanje istog problema napisane su i ne-Swing uslužne klase (koje se ne isporučuju u standardnoj distribuciji Jave). To su **SwingWorker** (naći ćete je na Sunovoj Web lokaciji) i *Foxtrot* (na adresi *http://foxtrot.sourceforge.net*). Dok sam pisao knjigu, te klase nisu bile modifikovane tako da iskoriste mehanizam Jave SE5 **Callable/Future**.

Često krajnjem korisniku valja dati nekakav vizuelni znak da se zadatak izvršava i obavestiti ga dokle je odmaklo izvršavanje. To se obično radi pomoću klasa **JProgressBar** ili **ProgressMonitor**. U narednom primeru upotrebiću **ProgressMonitor**:

```
//: gui/NadziranDugotrCallable.java
// Prikazivanje napredovanja zadatka pomoću klase ProgressMonitors.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingKonzola.*;

class NadziranCallable implements Callable<String> {
  private static int brojac = 0;
  private final int id = brojac++;
  private final ProgressMonitor monitor;
  private final static int MAX = 8;
  public NadziranCallable(ProgressMonitor monitor) {
    this.monitor = monitor;
    monitor.setNote(toString());
    monitor.setMaximum(MAX - 1);
    monitor.setMillisToPopup(500);
  }
  public String call() {
    System.out.println(this + " pokrenut");
    try {
      for(int i = 0; i < MAX; i++) {
        TimeUnit.MILLISECONDS.sleep(500);
        if(monitor.isCanceled())
          Thread.currentThread().interrupt();
        final int napredovanje = i;
        SwingUtilities.invokeLater(
          new Runnable() {
            public void run() {
              monitor.setProgress(napredovanje);
```

```
            }
          }
        );
      }
    } catch(InterruptedException e) {
      monitor.close();
      System.out.println(this + " prekinut");
      return "Rezultat: " + this + " prekinut";
    }
    System.out.println(this + " završen");
    return "Rezultat: " + this + " završen";
  }
  public String toString() { return "Zadatak " + id; }
};
public class NadziranDugotrCallable extends JFrame {
  private JButton
    d1 = new JButton("Pokreni dugotrajan zadatak"),
    d2 = new JButton("Ugasi dugotrajan zadatak"),
    d3 = new JButton("Daj rezultate");
  private MenadzerZadataka<String,NadziranCallable> menadzer =
    new MenadzerZadataka<String,NadziranCallable>();
  public NadziranDugotrCallable() {
    d1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        NadziranCallable zadatak = new NadziranCallable(
          new ProgressMonitor(
            NadziranDugotrCallable.this,
            "Dugotrajan Zadatak", "", 0, 0)
        );
        menadzer.add(zadatak);
        System.out.println(zadatak + " dodat u red za čekanje");
      }
    });
    d2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String rezultat : menadzer.purge())
          System.out.println(rezultat);
      }
    });
    d3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String rezultat : menadzer.getResults())
          System.out.println(rezultat);
      }
    });
    setLayout(new FlowLayout());
    add(d1);
    add(d2);
```

```
    add(d3);
  }
  public static void main(String[] args) {
    run(new NadziranDugotrCallable(), 200, 500);
  }
} ///:~
```

Konstruktor klase **NadziranCallable** prima objekat tipa **ProgressMonitor** kao argument, a njegova metoda **call( )** ažurira **ProgressMonitor** svako pola sekunde. Imajte u vidu da je **NadziranCallable** zaseban zadatak i da ne bi trebalo da neposredno kontroliše ulazno/izlazne operacije, pa sam **monitor**u metodom **SwingUtilities.invokeLater( )** podneo informacije o promeni stepena napredovanja. U Sunovom udžbeniku za Swing („Swing Tutorial" na adresi *http://java.sun.com*) prikazan je drugačiji pristup; tamo se upotrebljava Swing **Timer**, koji proverava status zadatka i ažurira monitor.

Ako se na monitoru pritisne dugme cancel, metoda **monitor.isCanceled( )** vraća **true**. Ovde zadatak samo poziva metodu **interrupt( )** za sopstvenu nit, što će ga dovesti u odredbu **catch** gde se **monitor** gasi metodom **close( )**.

Ostatak koda je uglavnom isti kao pre, sem što se pravljenje objekta tipa **ProgressMonitor** odvija unutar konstruktora zadatka **NadziranDugotrCallable**.

Vežba 33: (6) Izmenite **DugotrCallableKojiSeMozePrekinuti.java** tako da sve zadatke izvršava paralelno, a ne sekvencijalno.

Vizuelno višenitno programiranje

U narednom primeru napravićemo **Runnable** objekat tipa **JPanel** (pano) koji sebe boji raznim bojama. Aplikacija je podešena tako da s komandne linije prima parametre koji određuju veličinu tabele boja i – metodom **sleep( )** – trajanje spavanja između promena boja. Poigrajte se s tim parametrima i mogli biste otkriti zanimljiva i možda neobjašnjiva obeležja višenitnog izvršavanja na vašoj platformi:

```
//: gui/ObojeneKutije.java
// Vizuelni prikaz višenitnog izvršavanja.
// {Args: 12 50}
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingKonzola.*;

class ObKut extends JPanel implements Runnable {
  private int pauza;
  private static Random rand = new Random();
  private Color boja = new Color(0);
  public void paintComponent(Graphics g) {
    g.setColor(boja);
    Dimension velicina = getSize();
```

```
    g.fillRect(0, 0, velicina.sirina, velicina.visina);
  }
  public ObKut(int pauza) { this.pauza = pauza; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        boja = new Color(rand.nextInt(0xFFFFFF));
// Asinhron poziv metode paint(), tj. zahtev da se slika nanovo iscrta:
        repaint();
        TimeUnit.MILLISECONDS.sleep(pauza);
      }
    } catch(InterruptedException e) {
      // Prihvatljiv način izlaska
    }
  }
}

public class ObojeneKutije extends JFrame {
  private int brojcelija = 12;
  private int pauza = 50;
  private static ExecutorService exec =
    Executors.newCachedThreadPool();
  public ObojeneKutije() {
    setLayout(new GridLayout(brojcelija, brojcelija));
    for(int i = 0; i < brojcelija * brojcelija; i++) {
      ObKut ok = new ObKut(pauza);
      add(ok);
      exec.execute(ok);
    }
  }
  public static void main(String[] args) {
    ObojeneKutije kutije = new ObojeneKutije();
    if(args.length > 0)
      kutije.brojcelija = new Integer(args[0]);
    if(args.length > 1)
      kutije.pauza = new Integer(args[1]);
    run(kutije, 500, 400);
  }
} ///:~
```

ObojeneKutije konfiguriše objekat tipa **GridLayout** tako da u svakoj dimenziji ima **brojcelija**. Zatim dodaje odgovarajući broj **ObKut** objekata da popuni taj broj ćelija i svakom prosleđuje broj **pauza**. U metodi **main( )** vidite da **pauza** i **brojcelija** imaju podrazumevane vrednosti koje se mogu menjati prosleđivanjem argumenata na komandnoj liniji.

Sve se radi u klasi **ObKut**. Ona je izvedena iz klase **JPanel** i realizuje interfejs **Runnable**, tako da svaki objekat tipa **JPanel** može biti nezavisan zadatak. Tim zadacima upravlja grupa niti **ExecutorService**.

Tekuća boja ćelije je **boja**. Boje pravi konstruktor klase **Color** čiji ulazni argument treba da bude 24-bitni broj koji u ovom slučaju pravimo nasumično.

Metoda **paintComponent( )** sasvim je jednostavna; ona samo menja boju kako joj kaže **boja** i ceo pano (objekat tipa **JPanel**) popunjava njom.

U metodi **run( )** postoji beskonačna petlja koja objektu **boja** zadaje novu nasumično napravljenu boju i zatim poziva **repaint( )** da bi je pokazala. Potom ta nit – primenom metode **sleep( )** – odlazi na spavanje, za vreme zadato na komandnoj liniji.

Poziv metode **repaint( )** unutar metode **run( )** zaslužuje pažnju. Na prvi pogled izgleda da pravimo mnogo niti i da se sve moraju prebrojiti. Možda vam izgleda da time kršimo princip na osnovu koga sve zadatke treba poslati raspoređivaču, odnosno redu za čekanje. Međutim, te niti zapravo ne modifikuju deljeni resurs. Kada one pozovu metodu **repaint( )**, ona ne preduzima prebojavanje nego samo postavlja određeni indikator „zamrljano", kojim pokazuje da je ta oblast kandidat za prebojavanje kada raspoređivač sledeći put bude spreman za tu akciju. Zato program ne prouzrokuje probleme sa Swingovim nitima.

Kada nit raspoređivača događaja pozove metodu **paint( )**, ona prvo pozove **paintComponent( )**, zatim **paintBorder( )** i onda **paintChildren( )**. Ako u izvedenoj komponenti želite da redefinišete metodu **paint( )**, ne zaboravite da pozovete verziju **paint( )** iz osnovne klase, tako da se uradi šta treba.

Upravo zato što je ovaj dizajn prilagodljiv i niti su vezane za svaki element panoa, možete da eksperimentišete i pravite proizvoljan broj niti. (U praksi, ograničenje nameće broj niti s kojima vaša JVM može da izađe na kraj.)

Ovaj program omogućuje zanimljivo poređenje performansi, pošto može da pokaže velike razlike u ponašanju i performansama raznih realizacija JVM niti i raznih platformi.

Vežba 34: (4) Izmenite program **ObojeneKutije.java** tako da prvo nasumično prska tačke („zvezde") po platnu, a zatim nasumično menja boje tih „zvezda".

Vizuelno programiranje i zrna Jave

Dosad ste u ovoj knjizi videli koliko je Java korisna za pisanje višekratno upotrebljivog koda. Jedinica koda koja se najčešće može ponovo iskoristiti jeste klasa jer se sastoji od karakteristika (polja) i ponašanja (metoda) koje se mogu ponovo koristiti – kompozicijom ili nasleđivanjem.

Nasleđivanje i polimorfizam su osnovni principi objektno orijentisanog programiranja, ali kada pišete aplikaciju, najčešće se trudite da komponente rade tačno ono što vi hoćete. Želeli biste da ubacite te delove u svoj program kao što elektroinženjer postavlja čipove na ploču. Takođe, izgleda da bi trebalo da postoji neki način za ubrzavanje ovakvog stila programiranja koji bi se mogao nazvati „sklapanje od delova".

„Vizuelno programiranje" je doživelo svoj prvi uspeh, i to *reliki*, sa Microsoftovim jezikom Visual BASIC (VB). Nakon njega se pojavila druga generacija čiji je najistaknutiji predstavnik Borlandov Delphi; on je bio i najveća inspiracija za zrna Jave. U tim programerskim alatkama komponente su predstavljene vizuelno, što ima smisla pošto one obično prikazuju neki grafički element, kao što su dugme ili polje za tekst. Zapravo, takva

vizuelna predstava je često izgled komponente u programu koji se izvršava. Zbog toga deo postupka vizuelnog programiranja podrazumeva prevlačenje komponente s palete i spuštanje na obrazac. Alatka za pravljenje aplikacija piše kôd dok vi prevlačite komponente, a taj kôd prouzrokuje pravljenje komponente u izvršnom programu.

Samo prevlačenje komponente u obrazac obično nije dovoljno za kompletiranje programa. Često morate da promenite neka njena obeležja, npr. boju, tekst koji sadrži, bazu podataka s kojom je povezana itd. Obeležja koja se mogu promeniti tokom projektovanja označavaju se kao *svojstva* (engl. *properties*). Sa svojstvima komponente možete da radite pomoću alatke za pravljenje aplikacija, a kada napravite program, svojstva komponente će biti snimljena da bi se mogla rekonstruisati kada se program pokrene.

Dosad ste se sigurno navikli na ideju da objekat nema samo svojstva, već i ponašanje. Tokom projektovanja, ponašanje vizuelne komponente je delimično predstavljeno *događajima*. Događaj znači: „Evo nečega što bi se moglo dogoditi komponenti". Obično odlučujete šta treba da se desi nakon događaja tako što povezujete kôd sa određenim događajem.

Evo izuzetno važnog dela: alatka za pravljenje aplikacija koristi refleksiju za dinamičko ispitivanje komponente i otkrivanje svojstava i događaja koja komponenta podržava. Kada to sazna, može da prikaže svojstva i omogući da ih menjate (uz snimanje stanja), ali može i da prikaže događaje. Po pravilu, vi dvaput pritiskate mišem, a alatka za pravljenje aplikacija pravi kôd i povezuje ga s tim događajem. U tom trenutku treba samo da napišete kôd koji se izvršava kada se desi događaj.

Dakle, alatka za pravljenje aplikacija obavlja najveći deo posla umesto vas. Zbog toga se možete usredsrediti na izgled programa i na to šta on treba da radi, a da se oslonite da će ta alatka obaviti kompletno povezivanje. Alatke za vizuelno programiranje izuzetno su popularne zato što primetno ubrzavaju pravljenje aplikacije. To posebno važi za korisnička okruženja, ali često i za druge delove aplikacije.

Šta je zrno?

Kada se prašina slegne, vidi se da je komponenta samo blok koda koji je obično predstavljen u obliku klase. Ključno pitanje je mogućnost alatke za pravljenje aplikacija da otkriva svojstva i događaje te komponente. Da bi napravio VB komponentu, programer je morao da piše prilično složen kôd u kome je poštovao određena pravila za otkrivanje svojstava i događaja. Delphi je bila alatka za vizuelno programiranje druge generacije. Taj jezik je aktivno koristio vizuelno programiranje pa se vizuelna komponenta mnogo lakše pravila. Međutim, izradu vizuelnih komponenata na najviši nivo dovela je Java, i to pomoću zrna (engl. *Java Beans*), pošto je zrno samo klasa. Ne morate pisati nikakav dodatni kôd niti koristiti specijalna proširenja jezika da bi nešto postalo zrno. Zapravo, treba samo malo da promenite način imenovanja metoda. Na osnovu imena metode, alatka za pravljenje aplikacija zna da li se radi o svojstvu, događaju ili običnoj metodi.

U dokumentaciji za Javu ova pravila imenovanja pogrešno su označena kao „projektni obrazac". To je nesporazum, pošto su projektni obrasci dovoljan izazov i bez ove zbrke (naći ćete ih u knjizi *Thinking in Patterns with Java* koju možete preuzeti s lokacije

www.MindView.net). Zrna Jave nisu opisana projektnim obrascem, već samo pravilima za imenovanje, i to prilično jednostavnim:

1. Da bi se neko svojstvo nazvalo **xxx**, obično treba napraviti dve metode: **getXxx( )** i **setXxx( )**. Obratite pažnju na to da se prvo slovo iza get ili set automatski pretvara u malo da bi se dobilo ime svojstva. Tip koji vraća metoda get isti je kao tip argumenta metode set. Ime svojstva i tipovi koje vraćaju metode get i set ne moraju da budu povezani.
2. Za svojstvo tipa **boolean** možete da koristite prethodni pristup s metodama get i set, ali umesto get možete da koristite is.
3. Obične metode zrna ne poštuju prethodna pravila za dodeljivanje imena, ali su javne.
4. Za događaje se koristi pristup „prijemnika" iz biblioteke Swing. To je isto ono što ste već videli: metodama **addBounceListener(BounceListener)** i **removeBounceListener(BounceListener)** obrađuje se događaj **BounceEvent**. Najčešće će ugrađeni događaji i prijemnici zadovoljavati vaše potrebe, ali možete da napravite i sopstvene događaje i prijemničke interfejse.

Ova saznanja možemo da iskoristimo za pravljenje jednostavnog zrna:

```
//: frogbean:Frog.java
// Obično zrno Jave.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
  private int jumps;
  private Color color;
  private Spots spots;
  private boolean jmpr;
  public int getJumps() { return jumps; }
  public void setJumps(int newJumps) {
    jumps = newJumps;
  }
  public Color getColor() { return color; }
  public void setColor(Color newColor) {
    color = newColor;
  }
  public Spots getSpots() { return spots; }
  public void setSpots(Spots newSpots) {
    spots = newSpots;
  }
  public boolean isJumper() { return jmpr; }
  public void setJumper(boolean j) { jmpr = j; }
```

```
  public void addActionListener(ActionListener l) {
    //...
  }
  public void removeActionListener(ActionListener l) {
    // ...
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // "Obična" javna metoda:
  public void croak() {
    System.out.println("Ribbet!");
  }
} ///:~
```

Prvo uočavate da je ovo obična klasa. Sva njena polja obično će biti privatna i moći će da im se pristupa samo preko metoda. U skladu s pravilom za imenovanje, svojstva su **jumps**, **color**, **spots** i **jumper** (obratite pažnju na promenu veličine prvog slova imena svojstva). Iako je ime lokalnog identifikatora isto kao ime svojstva u prva tri slučaja, svojstvo **jumper** pokazuje da ime svojstva nije uslov za korišćenje tačno određenog identifikatora za lokalne promenljive. (Lokalne promenljive za to svojstvo čak ne moraju ni da postoje.)

Događaji koje generiše ovo zrno su **ActionEvent** i **KeyEvent**, što zaključujemo po imenima metoda add i remove za rad s prijemnicima. Konačno, vidite da je obična metoda **croak( )** i dalje deo zrna samo zato što je javna, a ne zato što poštuje pravilo dodeljivanja imena.

Ispitivanje zrna klasom **Introspector**

Jedna od najosetljivijih strana zrna uočava se kada zrno prevlačite s palete i spuštate na obrazac. Alatka za pravljenje aplikacije mora biti u stanju da napravi zrno (što može da uradi ako postoji podrazumevani konstruktor) i da zatim, bez pristupanja njegovom izvornom kodu, izdvoji sve neophodne informacije kako bi napravila spisak svojstava i blokove za obradu događaja.

Deo rešenja ćete naći na kraju poglavlja *Podaci o tipu*: Javina *refleksija* omogućuje da se otkriju sve metode bilo koje klase. To je savršeno za rešavanje problema sa zrnima bez korišćenja dodatnih rezervisanih reči jezika, što je potrebno u drugim jezicima za vizuelno programiranje. Jedan od najvažnijih razloga za dodavanje refleksije u Javu i jeste bilo održavanje zrna (mada refleksija omogućuje i serijalizaciju objekata i daljinsko pozivanje metoda). Dakle, može se očekivati da autor alatke za pravljenje aplikacija mora ispitivati svako zrno i pregledati njegove metode, da bi saznao koja su njegova svojstva i metode.

To je izvesno moguće, ali su projektanti Jave želeli da obezbede standardnu alatku, ne samo da bi se zrna lakše koristila, već i da bi se obezbedio standardizovan način za pravljenje složenijih zrna. Ta alatka je klasa **Introspector**, a njena najvažnija metoda je **static getBeanInfo( )**. Ovoj metodi se prosleđuje referenca na objekat klase **Class**, a ona iscrpno ispituje klasu i vraća objekat tipa **BeanInfo** koji potom možete analizirati da biste saznali svojstva, metode i događaje zrna.

Obično vam ništa od toga neće biti važno; najveći broj zrna koristićete gotova i nema potrebe da znate detalje koji se dešavaju unutar njih. Samo ćete prevlačiti zrna na obrazac, zatim podešavati njihova svojstva i pisati procedure za događaje koji vas zanimaju. Međutim, vežba korišćenja klase **Introspector** za prikazivanje informacija o zrnu zanimljiva je i poučna, pa evo alatke koja to radi:

```
//: gui/IspitivanjeZrna.java
// Ispitivanje Zrna klasom Introspector.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingKonzola.*;

public class IspitivanjeZrna extends JFrame {
  JTextField upit = new JTextField(20);
  JTextArea rezultati = new JTextArea();
  public void print(String s) { rezultati.append(s + "\n"); }
  public void ispitaj(Class<?>zrno){
    rezultati.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(zrno, Object.class);
    } catch(IntrospectionException e) {
      print("Ne mogu da ispitam " + zrno.getName());
      return;
    }
    for(PropertyDescriptor  d:  bi.getPropertyDescriptors()){
      Class<?> p = d.getPropertyType();
      if(p == null) continue;
      print("Tip svojstva:\n  " + p.getName() +
        "Ime svojstva:\n  " + d.getName());
      Method metodaCitanja = d.getReadMethod();
      if(metodaCitanja != null)
        print("Metoda za čitanje:\n  " + metodaCitanja);
      Method metodaUpisivanja = d.getWriteMethod();
      if(metodaUpisivanja != null)
        print("Metoda za upisivanje:\n  " + metodaUpisivanja);
      print("====================");
    }
```

```
      print("Javne metode:");
      for(MethodDescriptor m : bi.getMethodDescriptors())
        print(m.getMethod().toString());
      print("======================");
      print("Podrška za događaje:");
      for(EventSetDescriptor e : bi.getEventSetDescriptors()){
        print("Tip prijemnika:\n  " + e.getListenerType().getName());
        for(Method lm : e.getListenerMethods())
          print("Metoda prijemnika:\n  " + lm.getName());
        for(MethodDescriptor lmd : e.getListenerMethodDescriptors())
          print("Opis metode:\n  " + lmd.getMethod());
        Method dodavanje = e.getAddListenerMethod();
        print("Metoda za dodavanje:\n  " + dodavanje);
        Method uklanjanje = e.getRemoveListenerMethod();
        print("Metoda za uklanjanje:\n  " + uklanjanje);
        print("====================");
      }
    }
    class Ispitivac implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        String ime = upit.getText();
        Class<?>c = null;
        try {
          c = Class.forName(ime);
        } catch(ClassNotFoundException ex) {
          rezultati.setText("Ne mogu da pronađem" + ime);
          return;
        }
        ispitaj(c);
      }
    }
    public IspitivanjeZrna() {
      JPanel p = new JPanel();
      p.setLayout(new FlowLayout());
      p.add(new JLabel("Puno ime zrna:"));
      p.add(upit);
      cp.add(BorderLayout.NORTH, p);
      cp.add(new JScrollPane(rezultati));
      Ispitivac isp = new Ispitivac();
      upit.addActionListener(isp);
      upit.setText("frogbean.Frog");
      // Prinudna provera
      isp.actionPerformed(
        new ActionEvent(isp, 0, ""));
    }
    public static void main(String[] args) {
      run(new IspitivanjeZrna(), 600, 500);
    }
  } ///:~
```

Sav posao obavlja metoda **IspitivanjeZrna.ispitaj( )**. Ona prvo pokušava da napravi objekat interfejsa **BeanInfo**, a ako uspe, poziva metode koje vraćaju informacije o svojstvima zrna, metodama i događajima. U metodi **Introspector.getBeanInfo( )** primetićete drugi argument. On kaže objektu klase gde da se zaustavi u hijerarhiji nasleđivanja. Ovde se zaustavlja pre nego što analizira metode klase **Object**, pošto nas one ne interesuju.

Metoda **getPropertyDescriptors( )** vraća niz objekata tipa **PropertyDescriptor** koji opisuju svojstva. Za svaki deskriptor svojstava (objekat tipa **PropertyDescriptor**) možete da pozovete metodu **getPropertyType( )** da biste otkrili klasu objekta koji se prosleđuje unutra i napolje pomoću metoda svojstava. Zatim, za svako svojstvo možete da dobijete pseudonim (generisan na osnovu imena metoda) pomoću metode **getName( )**, metodu za čitanje pomoću **getReadMethod( )** i metodu za upisivanje pomoću **getWriteMethod( )**. Dve poslednje metode vraćaju objekat tipa **Method** koji se može koristiti za pozivanje odgovarajuće metode objekta (to je deo refleksije).

Što se tiče javnih metoda (uključujući i metode svojstava), **getMethodDescriptors( )** vraća niz objekata klase **MethodDescriptor**. Možete da dobijete objekat tipa **Method** povezan sa svakim od tih objekata i da ispišete njegovo ime.

Što se tiče događaja, metoda **getEventSetDescriptors( )** vraća niz (čega drugog nego) objekata tipa **EventSetDescriptor**. Svaki od njih se može ispitati da bi se otkrila klasa prijemnika, metode prijemničke klase, kao i metode za dodavanje i uklanjanje prijemnika. Program **IspitivanjeZrna** ispisuje sve ove informacije.

Pre izvršavanja, program proverava zrno **frogbean.Frog**. Nakon uklanjanja dodatnih, nepotrebnih detalja, rezultat izgleda ovako:

```
Tip svojstva:
  Color
Ime svojstva:
  color
Metoda za čitanje:
  public Color getColor()
Metoda za upisivanje:
  public void setColor(Color)
= = = = = = = = = = = = = = = = = = = = =
Tip svojstva:
  boolean
Ime svojstva:
  jumper
Metoda za čitanje:
  public boolean isJumper()
Metoda za upisivanje:
  public void setJumper(boolean)
= = = = = = = = = = = = = = = = = = = = =
Tip svojstva:
  int
Ime svojstva:
  jumps
Metoda za čitanje:
  public int getJumps()
```

```
Metoda za upisivanje:
  public void setJumps(int)
= = = = = = = = = = = = = = = = = = = = = = =
Tip svojstva:
  frogbean.Spots
Ime svojstva:
  spots
Metoda za čitanje:
  public Spots getSpots()
Metoda za upisivanje:
  public void setSpots(Spots)
= = = = = = = = = = = = = = = = = = = = = = =
Javne metode:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)
= = = = = = = = = = = = = = = = = = = = = = =
Podrška za događaje:
Tip prijemnika:
  KeyListener
Metoda prijemnika:
  keyTyped
Metoda prijemnika:
  keyPressed
Metoda prijemnika:
  keyReleased
Opis metode:
  public void keyTyped(KeyEvent)
Opis metode:
  public void keyPressed(KeyEvent)
Opis metode:
  public void keyReleased(KeyEvent)
Metoda za dodavanje:
  public void addKeyListener(KeyListener)
Metoda za uklanjanje:
  public void removeKeyListener(KeyListener)
= = = = = = = = = = = = = = = = = = = = = = =
Tip prijemnika:
  ActionListener
```

```
Metoda prijemnika:
  actionPerformed
Opis metode:
  public void actionPerformed(ActionEvent)
Metoda za dodavanje:
  public void addActionListener(ActionListener)
Metoda za uklanjanje:
  public void removeActionListener(ActionListener)
= = = = = = = = = = = = = = = = = = = = =
```

Ovaj program otkriva najveći deo svojstava koje vidi klasa **Introspector** dok na osnovu zrna pravi objekat tipa **BeanInfo**. Primetićete da su tip svojstva i njegovo ime nezavisni. Obratite pažnju na to da su sva slova u imenu svojstva mala. (Jedino se odstupa kada ime svojstva počinje s nekoliko uzastopnih velikih slova.) Zapamtite i to da se imena metoda kakva ovde vidite (npr. metode za čitanje i upisivanje) zapravo dobijaju iz objekta tipa **Method** koji se može koristiti za pozivanje odgovarajuće metode.

Spisak javnih metoda sadrži metode koje nisu povezane s nekim svojstvom ili događajem, kakva je metoda **croak**( ), ali i one koje jesu povezane sa svojstvima i događajima. To su sve metode zrna koje možete da pozovete iz programa, a alatka za pravljenje aplikacija može ih sve prikazati dok pišete program da bi vam olakšala posao.

Konačno, primećuje se da su događaji potpuno analizirani i razdvojeni na interfejs prijemnika, njegove metode i metode za dodavanje i uklanjanje prijemnika. U osnovi, kada imate objekat tipa **BeanInfo**, možete da saznate sve što je važno o zrnu. Možete i da pozivate metode zrna, čak i ako nemate nikakve druge informacije osim ovog objekta (to je još jedna osobina refleksije).

Naprednije zrno

Sledeći primer je samo nešto malo napredniji, mada je frivolan. To je objekat klase **JPanel** koji iscrtava mali krug oko miša kad god se on pomeri. Kada pritisnete taster miša, reč Bang! se pojavljuje u sredini ekrana i pokreće se prijemnik događaja.

Možete da promenite sledeća svojstva: veličinu kruga, boju, veličinu i reč koja se prikazuje kada se pritisne taster miša. Objekat **BangBean** takođe ima sopstvene metode **addActionListener**( ) i **removeActionListener**( ) pomoću kojih možete da povežete zrno sa svojim prijemnikom koji se pokreće kada korisnik pritisne mišem. Sada bi već trebalo da umete prepoznati podršku za svojstva i događaje:

```
//: bangbean:BangBean.java
// Grafičko zrno.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
```

```
public class BangBean extends JPanel implements Serializable {
  protected int xm, ym;
  protected int cSize = 20; // Velicina kruga
  protected String text = "Bang!";
  protected int fontSize = 48;
  protected Color tColor = Color.RED;
  protected ActionListener actionListener;
  public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  public int getCircleSize() { return cSize; }
  public void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public String getBangText() { return text; }
  public void setBangText(String newText) {
    text = newText;
  }
  public int getFontSize() { return fontSize; }
  public void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public Color getTextColor() { return tColor; }
  public void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
  }
  // Zrno dozvoljava samo jedan prijemnik, a to je
  // najjednostavniji oblik upravljanja prijemnicima:
  public void addActionListener (ActionListener l)
        throws TooManyListenersException {
    if(actionListener != null)
      throw new TooManyListenersException();
    actionListener = l;
  }
  public void removeActionListener(ActionListener l) {
    actionListener = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font("TimesRoman", Font.BOLD, fontSize));
      int width = g.getFontMetrics().stringWidth(text);
```

```
        g.drawString(text, (getSize().width - width) /2,
          getSize().height/2);
        g.dispose();
        // Poziva prijemničku metodu:
        if(actionListener != null)
          actionListener.actionPerformed(
            new ActionEvent(BangBean.this,
              ActionEvent.ACTION_PERFORMED, null));
      }
    }
    class MML extends MouseMotionAdapter {
      public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
      }
    }
    public Dimension getPreferredSize() {
      return new Dimension(200, 200);
    }
} ///:~
```

Prvo ćete primetiti da klasa **BangBean** realizuje interfejs **Serializable**. To znači da alatka za pravljenje aplikacija može da učita sve informacije o zrnu pomoću serijalizacije, nakon što projektant programa podesi vrednosti svojstava. Kada se zrno napravi kao deo izvršne aplikacije, snimljena svojstva se rekonstruišu pa se dobija tačno ono što ste projektovali.

U potpisu metode **addActionListener( )** vidite da ona može da generiše izuzetak **TooManyListenersException**. To znači da je događaj *jednosmeran*, tj. da se samo jedan prijemnik obaveštava o tome da se desio. Obično ćete koristiti *višesmerne* događaje tako da o događaju bude obavešteno više prijemnika. Međutim, ovde već ulazimo u teme koje će biti razmotrene u odeljku „Zrna Jave i sinhronizacija". Jednosmernim događajem zasad ćemo zaobići ovaj problem.

Kada pritisnete taster miša, u zrnu se ispisuje tekst, a ako polje **actionListener** nije **null**, pravi se nov objekat tipa **ActionEvent** i šalje se prijemniku. Kad god se pomeri miš, pamte se njegove nove koordinate i pozadina menja boju (uz brisanje teksta koji se nalazi na pozadini, kao što ćete videti).

Evo klase **BangBeanTest** koja omogućuje testiranje zrna kao apleta ili aplikacije:

```
//: gui/BangBeanTest.java
// {Timeout: 5} U testiranju, ugasi nakon 5 sekundi
package bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingKonzola.*;
```

```
public class BangBeanTest extends JFrame {
  private JTextField txt = new JTextField(20);
  // Tokom testiranja prijavljuju se akcije:
  class BBL implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("Akcija zrna"+ count++);
    }
  }
  public BangBeanTest() {
    BangBean bb = new BangBean();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Previše prijemnika");
    }
    add(bb);
    add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    run(new BangBeanTest(), 400, 500);
  }
} ///:~
```

Ova klasa se neće koristiti kada se zrno nalazi u razvojnom okruženju, ali je korisna kao način za brzo testiranje svih zrna. **BangBeanTest** stavlja zrno **BangBean** u aplet, povezujući s njim jednostavan prijemnik tipa **ActionListener** koji ispisuje broj događaja u tekstualnom polju kad god se desi događaj. Alatka za izradu aplikacija obično pravi najveći deo koda koji koristi zrno.

Kada ispitate ovo zrno klasom **IspitivanjeZrna** ili ga stavite u razvojno okruženje koje podržava zrna, primetićete da ima mnogo više svojstava i akcija nego što se vidi u gornjem kodu. Razlog je to što je klasa **BangBean** izvedena iz klase **JPanel**, a **JPanel** je takođe zrno, pa se prikazuju i njegova svojstva i događaji.

Vežba 35: (6) Pronađite na Internetu i preuzmite jednu ili više besplatnih vizuelnih alatki za pravljenje grafičkih korisničkih okruženja, ili kupite neku. Saznajte šta je neophodno da bi se zrno **BangBean** unelo i koristilo u tom okruženju.

Zrna Jave i sinhronizacija

Kad god napravite zrno, treba da očekujete da će ono raditi u višenitnom okruženju. To znači sledeće:

1. Kad god je moguće, sve javne metode jednog zrna treba sinhronizovati. Naravno, time izazivate povećane procesorske troškove koje povlači atribut **synchronized**. Ako su oni neprihvatljivi, metode koje neće izazvati probleme u kritičnim delovima programa mogu da ostanu nesinhronizovane, ali imajte u vidu da to nije uvek lako proceniti. Obično su kandidati za to male metode (poput metode **getCircleSize()** u sledećem primeru) i (ili) „atomične", pa se metoda izvršava u tako kratkom vremenu da objekat ne može da bude izmenjen tokom izvršavanja. Ako takve metode

ostanu nesinhronizovane, izvršavanje celog programa možda se neće značajno ubrzati. Stoga biste sve javne metode zrna mogli da sinhronizujete, a rezervisanu reč **synchronized** da uklonite samo ako znate da je to potrebno i da ćete time značajno ubrzati izvršavanje programa.

2. Prilikom obaveštavanja više prijemnika (engl. *multicast*) o nekom događaju, morate pretpostaviti da tokom kretanja kroz taj spisak neki prijemnici mogu biti dodati, a drugi uklonjeni.

Prvu tačku je lako uraditi, ali pri izvođenju druge treba malo razmisliti. Uzmimo primer zrna **BangBean.java** predstavljenog u prošlom poglavlju. Tamo je izbegnuto pitanje korišćenja više niti tako što je zanemarena rezervisana reč **synchronized** i obavešten samo jedan prijemnik (engl. *unicast*). Evo tog primera, izmenjenog tako da koristi više niti i da o događajima obaveštava više prijemnika:

```
//: gui/BangBean2.java
// Trebalo bi da svoja zrna ovako pišete,
// kako bi mogla da se izvršavaju u okruženju s više niti.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import net.mindview.util.SwingKonzola.*;

public class BangBean2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // veličina kružnice
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.RED;
  private ArrayList actionListeners =
    new ArrayList<ActionListener>();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() { return cSize; }
  public synchronized void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() { return text; }
  public synchronized void setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize() { return fontSize; }
  public synchronized void setFontSize(int newSize) {
    fontSize = newSize;
```

```
}
public synchronized Color getTextColor() { return tColor;}
public synchronized void setTextColor(Color newColor) {
  tColor = newColor;
}
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  g.setColor(Color.BLACK);
  g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
}
// Ovo zrno ima više prijemnika što se češće koristi
// od obaveštavanja samo jednog prijemnika,
// kako je bilo urađeno u programu BangBean.java:
public synchronized void
addActionListener(ActionListener l) {
  actionListeners.add(l);
}
public synchronized void
removeActionListener(ActionListener l) {
  actionListeners.remove(l);
}
// Pazite, ovo nije sinhronizovano:
public void notifyListeners() {
  ActionEvent a = new ActionEvent(BangBean2.this,
    ActionEvent.ACTION_PERFORMED, null);
  ArrayList lv = null;
  // Napravi površnu kopiju spiska za slučaj
  // da neko doda jedan prijemnik tokom
  // pozivanja drugih prijemnika:
  synchronized(this) {
    lv = new ArrayList<ActionListener>(actionListeners);
  }
  // Obavesti sve metode prijemnike:
  for(ActionListener al : lv)
    al.actionPerformed(a);
}
class ML extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    Graphics g = getGraphics();
    g.setColor(tColor);
    g.setFont(
      new Font("TimesRoman", Font.BOLD, fontSize));
    int width = g.getFontMetrics().stringWidth(text);
    g.drawString(text, (getSize().width - width) /2,
      getSize().height/2);
    g.dispose();
    notifyListeners();
  }
```

```
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BangBean2 bb2 = new BangBean2();
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("ActionEvent" + e);
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("BangBean2 action");
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("More action");
      }
    });
    JFrame frame = new JFrame();
    frame.add(bb2);
    run(bb, 300, 300);
  }
} ///:~
```

Dodavanje rezervisane reči **synchronized** metodama jeste mala i laka izmena. Međutim, uočite da se u metodama **addActionListener**( ) i **removeActionListener**( ) prijemnici sada dodaju na listu i uklanjaju s nje kako bi zrno moglo da ima više prijemnika.

Vidi se da metoda **notifyListeners**( ) *nije* sinhronizovana. Nju može istovremeno da pozove više niti. Moguće je i da metode **addActionListener**( ) i **removeActionListener**( ) budu pozvane usred poziva metode **notifyListeners**( ), što predstavlja problem jer ta metoda prolazi kroz listu prijemnika. Da bi se taj problem ublažio, spisak je unutar sinhronizovanog dela koda kloniran pomoću konstruktora spiska prijemnika **ArrayList** koji kopira elemente svog argumenta, pa se prolazi kroz klon. Tako se može raditi sa originalnim spiskom prijemnika, a da se pri tom ne utiče na metodu **notifyListeners**( ).

Ni metoda **paintComponent**( ) nije sinhronizovana. Odlučiti da li treba sinhronizovati redefinisane metode nije tako lako kao kad se samo dodaju sopstvene metode. U ovom primeru ispostavlja se da metoda **paintComponent**( ) naizgled dobro radi, bila sinhronizovana ili ne. Ali morate uzeti u obzir i sledeća pitanja:

1. Da li ta metoda menja stanje kritičnih promenljivih unutar objekta? Da biste otkrili koje su promenljive kritične, morate utvrditi da li će ih druge niti u programu čitati ili im zadavati vrednosti. (U ovom slučaju, čitanje i zadavanje gotovo se uvek obavlja unutar sinhronizovanih metoda, pa možete da ispitate samo njih.) Metoda **paintComponent()** nema takvih izmena stanja.
2. Da li metoda zavisi od stanja tih kritičnih promenljivih? Ako neka sinhronizovana metoda modifikuje promenljivu koju vaša metoda koristi, trebalo bi da i nju sinhronizujete. Na osnovu toga mogli biste uočiti da promenljivu **cSize** modifikuju sinhronizovane metode, pa bi stoga metodu **paintComponent( )** trebalo sinhronizovati. U ovom slučaju, međutim, možete da se zapitate: „Šta se najgore može dogoditi ako se sadržaj promenljive **cSize** promeni tokom izvršavanja metode **paintComponent( )**?" Ako zaključite da to što se može desiti nije tako strašno, a uz to je i prolazno, možete da ostavite metodu **paintComponent( )** nesinhronizovanu kako biste izbegli dodatne režijske troškove koje izaziva pozivanje sinhronizovane metode.
3. Da li je verzija metode u osnovnoj klasi sinhronizovana? Metoda **paintComponent( )** nije sinhronizovana u osnovnoj klasi. To nije baš besprekoran argument nego samo povod za razmišljanje. U ovom slučaju, recimo, *postoji* jedno polje (**cSize**) koje modifikuje sinhronizovana metoda, a njemu pristupa metoda **paintComponent( )**, pa je to moglo da izmeni situaciju. Imajte u vidu, međutim, da se atribut **synchronized** ne nasleđuje – znači, ukoliko je neka metoda sinhronizovana u osnovnoj klasi, ona *ne postaje* automatski sinhronizovana u redefinisanoj verziji izvedene klase.
4. Metode **paint( )** i **paintComponent( )** moraju biti najbrže moguće. Izuzetno se preporučuje korišćenje svega što iz njih uklanja režijske troškove, pa ako smatrate da te metode treba da sinhronizujete, to može biti znak lošeg projekta.

Kod za testiranje unutar metode **main( )** drugačiji je nego onaj u programu **BangBeanTest**, kako bi se pokazala sposobnost klase **BangBean2** da o događajima obaveštava više prijemnika.

Pakovanje zrna

Pre nego što se zrno uveze u vizuelnu alatku, mora se staviti u standardan paket za zrna, a to je JAR arhiva koja sadrži sve klase zrna i manifest datoteku. Manifest datoteka je tekstualna datoteka određenog oblika. Za zrno **BangBean**, ta datoteka izgleda ovako:

```
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
```

Prvi red prikazuje broj verzije manifest datoteke koji, do daljeg obaveštenja iz kompanije Sun, iznosi 1.0. Drugi red (prazni redovi se zanemaruju) navodi datoteku **BangBean.class**, a treći kaže „To je zrno". Bez trećeg reda, alatka za pravljenje programa ne bi prepoznavala klasu kao zrno.

Jedina nepovoljnost je to što morate da obezbedite pravilnu putanju u polju Name:. Ako ponovo pogledate program **BangBean.java**, videćete da se nalazi u paketu **bangbean** (pa stoga u poddirektorijumu **bangbean** koji nije u putanji klase), a ime manifest datoteke mora da sadrži informacije o paketu. Osim toga, manifest datoteku morate da smestite u direktorijum *iznad* korena putanje paketa, što u ovom slučaju znači da tu datoteku treba smestiti u direktorijum iznad poddirektorijuma bangbean. Zatim treba da pozovete program **jar** u istom direktorijumu u kome se nalazi manifest datoteka, kao ovde:

```
jar cmf BangBean.jar BangBean.mf bangbean
```

Ovo podrazumeva da želite da se dobijena JAR datoteka zove **BangBean.jar** i da se manifest datoteka zove **BangBean.mf**.

Možda se pitate šta se dešava sa svim drugim klasama koje se generišu prilikom prevođenja programa **BangBean.java**. One završavaju u poddirektorijumu **bangbean**, a primetićete da je poslednji argument na prethodnoj komandnoj liniji upravo taj poddirektorijum. Kada programu **jar** date ime nekog poddirektorijuma, on pakuje ceo taj poddirektorijum (uključujući, u ovom slučaju, i originalnu datoteku sa izvornim kodom programa **BangBean.java**; vi ćete možda odlučiti da izvorni kod ne pakujete zajedno sa zrnima). Pored toga, ako otpakujete JAR arhivu koja je upravo napravljena, otkrićete da se manifest datoteka ne nalazi u paketu, već da je program **jar** napravio sopstvenu datoteku koja se delimično zasniva na vašoj, a zove se **MANIFEST.MF** i nalazi se u poddirektorijumu **META-INF** (skraćenica za metainformacije). Kad otvorite tu datoteku, primetićete i da je alatka **jar** dodala informacije o digitalnom potpisu svake datoteke u sledećem obliku:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9Naecx8aFtqPI4udsx/OO=
MD5-Digest: O4NcS1hE3Smnzlp2Hhj6qeg==
```

O ovome uglavnom ne morate da vodite računa, a ako nešto menjate, možete samo da promenite prvobitnu manifest datoteku i da ponovo pozovete alatku **jar** da biste napravili novu arhivu. U JAR arhivu možete da dodate i nova zrna tako što ćete informacije o njima staviti u manifest datoteku.

Dobro je da se svako zrno stavi u sopstveni poddirektorijum, jer kada pravite arhivu možete da arhivirate ceo direktorijum. Vidi se da i **Frog** i **BangBean** imaju sopstvene poddirektorijume.

Kada pravilno smestite zrno u arhivu, možete da ga unesete u alatku za izradu programa koja podržava zrna. Način na koji se to radi razlikuje se od alatke do alatke, ali kompanija Sun obezbeđuje besplatno okruženje za testiranje Javinih zrna koje se zove **Bean Builder**. (Možete ga preuzeti s lokacije *http://java.sun.com/beans*.) Da biste smestili zrno u Bean Builder, kopirajte njegovu JAR datoteku u odgovarajući poddirektorijum.

Vežba 36: (4) Dodajte datoteku **Frog.class** u manifest datoteku kao što je prikazano u ovom poglavlju i pokrenite program **jar** da biste napravili JAR arhivu koja sadrži zrna **Frog** i **BangBean**. Zatim sa Interneta preuzmite i instalirajte Bean Builder ili upotrebite svoju alatku za pravljenje programa koja podržava zrna. Dodajte novu arhivu u to okruženje kako biste mogli da testirate ta dva zrna.

Složenija podrška za zrna

Vidite kako je jednostavno napraviti zrno. Međutim, niste ograničeni samo na ono što je ovde prikazano. Arhitektura Javinih zrna obezbeđuje lak početak, ali može da posluži i u složenijim situacijama koje prevazilaze okvire ove knjige i biće ukratko izložene. Više detalja saznaćete na lokaciji *java.sun.com*.

Svojstva zrna mogu da se poboljšaju. U prethodnim primerima prikazana su samo pojedinačna svojstva, ali je moguće predstaviti i više svojstava u obliku niza. To se zove *indeksirano svojstvo*. Vi obezbeđujete samo odgovarajuće metode (naravno, poštujući pravila za dodeljivanje imena metodama), a objekat klase **Introspector** prepoznaje indeksirano svojstvo tako da alatka za pravljenje aplikacija može da reaguje na odgovarajući način.

Svojstva mogu da budu *povezana* (engl. *bound*), što znači da obaveštavaju druge objekte o promenama pomoću događaja tipa **PropertyChangeEvent**. Drugi objekti potom mogu da odluče da li će se promeniti na osnovu promene zrna.

Svojstva mogu da budu *ograničena* (engl. *constrained*), što znači da drugi objekti mogu da zabrane promenu nekog svojstva ako je neprihvatljiva. Drugi objekti se obaveštavaju pomoću događaja tipa **PropertyChangeEvent**, a mogu da generišu izuzetak tipa **PropertyVetoException** kako bi sprečili promenu i vratili stare vrednosti.

Moguće je menjati i način predstavljanja zrna tokom projektovanja:

1. Možete da obezbedite namenski spisak svojstava za određeno zrno. Običan spisak svojstava koristiće se za sva ostala zrna, ali će se vaš spisak automatski pozivati kada se izabere vaše zrno.
2. Možete da napravite namensku klasu za prilagođavanje određenog svojstva (tako da se koristi običan spisak svojstava, ali kada se menja specijalno svojstvo, automatski će se pozivati vaša klasa).
3. Za zrno možete da obezbedite namensku klasu tipa **BeanInfo** koja daje informacije različite od onih koje pravi **Introspector**.
4. Možete uključivati i isključivati napredan režim u svim objektima klase **FeatureDescriptor** da bi se razlikovale osnovne i složenije mogućnosti.

Više o zrnima

Mnogi su pisali o zrnima Jave; na primer, Elliote Rusty Harold autor je knjige *JavaBeans* (IDG, 1998).

Alternative za Swing

Iako je biblioteka Swing GKO alatka koju preporučuje Sun, to nikako ne znači da ne postoje i druge alatke za pravljenje grafičkih korisničkih okruženja. Dve važne alternative su *Flash* kompanije *Adobe*, u kojem se za pravljenje klijentskih GKO preko Weba koristi Flashov sistem za programiranje *Flex*, i biblioteka otvorenog izvornog koda Eclipse *Standard Widget Toolkit* (SWT) za aplikacije namenjene stonim računarima.

Zašto biste koristili alternative? Za Web klijente se prilično uverljivo može tvrditi da apleti nisu uspeli. Iako su postojali od početka i mada se o njima toliko pričalo i obećavalo, danas je iznenađenje kada čovek naiđe na Web aplikaciju koja koristi aplete. Čak ni Sun ne koristi aplete svugde. Evo primera:

http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html

Iako je interaktivna mapa Javinih mogućnosti na Sunovoj lokaciji veoma dobar kandidat za Java aplet, oni su je napravili u Flashu. Izgleda da je to prećutno priznanje da apleti nisu uspeli. Još je važnije to što je Flash Player instaliran na preko 98 procenata računarskih platformi, pa se može smatrati prihvaćenim standardom. Kao što ćete videti, sistem Flex obezbeđuje veoma moćno okruženje za programiranje klijentskih aplikacija, svakako mnogo jače od JavaScripta, a njegov izgled i ponašanje često su bolji od apleta. Ukoliko ipak želite da koristite aplete, ostaje vam da ubedite klijenta da s Weba preuzme JRE koji je mnogo veći i duže se preuzima od Flash Playera.

Za stone aplikacije, jedan od problema sa Swingom je to što korisnici *opaze* da koriste drugu vrstu aplikacije, pošto se izgled i ponašanje Swing aplikacija razlikuje od onih uobičajenih na stonim računarima. Korisnici po pravilu ne žele da svaka aplikacija drugačije izgleda i ponaša se; oni se trude da što brže završe svoj posao i trude se da sve aplikacije izgledaju i ponašaju se jednako. SWT pravi aplikacije koje izgledaju kao matične, a pošto ta biblioteka koristi matične komponente koliko god je moguće, te aplikacije se obično izvršavaju brže od ekvivalentnih Swing aplikacija.

Pravljenje Flash Web klijenata pomoću Flexa

Pošto je laka virtuelna mašina Flash kompanije Adobe široko rasprostranjena, većina ljudi može da bez ikakve instalacije koristi okruženje zasnovano na Flashu, a ono će izgledati i ponašati se jednako na svim sistemima i platformama.[10]

Za razvoj Flash korisničkih okruženja za Java aplikacije, možete da upotrebite sistem *Flex* kompanije *Adobe*. Flex se sastoji od programskog modela čiju osnovu čine XML i jezik za pisanje skriptova, slični programskim modelima HTML i JavaScript, i robusne biblioteke komponenata. Za deklarisanje kontrolnih objekata i upravljanja rasporedom komponenata koristi se sintaksa MXML, a dinamički skriptovi se upotrebljavaju za dodavanje koda za obradu događaja i pozivanje usluga, koji korisničko okruženje povezuje s Java klasama, modelima podataka, Web uslugama itd. Flexov prevodilac prevodi MXML i skript datoteke u bajtkod. Flashova virtuelna mašina na klijentskom računaru radi kao Javina virtuelna mašina (JVM) utoliko što interpretira prevedeni bajtkod. Format Flashovog bajtkoda nazvan je SWF; SWF datoteke proizvodi Flexov prevodilac.

Vodite računa o tome da postoji alternativa Flexu na adresi *http://openlaszlo.org*; njen izvorni kôd je otvoren i strukture slične Flexu, te nekome može bolje poslužiti. Postoje i druge alatke za pravljenje Flash aplikacija na različite načine.

[10] Jezgro materijala u ovom odeljku napravio je Sean Neville.

Zdravo, Flex

Pročitajte ovaj MXML kôd, koji definiše korisničko okruženje (imajte u vidu da prvog i poslednjeg reda nema u kodu koji preuzimate u paketu izvornog koda ove knjige):

```
//:! gui/flex/zdravoflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"[11]
  backgroundColor="#ffffff">
  <mx:Natpis id="output" tekst="Zdravo, Flex!" />
</mx:Application>
///:~
```

MXML datoteke su XML dokumenti, te stoga počinju opisom XML verzije i kodiranja; to su atributi version i encoding. Krajnji spoljašnji MXML element je **Application** koji se vizuelno nalazi na vrhu i logički je kontejner za Flex korisničko okruženje. Unutar elementa **Application** možete deklarisati oznake koje predstavljaju vizuelne kontrole kao što je gornji element **Natpis**. Kontrole uvek smeštamo u kontejnere, a kontejner pored ostalih mehanizama kapsulira i raspoređivač (engl. *layout manager*) koji upravlja rasporedom kontrola u kontejneru. U najjednostavnijem slučaju, kao u gornjem primeru, **Application** deluje kao kontejner. Podrazumevani raspoređivač elementa **Application** smešta kontrole vertikalno niz prozor, redosledom kojim su bile deklarisane.

ActionScript je verzija ECMAScripta, ili JavaScripta, koja potpuno liči na Javu i podržava klase i strogu proveru tipova, kao i dinamičko pravljenje skriptova. Kada primeru dodamo skript, uvešćemo ponašanje. Ovde MXML kontrola **Script** smešta ActionScript neposredno u MXML datoteku:

```
//:! gui/flex/zdravoflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Script>
    <![CDATA[
    function updateOutput() {
      output.tekst = "Zdravo! " + input.tekst;
    }
    ]]>
  </mx:Script>
  <mx:TextInput id="input" width="200"
    change="updateOutput()" />
  <mx:Natpis id="output" tekst="Zdravo!" />
</mx:Application>
///:~
```

[11] Navedena adresa više nije dostupna. Više informacija o MXML-u naći ćete na adresi *www.adobe.com/devnet/flex/mxml_as.html*.

TextInput kontrola prima korisnikov unos, a **Natpis** (već tokom unosa) prikazuje unesene podatke. Vodite računa o tome da je atribut **id** svake kontrole u skriptu dostupan kao ime promenljive, pa skript može da referencira instance MXML oznaka. U polju **TextInput** vidite da je atribut **change** povezan s funkcijom **updateOutput( )** – dakle, ta funkcija se poziva kad god se desi bilo kakva promena (engl. *change*).

Prevođenje MXML-a

Najbolje je da sa adrese *www.adobe.com/products/flex/trial* preuzmete besplatnu probnu verziju Flexa.[12] Taj proizvod postoji u više izdanja – od besplatnih probnih do serverskih za velika preduzeća – a za razvoj Flex aplikacija Adobe nudi još neke alatke. Pošto se izdanja stalno menjaju, pročitajte pojedinosti na Web lokaciji proizvođača Adobe. Takođe, imajte u vidu da ćete možda morati da modifikujete datoteku **jvm.config** u direktorijumu **bin** Flexove instalacije.

Za prevođenje MXML koda u Flashov bajtkod imate dve opcije:

1. MXML datoteku možete smestiti u neku Java Web aplikaciju, pored JSP i HTML stranica u WAR datoteci, i dati da se u vreme izvršavanja zahtevi za **.mxml** datotekom prevode kad god bilo koji čitač zatraži URL adresu tog MXML dokumenta.
2. MXML datoteku možete prevesti pomoću Flexovog prevodioca **mxmlc** koji se pokreće s komandne linije.

Prva opcija, prevođenje na Webu u vreme izvršavanja, sem Flexa zahteva i neki kontejner servleta (kao što je Apacheov Tomcat). WAR datoteke kontejnera servleta moraju biti ažurirane pomoću informacija iz Flexove konfiguracije, kao što su mapiranja servleta dodata deskriptoru **web.xml**, i moraju obuhvatati Flexove JAR datoteke – sve se to obavlja automatski kada instalirate Flex. Nakon konfigurisanja WAR datoteke, MXML datoteke možete smestiti u Web aplikaciju i zatražiti URL adresu tog dokumenta bilo kojim čitačem. Flex će prevesti aplikaciju nakon prvog takvog zahteva, slično kao u modelu Javinih serverskih stranica (JSP), i potom će isporučivati prevedeni i keširani SWF unutar HTML ljuske.

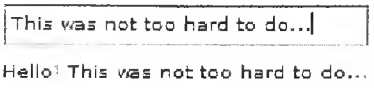
U drugoj opciji server nije potreban. SWF datoteke pravite pozivom Flexovog prevodioca **mxmlc** na komandnoj liniji. Njih možete upotrebiti kako god želite. Izvršna datoteka **mxmlc** je u direktorijumu **bin** Flexove instalacije, i kada je pozovete bez argumenata, ispisaće spisak validnih opcija u komandnoj liniji. Obično se lokacija Flexove biblioteke klijentskih komponenata zadaje kao vrednost opcije **-flexlib**, ali u veoma jednostavnim slučajevima kao što su dva prethodna, Flexov prevodilac će pretpostaviti gde je biblioteka komponenata. Zato prva dva primera možete prevesti ovako:

```
mxmlc.exe zdravoflex1.mxml
mxmlc.exe zdravoflex2.mxml
```

Time se proizvodi datoteka **zdravoflex2.swf** koja se može pokrenuti u Flashu ili smestiti uz HTML na neki HTTP server. (Nakon učitavanja Flasha u čitač Weba, često je dovoljno dvaput pritisnuti SWF datoteku da biste je pokrenuli u čitaču.)

[12] Pazite – treba da preuzmete Flex, a ne FlexBuilder, što je alatka za projektovanje programa.

Kada pokrenete **zdravoflex2.swf**, videćete u Flash Playeru sledeće korisničko okruženje:



U složenijim aplikacijama možete razdvojiti MXML i ActionScript tako što ćete funkcije referencirati u spoljnim ActionScript datotekama. U MXML-u upotrebite sledeću sintaksu za kontrolu **Script**:

```
<mx:Script source="MojSpoljniSkript.as" />
```

Taj kôd omogućuje MXML kontrolama da referenciraju funkcije smeštene u datoteku **MojSpoljniSkript.as** kao da su smeštene u samu tu MXML datoteku.

MXML i ActionScript

MXML je deklarativna stenografija za ActionScript klase. Za svaku MXML oznaku postoji istoimena ActionScript klasa. Kada Flexov prevodilac leksički analizira (raščlanjuje i razrešava) MXML, prvo transformiše XML u ActionScript i učitava referencirane ActionScript klase, a potom prevodi i povezuje taj ActionScript u SWF datoteku.

Celu Flex aplikaciju možete napisati u samom ActionScriptu, a da uopšte ne upotrebite MXML. Dakle, MXML nije neophodan, ali je pogodan. MXML se obično upotrebljava za deklarisanje komponenata korisničkog okruženja kao što su kontejneri i kontrole, dok se ActionScript i Java koriste za obradu događaja i ostalu klijentsku logiku.

Imate mogućnost da pravite sopstvene MXML kontrole i da ih referencirate u ActionScript klasama. Postojeće MXML kontejnere i kontrole možete da kombinujete u novom MXML dokumentu koji može biti referenciran kao oznaka u drugim MXML dokumentima. Više informacija o tome potražite na Web lokaciji kompanije Adobe.

Kontejneri i kontrole

Vizuelno jezgro biblioteke Flex komponenata je skup kontejnera koji upravljaju rasporedom i niz kontrola koje idu u te kontejnere. Među kontejnerima su panoi, vertikalni i horizontalni pravougaonici, pravougaonici koji se slažu kao pločice, pravougaonici koji se šire poput harmonike, pravougaonici sa unutrašnjom mrežom, tabele itd. Kontrole su spravice (engl. *widgets*) korisničkog okruženja kao što su dugmad, tekstualna polja, klizači, kalendari, tabele podataka itd.

Sada ćemo napraviti Flex aplikaciju koja prikazuje i uređuje listu audio datoteka. Videćete primere kontejnera, kontrola i otkrićete kako da ih iz Flasha povežete s Javom.

Na početak MXML datoteke postavićemo kontrolu **DataGrid** (koja spada u sofisticiranije Flex kontrole) u kontejner tipa **Panel**:

```
//:! gui/flex/pesme.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
```

```
xmlns:mx="http://www.macromedia.com/2003/mxml"
backgroundColor="#B9CAD2" pageTitle="Flex program za rukovanje pesmama"
initialize="dajPesme()">
<mx:Script source="skriptPesama.as" />
<mx:Style source="stiloviPesama.css"/>
<mx:Panel id="panoListePesama"
  titleStyleDeclaration="headerText"
  title="Flex MP3 biblioteka">
  <mx:HBox verticalAlign="bottom">
    <mx:DataGrid id="tabelaPesama"
      cellPress="izaberiPesmu(dogadjaj)" rowCount="8">
      <mx:columns>
        <mx:Array>
          <mx:DataGridColumn columnName="naslov"
            headerText="Naslov pesme" width="120" />
          <mx:DataGridColumn columnName="izvodjač"
            headerText="Izvodjač" width="180" />
          <mx:DataGridColumn columnName="album"
            headerText="Album" width="160" />
        </mx:Array>
      </mx:columns>
    </mx:DataGrid>
    <mx:VBox>
      <mx:HBox height="100" >
        <mx:Slika id="slikaAlbuma" source=""
          height="80" width="100"
          mouseOverEffect="promVelPovecaj"
          mouseOutEffect="promVelSmanji" />
        <mx:TextArea id="podaciOPesmi"
          styleName="boldText" height="100%" width="120"
          vScrollPolicy="off" borderStyle="none" />
      </mx:HBox>
      <mx:MediaPlayback id="plejerZaPesme"
        contentPath=""
        mediaType="MP3"
        height="70"
        width="230"
        controllerPolicy="on"
        autoPlay="false"
        visible="false" />
    </mx:VBox>
  </mx:HBox>
  <mx:ControlBar horizontalAlign="right">
    <mx:Button id="dugmeOsveziPesme"
      label="Osveži pesme" width="100"
      toolTip=" Osveži listu pesama"
      click="uslugaPesama.dajPesme()" />
  </mx:ControlBar>
```

```
    </mx:Panel>
    <mx:Effect>
      <mx:PromVel name="promVelPovecaj" heightTo="100"
        duration="500"/>
      <mx:PromVel name="promVelSmanji" heightTo="80"
        duration="500"/>
    </mx:Effect>
    <mx:RemoteObject id="uslugaPesama"
      source="gui.flex.UslugaPesama"
      rezultat="onPesme(dogadjaj.rezultat)"
      fault="alert(dogadjaj.fault.faultstring, 'Greška')">
      <mx:method name="dajPesme"/>
    </mx:RemoteObject>
  </mx:Application>
  ///:~
```

Kontejner tipa **DataGrid** sadrži ugnežđene oznake za svoj niz kolona. Svaki atribut ili ugnežđeni element na kontroli odgovara nekom svojstvu, događaju ili kapsuliranom objektu pripadne ActionScript klase. Taj **DataGrid** ima atribut **id** čija je vrednost **tabelaPesama**, pa ActionScript i MXML oznake mogu programski da referenciraju tu tabelu koristeći **tabelaPesama** kao ime promenljive. **DataGrid** eksponira mnogo više svojstava nego što sam ja ovde pokazao; celokupni API za MXML kontrole i kontejnere možete naći na adresi *http://livedocs.adobe.com/flex/1/asdocs*.

Iza kontejnera **DataGrid** sledi **VBox** sa elementom **Slika** koji pokazuje prednju stranu albuma i podatke o pesmi, kao i s kontrolom **MediaPlayback** za reprodukciju MP3 datoteka. U ovom primeru sadržaj se emituje u realnom vremenu, da bi se smanjila prevedena SWF datoteka. Ukoliko slike, audio i video datoteke ugradite u Flex aplikaciju umesto da ih emitujete u realnom vremenu, te datoteke postaju deo prevedenog SWF-a i isporučuju se zajedno s korisničkim okruženjem; u tom slučaju, u vreme izvršavanja nema emitovanja u realnom vremenu na zahtev.

Flash plejer sadrži ugrađene kodeke za reprodukciju i emitovanje audio i video datoteka raznih formata u realnom vremenu. Flash i Flex podržavaju najkorišćenije formate slika na Webu, a Flex ume da konvertuje datoteke *skalabilne vektorske grafike* (SVG) u SWF resurse koji se mogu ugraditi u Flex klijente.

Efekti i stilovi

Flash plejer iscrtava grafiku vektorski, pa u vreme izvršavanja može da obavi veoma izražajne transformacije. Flexovi *efekti* omogućuju da steknemo uvid u takve animacije. Efekti su transformacije koje pomoću MXML sintakse možete primeniti na kontrole i kontejnere.

Oznaka **Effect** prikazana u prethodnom MXML kodu proizvodi dva rezultata: prva ugnežđena oznaka dinamički povećava sliku kada se miš nalazi iznad nje, a druga je dinamički smanjuje kada miš ode s nje. Ti efekti se primenjuju na događaje miša dostupne za kontrolu **Slika** nazvanu **slikaAlbuma**.

Flex ima efekte i za uobičajene animacije kao što su prelazi, prebrisavanja i modulirajući alfa kanali. Pored ugrađenih efekata, Flex podržava i Flashov API za crtanje zaista inovativnih animacija. Dublje istraživanje ove teme obuhvata grafički dizajn i animaciju, i prevazilazi opseg ovog odeljka.

Flex podržava kaskadne opise stilova (Cascading Style Sheets, CSS), pa znači i standardne stilove. Ako MXML datoteci pridružite CSS datoteku, Flex kontrole će se pridržavati tih stilova. U ovom primeru, kaskadni opis stilova **stiloviPesama.css** sadrži sledeću CSS deklaraciju:

```
//:! gui/flex/stiloviPesama.css
.headerText {
  font-family: Arial, "_sans";
  font-size: 16;
  font-weight: bold;
}

.boldText {
  font-family: Arial, "_sans";
  font-size: 11;
  font-weight: bold;
}
///:~
```

Aplikacija biblioteke pesama uvozi tu datoteku i upotrebljava je preko oznake **Style** u MXML datoteci. Nakon uvoza opisa stilova, njegove deklaracije se mogu primeniti na Flex kontrole u MXML datoteci. Na primer, deklaraciju **boldText** iz opisa stilova upotrebljava kontrola **TextArea** čiji je identifikator (**id**) **podaciOPesmi**.

Događaji

Korisničko okruženje je mašina stanja; ono obavlja radnje kada se menja stanje. U Flexu se te promene prijavljuju pomoću događaja. Flexova biblioteka klasa sadrži mnoštvo najrazličitijih kontrola koje podržavaju mnoštvo događaja za sve aspekte kretanja miša i korišćenja tastature.

Primera radi, atribut **click** elementa **Button** predstavlja jedan od događaja dostupnih za tu kontrolu. Vrednost dodeljena atributu **click** može biti neka funkcija ili umetnut skript. Primera radi, u gornjoj MXML datoteci, element **ControlBar** sadrži **dugmeOsveziPesme** za osvežavanje liste pesama. Iz oznake vidite da događaj **click** izaziva poziv metode **uslugaPesama.dajPesme( )**. U tom primeru, događaj **click** dugmeta **Button** referencira **RemoteObject** (udaljeni objekat) koji odgovara toj Java metodi.

Povezivanje s Javom

Oznaka **RemoteObject** na kraju MXML datoteke uspostavlja vezu sa spoljnom Java klasom, **gui.flex.UslugaPesama**. Pomenuti Flex klijent – metodom **dajPesme( )** te Java klase – pribavlja podatke za **DataGrid**. Da bi to mogao da radi, mora izgledati kao *usluga* – krajnja

tačka s kojom klijent može da razmenjuje poruke. Usluga definisana u oznaci **RemoteObject** ima atribut **source** koji naznačuje Java klasu oznake **RemoteObject**, i specificira ActionScript povratnu funkciju, **onPesme( )**, koja će biti pozvana kada se Java metoda vrati. Ugnežđena oznaka **method** deklariše metodu **dajPesme( )**, što tu Java metodu čini dostupnom ostatku Flex aplikacije.

U Flexu se svi pozivi usluga vraćaju asinhrono, preko događaja koji prouzrokuju (izazivaju) te povratne funkcije. U slučaju greške, isti **RemoteObject** prikazuje dijalog upozorenja.

Metodu **dajPesme( )** sada možemo pozvati iz Flasha pomoću ActionScripta:

```
uslugaPesama.dajPesme();
```

Zbog konfiguracije MXML-a time će biti pozvana metoda **dajPesme( )** u klasi **UslugaPesama**:

```
//: gui/flex/UslugaPesama.java
package gui.flex;
import java.util.*;

public class UslugaPesama {
  private List<Pesma> pesme = new ArrayList<Pesma>();
  public UslugaPesama() { popuniProbnePodatke(); }
  public List<Pesma> dajPesme() { return pesme; }
  public void dodajPesmu(Pesma pesma) { pesme.add(pesma); }
  public void ukloniPesmu(Pesma pesma) { pesme.remove(pesma); }
  private void popuniProbnePodatke() {
    dodajPesmu(new Pesma("Chocolate", "Snow Patrol",
      "Final Straw", "sp-final-straw.jpg",
      "chocolate.mp3"));
    dodajPesmu(new Pesma("Koncert br. 2 u E-duru", "Hilary Hahn",
      "Bach: violinski koncerti", "hahn.jpg",
      "bachviolina2.mp3"));
    dodajPesmu(new Pesma("'Round Midnight", "Wes Montgomery",
      "The Artistry of Wes Montgomery",
      "wesmontgomery.jpg", "roundmidnight.mp3"));
  }
} ///:~
```

Svaki objekat tipa **Pesma** samo je kontejner podataka:

```
//: gui/flex/Pesma.java
package gui.flex;

public class Pesma implements java.io.Serializable {
  private String naslov;
  private String izvodjac;
  private String album;
  private String urlSlikeNaAlbumu;
```

```
  private String urlNosacaZvuka;
  public Pesma() {}
  public Pesma(String naslov, String izvodjac, String album,
  String urlSlikeNaAlbumu, String urlNosacaZvuka) {
    this.naslov = naslov;
    this.izvodjac = izvodjac;
    this.album = album;
    this.urlSlikeNaAlbumu = urlSlikeNaAlbumu;
    this.urlNosacaZvuka = urlNosacaZvuka;
  }
  public void setAlbum(String album) { this.album = album;}
  public String getAlbum() { return album; }
  public void setUrlSlikeNaAlbumu(String urlSlikeNaAlbumu) {
    this.urlSlikeNaAlbumu = urlSlikeNaAlbumu;
  }
  public String getUrlSlikeNaAlbumu() { return urlSlikeNaAlbumu;}
  public void setIzvodjac(String izvodjac) {
    this.izvodjac = izvodjac;
  }
  public String getIzvodjac() { return izvodjac; }
  public void setNaslov(String naslov) { this.naslov = naslov; }
  public String getNaslov() { return naslov; }
  public void setUrlNosacaZvuka(String urlNosacaZvuka) {
    this.urlNosacaZvuka = urlNosacaZvuka;
  }
  public String getUrlNosacaZvuka() { return urlNosacaZvuka; }
} ///:~
```

Prilikom inicijalizacije aplikacije ili kada pritisnete **dugmeOsveziPesme**, poziva se metoda **dajPesme( )**, a nakon povratka, poziva se ActionScript **onPesme(dogadjaj.rezultat)** za popunjavanje tabele **tabelaPesama**.

Sledi listing ActionScripta koji sadrži kontrola **Script** MXML datoteke:

```
//: gui/flex/skriptPesama.as
function dajPesme() {
  uslugaPesama.dajPesme();
}

function izaberiPesmu(dogadjaj) {
  var pesma = tabelaPesama.getItemAt(dogadjaj.itemIndex);
  prikazipodatkeOPesmi(pesma);
}

function prikazipodatkeOPesmi(pesma) {
  podaciOPesmi.tekst = pesma.naslov + newline;
  podaciOPesmi.tekst += pesma.izvodjac + newline;
  podaciOPesmi.tekst += pesma.album + newline;
  slikaAlbuma.source = pesma.urlSlikeNaAlbumu;
```

```
    plejerZaPesme.contentPath = pesma.urlNosacaZvuka;
    plejerZaPesme.visible = true;
  }
  function onPesme(pesme) {
    tabelaPesama.dataProvider = pesme;
  } ///:~
```

Da bismo obradili izbor **DataGrid** ćelija, atribut događaja **cellPress** dodajemo deklaraciji elementa **DataGrid** u MXML datoteci:

```
cellPress="izaberiPesmu(dogadjaj)"
```

Kada korisnik pritisne pesmu u tabeli **DataGrid**, pozvaće metodu **izaberiPesmu( )** u gornjem ActionScriptu.

Modeli podataka i povezivanje podataka

Kontrole mogu neposredno da pozivaju usluge, a povratni pozivi ActionScript događaja daju priliku da programski ažurirate vizuelne kontrole kada usluge vrate podatke. Iako je skript za ažuriranje kontrola jednostavan, ume da postane obiman i zamoran, a njegove funkcije su toliko uobičajene da Flex automatski upravlja tim ponašanjem tako što povezuje podatke.

U svom najjednostavnijem obliku, povezivanje podataka omogućuje kontroli da podatke referencira neposredno umesto da nekim šablonskim kodom kopira podatke u kontrolu. Prilikom ažuriranja podataka, automatski se ažurira i kontrola koja ih referencira, bez ikakve intervencije programera. Infrastruktura Flexa tačno odgovara na događaje promene podataka i ažurira sve kontrole koje su povezane s tim podacima.

Ovo je jednostavan primer sintakse povezivanja podataka:

```
<mx:Slider id="mojKlizac"/>
<mx:Text tekst="{mojKlizac.value}"/>
```

Podatke povezujete kada smestite reference u vitičaste zagrade. Sve unutar tih vitičastih zagrada Flex smatra izrazom koji treba izračunati.

Vrednost prve kontrole, elementa **Slider** (klizača), prikazuje druga kontrola – jedno tekstualno polje. S promenom vrednosti klizača, svojstvo **tekst** tekstualnog polja automatski se ažurira. Na taj način, programer ne mora da obrađuje događaje promena vrednosti elementa **Slider** da bi ažurirao tekstualno polje.

Ima i sofisticiranijih kontrola, kao što su **Tree** (stablo) i **DataGrid** u aplikaciji biblioteke pesama. Te kontrole imaju svojstvo **dataprovider** koje olakšava povezivanje s kolekcijama podataka. ActionScript funkcija **onPesme( )** prikazuje kako je metoda **UslugaPesama.dajPesme( )** povezana sa svojstvom **dataprovider** Flexovog elementa **DataGrid**. Kao što je deklarisano u oznaci **RemoteObject** MXML datoteke, tu povratnu funkciju poziva ActionScript kada se Java metoda vrati.

Sofisticiranija primena složenijeg modela podataka, kao što je intranet aplikacija koja upotrebljava objekte za prenos podataka (Data Transfer Objects) ili dostavljače poruka

čiji su podaci usklađeni sa složenim šemama, može podstaći dalje razdvajanje izvora podataka od kontrola. U razvoju Flex programa, to razdvajanje postižemo deklarisanjem objekta „modela", što je generički MXML kontejner podataka. Taj model ne sadrži logiku. On je ekvivalentan objektu za prenos podataka kakav se sreće pri razvoju intranet programa i odgovarajućim strukturama u ostalim programskim jezicima. Po tom modelu, povezujemo podatke kontrola s modelom i istovremeno model povezuje svoja svojstva sa ulazima i izlazima usluga. Time se izvori podataka, usluge, razdvajaju od vizuelnih potrošača podataka, čime se olakšava korišćenje obrasca *model–prikaz–kontroler* (*Model-View-Controller*, MVC). U većim, sofisticiranijim aplikacijama, početno povećanje složenosti zbog umetanja modela često se isplati jer se dobija MVC aplikacija s jasno razdvojenim podacima od kontrola.

Sem Java objekata, Flex pomoću kontrola **WebService** odnosno **HttpService** ume da pristupa i Web uslugama čiju osnovu čini SOAP – tj. RESTful HTTP uslugama. Pristupanje svim uslugama podleže bezbednosnim ograničenjima provere identiteta.

Izgradnja i primena

U prethodnim primerima mogli smo da se provučemo bez indikatora **-flexlib** na komandnoj liniji, ali da bismo preveli ovaj program, indikatorom **-flexlib** moramo specificirati mesto datoteke **flex-config.xml**. Sledeća komanda je prikladna za moju instalaciju, a vi ćete morati da je modifikujete u skladu sa sopstvenom konfiguracijom. (Komanda se piše u jednom redu koji se zbog dužine nastavlja i u narednom):

```
//:! gui/flex/build-command.txt
mxmlc -flexlib C:/"Program Files"/Adobe/Flex/jrun4/servers/default/flex/
WEB-INF/flex pesme.mxml
///:~
```

Ta će komanda izgraditi aplikaciju u obliku SWF datoteke koju možete prikazati u čitaču Weba, ali pošto u datoteci za distribuciju koda ove knjige nema ni MP3 datoteka niti JPG slika, kada pokrenete aplikaciju nećete čuti nijednu pesmu niti ćete videti omote albuma, nego samo osnovnu strukturu (engl. *framework*).

Pored toga, morate konfigurisati neki server da biste iz Flex aplikacije mogli da komunicirate s Java datotekama. Flexov probni paket obuhvata i server JRun, a njega nakon instalacije Flexa možete pokrenuti bilo koristeći menije operativnog sistema, bilo s komandne linije:

```
jrun -start default
```

Proverite da li je server uspešno pokrenut tako što ćete u čitaču Weba otvoriti *http://localhost:8700/samples* i prikazati razne uzorke (engl. *samples*). To je i dobar način za upoznavanje mogućnosti Flexa.

Umesto da aplikaciju prevodite na komandnoj liniji, možete da je prevedete posredstvom servera. To se radi tako što izvorne datoteke pesama, CSS opise stilova itd. smestite u direktorijum **jrun4/servers/default/flex** i pristupite im u čitaču Weba otvaranjem adrese *http://localhost:8700/flex/pesme.mxml*.

Morate konfigurisati i Javinu i Flexovu stranu da biste mogli pokrenuti aplikaciju.

Java: prevedene datoteke **Pesma.java** i **UslugaPesama.java** smestite u direktorijum **WEB-INF/classes**. Prema specifikaciji J2EE, to je mesto za WAR klase. Drugi način bi bio da datoteke pretvorite u JAR arhive i smestite ih u direktorijum **WEB-INF/lib**. One se moraju nalaziti u direktorijumu čija se struktura podudara sa odgovarajućim Java paketom. Ako koristite server JRun, datoteke treba smestiti u direktorijume **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/Pesma.class** i **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/UslugaPesama.class**. Potrebne su vam i datoteke slika i MP3 audio datoteke koje će biti dostupne u Web aplikaciji. (Za JRun, korenski direktorijum Web aplikacije je **jrun4/servers/default/flex**.)

Flex: iz bezbednosnih razloga, Flex ne može da pristupi Java objektima ukoliko mu to ne dozvolite tako što ćete modifikovati datoteku **flex-config.xml**. Za server JRun, ona treba da se nalazi u direktorijumu **jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml**. Pronađite stavku <**remote-objects**> u toj datoteci i njen odeljak <**whitelist**>; videćete sledeću napomenu:

```
<!--
For security, the whitelist is locked down by default. Uncomment the source element
below to enable access to all classes during development.
We strongly recommend not allowing access to all source files in production, since
this exposes Java and Flex system classes.
<source>*</source>
-->
(<!--
```

Iz bezbednosnih razloga, lista dostupnih klasa je podrazumevano zaključana. Ukoliko želite da tokom razvoja programa bude dozvoljen pristup svim klasama, uklonite znakove koji od donjeg elementa prave komentar.

Izričito preporučujemo da u kodu koji se isporučuje ne dozvolite pristupanje svim izvornim datotekama, pošto biste time eksponirali klase Jave i sistema Flex.

```
<source>*</source>
-->)
```

Da biste dozvolili pristupanje svim izvornim datotekama, uklonite znakove koji od stavke <**source**> prave komentar, tako da glasi <**source**>*</**source**>. Značenje ove i drugih stavki opisano je u dokumentima za konfigurisanje Flexa.

Vežba 37: (3) Izgradite goreprikazan „jednostavan primer sintakse za povezivanje podataka".

Vežba 38: (4) U datoteci koda ove knjige koja se može preuzeti sa Weba nema ni MP3 niti JPG datoteka navedenih u programu **UslugaPesama.java**. Nađite neku sliku i pesmu (MP3 odnosno JPG datoteku), izmenite program **UslugaPesama.java** tako da sadrži i imena tih datoteka, preuzmite s Weba probnu verziju Flexa i izgradite gorenavedenu aplikaciju.

Izrada SWT aplikacija

Kao što sam već rekao, Swing sve UI komponente pravi piksel po piksel pa zato mogu postojati sve željene komponente, bez obzira na to da li ih lokalni operativni sistem ima ili nema. SWT koristi matične komponente ako ih OS ima, a pravi samo ono što OS nema. Dobija se aplikacija koja korisniku liči na matičnu, a često je i znatno brža od ekvivalentnog Swing programa. Pored toga, SWT-ov programski model je jednostavniji od Swingovog, što je poželjno u većini primena.[13]

Pošto SWT upošljava matični OS koliko god je to moguće, on automatski može da iskoristi mogućnosti operativnog sistema koje su Swingu nedostupne – recimo, Windows podržava „iscrtavanje tačaka manjih od piksela", kojima se na LCD ekranima jasnije prikazuju fontovi.

SWT ume da pravi čak i aplete.

Ovaj odeljak nije sveobuhvatan uvod u SWT, nego samo omogućuje da ga probate i vidite koliko se razlikuje od Swinga. Videćete da SWT ima mnogo elemenata (vidžeta) i da se oni prilično lako koriste. Pojedinosti možete da proučavate u sveobuhvatnoj dokumentaciji i mnogim primerima dostupnim na adresi *www.eclipse.org*. O programiranju u SWT-u već postoji više knjiga, a pišu se i nove.

Instaliranje SWT-a

Za SWT aplikacije potrebno je da preuzmete i instalirate SWT biblioteku projekta Eclipse. Izaberite neki od preslikanih servera na lokaciji *www.eclipse.org/downloads/*. Sledite veze do tekućeg *builda* Eclipsea i pronađite komprimovanu datoteku čije ime počinje sa swt i sadrži ime vaše platforme (na primer, win32). U toj datoteci naći ćete **swt.jar**. Datoteku **swt.jar** najlakše ćete instalirati ukoliko je smestite u direktorijum **jre/lib/ext** (jer u tom slučaju nećete morati da menjate svoju putanju klasa). Kada dekomprimujete biblioteku SWT, možda ćete naći još datoteka koje treba instalirati na mestima odgovarajućim za vašu platformu. Na primer, Win32 distribucija obuhvata i DLL datoteke koje treba smestiti negde u **java.library.path**. (Obično je ta putanja jednaka onoj koju opisuje sistemska promenljiva PATH, ali stvarnu vrednost **java.library.path** možete saznati pomoću programom **object/ShowProperties.java**). Kada to uradite, trebalo bi da možete transparentno da prevodite i pokrećete SWT aplikacije, kao sve druge Java programe.

Dokumentacija za SWT preuzima se zasebno.

Druga mogućnost je da instalirate program za uređivanje teksta Eclipse; on obuhvata i SWT i SWT dokumentaciju koju možete prikazati pomoću Eclipseovog sistema pomoći.

Zdravo, SWT

Počnimo najjednostavnijom mogućom aplikacijom u stilu „zdravo svima":

```
//: swt/ZdravoSWT.java
// {Requires: org.eclipse.swt.widgets.Display; Morate
// instalirati SWT biblioteku s lokacije http://www.eclipse.org }
```

[13] Chris Grindstaff mi je mnogo pomogao pri prevođenju SWT primera i izboru informacija o SWT-u.

```
import org.eclipse.swt.widgets.*;

public class ZdravoSWT {
  public static void main(String [] args) {
    Display prikaz = new Display();
    Shell ljuska = new Shell(prikaz);
    ljuska.setText("Zdravo, SWT!"); // Na naslovnoj traci
    ljuska.open();
    while(!ljuska.isDisposed())
      if(!prikaz.readAndDispatch())
        prikaz.sleep();
    prikaz.dispose();
  }
} ///:~
```

Ukoliko preuzmete izvorni kôd ove knjige, videćete da komentarska direktiva „Requires" završava u Ant datoteci **build.xml** kao preduslov potrebno je pravljenje poddirektorijuma **swt**; za sve datoteke koje uvoze **org.eclipse.swt** zahtevaju da instalirate biblioteku SWT s lokacije *www.eclipse.org*.

Klasa **Display** upravlja vezom između SWT-a i operativnog sistema – ona je deo *Mosta* između operativnog sistema i SWT-a. Klasa **Shell** je glavni prozor najvišeg nivoa; u njoj se grade sve ostale komponente. Kada pozovete metodu **setText( )**, njen argument biva pretvoren u natpis na naslovnoj traci prozora.

Metodom **open( )** klase **Shell** prikazujete taj prozor, a time i aplikaciju.

Dok Swing od programera skriva petlju za obradu događaja, SWT ga primorava da je eksplicitno napiše. Na vrhu te petlje proverite da li je ljuska ugašena – to vam pruža priliku da umetnete kôd za čišćenje. Ali to znači da je glavna nit **main( )** *istovremeno i* nit korisničkog okruženja. U Swingu se iza scene pravi druga nit za obradu događaja, ali u SWT-u događaje korisničkog okruženja obrađuje glavna nit, tj. nit metode **main( )**. Pošto se podrazumeva postojanje samo jedne niti a ne dve, nešto je manje verovatno da ćete korisničko okruženje zatrpati nitima.

Imajte u vidu da zadatke ne morate da šaljete niti korisničkog okruženja kao u Swingu. Ne samo da se o tome stara SWT, nego on baca izuzetak ako vidžet pokušate da obradite u pogrešnoj niti. Međutim, ukoliko za dugotrajne operacije treba da napravite nove niti, događaje treba da šaljete isto kao u Swingu. Za to, SWT ima tri metode koje se mogu pozvati za objekat tipa **Display**: **asyncExec(Runnable)**, **syncExec(Runnable)** i **timerExec(int, Runnable)**.

U tom trenutku vaša glavna nit (**main( )**) treba da pozove metodu **readAndDispatch( )** za objekat **Display** (to znači da aplikacija može imati samo jedan objekat tipa **Display**). Metoda **readAndDispatch( )** vraća **true** ukoliko u redu za čekanje ima više događaja koji čekaju na obradu. U tom slučaju, treba je odmah ponovo pozvati. Međutim, ako nema poslova koji čekaju na izvršavanje, pozovite metodu **sleep( )** objekta **Display** da biste neko kratko vreme sačekali pre nego što ponovo pozovete red za čekanje.

Kada se izvršavanje programa završi, morate eksplicitno pozvati metodu **dispose( )** objekta **Display**. SWT često zahteva eksplicitno čišćenje resursa, pošto su to najčešće resursi pripadnog operativnog sistema, kojih bi inače moglo ponestati.

Sledeći program pravi više **Shell** objekata da bi pokazao da je ljuska (objekat tipa **Shell**) glavni prozor:

```
//: swt/ljuskeSuGlavniProzori.java
import org.eclipse.swt.widgets.*;

public class ljuskeSuGlavniProzori {
  static Shell[] ljuske = new Shell[10];
  public static void main(String [] args) {
    Display prikaz = new Display();
    for(int i = 0; i < ljuske.length; i++) {
      ljuske[i] = new Shell(prikaz);
      ljuske[i].setText("ljuska #" + i);
      ljuske[i].open();
    }
    while(!ljuskeOciscene())
      if(!prikaz.readAndDispatch())
        prikaz.sleep();
    prikaz.dispose();
  }
  static boolean ljuskeOciscene() {
    for(int i = 0; i < ljuske.length; i++)
      if(ljuske[i].isDisposed())
        return true;
    return false;
  }
} ///:~
```

Kada pokrenete ovaj program, dobićete deset glavnih prozora. Zbog načina na koji je program napisan, ako zatvorite bilo koji od tih prozora, zatvoriće te i sve ostale.

I SWT koristi rasporedivače – različite od onih u Swingu, ali iste u principu. U narednom neznatno složenijem primeru dodaćemo ljusci tekst rezultata metode **System.getProperties( )**:

```
//: swt/SvojstvaKlaseDisplay.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class SvojstvaKlaseDisplay {
  public static void main(String [] args) {
    Display prikaz = new Display();
    Shell ljuska = new Shell(prikaz);
    ljuska.setText("Svojstva klase Display");
```

```
    ljuska.setLayout(new FillLayout());
    Text tekst = new Text(ljuska, SWT.WRAP | SWT.V_SCROLL);
    StringWriter svojstva = new StringWriter();
    System.getProperties().list(new PrintWriter(svojstva));
    tekst.setText(svojstva.toString());
    ljuska.open();
    while(!ljuska.isDisposed())
      if(!prikaz.readAndDispatch())
        prikaz.sleep();
    prikaz.dispose();
  }
} ///:~
```

U SWT-u svi vidžeti moraju imati roditeljski objekat opšteg tipa **Composite** koji konstruktoru vidžeta morate proslediti kao prvi argument. To vidite u konstruktoru klase **Text** kojem je prvi argument **ljuska**. Gotovo svi konstruktori primaju i indikatorski argument pomoću kojega možete zadati proizvoljan broj direktiva u vezi sa stilom, u zavisnosti od toga šta vidžet zahteva. Sve direktive u vezi sa stilom podvrgavaju se logičkoj disjunkciji po bitovima (operator |), kao u prethodnom primeru.

Prilikom pravljenja **Text**( ) objekta dodao sam indikatore stila koji prelamaju tekst (engl. *wrap*) i automatski mu dodaju vertikalnu traku za pomeranje (engl. *scroll bar*), ako treba. Videćete da se u SWT-u mnogo toga radi pomoću konstruktora; mnoge atribute vidžeta teško je ili nemoguće menjati sem pomoću konstruktora. U dokumentaciji vidžetovog konstruktora uvek morate proveriti koji su dozvoljeni indikatori. Neki konstruktori zahtevaju indikatorski argument čak i kada u dokumentaciji nemaju nijedan „dozvoljen" indikator. Time se omogućuje buduće proširenje, a da se interfejs ne menja.

Izbegavanje redundantnosti

Pre nego što nastavimo, navešćemo šta morate da uradite u svakoj SWT aplikaciji. U SWT-u uvek morate da napravite prikaz tj. objekat klase **Display**, od tog prikaza napravite ljusku tj. objekat klase **Shell**, napišete petlju **readAndDispatch**( ) itd. Naravno, u nekim posebnim slučajevima to ne morate da radite, ali oni su toliko retki da se isplati sabrati sav taj kôd u jedan program i izbeći ponavljanje, kao što smo za Swing uradili u programu **net.mindview.util.SwingKonzola**.

Nateraćemo svaku aplikaciju da se usaglasi sa interfejsom:

```
//: swt/util/SWTAplikacija.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTAplikacija {
  void createContents(Composite roditelj);
} ///:~
```

Aplikaciji se predaje objekat tipa **Composite** (čija je **Shell** potklasa), pomoću kojeg aplikacija metodom **createContents( )** mora da napravi sav svoj sadržaj. U odgovarajućem trenutku, **SWTKonzola.run( )** poziva metodu **createContents( )**, zadaje veličinu ljuske u skladu sa argumentom koji je korisnik prosledio metodi **run( )**, otvara tu ljusku, pokreće petlju događaja i na kraju uklanja ljusku kada program završi s radom:

```
//: swt/util/SWTKonzola.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTKonzola {
  public static void
  run(SWTAplikacija SWTApl, int sirina, int visina) {
    Display prikaz = new Display();
    Shell ljuska = new Shell(prikaz);
    ljuska.setText(SWTApl.getClass().getSimpleName());
    SWTApl.createContents(ljuska);
    ljuska.setSize(sirina, visina);
    ljuska.open();
    while(!ljuska.isDisposed()) {
      if(!prikaz.readAndDispatch())
        prikaz.sleep();
    }
    prikaz.dispose();
  }
} ///:~
```

Pored već navedenog, time se na naslovnoj traci ispisuje ime klase kojoj pripada **SWTAplikacija** i zadaje vrednost parametara **sirina** i **visina** za ljusku tipa **Shell**.

Napisaćemo varijaciju programa **SvojstvaKlaseDisplay.java** koji pomoću objekta tipa **SWTKonzola** prikazuje sistemsko okruženje:

```
//: swt/PrikaziSistemskoOkruzenje.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;

public class PrikaziSistemskoOkruzenje implements SWTAplikacija {
  public void createContents(Composite roditelj) {
    roditelj.setLayout(new FillLayout());
    Text tekst = new Text(roditelj, SWT.WRAP | SWT.V_SCROLL);
    for(Map.Entry stavka: System.getenv().entrySet()) {
      tekst.append(stavka.getKey() + ": " +
        stavka.getValue() + "\n");
    }
  }
```

```
  public static void main(String [] args) {
    SWTKonzola.run(new PrikaziSistemskoOkruzenje(), 800, 600);
  }
} ///:~
```

SWTKonzola će nam omogućiti da se usredsredimo na zanimljive aspekte aplikacije, umesto na kôd koji se ponavlja.

Vežba 39: (4) Izmenite program **SvojstvaKlaseDisplay.java** tako da upotrebljava klasu **SWTKonzola**.

Vežba 40: (4) Izmenite program **PrikaziSistemskoOkruzenje.java** tako da *ne* upotrebljava klasu **SWTKonzola**.

Meniji

Prikazaćemo osnovne menije tako što će sledeći primer učitati sopstveni izvorni kôd i razdeliti ga na reči, a zatim njima popuniti menije:

```
//: swt/Meniji.java
// Zabava s menijima.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Meniji implements SWTAplikacija {
  private static Shell ljuska;
  public void createContents(Composite roditelj) {
    ljuska = roditelj.getShell();
    Menu linija = new Menu(ljuska, SWT.BAR);
    ljuska.setMenuBar(linija);
    Set<String> recci = new TreeSet<String>(
      new TextFile("Meniji.java", "\\W+"));
    Iterator<String> it = recci.iterator();
    while(it.next().matches("[0-9]+"))
      ; // Pređi iza brojeva.
    MenuItem[] stavkaMenija = new MenuItem;
    for(int i = 0; i < stavkaMenija.length; i++) {
      stavkaMenija[i] = new MenuItem(linija, SWT.CASCADE);
      stavkaMenija[i].setText(it.next());
      Menu podmeni = new Menu(ljuska, SWT.DROP_DOWN);
      stavkaMenija[i].setMenu(podmeni);
    }
    int i = 0;
    while(it.hasNext()) {
      addItem(linija, it, stavkaMenija[i]);
      i = (i + 1) % stavkaMenija.length;
```

```
    }
  }
  static Listener prijemnik = new Listener() {
    public void handleEvent(Event e) {
      System.out.println(e.toString());
    }
  };
  void
  addItem(Menu linija, Iterator<String> it, MenuItem stavkaMenija) {
    MenuItem stavka = new MenuItem(stavkaMenija.getMenu(),SWT.PUSH);
    stavka.addListener(SWT.Selection, prijemnik);
    stavka.setText(it.next());
  }
  public static void main(String[] args) {
    SWTKonzola.run(new Meniji(), 600, 200);
  }
} ///:~
```

Meni (objekat tipa **Menu**) mora biti smešten u neku ljusku tj. objekat tipa **Shell**, a natklasa **Composite** omogućuje da tu ljusku pribavite metodom **getShell( )**. **TextFile** je deo paketa **net.mindview.util** i opisana je u prethodnom delu knjige; ovde se rečima popunjava skup tipa **TreeSet**, pa će njihov poredak biti uređen. Početni elementi su brojevi, koji se potom odbacuju. Pomoću toka reči daju se imena menijima najvišeg nivoa u traci menija, zatim se prave podmeniji i popunjavaju rečima dok ih ne ponestane.

Kao odgovor na izbor stavke menija, prijemnik (objekat tipa **Listener**) samo ispisuje taj događaj tako da možete videti vrstu informacija koje sadrži. Kada pokrenete program, videćete da te informacije obuhvataju i natpis na meniju, pa na osnovu njega možete napraviti odgovor menija – ili ćete za svaki meni napraviti zaseban prijemnik (što je bezbednije u pogledu internacionalizacije).

Okna s karticama, dugmad i događaji

SWT ima bogat skup kontrola koje se u njemu nazivaju spravice ili *vidžeti* (engl. *widgets*). Osnovni vidžeti su navedeni u dokumentaciji za **org.eclipse.swt.widgets**, a oni složeniji u dokumentaciji od **org.eclipse.swt.custom**.

Prikazaćemo nekoliko osnovnih vidžeta tako što ćemo više komponenata smestiti u okna s karticama. Videćete kako se prave objekti tipa **Composite** (slično kao Swingovi **JPaneli**) da bi se stavke smestile unutar drugih stavki.

```
//: swt/OknoSKarticama.java
// Smeštanje SWT komponenata u okna s karticama.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;
```

```
public class OknoSKarticama implements SWTAplikacija {
  private static TabFolder direktorijum;
  private static Shell ljuska;
  public void createContents(Composite roditelj) {
    ljuska = roditelj.getShell();
    roditelj.setLayout(new FillLayout());
    direktorijum = new TabFolder(ljuska, SWT.BORDER);
    labelTab();
    directoryDialogTab();
    buttonTab();
    sliderTab();
    scribbleTab();
    browserTab();
  }
  public static void labelTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("Natpis"); // Tekst na kartici
    kartica.setToolTipText("Kratak natpis");
    Label natpis = new Label(direktorijum, SWT.CENTER);
    natpis.setText("Tekst natpisa");
    kartica.setControl(natpis);
  }
  public static void directoryDialogTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("Dijalog unutar direktorijuma");
    kartica.setToolTipText("Izaberite direktorijum");
    final Button dugme = new Button(direktorijum, SWT.PUSH);
    dugme.setText("Izaberite direktorijum");
    dugme.addListener(SWT.MouseDown, new Listener() {
        public void handleEvent(Event e) {
          DirectoryDialog dd = new DirectoryDialog(ljuska);
          String putanja = dd.open();
          if(putanja != null)
            dugme.setText(putanja);
        }
      });
    kartica.setControl(dugme);
  }
  public static void buttonTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("Dugmad");
    kartica.setToolTipText("Razne vrste dugmadi");
    Composite kompozita = new Composite(direktorijum, SWT.NONE);
    kompozita.setLayout(new GridLayout(4, true));
    for(int dir : new int[]{
        SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
      }) {
      Button dugme = new Button(kompozita, SWT.ARROW | dir);
      dugme.addListener(SWT.MouseDown, prijemnik);
    }
```

```
    newButton(kompozita, SWT.CHECK, "Polje za potvrdu");
    newButton(kompozita, SWT.PUSH, "Obično dugme");
    newButton(kompozita, SWT.RADIO, "Radio-dugme");
    newButton(kompozita, SWT.TOGGLE, "Preklopno dugme");
    newButton(kompozita, SWT.FLAT, "Ravno dugme");
    kartica.setControl(kompozita);
  }
  private static Listener prijemnik = new Listener() {
      public void handleEvent(Event e) {
        MessageBox poruka = new MessageBox(ljuska, SWT.OK);
        poruka.setMessage(e.toString());
        poruka.open();
      }
    };
  private static void newButton(Composite kompozita,
    int tip, String natpis) {
    Button dugme = new Button(kompozita, tip);
    dugme.setText(natpis);
    dugme.addListener(SWT.MouseDown, prijemnik);
  }
  public static void sliderTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("Klizači i trake napredovanja");
    kartica.setToolTipText("Klizač povezan s trakom napredovanja");
    Composite kompozita = new Composite(direktorijum, SWT.NONE);
    kompozita.setLayout(new GridLayout(2, true));
    final Slider klizac =
      new Slider(kompozita, SWT.HORIZONTAL);
    final ProgressBar napredovanje =
      new ProgressBar(kompozita, SWT.HORIZONTAL);
    klizac.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent dogadjaj) {
          napredovanje.setSelection(klizac.getSelection());
        }
      });
    kartica.setControl(kompozita);
  }
  public static void scribbleTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("Za žvrljanje");
    kartica.setToolTipText("Jednostavna grafika: crtanje");
    final Canvas platno = new Canvas(direktorijum, SWT.NONE);
    PrijemnikMisaZaZvrljanje pmz= new PrijemnikMisaZaZvrljanje();
    platno.addMouseListener(pmz);
    platno.addMouseMoveListener(pmz);
    kartica.setControl(platno);
  }
  private static class PrijemnikMisaZaZvrljanje
```

```
    extends MouseAdapter implements MouseMoveListener {
    private Point tacka = new Point(0, 0);
    public void mouseMove(MouseEvent e) {
      if((e.stateMask & SWT.BUTTON1) == 0)
        return;
      GC grafKontekst = new GC((Canvas)e.widget);
      grafKontekst.drawLine(tacka.x, tacka.y, e.x, e.y);
      grafKontekst.dispose();
      updatePoint(e);
    }
    public void mouseDown(MouseEvent e) { azurirajTacku(e); }
    private void azurirajTacku(MouseEvent e) {
      tacka.x = e.x;
      tacka.y = e.y;
    }
  }
  public static void browserTab() {
    TabItem kartica = new TabItem(direktorijum, SWT.CLOSE);
    kartica.setText("čitač");
    kartica.setToolTipText("čitač Weba");
    Browser citac = null;
    try {
      citac = new Browser(direktorijum, SWT.NONE);
    } catch(SWTError e) {
      Label natpis = new Label(direktorijum, SWT.BORDER);
      natpis.setText("Neuspela inicijalizacija čitača");
      kartica.setControl(natpis);
    }
    if(citac != null) {
      citac.setUrl("http://www.mindview.net");
      kartica.setControl(citac);
    }
  }
  public static void main(String[] args) {
    SWTKonzola.run(new OknoSKarticama(), 800, 600);
  }
} ///:~
```

Ovde metoda **createContents( )** zadaje raspored i zatim poziva metode koje prave različite vrste kartica. Tekst na svakoj kartici zadaje se metodom **setText( )** (pored toga, na karticama možete praviti dugmad i crteže), a za svaku je zadat i njen priručni tekstualni opis. Na kraju svake metode poziva se **setControl( )** koja kontrolu napravljenu u metodi smešta u prostor dijaloga te kartice.

Metoda **labelTab( )** prikazuje jednostavan tekstualni natpis. Metoda **directoryDialogTab( )** sadrži dugme koje otvara standardan objekat tipa **DirectoryDialog**, u kojem korisnik bira direktorijum. Rezultat te metode se prikazuje kao tekst tog dugmeta.

Metoda **buttonTab( )** prikazuje raznu osnovnu dugmad. Metoda **sliderTab( )** ponavlja Swing primer iz prethodnog dela poglavlja u kojem se klizač veže s trakom za prikazivanje napredovanja.

Metoda **scribbleTab( )** šaljivo prikazuje grafičke mogućnosti. Program za crtanje je napravljen pomoću nekoliko redova koda.

Najzad, metoda **browserTab( )** prikazuje snagu SWT komponente **Browser** – to je potpuni čitač Weba u jednoj komponenti.

Grafika

Ovo je Swing program **Sinusoida.java** preveden u SWT:

```
//: swt/Sinusoida.java
// Swing program Sinusoida.java preveden u SWT.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class CrtajSinusoidu extends Canvas {
  private static final int FAKTORSKALE = 200;
  private int ciklusi;
  private int tacke;
  private double[] sinusi;
  private int[] tck;
  public CrtajSinusoidu(Composite roditelj, int stil) {
    super(roditelj, stil);
    addPaintListener(new PaintListener() {
      public void paintControl(PaintEvent e) {
        int maksimalnaSirina = getSize().x;
        double hkorak = (double)maksimalnaSirina / (double)tacke;
        int maksimalnaVisina = getSize().y;
        tck = new int[tacke];
        for(int i = 0; i < tacke; i++)
          tck[i] = (int)((sinusi[i] * maksimalnaVisina / 2 * .95)
            + (maksimalnaVisina / 2));
        e.grafKontekst.setForeground(
          e.prikaz.getSystemColor(SWT.COLOR_RED));
        for(int i = 1; i < tacke; i++) {
          int x1 = (int)((i - 1) * hkorak);
          int x2 = (int)(i * hkorak);
          int y1 = tck[i - 1];
          int y2 = tck[i];
          e.grafKontekst.drawLine(x1, y1, x2, y2);
        }
      }
    });
```

```
    setCycles(5);
  }
  public void zadajCikluse(int noviCiklusi) {
    ciklusi = noviCiklusi;
    tacke = FAKTORSKALE * ciklusi * 2;
    sinusi = new double[tacke];
    for(int i = 0; i < tacke; i++) {
      double radijani = (Math.PI / FAKTORSKALE) * i;
      sinusi[i] = Math.sin(radijani);
    }
    redraw();
  }
}

public class Sinusoida implements SWTAplikacija {
  private CrtajSinusoidu sinusi;
  private Slider klizac;
  public void createContents(Composite roditelj) {
    roditelj.setLayout(new GridLayout(1, true));
    sinusi = new CrtajSinusoidu(roditelj, SWT.NONE);
    sinusi.setLayoutData(
      new GridData(SWT.FILL, SWT.FILL, true, true));
    sinusi.setFocus();
    klizac = new Slider(roditelj, SWT.HORIZONTAL);
    klizac.setValues(5, 1, 30, 1, 1, 1);
    klizac.setLayoutData(
      new GridData(SWT.FILL, SWT.DEFAULT, true, false));
    klizac.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent dogadjaj) {
        sinusi.zadajCikluse(klizac.getSelection());
      }
    });
  }
  public static void main(String[] args) {
    SWTKonzola.run(new Sinusoida(), 700, 400);
  }
} ///:~
```

Kao što je **JPanel** u Swingu osnovna podloga (površina) za crtanje, tako je u SWT-u objekat tipa **Canvas**, tj. „platno" (engl. *canvas*).

Ako ovu verziju programa uporedite sa Swing verzijom, videćete da je metoda **CrtajSinusoidu** gotovo identična. U SWT-u se grafički kontekst **grafKontekst** dobija iz objekta događaja koji se prosleđuje prijemniku **PaintListener**, a u Swingu se objekat tipa Graphics neposredno prosleđuje metodi **paintComponent**( ). Ali su operacije koje se izvršavaju nad grafičkim objektom jednake, a identična je i metoda **zadajCikluse**( ).

Za metodu **createContents**( ) potrebno je nešto više koda nego za Swing verziju za raspoređivanje komponenata i pripremu klizača i njegovog prijemnika, ali su osnovne operacije koje se izvršavaju opet približno jednake.

Paralelno izvršavanje u SWT-u

Iako se AWT/Swing izvršava u jednoj niti, to je lako narušiti i proizvesti program koji se izvršava nedeterministički. U suštini, prikaz ne smete da ispisujete pomoću više niti, inače će one početi da se sudaraju na neverovatne načine.

SWT to ne dozvoljava – baciće izuzetak kada prikaz pokušate da ispišete pomoću više niti. Time se sprečava da neiskusni programeri nenamerno naprave tu grešku i tako u program unesu „bubicu" koja se teško otkriva.

Ovo je Swing program **ObojeneKutije.java** preveden u SWT:

```
//: swt/ObojeneKutije.java
// SWT prevod Swing programa ObojeneKutije.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ObKut extends Canvas implements Runnable {
  class PrijemnikZaIscrtavanjeObKut implements PaintListener {
    public void paintControl(PaintEvent e) {
      Color boja = new Color(e.prikaz, bBoja);
      e.grafKontekst.setBackground(boja);
      Point velicina = getSize();
      e.grafKontekst.fillRectangle(0, 0, velicina.x, velicina.y);
      boja.dispose();
    }
  }
  private static Random pseudoSlucajanBroj = new Random();
  private static RGB newColor() {
    return new RGB(pseudoSlucajanBroj.nextInt(255),
      pseudoSlucajanBroj.nextInt(255), pseudoSlucajanBroj.nextInt(255));
  }
  private int pauza;
  private RGB bBoja = newColor();
  public ObKut(Composite roditelj, int pauza) {
    super(roditelj, SWT.NONE);
    this.pauza = pauza;
    addPaintListener(new PrijemnikZaIscrtavanjeObKut());
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        bBoja = newColor();
```

```
          getDisplay().asyncExec(new Runnable() {
            public void run() {
              try { redraw(); } catch(SWTException e) {}
              // Izuzetak SWTException ne smeta kada je roditelj
              // okončan na nižem nivou izvršavanja.
            }
          });
          TimeUnit.MILLISECONDS.sleep(pauza);
        }
      } catch(InterruptedException e) {
        // Prihvatljiv način izlaska
      } catch(SWTException e) {
        // Prihvatljiv način izlaska: roditelj
        // okončan na nižem nivou izvršavanja.
      }
    }
  }

  public class ObojeneKutije implements SWTAplikacija {
    private int brojcelija = 12;
    private int pauza = 50;
    public void createContents(Composite roditelj) {
      GridLayout tabela = new GridLayout(brojcelija, true);
      tabela.horizontalSpacing = 0;
      tabela.verticalSpacing = 0;
      roditelj.setLayout(tabela);
      ExecutorService exec = new DaemonThreadPoolExecutor();
      for(int i = 0; i < (brojcelija * brojcelija); i++) {
        final ObKut ok = new ObKut(roditelj, pauza);
        ok.setLayoutData(new GridData(GridData.FILL_BOTH));
        exec.execute(ok);
      }
    }
    public static void main(String[] args) {
      ObojeneKutije kutije = new ObojeneKutije();
      if(args.length > 0)
        kutije.brojcelija = new Integer(args);
      if(args.length > 1)
        kutije.pauza = new Integer(args);
      SWTKonzola.run(kutije, 500, 400);
    }
  } ///:~
```

Kao u prethodnom primeru, iscrtavanjem se upravlja tako što se napravi objekat tipa PaintListener s metodom **paintControl**( ) koja se poziva kada je SWT nit spremna da iscrta komponentu. Taj prijemnik tipa **PaintListener** prijavljuje se (registruje) u konstruktoru klase **ObKut**.

U ovoj verziji programa **ObKut** znatno se razlikuje metoda **run( )** koja metodu za iscrtavanje **redraw( )** ne može da pozove neposredno, nego mora da je pošalje metodi **asyncExec( )** objekta tipa **Display**, što radi približno kao metoda **SwingUtilities.invokeLater( )**. Ukoliko takav poziv zamenite neposrednim pozivom metode **redraw( )**, videćete da će program stati.

Kada pokrenete prethodni program, videćete da se u prozoru povremeno pojavljuju nekakve male horizontalne linije. To je posledica činjenice da SWT podrazumevano *nije* dvostruko baferisan kao Swing. Primetićete to jasnije kada Swing verziju pokrenete naporedo sa SWT verzijom. Kôd za dvostruko baferisanje SWT-a možete sami da napišete; primere za to naći ćete na Web lokaciji *www.eclipse.org*.

Vežba 41: (4) Izmenite **swt/ObojeneKutije.java** tako da počinje prskanjem tačaka („zvezdica") po platnu, kojima se potom boje menjaju nasumično.

Poređenje SWT-a i Swinga

Iz ovako kratkog uvoda teško je steći potpunu sliku, ali trebalo bi da barem počnete da uviđate kako SWT u mnogim situacijama omogućuje pisanje jednostavnijeg koda nego Swing. Međutim, programiranje grafičkih korisničkih okruženja u SWT-u ipak je složeno, pa bi razlozi za korišćenje SWT-a verovatno trebalo da budu sledeći: prvo, omogućiti korisniku transparentniji doživljaj prilikom korišćenja vaše aplikacije (pošto ona izgleda/ponaša se kao ostale aplikacije na platformi), i drugo, ukoliko je važna brzina reagovanja koju SWT donosi. U protivnom, odgovarajući izbor je verovatno Swing.

Vežba 42: (6) Izaberite neki od Swing primera koji u ovom odeljku nije bio preveden i prevedite ga u SWT. (Napomena: ovo je dobar domaći zadatak za ceo razred, pošto vodič *ne* sadrži rešenja.)

Sažetak

Od svih biblioteka u Javi, biblioteka grafičkih okruženja pretrpela je najveće izmene. AWT iz Jave 1.0 često je kritikovan kao jedan od najgorih projekata koji su se ikada pojavili, i mada je omogućavao izradu prenosivih programa, dobijeno grafičko korisničko okruženje bilo je „podjednako osrednje na svim platformama". Bilo je ograničenih mogućnosti i nezgrapno, a koristilo se teže nego matične alatke za razvoj aplikacija dostupne za razne platforme.

Kada su u Javu 1.1 uvedeni nov model događaja i zrna Jave, stanje je poboljšano: sada je moguće napraviti komponente grafičkog okruženja koje se lako prevlače unutar alatki za vizuelno pravljenje aplikacija. Osim toga, model događaja i zrna jasno je pokazao kako se vodilo računa o lakoći programiranja i održavanja koda (što nije bilo očigledno u biblioteci AWT Jave 1.0). Međutim, tek kada su se pojavile klase JFC/Swing, posao je okončan. Uz komponente biblioteke Swing, programiranje grafičkih okruženja prenosivih na razne platforme postalo je relativno bezbolno.

Prava revolucija odigrala se u oblasti alatki za pravljenje aplikacija. Ako želite da se poboljša neka komercijalna alatka za pravljenje aplikacija, morate da držite palčeve i nadate

se da će vam autori okruženja dati ono što želite. Ipak, Java je otvoreno okruženje, što znači da dozvoljava konkurentska okruženja za pravljenje aplikacija, i podržava takav pristup. Da bi te alatke iko uzeo u obzir, one moraju da podržavaju Javina zrna. To podrazumeva otvoreno igralište: ako se pojavi bolja alatka za pravljenje aplikacija, više nećete morati da koristite onu koju ste dotad koristili, već možete da pređete na novu i time povećate produktivnost. Ovakva vrsta konkurentskog okruženja za vizuelne alatke ranije nije postojala, a tržište koje se na ovaj način bude formiralo može dati samo pozitivne rezultate u pogledu produktivnosti programera.

Ovo poglavlje trebalo je da vas uvede u osnove programiranja GKO i da vam prikaže osnovne tehnike za prilično lako pronalaženje sopstvenog puta kroz biblioteku. Ono što ste dosad videli verovatno će zadovoljiti dobar deo vaših potreba pri projektovanju korisničkih okruženja. Međutim, Swing, SWT i Flash/Flex mogu i mnogo više od toga, jer sadrže potpun asortiman alatki za grafička okruženja. Verovatno postoji način da postignete gotovo sve što možete da zamislite.

Resursi

Mrežne prezentacije Bena Galbraitha na adresi *www.galbraiths.org/presentations* lepo objašnjavaju i Swing i SWT.

Rešenja odabranih vežbi data su u elektronskom dokumentu *The Thinking in Java Annotated Solution Guide*, koji se može kupiti na lokaciji *www.MindView.com*.

A: Dodaci

Ova knjiga ima više dodataka, među kojima su i sadržaji, seminari i usluge dostupni na Web lokaciji MindView.

TE DOPUNE SU OPISANE U OVOM DODATKU KAKO BISTE MOGLI DA PROCENITE DA LI VAM one mogu koristiti.

Obratite pažnju na to da se seminari obično drže javno, ali mogu biti držani i privatno, na vašoj lokaciji, samo za vaše osoblje.

Dodaci koji se mogu preuzeti sa Interneta

Kôd za ovu knjigu možete preuzeti na adresi *www.MindView.net*. Obuhvaćene su i Ant build datoteke i druge datoteke za podršku neophodne za uspešno automatizovano prevođenje i pakovanje programa i izvršavanje primera iz ove knjige.

Sem toga, neki delovi knjige prebačeni su u elektronski oblik. U njima su obrađene teme:

- Kloniranje objekata
- Prosleđivanje i vraćanje objekata
- Analiza i projektovanje
- Delovi drugih poglavlja iz trećeg izdanja knjige *Misliti na Javi* koji nisu bili dovoljno relevantni da bi bili objavljeni i u četvrtom izdanju.

Misliti na jeziku C: osnova za Javu

Na lokaciji *www.MindView.net* možete besplatno preuzeti seminar *Thinking in C*. Prezentaciju je smislio Chuck Allison, a razvila kompanija MindView. Radi se o multimedijalnom Flash kursu koji daje uvod u sintaksu, operatore i funkcije jezika C, na osnovu kojih je napravljena Java.

Vodite računa o tome da na računaru na kojem želite da pokrenete seminar *Thinking in C* morate imati instaliran Flash Player (koji možete besplatno preuzeti na lokaciji *www.Adobe.com*).

Seminar *Thinking in Java*

Moja kompanija (MindView, Inc.) drži petodnevne javne ili privatne seminare za obuku kroz neposredno iskustvo, čiju osnovu čini materijal ove knjige. Ovaj seminar (čije je prethodno ime bilo *Hands-On Java*) naš je glavni uvodni seminar i osnova za naprednije seminare. Svaku lekciju čini izabrani materijal iz pojedinog poglavlja. Zatim sledi period vežbanja pod nadzorom, kada se svakom studentu pažnja posvećuje pojedinačno. Na lokaciji *www.MindView.net* pronaći ćete informacije o rasporedu i mestu održavanja, svedočenja bivših polaznika i razne druge pojedinosti.

CD sa seminarom *Hands-On Java*

CD *Hands-On Java* napravljen je na osnovu ove knjige, a sadrži proširenu verziju materijala sa seminara *Thinking in Java.* Barem ćete delimično iskusiti kako je biti na seminaru a pri tom nećete putovati ni plaćati kotizaciju. Svako poglavlje ove knjige predstavljeno je na CD-u odgovarajućim zvučnim zapisom jednog predavanja i pratećim slajdovima. Ja sam osmislio seminar i ja izgovaram tekst predavanja na CD-u. Materijal je u formatu Flash i može biti reprodukovan na svakoj platformi koja podržava Flash Player. CD *Hands-On Java* možete kupiti na lokaciji *www.MindView.net,* a tamo su i probni primeri koji se mogu besplatno preuzeti.

Seminar *Thinking in Objects*

Ovaj seminar predstavlja objektno orijentisano programiranje viđeno očima projektanta. Ispituje se postupak razvoja i izgradnje sistema, prvenstveno „brzim" ili „lakim" metodama, naročito metodama Ekstremnog programiranja (XP). Predstavljam metodologije uopšte, male alatke kao što je tehnika planiranja pomoću „indeksnih kartica" (opisna u knjizi *Planning Extreme Programming* Becka i Fowlera, Addison-Wesley, 2001), CRC kartice za projektovanje objekata, programiranje parova, planiranje iteracija, testiranje jedinica, automatizovanu izgradnju programa, kontrolu izvornog koda i slične teme. Kurs obuhvata i jedan XP projekat koji razvijamo nedelju dana.

Ako krećete u neki projekat i želeli biste da upotrebljavate objektno orijentisane tehnike projektovanja, možemo upotrebiti vaš projekat kao primer i do kraja sedmice napraviti prvu grubu verziju rešenja.

Na lokaciji *www.MindView.net* pronaći ćete informacije o rasporedu i mestu održavanja, svedočenja bivših polaznika i razne druge pojedinosti.

Thinking in Enterprise Java

Ova knjiga je nastala od nekih naprednijih poglavlja iz ranijih izdanja knjige *Misliti na Javi.* To nije drugi tom knjige *Misliti na Javi,* nego fokusirano objašnjenje naprednijih tema programiranja za poslovne primene. Trenutno je dostupna (u nekom obliku, verovatno se još razvija) kao besplatan elektronski dokument koji se može preuzeti na lokaciji *www.MindView.net.* Pošto se radi o zasebnoj knjizi, ona se širi kad god treba da obuhvati neku novu temu. Svrha je ista kao ona knjige *Misliti na Javi*: dati veoma razumljiv uvod u osnovne tehnologije programiranja za poslovne primene, da bi čitalac bio spreman za naprednije obrade tih tema.

Među obrađenim temama biće i:

- Uvod u programiranje za poslovne primene
- Mrežno programiranje pomoću utičnica i kanala
- Daljinsko povezivanje metoda (RMI)
- Povezivanje s bazama podataka
- Usluge imenovanja i usluge imenika
- Servleti

- Javine serverske stranice (JSP)
- Oznake, JSP fragmenti i jezik za izraze
- Automatizovanje pravljenja korisničkih okruženja
- Zrna Jave u poslovnim primenama
- XML
- Web usluge
- Automatsko testiranje

Tekući sadržaj knjige *Thinking in Enterprise Java* možete preuzeti na lokaciji *www.MindView.net.*

Thinking in Patterns (with Java)

Jedan od najvažnijih doprinosa u projektovanju objektno orijentisanih programa jesu „projektni obrasci" čiji je razvoj hronološki opisan u knjizi *Design Patterns*, koju su napisali Gamma, Helm, Johnson i Vlissides (Addison-Wesley, 1995). Ta knjiga opisuje 23 opšte klase problema i njihova programska rešenja, napisana prvenstveno na C++-u. Knjiga *Design Patterns* je postala ključni, gotovo obavezan izvor OOP programera. Knjiga *Thinking in Patterns* predstavlja osnovne pojmove projektnih obrazaca na primerima iz Jave. To nije puki prevod knjige *Design Patterns* na Javu, već jedna nova perspektiva data sa stanovišta Jave. Ne bavi se samo sa 23 klasična obrasca, nego, po potrebi, i drugim idejama i tehnikama za rešavanje problema.

Nastala je kao poslednje poglavlje prvog izdanja knjige *Misliti na Javi.* Kako su se razvijale te ideje, postalo je jasno da je za njih potrebna zasebna knjiga. U vreme pisanja ovog teksta ona se još razvija, ali je materijal iz nje obrađivan i prerađivan u mnogim prezentacijama seminara *Objects & Patterns* (koji je sada podeljen u seminare *Designing Objects & Systems* i *Thinking in Patterns*).

Više informacija o toj knjizi potražite na adresi *www.MindView.net.*

Seminar *Thinking in Patterns*

Ovaj seminar se razvio iz seminara *Objects & Patterns* (*Objekti & obrasci*), koji smo Bill Venners i ja držali u nekoliko proteklih godina. On je postao preopširan, pa smo ga podelili na dva: ovaj i prethodno spomenuti *Designing Objects & Systems* (*Projektovanje objekata i sistema*).

Seminar se strogo drži materijala i načina predstavljanja materije iz knjige *Thinking in Patterns*, pa ćete sadržaj seminara najbolje upoznati iz te knjige, koju možete preuzeti na lokaciji *www.MindView.net.*

U dobrom delu prezentacije naglašava se proces evolucije projekta – od početnog rešenja, preko logike i evolucije rešenja do prikladnijeg dizajna. Poslednji prikazani projekat (simulacija reciklaže smeća) evoluirao je tokom vremena, i njegov razvoj može da posluži kao prototip za način na koji vaš dizajn može da nastane: kao adekvatno rešenje određenog problema, koje evoluira u prilagodljivo rešenje cele klase problema.

Ovaj seminar će vam pomoći da:

- značajno povećate prilagodljivost svojih projekata;
- dizajnirate projekat tako da bude proširiv i višekratno upotrebljiv;
- ostvarite bolju komunikaciju o projektima tako što ćete koristiti jezik obrazaca.

Nakon svake lekcije sledi niz vežbi koje rešavate pomoću obrazaca. Bićete vođeni tokom pisanja programa u kojima se konkretni obrasci primenjuju kao rešenja programskih problema.

Na lokaciji *www.MindView.net* možete pronaći informacije o rasporedu i mestu održavanja, svedočenja bivših polaznika i razne druge pojedinosti.

Konsultacije i revizije dizajna

Moja kompanija pruža i usluge konsaltinga, mentorskog rada, te revizije dizajna i realizacija koje olakšavaju vođenje projekta kroz ceo njegov ciklus razvoja; sve to možete dobiti i za prvi Java projekat vašeg preduzeća. Više informacija o pojedinostima i vremenskoj dostupnosti tih usluga potražite na adresi *www.MindView.net*.

B: Resursi

Softver

Razvojni programski paket za Javu (engl. *Java Development Kit, JDK*) može se preuzeti s lokacije *http://java.sun.com.* Čak i ako odlučite da koristite razvojno okruženje drugih proizvođača, uvek je dobro imati JDK pri ruci ako naiđete na nešto što bi moglo da bude greška prevodioca. JDK je standard, pa su greške u prevodiocu tog paketa uglavnom poznate.

HTML dokumentacija razvojnog programskog paketa za Javu na lokaciji *http://java.sun.com.* Još nisam pronašao referentni priručnik o standardnim Javinim bibliotekama koji nije bio zastareo ili onaj koji je sadržao dovoljnu količinu informacija. Iako je HTML dokumentacija kompanije Sun bušna, ima grešaka i ponekad je toliko štura da je potpuno neupotrebljiva, barem *sadrži* sve klase i metode. Ljudi često u početku imaju otpor prema korišćenju Web resursa jer više vole štampanu knjigu, ali vredi uložiti malo truda pa to prevazići i naviknuti se na HTML dokumente, barem zato što je pregled potpuniji nego u bilo kojoj knjizi. Tek ako ne možete da shvatite o čemu se radi u uputstvu na Webu, potražite štampane knjige.

Programi za uređivanje teksta i alatke za pravljenje aplikacija

U ovoj oblasti vlada zdrava konkurencija. Mnogo toga je besplatno (a ono što nije, najčešće se može besplatno isprobati), pa je najbolje da sami isprobate više alatki i nađete onu koju vam odgovara. Evo nekih od njih:

JEdit je besplatan program za uređivanje teksta čiji je autor Slava Pestov. Napisan je na Javi, pa je dodatna korist to što možete videti na delu ovu Java aplikaciju za stone računare. Ovaj program za uređivanje teksta ima mnogo softverskih dodataka, od kojih su najveći deo napisali članovi zajednice korisnika Jave. Preuzmite sa adrese *www.jedit.org.*

NetBeans, Sunova besplatna alatka za pravljenje aplikacija koja se može preuzeti sa *www.netbeans.org.* Projektovana za pravljenje GKO prevlačenjem komponenata, za uređivanje koda, otkrivanje i otklanjanje grešaka itd.

Eclipse, projekat otvorenog izvornog koda koji podržava IBM, među ostalima. Platforma Eclipse je osnova koja se može proširivati, pa povrh nje možete praviti sopstvene samostalne aplikacije. Deo tog projekta je biblioteka SWT opisana u poglavlju *Grafička korisnička okruženja.* Preuzmite sa adrese *www.Eclipse.org.*

IDEA kompanije **IntelliJ**, omiljen komercijalni proizvod većine programera na Javi, od kojih mnogi tvrde da je IDEA uvek korak-dva ispred Eclipsea, možda i zato što IntelliJ ne stvara i alatku za pravljenje aplikacija i platformu za razvoj, nego samo ono prvo. Probnu verziju možete besplatno preuzeti sa adrese *www.jetbrains.com.*

Knjige

Efikasno programiranje na Javi, Joshua Bloch (Mikro knjiga, 2004). Obavezna lektira. Napisao ju je čovek koji je popravio Javinu biblioteku kolekcija. Pisano po uzoru na klasik *Effective C++* koji je napisao Scott Meyer.

Core Java 2, sedmo izdanje, Horstmann & Cornell, u dva toma (Prentice Hall, 2005). Velika, sveobuhvatna, prva knjiga koju uzmem u ruke kada mi zatrebaju odgovori. Preporučujem vam ovu knjigu kada pročitate *Misliti na Javi* i poželite da pređete na viši nivo.

The Java Class Libraries: An Annotated Reference, Patrick Chan i Rosana Lee (Addison-Wesley, 1997). Nažalost, zastarela je i rasprodata. Takva je *trebalo* da bude Sunova HTML dokumentacija razvojnog programskog paketa za Javu: sa dovoljno opisa da bude upotrebljiva. Obimna je, skupa, a ni kvalitet primera me ne zadovoljava. *Pa ipak*, može vam poslužiti kada zapadnete u nevolje, a izgleda da su navedena detaljnija objašnjenja nego u većini drugih knjiga. Međutim, *Core Java 2* ima ažurnija objašnjenja mnogih komponenata biblioteka.

Java Network Programming, drugo izdanje, Eliotte Rusty Harold (O'Reilly, 2000). Nisam razumeo umrežavanje u Javi (niti umrežavanje uopšte, što se toga tiče) sve dok nisam pronašao ovu knjigu. Smatram i da Web lokacija autora knjige, Café au Lait, pruža stimulativan, stručan i aktuelan pogled na poboljšanja u Javi, neopterećen obavezama prema bilo kom proizvođaču. Zbog redovnih ažuriranja, lokacija je u toku s novinama u Javi koje se javljaju veoma brzo. Posetite lokaciju *www.cafeaulait.org*.

Design Patterns, Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). Originalna knjiga koja je započela razmatranje projektnih obrazaca i koja se spominje na više mesta u ovoj knjizi.

Refactoring to Patterns, Joshua Kerievsky (Addison-Wesley, 2005). Spaja preradu programa („ponovnu podelu na proste faktore") i projektne obrasce. Najvredniji aspekt ove knjige jeste to što pokazuje kako dalje razviti projekat korišćenjem projektnih obrazaca po potrebi.

The Art of UNIX Programming, Eric Raymond (Addison-Wesley, 2004). Iako je Java jezik za pisanje programa koji se izvršavaju na svim platformama, Unix/Linux treba znati zbog dominacije Jave na serverima. Ericova knjiga je odličan uvod u istoriju i filozofiju ovog operativnog sistema. Zanimljivo štivo i za one koji samo žele da saznaju nešto o korenima računarstva.

Analiza i projektovanje

Extreme Programming Explained, drugo izdanje, Kent Beck uz pomoć Cynthiae Andres (Addison-Wesley, 2005). Oduvek sam osećao da bi morao postojati mnogo drugačiji, mnogo bolji postupak razvijanja programa, i mislim da mu je XP prišao veoma blizu. Jedina knjiga koja je ostavila sličan utisak na mene bila je *Peopleware* (opisana u nastavku), koja se pretežno bavi okruženjem i opstankom u poslovnom svetu. *Extreme Programming Explained* govori o programiranju i posmatra mnogo toga, čak i najnovija dostignuća, iz

tog ugla. Autori čak tvrde da su slike u redu sve dok ne provodite mnogo vremena gledajući ih i spremni su da ih izbace. (Primetićete da se na knjizi *ne* nalazi „pečat UML".) Odlučivao sam se da li ću raditi za neku kompaniju isključivo na osnovu toga da li koriste XP. Mala knjiga, kratka poglavlja koja se lako čitaju, a teraju na razmišljanje. Počećete da zamišljate kako radite u takvoj atmosferi i otvoriće vam se novi vidici.

UML ukratko, Martin Fowler (Mikro knjiga, 2004). Kada se prvi put sretnete s jezikom UML, uplašićete se jer ima mnogo dijagrama i detalja. Prema Fowleru, veći deo toga je nepotreban i on će vam objasniti ono osnovno. Za većinu projekata potrebno je da poznajete samo nekoliko alatki za crtanje dijagrama, a Fowlerov cilj je da dobije dobar program, a ne da brine o tome kako je stigao do njega. Lepa, tanka, čitljiva knjiga; prva koju treba da nabavite ako želite da razumete UML.

Domain-Driven Design, Eric Evans (Addison-Wesley, 2004). Bavi se *modelom domena* kao prvenstvenim obeležjem postupka projektovanja. Na osnovu sopstvenog iskustva uvideo sam da je to premeštanje naglaska važno, jer pomaže programerima da ostanu na pravom nivou apstrakcije.

The Unified Software Development Process, Ivar Jacobsen, Grady Booch i James Rumbaugh (Addison-Wesley, 1999). Bio sam potpuno siguran da mi se ova knjiga neće svideti. Izgledalo je kao da ima sva obeležja dosadnog fakultetskog udžbenika. Ali, prijatno sam se iznenadio: na vrlo malo mesta sreću se objašnjenja za koja bi se moglo pomisliti da ni autorima nisu jasna. Veći deo knjige ne samo da je jasan, već je i prijatan. Najbolje od svega je to što su objašnjenja prilično praktična. To ipak nije *Extreme Programming* (i nije toliko jasna u pogledu testiranja), ali je takođe deo UML menažerije: čak i ako vam se ne sviđa XP, većina programera je sada usvojila stav „UML je dobar" (bez obzira na njihov *stvarni* nivo iskustva s tom metodom). Na kraju, i vi ćete morati da se ukrcate u taj voz. Mislim da je ova knjiga najbolja za proučavanje UML-a, i trebalo bi da je pročitate ako posle Fowlerove knjige *UML ukratko* poželite da saznate više detalja.

Pre nego što se odlučite za bilo koju metodu, korisno je da se posavetujete s nekim ko ne želi da vas ubedi da je koristite. Često se dešava da se neka metoda usvoji, a da se pri tom ne razume zaista šta se želi od nje ili koje će prednosti ostvariti. Drugi je koriste, što izgleda kao dovoljno privlačan razlog. Ljudi imaju jednu čudnu osobinu: ako žele da veruju kako će nešto rešiti njihove probleme, oni će to i probati. (To je eksperimentisanje, što je dobro.) Međutim, ako ne reše svoje probleme, možda će udvostručiti napore i na sva usta objaviti kako su otkrili nešto sjajno. (To je nepriznavanje stvarnosti, što nije dobro.) Možda je pouka da nećete biti usamljeni, ako nagovorite druge ljude da se ukrcaju na isti brod, čak i ako taj brod ne ide nikuda (ili tone).

Ovo ne znači da nijedna metodologija nikuda ne vodi, već da treba da budete opremljeni mentalnim alatkama koje će vam pomoći da ostanete na nivou eksperimentisanja („Ovo ne radi: hajde da probamo nešto drugo."), a da izbegavate odricanje („To zapravo i nije problem. Sve je divno, ne moramo ništa da menjamo"). Mislim da će vam sledeće knjige, ako ih pročitate *pre* nego što izaberete neku metodu, obezbediti te alatke.

Software Creativity, Robert Glass (Prentice Hall, 1995). Ovo je najbolja knjiga koja razmatra *perspektivu* celog problema metodologije. To je zbirka kratkih radova koje je Glass pisao, ponekad i dobijao od drugih (jedan od saradnika je P. J. Plauger), a koji izražavaju njegovo dugogodišnje razmišljanje i proučavanje ove teme. Dovoljno su zabavni,

a dugački samo toliko koliko je potrebno da kažu ono što je neophodno. Takođe, radovi nisu samo šuplja priča; nude na stotine referenci na druge radove i istraživanja. Svi programeri i menadžeri trebalo bi da pročitaju ovu knjigu pre nego što se zapute na klizav teren metodologije.

Software Runaways: Monumental Software Disasters, Robert Glass (Prentice Hall, 1997). Odlična osobina ove knjige je ta što naglas govori o onome o čemu se ne priča: koliko projekata ne samo da je propalo, već je propalo spektakularno. Ustanovio sam da većina programera misli: „To meni ne može da se desi" (ili „Ne može da se desi *ponovo*") i smatram da to nije dobro. Ako imate na umu da nešto uvek može da krene naopako, u mnogo ste boljem položaju da učinite da sve radi.

Peopleware, drugo izdanje, Tom DeMarco i Timothy Lister (Dorset House, 1999). Ovu knjigu *morate* da pročitate. Ne samo da je zabavna, ona će uzdrmati svet u kojem prebivate i uništiti pretpostavke na kojima počiva. Iako su autori projektanti softvera, ova knjiga je posvećena projektima i timovima uopšte. Međutim, istaknuta je važnost *ljudi* i njihovih potreba, a ne tehnologije i njenih potreba. Autori govore o pravljenju okruženja u kome će ljudi biti zadovoljni i produktivni, a ne donose pravila koja bi ljudi trebalo da poštuju kako bi bili odgovarajući delovi mašine. Mislim da je ovaj stav najviše doprineo tome se programeri podsmevaju usvajanju metode XYZ i da nastave da rade kako su oduvek radili.

Secrets of Consulting: A Guide to Giving & Getting Advice Successfully, Gerald M. Weinberg (Dorste House, 1985). Izvanredna knjiga, jedna od najboljih koju sam pročitao. Savršena je ukoliko pokušavate da radite kao konsultant ili plaćate konsultante i niste zadovoljni njihovim rezultatima. Iz kratkih poglavlja ispunjenih pričama i anegdotama, učite kako doći do suštine uz najmanje napora. Pročitajte i nastavak *More Secrets of Consulting* objavljen 2002. ili gotovo bilo koju drugu Weinbergovu knjigu.

Complexity, M. Mitchell Waldrop (Simon & Schuster, 1992). Knjiga opisuje sastanke grupe naučnika različitih struka u gradu Santa Fe u Meksiku, koji su raspravljali o različitim problemima što u njihovim disciplinama nisu rešeni (berza u ekonomiji, nastanak života u biologiji, zašto ljudi rade to što rade u sociologiji itd.). Suočavanjem gledišta fizike, ekonomije, hemije, matematike, računarstva, sociologije i drugih nauka, razvijen je multidisciplinarni pristup tim problemima. Međutim, važnije je bilo to što se počelo na drugi način *razmišljati* o pomenutim, veoma složenim problemima: udaljavanje od matematičkog determinizma i iluzije da se može napisati jednačina koja predviđa ponašanje, i približavanje stavu da treba prvo *posmatrati* i tražiti neku pravilnost, a potom je oponašati na sve moguće načine. (Knjiga dokumentuje, na primer, pojavljivanje genetskih algoritama.) Smatram da je ovakav način razmišljanja koristan jer ukazuje na načine upravljanja sve složenijim softverskim projektima.

Jezik Python

Learning Python, drugo izdanje, Mark Lutz i David Ascher (O'Reilly, 2003). Lep uvod za programere na mom omiljenom jeziku, odličnom pratiocu Jave. Knjiga sadrži i kratak uvod u Jython koji omogućuje kombinovanje Jave i Pythona u istom programu (interpretator jezika Jython pravi čist Javin bajtkod, pa vam ne treba ništa posebno da biste to postigli). Ujedinjenje ova dva jezika obećava velike mogućnosti.

Spisak mojih knjiga

Nabrojane su po redosledu objavljivanja, ali su neke rasprodate.

Computer Interfacing with Pascal & C (objavljena samostalno u kući Eisys imprint, 1988. Može se naći samo na lokaciji *www.MindView.net*). Uvod u elektroniku iz vremena kada je CP/M još uvek bio kralj, a DOS početnik. Koristio sam jezike visokog nivoa, a često i paralelni priključak računara za izvođenje raznih projekata u elektronici. Knjiga je sastavljena od mojih kolumni iz *Micro Cornucopiae*, prvog i najboljeg časopisa za koji sam pisao. Nažalost, Micro C je ugašen mnogo pre nego što se pojavio Internet. Pisanje ove knjige za mene je bilo veoma lepo iskustvo.

Using C++ (Osborne/McGraw-Hill, 1989). Jedna od prvih knjiga o jeziku C++. Više se ne štampa i zamenjena je drugim izdanjem, čije je ime *C++ Inside & Out*.

C++ Inside & Out (Osborne/McGraw-Hill, 1993). Kao što sam rekao, ovo je zapravo drugo izdanje knjige *Using C++*. Jezik C++ je u ovoj knjizi dosta tačno opisan, ali je ona iz 1992. godine, pa je knjiga *Thinking in C++* napisana da bi je zamenila. Više o ovoj knjizi možete da saznate na lokaciji *www.MindView.net*, odakle možete da preuzmete i izvorni kôd.

Thinking in C++, prvo izdanje (Prentice Hall, 1995). Od časopisa *Software Development Magazine* dobila nagradu za knjigu – podstrek godine.

Thinking in C++, drugo izdanje, Volume I (Prentice Hall, 2000). Može se preuzeti s lokacije *www.MindView.net*. Ažurirana u skladu s konačnim standardom jezika. Prevod na srpski objavila je Mikro knjiga, 2003.

Thinking in C++, drugo izdanje, Volume II, pisano zajedno sa Chuckom Alissonom (Prentice Hall, 2003). Može se preuzeti s lokacije *www.MindView.net*.

Black Belt C++, the Master's Collection, Bruce Eckel, urednik (M&T Books, 1994). Rasprodata. Zbirka radova raznih stručnjaka za C++ zasnovana na njihovim prezentacijama na konferenciji Software Development, kojom sam predsedavao. Omot ove knjige me je naterao da ubuduće kontrolišem sve korice svojih knjiga.

Misliti na Javi, prvo izdanje (Prentice Hall, 1998). Prvo izdanje ove knjige osvojilo je nagradu za produktivnost od časopisa *Software Development Magazine*, nagradu za najbolju knjigu po izboru urednika časopisa *Java Developer's Journal* i nagradu za najbolju knjigu po izboru čitalaca časopisa *JavaWorld*. Može se preuzeti s lokacije *www.MindView.net*.

Misliti na Javi, drugo izdanje (Prentice Hall, 2000, Mikro knjiga, 2002). Ovo izdanje je osvojilo nagradu za najbolju knjigu po izboru urednika časopisa *JavaWorld*. Može se preuzeti s lokacije *www.MindView.net*.

Misliti na Javi, treće izdanje (Prentice Hall, 2003). Ovo izdanje je osvojilo nagradu časopisa *Software Development Magazine* za knjigu – podsticaj godine, kao i druge nagrade navedene na zadnjoj korici. Može se preuzeti s lokacije *www.MindView.net*.

Spisak termina korišćenih u knjizi

alatka za obradu anotacija	*annotation processing tool, apt*
alijas, pseudonim	*alias*
analizator programskih jezika	*scanner*
automatizovano prevođenje i pakovanje, izgradnja programa	*build*
automatsko pakovanje	*autoboxing*
bacanje izuzetka, generisanje izuzetka	*throwing an exception*
bit indikator	*bit flag*
blok za obradu izuzetaka	*exception handler*
blok za pozive	*invocation handler*
brava	*lock*
cev	*pipe*
curenje memorije	*memory leak*
čitač klasa	*class browser*
čuvana oblast	*guarded region*
daljinsko pozivanje metoda	*Remote Method Invocation, RMI*
datoteka preslikana u memoriju	*memory-mapped file*
deljena brava	*shared lock*
dinamička oblast memorije	*heap*
dinamičko vezivanje	*dynamic binding*
dnevnik, zapisnik	*log*
dodeljiv	*assignable*
dokumentaciona oznaka	*doc tag*
dvostrani red za čekanje	*double-ended queue, deque*
dvrednost	*rvalue*
eksplicitna konverzija	*casting*
element, spravica	*widget*
fantomska referenca	*phantom reference*
generisanje izuzetka, bacanje izuzetka	*throwing an exception*
granica	*bound*
heširanje, transformisanje ključa	*hashing*
identifikacija tipa u vreme izvršavanja	*run-time type identification, RTTI*
identifikator	*handle*
imenski prostor	*namespace*
informacija o tipovima prilikom izvršavanja	*run-time type information, RTTI*
integrisano razvojno okruženje	*Integrated Development Environment, IDE*
isključiva brava	*exclusive lock*
ispis steka	*stack trace*
ispitni blok	*try block*
ispreturano	*shuffled*
izgradnja programa, automatizovano prevođenje i pakovanje	*build*
izuzetak	*exception*
izuzetak tokom izvršavanja	*runtime exception*
izvršilac	*executor*
jedinica za prevođenje	*compilation unit, translation unit*
jedinično testiranje	*unit testing*
jednokratno otkrivanje tipa	*single dispatching*
kasno vezivanje	*late binding*
klizač	*slider*
ključ za heširanje	*hash code*
komplet alatki za apstraktne prozore	*Abstract Window Toolkit, AWT*
konstruktor	*constructor*
konstruktor bez argumenata	*no-arg constructor*
konverzija hvatanjem	*capture conversion*
kooperativno višenitno programiranje	*cooperative multithreading*
kostur programa, struktura programa	*application framework*
lak objekat	*light-weight object*
laka trajnost	*lightweight persistence*
lažni objekat	*mock object*
leksema	*token*
lenja procena	*lazy evaluation*
literal klase	*class literal*
lokalna metoda	*native method*
mehanizam brzog otkazivanja	*fail-fast*
meka referenca	*soft reference*
miksin	*mixin*
nabrojani tip	*enumerated type*
najdavnije korišćenje	*least-recently-used, LRU*
nasleđivanje	*inheritance*
nastavljanje	*resumption*
natklasa	*superclass*
neprovereni izuzetak	*unchecked exception*
nit	*thread*
nit izvršavanja	*thread of execution*
nit za otpremu događaja	*event dispatch thread*
objekat član	*member object*
oblast važenja	*scope*
obrada izuzetaka	*exception handling*
odredište	*target*

odsečak	*slice*
okruženje za brzo razvijanje aplikacija	*Rapid Application Development, RAD*
operator pomeranja	*shift operator*
opseg	*range*
oznaka	*label*
oznaka kraja	*end sentinel*
oznaka tipa	*type tag*
paralelnost, paralelni rad	*concurrency*
počistiti	*clean up*
podobjekat	*subobject*
pokazivač steka	*stack pointer*
poslati	*submit*
posrednički program	*middleware*
posrednik	*proxy*
povezano zaključavanje	*hand-over-hand locking, lock coupling*
povezivač	*linker*
povratni poziv	*callback*
površno kopiranje	*shallow copy*
poziv ugrađen direktno u kôd	*inline call*
prazno finalno polje	*blank final*
prednapredno	*preemptive*
prekidanje	*termination*
preklapanje operatora	*operator overloading*
prijemnik	*listener*
prilagodljiv izgled i ponašanje	*pluggable look & feel*
priručni savet	*tool tip*
produžavanje uz dodavanje nula	*zero extension*
produžavanje uz očuvanje znaka	*sign extension*
profajler	*profiler*
program za učitavanje	*loader*
programiranje s klijentske strane	*client-side programming*
programiranje sa serverske strane	*server-side programming*
pseudonim, alijas	*alias*
putanja klasa	*classpath*
rano vezivanje	*early binding*
raspoređivač	*layout manager*
red za čekanje	*queue*
red za čekanje događaja	*event queue*
redefinisanje	*overriding*
rukovalac događajem	*event handler*
sakupljač smeća	*garbage collector*
servisna nit	*daemon thread*
sistem kojim upravljaju događaji	*event-driven system*
skladište	*pool*
skupljač s kopiranjem	*copy collector*
slaba referenca	*weak reference*
specifikator pristupa	*access specifier*
spoljno nadovezivanje	*external chaining*
sporedan efekat	*side effect*
spravica, element	*widget*
stablo	*tree*
stanje okončanja	*termination condition*
standardna biblioteka šablona	*Standard Template Library, STL*
status prekinutosti	*interrupted status*
struktura programa, kostur programa	*application framework*
sudar, sukob	*collision*
svojstvo	*property*
svođenje naniže	*downcasting*
svođenje naniže koje čuva tip	*type-safe downcast*
svođenje naviše	*upcasting*
šablon	*template*
tok podataka	*stream*
torka	*tuple*
trajni objekat	*persistent object*
traka napredovanja	*progress bar*
transformisanje ključa, heširanje	*hashing*
učitavač klasa	*class loader*
ugašen	*terminated*
ulančavanje unazad	*backward chaining*
unutrašnja klasa	*inner class*
uporediv	*comparable*
uslov za trku	*race condition*
uzajamna blokada	*deadlock*
uzajamno isključiva brava	*mutex*
uzajamno isključivanje	*mutual exclusion*
uzrok	*cause*
vanredno stanje	*exceptional condition*
vezivanje	*binding*
vezivanje prilikom izvršavanja	*runtime binding*
vezivna funkcija	*stub*
visoko spregnut	*highly coupled*
višekratno otkrivanje tipa	*multiple dispatching*
višeprogramski	*multitasking*
vremenska oznaka	*time stamp*
vremensko odlaganje	*timeout*
zaključak	*closure*
zapisnik, dnevnik	*log*
zaštitna barijera	*firewall*
zauzetost čekanjem	*busy waiting*
zrna Jave	*Java Beans*

Kompletan spisak termina koji se koriste u izdanjima Mikro knjige nalazi se na adresi `www.mk.co.yu/recnik`.

Indeks

!

!, 75
!=, 73

&

&, 79
&&, 75
&=, 80

.

.NET, 39
.new, sintaksa, 270
.this, sintaksa, 270

@

@, simbol za anotacije, 845
@author, 60
@Deprecated, anotacija, 845
@deprecated, Javadoc oznaka, 62
@docRoot, 60
@inheritDoc, 60
@interface, i rezervisana reč extends, 854
@link, 60
@Override, 845
@param, 61
@Retention, 846
@return, 61
@see, 60
@since, 61
@SuppressWarnings, 845
@Target, 846
@Test, 816
@Test, za @Unit, 864
@TestObjectCleanup, oznaka za alatku @Unit, 872
@TestObjectCreate, za @Unit, 869
@throws, 61
@Unit, 864
  upotreba, 864
@version, 60

[

[], operator indeksiranja, 145

^

^, 80
^=, 80

|

|, 79
|=, 80
||, 75

+

+, 71
  konverzija u tip String operatorom +, 67, 85, 393

<

<, 73
<=, 73
<<, 80
<<=, 81

=

==, 73

>

>, 73
>=, 73
>>, 80
>>=, 81

A

abecedno uredivanje, 325
abstract (apstraktna)
  klasa, 239
  nasleđivanje od apstraktnih klasa, 240
  rezervisana reč, 240
  u odnosu na interfejs, 253
AbstractButton, 1069
AbstractSequentialList, 684
AbstractSet, 629
ActionEvent, 1091, 1130
ActionListener, 1055
ActionScript, za Adobe Flex, 1139
Adapter, projektni obrazac, 250, 257, 338, 495, 582, 585, 631
adapteri prijemnika, 1065
adapterska metoda, 338
add( ), metoda klase ArrayList, 303
addActionListener( ), 1128, 1134
addChangeListener, 1094
addListener, 1060
Adler32, 777
Adobe Flex, 1138
Adobe, proizvođač sistema Flex, 1138
agentski orijentisano programiranje, 1042
agregacija, 21

aktivni objekti,
u paralelnom izvršavanju, 1039
alatka za prikazivanje,
za Swing, 1050
Allison, Chuck, 3, 11, 1166, 1174
allocate( ), 755, 797–798
allocateDirect( ), 755
angažovanje, Teorija rastućeg, 915
anonimna unutrašnja klasa, 275, 721, 1053
generička, 508
i kôd upravljan tabelom, 684
anotacija, 845
apt, alatka za obradu, 857
dozvoljeni tipovi elemenata, 849
elementi, 847
marker anotacija, 847
podrazumevana vrednost, 853
podrazumevane vrednosti elemenata, 847–848, 850
procesor, 848
procesor koji radi na osnovu refleksije, 854
apstrakcija, 15
apt, alatka za obradu anotacija, 857
argument
finalni, 204, 722
konstruktor, 116
kovarijantni tipovi argumenata, 559
liste promenljivih parametara (nepoznat broj i tip argumenata), 150
zaključivanje o tipovima generičkih argumenata, 497
argumenti promenljive dužine, 150, 578
i generičke metode, 499
Arnold, Ken, 1046
ArrayBlockingQueue, 971
ArrayList, lista, 311, 644
add( ), metoda, 303
get( ), metoda, 303
size( ), metoda, 303
Arrays
asList( ), 307, 340, 648
binarySearch( ), 624
uslužna metoda za pretvaranje sekvence ili niza u kontejner, 616
asCharBuffer( ), 757
asocijativan niz, 302, 306
drugo ime za mapu, 661
aspektno orijentisano programiranje (AOP), 565
assert, i @Unit, 867
AtomicInteger, 932
AtomicLong, 932
AtomicReference, 932
atomska operacija, 926
automatska konverzija tipova, 181
automatsko pakovanje, 326, 495
i generički tipovi, 497, 549
available( ), 740

B

bafer, nio, 754
BASIC, Microsoftov Visual BASIC, 1120
BasicArrowButton, 1070
BeanInfo, namenska klasa, 1137
Beck, Kent, 1171
bezuslovni skok, 106
biblioteka
autor u odnosu na programera klijenta, 159
projekat, 159
upotreba, 159
binarni
brojevi, 78
brojevi, ispisivanje, 84
operatori, 80
binarySearch( ), 624, 706
BitSet, 716
Bloch, Joshua, 130, 805, 914, 929
BlockingQueue, 971, 988
blok za obradu izuzetka, 348
blok za pozive, 466
blokiranje i metoda available( ), 740
u programima za paralelno izvršavanje, 887
Booch, Grady, 1172
boolean, 98
algebra, 79
i eksplicitna konverzija tipova, 87
operatori koji ne rade s logičkim argumentima, 73
u odnosu na C i C++, 75
Borlandov Delphi, 1120
BoxLayout, 1059
Brajanovo pravilo sinhronizacije, 922
brave
eksplicitne, u paralelnom izvršavanju, 923
optimističko zaključavanje, 1034
takmičenje za, u paralelnom izvršavanju, 1019
u paralelnom izvršavanju, 922

break, rezervisana reč, 107
brisanje, 551
  u generičkom kodu, 512
brojanje referenci, sakupljanje smeća, 133
brojevi, binarni, 78
Budd, Timothy, 16
BufferedInputStream, 733
BufferedOutputStream, 734
BufferedReader, 377, 736, 738
BufferedWriter, 736, 740
ButtonGroup, 1070, 1079
ByteArrayInputStream, 731
ByteArrayOutputStream, 732
ByteBuffer, 754

C

C#, programski jezik, 39
C++, 73
  obrada izuzetaka, 384
  Standardna biblioteka šablona (Standard Template Library, STL), 718
  šabloni, 485, 513
CachedThreadPool, 894
Callable, paralelno izvršavanje, 896
case, naredba, 112
CASE_INSENSITIVE_ORDER, String Comparator, 705, 720
cast( ), 444
catch
  hvatanje izuzetka, 348
  hvatanje bilo kog izuzetka, 356
  rezervisana reč, 348
cepanje reči, u paralelnom izvršavanju, 926
cepanje, cepanje reči, 926
cev, 731, 745
  i ulazno/izlazne operacije, 976
Chain of Responsibility, projektni obrazac, 826
CharArrayReader, 735
CharArrayWriter, 735
CharBuffer, 757
CharSequence, interfejs, 414
Charset, 758
checkedCollection( ), 562
CheckedInputStream, 775
checkedList( ), 562
checkedMap( ), 562
CheckedOutputStream, 775
checkedSet( ), 562
checkedSortedMap( ), 562
checkedSortedSet( ), 562
Checksum, klase, 777
Chiba, Shigeru, Dr., 881, 883
Class, 1071
  forName( ), 436, 1063
  generičke reference klasa, 441
  getCanonicalName( ), 438
  getClass( ), 357
  getConstructors( ), 464
  getInterfaces( ), 438
  getMethods( ), 464
  getSimpleName( ), 438
  getSuperclass( ), 438
  isAssignableFrom( ), 454
  isInstance( ), 453
  isInterface( ), 438
  newInstance( ), 438
  objekat tipa Class, 434, 795, 922
  postupak pravljenja objekata, 142
  reference, i džokeri, 442
  reference, i ograničenja, 442
  RTTI pomoću objekta tipa Class, 434
ClassCastException, 237, 445
ClassNotFoundException, 450
classpath, sistemska promenljiva, 163
clear( ), nio, 756
close( ), 738
collection, 332, 705
Collections, klasa
  enumeration( ), 714
  fill( ), 629
  metoda addAll( ), 307
  unmodifiableList( ), 648
Command, projektni obrazac, 295, 473, 822, 893
Comparable, interfejs, 619, 653, 658
Comparator, interfejs, 621
compareTo( ), u paketu java.lang.Comparable, 619, 655
ConcurrentHashMap, 664, 1027, 1032
ConcurrentLinkedQueue, 1027
ConcurrentModification-Exception, 709
  izbegavanje pomoću klase CopyOnWriteArray-List, 1027, 1041
Condition, klasa, u paralelnom izvršavanju, 968
continue, rezervisana reč, 107
Coplien, Jim
  šablonski obrazac koji se neobično ponavlja, 556
CopyOnWriteArrayList, 1003, 1027
CopyOnWriteArraySet, 1027

CountDownLatch, za paralelno izvršavanje, 983
CRC32, 777
crtanje na panou u Swingu, 1093
crtanje u Swingu, 1092
CSS (kaskadni opisi stilova), i Adobe Flex, 1144
CyclicBarrier, za paralelno izvršavanje, 986

Č

čista supstitucija, 25, 235
čišćenje
  i skupljač smeća, 191
  izvršavanje, 131
  provera stanja okončanja metodom finalize( ), 131
  uz finally, 368
čitač klasa, 175
čitanje standardnog ulaznog toka, 749
član
  funkcija članica, 19
  inicijalizatori, 225
  objekat, 21
čuvana oblast, u obradi izuzetaka, 348

D

DatagramChannel, 773
DataInput, 737
DataInputStream, 733, 736, 739
DataOutput, 737
DataOutputStream, 734, 737
datoteka
  dijalozi, 1099
  File, klasa, 719, 731, 737
  File.list( ), 719
  gubitak podataka izlazne datoteke, greške i pražnjenje bafera, 741
  JAR arhiva, 161
  obeležja, 728
  preslikana u memoriju, 769
  zaključavanje, 772
datoteka klase, analiza, 879
decode( ), šema kodiranja, 759
Decorator, projektni obrazac, 568
default, rezervisana reč u naredbi switch, 112
defaultReadObject( ), 792
defaultWriteObject( ), 792
DeflaterOutputStream, 775
Delayed, 990
DelayQueue, za paralelno izvršavanje, 988
delegiranje, 188, 568
Delphi, kompanije Borland, 1120
deljenje, 69
DeMarco, Tom, 1173
deque, dvostrani red za čekanje, 319, 660
destruktor, 129, 131, 368
  Java ga nema, 191
dijagram
  dijagrami nasledivanja klasa, 199
  nasleđivanje, 28
dijalog
  okvir za, 1095
  okno s jezičcima, 1083
  za rad s datotekama, 1099
Dijkstra, Edsger, 978
dim, 797–798
dinamička
  bezbednost tipova i kontejneri, 562
  posrednik, 466
  promena ponašanja pomoću kompozicije, 234
  provera tipova u Javi, 647
  sintaksa za skupnu inicijalizaciju nizova, 597
  vezivanje, 211, 214
direktorijum
  i paketi, 168
  listanje, 719
  pravljenje direktorijuma i putanja, 728
disjunkcija, 86
  (||), 75
dispose( ), 1096
dodela objekata, 67
dodela vrednosti, 67
događaj
  događaji i prijemnici, 1059–1060
  događaji i zrna Jave, 1121, 1130
  model, Swing, 1059
  odgovor na Swing događaj, 1052
  programiranje upravljano događajima, 1052
  sistem upravljan događajima, 290
dokumentacija, 11
  komentari i ugrađena dokumentacija, 57
dostavljač poruka, idiom, 487, 632, 684
dostižni objekti i skupljanje smeća, 710
double
  i višenitni rad, 926
  marker d ili D, literala, 78
do-while, 101
drugi komplement, 84

dugme
pravljenje sopstvenog, 1066
radio-dugme, 1078
Swing, 1051, 1069
dvokratno otkrivanje tipa, 836
pomoću mape EnumMap, 842
dvostrani red za čekanje (deque), 319
dvrednost, 67

DŽ

džokeri
i Class reference, 442
nadtipova, 539
neograničeni, 541
u generičkim tipovima, 534

E

East, BorderLayout, 1056
efikasnost
i nizovi, 593
i finalnost, 207
eksplicitno zadavanje tipova argumenata generičkih metoda, 309, 499
eksponencijalni zapis, 78
else, rezervisana reč, 99
encode( ), šema kodiranja, 759
endian
big endian, 763
little endian, 763
entrySet( ), metoda interfejsa Map, 673
enum. *Videti* nabrojani tipovi
Enumeration, 713
EnumMap, 821
EnumSet, 505, 718
umesto indikatora, 819
equals( ), 73
i hashCode( ), 653, 679
i strukture podataka s transformisanim ključevima, 671
redefinisanje za HashMap, 670
uslovi za pravilno definisanje, 670
Erlang, programski jezik, 888
EventSetDescriptor, 1126
Exchanger, klasa, u paralelnom izvršavanju, 1001
Executor (izvršilac), paralelno izvršavanje, 893
ExecutorService, 893
extends, rezervisana reč, 172, 185, 235
i @interface, 854
i interfejs, 254
rezervisana reč, 183
Externalizable, 785
alternativa za, 791
Extreme Programming (XP), 1171

F

Façade, 452
Factory Method, projektni obrazac, 262
faktor opterećenja, objekata tipa HashMap ili HashSet, 701
false, 75
FeatureDescriptor, 1137
Fibonacci, 494
Field, klasa za refleksiju, 462
FIFO (prvi izlazi onaj koji je prvi ušao), 329
FileChannel, 754
FileDescriptor, 731
FileInputReader, 738
FileInputStream, 731
FileLock, 773
FilenameFilter, 719
FileNotFoundException, 378
FileOutputStream, 732
FileReader, 377, 735
FileWriter, 735, 740
fillInStackTrace( ), 359
filozofi, koji večeraju, primer uzajamne blokade u paralelnom izvršavanju, 978
FilterInputStream, 731
FilterOutputStream, 732
FilterReader, 736
FilterWriter, 736
final, 488
final (finalno), 243
argument, 204, 722
i efikasnost, 207
i privatnost, 205
i reference objekata, 201
i statičnost, 201
klase, 206
metoda, 215
metode, 204, 232
podaci, 200
prazna finalna polja, 203
rezervisana reč, 200
statični prosti tipovi, 202
finalize( ), 129, 194, 379
direktno pozivanje, 131
i nasleđivanje, 226
finally, 191, 194
i konstruktori, 377
i return, 371
mana, 372

ne izvršava se u slučaju servisnih niti, 905
rezervisana reč, 367
FixedThreadPool, 894
Flex
OpenLaszlo, alternativa za Flex, 1138
sistem kompanije Adobe, 1138
flip( ), nio, 755
float
istina i neistina u poređenju brojeva s pokretnim zarezom, 76
marker (F) literala, 78
FlowLayout, 1057
Flyweight, projektni obrazac, 635, 1043
for, rezervisana reč, 101
foreach, 103, 107, 150–151, 167, 291, 305, 328, 334, 426, 494–495, 549, 805, 827
i Adapterska metoda, 338
i interfejs Iterable, 336
i Iterable, 336
format
preciznost, 402
prikazivanja znakovnih nizova, 400
specifikatori, 402
format( ), 400
Formatter, 401
forName( ), 436, 1063
FORTRAN, programski jezik, 79
Fowler, Martin, 159, 386, 1172
funkcija
funkcija članica, 19
redefinisanje, 24
funkcija za transformisanje ključeva (heš funkcija), 675
funkcijski objekat, 585
funkcijski programski jezici, 888
Future, 897

G

Gecov test, za izbegavanje sinhronizacije, 926
Generator, 218, 492, 500, 508, 550, 580, 606, 621, 630, 813
opšte namene, 501
popunjavanje kontejnera (podtipova klase Collection), 500
generički tipovi
anonimne unutrašnje klase, 508
argumenti promenljive dužine i generičke metode, 499
brisanje, 512, 551
Class reference, 441
džokerski argumenti, 534
džokerski argumenti nadtipova, 539
eksplicitna konverzija, 551
eksplicitna konverzija putem generičke klase, 553
eksplicitno zadavanje tipova argumenata generičkih metoda, 309, 499
granice, 514, 531
i kontejneri za bezbedan rad s tipovima, 303
instanceof, naredba, 522, 551
isInstance( ), 522
izuzeci, 563
koji se neobično ponavljaju, 556
konkretizacija, 516
kontravarijansa, 539
metode, 496, 631
najjednostavnija definicija klase, 321
neograničen džokerski argument, 541
niz generičkih objekata, 677
osnovni uvod, 303
oznaka tipa, 522
preklapanje, 553
primer strukture, 1028
samoograničeni tipovi, 555
testiranje sa @Unit, 873
unutrašnje klase, 508
generisanje izuzetka, 347
get( ), metoda klase ArrayList, 303
HashMap, realizacija klase Map, 326
u interfejsu Collection nema metode get( ), 644
getBeanInfo( ), 1124
getBytes( ), 740
getCanonicalName( ), 438
getChannel( ), 755
getClass( ), 357, 436
getConstructor( ), 1071
getConstructors( ), 464
getenv( ), 337
getenv( ), metoda, 337
getEventSetDescriptors( ), 1126
getInterfaces( ), 438
getMethodDescriptors( ), 1126
getMethods( ), 464
getName( ), 1126
getPropertyDescriptors( ), 1126
getPropertyType( ), 1126
getReadMethod( ), 1126
getSelectedValues( ), 1081

getSimpleName( ), 438
getState( ), 1090
getSuperclass( ), 438
getWriteMethod( ), 1126
Glass, Robert, 1172
Goetz, Brian, 922, 926, 1019, 1044
goto, nepostojanje te naredbe u Javi, 108
grafičko korisničko okruženje (GUI), 290, 1045
grafika, 1098
  Graphics, klasa, 1093
granice
  i Class reference, 442
  natklasa i Class reference, 444
  samoograničeni generički tipovi, 555
  u generičkim tipovima, 514, 531
greška
  obrada pomoću izuzetaka, 345
  oporavak, 345
  prijavljivanje, 385
  prijavljivanje grešaka u knjizi, 14
  standardni izlazni tok za greške, 350
GridBagLayout, 1058
GridLayout, 1058, 1119
Grindstaff, Chris, 1150
grupa niti, 914
grupa objekata, 997
grupe, regularnog izraza, 417
GUI
  grafički korisnički interfejs, 290, 1045
  razvojna okruženja za GUI, 1046
GZIPInputStream, 775
GZIPOutputStream, 775

H

Harold, Elliotte Rusty, 1137, 1171
  XOM, XML biblioteka, 799
hashCode( ), 663, 668, 670, 675
  recept za pisanje pristojne metode, 679
hashCode( )
  equals( ), 653
  i strukture podataka s transformisanim ključevima, 671
  na šta obratiti pažnju prilikom pisanja sopstvene metode, 678
HashMap, 664, 700, 1032, 1068
HashSet, realizacija skupa Set, 322, 653, 696
Hashtable, 700, 714
hasNext( ), metoda klase Iterator, 316
heksadecimalno, 78
Holub, Allen, 1039
HTML u Swing komponentama, 1101

I

IdentityHashMap, 664, 700
if-else naredba, 84, 99
ikonice i klasa Icon, 1071
IllegalAccessException, 449
IllegalMonitorStateException, 958
ima, relacija, 21
  relacija kompozicije, 197
ImageIcon, 1071
ime
  pravljenje jedinstvenih imena paketa, 163
  puno, 438
  sukobljavanje prilikom kombinovanja interfejsa, 255
  sukobi, 165
ime klase, otkrivanje u datoteci klase, 879
imenski prostori, 160
implements, rezervisana reč, 243
import, rezervisana reč, 160
indeksiranje, operator [], 145
indeksirano svojstvo, 1137
indexOf( ), metoda klase String, 464
indikator, korišćenje skupa EnumSet umesto, 819
InflaterInputStream, 775
inicijalizacija
  člana klase, 181
  i nasleđivanje, 208
  i učitavanje klasa, 208
  inicijalizacija instance, 144, 277
  inicijalizacija konstruktorima tokom nasleđivanja i kompozicije, 189
  inicijalizacija nestatičnih instanci, 144
  inicijalizacija niza, 145
  inicijalizacija pomoću konstruktora, 115
  inicijalizatori članova, 225
  klase, 440
  lenja, 182
  osnovna klasa, 185
  polja klase, 136
  promenljivih u metodama, 136
  redosled inicijalizacije, 139, 232
  statičnih, 209
inicijalizovanje izvedene klase, 185
InputStream, 730

InputStreamReader, 735
instanca
  inicijalizacija instance, 277
  inicijalizacija nestatičnih instanci, 144
  klase, 16
instanceof, 451
  dinamičko ispitivanje tipa objekta metodom isInstance( ) umesto sa instanceof, 453
  i generički tipovi, 551
  rezervisana reč, 445
Integer
  parseInt( ), 1099
  omotačka klasa, 148
interfejs
  i enum, 815
  i generički kôd, 484
  i nasleđivanje, 253
  i razdvajanje od realizacije, 20, 174, 1060
  inicijalizacija polja u interfejsima, 259
  interfejs osnovne klase, 218
  klase ugnežđene unutar, 283
  privatni, kao ugnežđeni interfejsi, 261
  rezervisana reč interface, 242
  sukobljavanje imena prilikom kombinovanja interfejsa, 255
  svođenje naviše na interfejs, 245
  u odnosu na apstraktnu klasu, 253
  u odnosu na realizaciju, 196
  ugnežđivanje interfejsa unutar klasa i drugih interfejsa, 260
  za objekat, 17
  zajednički interfejs, 239
internacionalizacija, u U/I biblioteci, 735
interrupt( )
  paralelno izvršavanje, 946
  višenitni rad, 912
Introspector, 1124
inženjering bajtkoda, 879
  Javassist, 881
isAssignableFrom( ), metoda klase Class, 454
iscrpljenje memorije, rešenje pomoću referenci, 710
isDaemon( ), 903
isInstance( ), 453
  i generički tipovi, 522
isInterface( ), 438
Iterable
  i foreach, 336
Iterable, interfejs, 494, 632
  i foreach sintaksa, 336
  i niz, 337
Iterator, 332
Iterator, klasa, 315, 318, 332
  hasNext( ), metoda, 316
  next( ), metoda, 316
Iterator, projektni obrazac, 269
izgled i ponašanje, promenljiv, 1103
izuzetak
  blok try, 348
  blok za obradu, 346
  blok za obradu izuzetka, 348
  čuvana oblast, 348
  Error, klasa, 365
  Exception, klasa, 365
  FileNotFoundException, 378
  fillInStackTrace( ), 359
  finally, 367
  generički kôd, 563
  generisanje izuzetka, 346, 347
  gubljenje izuzetka, mana, 372
  hijerarhije klasa, 382
  hvatanje bilo kog izuzetka, 356
  hvatanje izuzetka, 348
  i konstruktori, 376
  i konzola, 387
  i nasleđivanje, 374, 382
  i paralelnost, 924
  konstruktori, 377
  menjanje mesta nastanka izuzetka, 360
  neprovereni, 366
  NullPointerException, 366
  obrada, 33
  obrada izuzetaka, 345
  ograničenja, 374
  ponovno generisanje izuzetka, 359
  pravljenje sopstvenih, 349
  prekidanje ili nastavljanje, 349
  pretvaranje proverenih u neproverene izuzetke, 388
  prijavljivanje izuzetaka putem zapisnika, 353
  printStackTrace( ), 359
  problemi pri projektovanju, 379
  pronalaženje sličnih izuzetaka, 382
  proveren, 356, 384
  RuntimeException, 366
  specifikacija, 355, 385
  Throwable, klasa, 356

tipične upotrebe izuzetaka, 390
try, 368
ulančani izuzeci, 388
ulančavanje, 362
vanredno stanje, 346
zapisivanje, 352
izvedena klasa, 214
izvedeni tipovi, 22
izvorni kôd, 12
napomena o zaštiti autorskih prava, 12
izvršavanje Java programa, 56
izvršavanje programa operativnog sistema iz Jave, 752

J

Jacobsen, Ivar, 1172
JApplet, 1056
meniji, 1085
JAR, 1135
arhiva, 161
jar arhive i promenljiva classpath, 164
uslužni program, 779
Java
AWT, 1045
bajtkod, 394
i aparati za prikazivanje Interneta na TV-u, 79, 394
Java Web Start, 1105
Javina virtuelna mašina (JVM), 434
Javine osnovne klase (JFC/Swing), 1045
javni seminari o Javi, 9
prevođenje i izvršavanje programa, 56
JavaBeans, videti zrna Jave, 1120
javac, 57
javadoc, 58
javap, prevodilac unazad, 393, 479, 520
Javassist, 881
Javina standardna biblioteka, i bezbedno višenitno izvršavanje, 985
JButton, 1071
Swing, 1051
JCheckBox, 1071, 1077
JCheckBoxMenuItem, 1086, 1090
JComboBox, 1080
JComponent, 1073, 1093
JDialog, 1096
meniji, 1085
JDK 1.1 U/I tokovi, 734
JDK, preuzimanje i instaliranje, 56
je, relacija, 235
i svođenje naviše, 198
relacija nasleđivanja, 197
u odnosu na relacije je-kao, 25
je-kao, relacija, 235
jedinica za prevođenje, 161
jednakost
==, 73
jednakost objekata, 73
jednokratno otkrivanje tipa, 836
jednosmerni (događaj), 1130
JFC, Javine osnovne klase (Swing), 1045
JFileChooser, 1099
JFrame, 1056
meniji, 1085
JIT, prevodioci baš-kad-treba, 135
JLabel, 1075
JList, 1081
JMenu, 1085, 1090
JMenuBar, 1085, 1091
JMenuItem, 1071, 1085, 1090, 1092
JNLP, Java Network Launch Protocol, 1105
join( ), višenitni rad, 912
JOptionPane, 1083
Joy, Bill, 73
JPanel, 1070, 1093, 1119
JPopupMenu, 1091
JProgressBar, 1103
JRadioButton, 1071, 1079
JScrollPane, 1055, 1082
JSlider, 1103
JTabbedPane, 1083
JTextArea, 1054
JTextField, 1052, 1073
JTextPane, 1076
JToggleButton, 1070
JUnit, problemi sa, 864
JVM (Javina virtuelna mašina), 434

K

kanal, nio, 754
kapacitet, objekata tipa HashMap ili HashSet, 700
kapsuliranje, 174
upotreba refleksije za poništavanje, 476
kaskadni opisi stilova (CSS), i Adobe Flex, 1144
kasno vezivanje, 27, 211, 214
keySet( ), 700
kidanje veze, putem polimorfizma, 27, 211
klasa, 17
anonimna unutrašnja klasa, 275, 721, 1053
apstraktna klasa, 239
autori, 20
čitač, 175

dijagrami nasleđivanja, 199
ekvivalentnost, i instanceof/ isInstance( ), 459
finalne klase, 206
hijerarhije klasa i obrada izuzetaka, 382
inicijalizacija, 440
inicijalizacija članova, 181
inicijalizacija i učitavanje klasa, 208
inicijalizacija polja, 136
inicijalizovanje izvedene klase, 185
inicijalizovanje osnovne klase, 185
izvedena klasa, 214
javna klasa i jedinice za prevođenje, 161
literal klase, 439, 451
metode, 53
nasleđivanje od apstraktnih klasa, 240
nasleđivanje unutrašnje klase, 296
osnovna klasa, 172, 183, 214
podaci, 53
podobjekat, 185
povezivanje, 440
primerak, 16
pristup, 175
privatna unutrašnja klasa, 291
redosled inicijalizacije, 139
rezervisana reč class, 22
statične unutrašnje klase, 282
stil pravljenja klasa, 174
učitavanje, 209, 440
ugneždena klasa (statična unutrašnja klasa), 282
ugnežđivanje unutar interfejsa, 283
unutrašnja klasa, 266
unutrašnja klasa i prava pristupa, 268
unutrašnja klasa i svođenje naviše, 271
unutrašnja klasa i Swing, 1060
unutrašnja klasa, i redefinisanje, 297
unutrašnja klasa, i rezervisana reč super, 297
unutrašnja klasa, identifikatori i .class datoteke, 300
unutrašnja klasa, u metodama i oblastima važenja, 273
unutrašnja klasa, ugnežđivanje unutar proizvoljne oblasti važenja, 274
upućivanje na objekat spoljne klase iz unutrašnje klase, 270
višestruko ugnežđena, 284
klasa, učitavač, 434
klijent, programer, 20
klizač, 1102
konjunkcija
nad bitovima, 86
logička (&&), 75
kôd
izvorni, 12
podela i smeštanje, 168
ponovna upotreba, 180
standardi programiranja, 13
stil programiranja, 63
kôd bez zaključavanja, u paralelnom izvršavanju, 926
kôd upravljan tabelom, 823
i anonimne unutrašnje klase, 684
kolekcija, 29, 306, 332
klase, 302
popunjavanje pomoću Generatora, 500
spisak metoda za, 643
uslužne metode, 701
kolekcije, parametar, 565, 589
komanda akcije, 1090
kombinovane liste, 1080
komentari i ugrađena dokumentacija, 57
kompatibilnost
vertikalna, 516
migracijska, 516
komplet alatki za apstraktne prozore (Abstract Window Toolkit, AWT), 1045
komponenta, i zrna Jave, 1121
kompozicija, 21, 180
i dinamička promena ponašanja, 234
i projektovanje, 233
kombinovanje kompozicije i nasleđivanja, 189
u odnosu na nasleđivanje, 196, 200, 660, 714
komprimovanje, biblioteka za, 775
komunicirajući sekvencijalni procesi (CSP), 1042
konferencija, za razvoj softvera, 9
konkretizacija i generički tipovi, 516
konsalting i obuka koju pruža MindView, Inc., 1166

konstanta
grupe konstantnih vrednosti, 258
posredne konstante i String, 393
pri prevođenju, 200
uklapanje konstante, 201
konstruktor, 115
argumenti, 116
bez argumenata, 116, 124
Constructor, klasa za refleksiju, 462
i anonimne unutrašnje klase, 275
i finally, 377
i obrada izuzetaka, 376–377
i paralelno izvršavanje, 907
i polimorfizam, 224
i preklapanje, 117
inicijalizacija instance, 277
inicijalizacija tokom nasleđivanja i kompozicije, 189
konstruktor osnovne klase, 225
nivo pristupa automatski napravljenog podrazumevanog konstruktora, 464
podrazumevani, 124
ponašanje polimorfnih metoda unutar konstruktora, 230
povratna vrednost, 117
pozivanje iz drugih konstruktora, 127
pozivanje konstruktora osnovne klase sa argumentima, 186
redosled poziva konstruktora uz nasleđivanje, 224
statička odredba konstruktora, 143
statične metode, 142
kontejner, 29
ispitivanje performansi, 684
klasa, 302
poređenje s nizom, 593
kontejneri
bez zaključavanja, 1027
koji brzo otkazuju, 708
osnovno ponašanje, 309
za bezbedan rad s tipovima i generički tipovi, 303
kontravarijansa i generički kôd, 539
kontrola pristupa, 20, 178
konverzija
automatska, 181
proširujuća, 87
sužavajuća, 87
konverzija tipova, 28
asSubclass( ), 445
i generički tipovi, 551
i prosti tipovi, 98
operatori, 87
putem generičke klase, 553
konverzija vremenskih jedinica, 990
konzola
slanje izuzetaka na, 387
Swing alatka za prikazivanje u net.mindview.util. SwingKonzola, 1050
kopiranje niza, 617
korisnički interfejs
grafički korisnički interfejs (GUI), 290, 1045
koje brzo reaguje, realizacija pomoću niti, 913
kostur upravljanja i unutrašnje klase, 290
kovarijantni, 441
nizovi, 535
povratni tipovi, 232, 559
tipovi argumenata, 559
kritičan odeljak
i sinhronizovani blok, 933
kvantifikator
pohlepno, 413
posesivno, 413
regularnog izraza, 413
rezervisano, 413

L

lak
objekat, 315
trajnost, 781
latentni tipovi, 572, 582
lažni objekat, 476
leksikografsko uređivanje, 325
u odnosu na abecedno uređivanje, 623
length
član niza, 147
za nizove, 595
lenja inicijalizacija, 182
lepak, raspoređivača BoxLayout, 1059
LIFO (poslednji koji uđe, prvi izlazi), 320
LineNumberInputStream, 733
LineNumberReader, 736
linija napredovanja, 1102
LinkedBlockingQueue, 971
LinkedHashMap, 664, 667, 700
LinkedHashSet, realizacija skupa Set, 323, 653, 696–697

LinkedList, ulančana lista, 311, 319, 329, 649
List, klasa, 302, 306, 311, 649, 1081
  poređenje performansi, 688
  uređivanje i pretraživanje, 705
lista
  grafičke liste, 1081
  padajuća lista, 1080
Lister, Timothy, 1173
ListIterator, 649
literal
  double, 78
  float, 78
  klase, 439, 451
  long, 78
  vrednosti, 77
little endian, 763
logaritmi, prirodni, 79
logička
  konjunkcija, 86
  disjunkcija, 86
  operator i nepotpuno izračunavanje, 76
  operatori, 75
lokalna
  unutrašnja klasa, 274
  promenljiva, 50
long
  i višenitni rad, 926
  marker (L) literala, 78
LRU, algoritam najdavnijeg korišćenja, 667
lvrednost, 67

M

main( ), 184
manifest datoteka, za JAR datoteke, 779, 1135
manje ili jednako (<=), 73
manje od (<), 73
Map, klasa, 302, 306, 326
  EnumMap, 821
  iscrpno istraživanje, 661
  poređenje performansi, 698
Map.Entry, 673
MappedByteBuffer, 769
mark( ), 737
marker anotacija, 847
mašine stanja i nabrojani tipovi, 830
matcher( ), metoda za pronalaženje regularnog izraza, 414
matches( ), metoda klase String, 409
matematički operatori, 69, 773
Math.random( ), metoda
  opseg rezultata, 695
Math.random( ), metoda za generisanje slučajnih brojeva, 326
mehanizam za raspodelu procesorskog vremena nitima, 891
meni
  JDialog, JApplet, JFrame, 1085
  JPopupMenu, 1091
metaanotacije, 848
Metapodaci, 845
Method, 1126
  klasa za refleksiju, 462
MethodDescriptor, 1126
metoda
  alatka za pronalaženje, 1061
  dodavanje metoda programu, 179
  finalna, 204, 215, 232
  generička, 496
  inicijalizacija promenljivih u metodama, 136
  pojava pseudonima pri pozivanju metoda, 68
  polimorfni pozivi metoda, 211
  ponašanje polimorfnih metoda unutar konstruktora, 230
  preklapanje, 117
  primena metode na sekvencu, 577
  privatna, 232
  razlikovanje preklopljenih metoda, 119
  redefinisanje privatnih, 221
  rekurzivna, 398
  statična, 129, 215
  ugradivanje poziva metoda direktno u kôd, 204
  unutrašnje klase u metodama i oblastima važenja, 273
  vezivanje poziva metoda, 214
  zaštićene metode, 198
Meyer, Jeremy, 845, 879, 1105
Meyers, Scott, 20
Microsoftov Visual BASIC, 1120
migracijska kompatibilnost, 516
mikropoređenje performansi, 694
miksin, 565
mkdirs( ), 730
mnemonici (prečice s tastature), 1090
množenje, 69
modulo, 69
mogućnost ponovnog korišćenja, 21

monitor, za paralelno izvršavanje, 922
Mono, 39
mreža objekata, 782
mrežni ulaz/izlaz, 753
MXML, ulazni format Adobe Flexa, 1138
mxmlc, prevodilac za Adobe Flex, 1140

N

nabrojani tipovi, 155
  dodavanje metoda, 807
  grupe konstantnih vrednosti u jezicima C i C++, 258
  i interfejsi, 815
  i mašine stanja, 830
  i naredba switch, 809
  i nasleđivanje, 813
  i nasumičan izbor, 814
  i projektni obrazac Chain of Responsibility, 826
  i uvoz statičnih članova, 806
  i višekratno otkrivanje tipa, 836
  metode koje se menjaju u zavisnosti od konstante (nabrojanog tipa), 823, 840
  rezervisana reč enum, 155, 805
  values( ), 805, 810
način pisanja koda, 13
nad bitovima
  disjunkcija, 86
  isključiva disjunkcija XOR (^), 80
  konjunkcija, 86
  negacija ~, 80
  operator disjunkcije (|), 79
  operator konjunkcije (&), 79
  operatori, 79
nadtipovi, džokerski argumenti, 539
najdavnije korišćeno (algoritam, LRU), 667
napomena o zaštiti autorskih prava, izvorni kôd, 12
nasleđivanje, 22, 172, 180, 183, 211
  dijagram, 28
  dijagrami nasleđivanja klasa, 199
  i finalize( ), 226
  i finalnost, 206
  i generički kôd, 484
  i nabrojani tipovi, 813
  i rezervisana reč synchronized, 1135
  inicijalizacija i nasleđivanje, 208
  kombinovanje kompozicije i nasleđivanja, 189
  od apstraktnih klasa, 240
  poređenje potpunog nasleđivanja i proširivanja, 235
  preklapanje metoda u poređenju redefinisanjem, 194
  projektovanje pomoću nasleđivanja, 233
  proširivanje interfejsa pomoću nasleđivanja, 253
  proširivanje klase tokom, 23
  specijalizacija, 197
  u odnosu na kompoziciju, 196, 200, 660, 714
  unutrašnjih klasa, 296
  višestruko nasleđivanje u jezicima C++ i Java, 251
nasleđivanje višestruke realizacije, 287
nastavljanje, prekidanje ili nastavljanje, obrada izuzetaka, 349
nasumičan izbor i nabrojani tipovi, 814
natklasa, 185
  ograničenje, 444
negacija, logička (!), 75
neobično ponavljanje
  generički tipovi, 556
  šablonski obrazac u jeziku C++, 556
neograničen džokerski argument u generičkom kodu, 541
nepodržane metode, u Java kontejnerima, 646
nepotpuno izračunavanje i logički operatori, 76
nepravilan niz, 600
neprekidnost, u paralelnom izvršavanju, 918
nepromenljivost, 470
nepromenljivost, kako kolekciju ili mapu učiniti nepromenljivom, 706
neprovereni izuzeci, 366
  pretvaranje proverenih u, 388
net.mindview.util.SwingConsola, 1050
Neville, Sean, 1138
new, operator, 129
  i niz prostih tipova, 147
newInstance( ), 1071
  refleksija, 438

next( ),metoda klase Iterator, 316
nio, 753
  bafer, 754
  i prekid, 950
  kanal, 754
  performanse, 770
nit
  bezbednost, Javina standardna biblioteka, 985
  grupa, 914
  interrupt( ), 946
  isDaemon( ), 903
  lokalno skladište niti, 940
  mehanizam za raspodelu procesorskog vremena, 891
  notifyAll( ), 957
  prioritet, 899
  resume( ) i uzajamna blokada, 946
  stanja, 945
  stop( ) i uzajamna blokada, 946
  suspend( ) i uzajamna blokada, 946
  u odnosu na zadatak, terminologija, 911
  wait( ), 957
niz
  asocijativan niz, 306
  generičkih objekata, 677
  inicijalizacija, 145
  kao prvorazredni objekti, 595
  kopiranje niza, 617
  kovarijansa, 535
  length, član, 147, 595
  nepravilan, 600
  nije iterabilan, 337
  objekata, 595
  poređenje elemenata, 619
  poređenje nizova, 618
  poređenje s kontejnerom, 593
  prostih tipova, 595
  provera granica, 147
  sintaksa dinamičke agregatne inicijalizacije, 597
  višedimenzionalni, 599
  vraćanje, 597
North, BorderLayout, 1056
notifyAll( ), 957
notifyListeners( ), 1134
nove U/I klase, 753
n-torka, 487, 502, 509
null, 46
Null iterator, projektni obrazac, 469
Null objekat, projektni obrazac, 469
NullPointerException, 366

O

obavezno proveravanje statičkih tipova, 384
ObjectOutputStream, 781
objekat, 16
  brava, za paralelno izvršavanje, 922
  član, 21
  dodela objekata kopiranjem referenci, 67
  equals( ), 73
  finalni, 201
  getClass( ), 436
  hashCode( ), metoda korenske klase Object, 663
  interfejs za, 17
  jednakost, 73
  jednakost u odnosu na jednakost referenci, 73
  mreža objekata, 782
  nizovi su prvorazredni objekti, 595
  objektno orijentisano programiranje, 432
  pojava pseudonima, 68
  postupak pravljenja, 142
  pravljenje, 116
  serijalizovanje, 780
  tipa Class, 434, 795, 922
  wait( ) i notifyAll( ), 958
objekat za prenos podataka, 487, 632
objekat za prenos podataka (idiom Prenosilac), 684
oblast važenja
  ugnežđivanje unutrašnje klase unutar proizvoljne oblasti važenja, 274
  unutrašnje klase u metodama i oblastima važenja, 273
oblik
  primer, 22, 215
  primer, i prepoznavanje tipa u vreme izvršavanja, 432
odeljak, kritičan odeljak i sinhronizovani blok, 933
oduzimanje, 69
ograničena svojstva, 1137
okno s jeziščima, 1083
okruženja koja brzo reaguju, 913
oktalno, 78
okviri za poruke, u Swingu, 1083
OOP
  osnovna obeležja, 16
  osnovni koncepti, 15
  protokol, 243

Simula-67, programski jezik, 17
zamenljivost, 17
opcione metode, u Java kontejnerima, 646
OpenLaszlo, alternativa za Flex, 1138
operacija, atomska, 926
operativni sistem, izvršavanje programa iz Jave, 752
operator označenog pomeranja udesno (>>), 80
operator pomeranja ulevo(<<), 80
operator prvog komplementa, 80
operator umanjenja, 72
operator uslovljavanja, 84
operator uvećanja, 72
operatori, 66
+, za String, 393
binarni, 80
česte greške, 86
konverzija u tip String operatorom +, 67, 85
logički, 75
logički operatori i nepotpuno izračunavanje, 76
nad bitovima, 79
operator indeksiranja [], 145
pomeranja, 80
poređenja, 73
preklapanje, 85
preklapanje + i += za String, 184
preklapanje operatora za String, 393
prioriteti, 66
prvog komplementa, 80
ternarni, 84
unarni, 71, 80
za eksplicitnu konverziju tipova, 87
zarez kao operator, 103
ordinal( ), za nabrojane tipove, 806
OSIzvršenje, 752
osnova, 8, 16, 78
osnovna klasa, 172, 183, 214
apstraktna osnovna klasa, 239
inicijalizacija, 185
interfejs, 218
konstruktor, 225
osnovni koncepti objektno orijentisanog programiranja (OOP), 15
osnovni tipovi, 22
otkrivanje tipa
dvokratno, 836
višekratno nabrojani tipovi, 836
OutputStream, 730, 732
OutputStreamWriter, 735
označen
break, 108
continue, 108
oznaka, 108

P

padajuća lista, 1080
paintComponent( ), 1093, 1098
paket, 160
i struktura direktorijuma, 168
imena, pisanje velikim slovima, 52
paketni i prijateljski pristup, 169
paketni pristup i zaštićeni članovi, 197
podrazumevani, 161, 170
pravljenje jedinstvenih imena paketa, 163
pakovanje, 326, 495
i generički tipovi, 497, 549
paralelno izvršavanje
aktivni objekti, 918, 1039
ArrayBlockingQueue, 971
BlockingQueue, 971, 988
Brajanovo pravilo sinhronizacije, 922
cepanje reči, 926
Condition, klasa, 968
CountDownLatch, 983
CyclicBarrier, 986
DelayQueue, 988
eksplicitne brave, 923
Exchanger, 1001
Executor (izvršilac), 893
gašenje zadataka, 942
Gecov test za izbegavanje sinhronizacije, 926
i izuzeci, 924
i kontejneri, 708
i Swing, 1110
interfejs Callable, 896
kôd bez zaključavanja, 926
konstruktori, 907
LinkedBlockingQueue, 971
lokalno skladište niti, 940
neprekidnost, 918
nit u odnosu na zadatak, terminologija, 911
operacije na prostim tipovima long i double nisu neprekidne, 926
optimizacija performansi, 1018
prioritet, 899
PriorityBlockingQueue, 991
proizvođač potrošač, 965

propušteni signali, 961
ReadWriteLock, 1036
ScheduledExecutor, 994
semafor, 997
servisne niti, 901
sleep( ), 897
SynchronousQueue, 1008
takmičenje, za brave, 1019
ulazno/izlazne operacije između zadataka, realizovane pomoću cevi, 976
UncaughtException-Handler, 916
uslov za trku, 919
parametar kolekcije, 565, 589
parametrizovani tipovi, 484
performanse
i finalnost, 207
nio, 770
optimizacija, za paralelno izvršavanje, 1018
PhantomReference, 709
PipedInputStream, 731
PipedOutputStream, 731–732
PipedReader, 735, 976
PipedWriter, 735, 976
pisanje imenaa paketa velikim slovima, 52
pisanje imena sličnih grbama kamile, 63
Plauger, P.J., 1172
početni kapacitet, objekata tipa HashMap ili HashSet, 700
podaci
finalni, 200
inicijalizacija statičnih, 140
prosti tipovi podataka i kako se upotrebljavaju sa operatorima, 89
podela i smeštanje koda, 168
podobjekat, 185, 196
podrazumevani konstruktor, 124
dobija isti nivo pristupa kao klasa, 464
pravljenje, 186
podrazumevani paket, 161, 170
podupirači, raspoređivača BoxLayout, 1059
pohlepni kvantifikatori, 413
pokazivač, nema ga u Javi, 288
polimorfizam, 26, 211, 238, 432, 482
i konstruktori, 224
i višekratno otkrivanje tipa, 836
ponašanje polimorfnih metoda unutar konstruktora, 230
položaj, apsolutan, prilikom raspoređivanja Swing komponenata, 1059
polja, inicijalizacija u interfejsima, 259
polje za potvrdu, 1077
pomeranje sadržaja u Swingu, 1055
ponovna podela na proste faktore, 159
ponovna upotreba koda, 180
ponovno generisanje izuzetka, 359
ponovno transformisanje ključeva (heširanje), 701
poređenje nizova, 618
poređenje performansi, 1019
poruka, slanje, 17
posesivni kvantifikatori, 413
posrednik
i klasa java.lang.ref.Reference, 648
za nepromenljive metode klase Collections, 648, 710
potpis, metode, 50
povezano svojstvo, 1137
povezivanje, klase, 440
povratni poziv, 720, 1052
i unutrašnje klase, 288
prazna finalna polja, 203
pražnjenje bafera izlaznih datoteka, 741
prebacivanje, konteksta u paralelnom izvršavanju, 887
prečice, s tastature, 1090
preduslovi za razumevanje knjige, 15
Preferences, API, 802
prefiksno umanjenje, 72
prefiksno uvećanje, 72
prekidanje ili nastavljanje, obrada izuzetaka, 349
preklapanje
generičkih metoda, 553
i konstruktori, 117
izostanak sakrivanja imena tokom nasleđivanja, 194
metoda, 117
operatora + i += za String, 184, 393
povratnih vrednosti, 123
razlikovanje preklopljenih metoda, 119
u odnosu na redefinisanje, 194
prekoračenje, i prosti tipovi, 98
prenosivost u jezicima C, C++ i Java, 89

prepoznavanje tipa u vreme izvršavanja (RTTI), 237
ClassCastException, 445
Constructor, klasa za refleksiju, 462
Field, klasa, 462
getConstructor( ), 1071
instanceof, rezervisana reč, 445
isInstance( ), 453
Method, klasa, 462
newInstance( ), 1071
objekat tipa Class, 434, 1071
primer oblika, 432
razlika između refleksije i, 462
refleksija, 461
svođenje naniže koje čuva tip, 445
zloupotreba, 482
pretraživanje
niza, 624
uređivanje i pretraživanje Listi, 705
preusmeravanje standardnog ulaza/izlaza, 751
prevodilac unazad, javap, 393, 479, 520
prevođenje Java programa, 56
prijavljivanje grešaka u knjizi, 14
prijemnik
adapteri, 1065
i događaji, 1060
interfejsi, 1064
primena metode na sekvencu, 577
primordijalni učitavač klasa, 434
printf( ), 400
printStackTrace( ), 357, 359
PrintStream, 734
PrintWriter, 736, 740–742
prigodni konstruktor u Javi SE5, 746
prioritet, paralelno izvršavanje, 899
PriorityBlockingQueue, za paralelno izvršavanje, 991
PriorityQueue, 330, 657
PriorityQueue, prioritetni red za čekanje, 330, 657
prirodni logaritmi, 79
priručni saveti, 1073
pristup
klasi, 175
kontrola, 159, 178
kršenje kontrole pristupa pomoću refleksije, 476
paketni i prijateljski pristup, 169
specifikatori, 20, 159, 168
unutar direktorijuma, preko podrazumevanog paketa, 171
unutrašnja klasa i prava pristupa, 268
private, rezervisana reč, 20, 159, 168, 171, 197, 922
iluzija preklapanja privatnih metoda, 205
interfejsi, kada su ugnežđeni, 261
metode, 232
redefinisanje metoda, 221
unutrašnje klase, 291
proces, paralelan, 887
ProcessBuilder, 752
ProcessFiles, 878
produžavanje
uz očuvanje znaka, 80
uz dodavanje nula, 80
program
builder, 1121
kostur, 290
program za uređivanje teksta, pravljenje pomoću Swing klase JTextPane, 1076
programer klijent, 20
programer, klijent
u odnosu na autora biblioteke, 159
programiranje
Extreme Programming (XP), 1171
multistandardno, 16
objektno orijentisano, 432
osnovni koncepti OOP-a, 15
programi vođeni događajima, 1052
proizvođač potrošač, u paralelnom izvršavanju, 965
projektni obrazac
Adapter, 250, 257, 495, 582, 585, 631
adapterska metoda, 338
Chain of Responsibility, 826
Command, 295, 473, 822, 893
Decorator, 568
Façade, 452
Factory Method, 262
Flyweight, 635, 1043
Iterator, 269, 315
Null iterator, 469
Null objekat, 469
objekat za prenos podataka (idiom Dostavljač poruka), 487, 632, 684
Proxy, 465

Singleton, 177
Strategy, 247, 256, 585, 606, 619, 621, 720, 726, 826, 990
Template Method, 290, 449, 684, 772, 937, 1025, 1030
Visitor, 861
projektovanje, 235
dodavanje metoda projektu, 179
i kompozicija, 233
i nasleđivanje, 233
projekat biblioteke, 159
promenljiv izgled i ponašanje, 1103
promenljive
definisanje, 102
inicijalizacija u metodama, 136
liste promenljivih parametara (nepoznat broj i tip argumenata), 150
lokalne, 50
promene, vektor, 292
PropertyChangeEvent, 1137
PropertyDescriptor, 1126
ProptertyVetoException, 1137
propušteni signali, u paralelnom izvršavanju, 961
prosti tipovi
finalni, 201
finalni statični prosti tipovi, 202
inicijalizacija polja klase, 136
poređenje, 73
tipovi, 44
tipovi podataka, i kako se upotrebljavaju sa operatorima, 89
prostor
imenski prostori, 160
prostor problema, 15
prostor rešenja, 15
proširiv program, 218
proširivanje klase tokom nasleđivanja, 23
proširivanje, u odnosu na potpuno nasleđivanje, 235
proširujuća konverzija, 87
protected, rezervisana reč, 20, 159, 168, 172, 197
i paketni pristup, 197
takođe daje paketni pristup, 174
protokol, 243
provera granica niza, 147
provereni izuzeci, 356, 384
pretvaranje u neproverene, 388
Proxy, projektni obrazac, 465
public, 159, 168–169
i rezervisana reč interface, 243
klasa i jedinice za prevođenje, 161
public, rezervisana reč, 20
puno ime, 438
PushbackInputStream, 733
PushbackReader, 736
Python, 1, 3, 6, 36, 41, 572, 626, 888, 1173

R

RAD (okruženje za brzo razvijanje aplikacija), 461
radio-dugme, 1078
random( ), metoda za generisanje slučajnih brojeva, 326
RandomAccess, interfejs za kontejnere, 344
RandomAccessFile, 737, 744, 755
rano vezivanje, 27, 214
raspored, kontrolisanje pomoću rasporedivača, 1056
rastuće angažovanje, teorija, 915
razdvajanje interfejsa i realizacije, 20, 174, 1060
različito (!=), 73
read( ), 730
nio, 755
readDouble( ), 743
Reader, 730, 734–735
readExternal( ), 785
readLine( ), 379, 736, 741, 750
readObject( ), 781
dodavanje metode interfejsu Serializable, 791
ReadWriteLock, 1036
realizacija, 18
i interfejs, 196, 243
i razdvajanje od interfejsa, 20, 174, 1060
sakrivanje, 159, 174, 271
rečnik, 306
red za čekanje, 302, 319, 329, 657
performanse, 688
sinhronizovan, u paralelnom izvršavanju, 971
redefinisanje
funkcije, 24
i unutrašnje klase, 297
privatne metode, 221
u odnosu na preklapanje, 194

redosled
  inicijalizacije, 139, 208, 232
  pozivanja konstruktora s nasleđivanjem, 224
ReentrantLock, 925, 953
reference
  dodela objekata kopiranjem referenci, 67
  jednakost referenci u odnosu na jednakost objekata, 73
  null, 46
  otkrivanje tačnog tipa na koji upućuje referenca na osnovnu klasu, 434
Reference, klasa paketa java.lang.ref, 709
referenciranje unapred, 138
refleksija, 461, 1061, 1123
  i neobavezno proveravanje tipova, 387
  i zrna Jave, 1121
  latentni tipovi i generički kôd, 576
  poništavanje kapsuliranja pomoću, 476
  primer, 1070
  procesor anotacija, 848, 854
  razlika izmedu RTTI i refleksije, 462
regex, paket, 411
registrovane proizvodne metode, varijanta projektnog obrasca Factory Method, 456
regularni izrazi, 408
rekurzija, nenamerna, koju prouzrokuje metoda toString( ), 397
removeActionListener( ), 1128, 1134
removeXXXListener( ), 1060
renameTo( ), 730
reset( ), 737
resume( ) i uzajamna blokada, 946
rewind( ), 759
rezervisani kvantifikatori, 413
RoShamBo, 836
Rumbaugh, James, 1172
RuntimeException, 366, 388

S

sabiranje, 69
sakrivanje realizacije, 174
sakupljanje smeća, 129, 131
  dostižni objekti, 709
  i čišćenje, 191
  kako radi skupljač, 133
  redosled čišćenja objekata, 194
samoograničeni tipovi u generičkom kodu, 555
savršena funkcija za transformisanje ključeva, 675
ScheduledExecutor, za paralelno izvršavanje, 994
seek( ), 737, 744
sekvenca, primena metode na sekvencu, 577
semafor, 997
seminari
  javni o Javi, 9
  obuka koju pruža MindView, Inc., 1166
SequenceInputStream, 731, 737
Serializable, 781, 785, 789, 798, 1130
  readObject( ), 791
  writeObject( ), 791
serijalizovanje
  defaultReadObject( ), 792
  defaultWriteObject( ), 792
  i rezervisana reč transient, 789
  i skladištenje objekata, 793
  upravljanje postupkom serijalizovanja, 785
  zadavanje verzije, 793
servisne niti, 901
Set, 302, 306, 322, 652
  matematički odnosi, 504
  poređenje performansi, 696
setActionCommand( ), 1090
setBorder( ), 1075
setErr(PrintStream), 751
setIcon( ), 1073
setIn(InputStream), 751
setLayout( ), 1056
setMnemonic( ), 1090
setOut(PrintStream), 751
setToolTipText( ), 1073
shuffle( ), 706
signali, propušteni, u paralelnom izvršavanju, 961
Simula-67, programski jezik, 17
simulacija šalterskog službenika, 1003
SingleThreadExecutor, 895
singularni projektni obrazac, 177
sinusoida, 1092
sirov tip, 513
size( ), metoda klase ArrayList, 303

sizeof( ), operator
za određivanje veličine koji Java nema, 89
skok, bezuslovni, 106
skupna inicijalizacija nizova, 146
slanje poruke, 17
sleep( ), u paralelnom izvršavanju, 897
Smalltalk, 16
SocketChannel, 773
SoftReference, 709
softver, konferencija za razvoj, 9
SortedMap, 666
SortedSet, 656
South, BorderLayout, 1056
specifikacija, specifikacija izuzetaka, 355, 385
specifikator pristupa, 20, 159, 168
specijalizacija, 197
split( ), metoda klase String, 247, 409
sporedan efekat, 66, 72, 123
sprintf( ), 406
SQL kôd generisan putem anotacija, 850
stanje okončanja i finalize( ), 131
standardni ulazni tok, čitanje, 749
stateChanged( ), 1095
static, 243
finalni statični prosti tipovi, 202
i finalnost, 201
inicijalizacija, 209, 436
inicijalizacija statičnih podataka, 140
metoda, 129, 215
obavezno proveravanje tipova, 384
provera tipova, 483
rezervisana reč, 53, 129
sinhronizovane statičke metode, 922
statička odredba konstruktora, 143
statički blok, 143
u odnosu na dinamičku proveru tipova, 647
unutrašnje klase, 282
uvoz i nabrojani tipovi, 806
stek, 319, 320, 714
generički s kojeg se prvo uzima ono što je na njega poslednje stavljeno, 490
stil
stil programiranja, 63
pravljenja klasa, 174
STL, C++, 718
stop( ) i uzajamna blokada, 946
Strategy, projektni obrazac, 247, 256, 585, 606, 619, 621, 720, 726, 826, 990
StreamTokenizer, 736
String
CASE_INSENSITIVE_ ORDER Comparator, 705
format( ), 406
indexOf( ), 464
konverzija pomoću operatora +, 67, 85
leksikografsko u odnosu na abecedno uređivanje, 623
metoda toString( ), 181
metode, 398
nadovezivanje operatorom +=, 85
nepromenljivost, 392
podrška za regularne izraze u klasi, 409
preklapanje operatora + i +=, 184
split( ), metoda, 247
uređivanje, CASE_INSENSITIVE _ORDER, 720
format( ), 406
StringBuffer, 731
StringBufferInputStream, 731
StringBuilder, poređenje s klasom String i metodom toString( ), 394
StringReader, 735, 739
StringWriter, 735
Stroustrup, Bjarne, 158
strukturni tipovi, 572
sufiksno, 72
sufiksno umanjenje, 72
sufiksno uvećanje, 72
sukob
tokom heširanja, 675
ime, 165
super, 186
i unutrašnje klase, 297
rezervisana reč, 185
supstitucija
nasleđivanje u odnosu na proširivanje, 235
princip, 25
suspend( ) i uzajamna blokada, 946
sužavajuća konverzija, 87
svite, @Unit u odnosu na JUnit, 874
svođenje naniže, 200, 236
koje čuva tip, 445
svođenje naviše, 28, 198, 211
i interfejs, 215
i prepoznavanje tipa u vreme izvršavanja, 434
i unutrašnje klase, 271

svojstvo, 1121
indeksirano, 1137
namenska klasa za uređivanje svojstava, 1137
namenski spisak svojstava, 1137
ograničena svojstva, 1137
povezana svojstva, 1137
SWF, Flashov bajtkod format, 1138
Swing, 1045
i paralelno izvršavanje, 1110
komponente, korišćenje HTML-a sa, 1101
model događaja, 1059
primeri komponenata, 1068
switch
i enum, 809
rezervisana reč, 112
synchronized, rezervisana reč, 921
blok i kritičan odeljak, 934
Brajanovo pravilo sinhronizacije, 922
i metode wait( ) i notifyAll( ), 957
i nasleđivanje, 1135
odlučivanje koje metode sinhronizovati, 1134
red za čekanje, 971
sinhronizovani kontejneri, 708
statičko, 922
SynchronousQueue, za paralelno izvršavanje, 1008
System.arraycopy( ), 617
System.err, 350, 749
System.in, 749
System.out, 749
System.out, omotavanje toka u PrintWriter, 750
systemNodeForPackage( ), API Preferences, 803

Š

šabloni, C++, 485, 513

T

tabela baze podataka, generisana SQL kodom putem anotacija, 850
takmičenje, za brave, u paralelnom izvršavanju, 1019
tastatura
navigacija, i Swing, 1046
prečice, 1090
Template Method, projektni obrazac, 290, 449, 684, 772, 845, 850, 937, 1025, 1030
Teorija rastućeg angažovanja, 915
ternarni operator, 84
testiranje
jedinica na osnovu anotacija @Unit, 864
jedinice, 184
tehnike, 284
this, rezervisana reč, 125
ThreadFactory, namenska realizacija interfejsa, 902
throw, rezervisana reč, 347
Throwable, natklasa od Exception, 356
Timer, objekat tipa, 964
TimeUnit, 898, 990
tip
bezbednost tipova u Javi, 86
dinamička bezbednost tipova i kontejneri, 562
ekvivalencija tipa podataka i klase, 17
generički tipovi i kontejneri za bezbedan rad s tipovima, 303
izveden, 22
latentni tipovi, 572, 582
osnovni, 22
otkrivanje tačnog tipa na koji upućuje referenca na osnovnu klasu, 434
oznaka tipa u generičkom kodu, 522
parametrizovani, 484
prosti, 44
prosti tipovi podataka i kako se koriste sa operatorima, 89
provera tipova i nizovi, 593
statička provera, 384, 483
strukturni tipovi, 572
svođenje naniže koje čuva tip, 445
zaključivanje o tipovima generičkih argumenata, 497
To, 646
toArray( ), 700
tok, U/I, 730
TooManyListenersException, 1130
toString( ), 181
smernice za upotrebu klase StringBuilder, 396
trajnost, 793
laka trajnost, 781
traka napredovanja, 1102
transferFrom( ), 756
transferTo( ), 756
transformisanje ključeva (heširanje), 672, 674
i ključevi za heširanje, 668
savršena funkcija za, 675
spoljno nadovezivanje, 675

transient, rezervisana reč, 789
traženje .class datoteka tokom učitavanja, 163
TreeMap, 664, 666, 700
TreeSet, realizacija skupa Set, 323, 653, 655–656, 696
true, 75
try, rezervisana reč, 194, 368
  blok try u izuzecima, 348
tryLock( ), zaključavanje datoteke, 773
TYPE polje, za literale prostih klasa, 439

U

U/I
  available( ), 740
  biblioteka, 719
  biblioteka za komprimovanje, 775
  blokiranje i metoda available( ), 740
  BufferedInputStream, 733
  BufferedOutputStream, 734
  BufferedReader, 377, 736, 738
  BufferedWriter, 736, 740
  ByteArrayInputStream, 731
  ByteArrayOutputStream, 732
  cev, 731
  cevovodi, 745
  CharArrayReader, 735
  CharArrayWriter, 735
  CheckedInputStream, 775
  CheckedOutputStream, 775
  close( ), 738
  DataInput, 737
  DataInputStream, 733, 736, 739
  DataOutput, 737
  DataOutputStream, 734, 737
  DeflaterOutputStream, 775
  direktorijum, pravljenje direktorijuma i putanja, 728
  Externalizable, 785
  File, klasa, 719, 731, 737
  File.list( ), 719
  FileDescriptor, 731
  FileInputReader, 738
  FileInputStream, 731
  FilenameFilter, 719
  FileOutputStream, 732
  FileReader, 377, 735
  FileWriter, 735, 740
  FilterInputStream, 731
  FilterOutputStream, 732
  FilterReader, 736
  FilterWriter, 736
  GZIPInputStream, 775
  GZIPOutputStream, 775
  InflaterInputStream, 775
  InputStream, 730
  InputStreamReader, 735
  internacionalizacija, 735
  izlaz, 730
  izmedu zadataka, realizovane pomoću cevi, 976
  karakteristike datoteka, 728
  koje se mogu prekidati, 950
  laka trajnost, 781
  LineNumberInputStream, 735
  LineNumberReader, 736
  listanje direktorijuma, 719
  mark( ), 737
  mkdirs( ), 730
  mrežni ulaz/izlaz, 753
  nove U/I klase (nio), 753
  ObjectOutputStream, 781
  OutputStream, 730, 732
  OutputStreamWriter, 735
  PipedInputStream, 731
  PipedOutputStream, 731–732
  PipedReader, 735
  PipedWriter, 735
  preusmeravanje standardnog ulaza/izlaza, 751
  primeri jednostavne upotrebe, 737
  PrintStream, 734
  PrintWriter, 736, 740–742
  PushbackInputStream, 733
  PushbackReader, 736
  RandomAccessFile, 737, 744
  read( ), 730
  readDouble( ), 743
  Reader, 730, 734–735
  readExternal( ), 785
  readLine( ), 379, 736, 741, 750
  readObject( ), 781
  renameTo( ), 730
  reset( ), 737
  seek( ), 737, 744
  SequenceInputStream, 731, 737
  Serializable, 785
  setErr(PrintStream), 751
  setIn(InputStream), 751
  setOut(PrintStream), 751
  standardnog ulaznog toka, 749
  StreamTokenizer, 736
  StringBuffer, 731

StringBufferInputStream, 731
StringReader, 735, 739
StringWriter, 735
System.err, 749
System.in, 749
System.out, 749
tipične U/I konfiguracije, 737
transient, rezervisana reč, 789
ulaz, 730
Unicode, 735
upravljanje postupkom serijalizovanja, 785
write( ), 730
writeDouble( ), 743
writeExternal( ), 785
writeObject( ), 781
Writer, 730, 734–735
ZipEntry, 778
ZipInputStream, 775
ZipOutputStream, 775
učitavač klasa, 434
učitavanje
datoteka .class, 163
inicijalizacija i učitavanje klasa, 208
klase, 209, 440
ugnežđena klasa (statična unutrašnja klasa), 282
ugnežđivanje interfejsa, 260
ugradivanje poziva metoda direktno u kôd, 204
uklapanje konstante, 201
ulančani izuzeci, 362, 388
ulazno/izlazne operacije koje se mogu prekidati, 950
umanjenje, operator, 72
UML
jezik za modelovanje, 19, 21, 1172
pokazivanje kompozicije, 21
unapređenje tipova, u int, 89, 98
unarni
minus (-), 71
operator, 80
operatori, 71
plus (+), 71
UncaughtExceptionHandler, klase Thread, 916
Unicode, 735
Unified Modeling Language (UML, unifikovani jezik za modelovanje), 19, 21, 1172
unmodifiableList( ), metoda klase Collections, 648
UnsupportedOperation-Exception, 647
unutrašnja klasa, 266
anonimna, 721, 1053
generička, 508
i kôd upravljan tabelom, 684
i kosturi upravljanja, 290
i niti, 907
i redefinisanje, 297
i rezervisana reč super, 297
i svođenje naviše, 271
i Swing, 1060
identifikatori i .class datoteke, 300
lokalna, 274
motivacija, 285
nasleđivanje, 296
povratni poziv, 288
prava pristupa, 268
privatna, 291
skrivena referenca objekta okružujuće klase, 269
statične unutrašnje klase, 282
u metodama i oblastima važenja, 273
ugnežđivanje unutar proizvoljne oblasti važenja, 274
upućivanje na objekat spoljne klase, 270
zaključak, 287
upravljanje procesima, 752
uređivanje, 619
abecedno, 325
i pretraživanje listi, 705
leksikografsko, 325
userNodeForPackage( ),API Preferences, 803
uslov za trku, u paralelnom izvršavanju, 919
uslovno prevođenje, 168
uslužne metode, klase java.util.Collections, 701
uvećanje, operator, 72
i paralelno izvršavanje, 920
uzajamna blokada uprkos izvršavanja, 1043
uzajamna blokada, u paralelnom izvršavanju, 978
uzajamno isključivanje (mutex), u paralelnom izvršavanju, 921
uzorak, regularnog izraza, 412

V

values( ), za nabrojane tipove, 805, 810
Varga, Ervin, 5, 951
Vector, 694, 713
večera filozofa, primer uzajamne blokade u paralelnom izvršavanju, 978

veće ili jednako (>=), 73
veće od (>), 73
vektor promene, 292
veličina, objekata tipa HashMap ili HashSet, 700
Venners, Bill, 131
vertikalna kompatibilnost, 516
vezivanje
  dinamičko, 214
  dinamičko, kasno, ili vezivanje prilikom izvršavanja, 211
  kasno, 27, 214
  poziva metoda, 214
  prilikom izvršavanja, 214
  rano, 27
vezivna funkcija, 476
Visitor (projektni obrazac)
  i anotacije, API mirror, 861
Visual BASIC, Microsoftov, 1120
višedimenzionalni nizovi, 599
višekratno otkrivanje tipa
  i nabrojani tipovi, 836
  pomoću mape EnumMap, 842
višeprogramski rad, 887
višesmerni, 1130
  događaj, i zrna Jave, 1130
višestruko nasleđivanje, u jezicima C++ i Java, 251
višestruko ugnežđena klasa, 284
vizuelno programiranje, 1120
  okruženja, 1046
volatile, modifikator, 918, 926, 930
vraćanje rezultata
  i finally, 371
  kovarijantni povratni tipovi, 232, 559
  povratna vrednost konstruktora, 117
  preklapanje povratnih vrednosti, 123
  u obliku niza, 597
  vraćanje više objekata, 487
vrednost, sprečavanje izmena u vreme izvršavanja, 200

W

wait( ), 957
Waldrop, M. Mitchell, 1173
WeakHashMap, 664, 712
WeakReference, 709
Web Start, Java, 1105
West, BorderLayout, 1056
while, 100
windowClosing( ), 1096
write( ), 730
  nio, 756
writeDouble( ), 743
writeExternal( ), 785
writeObject( ), 781
  dodavanje metode interfejsu Serializable, 791
Writer, 730, 734–735

X

XDoclet, 845
XML, 799
XOM, XML biblioteka, 799
XOR (isključiva disjunkcija), 80
xPos, 797–798

Y

yPos, 797–798

Z

zadavanje verzije, serijalizovanje, 793
zadržavanje pokazivača miša iznad dugmeta bez pritiskanja, 1073
zahtev, u OOP, 17
zajednički interfejs, 239
zaključak, i unutrašnje klase, 288
zaključavanje, datoteke, 772–773
zaključivanje, o tipu generičkog argumenta, 497
zamenljivost, u OOP, 17
zapisivanje, ugradnja u izuzetke, 352
zarez kao operator, 103
zauzetost čekanjem, u paralelnom izvršavanju, 957
ZipEntry, 778
ZipInputStream, 775
ZipOutputStream, 775
zrna Jave
  alatka za pravljenje aplikacija, 1121
  događaji, 1121
  EventSetDescriptor, 1126
  FeatureDescriptor, 1137
  getBeanInfo( ), 1124
  getEventSetDescriptors( ), 1126
  getMethodDescriptors( ), 1126
  getName( ), 1126
  getPropertyDescriptors( ), 1126

getPropertyType( ), 1126
getReadMethod( ), 1126
getWriteMethod( ), 1126
i Borlandov Delphi, 1120
i Microsoftov Visual BASIC, 1120
indeksirano svojstvo, 1137
Introspector, 1124
JAR datoteke za arhiviranje, 1135
komponenta, 1121
manifest datoteka, 1135
Method, 1126
MethodDescriptor, 1126
namenska klasa BeanInfo, 1137
namenska klasa za uređivanje svojstava, 1137
namenski spisak svojstva, 1137
ograničena svojstva, 1137
pravilo za imenovanje, 1122
PropertyChangeEvent, 1137
PropertyDescriptor, 1126
ProptertyVetoException, 1137
refleksija, 1121, 1123
Serializable, 1130
svojstva, 1121
vizuelno programiranje, 1120