Programiranje u Javi

Marko Čupić

Programiranje u Javi

Marko Čupić

Ovo je verzija 0.3.26 knjige o programiranju u programskom jeziku Java. Napisani tekst odnosi se na verziju 8 programskog jezika Java.
Autorska prava © 2012. - 2015. Marko Čupić

Napomene

Uporaba ove knjige dozvoljava se studentima FER-a koji je žele koristiti u svrhu učenja programskog jezika Java. Za bilo koji drugi slučaj molim javite se elektroničkom poštom na adresu autora (Marko.Cupic@fer.hr).

Posveta

Posvećeno svima koji žele naučiti programirati, a potom i programirati u Javi.

Sadržaj

Predgovor xxi
1. Java – jezik, platforma, ekosustav 1
Postupak prevođenja i izvođenja Java programa 3
Javin virtualni stroj 5
Jezici podržani od JVM-a 6
Podržani tipovi podataka 7
Primjeri literala 9
Inicijalizacija vrijednosti 11
Pretvorbe primitivnih tipova u string 11
Omotači primitivnih tipova 12
Uvjetno izvođenje, petlje i decimalni brojevi 13
2. Prvi Java program 21
Prevođenje programa 23
Pokretanje izvršnog programa 24
Složeniji primjer 25
Pakiranje izvršnog programa u arhivu 27
Generiranje dokumentacije 29
Pisanje dokumentacije projekta 30
Dokumentiranje paketa 30
Dokumentiranje razreda 30
Dokumentiranje metoda 31
Automatizacija razvojnog ciklusa 32
Uporaba alata **ant** 32
Uporaba alata **maven** 38
Uporaba alata **gradle** 46
3. Osiguravanje kvalitete i ispravnosti koda 49
Alat CheckStyle 54
Alat PMD 56
Alat FindBugs 59
Alat JUnit 61
Alat JaCoCo 67
4. Jednostavniji primjeri 75
Primjer 1: ispis argumenata naredbenog retka 75
Primjer 2: izračun e^x 76
Primjer 3: formatirani ispis decimalnih brojeva 78
Primjer 4: izračun sume brojeva s tipkovnice 80
Naprednije čitanje ulaznih podataka 81
Formatirana izrada teksta 82
Stringovi u Javi 82
Primjer 5: metode s varijabilnim brojem argumenata 88
5. Razredi 93
Motivacijski primjer 93
Java je objektno-orijentirani programski jezik 98
Konstruktori 100
Destruktori 102
Delegiranje zadaće konstrukcije objekta 103
Prvi razredi 105
Zaštita privatnosti elemenata razreda 122
Na vrhu stabla: razred `Object` 123
`String toString();` 124
`int hasCode();` 125
`boolean equals(Object obj);` 125
Metode `wait`, `notify` i `notifyAll` 126
`protected void finalize()` 126
`Class<?> getClass();` 126

Inicijalizacijski blokovi 128
6. Upravljanje pogreškama. Iznimke. 131
Iznimke 134
Obrada više iznimaka 141
Bezuvjetno izvođenje koda 143
Automatsko upravljanje resursima 145
Definiranje novih vrsta iznimaka 146
7. Apstraktni razredi. Sučelja. 149
Apstraktni razredi 149
Sučelja 153
Primjer uporabe sučelja 156
Modifikator `static` 161
Vrste razreda 166
Modifikator `final` 175
Programiranje prema sučeljima 176
Motivacijski primjer 178
Oblikovni obrazac Iterator 181
Naprednije mogućnosti sučelja 183
Lambda izrazi 184
Funkcijska sučelja 194
8. Načela objektno-orijentiranog oblikovanja 195
Na čvrstim temeljima 197
Načelo jedinstvene odgovornosti 199
Načelo otvorenosti za nadogradnju a zatvorenosti za promjene 204
Liskovino načelo supstitucije 210
Načelo izdvanja sučelja 221
Načelo inverzije ovisnosti 226
9. Tehnologija Java Generics 239
Parametrizirani kod u Javi 241
Parametrizirani razredi i sučelja 245
Složeniji primjer 252
Odnosi tipova podataka uz parametrizaciju 255
Dodatna ograničenja pri radu s parametriziranim tipovima 256
10. Kolekcije 259
Motivacija 259
Elementi Javinog okvira kolekcija 261
Sučelje `java.util.Collection` 267
11. Kolekcije: skup 271
Konkretne implementacije skupova 271
Primjeri uporabe skupova 272
12. Kolekcija: lista 277
Sučelje `java.util.List` 277
Implementacije lista 280
Pomoćne metode 282
Primjeri 283
Zadatak 1 283
Zadatak 2 285
13. Kolekcija: mapa 289
Sučelje `java.util.Map` 290
Implementacije sučelja `java.util.Map` 295
Primjer uporabe 296
14. Kolekcije i novi razredi 301
Metoda `equals` 302
Metoda `hashCode` 307
Prirodan poredak i komparatori 312
Još malo o komparatorima 317
Izravna i reverzna usporedba 317
Usporedba po više kriterija 319

Izrada jednostavnih komparatora 323
15. Obrada podataka i tokovni API 325
Obrada modelom toka 334
Izvori 334
Međuobrada podataka 337
Završna obrada podataka 341
Izvori i poredak elemenata 353
16. Rad s datotečnim sustavom 355
Razred `java.io.File` 355
Paket java.nio.file 362
Razred `java.nio.file.Files` 367
Preostali važniji razredi 381
17. Tokovi okteta i znakova 383
Tokovi okteta: motivacija 384
Ulazni tokovi okteta 384
Izlazni tokovi okteta 408
Tokovi okteta i Javine biblioteke 409
Znakovi i okteti 423
Simboli, Java i Unicode 428
Tokovi znakova 430
Primjer uporabe 438
Oblikovni obrazac Okvirna metoda 454
Datoteke sa slučajnim pristupom 455
18. Studija slučaja: programski jezik *vlang* 457
Osnovni razredi 458
Provedba leksičke analize 460
Izrada parsera i izgradnja sintaksnog stabla 469
Izvođenje programa: pokušaj prvi 476
Izvođenje programa: pokušaj drugi 480
Izvođenje programa: pokušaj treći 485
Korišteni oblikovni obrasci 501
Oblikovni obrazac *Kompozit* 501
Oblikovni obrazac *Posjetitelj* 504
19. Višedretvene aplikacije 509
Njezino visočanstvo: Dretva 512
Model dretve u Javi 512
Metode razreda `Thread` 514
Načini stvaranja i pokretanja dretve 515
Jednostavna sinkronizacija dretvi 518
Tipični problemi višedretvenih aplikacija 521
Uporaba instrukcije CAS 526
Izgradnja kritičnog odsječka 529
Uporaba monitora 535
TODO: 541
20. Raspodijeljene aplikacije. Paket java.net. 543
Što je Internet? 543
Lokalne mreže i IPv4 548
Dohvat informacija o mrežnim karticama 548
Dva primjera lokalne mreže 554
Protokol DNS 556
Detekcija poslužitelja koji šalju neželjenu e-poštu 561
Protokol UDP - nepouzdan prijenos podataka 561
Uporaba UDP-a u Javi 562
Protokol TCP - pouzdan prijenos podataka 568
Protokol HTTP 572
Ukratko o URL 573
Ukratko o protokolu HTTP 573
21. Grafičko korisničko sučelje: uvod u Swing 587

Vršne komponente ... 588
Životni ciklus prozora ... 592
Anatomija Swing komponente ... 596
Primjer bordera ... 602
Crtanje po površini komponente ... 606
Promjena stanja komponente nakon što je prikazana ... 610
Automatsko razmještanje komponenata ... 611
Upoznavanje s osnovnim konceptima ... 613
Ilustrativni primjeri: `StretchLayout` i `HorizontalSplitLayout` ... 616
Postojeći razredi tipa `LayoutManager` ... 626
Potpora izmjenjivom izgledu komponenata ... 635
Trenutna tema ... 636
Primjer komponente koja podržava LAF ... 637
22. Osnovne komponente Swinga ... 649
Labela i dva gumba ... 649
Pokušaj 1 ... 653
Pokušaj 2 ... 654
Izbornici i izborničke strukture ... 655
Pokušaj 3 ... 658
Konačno rješenje: sučelje `Action` ... 661
Preglednik slika. Razred `SwingWorker` ... 665
23. Grafičko korisničko sučelje: Swing - modeli i pogledi ... 673
Prikaz liste ... 674
Prikaz elementa liste ... 688
Komponenta `JSpinner` ... 694
Rad s tablicama ... 703
Komponente za prikaz i uređivanje teksta ... 710
24. Uporaba relacijskih baza podataka iz Jave: SQL ... 715
25. Uporaba relacijskih baza podataka iz Jave: JPA ... 717
26. Razvoj web aplikacija ... 719
27. Web obrasci ... 721
28. Tehnologija Java Servlet i JSP ... 723
29. Višeslojne web-aplikacije ... 725
30. IOC kontejneri ... 727
31. Test Driven Development ... 729
32. Razvoj aplikacija za Android ... 731
A. Instalacija JDK ... 733
Operacijski sustav Windows ... 733
Operacijski sustav Linux ... 734
B. Instalacija alata **ant**, **maven** i **gradle** ... 735
Alat Apache Ant ... 735
Alat Apache Maven ... 735
Alat Gradle ... 736
C. Instalacija alata za kontrolu kvalitete i ispravnosti koda ... 737
Alat CheckStyle ... 737
Alat PMD ... 738
Alat FindBugs ... 738
Alat JUnit ... 738
Alat JaCoCo ... 739
Biblioteka Mockito ... 739
D. Instalacija poslužitelja servleta ... 741
Instalacija poslužitelja Apache Tomcat ... 741
Podešavanje kodne stranice i omogućavanje praćenja izvođenja ... 742
Dodatno podešavanje kodne stranice ... 743
Instalacija poslužitelja Jetty ... 743
E. Instalacija upravitelja bazama podataka ... 745
Instalacija upravitelja Apache Derby ... 745
Podešavanje upravitelja baze ... 745

Kazalo 751
Bibliography 753

Popis slika

1.1. Od izvornog koda do programa u izvođenju 3
2.1. Pojednostavljen popis zadataka u razvojnom procesu 33
2.2. Tražilica mavenovog centralnog repozitorija 39
2.3. Rezultati pretrage za projektima/artefaktima koji sadrže "junit" 39
2.4. Prikaz svih dostupnih verzija projekta junit grupe junit 40
2.5. Detaljni prikaz informacija o projektu verzije 4.12 40
2.6. Mavenovi repozitoriji: korisnički projekti, lokalni repozitorij, javni repozitoriji 41
3.1. Sažetak izvještaja o kontroli stila pisanja koda 56
3.2. Izvještaja o kontroli stila pisanja koda datoteke `Formule.java` 56
3.3. Sažetak izvještaja o kontroli stila pisanja koda 59
3.4. Sažetak izvještaja o dupliciranju koda 59
3.5. Sažetak izvještaja o kontroli kvalitete koda 61
3.6. Izvještaj o pokrenutim testovima 66
3.7. Detaljni pregled jednog izvještaja o pokrenutim testovima 67
3.8. Izvještaj o pokrivenosti koda 72
3.9. Detaljniji izvještaj o pokrivenosti koda 72
3.10. Izvještaj o pokrivenosti koda u razredi `PrirodniTest` 72
3.11. Izvještaj o pokrivenosti koda u razredi `Prirodni` 72
3.12. Detaljni grafički prikaz o pokrivenosti koda u razredu `Prirodni` 73
4.1. Stanje u memoriji (1) 83
4.2. Stanje u memoriji (2) 83
4.3. Stanje u memoriji (3) 84
4.4. Stanje u memoriji (4) 85
5.1. Stanje u memoriji 107
5.2. Stanje u memoriji 110
5.3. Detaljno stanje u memoriji 112
5.4. Dijagram razreda 115
6.1. Struktura iznimki u Javi 138
7.1. Stanje u memoriji 162
7.2. Stanje u memoriji 164
7.3. Stanje u memoriji 169
7.4. Razredi i sučelja motivacijskog primjera 181
7.5. Dijagram razreda za oblikovni obrazac Iterator 182
8.1. Organizacija dnevničkog podsustava 209
8.2. Organizacija lošeg rješenja 223
8.3. Organizacija boljeg rješenja 226
8.4. Organizacija prvog predloženog rješenja 228
8.5. Organizacija drugog predloženog rješenja 232
8.6. Organizacija rješenja uz inverziju i injekciju ovisnosti 237
10.1. Oblikovni obrazac Strategija 267
10.2. Kolekcije izvedene iz sučelja `Collection` 269
14.1. Stanje u memoriji 304
15.1. Dijagram obrade podataka 328
15.2. Model obrade podataka 334
17.1. Primjer ulaznih tokova okteta. 388
17.2. Primjer ulaznih tokova okteta. 394
17.3. Komunikacija klijent - izvorni tok. 400
17.4. Komunikacija klijent - izvorni tok. 401
17.5. Komunikacija klijent - izvorni tok. 402
17.6. Struktura razreda razvijenih u prethodnom razmatranju 405
17.7. Oblikovni obrazac Dekorator: strukturni dijagram 408
17.8. Odabrani ulazni tokovi okteta u Javi 410
17.9. Odabrani izlazni tokovi okteta u Javi 419
17.10. ASCII tablica 423
17.11. Znakovni niz i Unicode 425

17.12. Spremnici znakova, stringovi, izvori i ponori, tokovi 431
17.13. Dio podstabla od `java.io.Reader` 434
17.14. Dio podstabla od `java.io.Writer` 435
17.15. Model domene 439
17.16. Dijagram razreda rješenja strategijom 445
17.17. Dijagram razreda rješenja okvirnom metodom 450
17.18. Strukturni dijagram oblikovnog obrasca Okvirna metoda 455
18.1. Dijagram toka obrade programa 458
18.2. Razred Vector 460
18.3. Podstablo iznimaka 460
18.4. Podstablo za pamćenje naredbi 470
18.5. Podstablo za pamćenje izraza 471
18.6. Modificirano podstablo za pamćenje izraza 481
18.7. Apstraktna operacija 486
18.8. Apstraktna operacija 488
18.9. Apstraktna operacija nad naredbama 494
18.10. Apstraktna operacija 495
18.11. Strukturni dijagram Kompozita (sigurna verzija) 502
18.12. Strukturni dijagram Kompozita (transparentna verzija) 502
18.13. Oblikovni obrazac Kompozit primijenjen na organizaciju geometrijskih likova 504
18.14. Oblikovni obrazac Posjetitelj: strukturni dijagram 507
19.1. Prikaz dretvi u programu 516
19.2. Prikaz dretvi u programu 521
19.3. Jedan mogući slijed izvođenja instrukcija dviju dretvi 524
19.4. Još jedan mogući slijed izvođenja instrukcija dviju dretvi 524
19.5. Sekvencijski dijagram za prikazano izvođenje programa 537
20.1. Izgradnja kućne lokalne mreže 543
20.2. Lokalna mreža 544
20.3. Diskretna mrežna kartica 544
20.4. ISO-OSI referentni model 544
20.5. RJ45 konektor 545
20.6. Format okvira protokola Ethernet 546
20.7. Format zaglavlja IP-paketa (verzije 4) 547
20.8. Ispis parametara mrežnih kartica na Windowsima 553
20.9. Ispis parametara mrežnih kartica na Linuxu 553
20.10. Lokalna mreža s javnim IP adresama 554
20.11. Lokalna mreža s privatnim IP adresama 554
20.12. Pojašnjenje rada tehnologije SNAT 555
20.13. Naredba nslookup 557
20.14. Organizacija DNS sustava 557
20.15. UDP-datagram 561
20.16. Jednostavan aplikacijski protokol "Zbrajanje" 562
20.17. Konceptualan prikaz komunikacije uporabom protokola TCP 569
20.18. Komunikacija uporabom protokola HTTP 574
20.19. Stranica generirana od strane našeg web-poslužitelja 582
20.20. Stranica generirana od strane našeg web-poslužitelja: izvorni kod 582
20.21. Automat za detekciju kraja zaglavlja 586
21.1. Primjer otvaranja prozora 589
21.2. Pozicioniranje komponente u roditeljskoj komponenti 596
21.3. Margine u komponenti 599
21.4. Prozor s novom komponentom i borderom 605
21.5. Prozor s novom komponentom i borderom (2) 605
21.6. Crtanje površine komponente 607
21.7. Crtanje površine komponente (2) 608
21.8. Osnovne informacije o fontu 608
21.9. Crtanje površine komponente (3) 610
21.10. `StretchLayout` na djelu (1) 617
21.11. `StretchLayout` na djelu (2) 617

21.12. `HorizontalSplitLayout` na djelu (1) 621
21.13. `HorizontalSplitLayout` na djelu (2) 621
21.14. `BorderLayout` na djelu 627
21.15. `BoxLayout` na djelu uz `BoxLayout.LINE_AXIS` 628
21.16. `BoxLayout` na djelu uz `BoxLayout.PAGE_AXIS` 629
21.17. `BoxLayout` na djelu uz `BoxLayout.LINE_AXIS` 629
21.18. `CardLayout` na djelu 630
21.19. `FlowLayout` na djelu 631
21.20. `GridLayout` na djelu 632
21.21. Složeni razmještaj 633
21.22. Složeni razmještaj (2) 634
21.23. Komponenta `Kvacica` 638
21.24. Delegati za komponentu `Kvacica` 640
21.25. Demonstracijski program 646
22.1. Demonstracijski program (1) 649
22.2. Pozicioniranje teksta i ikone u labeli 650
22.3. Modeli gumbi 650
22.4. Demonstracijski program (2) 654
22.5. Primjer izborničke strukture 656
22.6. Demonstracijski program (3) 661
22.7. Jednostavan preglednik slika 666
23.1. JList i pripadni model podataka (konceptualni prikaz) 675
23.2. JList i pripadni model podataka: jednostavan primjer 678
23.3. JList i pripadni model podataka: jednostavan primjer (2) 678
23.4. JList i pripadni model podataka: jednostavan primjer (3) 679
23.5. JList i pripadni model podataka: jednostavan primjer (4) 682
23.6. JList i pripadni model podataka: jednostavan primjer (5) 686
23.7. Model liste i pomoćne implementacije 688
23.8. Prikaz elemenata liste 690
23.9. Prikaz elemenata liste (2) 690
23.10. Prikaz elemenata liste (3) 693
23.11. Prikaz elemenata liste (4) 694
23.12. Komponenta `JSpinner` 698
23.13. Komponenta `JSpinner` (2) 700
23.14. Komponenta `JSpinner` (3) 703
23.15. Demonstracijski program za jednostavnu tablicu 708
23.16. Demonstracijski program 710
23.17. Komponente za prikaz i uređivanje teksta 711
23.18. Modeli koje koriste komponente za prikaz i uređivanje teksta 711
23.19. Demonstracijski program 713

Popis tablica

1.1. Primitivni tipovi u programskom jeziku Java 7
5.1. Razine pristupa članskim elementima razreda 123
19.1. Pregled izbora nestatičkih metoda razreda Thread 514
23.1. Sample Table 674

Popis primjera

1.1. Ilustracija računanja s ograničenom preciznošću te usporedba rezultata 17
1.2. Usporedba jednakosti decimalnih brojeva 18
2.1. Primjer programa napisanog u programskom jeziku Java 22
2.2. Složeniji primjer -- ispis slučajno odabrane formule 25
2.3. Primjer dokumentacije paketa 30
2.4. Primjer dokumentacije razreda 31
2.5. Primjer dokumentacije metode 31
2.6. Primjer konfiguracije za alat **ant** 34
2.7. Definicija cilja runjar za alat **ant** 37
2.8. Pregledna stranica za dokumentaciju programa 37
2.9. Definicija cilja javadoc za alat **ant** 37
2.10. Primjer osnovne verzije datoteke `pom.xml` 42
2.11. Zadavanje verzije prevodioca koji je potrebno koristiti u datoteci `pom.xml` 43
2.12. Definiranje kodne stranice i podataka o autoru u datoteci `pom.xml` 44
2.13. Podešavanje verzije plugina za izvještaje u datoteci `pom.xml` 45
2.14. Podešavanje izvještaja koje želimo dobiti u datoteci `pom.xml` 45
2.15. Konfiguracijska datoteka `build.gradle` za alat **gradle** 46
3.1. Implementacija biblioteke za rad s prirodnim brojevima 61
3.2. Testovi za razred `Prirodni`. 62
3.3. Datoteka `build.xml` za pokretanje testova. 63
3.4. Cjelokupna datoteka `build.xml` koja koristi alat JaCoCo 69
4.1. Ispis argumenata naredbenog retka 75
4.2. Izračun e^x 76
4.3. Formatirani ispis brojeva 78
4.4. Sumiranje brojeva unesenih preko tipkovnice 80
4.5. Konkatenacija stringova 85
4.6. Metode s varijabilnim brojem argumenata 88
5.1. Modeliranje pravokutnika u programskom jeziku C 93
5.2. Funkcije za rad s pravokutnicima u programskom jeziku C 93
5.3. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 1 96
5.4. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 2 97
5.5. Definiranje razreda `Pravokutnik`: Java 98
5.6. Definiranje razreda `Pravokutnik`: C++ 99
5.7. Definiranje razreda `Pravokutnik`: Python 99
5.8. Definiranje više konstruktora 103
5.9. Definiranje više konstruktora -- bolje 104
5.10. Razred `GeometrijskiLik` 105
5.11. Primjer uporabe razreda `GeometrijskiLik` 106
5.12. Definiranje razreda `Pravokutnik` 108
5.13. Primjer uporabe razreda `Pravokutnik` 109
5.14. Nadjačavanje metoda u razredu `Pravokutnik` 110
5.15. Definiranje razreda `Krug` 113
5.16. Definiranje razreda `Slika` i primjer uporabe 115
5.17. Dodavanje mogućnosti crtanja likova 117
6.1. Primjer stvaranja iznimke 134
6.2. Primjer obrade iznimke 136
6.3. Iznimke porodice `RuntimeException` 138
6.4. Primjer rada s resursima 143
6.5. Definiranje nove vrste iznimke 146
7.1. Razred `GeometrijskiLik` u apstraktnom izdanju 150
7.2. Razred `Krug` 151
7.3. Razred `ComplexNumber` i primjer uporabe 161
7.4. Nova inačica razreda `ComplexNumber` i primjer uporabe 163
7.5. Nadopunjena inačica razreda `ComplexNumber` i primjer uporabe 166
7.6. Primjer definicije i uporabe lokalnog razreda 170

7.7. Primjer definicije i uporabe anonimnih razreda 172
7.8. Sučelje `java.util.Iterator<E>` 176
7.9. Sučelje `java.lang.Iterable<E>` 177
7.10. Obilazak skupnog objekta koji implementira sučelje `java.lang.Iterable<E>` 177
7.11. Implementacija skupnog objekta koji predstavlja zadani podniz parnih brojeva 178
7.12. Implementacija klijenta za skupni objekt 179
7.13. Osnovno rješenje 184
7.14. Nadopunjeno rješenje 185
7.15. Poboljšano rješenje 187
7.16. Rješenje uz uporabu anonimnog razreda 188
7.17. Pomoćni razred Funkcije 190
8.1. Generiranje izvještaja, prvi pokušaj 199
8.2. Generiranje izvještaja, drugi pokušaj 200
8.3. Generiranje izvještaja, treći pokušaj 201
8.4. Razred `Student` 202
8.5. Razred `Student` i primjena načela jedinstvene odgovornosti 203
8.6. Primjer problematične metode `racunajPovrsinu` 204
8.7. Bolji primjer metode `racunajPovrsinu` 206
8.8. Primjer izrade dnevničkog podsustava 207
8.9. Bolji primjer izrade dnevničkog podsustava 208
8.10. Višestruko zapisivanje dnevničkih poruka 210
8.11. Geometrijski podsustav 211
8.12. Primjer uporabe 212
8.13. Implementacija kvadrata 213
8.14. Popravljena implementacija kvadrata 213
8.15. Nadogradnja glavnog programa 214
8.16. Moguće rješenje 215
8.17. Lokalizirane poruke 217
8.18. Poruke bez mogućnosti ažuriranja 218
8.19. Poruke bez mogućnosti ažuriranja 218
8.20. Problematična organizacija koda 221
8.21. Bolja organizacija koda 224
8.22. Podsustav za provjeru valjanosti transakcija, prvi pokušaj 227
8.23. Podsustav za provjeru valjanosti transakcija, drugi pokušaj 229
8.24. Podsustav za provjeru valjanosti transakcija, rješenje 233
9.1. Izračun srednje vrijednosti i obrasci (jezik C++) 239
9.2. Izračun srednje vrijednosti nakon uklanjanja obrasca (jezik C++) 240
9.3. Primjer parametriziranih metoda 242
9.4. Ograničavanje parametara 244
9.5. Omotač podatka tipa `Integer` 247
9.6. Uporaba omotača podatka tipa `Integer` 247
9.7. Omotač podatka tipa `Double` 248
9.8. Omotač podatka tipa `Object` 249
9.9. Uporaba omotača podatka tipa `Object` 249
9.10. Parametrizirani omotač 250
9.11. Uporaba parametriziranog omotača 252
9.12. Parametrizirano sučelje 253
9.13. Prva implementacija parametriziranog sučelja 253
9.14. Druga implementacija parametriziranog sučelja 254
9.15. Uporaba parametriziranog sučelja 254
10.1. Izračun sume brojeva 262
10.2. Izračun sume brojeva kraćim oblikom petlje `for` 263
10.3. Primjer enkapsulacije obrada u zasebne razrede 264
11.1. Ispis argumenata programa bez duplikata 272
11.2. Filtrirani ispis imena 276
12.1. Sučelje `java.util.List` 277
12.2. Rješenje zadatka 284

12.3. Rješenje 1. 285
12.4. Rješenje 2. 286
12.5. Rješenje 3. 286
12.6. Rješenje 4. 287
12.7. Rješenje 5. 287
13.1. Sučelje `java.util.Map` 290
13.2. Rješenje zadatka 296
13.3. Malo drugačije rješenje zadatka 297
13.4. Još drugačije rješenje zadatka 298
14.1. Razred `Zaposlenik` 301
14.2. `Zaposlenik` i `ArrayList` 302
14.3. `Zaposlenik` i `HashSet` 307
14.4. `Zaposlenik` i `HashMap` 308
14.5. `Zaposlenik` i `TreeSet` 312
14.6. Sučelje `java.lang.Comparable<T>` 313
14.7. Sučelje `java.util.Comparator<T>` 314
14.8. Uporaba komparatora 315
14.9. Reverzni komparator 317
14.10. Kompozitni komparator 320
14.11. Ulančani komparator 321
15.1. Razred `Zaposlenik` 325
15.2. Struktura demonstracijskog programa 326
15.3. Poopćena usluga filtriranja 328
15.4. Poboljšana inačica programa 331
16.1. Ispis separatora 356
16.2. Ispis vršnih datotečnih objekata 356
16.3. Rekurzivni ispis sadržaja direktorija 359
16.4. Ispis stabla direktorija 360
16.5. Filtrirani ispis sadržaja direktorija 361
16.6. Modeliranje staze sučeljem `Path` 363
16.7. Nadzor direktorija uporabom `WatchService` 365
16.8. Ispis sadržaja direktorija 370
16.9. Čitanje iz i zapisivanje u datoteku 371
16.10. Izračun statističkih podataka, 1. pokušaj 373
16.11. Sučelje `FileVisitor` 376
16.12. Izračun statističkih podataka, bolje 377
17.1. Primjer ulaznih tokova okteta. 384
17.2. Primjer uporabe ulaznih tokova okteta. 388
17.3. Ulazni tok modeliran apstraktnim razredom. 392
17.4. Primjer uporabe ulaznih tokova okteta. 394
17.5. Ulazni tok koji transparentno dekriptira podatke. 396
17.6. Ulazni tok koji koristi međuspremnik okteta. 397
17.7. Apstraktni dekorator i konkretni dekoratori 402
17.8. Sučelje `java.lang.AutoCloseable` 410
17.9. Sučelje `java.io.Closeable` 411
17.10. Izračun zaštitne sume zadane datoteke (1) 412
17.11. Izračun zaštitne sume ulaznog toka okteta 414
17.12. Izračun zaštitne sume ulaznog toka okteta 415
17.13. Kopiranje datoteke 420
17.14. Sažimanje podataka 421
17.15. Apstraktni razred `java.io.Reader`. 431
17.16. Apstraktni razred `java.io.Writer`. 432
17.17. Razred `Student`. 439
17.18. Razred `Predmet`. 440
17.19. Razred `Upis`. 440
17.20. Program `Upisi`. 441
17.21. Apstraktno sučelje strategije (`IObrada`) 444
17.22. Programski kod rješenja strategijom 445

17.23. Programski kod rješenja okvirnom metodom 450
18.1. Primjer programa napisanog jezikom *vlang* 457
18.2. Model nepromjenjivog vektora realnih brojeva. 458
18.3. Enumeracija različitih vrsta tokena. 461
18.4. Model jednog tokena. 462
18.5. Implementacija tokenizatora. 463
18.6. Demonstracija rada tokenizatora. 468
18.7. Gramatika jezika *vlang* 469
18.8. Implementacija parsera. 471
18.9. Demonstracija uporabe parsera. 476
18.10. Izvođenje programa (1) 476
18.11. Izvođenje programa (1) 479
18.12. Izmjene razreda koji predstavljaju izraze 481
18.13. Poboljšana izvedba izvođenja programa 483
18.14. Definicija apstraktne operacije nad izrazom 486
18.15. Uvođenje potpore za primjenu oblikovnog obrasca *Posjetitelj* u model izraza 488
18.16. Posjetitelj koji računa vrijednost izraza 490
18.17. Posjetitelj koji izraz pretvara u tekst 491
18.18. Poboljšani kod za izvođenje programa 492
18.19. Apstraktna definicija operacije nad naredbama 494
18.20. Modifikacije modela naredbe 495
18.21. Posjetitelj koji izvodi napisani program 496
18.22. Posjetitelj koji rekonstruira izvorni kod programa 498
18.23. Konačna implementacija koda koji izvodi napisani program 499
18.24. Primjer pokretanja prevođenja i izvođenja koda 500
19.1. Ispis svih grupa dretvi i dretvi po pokretanju programa 513
19.2. Modeliranje posla sučeljem `java.lang.Runnable` 515
19.3. Modeliranje posla nasljeđivanjem dretve 517
19.4. Sinkronizacija dretvi metodom `join()` 519
19.5. Čitanje i pisanje više dretvi po istoj varijabli 521
19.6. Čitanje i pisanje više dretvi po istoj varijabli uporabom `AtomicLong` 527
19.7. Čitanje i pisanje više dretvi po istoj varijabli u kritičnom odsječku 531
19.8. Čitanje i pisanje više dretvi po istoj varijabli u kritičnom odsječku (2) 532
19.9. Tko upravlja zaključavanjem? 534
19.10. Prvi model spremnika 535
19.11. Glavni program proizvođača i potrošača 536
19.12. Višedretveno-korektno napisan spremnik 539
20.1. Pomoćni razred za ispisivanje adresa 549
20.2. Ispis podataka o mrežnim karticama 550
20.3. Razrješavanje simboličkog imena u IP adresu 559
20.4. Pomoćni razred za pretvorbu `short` u/iz okteta 563
20.5. UDP-klijent za specificirani protokol 564
20.6. UDP-posluzitelj za specificirani protokol 566
20.7. Jednostavan program za dohvat resursa preko HTTP-a 580
20.8. Jednostavan HTTP-poslužitelj 582
21.1. Stvaranje prozora i prikaz na ekranu. 588
21.2. Stvaranje prozora: bolja organizacija koda 591
21.3. Dojava događaja vezanih uz prozor 592
21.4. Zatvaranje prozora na treći klik gumba za zatvaranje 595
21.5. Swing-komponenta i border 602
21.6. Crtanje površine Swing-komponente 606
21.7. Crtanje površine Swing-komponente (2) 607
21.8. Crtanje površine Swing-komponente (3) 609
21.9. Crtanje površine Swing-komponente (4) 610
21.10. Modifikacija grafičke komponente `Poruka` 611
21.11. Upravljač razmještajem `StretchLayout` 617
21.12. Demonstracijski program za `StretchLayout` 619
21.13. Upravljač razmještajem `HorizontalSplitLayout` 621

21.14. Demonstracijski program za `HorizontalSplitLayout` 624
21.15. Komponenta `Kvacica` 638
21.16. Komponenta `KvacicaUI` 640
21.17. Komponenta `Tema1KvacicaUI` 640
21.18. Komponenta `Tema2KvacicaUI` 642
21.19. Demonstracijski program za komponentu `Kvacica` i LAF 645
22.1. Demonstracijski program 652
22.2. Uporaba zajedničkog promatrača 653
22.3. Uporaba zasebnih promatrača 655
22.4. Izgradnja izborničke strukture 656
22.5. Dopuna izborničkom strukturom 658
22.6. Rješenje uporabom akcija 662
22.7. Jednostavan preglednik slika 666
23.1. Implementacija jednostavnog nepromijenjivog modela. 676
23.2. Prikaz liste u prozoru. 677
23.3. Implementacija modela liste koja čuva podatke u memoriji. 679
23.4. Demonstracijski program uz predstavljeni model liste. 680
23.5. Implementacija modela liste koja čuva podatke u memoriji (2). 682
23.6. Demonstracijski program uz predstavljeni model liste (2). 684
23.7. Demonstracijski program uz prikaz elementa liste. 689
23.8. Demonstracijski program uz prikaz elementa liste (nadogradnja). 691
23.9. Jedna implementacija modela `SpinnerModel` 695
23.10. Demonstracijski program za `JSpinner` 696
23.11. Demonstracijski program za `JSpinner` (2) 698
23.12. Demonstracijski program za `JSpinner` (3) 700
23.13. Model tablice s različitim informacijama o cijelim brojevima. 705
23.14. Demonstracijski program uz predstavljeni model tablice. 707
23.15. Demonstracijski program i izvedeni model. 709
23.16. Demonstracijski program i izvedeni model. 712

Predgovor

Bit će jednom napisan.

Poglavlje 1. Java – jezik, platforma, ekosustav

Programski jezik Java nastao je iz programskog jezika Oak. Počev od 1995. godine kada je u u internet preglednik Netscape Navigator ugrađena podrška za Javu pa sve do danas, jezik se razvija i širi te je danas praktički sveprisutan.

Jedna od temeljnih vodilja u razvoju ovog jezika, koja je dovela do ovakve opće prihvaćenosti, jest ideja *napiši-jednom-pokreni-bilo-gdje*. Centralna tema u razvoju Jave je potpora za višeplatformnost, odnosno ideja da se programeru ponudi apstraktni pogled na računalo -- pogled koji ne uključuje platformski specifične detalje poput širine podatkovne riječi centralnog procesora, direktan pristup upravljačkim programima operacijskog sustava ili podržanom sklopovlju. Umjesto toga, Java definira apstraktni računski stroj kod kojeg je sve propisano i opisano specifikacijom. Za programera je u konačnici skroz nebitno hoće li se njegov program izvoditi na 16-bitnom, 32-bitnom ili 64-bitnom procesoru.

Da bi to bilo moguće, prilikom programiranja u Javi programi se pišu za Javin virtualni stroj. To je stroj koji je definiran specifikacijom [7] i na kojem se izvode svi programi pisani u Javi. Zahvaljujući ovakvoj virtualizaciji, programer doista ne treba (i ne može) razmišljati o specifičnostima platforme na kojoj će se izvoditi program koji je napisao, a to upravo i jest ideja izrade aplikacija za više platformi odjednom. Dakako, na svakoj konkretnoj platformi morat će postojati implementacija Javinog virtualnog stroja koja će u konačnici biti svjesna na kojoj se platformi izvodi te kako se na toj platformi obavljaju pojedini zadatci (poput pristupa datotečnom sustavu, mrežnim resursima, višedretvenosti i slično).

Pa što je sve danas Java? Termin Java danas se koristi u nekoliko različitih značenja.

- Java kao programski jezik
- Java kao platforma
- Javin virtualni stroj (JVM, engl. *Java virtual machine*)
- JRE, JDK (SDK)
- Java ekosustav

U najužem značenju, pojam Java odnosi se na programski jezik. Najnovija verzija programskog jezika Java definirana je specifikacijom [6]. U okviru te specifikacije definira se način pisanja literala, ključne riječ jezika, njihova semantika te način pisanja valjanog koda. Pod pojmom "valjanog" ovdje podrazumjevamo sintaksno i semantički ispravnog koda sa stajališta jezičnog prevodioca koji izvorni Java kod treba obraditi i stvoriti izvršni kod za Javin virtualni stroj.

Naučiti programski jezik Javu (u najužem mogućem smislu riječi) može se vrlo brzo: pravila pisanja koda su vrlo jednostavna, ključnih riječi nema puno (par desetaka) i to se sve može naučiti ekspresno. Međutim, jezik sam po sebi je zapravo poprilično beskoristan. Da bismo mogli djelotvorno pisati programe, potreban nam je skup biblioteka čijom ćemo uporabom moći obavljati željene funkcije (pristupiti datoteci, stvoriti prozor u grafičkom korisničkom sučelju, raditi s različitim vrstama podatkovnih kolekcija i slično). Osnovni skup biblioteka koje programeru stoje na raspolaganju dio su Java platforme i opisane su u okviru dokumenta Java API [2]. Neovisno o platformi na kojoj se Java program izvodi, programeru se garantira da će mu na raspolaganju biti sve bibloteke koje su opisane u okviru tog dokumenta.

Javin virtualni stroj još je jedan od načina na koji se može interpretirati pojam Java. To je virtualni stroj definiran specifikacijom [7] koji stoga ima definiran skup podržanih instrukcija (takozvani bajtkod, engl. *bytecode*), način dobivanja programa te način na koji se kod izvodi. Umjes-

to kroz programski jezik Java, programeri programe mogu pisati koristeći direktno bajtkod koji je namijenjen upravo Javinom virtualnom stroju, pri čemu se i dalje zadržava portabilnost tako napisanih programa. To je, u određenom smislu, jednako kao da pišemo program za neki "virtualni procesor". U okviru ove knjige nećemo se baviti time.

Programere koji se po prvi puta susreću s programskim jezikom Java često znaju zbuniti dva pojma: JRE te JDK (ili u ranijim verzijama SDK). Evo o čemu se radi. Programeri programe pišu zadavanjem izvornog koda u programskom jeziku Java (tekstovne datoteke s ekstenzijom `.java`. Izvorni se kod potom prevodi u izvršni kod odnosno bajtkod (datoteke s ekstenzijom `.class`). Da bi se izvršni program mogao pokrenuti, potreban je Javin virtualni stroj te skup biblioteka čije se postojanje garantira svim Java programima. Ovaj postupak prikazan je na slici 1.1. S obzirom na opisani proces, sada je jasna podjela platforme na dvije komponente. JRE predstavlja osnovni podskup Java platforme koji korisnicima nudi mogućnost pokretanja prevedenih programa. JRE se sastoji od implementacije Javinog virtualnog stroja te obaveznih biblioteka čije se postojanje garantira programima. Ovo je minimum koji će svima omogućiti da pokreću izvršne Java programe. Uz navedeno, JRE sadrži i implementaciju dodatka za web-preglednike koji i njima omogućava izvođenje Java programa koje korisnici preuzimaju direktno s Interneta. Pojam JDK predstavlja nadskup opisane platforme. JDK sadrži sve što je potrebno kako bi programer mogao prevoditi izvorni kod Java programa u bajtkod te kako bi mogao izvoditi Java programe. To znači da JDK u sebi uključuje JRE te donosi još i implementaciju prevodioca i drugih pomoćnih alata. JDK pri tome ne sadrži okolinu za razvoj Java programa -- pretpostavka je da programeru treba ponuditi samo još mogućnost prevođenja koda. Na stranicama proizvođača Jave (od nedavno, to je tvrtka Oracle) moguće je pronaći i paket koji sadrži okolinu za razvoj programa pod nazivom NetBeans. Osim navedene, danas je dostupan još niz drugih okolina, što besplatnih a što komercijalnih, od čega ćemo svakako preporučiti okolinu otvorenog koda pod nazivom Eclipse (http://www.eclipse.org/).

Konačno, ovaj pregled ne bi bio potpun kada ne bismo spomenuli da sve do sada opisano predstavlja samo jedan mali podskup Java platforme koji je poznat pod nazivom *Java standard edition*, odnosno *Java SE*. Puno šira specifikacija poznata pod nazivom *Java Enterprise Edition* [3] i donosi niz tehnologija koje pokrivaju izradu web aplikacija, rad s bazama podataka i raspodijeljeno transakcijsko poslovanje, komuniciranje porukama i još niz drugih primjena. Postoji i specifikacija koja pokriva izradu Java aplikacija za mobilne uređaje [4] koja međutim iz različitih razloga u ovom trenutku nije baš najbolje prihvaćena, što je pak dovelo do razvoja Googleove platforme Android [5] koja je postala daleko raširenija (za tu platformu programi se također pišu u programskom jeziku Javi; razlika je u skupu podržanih biblioteka koje je u ovom slučaju definirao i održava Google).



Važna mrežna mjesta

Mrežno mjesto s kojega je moguće dohvatiti JRE odnosno JDK dostupno je na adresi http://www.oracle.com/technetwork/java/javase/downloads/index.html. Kako biste na računalo dodali mogućnost izvođenja gotovih Java programa potrebno je skinuti i instalirati samo JRE. Ako želite razvijati Java programe, tada je potrebno skinuti i instalirati JDK; u tom slučaju nije potrebno posebno skidati još i JRE jer je on uključen. U trenutku pisanja ovog teksta posljednja verzija JDK-a je bila Java SE 7u15.

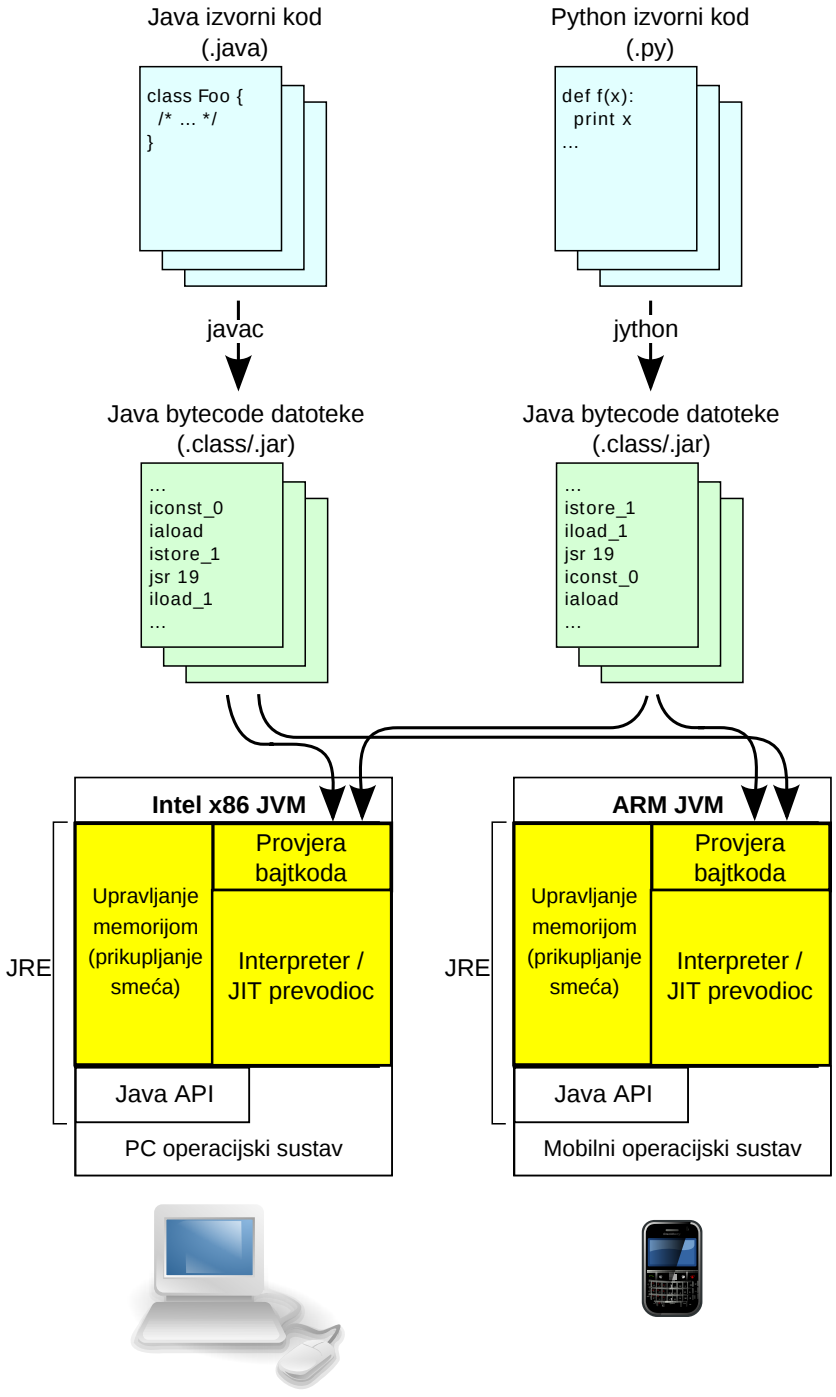
Spomenimo na kraju pregleda još i Javin ekosustav. Osim standardnih biblioteka koje su dostupne u okviru Java platforme, oko Java platforme stvoren je bogat i živ ekosustav koji je iznjedrio čitav niz drugih biblioteka, razvojnih okvira pa čak i novih programskih jezika koji su danas u uporabi. Primjerice, ako pogledamo izradu aplikacija za web -- uz standardne tehnologije danas na raspolaganju imamo i niz razvojnih okvira poput Apache Struts, Apache Tapestry, Grails, Google Web Toolkit (GWT), Spring i druge. Imamo na raspolaganju i prave relacijske baze podataka koje su u potpunosti pisane u Javi i koje se mogu koristiti čak i kao komponente lokalnu unutar vlastitih programa bez eksplicitne instalacije baze podataka -- primjeri su baza Apache Derby koja je u posljednjim verzijama Jave uključena čak i u SDK (pod nazivom JavaDB), baza H2 te druge. A s obzirom da je Javin virtualni stroj posebni dio Javine platforme, čak i za njega postoji razvijen niz implementacija drugih programskih jezika koje programerima

omogućavaju da pišu kod u jezicima poput Pythona, Haskella, JavaScripta i sličnima te da te programe izvode na Javinom virtualnom stroju čime se i tim jezicima nudi portabilnost u punom smislu te riječi.

Postupak prevođenja i izvođenja Java programa

Pogledajmo sada što se sve događa od izvornog koda pa do Java programa u izvođenju. Slika 1.1 prikazuje najvažnije korake.

Slika 1.1. Od izvornog koda do programa u izvođenju



Prvi korak ovog postupka je pisanje izvornog koda u programskom jeziku Java. Dva su načina kako može nastati izvorni kod: može ga napisati programer ili izvorni kod može biti strojno generiran.

- Slučaj u kojem programer piše izvorni kod najčešći je oblik nastanka izvornog koda. Izvorni kod programa može se pisati u jednostavnim uređivačima teksta (primjerice, koristeći programe **Notepad**, **Notepad++**, **UltraEdit** i slične na Windowsima, **pico**, **joe**, **vim**, **emacs** na Linuxu, višeplatformne uređivače teksta poput alata **jEdit** (koji je i sam pisan u Javi) te sličnim alatima na drugim operacijskim sustavima. U slučaju izrade iole kompliciranijih programskih rješenjapreporučljivo je koristiti okruženja za razvoj programa (engl. IDE - *Integrated Development Environment*) koja će uz omogućavanje pisanja jedne ili više datoteka s izvornim kodom automatski ponuditi i mogućnost upravljanja projektima, automatsko pokretanje postupka prevođenja, automatsku nadopunu prilikom pisanja izvornog koda te pokretanje i testiranje napisanog programa. Popis ovakvih alata je relativno velik, a najpoznatija su okruženja Eclipse te NetNeans.
- Slučaj u kojem se izvorni kod generira automatski rjeđe se susreće prilikom izrade jednostavnih programa; međutim, prisutan je u situacijama u kojima se za razvoj složenih aplikacija koriste gotovi razvojni okviri. Takvi razvoji okviri specijalizirani su za razvoj određenih vrsta aplikacija pa omogućavaju programeru da definira i podesi grube obrise aplikacije, na temelju čega razvojni okvir automatski generira potreban izvorni kod koji je naknadno moguće po potrebi još i korigirati.

Jednom kada raspolažemo izvornim kodovima Java programa (namjerno koristimo množinu -- vidjet ćemo kasnije da je loša praksa pisati ogromne monolitne datoteke i da ćemo težiti modularnim rješenjima), sljedeći korak predstavlja prevođenje izvornog Java koda u bajtkod. U tu svrhu u okviru instalacije JDK na raspolaganju nam stoji program **javac**, čije je ime konkatenacija riječi *java* te prvog slova engleskog izvornika za jezični procesor: *compiler*. Program **javac** kao ulaz uzima izvorne kodove (te vanjske biblioteke ako ih izvorni kod koristi). Datoteke s izvornim kodovima programskog jezika Java moraju imati ekstenziju `.java`. Program potom generira datoteke s izvršnim kodom za Javin virtualni stroj, odnosno datoteke koje sadrže bajtkod. Datoteke koje sadrže bajtkod imat će ekstenziju `.class`.

Uočimo odmah da izvorni kod ne treba nužno biti pisan jezikom Java, kao što prethodna slika i pokazuje. Primjerice, ako smo program pisali koristeći programski jezik *Python*, izvorne datoteke imat će ekstenziju `.py`. U tom slučaju na računalu bismo morali imati instaliran *Jython*: prevodioc koji izvorni kod pisan u jeziku *Python* prevodi u Javin bajtkod te opet generira datoteke s ekstenzijom `.class`. Kako danas za Javin virtualni stroj postoji niz prevodioca, izbor jezika u kojima možemo pisati relativno je velik.

Jednom kada smo generirali bajtkod, postupak prevođenja (barem što se tiče nas kao programera) time završava. Bajtkod je izvršni kod za Javin virtualni stroj. Bajtkod je upravo ono što čini našu aplikaciju i bajtkod se predaje drugim korisnicima i instalira na druga računala (kada korisnici instaliraju naš program).

Pokretanja Java programa obavlja se uporabom programa **java**. Taj program zapravo predstavlja implementaciju virtualnog stroja i sastavni je dio svakog JRE-a. Program **java** kao ulaz uzima upravo bajtkod i potom ga izvodi. Ovom programu pri tome nije važno kako je taj bajtkod nastao -- je li nastao iz *Java* izvornog koda, iz *Python* izvornog koda ili nekog trećeg programskog jezika. Jednom kada imamo generirani bajtkod, imamo sve što je potrebno da bismo program pokrenuli. Problem različitih platforma pri tome nas kao programere više ne zanima -- specifičnosti ostvarivanja svih usluga koje Javin virtualni stroj nudi programima problem su samog virtualnog stroja i riješene su na toj razini. To je upravo i razlog zbog kojeg, da bismo dobili mogućnost pokretanja Java aplikacija, moramo otići na web-stranicu proizvođača JRE-a te skinuti i instalirati Javin virtualni stroj za operacijski sustav koji koristimo. Međutim, kako je kompletna Java platforma definirana kroz niz specifikacija, danas imamo mogućnost izbora proizvođača Javinog virtualnog stroja -- to može biti *Oracle* koji je kupio tvrtku *Sun* koja je idejni tvorac *Jave* ali i ne mora biti -- danas postoje i druge implementacija Javinih virtualnih strojeva (primjerice, od tvrtke *IBM* ili od organizacije *Apache*) kao i alternativne izvedbe prevodioca za izvorne kodove programa.

Javin virtualni stroj

Iz dosadašnje priče sada već možemo naslutiti koja je uloga Javinog virtualnog stroja: to je apstraktni stogovni stroj čija je namjena izvođenja programa pisanih u bajtkodu. Fundamentalni razlog razvoja Javinog virtualnog stroja je osiguravanje platformske neovisnosti programa koji su pisani u bajtkodu. Da bismo takav program mogli pokrenuti na bilo kojoj platformi, nužno je i dovoljno raspolagati implementacijom Javinog virtualnog stroja za tu platformu. U današnje doba, ove implementacije postoje za sve moderne platforme; počev od klasičnih platformi (Intelovi procesori, AMD-ovi procesori; operacijski sustavi Windows, Linux, OS X) pa do mobilnih platformi.

Sam virtualni stroj aplikacijama nudi apstrakciju dretvi i potporu za njihovo izvođenje (što uključuje Java stogove) odnosno potporu za pozivanje nativnih biblioteka (što uključuje nativne stogove i tehnologiju JNI). Virtualni stroj također raspolaže mehanizmom za dohvat bajtkoda koji treba izvoditi te apstrakcijom izvršnih jedinica (virtualnih procesora odnosno Java dretvi). Način na koji će se izvršne jedinice implementirati nije propisan nikakvom specifikacijom -- tvorci Javinih virtualnih strojeva ovdje imaju potpunu slobodu. Ono što specifikacije propisuju jest format datoteke `.class` te popis i semantiku podržanih instrukcija. Stoga su prve i najjednostavnije izvedbe Javinih virtualnih strojeva bili upravo interpreterski strojevi: programski sustavi koji su čitali instrukciju po instrukciju bajtkoda te interpretirali što ta instrukcija treba napraviti i potom to napravili. Ovakve implementacije Javinih virtualnih strojeva čitavu su platformu dovele na loš glas -- *Java je bila spora*. Danas kada je tehnologija izrade Java virtualnih strojeva značajno napredovala, virutalni strojevi i sami se ponašaju kao jezični procesori: jezični procesori koji bajtkod tumače kao izvorni kod i potom provode postupak generiranja strojnog koda za procesor na kojem se izvodi i sam virtualni stroj i pri tome provode još i dodatne optimizacije. Vrhunac ove tehnologije danas predstavljaju jezični procesori koji u letu prevode bajtkod u strojni kod uporabom JIT prevodioca (engl. *Just-In-Time Compilers*). Ideja je pri tome izuzetno jednostavna: prevođenje bajtkoda u strojni kod je vremenski skupa operacija; stoga Javin virtualni stroj dijelove koda koji se rijetko izvode interpretira, što uzevši u obzir činjenicu da se ti dijelovi rijetko izvode ne povlači velike vremenske penale a dijelove koda za koje se utvrdi da se često izvode u letu prevodi u strojni kod. Pri tome se čak i sam postupak prevođenja u strojni kod radi u više koraka i s različitim razinama optimizacije; naime, što se uključi više optimizacija, to je postupak generiranja konačnog strojnog koda sporiji. Stoga virtualni strojevi neprestano prate intenzitet izvođenja pojedinih dijelova programa i potom adaptivno najčešće izvođene dijelove koda malo po malo prevode u strojni kod uz sve više i više uključenih optimizacija.

Osim učinkovitog izvođenja bajtkoda i pružanja izolacije od stvarne platforme na kojoj se program izvodi, Javin virtualni stroj ima još jednu važnu zadaću: automatsko upravljanje memorijom programa. Naime, programski jezik Java ne poznaje metodu za oslobađanje zauzete memorije. Analizirajući pogreške koje programeri rade u programskim jezicima niže razine poput programskog jezika C, Pascal i sličnih ali i kod programskih jezika više razine poput jezika C++ ustanovljeno je da velik niz pogrešaka koje programeri rade spadaju u pogreške vezane uz loše upravljanje memorijom. S druge pak strane, također je ustanovljeno da u programima koji su korektni nezanemariv dio koda koji je napisan služi isključivo za upravljanje memorijom -- zauzimanje memorije, provjeravanje treba li neki dio memorije osloboditi te oslobađanje memorije. Kako bi jednim udarcem riješila sve navedene probleme, Java zadaću upravljanja memorijom prebacuje na Javin virtualni stroj i programera u potpunosti oslobađa brige o oslobađanju memorije. Misao vodilja je pri tome vrlo jasna: umjesto da svaki programer za sebe u svakom svojem programu piše dijelove koda koji se brinu za oslobađanje memorije, napišimo to jednom direktno u virtualnom stroju i onemogućimo programeru da uopće brine o tom zadatku.

Podsustav unutar Javinog virtualnog stroja koji se bavi ovim zadatkom zove se *garbage collector*. Specifikacija Javinog virtualnog stroja ne propisuje kako ovaj podsustav mora biti implementiran -- propisuje samo funkciju koju on treba ostvarivati. Dva su tipična načina izvođenja ovakvog podsustava: brojanjem referenci ili pak periodičkim zamrzavanjem programa i praćenjem živih referenci. Međutim, svaka od ovih izvedbi, pa dapače i različite izvedbe unutar iste kategorije sustava nude različito dinamičko ponašanje izvođenja programa. S obzirom da u višedretvenom okruženju postoji čitav niz problema koje treba uzeti u obzir i nekako riješiti te

s obzirom da se ta rješenja ponašaju različito na sustavima koji ukupno imaju malo memorije u odnosu na sustave koji ukupno imaju puno memorije odnosno na sustavima koji imaju jednu jezgru ili pak više jezgri, danas je sasvim uobičajena situacija da Javin virtualni stroj dolazi s više implementiranih strategija i da omogućava korisniku da prilikom pokretanja virtualnog stroja odabere i dodatno podesi strategiju koja će se koristiti. Bez dodatnih podešavanja virtualni stroj se pokreće sa strategijom za koju je utvrđeno da u prosjeku radi dobro na relativno velikom broju različitih programa.

Jezici podržani od JVM-a

U posljednjih desetak godina broj jezika koji su podržani od strane JVM-a značajno je narastao. Relativno iscrpan popis može se pogledati na http://en.wikipedia.org/wiki/List_of_JVM_languages. Razloga za implementaciju ovakvih jezika je mnoštvo, od kojih su možda dva najvažnija portabilnost te mogućnost uporabe svih mehanizama koje Javin virtualni stroj nudi, počev od ugrađenog i vrlo kvalitetnog podsustava za automatsko upravljanje memorijom pa do odlične potpore za višedretvenost. U ovom kontekstu i portabilnost je izuzetno značajna: osim što omogućava da se program napisan u nekom od jezika koji su podržani na Javinom virtualnom stroju izvode na bilo kojoj platformi na kojoj postoji Javin virtualni stroj, nudi se i više -- takvi jezici često mogu direktno komunicirati s dijelovima koda koji su napisani u drugim jezicima što otvara vrlo zanimljive mogućnosti za izradu aplikacija.

Kako je ovakvih jezika danas mnogo, u nastavku ćemo nabrojati samo neke od najinteresantnijih.

Popis drugih zanimljivijih jezika za Javin virtualni stroj

Rhino Implementacija jezika *JavaScript*. Izvorno implementirana od tvrtke Mozilla, biblioteka koja omogućava izvođenje programa pisanih u jeziku *JavaScript* na Javinom virtualnom stroju danas je sastavni dio Javinog izvršnog okruženja. Stoga za njegovu uporabu nije potrebno nikakva dodatna instalacija.

U trenutku pisanja ovog teksta aktualna verzija jezika bila je 1.7R4. Stranica projekta *Rhino* je https://developer.mozilla.org/en-US/docs/Rhino.

Scala Ovo je implementacija funkcijskog jezika koji se izvodi na Javinom virtualnom stroju. U posljednjih nekoliko godina ovaj je jezik postao poprilično rasprostranjen što je vidljivo i iz niza napisanih knjiga koje ga opisuju.

U trenutku pisanja ovog teksta aktualna verzija jezika bila je 2.9.2. Stranica jezika *Scala* je http://www.scala-lang.org/.

Clojure Jezik *Clojure* još je jedan funkcijski programski jezik za Javin virtualni stroj. Ovaj jezik jedan je od dijalekata jezika *Lisp* koji sa sobom donosi bogat skup nepromjenjivih struktura podataka, a za potrebe izmjenjivih podataka nudi mehanizme poput transakcijske memorije te reaktivnih agenata koji omogućavaju jednostavnu izradu višedretvenih programa.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 1.4. Stranica jezika *Clojure* je http://clojure.org/.

JRuby Jezik *JRuby* predstavlja implementaciju jezika *Ruby* za Javin virtualni stroj. Razvoj jezika *Ruby* započeo je oko 1995. godine; sam jezik nastao je kao spoj jezika Perl, Smalltalk, Eiffel, Ada te Lisp.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 1.7.1. Stranica jezika *JRuby* je http://jruby.org/ dok se više o originalnom jeziku *Ruby* može pronaći na http://www.ruby-lang.org/en/.

Jython Jezik *Jython* predstavlja implementaciju jezika *Python* za Javin virtualni stroj. Jezik *Python* kroz posljednjih je nekoliko godina postao vrlo proširen i popularan pro-

gramski jezik. Jedan od razloga tome leži u bogatstvu stilova programiranja i jezičnih konstrukata koje ovaj jezik nudi programerima na uporabu. Što se tiče performansi programa pisanih u ovom jeziku, one često nisu baš na zavidnoj razini, no u mnoštvu situacija kada to nije od presudne važnosti ovaj se jezik pokazuje kao odličan izbor.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 2.5.3. Stranica jezika *Jython* je http://www.jython.org/. Zainteresirani čitatelji upućuju se i na knjigu o ovom jeziku koja je javno dostupna direktno na Internetu [8.]

Groovy — *Groovy* predstavlja implementaciju dinamičkog jezika za Javin virtualni stroj. Ovaj jezik također je u posljednjih nekoliko godina postao dosta popularan i ima niz primjena te popriličan broj knjiga koje su o njemu napisane.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 2.0. Stranica jezika *Groovy* je http://groovy.codehaus.org/.

Jaskell — *Jaskell* predstavlja implementaciju jednog od vrlo popularnih funkcijskih jezika: jezika *Haskell*. Jezik nudi statičko izvođenje tipova i niz drugih funkcijskih konstrukata.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 1.0. Stranica jezika *Jaskell* je http://jaskell.codehaus.org/.

Yeti — Iskreno govoreći, jezik *Yeti* ne spada među baš popularnije jezike za Javin virtualni stroj; ovdje ga spominjemo naprosto stoga što predstavlja implementaciju jednog od vrlo zanimljivih funkcijskih jezika: jezika *ML*. Jezik nudi statičko zaključivanje o tipovima Hindley-Milnerovim algoritmom, curryjev oblik funkcija, podudaranje uzoraka, monadičke kombinatore, funkcije više reda i slično.

U trenutku pisanja ovog teksta aktualna stabilna verzija jezika bila je 0.9.7. Stranica jezika *Yeti* je http://mth.github.com/yeti/.

Podržani tipovi podataka

Java je objektno-orijentirani programski jezik. Pri prvom susretu s Javom stoga mnogi pretpostave da u Javi postoji tip podataka koji predstavlja objekte. I nakon toga krenu bespotrebne rasprave o tome kako se u Javi obavlja prijenos parametara pri pozivu metoda: po vrijednosti ili po referenci (ako u ovom trenutku nešto od ove rečenice nije jasno -- to je OK). Da bismo odmah u početku razriješili ove probleme, idemo odmah jasno definirati čime to Java raspolaže.

Prema službenoj specifikaciji jezika Java SE 7 [6], programski jezik Java prepoznaje dvije vrste podataka: *primitivne tipove* te *reference*; i to je to -- ništa manje i ništa više. Primitivni tipovi podataka su numeričke vrijednosti, vrijednosti istinitosti te znakovi. Postoji osam primitivnih tipova podataka i oni su opisani u tablici Tablica 1.1. Primitivne vrijednosti možemo podijeliti u numeričke te tip za prikaz istinitosti. Numeričke tipove možemo podijeliti u cjelobrojne (byte, short, int, long, char) i decimalne (float i double). Za prikaz istinitosti definiran je samo jedan tip: boolean.

Tablica 1.1. Primitivni tipovi u programskom jeziku Java

Tip	Opis
byte	Predstavlja cijeli broj čiji je raspon ograničen na interval od -128 do +127. Konceptualno, radi se o 8-bitnom cijelom broju s predznakom koji koristi dvojni komplement.
short	Predstavlja cijeli broj čiji je raspon ograničen na interval od -32768 do +32767. Konceptualno, radi se o 16-bitnom cijelom broju s predznakom koji koristi dvojni komplement.

Tip	Opis
int	Predstavlja cijeli broj čiji je raspon ograničen na interval od -2,147,483,648 do +2,147,483,647. Konceptualno, radi se o 32-bitnom cijelom broju s predznakom koji koristi dvojni komplement.
long	Predstavlja cijeli broj čiji je raspon ograničen na interval od -9,223,372,036,854,775,808 do +9,223,372,036,854,775,807. Konceptualno, radi se o 64-bitnom cijelom broju s predznakom koji koristi dvojni komplement.
float	Predstavlja decimalni broj skromne preciznosti. Konceptualno, odgovara specifikaciji IEEE 754 za decimalne brojeva jednostruke preciznosti (troši se 32 bita odnosno 4 okteta za svaku vrijednost).
double	Predstavlja decimalni broj skromne preciznosti. Konceptualno, odgovara specifikaciji IEEE 754 za decimalne brojeva dvostruke preciznosti (troši se 64 bita odnosno 8 okteta za svaku vrijednost).
boolean	Predstavlja tip podataka koji se koristi za prikaz istinitosti. Podatci ovog tipa mogu poprimiti jednu od dviju vrijednosti: `true` te `false`.
char	Predstavlja tip podataka koji se koristi za prikaz jednog znaka. Konceptualno, to je cijeli 16-bitni broj bez predznaka, tako da su legalne vrijednosti cijeli brojevi iz raspona od 0 do 65535, ili alternativno od '\u0000' do '\uffff'. Ovo je značajna razlika u odnosu na programski jezik C kod kojeg je jedan znak koristio za pohranu jedan oktet. Ova razlika uvedena je kako bi se jeziku omogućilo da transparentno radi s velikim brojem različitih znakova koristeći pri tome Unicode specifikaciju [1].

Nad cjelobrojnim tipovima podataka moguće je obavljati niz operacija. Operatori < (manje), <= (manje ili jednako), > (veće), >= (veće ili jednako), == (jednako) te != (različito) omogućavaju usporedbu dviju vrijednosti. Postoje unarni operatori + i − te binarni operatori + (zbrajanje), − (oduzimanje), % (ostatak cjelobrojnog dijeljenja), * (množenje) i / (dijeljenje) koji omogućavaju provođenje aritmetičkih operacija. Na raspolaganju su prefiks i postfiks inačice operatora za inkrement vrijednosti (++) te operatora za dekrement vrijednosti (−−). Postoje tri operatora posmaka: << koji predstavlja posmak u lijevo, >> koji predstavlja aritmetički posmak u desno (čuva se predznak) te >>> koji predstavlja logički posmak u desno (upražnjeno mjesto uvijek se puni s vrijednošću 0). Konačno, na raspolaganju su još i unarni operator bitovnog komplementa (~) koji u binarnom zapisu vrijednosti komplementira svaki bit zasebno te binarni bitovni operatori za provođenje operacija I (&), ILI (|) te isključivo ILI (^). Konačno, ova vrsta podataka može se naći kao operand ternarnog operatora odabira (? :).

Prilikom provođenja operacija ako je barem jedan od argumenata tipa long, tada se operacija provodi (i rezultira s) tipom long; u suprotnom, operacija se provodi u domeni tipa int i rezultat je tog tipa (neovisno o tome jesu li operandi također bili tog tipa ili nekog "jednostavnijeg", poput byte).

Skup operatora koji su podržani nad decimalnim brojevima nešto je manji u odnosu na operacije podržane nad cijelobrojnim tipovima jer pojedine operacije u ovoj domeni nemaju smisla; tako primjerice ne postoje operatori posmaka, binarnog komplementa te bitovne operacije. Dodatno, decimalni brojevi mogu poprimiti i neke specifične vrijednosti kojima se signalizira da je vrijednost jednaka pozitivnoj ili negativnoj beskonačnosti, ili pak da nije valjani broj (npr. rezultat dijeljenja `0/0`).

Tip boolean podržava najmanji skup operatora koji mogu djelovati nad njime. Tu su operatori za ispitivanje jednakosti dviju vrijednosti (== te !=), unarni operator komplementa vrijednosti (!), binarni logički operatori &, | te ^), uvjetni logički operatori (&& i ||) te ternarni operator odabira (? :). Uvjetni logički operatori argumente računaju samo ako je to potrebno. Primjerice, ako imamo izraz *izraz1* || *izraz2*, izraz *izraz2* će se računati samo ako izračun izraza *izraz1* rezultira s vrijednošću `false`. Ako izračun izraza *izraz1* rezultira s vrijednošću `true`, tada će i vrijednost logičke operacije ILI rezultirati s vrijednošću `true` i desna strana operatora

se uopće neće računati. To je pak zgodno jer omogućava pisanje koda poput `car==null || car.emptyTank()` koji će se razrješiti u vrijednost `true` bilo u slučaju da je varijabla `car` inicijalizirana s `null` referencom, bilo da pokazuje na neki primjerak automobila za koji metoda `emptyTank` vraća vrijednost `true`. Uporaba klasičnog logičkog operatora `|` u ovom kontekstu rezultirala bi izazivanjem pogreške u slučaju da je prvi uvijet ispunjen jer se tada nad takvom varijablom metoda `emptyTank` ne bi smjela pozvati pa bismo trebali pisati kompliciraniji kod koji bi proveo ovo ispitivanje.

Primjeri literala

Za iscrpan i potpun opis svih načina na koji se mogu pisati literali pojedinih tipova čitatelj se upućuje na [6]. U nastavku ćemo prikazati samo nekoliko primjera koji će biti dovoljni za daljnje praćenje ove knjige.

- Tip byte: `(byte)12`, `(byte)-74`. Nužna je uporaba operatora ukalupljivanja (engl. *cast operator*).
- Tip short: `(short)12`, `(short)-74`. Nužna je uporaba operatora ukalupljivanja (engl. *cast operator*).
- Tip int: `12`, `-74`, `0x2f3` (zadano heksadekadski), `0b01001110` (zadano binarnim zapisom), `02731` (zadano oktalno). Literal se tumači kao heksadekadski ako započinje s `0x` a kao binarni ako započinje s `0b`; ako započinje s `0` nakon čega slijede brojevi, tumači se kao oktalni. Kako bi se pospješila čitljivost, između znamenaka može se umetnuti jedna ili više podvlaka: `123_451`.
- Tip long: `12L`, `-74L`. Osim navedenog, mogu se zadavati i binarni, oktalni te heksadekadski oblici pri čemu na kraju moraju imati slovo `L`; primjerice: `0x7eab3281552fL`.
- Tip float: `12.0f`, `-74.0f`, `3f`, `2.f`, `21.3e+2f`, `21.3e-3f`. Nužna je uporaba slova `f` na kraju.
- Tip double: `12.0`, `-74.0`, `3d`, `2.`, `21.3e+2`, `21.3e-3`. Sufiks `d` nije obavezan; nužno ga je pisati samo kada bi bez njega literal bio protumačen drugačije; primjerice `3` je literal tipa int dok je `3d` literal tipa double.
- Tip char: `'A'`, `'\u0041'`, `'\t'`, `'\\'`. Uporaba markera `\u` omogućava unos 4 heksadekadske znamenke kojima je jednoznačno određen jedan znak. Prilikom tumačenja uneseng broja Java koristi UTF-16 specifikaciju. Heksadekadska sekvenca `0041` predstavlja dekadsku vrijednost `65` odnosno veliko slovo `A`. Znak `\` uvijek uvodi posebno interpretiranje onoga što slijedi; ako je sljedeće slovo `u`, očekuje se heksadekadska sekvenca koja u skladu sa specifikacijom UTF-16 određuje o kojem se znaku radi. Međutim, druga slova omogućavaju unos drugih često korištenih znakova; primjerice, slijed `\t` tumači se kao znak `tab`, slijed `\n` tumači se kao znak za prelazak u novi red (tipično separator redaka u tekstovnim datotekama) a slijed `\'` se tumači kao znak `'`. Ako baš želimo unijeti znak `\`, tada moramo posegnuti za slijedom `\\` koji će se protumačiti upravo kao jedan znak `\`.
- Za tip boolean moguće su samo dvije vrijednosti i obje su ključne riječi: `true` odnosno `false`.
- Za sve reference postoji samo jedan literal: `null`.

Stringovi

Primjetite da do sada ništa nismo govorili o stringovima. Programski jezik Java ne poznaje stringove kao primitivne tipove podataka. Svaki string pohranjuje se kao primjerak razreda java.lang.String (tj. razreda String smještenog u paket java.lang. Međutim, kako bi se programerima ponudio što je moguće jednostavniji i intuitivniji rad sa stringovima, od Java prevodioca se očekuje da stringove u izvornom kodu prepoznaje i po potrebi automatski generira kod

koji će stvoriti potrebne objekte. String literali se, kao i u većini drugih programskih jezika pišu pod dvostrukim navodnicima. Tako su primjeri stringova: `""` (prazan string tj. string duljine 0), `"Hello world!"` (string koji sadrži 12 znakova), `"First line.\nSecond line."` (string koji sadrži dva retka; `\n` je sekvenca koja predstavlja prijelom retka i tumači se kao jedan ascii znak čija je vrijednost 10). Jezični prevodioc dopustit će nam i spajanje više stringova operatorom `+`, pa tako možemo pisati:

```
String s = "Ovo " + "je " + "tekst";
```

što će rezultirati jednim stringom sadržaja:

```
"Ovo je tekst."
```

Operator plus, ako se koristi u kontekstu stringova, može poslužiti i za pretvorbu drugih tipova podataka u stringove. Primjerice,

```
String s = "Vidim " + 15 + "zečeva";
```

rezultirati će jednim stringom sadržaja:

```
"Vidim 15 zečeva."
```

Tijekom prevođenja, `15` je literal tipa int; međutim, konkatenacijom sa stringom on se prevodi u string reprezentaciju. Više o ovome će biti govora nešto kasnije. Konkatenacija ne zahtjeva da se navode samo literali. I sljedeći isječak koda će raditi prema očekivanjima.

```
int n = 15;
String s = "Vidim " + n + "zečeva";
```

Rezultat će biti jednak rezultatu prethodnog primjera.

S obzirom da su stringovi objekti, nad svakim stringom programerima na raspolaganju stoji niz metoda koje je moguće pozvati kako bi se dobile neke informacije o stringu ili kako bi se provele određene operacije. Evo jednog primjera.

```
String poruka = "Došao sam u grad Zagreb.";
int duljina = poruka.length();❶
String velika = poruka.toUpperCase();❷
String grad = poruka.substring(17,23);❸
```

❶ Vrijednost varijable `duljina` postat će 24 -- metoda `.length()` vratit će duljinu teksta nad kojim je pozvana, izraženo u znakovima, što je u ovom slučaju 24.

❷ Varijable `velika` predstavljat će novi string u kojem su sva slova velika. Metoda `.toUpperCase()` stvara novi string koji je sadržajno jednak stringu nad kojim je pozvana uz razliku da će sva mala slova u vraćenom stringu biti zamijenjena velikim slovima.

❸ Varijable `grad` predstavljat će novi string `Zagreb`. Metoda `.substring()` pozvana uz dva argumenta vraća novi string koji predstavlja podniz originalnog stringa; prvi argument određuje poziciju znaka s kojim započinje taj podniz a drugi argument određuje poziciju prvog znaka koji više nije dio vraćenog podniza.

Prilikom rada sa stringovima važno je zapamtiti da su u Javi stringovi nepromjenjivi (engl. *immutable*). Jednom stvoreni string nemoguće je promijeniti, i metode koje naoko čine neke izmjene (poput metode `.toUpperCase()` ili pak metode `.replace(...)`) zapravo vraćaju novi string -- originalni string se pri tome ne mijenja. Zahvaljujući takvom radu, neke se operacije u Javi mogu izvesti izuzetno efikasno, poput operacije `.substring(...)` koja u memoriji neće raditi novu kopiju sadržaja već će doslovno samo zapamtiti poziciju početka i kraja u originalnom stringu.

Kako bi se podržala nepromjenjivost stringova, Java programerima ne omogućava direktan pristup memorijskom spremniku u kojem se čuva sadržaj stringa. Međutim, nudi se metoda kojom se mogu dohvaćati (ali ne i mijenjati) pojedini znakovi. Evo primjera.

```
String poruka = "Moje ime je Ivana.";
char prviZnak = poruka.charAt(0);❶
char sedmiZnak = poruka.charAt(6);❷
```

❶ Varijabla `prviZnak` dobit će vrijednost `'M'`
❷ Varijabla `sedmiZnak` dobit će vrijednost `'m'`

Argument metode `charAt(...)` je indeks znaka koji želimo dohvatiti. Znakovi su pri tome indeksirani od vrijednosti 0.

Jedan od načina stvaranja novih stringova je i direktno predavanjem polja znakova koje predstavlja sadržaj stringa. Evo primjera.

```
char sadrzaj[] = {'Z', 'a', 'g', 'r', 'e', 'b'};
String grad = new String(sadrzaj);❶
sadrzaj[1] = 'B';❷
```

❶ Stvara se novi string čiji je sadržaj `"Zagreb"`. Međutim, string koji nastaje kao interni spremnik teksta ne koristi predano polje već radi svoju vlastitu kopiju.
❷ Naknadna promjena elemenata polja `sadrzaj` stoga ne mijenja prethodno stvoreni string -- njegov sadržaj je i dalje `"Zagreb"`.

Inicijalizacija vrijednosti

Specifikacija programskog jezika Java garantira da će sve varijable, osim lokalnih kao i svi elementi polja automatski biti inicijalizirani na početne vrijednosti. Za sve tipove podataka koje Java podržava vrijednosti su navedene u nastavku.

- Za tip byte početna vrijednost je nula, odnosno `(byte)0`.
- Za tip short početna vrijednost je nula, odnosno `(short)0`.
- Za tip int početna vrijednost je nula, odnosno `0`.
- Za tip long početna vrijednost je nula, odnosno `0L`.
- Za tip float početna vrijednost je nula, odnosno `0.0f`.
- Za tip double početna vrijednost je nula, odnosno `0.0d`.
- Za tip char početna vrijednost je nula, odnosno `'\u0000'`.
- Za tip boolean početna vrijednost je nula, odnosno `false`.
- Za sve reference početna vrijednost je `null`.

Početne vrijednosti automatski se ne dodjeljuju *lokalnim varijablama* (lokalne varijable su varijable koje su deklarirane unutar funkcija). Svim lokalnim varijablama vrijednost treba dodijeliti eksplicitno prije prvog čitanja njihove vrijednosti.

Pretvorbe primitivnih tipova u string

Za pretvorbu osnovnih tipova podataka u string, u Javi nam na raspolaganju stoji već niz gotovih metoda. U okviru razreda `String` postoji porodica metoda `.valueOf(...)` koje mogu primaju vrijednosti primitivnih tipova a generiraju tekstovne prikaze istih. Evo primjera.

```
char c = 'A';
double d = 1.1;
float f = 4.25f;
int n = 125;
String sc = String.valueOf(c);  // Rezultirat će s "A"
```

```
String sd = String.valueOf(d);  // Rezultirat će s "1.1"
String sf = String.valueOf(f);  // Rezultirat će s "4.25"
String sn = String.valueOf(n);  // Rezultirat će s "125"
```

Omotači primitivnih tipova

U programskom jeziku Java, za sve primitivne tipove definirani su i razredi-omotači (engl. *wrappers*). Tako postoje razredi `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` te `Boolean`. Svaki razred omotač korisnicima nudi funkcionalnost pretvorbe primitivne vrijednosti u string, pretvorbe stringa u primitivnu vrijednost te još niz drugih korisnih metoda. Evo nekoliko primjera koji koriste razred `Integer`.

```
int broj = 125;
String sBroj = "517";

int vrijednost = Integer.parseInt(sBroj);       // 517
String tekst = Integer.toString(broj);          // "125"
String tekst8 = Integer.toString(broj, 8);      // "175" (u bazi 8)
String tekstBin = Integer.toBinaryString(broj); // "1111101"
String tekstHex = Integer.toHexString(broj);    // "7D"
```

U prethodnom primjeru koristili smo metode razreda `Integer` kako bismo obavljali konverzije između stringa i primitivne vrijednosti. Omotači su, međutim, prikladni i za reprezentaciju primitivnih vrijednosti. Opet ćemo dati primjer kroz razred `Integer`.

```
int broj1 = 50;
int broj2 = 200;

Integer b1 = Integer.valueOf(broj1);
Integer b2 = Integer.valueOf(broj2);

boolean manji = b1.compareTo(b2) < 0;
int suma = b1.intValue() + b2.intValue();
int razlika = b1 - b2;

String b2KaoString = b2.toString();  // "200"
```

Za razliku od primitivnih tipova, primjerci razreda omotača su upravo to -- primjerci razreda, odnosno objekti. U prethodnom primjeru će `b1` predstavljati objekt koji enkapsulira cijeli broj `50` dok će `b2` predstavljati objekt koji enkapsulira cijeli broj `200`. S obzirom da su to objekti, nad njima ćemo moći pozivati podržane metode, poput metode `compareTo(...)` koja će usporediti vrijednost broja koji enkapsulira omotač nad kojim smo pozvali metodu s vrijednošću broja koji enkapsulira omotač koji predajemo kao argument, i koja će vratiti vrijednost `-1` ako je prva vrijednost manja od druge.

Omotači vrijednosti, baš kao i stringovi, u Javi predstavljaju nepromjenjive objekte. Stoga nude samo metodu kojom je moguće dohvatiti trenutnu primitivnu vrijednost (ali je nije moguće promijeniti). Primjer je metoda `.intValue()` kod razreda `Integer` koja vraća primitivnu vrijednost kao tip int. Java prevodioc će za programera automatski pozivati te metode gdje je potrebno kako bi kod učinio kompaktnijim. U prethodnom primjeru to je vidljivo u retku u kojem računamo

```
int razlika = b1 - b2;
```

što će se efektivno prevesti u kod

```
int razlika = b1.intValue() - b2.intValue();
```

bez ikakve intervencije od programera.

Stvaranje primjeraka omotača uobičajeno je moguće obaviti na dva načina: direktnim pozivom metode koja u memoriji zauzima mjesto za novi primjerak razreda (tzv. konstruktor razreda), ili pak pozivom metode `.valueOf(...)` koju nude svi omotači. Primjer je prikazan u nastavku.

```
Integer i1 = new Integer(17);
Integer i2 = new Integer(17);
Integer i3 = Integer.valueOf(17);
Integer i4 = Integer.valueOf(17);
```

Prva dva retka pokazuju stvaranje dva primjerka omotača oko primitivne vrijednosti 17. Svaki poziv zauzet će novu količinu memorije za objekt koji, zapravo, predstavlja istu vrijednost. Poziv metode `.valueOf(...)` omogućava programskom jeziku Java da, gdje je to prikladno, privremeno interno pohrani (kešira) već stvorene objekte te umjesto stvaranja novih objekata vraća objekte iz priručnog spremnika (keša). Primjerice, uobičajene implementacije interno čuvaju primjerke razreda `Integer` za primitivne vrijednosti od `-128` do `127` pa će opetovani pozivi metode za istu vrijednost iz tog raspona uvijek vratiti isti, unaprijed alocirani objekt. Uočimo da je to moguće jer omotači predstavljaju nepromijenjive objekte pa je sigurno vraćati isti objekt više puta bez bojazni da će se pohranjena vrijednost negdje promijeniti. U prethodnom primjeru stoga će `i3` i `i4` predstavljati identičan objekt u memoriji.

Uvjetno izvođenje, petlje i decimalni brojevi

U programskom jeziku Java za uvjetno izvođenje koda na raspolaganju nam stoji `if-else if-...-else if-else` konstrukt. Evo nekoliko primjera uporabe. Prvi primjer ilustrira kod koji ispituje je li sadržaj varijable `x` neparan i ako je, uvećava sadržaj varijable `broj` za jedan.

```
int broj = 0;
int x = 5;
if(x % 2 != 0) {
  broj++;
}
```

Java je strogo tipizirani programski jezik. Posljedica je da se na mjestu gdje se očekuje vrijednost tipa `boolean` ne može pojaviti ništa drugo. Primjerice, u programskom jeziku C bilo bi legalno napisati:

```
if(x % 2) {
  broj++;
}
```

što u Javi, međutim, neće proći. Evo složenijeg primjera koji ilustrira program koji će, ovisno o ostatku cjelobrojnog dijeljenja broja `x` s četiri uvećati sadržat varijable `broj` za različite iznose.

```
int broj = 0;
int x = 5;
if(x % 4 == 0) {
  broj++;
} else if(x % 4 == 1) {
  broj += 17;
} else if(x % 4 == 2) {
  broj += 6;
} else {
  broj += 14;
}
```

Umjesto višestrukog ispitivanja naredbom `if`, isto smo mogli postići konciznije uporabom naredbe `switch` kako je prikazano u nastavku.

```
int broj = 0;
int x = 5;
```

```
switch(x % 2) {
  case 0:
    broj++;
    break;
  case 1:
    broj += 17;
    break;
  case 2:
    broj += 6;
    break;
  default:
    broj += 14;
    break;
}
```

Primjetimo usput da opisano ispitivanje radi korektno za `x>=0`. U slučaju da je sadržaj varijable `x` negativan, ostatci cjelobrojnog dijeljenja će također biti negativni što treba uzeti u obzir ako je takva situacija moguća.

Osim nad numeričkim i logičkim tipovima podataka, od verzije 7 programskog jezika Java naredba `switch` podržava i ispitivanje stringova, pa je sljedeći primjer korektan.

```
int broj = 0;
String grad = "Zagreb";
switch(grad) {
  case "Bjelovar":
    broj++;
    break;
  case "Split":
    broj += 2;
    break;
  case "Zagreb":
    broj += 3;
    break;
  default:
    broj += 4;
    break;
}
```

Od naredbi za ponavljanje, u Javi možemo koristiti naredbu `for`, `while` te `do-while`. Preskakanje izvođenja ostatka trenutne iteracije postiže se naredbom `continue` a kompletno prekidanje iteriranja naredbom `break`. Pogledajmo naprije primjer uporabe petlje `for`.

```
int suma = 0;
int[] polje = {1, 4, 7, 11, 13, 21};
for(int i = 0; i < polje.length; i++) {
  if(polje[i] == 4) {
    continue;
  }
  if(polje[i] > 10) {
    break;
  }
  suma += polje[i];
}
```

Uz pretpostavku da je zadano polje sortirano, napisani program računa sumu elemenata polja koji nisu veći od 10 i koji su različiti od 4. Tako naredba `continue` preskače ažuriranje sume ako je trenutni element jednak 4 a naredba `break` prekida zbrajanje čim dođe prvi element koji je veći od 10. Za navedeni primjer, sadržaj varijable `suma` na kraju će biti jednak 8.

Prethodni primjer mogli smo napisati i na sljedeći način.

```
int suma = 0;
int[] polje = {1, 4, 7, 11, 13, 21};
for(int i = 0, granica = polje.length; i < granica; i++) {
  if(polje[i] == 4) {
    continue;
  }
  if(polje[i] > 10) {
    break;
  }
  suma += polje[i];
}
```

Umjesto da se na kraju svake iteracije nanovo dohvaća vrijednost `polje.length`, tu smo vrijednost dohvatili jednom na ulasku u petlju, pohranili smo je u pomoćnu varijablu koju smo nazvali `granica` i potom na kraju svake iteracije dohvaćamo tu vrijednost. Kod pristupanja veličini polja, ovo neće napraviti veliku razliku u brzini izvođenja koda, no da smo u uvjetu za prekid imali poziv neke funkcije koja bi vraćala primjerice veličinu kolekcije po kojoj prolazimo, razlika između jednog poziva i višestrukih poziva mogla bi značajnije utjecati na performanse programa.

Počevši od Jave verzije 5, u jezik je uvedena potpora za još jedan oblik petlje `for` koji je u drugim jezicima poznat pod nazivom `foreach`. Imamo li, primjerice, polje (ili neku kolekciju, što ćemo vidjeti kasnije), sve elemente možemo obići kako je prikazano u sljedećem primjeru.

```
int[] polje = {1, 4, 7, 11, 13, 21};
for(int element : polje) {
  System.out.println("Trenutni element je: " + element);
}
```

Ovaj obilazak ekvivalentan je obilasku prikazanom u nastavku.

```
int[] polje = {1, 4, 7, 11, 13, 21};
for(int i = 0, granica = polje.length; i < granica;  i++) {
  int element = polje[i];
  System.out.println("Trenutni element je: " + element);
}
```

Prednost kraćeg oblika jest što je koncizniji -- ne trebamo sami stvarati pomoćnu varijablu koja će se koristiti kao indeks i ne trebamo sami dohvaćati vrijednost pojedinog elementa i pohranjivati je u novu pomoćnu varijablu. Umjesto toga, možemo napisati kraći oblik i pustiti Java-prevodilac da za nas izgenerira ekvivalentni kod. Pri tome u petlji `for` samo deklariramo tip varijable koji odgovara tipu elemenata iz polja (ili kolekcije), definiramo naziv lokalne varijable koja će poprimati sve vrijednosti redom te nakon dvotočke dajemo polje (ili kolekciju) po kojem treba obići. Ovu jezičnu konstrukciju često čitamo ovako: *za int* `element` *u* `polje`.

Naredbe `continue` i `break` mogu imati i dodatni argument (labelu) koji određuje petlju na koju se naredba odnosi. Evo primjera. Pretpostavimo da imamo nesortirano polje brojeva te da trebamo utvrditi postoji li u tom polju duplikata. Oslanjajući se na osnovnu uporabu naredbe `break`, taj bismo kod mogli napisati na sljedeći način.

```
long[] polje = {1, 4, 7, 11, 13, 4, 21};
boolean imaDuplikata = false;
for(int i = 0; i < polje.length; i++) {
  long element1 = polje[i];
  for(int j = i+1; j < polje.length; j++) {
    long element2 = polje[j];
    if(i == j) {
      continue;
```

```
    }
    if(element1 == element2) {
      imaDuplikata = true;
      break;
    }
  }
  if(imaDuplikata) {
    break;
  }
}
System.out.println("Pronađeni su duplikati: "+ imaDuplikata);
```

Ne razmišljajući sada o efikasnosti izvođenja prethodnog koda, uočimo da imamo dvije ugniježđene `for`-petlje. Vanjska petlja ide element po element polja, te unutarnja koja za svaki fiksirani element ponovno prolazi kroz polje i provjerava ima li još na kojoj lokaciji upravo takav element.U slučaju da je takav element pronađen, idealno bi bilo nakon ažuriranja zastavice `imaDuplikata` prekinuti obje petlje: i unutarnju i vanjsku. Međutim, koristeći osnovni oblik naredbe `break` to nije moguće, pa se nakon unutarnje petlje provjerava je li postavljena zastavica `imaDuplikata` i ako je, prekida se i vanjska petlja novom naredbom `break`.

Uporabom labela, ovo se može riješiti jednostavnije. Vanjskoj petlji dat ćemo naziv (labelu) "Vanjska". Potom u unutarnjoj petlji naredbi `break` možemo predati tu labelu čime će naredba prekinuti i unutarnju i vanjsku petlju. Evo koda.

```
         long[] polje = {1, 4, 7, 11, 13, 4, 21};
         boolean imaDuplikata = false;
Vanjska: for(int i = 0; i < polje.length; i++) {
           long element1 = polje[i];
           for(int j = i+1; j < polje.length; j++) {
             long element2 = polje[j];
             if(i == j) {
               continue;
             }
             if(element1 == element2) {
               imaDuplikata = true;
               break Vanjska;
             }
           }
         }
         System.out.println(
             "Pronađeni su duplikati: "+ imaDuplikata
         );
```

Pogledajmo sada i primjer uporabe petlje `while`.

```
int suma = 0;
int[] polje = {1, 4, 7, 11, 13, 21};
int i = 0;
while(i < polje.length) {
  if(polje[i] > 10) {
    break;
  }
  suma += polje[i];
  i++;
}
```

Slično možemo postići i `do-while` konstruktom.

```
int suma = 0;
```

```
int[] polje = {1, 4, 7, 11, 13, 21};
int i = 0;
do {
  if(polje[i] > 10) {
    break;
  }
  suma += polje[i];
  i++;
} while(i < polje.length);
```

Treba međutim pripaziti na osnovnu razliku između ove dvije naredbe. Naredba `do-while` signurno barem jednom ulazi u tijelo petlje pa bi prethodni primjer mogao neslavno završiti ako polje nema barem jedan element. Stoga se posljednji oblik koristi kada je na drugi način već osigurano da će prvi prolaz sigurno završiti korektno.

Konačno, u okviru ovog podpoglavlja osvrnut ćemo se i na pisanje koda koji koristi operator == za utvrđivanje ekvivalencije. Važno je zapamtiti da taj operator uvijek uspoređuje vrijednost *primitivnih* tipova podataka. Sjetimo se, u javi su primitivni tipovi podataka numerički, logički te reference. Četvrto ne postoji. Posljedica toga je da sljedeći kod neće raditi onako kako bi početnici to očekivali.

```
String prvi = new String("Zagreb");
String drugi = new String("Zagreb");
if(prvi == drugi) {
  System.out.println("Jednaki su!");
} else {
  System.out.println("Nisu jednaki!");
}
```

Rezultat će biti ispis

```
Nisu jednaki!
```

. Naime, iako bi na prvu mnogi rekli da su varijable `prvi` i `drugi` stringovi, one to nisu: te su varijable po tipu reference na dva objekta koja predstavljaju stringove. Operator `new` je operator koji u memoriji zauzima mjesto za novi objekt i potom ga inicijalizira. S obzirom da u prva dva retka programa postoje dva poziva operatora `new`, jasno je da će rezultat biti dva objekta u memoriji na različitim memorijskim lokacijama. Operator == uspoređuje primitivne vrijednosti -- u ovom slučaju uspoređuje dvije reference. Rezultat će biti `true` ako i samo ako te dvije reference pokazuju na isti objekt u memoriji. Kako to ovdje nije slučaj, rezultat usporedbe će biti `false` i u naredbi `if` izvršit će se kod zadan u `else` dijelu.

Druga važna napomena odnosi se na usporedbu decimalnih brojeva. Za usporedbu decimalnih brojeva nikada nije dobro koristiti operator ==. Naime, aritmetika s decimalnim brojeva u Javi je aritmetika s ograničenom preciznošću. Pogledajmo naprije primjer 1.1

Primjer 1.1. Ilustracija računanja s ograničenom preciznošću te usporedba rezultata

```
 1 public static void main(String[] args) {
 2   double a = 1000;
 3   double b = 7;
 4   double c = Math.PI;
 5
 6   double rez1 = prviNacin(a, b, c);
 7   double rez2 = drugiNacin(a, b, c);
 8
 9   if(rez1==rez2) {
10    System.out.println("Isti su.");
```

```
11   } else {
12    System.out.println("Nisu isti. Razlika je: " + (rez1-rez2));
13   }
14  }
15
16  public static double prviNacin(double a, double b, double c) {
17   return (a + b + c) / 7;
18  }
19
20  public static double drugiNacin(double a, double b, double c) {
21   return a/7 + b/7 + c/7;
22  }
```

Ovaj kod predstavlja isječak programa koji na dva načina računa isti izraz. Metoda `prviNacin` računa izraz:

$$\frac{a+b+c}{7}$$

dok metoda `drugiNacin` računa izraz:

$$\frac{a}{7}+\frac{b}{7}+\frac{c}{7}$$

Uvažavajući matematička znanja s kojima vladamo, znamo da bi ta dva izraza trebala predstavljati isto, odnosno znamo da bi trebalo vrijediti:

$$\frac{a+b+c}{7}=\frac{a}{7}+\frac{b}{7}+\frac{c}{7}.$$

što je pak motivacija da za ispitivanje jednakosti koristimo operator == kako je to prikazano u retku 9 isječka koda prikazanog na ispisu 1.1. Nažalost, u stvarnosti, ovaj bi program ispisao:

```
Nisu isti. Razlika je: -2.8421709430404007E-14.
```

Razlog nastanka ove razlike je činjenica da su tipovi podataka `double` i `float` tipovi za prikaz vrijednost decimalnih brojeva do na unaprijed određenu preciznost, što znači da se dio informacije gubi. Tada više nije nebitno hoćemo li izraz izračunati tako da najprije izračunamo sumu sva tri broja i potom je podijelimo sa sedam, ili ćemo najprije računati sedmine pojedinih pribrojnika i potom ih pribrajati. Da je tome tako, upravo nam demonstrira prethodni primjer.

Upravo zbog ograničene preciznosti kojom se pohranjuju decimalni brojevi važno je za usporedbu decimalnih brojeva ne koristiti operator == već se dogovoriti o dozvoljenom odstupanju uz koje ćemo brojeve ipak smatrati jednakima i potom provjeriti kakav je apsolutni iznos njihove razlike s obzirom na dogovoreno dopušteno odstupanje. Ispravan način usporedbe "jednakosti" decimalnih brojeva prikazuje primjer 1.2.

Primjer 1.2. Usporedba jednakosti decimalnih brojeva

```
1  public static void main(String[] args) {
2   double a = 1000;
3   double b = 7;
4   double c = Math.PI;
5
6   double rez1 = prviNacin(a, b, c);
7   double rez2 = drugiNacin(a, b, c);
8
9   if(Math.abs(rez1-rez2) < 1E-8) {
10   System.out.println("Isti su.");
11  } else {
12   System.out.println("Nisu isti. Razlika je: " + (rez1-rez2));
```

```
13   }
14 }
15
16 public static double prviNacin(double a, double b, double c) {
17   return (a + b + c) / 7;
18 }
19
20 public static double drugiNacin(double a, double b, double c) {
21   return a/7 + b/7 + c/7;
22 }
```

Poglavlje 2. Prvi Java program

Nakon što smo se upoznali s osnovnim pojmovima o platformi Java, pogledajmo koji su koraci potrebni kako bismo došli do prvog izvršnog programa napisanog u Javi. Fokus ovog poglavlja su osnove -- radit ćemo bez razvojnih okruženja i "magije" koju oni donose sa sobom. Ideja je razumjeti što se događa u pozadini svih tih procesa kako bismo kasnije mogli djelotvornije koristiti razvojna okruženja.



Preduvjeti

Da biste mogli isprobati primjere navedene u ovom poglavlju, na računalu trebate imati instaliran JDK te neki uređivač teksta. Za kratku uputu o instalaciji JDK-a pogledajte dodatak A. Radite li na operacijskom sustavu Windows, preporuka je da koristite uređivač poput programa Notepad++ koji će Vam omogućiti da izvorne programe spremite uporabom kodne stranice UTF-8 bez *BOM* markera. Na modernijim inačicama operacijskog sustava Linux možete koristiti bilo koji uređivač teksta jer oni danas uobičajeno koriste UTF-8 za pohranu tekstova.

U izradu Java programa krenut ćemo s najjednostavnijim mogućim primjerom: programom koji će na zaslon ispisati poruku `Hello, world! Učimo Javu!`. Za potrebe ovog poglavlja pretpostavit ćemo da će svi programi biti smješteni na datotečnom sustavu unutar vršnog poddirektorija `D:\javaproj`. Ako se odlučite za neku drugu lokaciju, obratite pažnju da na svim mjestima u nastavku tu stazu zamijenite stazom koju ste sami odabrali. Napravite navedeni direktorij i unutar njega napravite direktorij `projekt1`. Potom se pozicionirajte u navedeni direktorij. Naredbe kojima ćete to napraviti su sljedeće.

```
D:\>mkdir javaproj
D:\>cd javaproj
D:\javaproj>mkdir projekt1
D:\javaproj>cd projekt1
D:\javaproj\projekt1>
```

Direktorij `D:\javaproj\projekt1` poslužit će nam kao vršni direktorij za izradu ovog jednostavnog programa. Unutar tog direktorija stvorit ćemo još dva poddirektorija: `src` unutar kojeg ćemo smjestiti izvorni kod programa (engl. *source files*) te direktorij `bin` unutar kojeg ćemo generirati izvršni kod programa. Potrebne naredbe prikazane su u nastavku.

```
D:\javaproj\projekt1>mkdir src
D:\javaproj\projekt1>mkdir bin
```

U ovom trenutku cjelokupna struktura direktorija koju smo pripremili trebala bi izgledati kako je prikazano u nastavku.

```
D:\
  +-- javaproj
        +-- projekt1
              +-- src
              +-- bin
```

Program s kojim ćemo krenuti smjestit ćemo u datoteku `HelloWorld.java` čiji je sadržaj prikazan u primjeru 2.1. Datoteku je potrebno napraviti u direktoriju `src` pa u njegovom poddirektoriju `hr\fer\zemris\java\tecaj_1`. Stvaranje cjelokupne podstrukture direktorija moguće je obaviti naredbom **mkdir** odjednom.

U uređivaču teksta koji ćete koristiti stvorite praznu datoteku `hr\fer\zemris\java\tecaj_1\HelloWorld.java` i prepišite sadržaj prikazan u primjeru 2.1. Datoteku obavezno pohranite koristeći UTF-8 kodnu stranicu bez BOM-a.

Primjer 2.1. Primjer programa napisanog u programskom jeziku Java

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 /**
 4  * Program čija je zadaća na ekran ispisati poruku
 5  * "Hello, world!".
 6  *
 7  * @author Marko Čupić
 8  * @version 1.0
 9  */
10 public class HelloWorld {
11
12  /**
13   * Metoda koja se poziva prilikom pokretanja
14   * programa. Argumenti su objašnjeni u nastavku.
15   *
16   * @param args argumenti komandne linije. U ovom
17   *             primjeru se ne koriste.
18   */
19  public static void main(String[] args) {
20   System.out.println("Hello, world! Učimo Javu!");
21  }
22
23 }
24
```

Pogledajmo malo detaljnije primjer 2.1. Java je objektno orijentirani programski jezik; stoga su osnovni dijelovi svakog programa objekti. Primjer 2.1 ilustrira definiranje razreda `HelloWorld`. Definiranje novog razreda obavlja se uporabom ključne riječi class, što je vidljivo u retku 10 gdje deklaracija započinje navođenjem modifikatora public, ključne riječi class, nazivom razreda koji se deklarira `HelloWorld` i otvorene vitičaste zagrade. Deklaracija razreda proteže se sve do odgovarajuće zatvorene vitičaste zagrade u retku 23.



Upozorenje

Naziv datoteke mora odgovarati nazivu razreda koji je definiran u datoteci. Pri tome se mora paziti i na velika i mala slova, neovisno o tome je li datotečni sustav osjetljiv na velika i mala ili nije.

Da bi se programerima olakšalo odabiranje smislenih imena razreda (a time i pisanje kvalitetnijeg koda), Java omogućava smještanje razreda u hijerarhijski definirane pakete. Na taj način programeru se omogućava da za pisanje novog programa ili biblioteke odabere novi paket i potom ne treba brinuti je li netko već negdje u nekom drugom paketu napisao razred koji se zove jednako -- razredi koji nose isto ime ali su smješteni u različite pakete za programski jezik Java nisu isti razredi i definiran je jasan skup pravila koji osiguravaju da se program može jednoznačno pozvati i koristiti upravo one razrede koje je programer želio koristiti. U primjeru 2.1 odabrali smo razred `HelloWorld` smjestiti u paket naziva hr.fer.zemris.java.tecaj_1. Ova odluka definirana je u retku 1 gdje nakon ključne riječi package slijedi odabrani naziv paketa i nakon toga znak točka-zarez.

Konačno, u razredu `HelloWorld`, počevši od retka 19, definirali smo jednu javnu statičku metodu imena `main` čija je definicija prikazana u nastavku.

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

Koje je točno značenje pojmova *javnu statičku* bit će objašnjeno kasnije kada se malo bolje upoznamo s Javom i objektnim oblikovanjem. Ostanimo za sada na pojašnjenju da metoda mora biti javna i statička kako bi je virtualni stroj mogao pozvati kao početnu točku izvođenja

programa. Povratna vrijednost mora biti void te metoda mora deklarirati samo jedan argument -- polje stringova preko kojeg će dobiti pristup argumentima koje je korisnik unio u komandnu liniju prilikom pokretanja programa.

Primjer 2.1 ujedno ilustrira i uporabu komentara. Osim što se u Javi mogu koristiti uobičajeni komentari koji započinju s dvije kose crte i protežu se do kraja retka ili pak višelinijski komentari koji započinju s /* i koji se protežu do prve pojave */, Java nam dozvoljava i uporabu takozvanih *Javadoc* komentara. *Javadoc* komentari su komentari čija je namjena omogućiti automatsko generiranje dokumentacije iz izvornog koda. Ovu vrstu komentara prepoznat ćemo po tome što započinju s /** i tipično sadrže posebne oznake koje započinju znakom @. U prikazanom primjeru tako imamo dokumentiran razred `HelloWorld` uz navođenje njegove svrhe, autora i verzije te imamo dokumentiranu metodu `main` pri čemu je dat opis metode kao i opis njezinih parametara. Više o ovome bit će riječi u nastavku.

Prevođenje programa

Prevođenje izvornog programa u izvršni Java program obavlja se pozivom java prevodioca. Java prevodioc koji je dostupan u ukviru JDK-a poziva se naredbom **javac**. Zadat ćemo sljedeću naredbu.

```
d:\javaproj\projekt1>javac -sourcepath src -d bin -encoding UTF-8 \
                     -deprecation -g:lines,vars,source \
                     src\hr\fer\zemris\java\tecaj_1\*.java
```



Upozorenje

S obzirom da naredba zahtjeva nekoliko argumenata, čitav poziv ne stane u jedan redak te je za potrebe ispisa razlomljen u više redaka. Na kraju svakog retka koji je slomljen dodan je znak \ koji označava prijelom -- prilikom unosa naredbe u naredbeni redak taj znak treba preskočiti.

Parametar `-sourcepath` koristi se kako bi se zadao direktorij koji sadrži izvorne kodove i pripadnu strukturu poddirektorija koja odgovara razmještaju razreda u pakete. U našem slučaju to je direktorij `src`. Parametar `-d` određuje direktorij u koji će se smjestiti izvršni kodovi Java programa. Java prevodioc će prilikom prevođenja izvornih kodova programa u direktoriju za smještaj izvršnih kodova rekonstruirati strukturu direktorija koja odgovara paketima kako je to određeno u izvornom kodu.

Parametar `-encoding` određuje kodnu stranicu koju će Java prevodioc koristiti prilikom čitanja izvornih kodova programa. Ovo je važno zadati ako u izvornom kodu postoje dijakritički znakovi i slični specifični simboli. Tada je nužno znati koju je kodnu stranicu uređivač teksta koristio prilikom pohrane izvornog koda programa i tu kodnu stranicu treba ovdje navesti. Kako bi se riješio taj problem, najjednostavnije je podesiti uređivač teksta tako da izvorne kodove uvijek pohranjuje koristeći kodnu stranicu UTF-8 i to bez markera BOM.

Parametar `-deprecation` nalaže prevodiocu da prilikom prevođenja izvornog koda dojavi sve lokacije na kojima je pronašao uporabu razreda i metoda koje su proglašene napuštenima i koje stoga treba izbjegavati. Razredi i metode koji se u nekoj verziji Jave proglase napuštenima kandidati su za uklanjanje iz budućih verzija Jave i za njih obično postoji dokumentirana zamjena koju bi trebalo koristiti.

Parametar `-g` definira da se u izvršni kod pohrane i dodatne informacije koje će kasnije omogućiti lakše debugiranje programa. Tijekom razvoja i ispitivanja programa stoga je uobičajeno tražiti da se u izvršni kod ubace informacije poput originalnih naziva lokalnih varijabli, točnog retka u kojem se pojedine naredbe nalaze u izvornom kodu i slično.

Konačno, posljednji argument naredbe čine datoteke koje je potrebno prevesti. U našem primjeru traži se prevođenje svih datoteka koje su smještene u direktoriju `src\hr\fer\zemris\java\tecaj_1` i koje imaju ekstenziju `.java`.

Ako u izvornom programu postoji koja pogreška, to će biti prijavljeno na ekranu. Ako pak nema pogrešaka, u direktoriju `bin` dobit ćemo izgeneriranu strukturu direktorija koja odgovara paketu hr.fer.zemris.java.tecaj_1 i unutra će se nalaziti datototeka `HelloWorld.class`. Datoteke s ekstenzijom `.class` sadrže izvršni Java kod (odnosno *byte-code*). Temeljem datoteke s izvornim kodom `HelloWorld.java` nastat će datoteka s izvršnim kodom `HelloWorld.class`. Po uspješnom prevođenju programa sada ćemo na disku imati sljedeće direktorije i datoteke.

```
D:\
  +-- javaproj
        +-- projekt1
              +-- src
              |    +-- hr
              |         +-- fer
              |              +-- zemris
              |                   +-- java
              |                        +-- tecaj_1
              |                             +-- HelloWorld.java
              +-- bin
                   +-- hr
                        +-- fer
                             +-- zemris
                                  +-- java
                                       +-- tecaj_1
                                            +-- HelloWorld.class
```

Pokretanje izvršnog programa

Jednom kada smo dobili izvršni program, njegovo pokretanje obavlja se pozivom Javinog virtualnog stoja i predavanjem punog naziva razreda koji radrži metodu `main`. Pod pojmom *puni naziv razreda* pri tome se podrazumijeva zadavanje paketa u kojem se razred nalazi kao i lokalnog imena razreda u tom paketu. U našem slučaju, imamo razred čije je ime `HelloWorld` i koji se nalazi u paketu hr.fer.zemris.java.tecaj_1; njegovo puno ime tada je `hr.fer.zemris.java.tecaj_1.HelloWorld`, odnosno dobije se tako da se na naziv paketa nadoda ime razreda.

Javin virtualni stroj poziva se naredbom **java**. Pokretanje našeg programa obavit ćemo kako je prikazano u nastavku.

```
d:\javaproj\projekt1>java -cp bin \
                          hr.fer.zemris.java.tecaj_1.HelloWorld
```

Ako je sve u redu, pokretanjem virtualnog stroja naš će se program pokrenuti i na ekran će ispisati definiranu poruku. Da bi virtualni stroj znao gdje da traži datoteke s izvršnim programom, koristi se parametar *-cp* (što je pokrata od *classpath*) pomoću kojeg se zadaju lokacije unutar kojih virtualni stroj treba tražiti izvršne kodove razreda koje izvodi. U određenom smislu, to je sinonim varijabli okruženja `PATH` koja za ljusku navodi direktorije u kojima treba tražiti izvršne verzije programa koje korisnik pokušava pokrenuti navođenjem samo njihovog imena. Kako smo izvorni kod programa smjestili u direktorij `bin`, tu vrijednost smo specificirali kao vrijednost parametra.



Ljuska na Windowsima i problemi s kodnom stranicom

Korisnici koji ove primjere izvode kroz ljusku **Command Prompt** na operacijskom sustavu Windows mogu se naći u situaciji da je ispis dijakritičkih znakova (i možda nekih drugih simbola) neispravan. Do problema može doći ako Javin virtualni

stroj prilikom ispisa na konzolu koristi drugačiju kodnu stranicu u odnosu na kodnu stranicu koju koristi sama konzola. U tom slučaju prilikom ispisa dijakritičkih znakova Javin virtualni stroj generira i konzoli pošalje kod koji konzola protumači u skladu s drugom kodnom stranicom kao neki drugi znak. U tom slučaju pokrenite naredbu **chcp** koja će ispisati koju kodnu stranicu koristi konzola (primjerice, ispis bi mogao biti `Active code page: 852`). Jednom kada ste utvrdili koju kodnu stranicu koristi konzola, iskoristite parametar *`-Dfile.encoding`* kako biste biste Javinom virtualnom stroju zadali kodnu stranicu koju mora koristiti prilikom komuniciranja s okolinom. Primjer je dan u nastavku.

```
d:\javaproj\projekt1>java -cp bin \
                     -Dfile.encoding=IBM852 \
                     hr.fer.zemris.java.tecaj_1.HelloWorld
```

Složeniji primjer

Pogledajmo za vježbu jedan malo složeniji primjer. Napisat ćemo program koji će se sastojati iz dva razreda. Razred `Formule` sadržavat će kod koji će svojim klijentima omogućiti dobivanje usluge ispisa neke slučajno odabrane popularne formule iz fizike na ekran. Potom ćemo definirati i jednog klijenta tog razreda, razred `Glavni` koji će ujedno predstavljati i ulaznu točku u naš program. Nakon ispisa pozdravne poruke, glavni će program zatražiti ispis jedne slučajno generirane popularne formule iz fizike na zaslon, i potom će završiti s radom. Izvorni kod ovog programa dan je u primjeru 2.2.

Iako je ovo tek drugi program koji razmatramo, možda nije loše osvrnuti se na razlog zbog kojeg su ovdje napravljena čak dva razreda, iako smo sve mogli staviti u isti razred i dapače, sve je moglo biti u istoj metodi (primjerice, u metodi `main`). Jedna od karakteristika dobrog koda jest jasnoća koda i modularnost. Kod ne smije biti monolitan a njegovi sastavni dijelovi trebaju biti takvi da neki drugi programer koji čita taj kod može bez značajnog truda razumjeti što se događa u pojedinim dijelovima koda.

Jedna od posljedica prethodnog zahtjeva jest da se programere potiče da rješenja bilo kojeg problema strukturiraju na način da postoji dio koda koji je konceptualno na višoj razini i koji problem rješava dekompozicijom na podprobleme koji se pak delegiraju drugim dijelovima koda. Što to točno znači, ovisi o paradigmi koju koristimo (proceduralna paradigma, funkcijska paradigma, objektno-orijentirana paradigma i slično). Primjerice, u čistoj proceduralnoj paradigmi to bismo mogli tumačiti kao: neka svaka metoda problem rješava oslanjajući se na druge metode koje rješavaju jednostavnije podprobleme. Dapače, u nedostatku drugačijeg načina grupiranja, tu bismo mogli reći i sljedeće: cjelokupni izvorni kod razložit ćemo u više datoteka tako da svaka datoteka sadrži funkcije i metode koje čine jednu konceptualnu cjelinu. U objektno-orijentiranoj paradigmi na raspolaganju imamo i puno ekspresivnija sredstva: program možemo razložiti u pakete, svaki paket može sadržavati razrede i podpakete a svaki razred može sadržavati više metoda.

U ovom konkretnom primjeru, uočimo da jedan dio programa zahtjeva funkcionalnost koja bi mogla biti bitna i zanimljiva i drugim programima (definiranje poznatih formula fizike i njihov ispis na zaslon) što je već samo po sebi dosta dobar argument da se implementacija te funkcionalnosti izdvoji u zasebnu cjelinu koja nije direktno ovisna o načinu na koji ćemo je iskoristiti u ovom programu. Stoga je ta funkcionalnost izolirana u zaseban razred `Formule` u okviru kojeg je definirana metoda `ispisiSlucajnuFormulu` koja nudi pristup toj funkcionalnosti.

Primjer 2.2. Složeniji primjer -- ispis slučajno odabrane formule

Najprije je dan sadržaj datoteke `Formule.java` a potom datoteke `Glavni.java`.

```
1 package hr.fer.zemris.java.tecaj_1;
2
```

```
 3 /**
 4  * Razred sadrži funkcionalnost ispisa slučajno odabrane
 5  * poznate formule iz fizike na zaslon. Program trenutno
 6  * raspolaže s dvije formule iz područja relativističke
 7  * te Newtonove fizike; autori su, naravno, Einstein i
 8  * Newton.
 9  *
10  * @author Marko Čupić
11  * @version 1.0
12  */
13 public class Formule {
14
15  /**
16   * Metoda posredstvom slučajnog mehanizma prilikom svakog
17   * poziva temeljem uniformne distribucije odabire jednu od
18   * formula s kojima raspolaže i ispisuje je na zaslon.
19   * Ispisuje je i popratno zaglavlje.
20   */
21  public static void ispisiSlucajnuFormulu() {
22   boolean prva = Math.random() < 0.5;
23   System.out.println("Jedna od najpoznatijih formula fizike:");
24   System.out.println("======================================");
25   if(prva) {
26    System.out.println("  E = m * c^2");
27   } else {
28    System.out.println("  F = m * a");
29   }
30  }
31
32 }
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 /**
 4  * Program koji ilustrira razdvajanje koda u više razreda.
 5  *
 6  * @author Marko Čupić
 7  * @version 1.0
 8  */
 9 public class Glavni {
10
11  /**
12   * Metoda od koje kreće izvođenje programa.
13   * @param args argumenti komandne linije. Ne koristimo ih.
14   */
15  public static void main(String[] args) {
16   System.out.println();
17   System.out.println("Pozdrav! Učimo javu!");
18   System.out.println();
19   Formule.ispisiSlucajnuFormulu();
20  }
21
22 }
```

Drugi važan segment ovog programa jest osigurati da se na zaslon najprije ispiše pozdravna poruka te da se potom ispiše jedna slučajno odabrana formula fizike. Ovaj podproblem riješen je izradom razreda `Glavni` koji je opremljen metodom `main` čime može predstavljati ulaznu točku u program. U okviru te metode dalje je ponuđena implementacija koja ispisuje pozdrav-

nu poruku te potom koristeći uslugu razreda `Formule` dovršava posao ispisa jedne slučajno odabrane formule.

Napravite direktorij za novi projekt (nazovite direktorij `projekt2`). U tom novom direktoriju napravite poddirektorije `src` i `bin`. U direktoriju `src` napravite potrebnu strukturu poddirektorija koja će odgovarati paketu hr.fer.zemris.java.tecaj_1 i unutra stvorite datoteke `Formule.java` i `Glavni.java` u skladu s izvornim kodom prikazanim u primjeru 2.2. Time ćete dobiti strukturu direktorija koja je prikazana u nastavku.

```
D:\
  +-- javaproj
        +-- projekt2
              +-- src
              |    +-- hr
              |         +-- fer
              |              +-- zemris
              |                   +-- java
              |                        +-- tecaj_1
              |                             +-- Formule.java
              |                             +-- Glavni.java
              +-- bin
```

Uz pretpostavku da ste pozicionirani u direktoriju `D:\javaproj\projekt2`, program ćete prevesti i pokrenuti na sljedeći način.

```
d:\javaproj\projekt2>javac -sourcepath src -d bin -encoding UTF-8 \
                     -deprecation -g:lines,vars,source \
                     src\hr\fer\zemris\java\tecaj_1\*.java


d:\javaproj\projekt2>java -cp bin \
                          hr.fer.zemris.java.tecaj_1.Glavni

Pozdrav! Učimo javu!

Jedna od najpoznatijih formula fizike:
======================================
  F = m * a
```

Pakiranje izvršnog programa u arhivu

Prilikom prevođenja Java programa za svaku datoteku s izvornim kodom nastat će jedna datoteka s izvršnim kodom, osim ako se u toj datoteci ne nalazi više razreda ili pak ugniježđeni ili anonimni razredi -- tada će nastati i više od jedne datoteke s izvršnim kodom. Posljedica je da Java program nije, u onom klasičnom smislu na koji smo navikli, jedna datoteka. To pak ponekad može učiniti distribuciju aplikacija pisanih u Javi problematičnom. Kako bismo Java program pretvorili u jednu datoteku, na raspolaganju nam stoji alat **jar** (engl. *Java Archiver*) čijom uporabom možemo sve izvršne kodove aplikacije zapakirati zajedno u jednu arhivu.

Uz pretpostavku da ste i dalje pozicionirani u direktoriju `projekt2`, sljedećom naredbom stvorit ćete arhivu s izvršnim kodovima. Pri tome je važno da ste prije toga program doista preveli, odnosno da se njegovi izvršni kodovi nalaze u direktoriju `bin`.

```
d:\javaproj\projekt2>jar cf projekt2.jar -C bin\ .
```

Slova `cf` su prvi argument naredbe. Slovo `c` naredbi nalažu naredbi da stvori arhivu (c - *create*) a slovo `f` definira da će prvi sljedeći parametar biti naziv te arhive. I doista, drugi argument

naredbe je naziv arhive koja će biti stvorena (`projekt2.jar`). Ekstenzija koju koriste arhive s izvršnim Java kodom je `.jar`).

Parametrom *`-C`* definira se direktorij čiji će se sadržaj uključiti u arhivu. Konačno, posljednji argument definira što će se točno iz odabranog direktorija uključiti. Kako smo htjeli uključiti sve, zadan je direktorij `.` što je oznaka za trenutni direktorij.

Izvođenjem ove naredbe u direktoriju `projekt2` nastat će arhiva naziva `projekt2.jar`. Iako ima ekstenziju `.jar`, to je obična ZIP arhiva koju je moguće otvoriti bilo kojim programom koji zna raditi s formatom ZIP. Otvorite tu arhivu i uvjerite se da je njezin sadržaj sljedeći.

```
projekt2.jar
   +-- hr
   |    +-- fer
   |         +-- zemris
   |              +-- java
   |                   +-- tecaj_1
   |                        +-- Formule.class
   |                        +-- Glavni.class
   +-- META-INF
        +- MANIFEST.MF
```

Uz očekivanu strukturu direktorija koja odgovara korištenom paketu, u arhivi se nalazi i poseban direktorij `META-INF`. U tom direktoriju obavezno ćemo pronaći datoteku s informacijama o JAR arhivi -- to je datoteka naziva `MANIFEST.MF`. Primjer automatski genrirane datoteke prikazan je u nastavku.

```
Manifest-Version: 1.0
Created-By: 1.7.0_03 (Oracle Corporation)
```

Ova datoteka može se iskoristiti za unos niza dodatnih informacija o pakiranom sadržaju i s nekim dijelovima ćemo se upoznati u nastavku.

Jednom kada smo dobili JAR datoteku, dovoljno je njezin sadržaj distribuirati na računalo na kojem se želi pokrenuti Java program. Naime, u toj datoteci se nalaze zapakirane sve izvršne datoteke koje čine naš program. Ovo sada možemo provjeriti tako da, umjesto da prilikom pokretanja virtualnog stroja parametrom *`-cp`* navodimo direktorij `bin` kao direktorij u kojem se mogu pronaći izvršne verzije svih razreda, navedemo stvorenu arhivu. Primjer pokretanja istog programa prikazan je u nastavku.

```
d:\javaproj\projekt2>java -cp projekt2.jar \
                          hr.fer.zemris.java.tecaj_1.Glavni

Pozdrav! Učimo javu!

Jedna od najpoznatijih formula fizike:
=====================================
  E = m * c^2
```

Stvaranje JAR arhive na ovaj način od krajnjeg korisnika ipak očekuje da zna ponešto o Javi -- primjerice, kako se pokreće virtualni stroj, i možda još gore, kako se zove razred koji ste si Vi zamislili da bi ga trebalo pokrenuti prilikom startanja programa. Bilo bi super kada bismo tu informaciju nekako mogli ugraditi u JAR arhivu kako je krajnji korisnik kasnije ne bi trebao navoditi. To se, dakako, može postići, i ključ je upravo datoteka `MANIFEST.MF`.

U trenutnom direktoriju (`projekt2`) napravite datoteku `mojmanifest.txt` i u nju upišite samo jedan redak kako je prikazano u nastavku.

```
Main-Class: hr.fer.zemris.java.tecaj_1.Glavni
```

Potom nanovo napravite arhivu ali tako da programu **jar** definirate što treba koristiti za stvaranje datoteke `MANIFEST.MF`. To se radi navođenjem slova *m* u drugom parametru i potom navođenjem datoteke koja sadrži podatke za manifest. Obratite pažnju na redosljed slova *f* i *m* jer on određuje i redosljed kojim se tumače daljnji parametri. Primjer je prikazan u nastavku.

```
d:\javaproj\projekt2>jar cfm projekt2.jar mojmanifest.txt -C bin\ .
```

Pogledate li sada sadržaj nastale manifest datoteke koja je prisutna u JAR arhivi, on će biti poprilici kako je prikazano u nastavku.

```
Manifest-Version: 1.0
Created-By: 1.7.0_03 (Oracle Corporation)
Main-Class: hr.fer.zemris.java.tecaj_1.Glavni
```

Redak s informacijom o razredu koji treba pokrenuti bit će nadodan na sadržaj koji stvara sam alat **jar**. Ako raspolažemo ovakvom arhivom, pokretanje programa može se obaviti bitno jednostavnije: umjesto navođenja putanje u kojoj se pretražuju izvršne datoteke i navođenja razreda koji je potrebno pokrenuti, dovoljno je iakoristiti opciju *-jar* i predati naziv JAR arhive. Evo primjera u nastavku.

```
d:\javaproj\projekt2>java -jar projekt2.jar

Pozdrav! Učimo javu!

Jedna od najpoznatijih formula fizike:
=======================================
  E = m * c^2
```

Ovako stvorene arhive u grafičkom korisničkom sučelju se mogu pokrenuti dvoklikom, pa ako je aplikacija pisana tako da s korisnikom komunicira uporabom grafičkog korisničkog sučelja, dobiva se dojam kao da radite s *normalnim* programom koji je pisan za nativni operacijski sustav.



Jednostavnije zadavanje početnog razreda

Ako je zadavanje početnog razreda jedina modifikacija manifesta koju želite provesti, tada nema potrebe da stvarate zasebnu datoteku s tim informacijama. Umjesto toga, programu **jar** možete u prvom parametru umjesto slova *m* predati slovo *e* i potom na mjestu gdje biste dali naziv datoteke s informacijama za manifest navedete puni naziv početnog razreda.

```
jar cfe projekt2.jar hr.fer.zemris.java.tecaj_01.Glavni \
    -C bin\ .
```

Generiranje dokumentacije

Pretpostavimo za sada da ste napisali (tj. prepisali) Javadoc komentare koji su navedeni u primjerima. Koristeći gotov alat odnosno naredbu **javadoc** moguće je generirati HTML dokumentaciju temeljem obrade izvornog koda.

Pretpostavit ćemo da ste pozicionirani u direktoriju `D:\javaproj\projekt2`. Stvorite direktorij `doc` u koji ćemo pohraniti generiranu dokumentaciju.

```
D:\javaproj\projekt2>javadoc -sourcepath src -encoding UTF-8 \
                            -d doc -docencoding UTF-8 \
                            -charset UTF-8 -use -version \
```

```
-author -splitindex -subpackages hr
```

Pokretanjem naredbe **javadoc** u poddirektoriju `doc` nastat će dokumentacija u formatu HTML. Otvorite u web-pregledniku datoteku `doc\index.html` i pogledajte što ste dobili.

Broj parametara koje je potrebno predati ovoj naredbi je nezanemariv. Direktorij u kojem se nalaze izvorni kodovi programa zadaje se preko parametra *`-sourcepath`*. Direktorij u koji treba pohraniti generiranu dokumentaciju zadaje se parametrom *`-d`*. Kodne stranice u skladu s kojima treba tumačiti izvorne kodove programa odnosno generirati HTML dokumente zadaju se preko čak tri parametra: *`-encoding`*, *`-docencoding`* te *`-charset`*. Nakon još nekoliko parametara za koje se čitatelj upućuje na dokumentaciju naredbe **javadoc** posljednji parametar u ovom primjeru *`-subpackages`* omogućava navođenje paketa za koje je potrebno stvoriti dokumentaciju.

Pisanje dokumentacije projekta

Kroz prethodne primjere već smo se upoznali s posebnom vrstom komentara koja se koristi za automatsko generiranje dokumentacije. Radi se o *javadoc* komentarima koji za razliku od običnih višelinijskih komentara započinju s /**. Detaljna dokumentacija alata **javadoc** kao i opis svega podržanoga u okviru načinu kako taj alat temeljem izvornog koda i pomoćnih oznaka u komentarima generira konačnu dokumentaciju dostupni su na adresi http://docs.oracle.com/javase/6/docs/technotes/tools/windows/javadoc.html.

U nastavku ćemo se stoga samo osvrnuti na nekoliko općenitih naputaka.

Dokumentiranje paketa

U svaki paket koji se ži dokumentirati (čitaj: *u svaki paket*) potrebno je dodati datoteku imena `package-info.java`. Sadržaj te datoteke treba sadržavati javadoc komentar koji je napisan direktno iznad deklaracije paketa. Nakon deklaracije paketa više ništa ne bi trebalo biti napisano. Primjerice, datoteka `package-info.java` koja dokumentira paket hr.fer.zemris.java.tecaj_1 bit će smještena u odgovarajuću strukturu direktorija `hr/fer/zemris/java/tecaj_1` i izgledat će poprilici kako je navedeno u nastavku.

Primjer 2.3. Primjer dokumentacije paketa

```
1 /**
2  * Paket sadrži sve razrede koji se koriste u demonstraciji
3  * prvog programa napisanog u Javi. Kako je ovo jednostavan
4  * primjer, nema detaljnije razrade u podpakete.
5  *
6  * @since 1.0
7  */
8 package hr.fer.zemris.java.tecaj_1;
9
```

Bitno je uočiti da se *javadoc* komentar uvijek piše neposredno iznad elementa koji dokumentira. Na to je potrebno paziti kako bi automatsko generiranje dokumentacije alatom **javadoc** rezultiralo korektnom dokumentacijom.

Dokumentiranje razreda

Dokumentacija razreda (a kasnije i sučelja te enumeracija) piše se neposredno ispred deklaracije istoga u izvornom kodu. Pri tome se na tom mjestu navode općeniti komentari, upozorenja te primjeri uporabe koju razred nudi klijentima kao cjelinu. Na tom se mjestu ne dokumentira popis članskih varijabli, metoda te konstruktora. Uz općeniti komentar, često se na kraju

navode još i posebne oznake kojima se definira autor odnosno autori, verzija razreda i slično. Primjer dokumentacije razreda prikazan je u nastavku.

Primjer 2.4. Primjer dokumentacije razreda

```
/**
 * Razred <code>ParniBrojevi</code> korisnicima omogućava rad
 * i obilazak po cijelim parnim brojevima.
 *
 * <p>Najčešći način uporabe jest stvaranje ograničenog
 * podskupa cijelih brojeva i njihov obilazak petljom, kako
 * je ilustrirano u sljedećem primjeru.</p>
 *
 * <pre>
 * ParniBrojevi pb = new ParniBrojevi(100);
 * for(Integer broj : pb) {
 *   // radi nešto s brojem
 * }
 * </pre>
 *
 * @author Marko Čupić
 * @version 1.1
 */
public class ParniBrojevi {

 ...

}

```

Dokumentiranje metoda

Dokumentacija za metode piše se neposredno ispred deklaracije metode u izvornom kodu razreda ili sučelja. Pri tome se kao prva rečenica navodi sažet opis metode koji će alat za generiranje dokumentacije pročitati i preuzeti prilikom generiranja sumarnog popisa svih metoda i njihovog kratkog opisa. Od sljedeće rečenice na dalje navodi se detaljna dokumentacija metode koja treba sadržavati opis funkcionalnosti metode te eventualna upozorenja kojih klijent mora biti svjestan.

Ono što se u dokumentaciji metode ne piše jest kako je sama metoda implementirana (odnosno, nije ideja dokumentacije da svaku metodu isprogramirate dva puta -- jednom u pseudokodu a drugi puta u Javi). Osim općeg opisa dokumentacija metode bi trebala sadržavati i dokumentaciju svih argumenata koji se prosljeđuju metodi (uporabom oznake @param), dokumentaciju povratne vrijednosti (uporabom oznake @return) te dokumentaciju iznimaka koje metoda može generirati te uvjeta pod kojim se one javljaju (uporabom oznake @throws).

Primjer dokumentacije metode prikazan je u nastavku.

Primjer 2.5. Primjer dokumentacije metode

```
/**
 * Metoda provjerava nalazi li se broj predan kao argument
 * u skupu parnih brojeva koje trenutni primjerak razreda
 * predstavlja. Argument može biti bilo koji cijeli broj,
 * međutim, ne smije biti <code>null</code>.
 *
 * @param broj broj koji treba ispitati
 * @return <code>true</code> ako je predani argument u
```

```
 9  *          skupu brojeva koje predstavlja primjerak razreda;
10  *          <code>false</code> inače.
11  * @throws NullPointerException ako se kao argument preda
12  *          vrijednost <code>null</code>.
13  *
14  * @since 1.1
15  */
16 public boolean contains(Integer broj) {
17  ...
18 }
19
```

Ponekad je teško odlučiti što (i u kojoj mjeri) treba navesti u okviru dokumentacije. Jedno od pravila kojih se dobro držati jest da dokumentacija ne bi trebala detaljizirati korake implementiranog algoritma već bi se trebala odnositi na semantiku. Zamislimo tako primjerice metodu:

```
double racunajENaPotenciju(double x);
```

čija je zadaća izračunati vrijednost prirodne konstante *e* dignute na zadanu potenciju *x*. U komentaru takve metode rečenice poput sljedećih nemaju što tražiti.

> Najprije se pomoćna varijabla postavi na vrijednost nula. Potom se u `for`-petlji ta vrijednost malo po malo uvećava dodavanjem novih članova kojima se traženi izraz aproksimira uporabom *Taylorovog razvoja*. Petlja se ponavlja 10 puta i potom se vraća vrijednost prethodno spomenute pomoćne varijable.

Sljedeće rečenice bi pak mogle (i trebale) činiti dio dokumentacije navedene metode.

> Metoda računa vrijednost prirodne konstante dignute na zadanu potenciju pri čemu se koristi *Taylorov razvoj* oko točke 0 uz prvih deset članova. Obratite pažnju da je ovime definirana i procjena maksimalne pogreške koju metoda radi -- ako je ta pogreška neprihvatljiva, metodu ne treba koristiti.

Iz ovih primjera slijedi da bi dokumentacija metoda trebala sadržavati upute korisnicima odnosno klijentima iz kojih bi moralo biti vidljivo što metoda radi, koja su ograničenja na ulazne vrijednosti te koja su ograničenja na rezultat koji tako nastaje. Zadaća *javadoc* komentara nije da drugom programeru objasne *kako* je metoda implementirana.

Automatizacija razvojnog ciklusa

Kroz prethodna dva primjera upoznali smo se s temeljnim alatima koji su sastavni dio platforme JDK i koji nam omogućavaju prevođenje, pakiranje i pokretanje Java programa kao i generiranje dokumentacije u formatu HTML. Vjerojatno ste kroz te primjere uočili jednu negativnu stranu takvog pristupa: pokretanje svakog od alata zahtjeva navođenje popriličnog broja parametara i vremenski je vrlo zahtjevno. Međutim, to nikako nije problem koji je specifičan za Javu -- isti problem prisutan je i pri razvoju u drugim programskim jezicima, poput razvoja u programskom jeziku C ili C++. Na sreću, i rješenja su ista -- umjesto da svaki puta ručno pokrećemo sve alate, možemo se poslužiti alatom za automatizaciju različitih faza u razvojnom ciklusu. Tako je primjerice za programski jezik C i C++ na raspolaganju alat **make** i njegove varijante. Za programski jezik Java popis raspoloživih alata je širok i raznolik. Međutim, dugo vremena primat u ovom području imao je alat **ant**; kroz posljednjih nekoliko godina razvija se i dosta koristi i alat **maven**, a možemo još izdvojiti i alat **gradle**.

Uporaba alata ant

Skinite i instalirajte alat Apache Ant. Malo detaljnija uputa dostupna je u dodatku B. Također, napravite novi direktorij za naš sljedeći projekt: `projekt3`. U tom vršnom direktoriju napravite

poddirektorij `src/main/java` te unutar tog direktorija iskopirajte samo sadržaj direktorija `src` izprethodnog projekta `projekt2`. Struktura koja će time nastati prikazana je u nastavku.

```
D:\
  +-- javaproj
      +-- projekt3
          +-- src
              + main
                +-- java
                    +-- hr
                        +-- fer
                            +-- zemris
                                +-- java
                                    +-- tecaj_1
                                        +-- Formule.java
                                        +-- Glavni.java
```

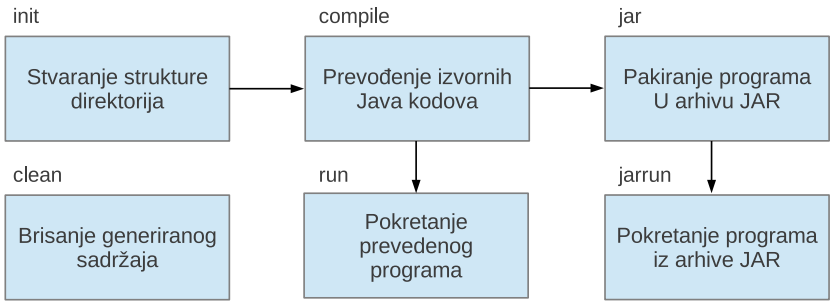
Razlog za ovakvu modifikaciju je kasnije dodavanje testova napisanog programskog koda i želja da se testovi odvoje od izvornog koda programa. Stoga ćemo izvorne kodove programa smještati u direktorij `src/main/java` dok će izvorni kodovi testova biti smješteni u direktorij `src/test/java`.

Sljedeći korak je izrada konfiguracijske datoteke za alat **ant**. Ova datoteka sastojat će se iz definicije niza zadataka koje je potrebno obaviti u okviru razvojnog ciklusa projekta. Alat **ant** ove zadatke naziva *ciljevima* (engl. *target*).

Prisjetimo se koje smo sve korake do sada radili preko komandne linije: najprije smo stvarali potrebnu strukturu direktorija (primjerice, direktorij `bin`). Potom smo program pokretali navođenjem svih potrebnih parametara kako bi Javin virtualni stroj mogao pronaći sve datoteke. Konačno, uporabom alata **jar** izvršni kod smo pakirali u arhivu čime smo program mogli jednostavnije distribuirati krajnjim korisnicima (ali i pokretati na jednostavniji način).

Slika 2.1 daje pregledan grafički prikaz ovih zadataka kao i njihovih međuovisnosti. Svakom zadatku dano je kratko ime (navedeno iznad kvadratića koji prikazuje zadatak) te kratki opis. Strelicama su prikazane međuovisnosti pojedinih zadataka. Tako primjerice zadatak prevođenja izvornog koda očekuje da su na disku stvorene potrebne strukture direktorija (konkretno, direktorij `bin`) dok postupak izrade JAR arhive očekuje da generira izvršni kod (a time tranzitivno i da su stvorene potrebne strukture direktorija). Svaki od ovih zadataka može se preslikati u jedan cilj alata **ant**.

Slika 2.1. Pojednostavljen popis zadataka u razvojnom procesu



Opis jednostavnog razvojnog procesa koji se sastoji od nekoliko koraka.

Za potrebe ovog primjera (ali i svih daljnjih primjera) malo ćemo modificirati strukture direktorija s kojima ćemo raditi. Pretpostavit ćemo da postoji direktorij `build` u koji ćemo pohranjivati sve privremene sadržaje koji se generiraju. U njegov poddirektorij `classes` odnosno punom

stazom `build/classes` pohranit ćemo izvršni Java kod koji će generirati prevodilac temeljem izvornih kodova. Direktorij `dist` (pokrata od *distribucija*) služit će za generiranje konačnih produkata. Tako ćemo unutra primjerice u poddirektorij `lib` smjestiti JAR arhivu koju ćemo generirati a u poddirektorij `doc` HTML dokumentaciju. Očekivana struktura direktorija s kojima ćemo dalje raditi prikazana je u nastavku.

```
D:\
  +-- javaproj
        +-- projekt3
              +-- src
              |    +-- main
              |         +-- java
              |               +-- ..
              +-- build
              |    +-- classes
              |         +-- ...
              +-- dist
              |    +-- lib
              |         +-- ...
```

Ugradnja podrške za alat **ant** svodi se na izradu datoteke naziva `build.xml` koja mora biti mještena u korijenski direktorij projekta da bi je alat **ant** automatski pronašao. Ogledni sadržaj ove datoteke prikazan je u primjeru 2.6.

Primjer 2.6. Primjer konfiguracije za alat ant

```
 1 <project name="Projekt3" default="jar" basedir=".">
 2
 3  <description>
 4   Build datoteka za projekt 3.
 5  </description>
 6
 7  <!-- Postavljanje globalnih varijabli -->
 8  <property name="src" location="src"/>
 9  <property name="src.java" location="${src}/main/java"/>
10  <property name="build" location="build"/>
11  <property name="build.classes" location="${build}/classes"/>
12  <property name="dist"  location="dist"/>
13  <property name="dist.lib"  location="${dist}/lib"/>
14  <property name="program"
15   value="hr.fer.zemris.java.tecaj_1.Glavni" />
16
17  <target name="init">
18   <!-- Stvaranje vremenske oznake -->
19   <tstamp/>
20   <!-- Stvaranje potrebnih direktorija -->
21   <mkdir dir="${build}"/>
22   <mkdir dir="${dist}"/>
23  </target>
24
25  <target name="compile" depends="init"
26   description="Prevođenje izvornog koda">
27   <mkdir dir="${build.classes}"/>
28   <!-- Prevođenje Java koda iz ${src} u ${build} -->
29   <javac srcdir="${src.java}" destdir="${build.classes}"
30    encoding="UTF-8" debug="on"
31    debuglevel="lines,vars,source"
32    includeAntRuntime="false" />
33  </target>
```

```
34
35  <target name="run" depends="compile"
36   description="Pokretanje programa">
37   <!-- Poziv virtualnog stroja koji će pokrenuti
38    napisani program. -->
39   <java classname="${program}" classpath="${build.classes}"
40    fork="true">
41    <jvmarg value="-Dfile.encoding=IBM852" />
42   </java>
43  </target>
44
45  <target name="jar" depends="compile"
46   description="Pakiranje programa u arhivu JAR" >
47   <!-- Stvaranje direktorija za distribuciju -->
48   <mkdir dir="${dist.lib}"/>
49
50   <!-- Arhiviranje svega iz ${build} u arhivu
51    ${ant.project.name}-${DSTAMP}.jar -->
52   <jar jarfile="${dist.lib}/${ant.project.name}-${DSTAMP}.jar"
53    basedir="${build.classes}">
54    <manifest>
55     <attribute name="Main-Class" value="${program}"/>
56    </manifest>
57   </jar>
58  </target>
59
60  <target name="clean"
61   description="Brisanje generiranog sadržaja" >
62   <!-- Obriši direktorije ${build} i ${dist} -->
63   <delete dir="${build}" failonerror="false" />
64   <delete dir="${dist}" failonerror="false" />
65  </target>
66
67 </project>
68
```

Datoteka `build.xml` je XML datoteka čiji je vršni tag nazvan project. Atribut name omogućava definiranje imena projekta, atribut default omogućava definiranje cilja koji je potrebno pokrenuti ako se cilj eksplicitno ne zada prilikom pokretanja alata **ant** te atribut basedir omogućava definiranje direktorija s obzirom na koji će se razriješavati sve relativne staze.

Retci 3--5 sadrže opis projekta dok retci 7--15 sadrže definicije pomoćnih varijabli koje će se dalje koristiti kroz skriptu. Vrijednost varijabli kroz skriptu ćemo dohvaćati koristeći sintaksu `${ime}`. Redak 8 definira sadržaj varijable `src`. Ona čuva naziv direktorija u kojem će biti pohranjeni svi izvorni kodovi programa. Unutar tog direktorija, izvorni kod programa smjestit ćemo u poddirektorij `main/java` (što definira varijabla `src.java` u retku 9, dok će izvorni kod testova (jednom kada ih krenemo pisati) biti smješten u poddirektorij `test/java` (ovaj dio konfiguracije još nije napravljen pa nije niti prikazan). Uočite kako se već u definiciji varijable `src.java` pozivamo preko prethodno definirane varijable `src` na naziv direktorija unutar kojeg bi svi izvorni kodovi trebali biti smješteni.

Retci 10 i 11 definiraju direktorij u koji će se pohraniti sav privremeno generirani sadržaj (varijabla `build`) te poddirektorij u koji će se smjestiti izvršni kodovi programa (varijabla `build.classes`). U retcima 12 i 13 definiraju se direktorij u koji će se pohraniti konačni sadržaj projekta (varijabla `dist`) te direktorij u koji će pohraniti generirana JAR arhiva (varijabla `dist.lib`).

Konačno, kroz retke 14 i 15 definira se varijabla koja sadrži puni naziv razreda koji predstavlja ulaznu točku u naš program (odnosno koji sadrži metodu `main`).

Osnovna prednost definiranja konfiguracije na opisani način jest jednostavna mogućnost promjene strukture direktorija na jednom mjestu. Dovoljno je promijeniti sadržaj ovih varijabli jer svi ciljevi koji su kasnije definirani ne hardkodiraju ove vrijednosti već se pozivaju na odgovarajuće varijable.

Svaki od ciljeva u skripti predstavljen je jednim tagom target. Ovaj tag omogućava korisniku definiranje imena cilja, opisa cilja kao i popisa ciljeva koje potrebno izvršiti prije izvršavanja tog cilja (odnosno, navođenje preduvjeta) -- ovome služi atribut depends.

Unutar svakog cilja navodi se skup koraka koje će **ant** izvršiti redoslijedom kojim su navedeni. Primjerice, cilj init koji je definiran u retcima 17 do 23 sastoji se od tri koraka -- inicijalizacije vremenske značke te stvaranja dvaju direktorija: jednog zadanog varijablom `build` te jednog zadanog varijablom `dist`. Alat **ant** podržava izuzetno velik skup naredbi koje se mogu koristiti i čitatelja se upućuje na upoznavanje s njegovom dokumentacijom.

U retcima 25 do 33 definira se cilj naziva compile. U okviru tog cilja najprije se stvara direktorij u koji će se smjestiti izvršni kodovi a potom se poziva Java prevodioc. Uočite kako se preko atributa zadaju sve one informacije koje smo inače morali unositi preko komandne linije -- koji direktorij sadrži izvorne kodove programa (atribut srcdir), u koji direktorij treba smjestiti generirani izvršni kod (atribut destdir), koju kodnu stranicu treba koristiti prilikom obrade izvornog koda programa (atribut encoding) te želimo li da se u generirani izvršni kod uključe i informacije koje će pomoći lakšem provođenju debugiranja (i koje; atributi debug i debuglevel). Prilikom deklaracije ovog cilja definirano je da je preduvjet za njegovo izvođenje izvođenje cilja init.

U retcima 35 do 43 na sličan se način definira cilj run čiji je zadatak omogućiti pokretanje razvijenog programa direktno koristeći generirani izvršni kod. Stoga je kao preduvjet ovog cilja naveden cilj compile jer je njegova zadaća upravo stvaranje izvršnog koda. Cilj se sastoji od samo jedne naredbe: od poziva virtualnog stroja kojemu se predaje naziv razreda koji treba pokrenuti i staza do direktorija u kojem su smješteni izvršni kodovi. Dodatno, samom se virtualnom stroju predaje argument koji definira koju kodnu stranicu treba koristiti prilikom komuniciranja s okolinom. Kako smo već prethodno napomenuli, ovo je važno na operacijskom sustavu Windows kako bi se ispravno tretirali dijakritički i slični znakovi. Radite li na modernijim verzijama operacijskog sustava Linux zadavanje tog argumenta je nepotrebno jer će operacijski sustav koristiti Unicode koji će također biti podržan i od konzole u kojoj pokrećete program.

Retci 45 do 58 sadrže definiciju cilja jar koji se sastoji od dvije naredbe: stvaranja potrebnog direktorija u koji će se smjestiti generirana arhiva te poziva alata **jar** koji će obaviti potrebno pakiranje. Preko odgovarajućih atributa i ugniježđenih tagova zadaju se sve potrebne informacije (naziv JAR arhive koju treba stvoriti, koji se direktorij pakira te što se dodaje u datoteku manifest). Uočite način na koji se gradi ime JAR arhive koju želimo stvoriti: najprije se pozivamo na varijablu `ant.project.name` čija je vrijednost jednaka imenu trenutnog projekta (u ovom slučaju to je projekt3), potom dodajemo crticu i zatim umećemo sadržaj varijable `DSTAMP` i naziv završavamo dodavanjem podniza ".jar". Varijablu `DSTAMP` nismo nigdje eksplicitno definirali; međutim, ona nam stoji na raspolaganju i njezin je sadržaj jednak trenutnom datumu u formatu `yyyyMMdd`. Primjerice, pokrenemo li ovaj cilj u projektu naziva Projekt3 na datum 5. ožujka 2013., ime arhive će biti Projekt3-20130305.jar.

Konačno, retci 60 do 65 sadrže definiciju cilja clean čija je zadaća obrisati sav sadržaj koji se automatski generira.

Pokretanja pojedinih poslova obavlja se pozivom naredbe **ant**. Ako se naredba pokrene bez ikakvih argumenata, naredba će u trenutnom direktoriju potražiti datoteku `build.xml`, pogledati koji je cilj u deklaraciji projekta proglašen kao pretpostavljeni (atributom default) i potom pokrenuti taj cilj. Naravno, ako taj cilj ima preduvjete, najprije će se pokrenuti izvođenje tih preduvijeta, odnosno ako oni imaju preduvjete, izvođenje njihovih preduvjeta i tako dalje.

Ako se želi pokrenuti neki konkretni cilj, tada se njegovo ime može predati kao argument. Primjerice, ako želimo pokrenuti cilj run, napisali bismo sljedeće.

```
D:\javaproj\projekt3>ant run
```

Prepišite danu skriptu, zadajte sljedeće tri naredbe i proučite rezultat nakon svake od njih.

```
D:\javaproj\projekt3>ant jar
D:\javaproj\projekt3>ant run
D:\javaproj\projekt3>ant clean
```

U odnosu na razvojni ciklus prikazan slikom 2.6, trenutnoj konfiguracijskoj datoteci nedostaje još cilj koji bi program pokrenuo temeljeći se na stvorenoj JAR arhivi. Taj cilj je prikazan u primjeru 2.7; dodajte definiciju tog cilja i provjerite da se program doista može pokrenuti.

Primjer 2.7. Definicija cilja runjar za alat ant

```
<target name="jarrun" depends="jar"
 description="Pokretanje programa iz JAR-a">
 <!-- Poziv virtualnog stroja koji će pokrenuti
  napisani program. -->
 <java classname="${program}"
  classpath="${dist.lib}/${ant.project.name}-${DSTAMP}.jar"
  fork="true">
  <jvmarg value="-Dfile.encoding=IBM852" />
 </java>
</target>


```

Ostalo nam je još u skriptu dodati i cilj kojim ćemo pokrenuti generiranje dokumentacije. Stvorite najprije datoteku `src/main/java/overview.html` čiji je sadržaj prikazan u primjeru 2.8. Navedena HTML datoteka omogućava definiranje teksta koji će poslužiti kao pregled razvijenog programa. Potom u `build.xml` dodajte cilj javadoc koji je prikazan u primjeru 2.9. Snimite datoteku i pokrenite naredbu **ant** uz argument javadoc. Skripta će generirati dokumentaciju u HTML formatu i sve će datoteke smjestiti u direktorij `dist/doc`. Pronađite unutra datoteku `index.html` i otvorite je u web-pregledniku. Proučite generirani sadržaj.

Primjer 2.8. Pregledna stranica za dokumentaciju programa

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
 <HEAD>
  <TITLE>Prvi program</TITLE>
 </HEAD>
 <BODY>
  Ovaj program služi demonstraciji osnovnih koraka pri izradi
  Java programa, njegovog prevođenja, pakiranja, pokretanja
  te generiranja dokumentacije. Više informacija o samim
  razredima dostupno je u okviru dokumentacije istih.
 </BODY>
</HTML>


```

Primjer 2.9. Definicija cilja javadoc za alat ant

```
<target name="javadoc" depends="compile"
 description="Generiranje dokumentacije" >
 <!-- Stvaranje direktorija za distribuciju -->
 <mkdir dir="${dist}/doc"/>
 <javadoc packagenames="hr.*"
  sourcepath="${src.java}"
```

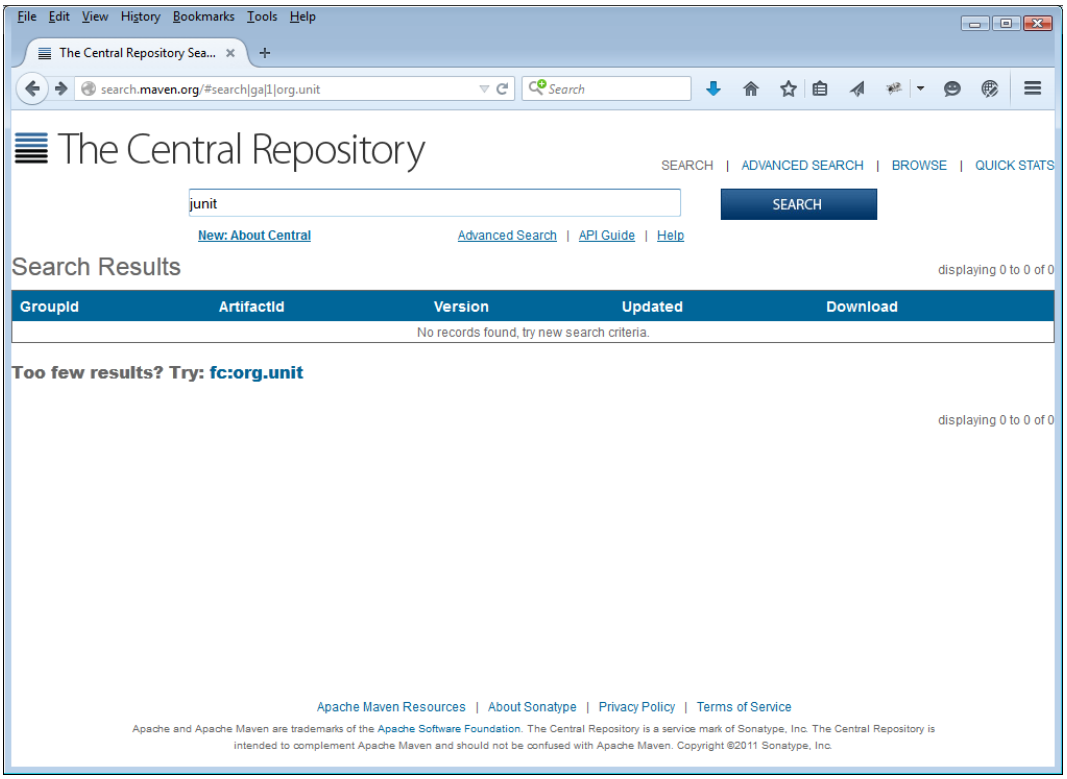
```
 7    overview="${src.java}/overview.html"
 8    defaultexcludes="yes"
 9    destdir="${dist}/doc"
10    encoding="UTF-8"
11    docencoding="UTF-8"
12    charset="UTF-8"
13    author="true"
14    version="true"
15    use="true"
16    windowtitle="Dokumentacija programa">
17    <doctitle>
18     <![CDATA[<h1>Prvi program</h1>]]>
19    </doctitle>
20    <bottom>
21     <![CDATA[<i>Copyright &#169; 2013 ACME.
22     All Rights Reserved.</i>]]>
23    </bottom>
24   </javadoc>
25  </target>
26
27
```

Uporaba alata maven

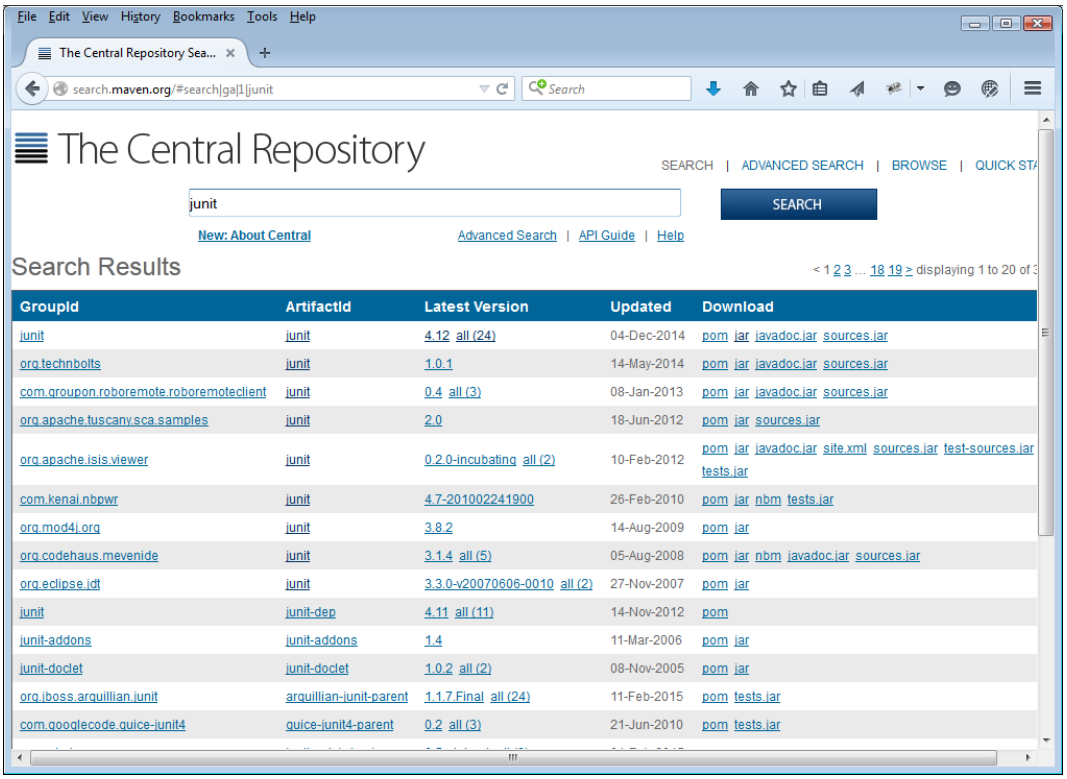
Skinite i instalirajte alat Apache Maven. Malo detaljnija uputa dostupna je u dodatku B. U odnosu na alat **ant**, maven uz mogućnost upravljanja razvojnim ciklusom projekta donosi integriranu mogućnost dohvata svih potrebnih biblioteka direktno iz javno dostupnih repozitorija na Internetu. Tako, primjerice, ako u projektu zatrebamo biblioteku za testiranje koda (junit), bit će dovoljno u konfiguracijsku datoteku projekta dodati nekoliko redaka XML-a kojima definiramo ovisnost prema toj biblioteci. Obratite pažnju da biste prilikom rada s ovim alatom morali biti spojeni na Internet kako bi alat mogao dohvatiti potrebne verzije biblioteka (barem prvi puta, a kasnije pokušava provjeravati ima li još uvijek zadnju verziju biblioteka ili je u međuvremenu izašlo nešto novije).

Da bi mehanizam automatskog dohvaćanja potrebnih ovisnosti radio, maven zahtjeva da se svaki projekt jednoznačno definira s tri parametra: *identifikatorom grupe* u koju pripada, *identifikatorom projekta* te *verzijom projekta*. Za ova tri parametra maven koristi nazive *groupId*, *artifactId* te *version*. Grupa (ili početak naziva grupe) u određenom smislu predstavlja naziv koji je jedinstven za pojedinu organizaciju (ili pojedinca). Grupe se, kao i i paketi u Javi, grade hijerarhijski. Primjerice, danas najčešće korištena biblioteka za testiranje koda u Javi (biblioteka JUnit) nalazi se u grupi `junit`. Poštivanje ove konvencija važno je za slučaj da projekt koji razvijate želite učiniti javno dostupnim - tada treba paziti da ne dođe do kolizije s postojećim projektima. Artefakt je naziv projekta. Unutar iste grupe moguće je imati više različitih projekata (razlikovat će se po ovom parametru). Konačno, verzija je parametar kojim je moguće pratiti različite verzije istog projekta. U mavenovim javno dostupnim repozitorijima pronaći ćete projekte s nizom različitih verzija. Na ovaj se način omogućava da konkretan projekt definira ovisnost o točno određenoj verziji projekta (čime se osigurava otpornost na nekompatibilne promjene biblioteka o kojima ovisimo). Na Internetu postoji i tražilica mavenovog centralnog repozitorija koja je dostupna na adresi http://search.maven.org/. Otiđite na tu stranicu i unesite upit `junit` (slika 2.2). Dobit ćete ispis svih projekata koji imaju "junit" u nazivu grupe ili artefakta (slika 2.3). Kliknite u retku koji odgovara grupi `junit` i artefaktu `junit` na tekst "all" u stupcu *verzije*. Dobit ćete stranicu na kojoj su prikazane sve verzije ovog projekta (slika 2.4). Obratite odmah pažnju da je to ekvivalentno kao da ste postavili upit `g:"junit" AND a:"junit"` što je ispisano u kutiji za unos upita. Kliknite sada u prikazanoj tablici na posljednju verziju (baš u stupcu *verzije*; u tablici su rezultati poredani kronološki od najnovijih prema najstarijima; u trenutku pisanja ovog teksta najnovija verzija je bila 4.12). Otvorit će se stranica (slika 2.5) s detaljnim informacijama o projektu (i to baš o toj verziji).

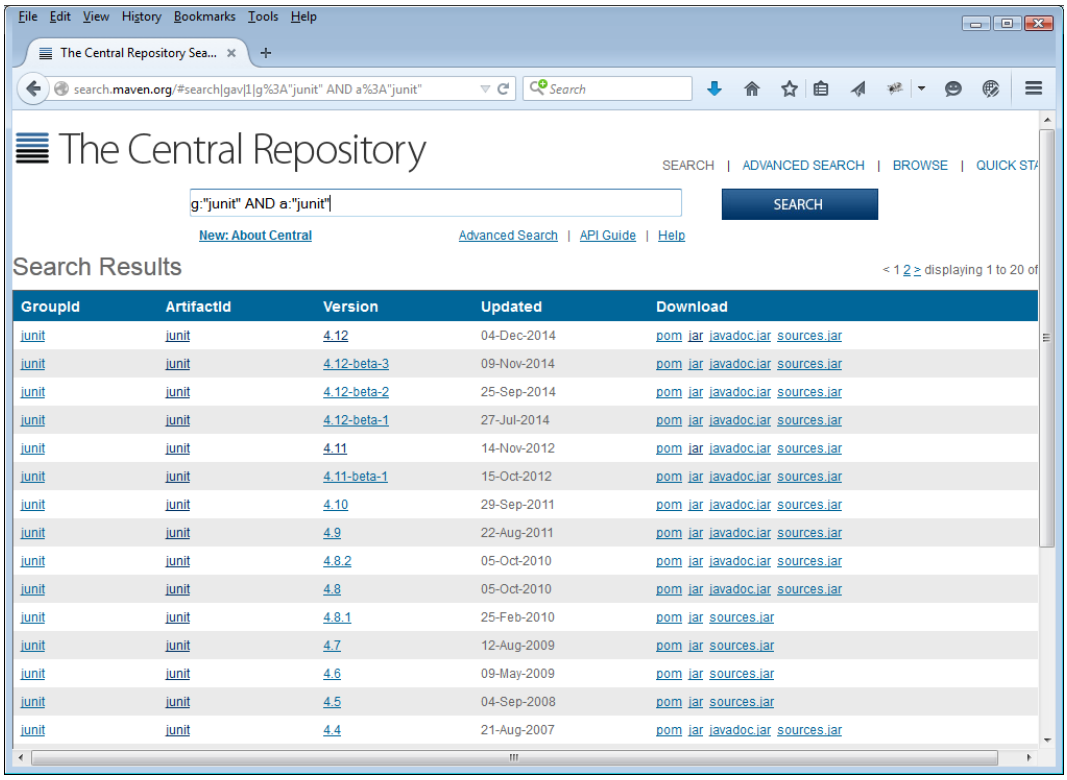
Slika 2.2. Tražilica mavenovog centralnog repozitorija



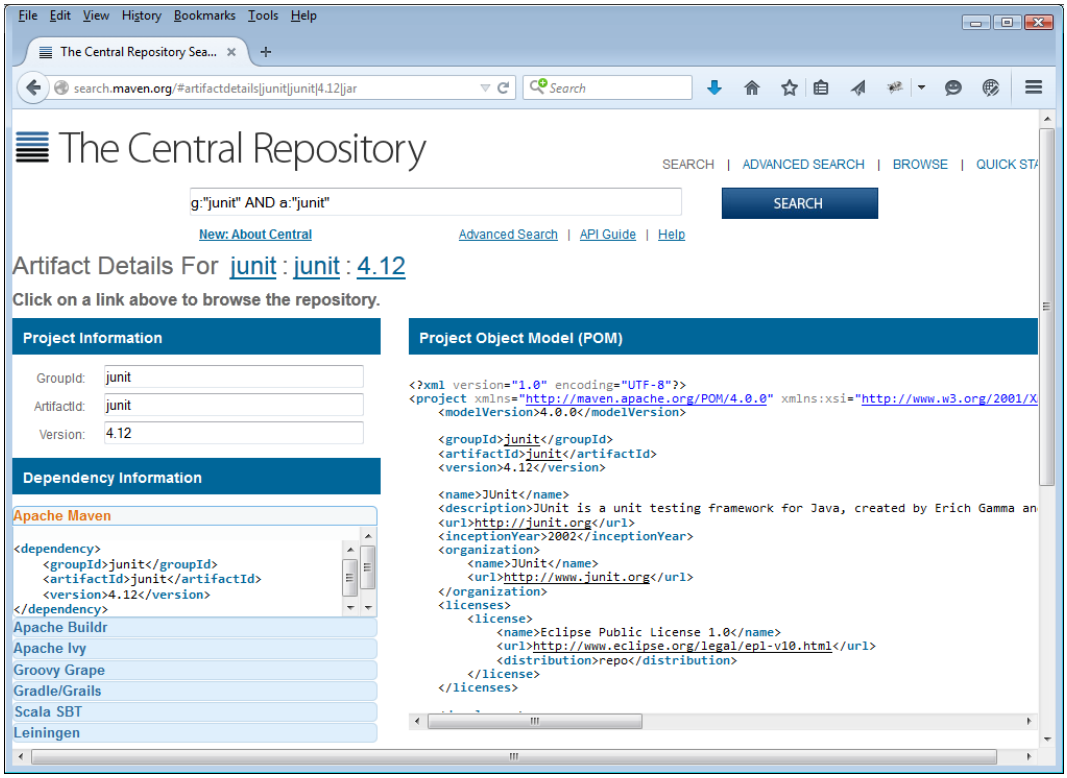
Slika 2.3. Rezultati pretrage za projektima/artefaktima koji sadrže "junit"



Slika 2.4. Prikaz svih dostupnih verzija projekta junit grupe junit



Slika 2.5. Detaljni prikaz informacija o projektu verzije 4.12



Pri dnu stanice (nije prikazano na slici) pronaći ćete linkove preko kojih možete dohvatiti biblioteku i izvorni kod samog projekta. Ono što je međutim interesantnije je uočiti s lijeve strane karticu *Dependency Information*, gdje je prikazano što treba dodati u konfiguraciju projekta za

pojedine alate da bi ti alati sami povukli ovu biblioteku s Interneta. Na prikazanoj slici već je prikazano kako to izgleda za alat Maven:

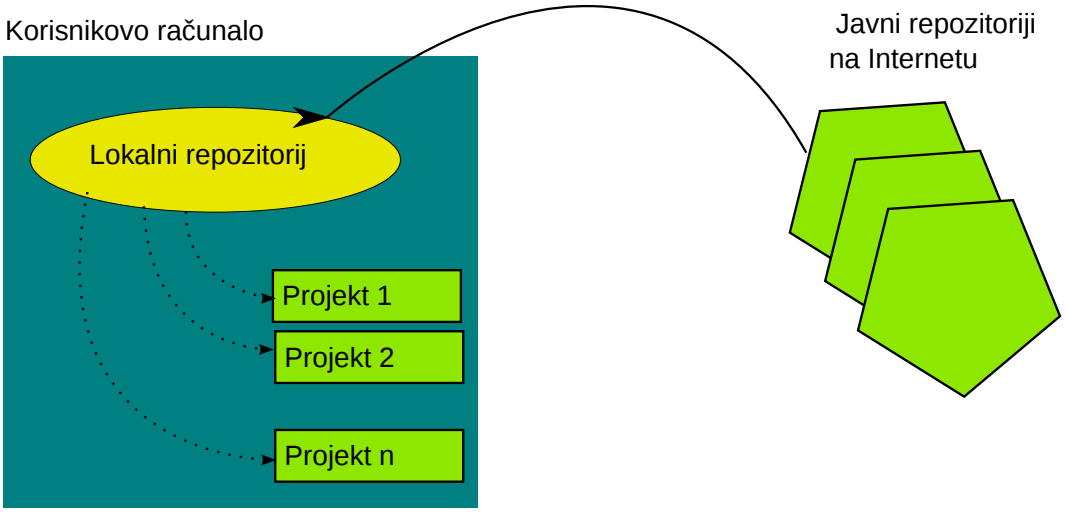
```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

Uočite: prilikom deklariranja ovisnosti, definirat ćemo naziv grupe, naziv projekta te verziju projekta. Ako nam ta biblioteka treba samo za potrebe testiranja (a nije potrebna za izvođenje našeg koda), tada ćemo deklaraciju ovisnosti dopuniti definicijom dosega ovisnosti kako je prikazano u nastavku.

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Prije no što krenemo na izradu prvog projekta uporabom alata maven, pojasnimo još malo kako je čitav sustav zamišljen. Već smo spomenuli da maven biblioteke može automatski dohvaćati s Interneta. Ako na računalu razvijate dva projekta od kojih oba trebaju iste biblioteke, uobičajena je praksa bila (prije mavena) da se u oba projekta naprave poddirektoriji `lib` te da se ista biblioteka iskopira u oba projekta. Ovo, osim što nepotrebno zauzima diskovni prostor, vrlo brzo postaje nepraktično. Stoga će maven na korisnikovom računalu u jednom zajedničkom direktoriju (obično u *home* direktoriju korisnika, pa poddirektoriju `.m2`) napraviti lokalni repozitorij u koji smješta sve što skida s Interneta. Svaki od korisnikovih projekata potom će u classpath dodati biblioteke direktno s ovog mjesta. Slika 2.6 prikazuje ovakvu organizaciju.

Slika 2.6. Mavenovi repozitoriji: korisnički projekti, lokalni repozitorij, javni repozitoriji





Bilješka

U slučaju kada sam korisnik na vlastitom računalu razvija više projekata od kojih je jedan biblioteka koju koristi drugi projekt, važno je zapamtiti da nakon svake izmjene projekta koji gradi biblioteku istu treba ubaciti u mavenov lokalni repozitorij jer se od tamo dohvaćaju sve deklarirane ovisnosti. Ovo ćete u mavenu napraviti naredbom **mvn install**.

Nastavimo sada s primjerom koji smo započeli kod alata Apache Ant. Pozicionirajte se u vršni direktorij u kojem smo radili sve projekte (`D:\javaproj`). Potom zadajte sljedeću naredbu.

```
D:\javaproj>mvn -B archetype:generate \
               -DarchetypeGroupId=org.apache.maven.archetypes \
               -DarchetypeArtifactId=maven-archetype-quickstart \
               -DgroupId=hr.fer.zemris.java.tecaj_1 \
               -DartifactId=projekt3m
```

Pokrenut će se alat **mvn** (maven) koji će za nas izgenerirati prazan projekt i cjelokupnu strukturu direktorija. Prva dva argumenta koja smo zadali u naredbi (archetypeGroupId i archetypeArtifactId) određuju predložak prema kojem će se napraviti novi projekt (postoje predlošci za jednostavne aplikacije, web-aplikacije i druge). Sljedeća dva argumenta (groupId i artifactId) definiraju grupu i naziv našeg projekta. Zastavica **-B** govori da ne želimo raditi u interaktivnom modu gdje bi nas alat pitao za dodatne podatke. Po pokretanju alata, vršni direktorij projekta koji će alat napraviti odgovarat će zadanom nazivu artefakta i bit će `projekt3m`. U tom vršnom direktoriju pronaći ćete poddirektorij `src` odnosno cjelokupnu podgranu direktorija koja je zadužena za pohranu izvornih kodova programa `src/main/java`. Dodatno, u vršnom direktoriju projekta naći ćete i konfiguracijsku skriptu naziva `pom.xml` koju koristi alat **mvn**. Struktura direktorija i datoteka koja je nastala pokretanjem prethodne naredbe prikazana je u nastavku.

```
D:\
  +-- javaproj
      +-- projekt3m
          +-- src
          |   + main
          |     +-- java
          |         +-- hr
          |             +-- fer
          |                 +-- zemris
          |                     +-- java
          |                         +-- tecaj_1
          |                             +-- App.java
          +-- pom.xml
```

Otvorite datoteku `pom.xml` i pogledajte njezin sadržaj. Sadržaj datoteke trebao bi odgovarati onome prikazanom u primjeru 2.10. U ovoj datoteci pronaći ćete sve informacije koje smo naveli preko komandne linije te neke dodatne. Primjerice, tag packaging alatu **maven** definira što treba biti rezultat ovog projekta: u našem primjeru u konfiguracijskoj datoteci piše `jar` čime će **maven** automatski generirati JAR arhivu.

Primjer 2.10. Primjer osnovne verzije datoteke `pom.xml`

```
 1 <project xmlns="http://maven.apache.org/POM/4.0.0"
 2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 4  http://maven.apache.org/maven-v4_0_0.xsd">
 5   <modelVersion>4.0.0</modelVersion>
 6   <groupId>hr.fer.zemris.java.tecaj_1</groupId>
 7   <artifactId>projekt3m</artifactId>
 8   <packaging>jar</packaging>
 9   <version>1.0-SNAPSHOT</version>
10   <name>projekt3m</name>
11   <url>http://maven.apache.org</url>
12   <dependencies>
13     <dependency>
14       <groupId>junit</groupId>
15       <artifactId>junit</artifactId>
16       <version>3.8.1</version>
```

```
17        <scope>test</scope>
18      </dependency>
19    </dependencies>
20 </project>
21
```

Uočite također kako u konfiguracijskoj datoteci nigdje nije navedeno koji su sve ciljevi potrebni, kojim se redosljedom izvode, koji ovisi o kojemu i slično. To je fundamentalna razlika između alata poput **ant** i alata poput **maven**: dok se **ant** vodi načelom *konfiguracija a ne konvencija*, **maven** je predstavnik alata koji se vode načelom *konvencija a ne konfiguracija* (nažalost, ne baš pretjerano uspješno). Naime, **maven** već inicijalno dolazi s unaprijed definiranim i pretpostavljenim ciljevima i načinom provođenja razvojnog procesa. U tom smislu, sam alat unaprijed definira u kojem se točno direktoriju moraju nalaziti svi izvorni kodovi programa, u kojem se točno direktoriju moraju nalaziti izvorni kodovi testova i slično. Ako ste s time zadovoljni, odlično. Ako niste, e onda ste često u problemima. Ovisno o tome što želite promijeniti, to se ponekad da relativno jednostavno iskonfigurirati (*khm, khm, konvencija a ne konfiguracija?*) intervencijom u datoteku `pom.xml`; međutim, ponekad je dobiti ono što želite (ako se to ne uklapa u mavenovu viziju strukture projekta) gotovo nemoguće.

Pogledajmo jedan primjer. Inicijalno, alat **maven** pretpostavlja da ćete kod pisati koristeći sintaksu koja je bila definirana specifikacijom programskog jezika Java verzije 1.5. Kako je danas ta verzija već zastarjela, pretpostavimo da želimo konfigurirati **maven** tako da koristi verziju 1.8. Potrebna modifikacija prikazana je u primjeru 2.11. Ovaj segment koda potrebno je ubaciti u originalnu datoteku nakon što je zatvoren tag dependencies, tj. u redak 20 tako da upadne nakon tog taga ali prije no što je zatvoren sam projekt.

Primjer 2.11. Zadavanje verzije prevodioca koji je potrebno koristiti u datoteci `pom.xml`

```
 1 <build>
 2   <plugins>
 3  <plugin>
 4   <groupId>org.apache.maven.plugins</groupId>
 5   <artifactId>maven-compiler-plugin</artifactId>
 6   <version>3.3</version>
 7   <configuration>
 8    <source>1.8</source>
 9    <target>1.8</target>
10   </configuration>
11  </plugin>
12   </plugins>
13 </build>
14
```

Što smo točno ovime postigli? Maven se za svaki korak izgradnje projekta oslanja na pluginove. Tako postoji plugin za kompajliranje koda, plugin za testiranje koda, plugin za generiranje dokumentacije i tako dalje. Neke od pluginova maven deklarira automatski (svaki projekt želimo moći iskompajlirati a bez da to eksplicitno tražimo). Ako nam pretpostavljene postavke tog plugina ne odgovaraju pa ih želimo promijeniti (ili ako želimo dodati novi plugin koji će obaviti određen zadatak), to radimo tako da u sekciji *build* u podsekciju *plugins* dodamo definiciju plugina i podesmo ga. Uočite da baš kao i projekti, u mavenu i pluginovi imaju svoju grupu, naziv i verziju pa se dodavanje i konfiguriranje plugina svodi na definiranje grupe, imena i verzije te konfiguracije. Pri prvom pokretanju maven će s Interneta u lokalni repozitorij skinuti programski kod traženog plugina i dalje će ga normalno koristiti.

Ovako pripremljeni projekt već možemo prevesti, testirati i zapakirati u JAR arhivu. Postoji samo još jedan detalj na koji treba pripaziti: kako ništa nismo eksplicitno podesili, maven će pri čitanju tekstovnih datoteka (uključivo i izvornih kodova programa) koristiti kodnu stranicu koja

je u tom trenutku podešena na računalu, čime dolazimo u opasnost da pišemo neprenosive skripte. Stoga ćemo još dodati nekoliko redaka u skriptu `pom.xml` koji su prikazani u nastavku.

Primjer 2.12. Definiranje kodne stranice i podataka o autoru u datoteci `pom.xml`

```
 1 <properties>
 2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 3 </properties>
 4
 5 <description>
 6   Ovo je testni projekt koji ilustrira uporabu mavena za
 7   upravljanje java projektom.
 8 </description>
 9
10 <organization>
11   <name>
12     ZEMRIS, Fakultet elektrotehnike i računarstva,
13     Sveučilište u Zagrebu
14   </name>
15   <url>http://www.fer.unizg.hr</url>
16 </organization>
17
18 <developers>
19   <developer>
20     <id>Marko.Cupic</id>
21     <name>Marko Čupić</name>
22     <email>marko.cupic@fer.hr</email>
23     <url>http://ferko.fer.hr/people/Marko.Cupic/</url>
24     <organization>
25       ZEMRIS, Fakultet elektrotehnike i računarstva,
26       Sveučilište u Zagrebu
27     </organization>
28     <organizationUrl>http://www.fer.unizg.hr</organizationUrl>
29     <roles>
30       <role>architect</role>
31       <role>developer</role>
32     </roles>
33     <timezone>+1</timezone>
34   </developer>
35 </developers>
36
37
```

Ovaj dio ubacite u `pom.xml` prije sekcije *dependencies*. Kroz postavljena svojstva podesili smo da se koristi kodna stranica UTF-8 a preostali dio navodi podatke o organizaciji i autoru projekta. Prilagodite te podatke tako da odgovaraju Vama.

Kako izgleda pretpostavljeni proces izgradnje projekta kao i kako izgleda pretpostavljena struktura projekta koje se treba pridržavati moguće je pronaći u dokumentaciji ovog alata. Uočite da je u okviru inicijalizacije projekta alat **maven** već izgradio i početni razred programa koji je smjestio u paket koji smo zadali i koji je smješten u datoteku `App.java`. Ovaj projekt već je sada moguće izgraditi, testirati, zatražiti generiranje dokumentacije i slično, a sve bez ikakvog daljnjeg konfiguriranja ako ste zadovoljni s pretpostavljenim verzijama. Primjerice, naredbom

```
D:\javaproj\projekt3m>mvn package
```

automatski ćete pokrenuti prevođenje koda, njegovo testiranje te generiranje JAR arhive -- isprobajte to.

U skladu s definiranim konvencijama, alat **maven** će izvršne verzije Java datoteka generirati u direktoriju `target/classes` dok će zatraženu JAR datoteku smjestiti direktno u direktorij `target`.

Za generiranje raznovrsnih izvještaja, na raspolaganju nam je plugin maven-site-plugin. Dodat ćemo u datoteku `pom.xml` u sekciju <build> ispred podsekcija: <plugins> novu podsekciju <pluginManagement> kako je prikazano u nastavku.

Primjer 2.13. Podešavanje verzije plugina za izvještaje u datoteci `pom.xml`

```
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.4</version>
        </plugin>
      </plugins>
    </pluginManagement>

```

Potom ćemo na kraj datoteke `pom.xml` (ali prije konačnog taga) dodati novu sekciju <reporting>, kako je prikazano u nastavku.

Primjer 2.14. Podešavanje izvještaja koje želimo dobiti u datoteci `pom.xml`

```
<reporting>
  <excludeDefaults>true</excludeDefaults>
  <outputDirectory>${project.build.directory}/site</outputDirectory>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.8</version>
      <configuration>
        <dependencyDetailsEnabled>false</dependencyDetailsEnabled>
        <dependencyLocationsEnabled>false</dependencyLocationsEnabled>
      </configuration>
      <reportSets>
        <reportSet>
          <reports>
            <report>dependencies</report>
            <report>summary</report>
            <report>project-team</report>
            <report>index</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.10.2</version>
      <configuration>
        <charset>UTF-8</charset>
        <docencoding>UTF-8</docencoding>
```

```
31          <encoding>UTF-8</encoding>
32        </configuration>
33      </plugin>
34      <plugin>
35        <groupId>org.apache.maven.plugins</groupId>
36        <artifactId>maven-jxr-plugin</artifactId>
37        <version>2.5</version>
38      </plugin>
39      <plugin>
40        <groupId>org.apache.maven.plugins</groupId>
41        <artifactId>maven-surefire-report-plugin</artifactId>
42        <version>2.18.1</version>
43      </plugin>
44    </plugins>
45  </reporting>
46
47
```

Proučite malo napisani dio konfiguracije. Vidite li gdje je definiran plugin za generiranje dokumentacije koda (javadoc) a gdje plugin za generiranje izvještaja nakon obavljenog testiranja koda (surefire-report)? Plugin *project-info-reports*[1] definira što se od općenite dokumentacije o projektu treba izgenerirati. Na sličan način moguće je dodati i generiranje izvještaja o kvaliteti koda alatima *CheckStyle*[2], *PMD*[3], *FindBugs*[4] kao i dobivanje izvještaja o pokrivenosti koda testovima alatom *Cobertura*[5].

Generiranje svih izvještaja pokrenut ćete naredbom:

```
mvn site
```

pri čemu će rezultat nastati u direktoriju `build/site` pa tamo pogledajte.

Uporaba alata gradle

Skinite i instalirajte alat Gradle. Malo detaljnija uputa dostupna je u dodatku B. Pozicionirajte se u vršni direktorij u kojem smo radili sve projekte (`D:\javaproj`). Stvorite potom direktorij `projekt3g` i iz projekta `projekt3` unutra prekopirajte direktorij `src`. Potom se pozicionirajte u vršni direktorij projekta (dakle u `projekt3g`). Stvorite konfiguracijsku datoteku `build.gradle` i u nju unesite sadržaj koji je prikazan u primjeru 2.15.

Primjer 2.15. Konfiguracijska datoteka `build.gradle` za alat gradle

```
1 apply plugin: 'java'
2
3 sourceCompatibility = 1.5
4 version = '1.0'
5 description = 'Build datoteka za prvi program.'
6
7 jar {
8  manifest {
9   attributes 'Main-Class': 'hr.fer.zemris.java.tecaj_1.Glavni'
10  }
11 }
12
```

[1] https://maven.apache.org/plugins/maven-project-info-reports-plugin/
[2] https://maven.apache.org/plugins/maven-checkstyle-plugin
[3] https://maven.apache.org/plugins/maven-pmd-plugin/
[4] https://gleclaire.github.io/findbugs-maven-plugin/index.html
[5] http://mojo.codehaus.org/cobertura-maven-plugin/

Alat **gradle** također očekuje uobičajenu strukturu projekta: izvorni kodovi programa bi trebali biti u direktoriju `src/main/java` a izvorni kodovi testova u direktoriju `src/test/java`. Generirani produkti bit će smješteni u direktorij `build`. Pri tome će izvršni Java kod od izvornih kodova projekta biti smješteni u poddirektorij `classes/main` dok će generirana JAR arhiva biti smještena u poddirektorij `libs`.

Kako biste pokrenuli izgradnju projekta, zadajte sljedeću naredbu.

```
D:\javaproj\projekt3g>gradle build
```

Pokrenut će se alat **gradle** koji će za nas obaviti cjelokupan posao prevođenja, testiranja i pakiranja izvršnih kodova u JAR arhivu. Struktura direktorija i datoteka koja je nastala pokretanjem prethodne naredbe prikazana je u nastavku.

```
D:\
  +-- javaproj
      +-- projekt3g
          +-- src
          |   + main
          |     +-- java
          |         +-- hr
          |             +-- fer
          |                 +-- zemris
          |                     +-- java
          |                         +-- tecaj_1
          |                             +-- App.java
          +-- build.gradle
          +-- build
              +-- classes
              |   +-- main
              +-- libs
              +-- ...
```

Detaljnija dokumentacija o ciljevima koji su u alatu **gradle** dostupni uporabom dodatka za Javu (čiju uporabu smo u danom primjeru skripte `build.gradle` definirali u retku 1) može se pronaći na adresi http://www.gradle.org/docs/current/userguide/tutorial_java_projects.html.

Poglavlje 3. Osiguravanje kvalitete i ispravnosti koda

Tijekom razvoja programa važno je pisati kvalitetan i ispravan kod. Pod pojmom kvalitete koda pri tome podrazumijevamo kod koji je lagano čitljiv, kod koji je jasan, kod koji kod čitatelja ne izaziva zbunjenost i kod kojeg je lagano pratiti što se događa. Kako pisati takav kod, ovisi o programskom jeziku koji koristimo odnosno o paradigmi koju možemo koristiti. Kako je ovo knjiga o programskom jeziku Java, koristimo objektno-orijentiranu paradigmu. Navedimo nekoliko općenitih savjeta kojih se dobro pridržavati.

Savjeti za pisanje kvalitetnog koda

- *Poštujte konvencije koje definiraju stil pisanja koda.*

  Povijesno gledano, svaki je programski jezik sa sobom donio neka uobičajena pravila koja govore kako treba formatirati izvorni kod. Tako, primjerice, tipična razlika između konvencija koje se koriste u programskom jeziku C te programskom jeziku Java je položaj vitičaste zagrade koja otvara tijelo neke naredbe. U programskom jeziku C ta se zagrada tipično prebacuje u novi red:

  ```
  if(dan<=20)
  {
    ...
  }
  ```

  dok se u Javi ta zagrada stavlja na kraj retka nakon jedne praznine:

  ```
  if(dan<=20) {
    ...
  }
  ```

  Imate li u uvjetu koji ispitujete više uvjeta, između logičkih operatora ćete ubaciti praznine ali oko operatora najčešće nećete.

  ```
  if(dan<=20 || dan>=31) {
    ...
  }
  ```

  Uobičajen način pisanja `for`-petlje (i uporabe razmaka) prikazan je u nastavku.

  ```
  for(int i=0; i<10; i++) {
    ...
  }
  ```

  Također, u `for` petlji se na kraju ne koristi prefiks operator uvećavanja već upravo prikazani postfiks operator, iako načelno oba na tom mjestu mogu stajati i obavljala bi isti posao.

  Pazite kako imenujete varijable, metode, razrede i druge konstrukte koje Vam nudi jezik. Danas kada više nemamo problema s memorijskim zahtjevima jezičnih procesora više nema potrebe koristiti strategije skraćivanja imena. Primjerice, jedna od često korištenih strategija je bila izbacivanje samoglasnika pa bi tako od varijable `salary` nastalo `slr` i slično. Danas doista nema potrebe zbunjivati sebe i druge koji bi nakon Vas mogli gledati kod koji ste napisali -- naziv `salary` potpuno je prihvatljiv i jasan.

  Koristite li imena koja su sastavljena od više riječi, u Javi je uobičajeno koristiti takozvani *camelcase*: riječi se međusobno slijepe i pišu malim slovima osim početnog slova svake riječi od druge na dalje. Primjerice, ispravno napisani naziv metode bi bio `calculateSalary`.

Nazivi razreda i sučelja tipično se pišu velikim početnim slovom. Nazivi metoda, članskih varijabli, lokalnih varijabli te argumenata se pišu malim početnim slovom. Nazivi konstanti se tipično u cijelosti pišu velikim slovima.

Upoznajte se s preporučenim stilom za programski jezik koji koristite. Detaljnu specifikaciju koja opisuje kako se uobičajeno formatira kod u Javi proučite na adresi [9.].

- *Varijable imenujte smisleno.*

  Izbjegavajte nazive varijabli poput `i`, `j`, `tmp`, `tmp7`, `vrijednost` i slično. Ovakva generička imena možete koristiti samo ako im je doseg uporabe izuzetno kratak i namjena sasvim jasna. Primjerice, sljedeći kod je korektan

```
if(a<b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

  jer definira varijablu `tmp` čiji je doseg ograničen doslovno na tri linije koda i sasvim je jasno koja je njezina uporaba. Varijable poput `i`, `j` i slično slobodno koristite u jednostavnim `for`-petljama ali samo ako one nisu ugnijeđene. Primjerice, sljedeći kod je korektan.

```
double suma = 0.0;
for(int i=0; i<bodovi.length; i++) {
  suma += bodovi[i];
}
```

  Sljedeći kod je međutim problematičan.

```
double ukupniGodisnjiIzdatci = 0.0;
for(int i=0; i<brojZaposlenika; i++) {
  for(int j=0; j<brojMjeseci; j++) {
    ukupniGodisnjiIzdatci += izdatci[i][j];
  }
}
```

  Pri uporabi generički definiranih varijabli indeksacije česta je pogreška da se kod višedimenzijskih polja zamijeni redoslijed indeksa što je problem koji je kasnije vrlo teško otkriti -- posebice ako su ograde na oba indeksa jednake. Ovaj problem je toliko raširen da je dobio i svoje ime: *index cross-talk*. Koji je problem s prethodnim kodom bit će jasno ako kažemo da polje `izdatci` za svaki mjesec sadrži izdatke za svakog zaposlenika. Ovo je semantika koju programer tipično ima na umu prilikom pisanja koda no ta mu semantika ne pomaže ako su varijable nazvane kao u primjeru. S druge strane, programer koji zna strukturu tog polja nikada ne bi napisao sljedeći kod:

```
double ukupniGodisnjiIzdatci = 0.0;
for(int zaposlenik=0; zaposlenik<brojZaposlenika; zaposlenik++) {
  for(int mjesec=0; mjesec<brojMjeseci; mjesec++) {
    ukupniGodisnjiIzdatci += izdatci[zaposlenik][mjesec];
  }
}
```

  jer bi mu odmah bilo jasno da to ne valja -- najprije treba indeksirati po mjesecu a potom po zaposleniku; takva se pogreška ne bi niti dogodila.

- *Varijable definirajte tamo gdje su potrebne.*

Za razliku od starih verzija programskog jezika C, u Javi nije potrebno (a niti je poželjno) sve što će se ikada koristiti u nekoj metodi definirati na samom početku metode. To stvara nepregledan kod i udaljava deklaraciju i inicijalizaciju varijable od mjesta njezine uporabe što kasnije unosi dodatni kognitivni napor prilikom čitanja i razumijevanja tog koda. Da biste to izbjegli, varijable deklarirajte na mjestu gdje su Vam potrebne i gdje ih prvi puta želite iskoristiti.

- *Nemojte mijenjati semantiku varijable.*

  Prilikom pisanja koda čovjek se često nađe u situaciji da mu na više mjesta treba neka pomoćna varijabla kako bi privremeno u nju pohranio neki međurezultat. Ako je taj međurezultat na više mjesta istog tipa, tada bi se moglo posegnuti za rješenjem koje uključuje deklariranje samo jedne takve varijable nekog generičkog imena i potom najprije pospremanje jednog a potom i drugog međurezultata u nju. Ovo je tehnika koja je izvorno razvijena dok su računala imala vrlo malo memorije i dok je svaki potrošeni oktet memorije činio razliku između dva programa. Problem s ovim rješenjem je što u nekoj proizvoljnoj liniji koda gdje ugledate tu varijablu više ne znate što je unutra zapisano -- umjesto toga, trebate tražiti kroz kod prema početku metode gdje je zadnja naredba koja je unutra nešto upisala i što je to bilo. To od svakoga tko čita takav kod zahtjeva ulaganje velikog kognitivnog napora (čak i na samog autora koda ako u takvom kodu upravo traži razlog zašto kod ne radi onako kako bi trebao) i nikako ne doprinosi njegovoj jasnoći. Nemojte to raditi. Kada Vam negdje zatreba međurezultat koji će čuvati sumu parnih brojeva, stvorite varijablu `sumaParnihBrojeva` i iskoristite je. Ako Vam kasnije zatreba pomoćna varijabla koja će privremeno čuvati sumu neparnih brojeva, stvorite novu varijablu `sumaNeparnihBrojeva` i iskoristite je.

- *Izbjegavajte magične brojeve!*

  Unos numeričkih konstanti u kod uvijek treba lokalizirati. Krenimo od jednostavnog primjera.

```
double[] polje = new double[5];
for(int i=0; i<5; i++) {
  polje[i] = stvoriVrijednost(i);
}
double suma = 0.0;
for(int i=0; i<5; i++) {
  suma += polje[i];
}
```

  Primjer je korektan i pridržava se svega što smo prethodno naveli. Ali ima jedan problem. Što u kodu treba mijenjati ako program umjesto nad 5 elemenata treba raditi nad 7 elemenata? Nepažljivi programer brzo bi rekao: “promijeni samo deklaraciju polja tako da umjesto 5 elemenata zauzme mjesta za 7 elemenata”. Marljivi programer bi rekao: “pazi, osim toga, treba još proći kroz kod i vidjeti jesno li još negdje unijeli broj 5 kao fiksnu ogradu i potom i ta mjesta promijeniti tako da se unese nova vrijednost (odnosno 7)”. Oprezni programer to bi dopunio opaskom: “pri tome treba još i provjeriti da li se svaki tako pronađeni broj 5 doista odnosi na ovo polje koje trenutno mijenjamo ili je to ograda nekog drugog polja koje je igrom slučaja također ograničeno s 5”. *Pametni* programer bi rekao: “ma tko je načičkao taj kod s hrpom magičnih brojeva? Za stvari koje su varijabilne granicu treba izolirati u jednu pomoćnu varijablu (ili u slučaju polja koristiti ugrađeno svojstvo polja `.length`) a za stvari koje su konstantne treba koristiti konstante!”. Taj bi programer prethodni kod prepisao na sljedeći način.

```
final int brojElemenata = 5;
double[] polje = new double[brojElemenata];
for(int i=0; i<polje.length; i++) {
  polje[i] = stvoriVrijednost(i);
}
double suma = 0.0;
```

```
for(int i=0; i<polje.length; i++) {
  suma += polje[i];
}
```

Ovakvom organizacijom koda promjene su lokalizirane na samo jedan redak izvornoga koda.

- *Ne pišite monsterske metode.*

  Kakav god algoritam da programirate, teško ćete naći opravdanje za metodu koja se proteže kroz 100 ili više redaka izvornog koda. Ako je to doista metoda koja obavlja puno složenog posla, tada je i u samoj metodi taj posao razložen u manje podzadatke. Izolirajte te podzadatke u nove metode. Svaka metoda bi trebala raditi samo jedan zadatak i biti kratka, jasna i konceptualno zatvorena jedinica. Oprezni programeri sada bi mogli reći: “ali pazi, poziv svake metode zahtjeva prijenos parametara preko stoga, spremanje povrate adrese i to bi sve skupa moglo usporiti program!”. Hm, da. Istina. Želite li brojati impulse signala takta koje potrošite, pišite u asembleru; tamo ćete imati potpunu kontrolu. Šalu na stranu, razmišljati o optimizaciji i slijediti neke generalne preporuke je u redu. Biti zaslijepljen optimizacijom svega neće Vas međutim daleko dovesti. Postoji jedna divna izreka: “prerana optimizacija majka je svih zala”. Jezični prevodioci koji se danas koriste su pametni. Nije rijetkost da se, ako se prevodioc uvjeri da je to moguće i korisno, kod koji ste Vi napisali kao zasebnu metodu ubaci direktno na mjesto poziva (postupak se zove *code inlining*). Pustite prevodioc da radi svoj posao a Vi pišite jasan kod. Jednom kada ste gotovi, iskoristite alate koji omogućavaju analizu performansi programa i koji će Vam ukazati koja su uska grla u Vašem kodu. Ako je kod prespor, tada se fokusirajte na rješavanje tih izoliranih dijelova koda (a i to vrlo često ne mora ili na račun čitljivosti i jasnoće koda već se može postići uporabom neke druge podatkovne strukture i slično).

- *Nemojte duplicirati kod.*

  Mnogi o sebi misle da su vrhunski programeri. Pa ako su nešto uspjeli napisati jednom bez greške, uspjet će to i drugi puta, i treći puta, i koliko god puta da treba u kodu koji pišu. Ovo je međutim loša praksa iz dva razloga. Prvo, ako se kasnije pokaže da je kod ipak imao pogrešku, trebat ćete pronaći sva druga mjesta na kojima ste problem rješavali na jednak način te i tamo provesti potrebnu korekciju; osim što to zna biti mukotrpno, često se neko mjesto zna i previdjeti. Drugi razlog leži u situaciji koja se može pojaviti naknadno (recimo, nekoliko mjeseci kasnije) i koja od Vas traži da promijenite način na koji ste radili odgovarajuću obradu. Ako ste kod duplicirali, opet ste pred istim zadatkom pretraživanja cjelokupnog izvornog koda i unošenja modifikacija na svim tim mjestima (uz pretpostavku da ste ih sve uspjeli i pronaći). Redundancija u kodu je zlo -- izbjegavajte je koliko god možete.

- *Pišite kod kod omogućava da se svaka promjena provede samo na jednom mjestu.*

  Ovo je vrlo važno pravilo. Ako provedba neke promjene funkcionalnosti zahtjeva mijenjanje koda na puno različitih mjesta, taj kod ima ozbiljnih problema i dosta je velika šansa da će te promjene biti teško provedive i da će postupak zahtjevati dosta posla oko korigiranja pogrešaka koje će nastati u postupku provođenja izmjena. Pišite kod koji je modularan i konceptualno dobro zatvoren. Pišite metode koje rade jednu dobro definiranu stvar. Nemojte pisati metode koje istovremeno kuhaju ručak, spremaju kavu i rezerviraju avionsku kartu za Havaje. Ako se pokaže da metoda nešto od toga radi loše, puno će teže biti otkriti pogrešku nego ako imate tri zasebne metode od kojih svaka obavlja svoj dio posla. Tada ćete puno lakše pronaći pogreške a i kasnije modificirati potrebno ponašanje ako to bude potrebno.

- *Komentirajte kod ali pišite i samodokumentirajući kod.*

  Pisanje komentara nužno je kako biste pomogli Vašim klijentima da jednostavnije koriste Vaš kod. Pisanje samodokumentirajućeg koda nužno je kako biste pomogli sebi i osobama koje će održavati i po potrebi modificirati Vaš kod. Pod pojmom *samodokumentirajući* kod pri tome se upravo podrazumjeva kod koji je lagan za čitanje i shvaćanje. Da bi kod bio takav, mora zadovoljavati sve što je navedeno u prethodnim točkama (pa i više od toga).

- *Koristite oblikovne obrasce.*

  Pisanje koda je stara disciplina (u smislu da to radimo već desetljećima). Kada rješavate neki problem, vjerojatno niste niti prva niti posljednja osoba koja rješava takav ili sličan problem ili se pak Vaš problem može riješiti tako da ga se dekomponira na probleme koje su drugi već riješili. Velika pomoć u tome Vam mogu biti oblikovni obrasci. Taj mnogima mističan pojam zapravo je vrlo jednostavan za shvatiti: oblikovni obrazac je opisano rješenje problema koje po mnogim kriterijima iskazuje svojstvo izvrsnosti (drugim riječima, radi se o kvalitetnom rješenju). Oblikovnih obrazaca ima mnogo. Svaki oblikovni obrazac najprije opisuje situaciju (tj. problem) koji pokušava riješiti, zatim definira implementaciju rješenja (često na konceptualnoj razini) i potom analizira prednosti i mane takvog rješenja; gleda se koji su preduvjeti da bi bi obrazac bio primjenjiv na problem i koje su posljedice njegove primjene. Nažalost, mnogi se prilikom upoznavanja s oblikovnim obrascima nekako uspiju fokusirati isključivo na centralni dio: kako se to implementira, i tada se zna dogoditi da pojedini oblikovni obrazac postane *čekić za svaki čavao*. Primjerice, ako se uhvatite u razmišljanju poput: “tu moram nekako javiti onom nešto -- dakle, uzimam obrazac Promatrač” (engl. *Observer pattern*), nešto je tu krivo. Lijepo da znate za taj obrazac, ali zašto baš njega? Što Vam treba u tom kodu? Koji problem rješavate? Je li taj obrazac uopće prikladan i jeste li spremni živjeti s cijenom koju on donosi sa sobom? To su pitanja koja su centralna prije no što se odlučite za ispravan obrazac; naravno, uz pretpostavku da znate koje su alternative. Ako je jedino što znate obrazac Promatrač, onda smo opet natrag na čekiću. A to nije dobro. Upoznajte se s oblikovnim obrascima. Upoznajte se s drugim tehnikama rješavanja problema. Upoznajte se s gotovim rješenjima i iskoristite ih -- nije istina da uvijek morate sve programirati od početka (“osim naravno na tečaju programskog jezika Java” -- opaska autora).

Savjeta za pisanje ispravnog koda nećemo navoditi u tolikoj mjeri kao što je to bio slučaj sa svjetima za pisanje kvalitetnog koda jer se u načelu do ispravnog koda može doći na samo dva načina (a i to nije u potpunosti istina):

- uz puno iskustva, discipline i testiranja te
- formalnom verifikacijom napisanog koda.

Iako su ove opcije napisane kao isključive, treba imati na umu što će čiji cilj. Zadaća programera bi trebala biti pisanje koda u kojem nema pogrešaka -- dakle, ispravnog koda. U tome mu može pomoć iskustvo ali i dobar alat za testiranje koda. Međutim, to od programera zahtjeva disciplinu. Nažalost, čest je slučaj da su ljudi sigurni u sebe i kod koji pišu pa zbog toga ne pišu testove.

Pisanje testova, ovisno kako se radi, može polučiti dva pozitivna rezultata. Ako se prilikom izgradnje programskog proizvoda pišu i testovi za svaki dio funkcionalnosti sustava, tada je kasnije olakšano modificiranje i nadogradnja funkcionalnosti sustava. S obzirom da već imate napisan čitav niz testova, lagano je provjeriti hoće li modifikacija koju ste napravili prouzročiti neispravan rad prethodno napisanih dijelova sustava što svakako nudi dobru sigurnosnu mrežu za rano hvatanje pogrešaka.

Drugi pozitivan rezultat je često kvalitetno oblikovani kod koji ima čitav niz primjera uporabe. Ovaj rezultat čest je ako se primjenjuje paradigma poznata pod nazivom testovima-vođeni-razvoj (engl. *Test Driven Development*, TDD). Programeri koji koriste TDD, razvoj programa rade iterativno kroz male korake. Najprije se napiše test koji provjerava funkcionalnost koju treba razviti nakon čega se piše implementacija te funkcionalnosti. Posljedica je da programer, prije no što krene kodirati rješenje mora razmisliti o tome kako ga želi koristiti -- to je nužno jer se to zahtjeva kako bi se uopće napisao test. A jednom kada je napisao test, programer je zapravo ostavio i koncizan primjer uporabe te funkcionalnosti i to napisan direktno u programskom jeziku Java.

Konačno, uporaba TDD-a često zna obuzdati pisanje funkcionalnosti koja je inače nepotrebna a koje se tipično pojavljuje kada se razvoj okrene naglavačke i kada se krene primjerice ovako: “trebamo jedan razred koji se ponaša kao struktura stog; što bi sve hipotetski jedan

takav stog mogao nuditi potencijalnim klijentima?”. Ovakvo razmišljanje je u redu ako razvijate općenitu biblioteku koja će biti korištena u mnoštvu drugih programa. Tada želite ponuditi svu moguću funkcionalnost koju bi Vaši klijenti mogli zatrebati kako taj dio koda ne bi morali pisati sami klijenti (i to svatko svoju kopiju funkcionalnosti koja nedostaje). Međutim, ako rješavate u Vašem vlastitom programu neki problem u kojem Vam trebaju samo dvije operacije (gurni na stog i skini sa stoga) i u kojem nemate pristup nekoj gotovoj implemetaciji stoga, ima li smisla uopće razmišljati o drugim operacijama i raditi na njihovoj implementaciji? Problem je što sve što implementirate može imati pogreške -- što više nepotrebne funkcionalnosti, više nepotrebnih pogrešaka. Što ako Vam kasnije u okviru dodavanja sljedeće funkcionalnosti zatreba stog koji ima još jednu funkcionalnost koja Vam treba? U tom slučaju ćete tu funkcionalnost dodati i to na način da ćete napisati jedan ili više testova koji provjeravaju njezinu ispravnost. Pri tome možete biti razmjerno sigurni da dodavanje te nove funkcionalnosti neće potrgati postojeći kod jer za onaj preostali dio funkcionalnosti već imate napisane testove od prije i ako tu bude problema, ti će testovi to otkriti. S druge pak strane, ako Vam kasnije takva funkcionalnost ne zatreba, uštedili ste vrijeme (i stabilnost programa) jer niste radili implementaciju nečega što Vam ne treba (i to još dodatno debugirali).

Prilikom uporabe testiranja kao sredstva za osiguranje ispravnosti koda treba biti svjestan još jednog ograničenja ovog pristupa: testiranje može dokazati postojanje pogreške (tako da je izazove) ali ne može dokazati da pogrešaka nema. Problem sa testiranjem je da test provjerava samo onaj aspekt koda koji ste pokrili tim testom. Aspetke koje niste pokrili testom niste niti testirali i shodno tome, ne znate ima li u tom slučaju pogreške ili nema.

Zadaća formalne verifikacije koda dijametralno je suprotna: formalnom verifikacijom ne pomažemo si da napišemo ispravan kod, već jednom kada imamo napisan kod možemo formalno provjeriti odnosno dokazati nepostojanje pogrešaka. Nažalost, zbog kompleksnosti ove problematike danas još nemamo uporabljivih alata kojima bismo mogli formalno dokazivati korektnost nekog proizvoljno složenog programskog produkta u prihvatljivom vremenu (ili uopće).

U nastavku ovog poglavlja pozabavit ćemo se stoga nizom alata koji će nam pomoći da pišemo kvalitetniji i što je moguće ispravniji kod. Pogledat ćemo alate koji će nam omogućiti da provjerimo kvalitetu napisanog koda (poštivanje zadanih konvencija, postoji li dupliciranog koda, jesu li komentari prihvatljivi), da provjerimo jesmo li gdje u kodu napisali nešto što tipično vodi prema pogreškama, da napišemo testove za naš kod te da provjerimo koliki smo dio koda pokrili s testovima.

Alat CheckStyle

Skinite i instalirajte alat CheckStyle. Malo detaljnija uputa dostupna je u dodatku C. Potom prekopirajte direktorij `projekt3` u direktorij `projekt4` i pozicionirajte se u njega. Otvorite datoteku `build.xml` i promijenite sve spomene projekta 3 u projekt 4 (dva su mjesta: naziv projekta i opis projekta). Potom napravite još i korekcije koje su opisane u nastavku.

- U deklaraciju projekta dodana je deklaracija prostora imena cs definiranjem atributa `xmlns:cs="antlib:com.puppycrawl.tools.checkstyle"`.

- Dodajte deklaracije dviju globalnih varijabli: `checkstyle.home` i `xalan.home`. Ako ste ove alate instalirali prema uputama koje su dostupne u dodatku C, retci koje ćete dodati trebali bi izgledati kako je prikazano u nastavku.

  ```
  <property name="checkstyle.home" value="d:/usr/checkstyle-5.6" />
  <property name="xalan.home" value="d:/usr/xalan-j_2_7_1" />
  ```

- Naredbi **ant** potrebno je objasniti da na raspolaganju ima novu naredbu koju kasnije želite koristiti. To se radi na način da se iskoristi naredba za definiranje novih naredbi (**taskdef**) koja ima više oblika pozivanja. Primjer ove naredbe prikazan je u nastavku.

  ```
  <taskdef uri="antlib:com.puppycrawl.tools.checkstyle"
  ```

```
  resource="checkstyletask.properties"
  classpath="${checkstyle.home}/checkstyle-5.6-all.jar"/>
```

Uočite da je kao posljednji atribut navedena putanja do JAR arhive koja sadrži implementaciju ove biblioteke i koja je napisana na način koji će alatu **ant** omogućiti da učita i koristi njezinu funkcionalnost. Ovime će nam postati dostupna naredba `<cs:checkstyle>` kojom ćemo moći pokrenuti analizu kvalitete koda.

Navedenu naredbu dodajte u `build.xml` pri vrhu datoteke, nakon deklaracija svih varijabli ali prije deklaracije cilja `init`.

- Konačno, dodajte novi cilj koji poziva prethodno dodanu naredbu. Isječak XML-a koji to radi prikazan je u nastavku. Ovaj kod možete dodatni kao posljednji cilj u datoteci. Uočite da smo kao ovisnost cilja definirali cilj `compile` tako da se provjere uopće ne pokreću ako je kod takav da ga se čak niti ne da prevesti.

```
<target name="cs" depends="compile">
 <mkdir dir="${dist}/checkstyle/xml"/>
 <mkdir dir="${dist}/checkstyle/html"/>

 <cs:checkstyle config="${checkstyle.home}/sun_checks_2.xml"
         failOnViolation="false">
  <fileset dir="${src.java}" includes="**/*.java"/>
  <formatter type="xml"
     toFile="${dist}/checkstyle/xml/checkstyle_errors.xml"/>
 </cs:checkstyle>

 <xslt basedir="${dist}/checkstyle/xml"
       destdir="${dist}/checkstyle/html" extension=".html"
       style="${checkstyle.home}/contrib/checkstyle-frames.xsl">
  <classpath>
   <fileset dir="${xalan.home}">
    <include name="*.jar"/>
   </fileset>
  </classpath>
 </xslt>
</target>
```

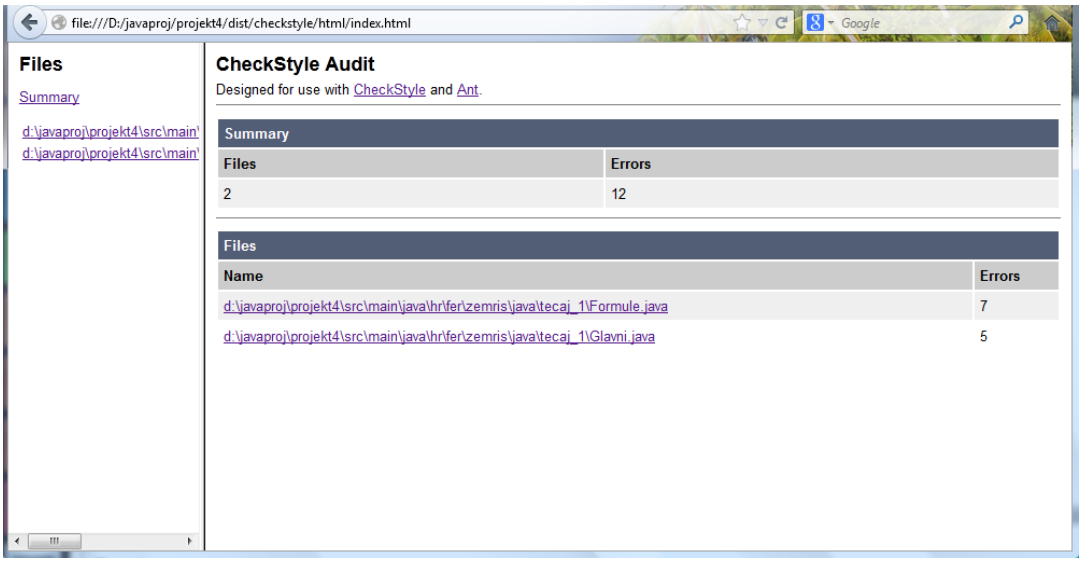
Naredba `<cs:checkstyle>` pokreće analizu koda i generira izvještaj koji se u formatu XML sprema u datoteku `checkstyle_errors.xml`. Potom se poziva naredba `<xslt>` koja koristeći procesor Xalan kao ulaz čita prethodno stvorenu XML datoteku `checkstyle_errors.xml` i temeljem njezinog sadržaja i pravila koja su zapisana u datoteci `checkstyle-frames.xsl` stvara HTML dokument koji podatke iz XML datoteke formatira uporabom HTML specifikacije kako bi poslali lakše čitljivi za čovjeka.

Nakon ovih modifikacija pokrenite cilj `cs`.

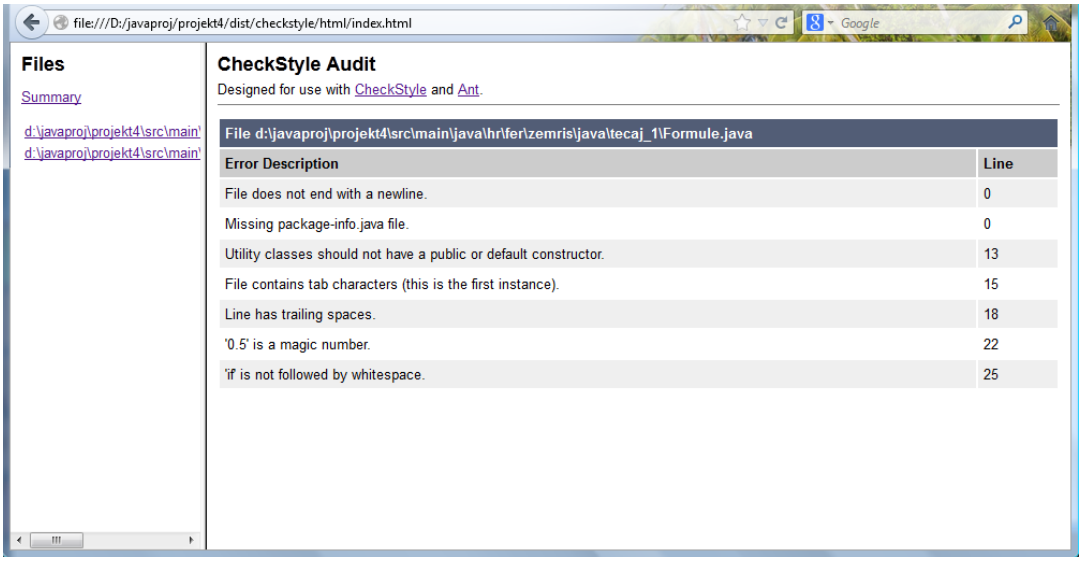
```
D:\javaproj\projekt4>ant cs
```

Kao rezultat ove akcije u direktoriju `dist/checkstyle/html` nastat će kompletan izvještaj za sve izvorne datoteke u formatu HTML. Izvorni izvještaj u formatu XML bit će spremljen u direktoriju `dist/checkstyle/xml` tako da je temljem njega lagano generirati i izvještaje u drugim formatima. Otvorite datoteku `index.html` -- to će bit sumarni izvještaj koji bi trebao odgovarati izvještaju prikazanom na slici 3.1. Odaberite prikaz detaljnog izvještaja za datoteku `Formule.java`. Trebali biste dobiti prikaz koji je sličan izvještaju prikazanom na slici 3.2.

Slika 3.1. Sažetak izvještaja o kontroli stila pisanja koda



Slika 3.2. Izvještaja o kontroli stila pisanja koda datoteke `Formule.java`



Alat PMD

Skinite i instalirajte alat PMD. Malo detaljnija uputa dostupna je u dodatku C. Potom prekopirajte direktorij `projekt3` u direktorij `projekt5` i pozicionirajte se u njega. Otvorite datoteku `build.xml` i promijenite sve spomene projekta 3 u projekt 5 (dva su mjesta: naziv projekta i opis projekta). Potom napravite još i korekcije koje su opisane u nastavku.

- Dodajte deklaracije dviju globalnih varijabli: `pmd.home` i `xalan.home`. Ako ste ove alate instalirali prema uputama koje su dostupne u dodatku C, retci koje ćete dodati trebali bi izgledati kako je prikazano u nastavku.

```
<property name="pmd.home" value="d:/usr/pmd-bin-5.0.2" />
<property name="xalan.home" value="d:/usr/xalan-j_2_7_1" />
```

- Naredbi **ant** potrebno je objasniti da na raspolaganju ima nove naredbe koje kasnije želimo koristiti. Za slučaj alata PMD to je najjednostavnije napraviti tako da se najprije definira po-

moćna staza u koju ćemo zatražiti od alata **ant** da doda sve JAR arhive alata **PMD** nakon čega definiramo dvije naredbe koje obje koriste prethodno definiranu stazu.

```
<!-- Definiranje staze do PMD biblioteka: -->
<path id="pmd.lib" >
 <fileset dir="${pmd.home}/lib">
  <include name="*.jar"/>
 </fileset>
</path>

<!-- Definiranje naredbi koje pokreću PMD analize -->
<taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"
  classpathref="pmd.lib" />
<taskdef name="cpd" classname="net.sourceforge.pmd.cpd.CPDTask"
  classpathref="pmd.lib" />
```

Ovime će nam postati dostupne naredbe `<pmd>` te `<cp>` kojima ćemo moći pokrenuti analizu kvalitete koda.

Navedene naredbe dodajte u `build.xml` pri vrhu datoteke, nakon deklaracija svih varijabli ali prije deklaracije cilja `init`.

- Konačno, dodajte novi cilj koji poziva prethodno dodane naredbe. Isječak XML-a koji to radi prikazan je u nastavku. Ovaj kod možete dodatni kao posljednji cilj u datoteci. Uočite da smo kao ovisnost cilja definirali cilj `compile` tako da se provjere uopće ne pokreću ako je kod takav da ga se čak niti ne da prevesti.

```
<target name="pmd" depends="compile">

 <mkdir dir="${dist}/pmd/xml"/>
 <mkdir dir="${dist}/pmd/html"/>

 <pmd shortFilenames="true" encoding="UTF-8">
  <ruleset>rulesets/java/basic.xml</ruleset>
  <ruleset>rulesets/java/braces.xml</ruleset>
  <ruleset>rulesets/java/codesize.xml</ruleset>
  <ruleset>rulesets/java/controversial.xml</ruleset>
  <ruleset>rulesets/java/design.xml</ruleset>
  <ruleset>rulesets/java/finalizers.xml</ruleset>
  <ruleset>rulesets/java/imports.xml</ruleset>
  <ruleset>rulesets/java/naming.xml</ruleset>
  <ruleset>rulesets/java/optimizations.xml</ruleset>
  <ruleset>rulesets/java/strictexception.xml</ruleset>
  <ruleset>rulesets/java/strings.xml</ruleset>
  <ruleset>rulesets/java/sunsecure.xml</ruleset>
  <ruleset>rulesets/java/typeresolution.xml</ruleset>
  <ruleset>rulesets/java/unusedcode.xml</ruleset>
  <ruleset>rulesets/java/unnecessary.xml</ruleset>
  <formatter type="xml" toFile="${dist}/pmd/xml/pmd_report.xml"/>
  <fileset dir="${src.java}">
   <include name="**/*.java"/>
  </fileset>
 </pmd>

 <cpd minimumTokenCount="10"
      outputFile="${dist}/pmd/xml/cpd_report.xml"
      format="xml" encoding="UTF-8">
  <fileset dir="src">
   <include name="**/*.java"/>
```

```
    </fileset>
  </cpd>

  <xslt includes="cpd_report.xml" basedir="${dist}/pmd/xml"
        destdir="${dist}/pmd/html" extension=".html"
        style="${pmd.home}/etc/xslt/cpdhtml.xslt" >
   <classpath>
    <fileset dir="${xalan.home}">
     <include name="*.jar"/>
    </fileset>
   </classpath>
  </xslt>

  <xslt includes="pmd_report.xml" basedir="${dist}/pmd/xml"
        destdir="${dist}/pmd/html" extension=".html"
        style="${pmd.home}/etc/xslt/wz-pmd-report.xslt" >
   <classpath>
    <fileset dir="${xalan.home}">
     <include name="*.jar"/>
    </fileset>
   </classpath>
  </xslt>

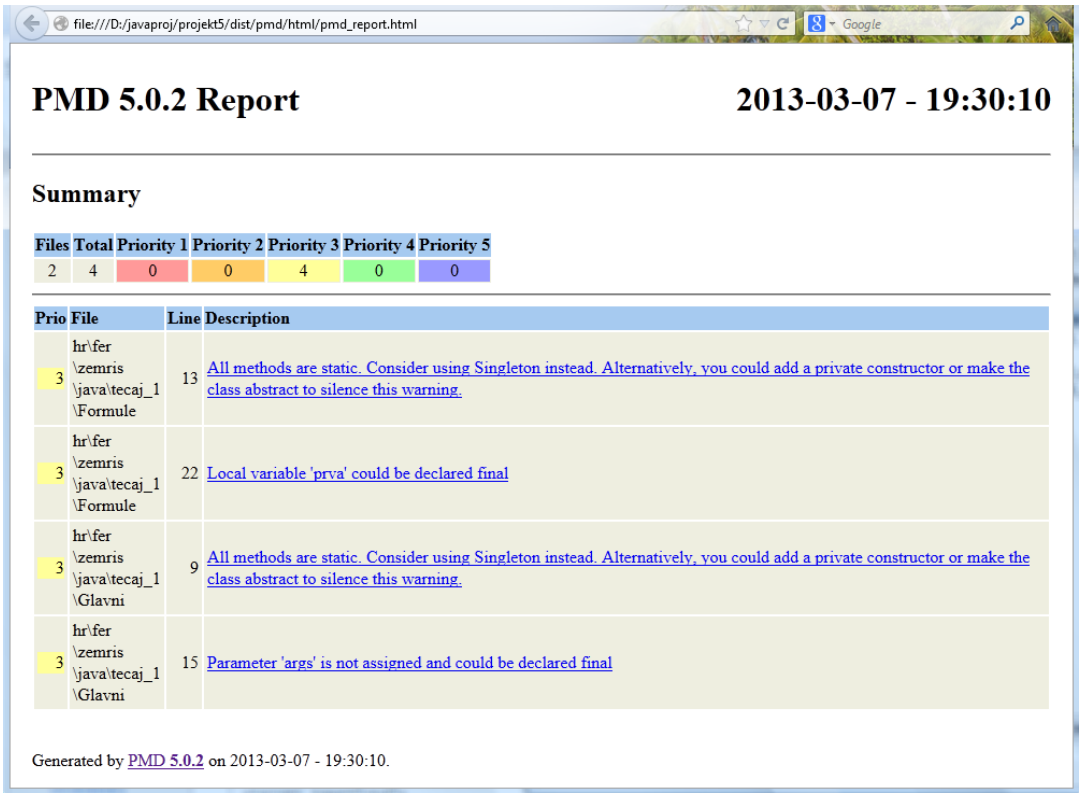
 </target>
```

Nakon ovih modifikacija pokrenite cilj `pmd`.

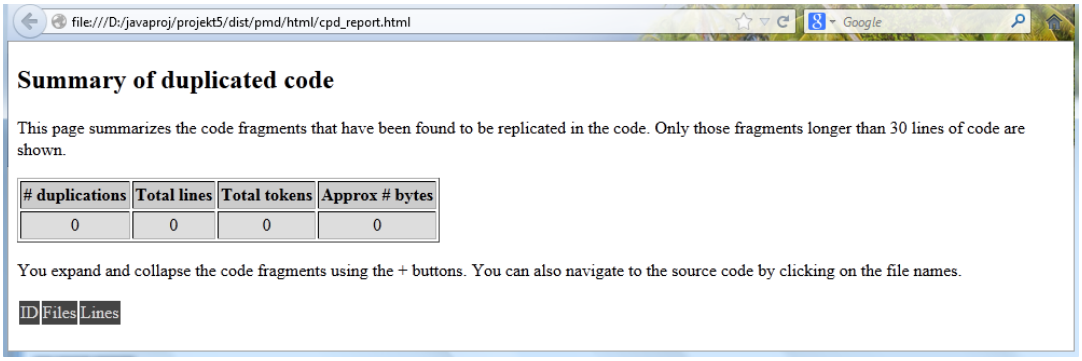
```
D:\javaproj\projekt5>ant pmd
```

Kao rezultat ove akcije u direktoriju `dist/pmd/html` nastat će dva izvještaja za sve izvorne datoteke u formatu HTML. Izvorni izvještaji u formatu XML bit će spremljeni u direktoriju `dist/pmd/xml` tako da je temljem njega lagano generirati i izvještaje u drugim formatima. Otvorite datoteku `pmd_report.html` -- to će bit sumarni izvještaj koji bi trebao odgovarati izvještaju prikazanom na slici 3.3. Proučite izvještaj. Potom otvorite datoteku `cpd_report.html` -- to će bit sumarni izvještaj koji bi trebao odgovarati izvještaju prikazanom na slici 3.4.

Slika 3.3. Sažetak izvještaja o kontroli stila pisanja koda



Slika 3.4. Sažetak izvještaja o dupliciranju koda



Alat FindBugs

Skinite i instalirajte alat FindBugs. Malo detaljnija uputa dostupna je u dodatku C. Potom prekopirajte direktorij `projekt3` u direktorij `projekt6` i pozicionirajte se u njega. Otvorite datoteku `build.xml` i promijenite sve spomene projekta 3 u projekt 6 (dva su mjesta: naziv projekta i opis projekta). Potom napravite još i korekcije koje su opisane u nastavku.

- Dodajte deklaracije dviju globalnih varijabli: `findbugs.home` i `xalan.home`. Ako ste ove alate instalirali prema uputama koje su dostupne u dodatku C, retci koje ćete dodati trebali bi izgledati kako je prikazano u nastavku.

```
<property name="findbugs.home" value="d:/usr/findbugs-2.0.2" />
<property name="xalan.home" value="d:/usr/xalan-j_2_7_1" />
```

- Naredbi **ant** potrebno je objasniti da na raspolaganju ima novu naredbu koju kasnije želimo koristiti. Za slučaj alata FindBugs to je moguće napraviti na način prikazan u nastavku.

```
<taskdef name="findbugs"
         classname="edu.umd.cs.findbugs.anttask.FindBugsTask">
 <classpath path="${findbugs.home}/lib/findbugs-ant.jar"/>
</taskdef>
```

Ovime će nam postati dostupna naredba `<findbugs>` kojom ćemo moći pokrenuti analizu kvalitete ali i djelomične ispravnosti koda.

Navedenu naredbu dodajte u `build.xml` pri vrhu datoteke, nakon deklaracija svih varijabli ali prije deklaracije cilja `init`.

- Konačno, dodajte novi cilj koji poziva prethodno dodane naredbe. Isječak XML-a koji to radi prikazan je u nastavku. Ovaj kod možete dodatni kao posljednji cilj u datoteci. Uočite da smo kao ovisnost cilja definirali cilj `compile` tako da se provjere uopće ne pokreću ako je kod takav da ga se čak niti ne da prevesti.

```
<target name="findbugs" depends="compile">
 <mkdir dir="${dist}/findbugs/xml"/>
 <mkdir dir="${dist}/findbugs/html"/>

 <findbugs home="${findbugs.home}"
    output="xml:withMessages"
    outputFile="${dist}/findbugs/xml/report.xml"
    workHard="true" effort="max"
    projectName="${ant.project.name}">
  <sourcePath path="${src.java}" />
  <class location="${build.classes}" />
 </findbugs>

 <xslt includes="report.xml" basedir="${dist}/findbugs/xml"
       destdir="${dist}/findbugs/html" extension=".html"
       style="${findbugs.home}/src/xsl/fancy.xsl" >
  <classpath>
   <fileset dir="${xalan.home}">
    <include name="*.jar"/>
   </fileset>
  </classpath>
 </xslt>
</target>
```

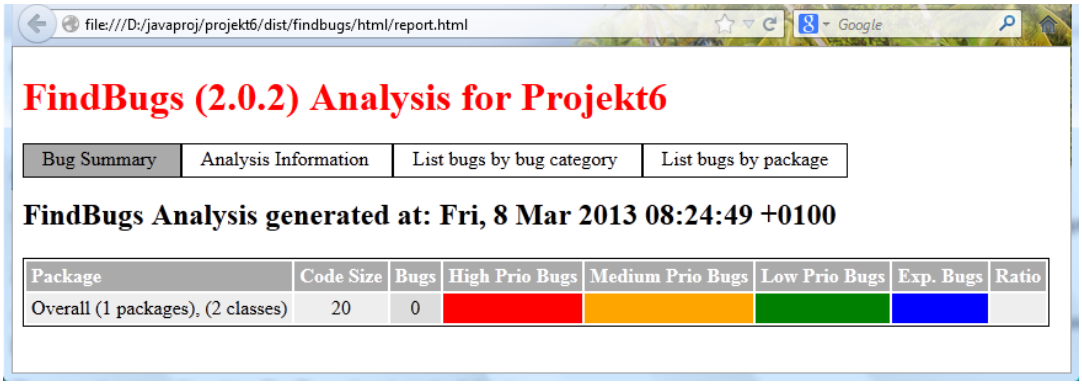
Stil koji se u ovom primjeru koristi za prevođenje izvještaja u XML formatu u izvještaj u HTML formatu je *`${findbugs.home}`*`/src/xsl/fancy.xsl`. U istom direktoriju nalazi se još nekoliko stilova pa možete isprobati i njih.

Nakon ovih modifikacija pokrenite cilj `findbugs`.

```
D:\javaproj\projekt6>ant findbugs
```

Kao rezultat ove akcije u direktoriju `dist/findbugs/html` nastat će izvještaj za sve izvorne datoteke u formatu HTML. Izvorni izvještaj u formatu XML bit će spremljen u direktoriju `dist/findbugs/xml` tako da je temljem njega lagano generirati i izvještaje u drugim formatima. Otvorite datoteku `report.html` -- to će bit sumarni izvještaj koji bi trebao odgovarati izvještaju prikazanom na slici 3.5.

Slika 3.5. Sažetak izvještaja o kontroli kvalitete koda



Alat JUnit

Zadaća alata JUnit jest automatizirati testiranje koda. Prisjetimo se, testiranjem nije moguće dokazati da pogreške ne postoje ali je svakako moguće izgraditi sigurnosnu mrežu koja će nam pomoći da uhvatimo veliki broj pogrešaka te da kasnije lakše modificiramo postojeći kod.

Skinite alat JUnit. Malo detaljnija uputa dostupna je u dodatku C pa postupite u skladu s tom uputom.

Da bismo ilustrirali kako se radi s ovim alatom, najprije moramo pripremiti novi projekt koji će nam poslužiti kao projekt za razvoj jednostavne biblioteke koja će se sastojati od samo jednog razreda. Zadaća biblioteke je ponuditi funkcionalnost rada s prirodnim brojevima.

Napravite stoga direktorij `brojevi` i unutar njega strukturu poddirektorija kako je prikazano u nastavku.

```
D:\
  +-- javaproj
        +-- brojevi
              +-- src
                    +-- main
                    |     +-- java
                    |           +-- ..
                    +-- test
                          +-- java
                                +-- ..
```

U direktorij `src/main/java` ubacite u paket hr.fer.zemris.java.tecaj_1.brojevi datoteku naziva `Prirodni.java` čiji je izvorni kod prikazan u primjeru 3.1. Sjetite se, ubaciti u paket znači i na disku napraviti odgovarajuću strukturu direktorija koja odgovara strukturi paketa.

Primjer 3.1. Implementacija biblioteke za rad s prirodnim brojevima

```
 1 package hr.fer.zemris.java.tecaj_1.brojevi;
 2
 3 /**
 4  * Ovaj razred sadrži razne funkcije za rad s prirodnim
 5  * brojevima. Trenutno ne postoji konsenzus što su točno
 6  * prirodni brojevi -- jesu li to to pozitivni cijeli
 7  * brojevi ili su to nenegativni cijeli brojevi; razlika
 8  * izmedu ove dvije definicije je u broju nula. Više detalja
 9  * može se pogledati na, primjerice,
10  * <a href="http://mathworld.wolfram.com/NaturalNumber.html">
11  * ovoj adresi</a>.
12  * U okviru ovog razreda prirodnim brojevima će se smatrati
```

```
 * pozitivni cijeli brojevi (dakle, nula se neće tretirati kao
 * pozitivan cijeli broj).
 *
 * <p>Evo i jednostavnog primjera uporabe. Da biste postavili
 * vrijednost zastavice {@code prirodni} na {@code true} ako
 * je predani broj {@code n} prirodni, možete se poslužiti
 * sljedećim isječkom koda.</p>
 * <pre>
 *   int n = 153;
 *   boolean prirodni = Prirodni.jePrirodni(n);
 *   System.out.println("Broj " + n + " je prirodni? "
 *                              + prirodni);
 * </pre>
 *
 * @since 1.0
 * @author Marko Čupić
 */
public class Prirodni {

 /**
  * Funkcija ispituje je li predani broj prirodni broj.
  * @param n cijeli broj koji se ispituje
  * @return <code>true</code> ako je predani cijeli broj
  *         ujedno i prirodni broj; inače vraća
  *         <code>false</code>.
  */
 public static boolean jePrirodni(int n) {
  return n > 0;
 }

 /**
  * Funkcija vraća sljedbenika od predanog broja.
  * @param n broj čijeg sljedbenika treba vratiti
  * @return <code>true</code> ako je predani cijeli broj
  *         ujedno i prirodni broj; inače vraća
  *         <code>false</code>.
  */
 public static int sljedbenik(int n) {
  return n+1;
 }

}
```

Kako je ovo biblioteka, nećemo raditi razred koji bi imao metodu `main` već je ideja da se ovaj kod prevede u izvršnu verziju i potom koristi u okviru drugih projekata.

Ovaj ćemo kod, međutim, željeti istestirati. Zbog toga smo i napravili zasebnu granu u `src` direktoriju -- granu `src/test/java`. Ta grana će nam biti vršna grana za pohranu testova. Za razred `Prirodni.java` napravit ćemo razred koji ga testira i čije će ime, u skladu s često korištenim konvencijama biti jednako nazivu razreda na koji je nadodan nastavak `Test`: dakle `PrirodniTest`. Taj ćemo razred smjestiti u isti paket kao i razred koji je testiran. Napravite stoga u direktoriju `src/test/java` strukturu poddirektorija koja odgovara paketu hr.fer.zemris.java.tecaj_1.brojevi i potom unutra dodajte datoteku `PrirodniTest.java` čiji je sadržaj prikazan u primjeru 3.2.

Primjer 3.2. Testovi za razred `Prirodni`.

```
1 package hr.fer.zemris.java.tecaj_1.brojevi;
2
```

```
 3 import org.junit.Test;
 4 import org.junit.Assert;
 5
 6 public class PrirodniTest {
 7
 8     @Test
 9     public void negativanBrojNijePrirodan() {
10         Assert.assertFalse(Prirodni.jePrirodni(-1));
11     }
12
13     @Test
14     public void pozitivniBrojJePrirodan() {
15         Assert.assertTrue(Prirodni.jePrirodni(4));
16     }
17
18     @Test
19     public void nulaNijePrirodniBroj() {
20         Assert.assertFalse(Prirodni.jePrirodni(0));
21     }
22
23 }
```

Razred prikazan u primjeru 3.2 primjer je razreda koji sadrži tri testa. Sa stajališta alata JUnit, test je svaka metoda koja je anotirana s `@Test`. U danom primjeru, ta se anotacija nalazi iznad tri metode, u retcima 8, 13 i 18. Metoda koja je tako anotirana treba imati signaturu `public void` što znači da ne smije ništa vraćati. Također, metoda ništa niti ne prima odnosno nema ulaznih argumenata. Ime metode trebalo bi odgovarati svrhi testa -- što taj test provjerava. Svaki test se potom piše na način da se pripremi sve potrebno, i potom se jednom ili više `assert` naredbi definira što mora vrijediti. Tako primjerice, test `negativanBrojNijePrirodan` tvrdi da rezultat poziva metode `Prirodni.jePrirodni(-1)` mora vratiti `false`. Ako svi uvjeti koje test tako postavi budu zadovoljeni, kažemo da je test "prošao" (engl. *passed*); u suprotnom kažemo da je test "pao" (engl. *failed*).

Prilikom pisanja testova, korisniku na raspolaganju stoji čitav niz različitih metoda `assert`, poput `assertTrue`, `assertFalse`, `assertEquals`, `assertArrayEquals`, `assertNull`, `assertNotNull`, `assertSame`, `assertNotSame` i slične. Također, ako se želi definirati složeni uvjet za rušenje testa koji nije moguće jednostavno postići prethodnim metodama, korisniku na raspolaganju stoji i direktno metoda `fail()` kojom može srušiti test.

Anotacija `@Test` omogućava i definiranje uvjeta prolaznosti testa zahtjevom da test koji se testira mora izazvati određenu iznimku. Primjerice, ako imamo test koji zahtjeva da kod koji se testira izazove iznimku `IndexOutOfBoundsException` kako bi prošao (a pada ako je ne izazove), tada se anotacija `@Test` koristi uz atribut `expected` kako je prikazano u nastavku.

```
@Test(expected=IndexOutOfBoundsException.class)
public void dohvatElementaKojiNepostoji() {
  // ... kod koji pokušava dohvatiti element s nepostojeće pozicije
}
```

Uočite da je za rad s navedenom anotacijom te razredom `Assert` nužno dodati dvije direktive `import` (retci 3 i 4) kojima ih uključujemo u izvorni kod testa.

Jednom kada smo napisali testove, potrebno je osigurati da ih možemo automatski pokretati. To je najjednostavnije napraviti uporabom odgovarajućeg cilja alata **ant**. Stoga u vršni direktorij dodajte datoteku `build.xml` čiji je sadržaj prikazan u nastavku.

Primjer 3.3. Datoteka `build.xml` za pokretanje testova.

```
1 <project name="Brojevi" default="jar" basedir=".">
2
```

```
 3  <description>
 4   Build datoteka za projekt Brojevi.
 5  </description>
 6
 7  <!-- Postavljanje globalnih varijabli -->
 8  <property name="src" location="src"/>
 9  <property name="src.java" location="${src}/main/java"/>
10  <property name="src.test" location="${src}/test/java"/>
11  <property name="build" location="build"/>
12  <property name="build.classes" location="${build}/classes"/>
13  <property name="build.test" location="${build}/test"/>
14  <property name="dist"  location="dist"/>
15
16  <!-- Postavljanje varijabli za alate -->
17  <property name="junit.home" value="d:/usr/junit-4.11" />
18
19  <!-- Definiranje staze za prevođenje koda -->
20  <path id="compile.path">
21   <pathelement location="${build.classes}"/>
22  </path>
23
24  <!-- Definiranje staze za prevođenje testova -->
25  <path id="test.path">
26   <path refid="compile.path"/>
27   <pathelement location="${build.test}"/>
28   <fileset dir="${junit.home}">
29    <include name="**/*.jar"/>
30   </fileset>
31  </path>
32
33  <target name="init">
34   <!-- Stvaranje vremenske oznake -->
35   <tstamp/>
36   <!-- Stvaranje potrebnih direktorija -->
37   <mkdir dir="${build}"/>
38   <mkdir dir="${dist}"/>
39  </target>
40
41  <target name="compile" depends="init"
42   description="Prevođenje izvornog koda">
43   <mkdir dir="${build.classes}"/>
44   <!-- Prevođenje Java koda iz ${src} u ${build} -->
45   <javac srcdir="${src.java}" destdir="${build.classes}"
46    classpathref="compile.path"
47    encoding="UTF-8" debug="on"
48    debuglevel="lines,vars,source"
49    includeAntRuntime="false" />
50  </target>
51
52  <target name="compile-tests" depends="compile"
53   description="Prevođenje izvornog koda testova">
54   <mkdir dir="${build.test}"/>
55   <!-- Prevođenje Java koda iz ${src} u ${build} -->
56   <javac srcdir="${src.test}" destdir="${build.test}"
57    classpathref="test.path"
58    encoding="UTF-8" debug="on"
59    debuglevel="lines,vars,source"
60    includeAntRuntime="false" />
```

```
61  </target>
62
63  <target name="run-tests" depends="compile-tests"
64   description="Izvođenje definiranih testova" >
65   <mkdir dir="${dist}/test-reports/xml"/>
66   <mkdir dir="${dist}/test-reports/html"/>
67
68   <junit printsummary="yes" haltonfailure="yes">
69    <classpath refid="test.path" />
70
71    <formatter type="plain"/>
72    <formatter type="xml"/>
73
74    <batchtest fork="yes" todir="${dist}/test-reports/xml">
75     <fileset dir="${src.test}">
76      <include name="**/*Test*.java"/>
77     </fileset>
78    </batchtest>
79   </junit>
80   <junitreport todir="${dist}/test-reports/xml">
81    <fileset dir="${dist}/test-reports/xml">
82     <include name="TEST-*.xml"/>
83    </fileset>
84    <report format="frames" todir="${dist}/test-reports/html"/>
85   </junitreport>
86  </target>
87
88  <target name="clean"
89   description="Brisanje generiranog sadržaja" >
90   <!-- Obriši direktorije ${build} i ${dist} -->
91   <delete dir="${build}" failonerror="false" />
92   <delete dir="${dist}" failonerror="false" />
93  </target>
94
95 </project>
96
```

U odnosu na dosadašnje datoteke `build.xml`, razlike su sljedeće.

- Dodane su dvije nove varijable: `src.test` koja predstavlja direktorij u koji će biti smješteni izvorni kodovi testova te `build.test` koja predstavlja direktorij u koji će se spremiti izvršni oblik testova. Ovakvo razdvajanje koda od testova uobičajena je praksa kod svih većih projekata.

- Definirana je varijabla `junit.name` koja pokazuje na direktorij u kojem se nalaze sve JAR arhive koje čine alat JUnit.

- Zasebno se vode staze koje se virtualni stroj koristi prilikom prevođenja izvornog koda programa i prevođenja izvornog koda testova. Prilikom prevođenja izvornog koda programa virtualni stroj u našem slučaju treba samo odredišni direktorij u koji sprema prevedene datoteke. U složenijim programima koji koriste neke vanjske biblioteke ovdje bismo još uključili putanje i do tih biblioteka. Staza koju prevodioc mora koristiti prilikom prevođenja testova mora uključivati sve što uključuje i staza za sam program (pa je stoga uključuje -- vidi redak 26), a dodatno mora uključivati i direktorij u koji se spremaju izvršne verzije testova te sve potrebne biblioteke koje čine i sam alat JUnit; naime, sjetite se da u izvornom kodu testova postoje zasebne anotacije kao i pozivi metoda razreda `Assert` koje su definirane u okviru samog alata JUnit.

- Definirana su dva cilja: `compile` i `compile-tests` od kojih je prvi postavljen kao preduvjet drugome. Prvi cilj zadužen je za prevođenje programa a drugi cilj za prevođenje testova.

- Konačno, definiran je novi cilj `run-tests` koji najprije stvara potrebne direktorije, potom poziva naredbu `<junit>` koja generira izvještaj u dva formata (tekstualni i XML) i koja pokreće sve testove koje pronađe a identificira ih po nazivu datoteke izvornog koda (čekuje da u nazivu postoji `Test`), i na kraju svega poziva naredbu `<junitreport>` koja pokreće transformiranje izvještaja generiranog u obliku XML-a u izvještaj u obliku HTML-a koji je jednostavno pregledavati.

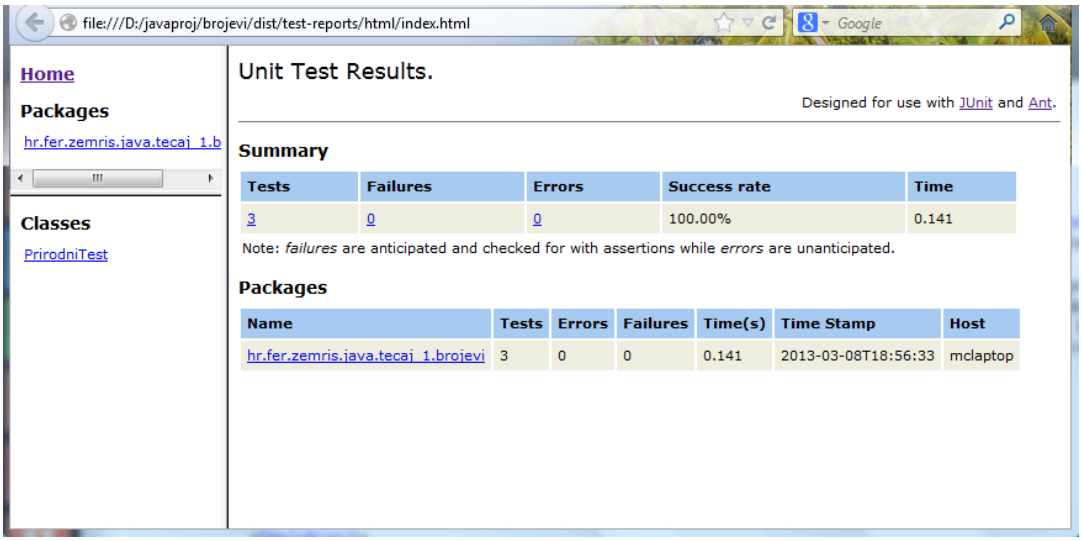
Prikazana datoteka `build.xml` je nešto jednostavnija od prethodnih jer su iz nje uklonjeni svi dijelovi koji nam trenutno nisu bili u fokusu (primjerice, izrada JAR arhive, izrada dokumentacije i slično).

Nakon ovih modifikacija pokrenite cilj `run-tests`.

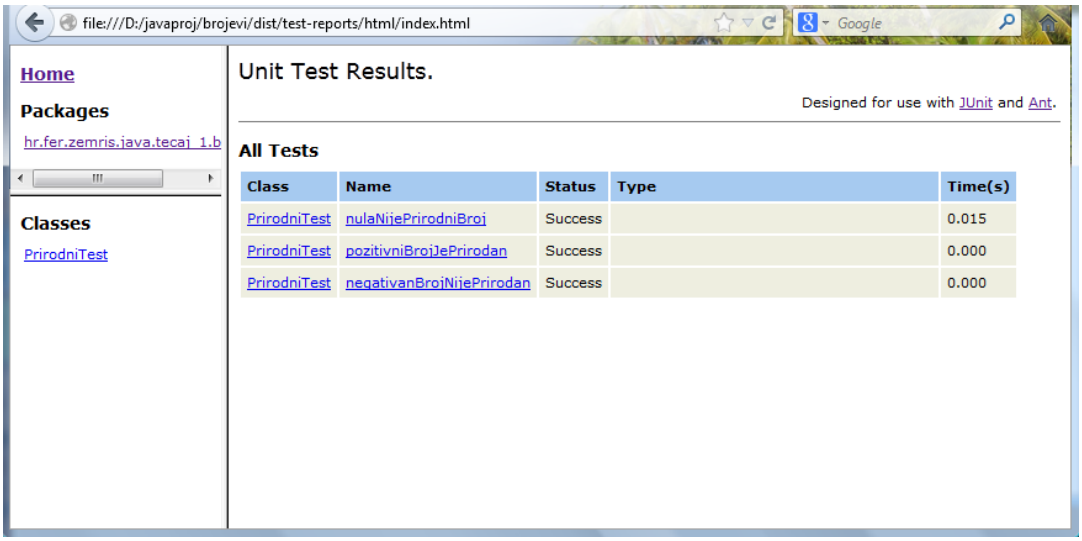
```
D:\javaproj\brojevi>ant run-tests
```

Kao rezultat ove akcije u direktoriju `dist/test-reports/html` nastat će izvještaj za sve pokrenute testove u formatu HTML. Izvorni izvještaj u formatu XML bit će spremljen u direktoriju `dist/test-reports/xml` tako da je temljem njega lagano generirati i izvještaje u drugim formatima. Otvorite datoteku `index.html` -- to će bit sumarni izvještaj o pokrenutim testovima koji bi trebao odgovarati izvještaju prikazanom na slici 3.6. Odabere li se tada primjerice detaljniji prikaz pokrenutih testova, dobit će se rezultat poput onog prikazanog na slici 3.7

Slika 3.6. Izvještaj o pokrenutim testovima



Slika 3.7. Detaljni pregled jednog izvještaja o pokrenutim testovima



Alat JaCoCo

Alat JaCoCo je alat koji omogućava analizu pokrivenosti koda. Pod pojmom *pokrivenost* koda podrazumijevamo analizu temeljem koje ćemo moći utvrditi koji su dijelovi koda prilikom pokretanja bili izvršeni a koji nisu. Ovakvi alati posebno su zanimljivi za uparivanje s alatima za testiranje koda (poput alata JUnit) jer omogućavaju korisniku da lagano utvrdi jesu li testovi koje je napisao dovoljni da prođu barem jednom kroz svaku napisanu naredbu ili ipak postoje dijelovi koda koji se prilikom pokretanja testova nisu izvršili (primjerice, ako u kodu postoji grananje, moguće je da su svi testovi kod pokrenuli na način da je svaki puta izvršena jedna grana i to uvijek ista, čime ne znamo je li kod koji se nalazi u drugoj grani ispravan). U slučaju da postoji takav kod, to je jasna indikacija da je potrebno napisati još testova koji će se pobrinuti da se izvede i tako identificirani dio koda.

Važno je napomenuti da iz činjenice da je sav kod pokriven ne slijedi i zaključak da je ponašanje koda ispitano u cijelosti. Evo jednostavnog primjera.

```
int metoda(boolean a, boolean b) {
 int broj = 0;
 if(a) {
   // nekako modificiraj broj (grana a-1)
 } else {
   // nekako modificiraj broj (grana a-2)
 }
 // radi nešto dalje s brojem
 if(b) {
   // nekako modificiraj broj (grana b-1)
 } else {
   // nekako modificiraj broj (grana b-2)
 }
 // radi nešto dalje s brojem
 return broj;
}
```

Potpuna pokrivenost koda može se postići uz samo dva testa. Prvi bi, primjerice, pozvao opisanu metodu uz oba argumenta postavljena na `true` čime bi se izvele grane a-1 i b-1 dok bi drugi test pozvao metodu uz oba argumenta postavljena na `false` čime bi se izvele grane a-2 i b-2. Na ovaj način sve bi naredbe iz napisane metode bile izvršene barem jednom (grane a-1, a-2, b-1 i b-2). Međutim, to nam i dalje ne bi ponudilo garanciju da smo u cijelosti ispitali kod. Naime, nismo provjerili bi li metoda generirala ispravan rezultat kada bi ulazni parametri bili

takvi da se prilikom izvođenja metode prolazi kroz grane a-1 te b-2 ili pak kroz grane a-2 i b-1. Stoga je važno zapamtiti da potpuna pokrivenost koda ne jamči da smo u cijelosti ispitali korektnost koda; međutim, nepotpuna pokrivenost koda jamči da u cijelosti *sigurno nismo* ispitali korektnost koda i stoga je ovaj pokazatalj vrlo važan.



Upozorenje

Ako nismo postigli potpunu pokrivenost koda, sigurni smo da nismo u cijelosti ispitali ponašanje koda. Obrat ne vrijedi.

Skinite alat JaCoCo. Malo detaljnija uputa dostupna je u dodatku C pa postupite u skladu s tom uputom.

Biblioteku JaCoCo integrirat ćemo s prethodno opisanim alatom JUnit. Ostanite stoga u projektu `brojevi`. Otvorite u uređivaču teksta datoteku `build.xml` i napravite sljedeće izmjene.

- Dodajte u definiciju projekta novi prostor imena kako je prikazano u nastavku.

```
xmlns:jacoco="antlib:org.jacoco.ant"
```

- Dodajte novu globalnu varijablu koja pokazuje na direktorij u koji ste smjestili alat JaCoCo. Ako ste radili prema uputama iz dodatka ove knjige, naredba će odgovarati naredbi prikazanoj u nastavku.

```
<property name="jacoco.home" value="d:/usr/jacoco-0.6.3" />
```

- Definirajte novi skup naredbi koje alat JaCoCo nudi za uporabu u alatu ant. Potrebna naredba prikazana je u nastavku.

```
<taskdef uri="antlib:org.jacoco.ant"
         resource="org/jacoco/ant/antlib.xml">
 <classpath path="${jacoco.home}/lib/jacocoant.jar"/>
</taskdef>
```

- Modificirajte cilj `run-tests` tako da koristi biblioteku JaCoCo. Najprije dodajte naredbu koja će stvoriti novi direktorij u koji će biti pohranjeni izvještaji analize pokrivenosti u formatu HTML.

```
<mkdir dir="${dist}/test-reports/coverage"/>
```

  Potom naredbu `<junit>` koja se koristi za pokretanje testova zamotajte u naredbu `<jacoco:coverage>` koja će obaviti analizu.

```
<jacoco:coverage destfile="${dist}/test-reports/xml/jacoco.exec">
 <junit printsummary="yes" haltonfailure="yes" fork="true"
        forkmode="once">
   ...
 </junit>
</jacoco:coverage>
```

  Konačno, nakon generiranja izvještaja alata ant dodajte naredbu koja će generirati izvještaj alata JaCoCo.

```
<!-- Generiraj izvještaj o pokrivenosti koda testovima -->
<jacoco:report>
 <executiondata>
  <file file="${dist}/test-reports/xml/jacoco.exec"/>
 </executiondata>

 <structure name="${ant.project.name}">
  <classfiles>
```

```
    <fileset dir="${build.classes}"/>
    <fileset dir="${build.test}"/>
   </classfiles>
   <sourcefiles encoding="UTF-8">
    <fileset dir="${src.java}"/>
    <fileset dir="${src.test}"/>
   </sourcefiles>
  </structure>

  <html destdir="${dist}/test-reports/coverage"/>

 </jacoco:report>
```

Ovako korigirana datoteka `build.xml` u cijelosti je prikazana u nastavku.

Primjer 3.4. Cjelokupna datoteka `build.xml` koja koristi alat JaCoCo

```
 1 <project name="Brojevi" default="jar" basedir="."
 2          xmlns:jacoco="antlib:org.jacoco.ant">
 3
 4  <description>
 5   Build datoteka za projekt Brojevi.
 6  </description>
 7
 8  <!-- Postavljanje globalnih varijabli -->
 9  <property name="src" location="src"/>
10  <property name="src.java" location="${src}/main/java"/>
11  <property name="src.test" location="${src}/test/java"/>
12  <property name="build" location="build"/>
13  <property name="build.classes" location="${build}/classes"/>
14  <property name="build.test" location="${build}/test"/>
15  <property name="dist"  location="dist"/>
16
17  <!-- Postavljanje varijabli za alate -->
18  <property name="junit.home" value="d:/usr/junit-4.11" />
19  <property name="jacoco.home" value="d:/usr/jacoco-0.6.3" />
20
21  <taskdef uri="antlib:org.jacoco.ant"
22           resource="org/jacoco/ant/antlib.xml">
23   <classpath path="${jacoco.home}/lib/jacocoant.jar"/>
24  </taskdef>
25
26  <!-- Definiranje staze za prevođenje koda -->
27  <path id="compile.path">
28   <pathelement location="${build.classes}"/>
29  </path>
30
31  <!-- Definiranje staze za prevođenje testova -->
32  <path id="test.path">
33   <path refid="compile.path"/>
34   <pathelement location="${build.test}"/>
35   <fileset dir="${junit.home}">
36    <include name="**/*.jar"/>
37   </fileset>
38  </path>
39
40  <target name="init">
41   <!-- Stvaranje vremenske oznake -->
```

```
42   <tstamp/>
43   <!-- Stvaranje potrebnih direktorija -->
44   <mkdir dir="${build}"/>
45   <mkdir dir="${dist}"/>
46  </target>
47
48  <target name="compile" depends="init"
49   description="Prevođenje izvornog koda">
50   <mkdir dir="${build.classes}"/>
51   <!-- Prevođenje Java koda iz ${src} u ${build} -->
52   <javac srcdir="${src.java}" destdir="${build.classes}"
53    classpathref="compile.path"
54    encoding="UTF-8" debug="on"
55    debuglevel="lines,vars,source"
56    includeAntRuntime="false" />
57  </target>
58
59  <target name="compile-tests" depends="compile"
60   description="Prevođenje izvornog koda testova">
61   <mkdir dir="${build.test}"/>
62   <!-- Prevođenje Java koda iz ${src} u ${build} -->
63   <javac srcdir="${src.test}" destdir="${build.test}"
64    classpathref="test.path"
65    encoding="UTF-8" debug="on"
66    debuglevel="lines,vars,source"
67    includeAntRuntime="false" />
68  </target>
69
70  <target name="run-tests" depends="compile-tests"
71          description="Izvođenje definiranih testova" >
72   <mkdir dir="${dist}/test-reports/xml"/>
73   <mkdir dir="${dist}/test-reports/html"/>
74   <mkdir dir="${dist}/test-reports/coverage"/>
75
76   <!-- Pokreni testove uz analizu pokrivenosti -->
77   <jacoco:coverage
78      destfile="${dist}/test-reports/xml/jacoco.exec">
79    <junit printsummary="yes" haltonfailure="yes"
80           fork="true" forkmode="once">
81     <classpath refid="test.path" />
82
83     <formatter type="plain"/>
84     <formatter type="xml"/>
85
86     <batchtest fork="yes" todir="${dist}/test-reports/xml">
87      <fileset dir="${src.test}">
88       <include name="**/*Test*.java"/>
89      </fileset>
90     </batchtest>
91    </junit>
92   </jacoco:coverage>
93
94   <!-- Generiraj izvještaj na temelju testova -->
95   <junitreport todir="${dist}/test-reports/xml">
96    <fileset dir="${dist}/test-reports/xml">
97     <include name="TEST-*.xml"/>
98    </fileset>
99    <report format="frames" todir="${dist}/test-reports/html"/>
```

```
100   </junitreport>
101
102   <!-- Generiraj izvještaj o pokrivenosti koda testovima -->
103   <jacoco:report>
104    <executiondata>
105     <file file="${dist}/test-reports/xml/jacoco.exec"/>
106    </executiondata>
107
108    <structure name="${ant.project.name}">
109     <classfiles>
110      <fileset dir="${build.classes}"/>
111      <fileset dir="${build.test}"/>
112     </classfiles>
113     <sourcefiles encoding="UTF-8">
114      <fileset dir="${src.java}"/>
115      <fileset dir="${src.test}"/>
116     </sourcefiles>
117    </structure>
118
119    <html destdir="${dist}/test-reports/coverage"/>
120
121   </jacoco:report>
122
123  </target>
124
125  <target name="clean"
126   description="Brisanje generiranog sadržaja" >
127   <!-- Obriši direktorije ${build} i ${dist} -->
128   <delete dir="${build}" failonerror="false" />
129   <delete dir="${dist}" failonerror="false" />
130  </target>
131
132 </project>
133
```

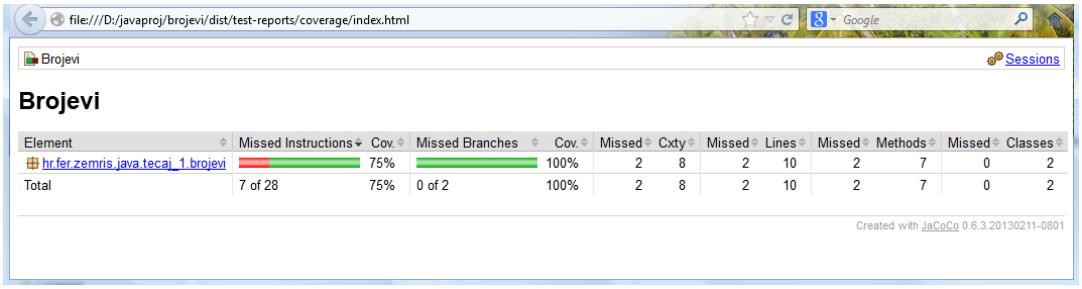
Nakon ovih modifikacija pokrenite cilj `run-tests`.

```
D:\javaproj\brojevi>ant run-tests
```

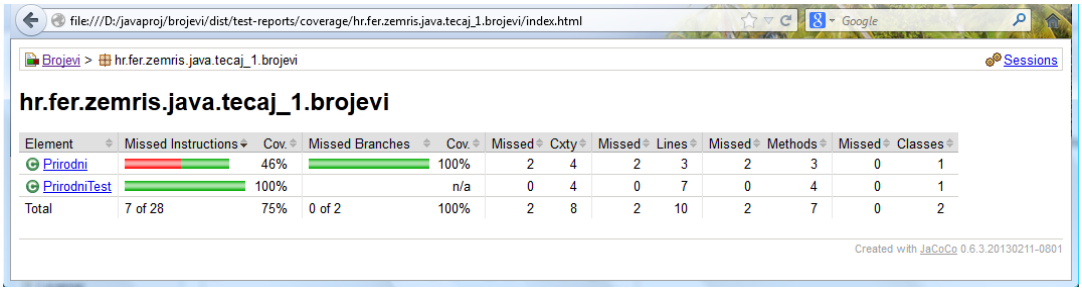
Kao rezultat ove akcije u direktoriju `dist/test-reports/html` nastat će kao i prije izvještaj za sve pokrenute testove u formatu HTML. Izvorni izvještaj u formatu XML bit će spremljen u direktoriju `dist/test-reports/xml` tako da je temljem njega lagano generirati i izvještaje u drugim formatima. Uz ova dva direktorija, sada imamo i treći: `dist/test-reports/coverage`. Uđite u taj direktorij i otvorite datoteku `index.html` -- to će bit sumarni izvještaj o pokrivenosti koda. Izvještaj bi trebao odgovarati izvještaju prikazanom na slici 3.8. Prema stanju na slici, pokrivenost koda u paketu hr.fer.zemris.java.tecaj_1.brojevi iznosi 75%. Slijedimo li poveznicu koju predstavlja ime paketa, dobit ćemo detaljniji izvještaj kakav je prikazan na slici 3.9. Tu vidimo pokrivenost koda razloženu po pojedinim razredima. U paketu imamo dva prikazana razreda: `Prirodni` koji predstavlja implementaciju naše demonstracijske biblioteke te `PrirodniTest` koji predstavlja implementaciju testova. Zatražimo li detaljniji izvještaj za test, dobit ćemo izvještaj kakav je prikazan na slici 3.10. Sa slike možemo primjetiti da je kod svih metoda u testu pokriven. To je i bilo za očekivati -- budući da smo pokrenuli testiranje, sve naredbe *testa* su doista i izvršene. Vratimo li se stranicu natrag pa zatražimo detaljniji izvještaj za razred `Prirodni`, dobit ćemo izvještaj kakav je prikazan na slici 3.11. Ovdje situacija više nije sjajna. Od tri metode, jedna je pokrivena u cijelosti a dvije uopće nisu. Ovaj izvještaj je važano -- s njega je jasno vidljivo da imamo, primjerice, metodu `sljedbenik` čije naredbe nisu izvršene. To automatski ukazuje da nam nedostaje još testova i da je sada pravo vrijeme za napisati ih. Cilj je dobiti pokrivenost svih metoda koje smo napisali u iznosu od 100%. Zatražimo li detaljniji izvještaj za metodu `sljedbenik`, dobit ćemo izvještaj kao što je to prikazano

na slici 3.12. S te slike lagano je identificirati i naredbe koje nisu izvršene -- prikazane su s crvenom pozadinom.

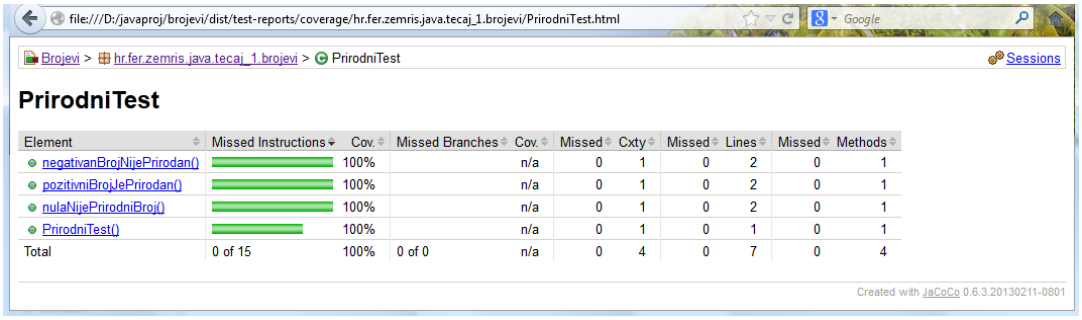
Slika 3.8. Izvještaj o pokrivenosti koda



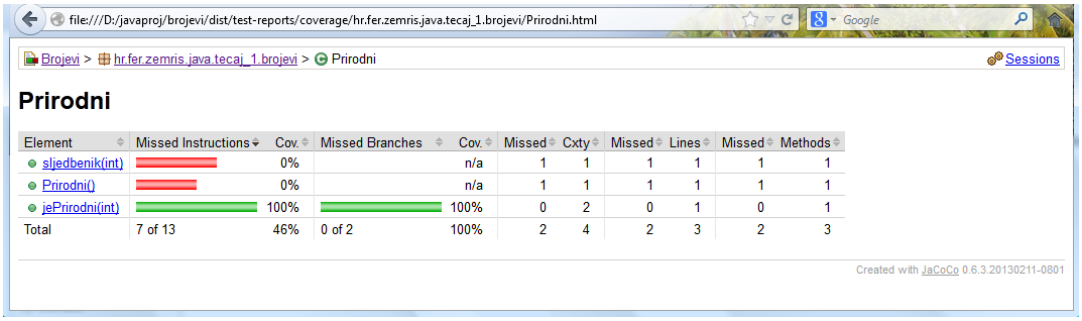
Slika 3.9. Detaljniji izvještaj o pokrivenosti koda



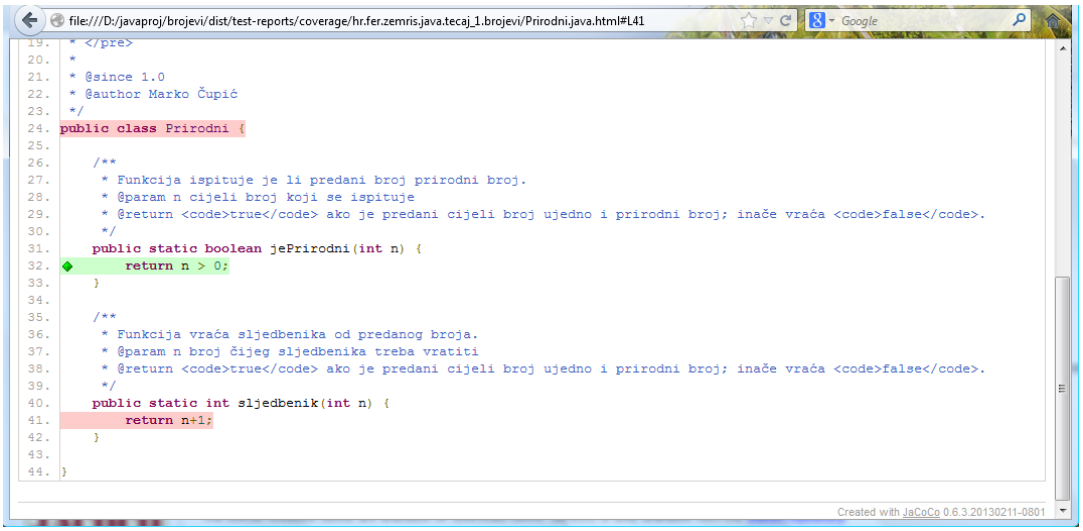
Slika 3.10. Izvještaj o pokrivenosti koda u razredi `PrirodniTest`



Slika 3.11. Izvještaj o pokrivenosti koda u razredi `Prirodni`



Slika 3.12. Detaljni grafički prikaz o pokrivenosti koda u razredu `Prirodni`



Poglavlje 4. Jednostavniji primjeri

U okviru ovog poglavlja krenut ćemo u istraživanje programskog jezika Java kroz nekoliko jednostavnih primjera. Na kraju poglavlja detaljnije ćemo se upoznati i s načinom kako se u Javi radi sa stringovima.

Primjer 1: ispis argumenata naredbenog retka

U okviru prvog zadatka napisat ćemo program koji će na zaslon ispisati argumente predane programu prilikom pokretanja iz naredbenog retka. Program ćemo nazvati `IspisArgumenata` i smjestit ćemo ga u paket `hr.fer.zemris.java.primjeri`. Rješenje je prikazano u nastavku.

Primjer 4.1. Ispis argumenata naredbenog retka

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 /**
 4  * Program ispisuje argumente koje prima putem naredbenog
 5  * retka.
 6  *
 7  * @author Marko Cupic
 8  * @version 1.0
 9  */
10 public class IspisArgumenata {
11
12  /**
13   * Metoda koja se poziva prilikom pokretanja
14   * programa. Argumenti su objasnjeni u nastavku.
15   *
16   * @param args Argumenti iz komandne linije.
17   */
18  public static void main(String[] args) {
19
20   int brojArgumenata = args.length;
21
22   for(int i = 0; i < brojArgumenata; i++) {
23    System.out.println(
24     "Argument " + (i+1) + ": " + args[i]
25    );
26   }
27  }
28
29 }
30
```

Program s izvođenjem započinje u metodi `main` koja je prikazana počev od retka 18. Jedini argument koji metoda `main` prima upravo je polje stringova koje predstavlja argumente koje je korisnik zadao kroz naredbeni redak. Koristeći svojstvo `.length` koje ima svako polje, u retku 20 dohvaćamo broj elemenata polja što je ekvivalentno broju argumenata koji su predani kroz naredbeni redak.

Nakon što smo utvrdili koliko imamo argumenata, u `for`-petlji prolazimo kroz svaki argument te ga na zaslon ispisujemo pozivom metode `System.out.println(...)`. Kao argument

toj metodi predajemo string koji je konkatenacija teksta `Argument`, rednog broja, dvotočke, razmaka te predanog argumenta.

Pretpostavimo li da smo program smjestili u projekt koji prevedene datoteke smješta u direktorij `bin`, program bismo mogli pokrenuti na sljedeći način, uz pretpostavku da smo pozicionirani u korijenski direktorij projekta (kako je redak predugačak, prikazan je razlomljen u dva; međutim, prilikom unosa to treba zadati kao jednu naredbu ljuske).

```
projekt> java -cp bin hr.fer.zemris.java.primjeri.IspisArgumenata
         Zagreb Dubrovnik Split
```

Rezultat izvođenja bio bi sljedeći ispis.

```
Argument 1: Zagreb
Argument 2: Dubrovnik
Argument 3: Split
```

Umjesto ručne konkatenacije stringova operatorom `+`, ispis smo mogli postići i uporabom naredbe `format` koja je vrlo slična naredbi `printf` koju nudi programski jezik C. U tom slučaju, u tijelu petlje `for` stajalo bi sljedeće.

```
System.out.format("Argument %d je: %s\n", i+1, args[i]);
```

Prvi argument naredbe je formatni specifikator; vrijednost `%d` definira da je argument koji na tom mjestu treba ispisati broj a vrijednost `%s` da se radi o stringu. Redosljedom kojim su navedeni u formatnom specifikatoru, naredba uzima argumente iz nastavka i gradi konačni string koji potom ispisuje.

Primjer 2: izračun e^x

U okviru ovog zadatka napisat ćemo program koji će izračunati i na zaslon ispisati rezultat aproksimacije izraza *e^x* za vrijednost *x* koju će korisnik zadati kao argument naredbenog retka. Program aproksimaciju treba izračunati uporabom Taylorovog reda i to koristeći prvih 10 članova tog reda. Program ćemo nazvati `SumaReda` i smjestit ćemo ga u paket `hr.fer.zemris.java.primjeri`. Rješenje je prikazano u nastavku.

Primjer 4.2. Izračun e^x

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 /**
 4  * @author Marko Cupic
 5  * @version 1.0
 6  */
 7 public class SumaReda {
 8
 9  /**
10   * Metoda koja se poziva prilikom pokretanja
11   * programa. Argumenti su objašnjeni u nastavku.
12   *
13   * @param args Argumenti iz komandne linije.
14   */
15  public static void main(String[] args) {
16
17   if(args.length != 1) {
18    System.err.println(
19     "Program mora imati jedan argument!"
20    );
21    System.exit(1);
```

```
22    }
23
24    double broj = Double.parseDouble(args[0]);
25
26    System.out.println("Računam sumu...");
27
28    double suma = racunajSumu(broj);
29
30    System.out.println("f(" + broj + ")=" + suma);
31  }
32
33  /**
34   * Računa e^x razvojem u Taylorov red, prema formuli:
35   * e^x=1+x+(x^2/(2!))+(x^3/(3!))+(x^4/(4!))+...
36   *
37   * @param broj argument funkcije e^x
38   *
39   * @return iznos funkcije u točki x=broj dobiven kao
40   *         suma prvih 10 članova Taylorovog reda.
41   */
42  private static double racunajSumu(double broj) {
43    double suma = 0.0;
44    double potencija = 1.0;
45    double faktorijela = 1.0;
46
47    suma += 1.0;
48
49    for(int i = 1; i < 10; i++) {
50      potencija = potencija * broj;
51      faktorijela = faktorijela * i;
52      suma += potencija/faktorijela;
53    }
54
55    return suma;
56  }
57 }
58
```

Tijelo programa razbijeno je u dvije metode. Zadaća metode `main` je prikupiti podatke iz naredbenog retka, pozvati metodu za izračun aproksimacije vrijednosti te ispisati rezultat. Zadaća izračuna aproksimacije izdvojena je u zasebnu metodu `racunajSumu` kako bi se povećala čitljivost koda i kako bi se omogućilo da se dokumentira postupak kojim se obavlja izračun.

Obratite pažnju kako je napisana dokumentacija -- napisano je što metoda radi i kojim postupkom, ali ne i kako to metoda radi. Korisnička dokumentacija ne treba sadržavati informacije poput "ići ćemo u `for`-petlji i onda ćemo akumulirati trenutni rezultat u pomoćnoj varijabli". Takva dokumentacija je loša dokumentacija. Javadoc komentari koje ovdje pišemo su *dokumentacija za korisnika Vaših metoda*. Korisnika može zanimati algoritam koji koristite te parametri algoritma. U ovom slučaju, korisniku može biti važno da zna da koristite Taylorov razvoj i da uzimate u obzir samo 10 članova jer se temeljem tih informacija može doći do procjene ograde pogreške koju opisana metoda čini prilikom aproksimacije. Takve informacije pripadaju u Javadoc; implementacijski detalji nikako (jeste li koristili petlju `for` ili `while` i slično).

Pogledajmo i kakav će biti rezultat izvođenja programa. Pretpostavimo da smo ga pokrenuli na sljedeći način.

```
projekt> java -cp bin hr.fer.zemris.java.primjeri.SumaReda 1
```

Rezultat izvođenja bio bi sljedeći ispis.

```
Računam sumu...
f(1.0)=2.7182815255731922
```

Primjer 3: formatirani ispis decimalnih brojeva

Potrebno je napisati program koji definira metodu koja prima polje `double`-ova i ispisuje ih na zaslon po zadanom formatu. Napisati glavni program koji će brojeve ispisati na sljedeće načine:

- najmanje tri mjesta za cijelobrojni dio, dva mjesta za decimalni te
- dva + dva mjesta s obaveznim ispisom predznaka.

Program ćemo nazvati `FormatiraniIspisDecBrojeva` i smjestit ćemo ga u paket `hr.fer.zemris.java.primjeri`. Rješenje je prikazano u nastavku.

Primjer 4.3. Formatirani ispis brojeva

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.text.DecimalFormat;
 4
 5 /**
 6  * Program ilustrira formatirani ispis decimalnih
 7  * brojeva.
 8  *
 9  * @author Marko Cupic
10  * @version 1.0
11  */
12 public class FormatiraniIspisDecBrojeva {
13
14  /**
15   * Metoda na standardni izlaz ispisuje polje decimalnih
16   * brojeva prema zadanom formatu.
17   *
18   * @param polje polje decimalnih brojeva koje treba ispisati
19   * @param format format koji govori kako polje treba ispisati
20   *
21   * @see DecimalFormat
22   */
23  public static void ispis(double[] polje, String format) {
24   DecimalFormat formatter = new DecimalFormat(
25    format
26   );
27   for(int i = 0; i < polje.length; i++) {
28    System.out.println(
29     "(" + i + "): [" + formatter.format(polje[i]) + "]"
30    );
31   }
32  }
33
34  /**
35   * Metoda koja se poziva prilikom pokretanja
36   * programa. Argumenti su objašnjeni u nastavku.
37   *
38   * @param args Argumenti iz komandne linije.
39   */
```

```
40  public static void main(String[] args) {
41
42   double[] brojevi = new double[] {
43    3.712, 55.813, 55.816, -4.18
44   };
45
46   ispis(brojevi, "000.00");
47   System.out.println("--------------");
48   ispis(brojevi, "+00.00;-00.00");
49  }
50
51 }
52
```

Program se sastoji od dvije metode. Zadaća metode `main` je pripremiti polje brojeva te pozvati metodu za formatirani ispis uz definiranje željenog načina formatiranja. Metoda `main` ujedno pokazuje kako se dinamički stvara polje decimalnih brojeva.

Za formatiranje decimalnih brojeva, odnosno za njihovu pretvorbu u string prema zadanoj formatnoj specifikaciju koristit ćemo ugrađeni razred `DecimalFormat`. Da bismo ga mogli koristiti, razred je potrebno uključiti u izvorni kod uporabom naredbe `import` kako je to prikazano u retku 3. Prema pravilima programskog jezika Java, da bismo mogli samo navođenjem njihova imena koristiti razrede koji su definirani u drugim paketima u odnosu na paket u kojem se nalazi trenutni razred koji pišemo, nužno ih je najprije uključiti naredbom `import`. Iznimka su razredi definirani u paketu `java.lang` koji su automatski uključeni. Razred `DecimalFormat` koristi se na način kako to ilustrira sljedeći isječak koda.

```
1 DecimalFormat df = new DecimalFormat("000.00");
2 String s1 = df.format(3.51);
3 String s2 = df.format(52.48);
```

Redak 1 primjera pokazuje stvaranje pomoćnog objekta koji će obavljati formatiranje brojeva. Prilikom stvaranja, predajemo jedan string koji predstavlja formatnu specifikaciju. Jednom kada smo objekt stvorili, u retcima 2 i 3 ga koristimo kako bismo formatirali dva decimalna broja pozivom metode `format`. Više o načinu stvaranja objekata i što se točno događa kada pozivamo operator `new` bit će objašnjeno u kasnijim dijelovima knjige. Za sada pokušajte opisani način uporabe naprosto prihvatiti tako kako je naveden.

Detalji o formatnoj specifikaciji mogu se pronaći uz dokumentaciju razreda `DecimalFormat`. Format se može pisati na dva načina: ili se posebno navodi format za pozitivne pa posebno format za negativne brojeve pri čemu su ti formati međusobno razdvojeni znakom točke-zareza, ili se navodi samo jedan format koji se univerzalno primjenjuje i za pozitivne i za negativne brojeve. U formatu, znamenka 0 označava mjesto na kojem treba ispisati znamenku broja koji se formatira ili doslovno nulu ako broj na tom mjestu nema znamenke. Znak točke označava poziciju decimalne točke. Tako će, primjerice, formatni specifikator `"000.00"` definirati da se i pozitivni brojevi i negativni brojevi ispisuju na tri znamenke u cjelobrojnom dijelu i dvije u decimalnom, uz dopunu nula ako je potrebno. Dodatno, kako je to univerzalni format, negativnim brojevima će se nadodati znak `-` na početak.

Formatni specifikator `"+000.00;-00.00"` posebno format za pozitivne brojeve a posebno za negativne. Prema njemu, pozitivnom broju bi se najprije dodao znak plus, potom bi se cijelobrojni dio ispisao na tri znamenke a decimalni na dvije. Drugi dio formatnog specifikator kaže da se za negativne brojeve najprije ispisuje znak minusa a potom cjelobrojni dio na dvije znamenke i decimalni na još dvije. Tako bi primjerice 3.14 bio pretvoren u string `+003.14` dok bi -1 bio pretvoren u string `-01.00`. Važno je uočiti da se lijevi dio specifikatora odnosi na pozitivne brojeve a desni na negativne ali ne zato što smo lijevo napisali znak plus već stoga što je tako definirano u dokumentaciji tog razreda. Primjerice, uz formatnu specifikaciju `"-000.00;+00.00"` broj 3.14 koji je pozitivan bio bi pretvoren u string `-003.14` (jer tako to propisuje prvi dio specifikatora) dok bi negativan broj -3.2 bio pretvoren u string `+03.20`.

Pokretanjem ovog programa dobit ćemo ispis koji je prikazan u nastavku.

```
(0): [003.71]
(1): [055.81]
(2): [055.82]
(3): [-004.18]
--------------
(0): [+03.71]
(1): [+55.81]
(2): [+55.82]
(3): [-04.18]
```

Primjer 4: izračun sume brojeva s tipkovnice

Naš sljedeći zadatak je napisati program koji će s tipkovnice učitavati decimalne brojeve broj po broj i računati njihovu sumu, sve dok se upisuju nenegativni brojevi. U trenutku upisa negativnog broja ispisuje se izračunata suma i program prekida s radom. Program ćemo nazvati `CitanjeSTipkovnice` i smjestit ćemo ga u paket `hr.fer.zemris.java.primjeri`. Rješenje je prikazano u nastavku.

Primjer 4.4. Sumiranje brojeva unesenih preko tipkovnice

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.BufferedReader;
 4 import java.io.IOException;
 5 import java.io.InputStreamReader;
 6
 7 public class CitanjeSTipkovnice {
 8
 9  public static void main(String[] args) throws IOException {
10   System.out.println(
11    "Program za računanje sume pozitivnih brojeva."
12   );
13   System.out.println("Unosite brojeve, jedan po retku.");
14   System.out.println(
15    "Kada unesete negativan broj, ispisat ce se suma."
16   );
17
18   BufferedReader reader = new BufferedReader(
19    new InputStreamReader(System.in)
20   );
21
22   double suma = 0.0;
23   while(true) {
24    String redak = reader.readLine();
25    if(redak==null) break;
26    double broj = Double.parseDouble(redak);
27    if(broj<0) break;
28    suma += broj;
29   }
30
31   System.out.print("Suma je: ");
32   System.out.println(suma);
33
34   reader.close();
35  }
36
```

```
37 }
38
```

Ovaj program ilustrira pisanje aplikacije koja podatke dohvaća preko tipkovnice ili bilo čega što program dobiva kao standardni ulaz. Da bismo mogli sa standardnog ulaza čitati podatke, na raspolaganju nam stoji objekt `System.in`. Kako je taj objekt predviđen za čitanje binarnih podataka, uobičajeno ga je dekorirati drugim objektom preko kojega ćemo raditi čitanje. Isječak koda koji to radi prikazan je u nastavku, i opet, preporučamo da na ovom mjestu taj dio koda prihvatite kao kod koji treba pisati kada se čita s tipkovnice. Što se točno događa u tom isječku koda, bit će objašnjeno u kasnijim poglavljima kada ćemo proučavati datoteke i tokove podataka.

```
BufferedReader reader = new BufferedReader(
 new InputStreamReader(System.in)
);
```

Jednom kada smo izgradili pomoćni objekt `reader`, jedan redak s tipkovnice možemo pročitati pozivom njegove naredbe `readLine()`. Ta metoda vraća pročitani redak kao string ili vraća `null` u slučaju da čitanje više nije moguće. Poziv metode čitanja može prilikom izvođenja generirati pogrešku tipa `java.io.IOException`. Kako tu metodu pozivamo u metodi `main` i kako još nismo objasnili upravljanje pogreškama (kada ćemo znati ovo riješiti bolje), deklaraciji metode `main` potrebno je dodati `throws IOException` kako bismo kod mogli prevesti (vidljivo u retku 9). Sva tri spomenuta razreda (`BufferedReader`, `InputStreamReader` te `IOException`) prije uporabe potrebno je još uključiti naredbom `import` što je prikazano u retcima 3 do 5.

Jednom kada je program gotov sa čitanjem podataka, korišteni objekt potrebno je zatvoriti pozivom naredbe `close()` što je učinjeno u retku 34. Nakon toga čitanje s tog ulaza više neće biti moguće.

Naprednije čitanje ulaznih podataka

Kroz prethodni zadatak dali smo primjer kako čitati podatke s tipkovnice i to redak po redak. Često nam je takav način dohvata podataka neprikladan: primjerice, što ako u svakom retku može biti jedan ili više podataka (tekstovi, brojevi i slično) ili što ako od korisnika tražimo da unese nekoliko podataka ali nam uopće nije bitno hoće li ih sve unijeti u istom retku ili će svaki unijeti u zasebnom retku ili će napraviti nekakvo drugačije grupiranje.

Za rješavanje takvih problema na raspolaganju nam stoji razred `Scanner` definiran u paketu `java.util`. Konkretan objekt ovog razreda možemo stvarati na nekoliko načina. Pogledajmo stoga nekoliko primjera.

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Prikazani isječak koda stvara primjerak razreda `Scanner` koji podatke čita preko tipkovnice. Potom nad njim poziva metodu `nextInt()` kako bi pročitala jedan cijeli broj.

Primjer koji podatke čita iz datoteke prikazan je u nastavku.

```
Scanner sc = new Scanner(new File("brojevi"));
while (sc.hasNextLong()) {
 long aLong = sc.nextLong();
 // napravi nešto s tim brojem
 // ...
}
```

Prethodno prikazani isječak stvara objekt `Scanner` koji će podatke čitati iz tekstovne datoteke nazvane `brojevi`. Korišteni razred `File` nalazi se u paketu `java.io` pa ga je također po-

trebno uključiti. program potom u petlji `while` stvoreni objekt ispituje je li moguće iz datoteke pročitati još jedan cijeli broj (tipa `long`) i ako je, onda ga čita.

Konačno, pogledajmo još jedan primjer koji podatke čita iz unaprijed pripremljenog stringa.

```
String input = "1    aba 2 aba red aba blue aba";
Scanner s = new Scanner(input).useDelimiter("\\s*aba\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Stvoreni objekt `Scanner` kao ulaz koristi pripremljeni tekst koji potom razlama u skladu s predanim regularnim izrazom. U tom regularnom izrazu, dio `\\s` predstavlja bilo kakvu prazninu (razmaknicu, tabulator, prelazak u novi red). Dodavanje zvjezdice nakon toga uvodi semantiku: "prethodno se mora pojaviti nula, jednom ili više puta". Time početak regularnog izraza odnosno `\\s*` zapravo kaže: pronađi nula, jednu ili više praznina. Pogledate li dalje regularni izraz, traži se pojava niza `aba` i nakon toga opet pojava nula, jedne ili više praznine. Svi dijelovi teksta koji se uspiju poklopiti s ovim izrazom bit će iz teksta izrezani i na tim mjestima tekst će se raspasti na više elemenata. U našem slučaju, elementi će biti `1`, `2`, `red`, `blue` te prazan element (desno od zadnjeg separatora `aba`). Nakon toga, elemente dohvaćamo pozivima odgovarajućih metoda stvorenog objekta.

Formatirana izrada teksta

U situacijama u kojima trebamo generirati tekst u skladu s određenim formatom, već smo se upoznali s jednom mogućnošću: za pretvorbu decimalnih brojeva možemo koristiti razred `DecimalFormat`. No što ako trebamo istovremeno formatirati i decimalne brojeve, i cijele brojeve, i druge tekstove te datume i vremena? Za tu svrhu Java nam nudi metodu `format` definiranu unutar razreda `String`. Naredba `format` u određenom je smislu ekvivalent naredbi `sprintf` koja se uobičajeno koristi za te svrhe u programskom jeziku C. Evo primjera uporabe.

```
String tekst = String.format(
 "%d je veće od %.3f dok je %s ime a %1$d je prvi predani broj.\n",
 4, 3.14, "Pero"
);
System.out.println(tekst);
```

Izvođenjem ovog koda dobit ćemo ispis prikazan u nastavku.

```
4 je veće od 3.140 dok je Pero ime a 4 je prvi predani broj.
```

Uočite kako se uporabom sintakse `x$` možemo direktno referencirati na `x`-ti predani argument. Tako posljednji navedeni specifikator `%1$d` traži da se ponovno u tekst doda cijeli broj koji je dan kao prvi argument naredbe. Za detaljnije informacije o načinu uporabe ove funkcije svakako pogledajte dokumentaciju dostupnu na adresi: http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html.

Stringovi u Javi

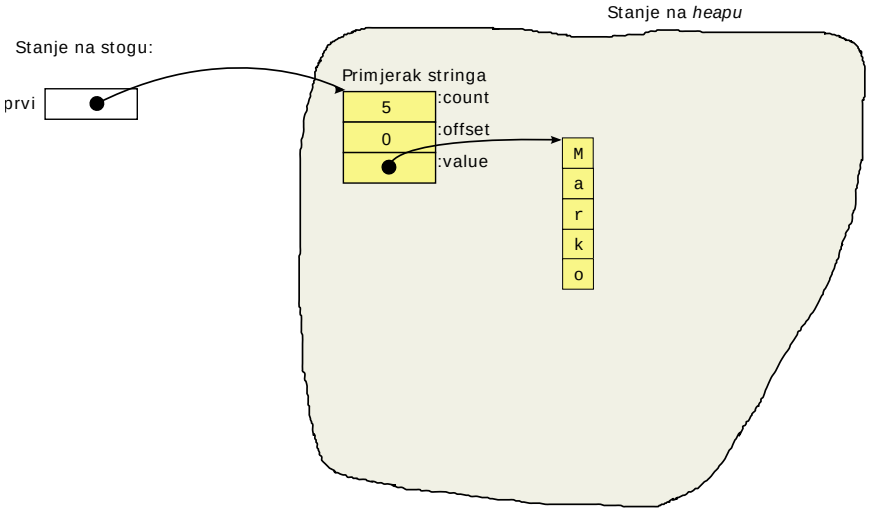
Stringovi su izuzetno često korišten tip podatka. Zbog toga svi moderni programski jezici korisnicima izlaze u susret pokušavajući rad sa stringovima učiniti što jednostavnijima i sakriti kompleksnost koja je povezana s provođenjem operacija nad njima. To pak, nažalost, često dovodi do situacije u kojoj korisnik nesvjesno piše loš kod jer nije svjestan što je zapravo napravio. Zadaća ovog podpoglavlja jest pojasniti kako se u Javi radi sa stringovima i koje su opasnosti. Kroz ovaj tekst osvrnut ćemo se na implementaciju Stringova kakva je bila do verzije 7u6 Jave. Od verzije 7u6 prikazani mehanizam dijeljenja znakovnih polja je izbačen čime je složenost metode `substring` s konstantne povećana na linearnu.

Prvo pitanje na koje ćemo odgovoriti jest: kako razred `String` interno pohranjuje podatke o svome sadržaju? Pogledajmo prvi primjer.

```
1 String prvi = new String(new char[] {'M', 'a', 'r', 'k', 'o'});
2 String drugi = new String(prvi);
3 String treci = drugi.substring(1, 4);
4 String cetvrti = prvi.replace('M', 'D');
```

Krenut ćemo s retkom 1. Njegovim izvođenjem, u memoriji nastaje objekt koji predstavlja string čiji je sadržaj `Marko`. Interno, taj objekt čuva tri podatka: referencu na kopiju znakovnog polja koje predstavlja sadržaj stringa (varijabla `value`), indeks prvog znaka u tom polju od kojeg započinje sadržaj tog stringa (varijabla `offset`) te broj znakova u tom polju koji pripadaju sadržaju tog stringa. Posljedica ovakve organizacije podataka je da prilikom izvođenja retka 1 u memoriji računala nastaju dva nova objekta što je prikazano na slici u nastavku.

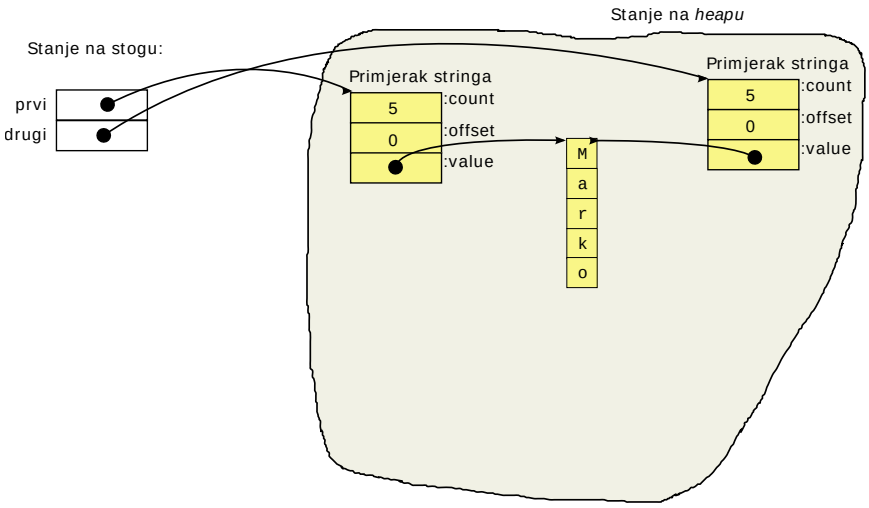
Slika 4.1. Stanje u memoriji (1)



Nakon izvedenog retka 1, definirana je lokalna varijabla `prvi` koja je po tipu referenca na objekt koji se nalazi na *heapu* i koji predstavlja string `Marko`. Taj objekt čuva referencu na znakovno polje u koje je pospremljen sadržaj stringa, varijablu `offset` ima postavljenu na 0 jer njegov sadržaj kreće od početka a varijablu `count` ima postavljenu na 5 jer mu pripada toliko znakova.

Izvođenjem retka 2 stvara se novi string koji predstavlja kopiju prvog stringa. Situacija u memoriji nakon tog koraka prikazana je na slici u nastavku.

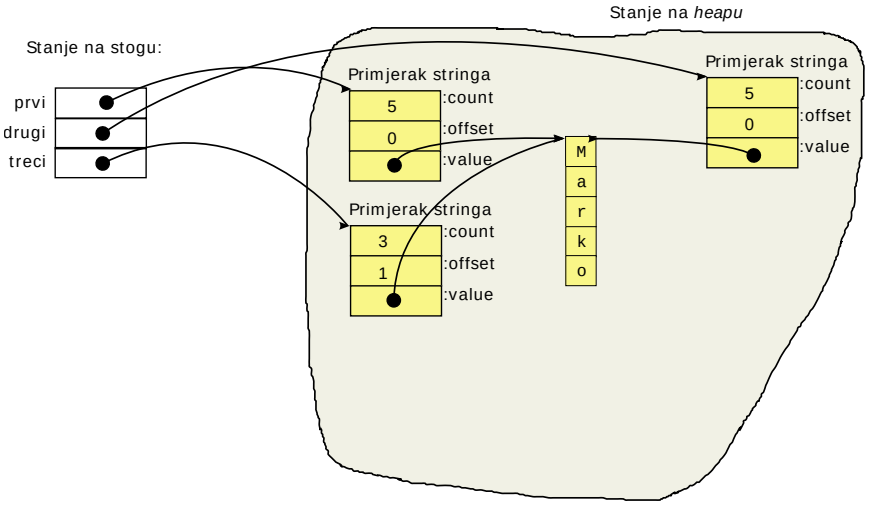
Slika 4.2. Stanje u memoriji (2)



Ovdje je sada važno uočiti važan implementacijski detalj. Stvaranjem kopije stringa u memoriji se ne stvara doslovno još jedna kopija stringa. Primjerice, da smo imali prvi string koji zauzima 1MB memorije, stvaranjem njegove kopije ne bismo potrošili još 1MB memorije. Java kopiranje stringova izvodi u složenosti O(1) na način da stvori novi objekt koji koristi isto znakovno polje koje koristi i originalni objekt. Na taj način izbjegava se kopiranje sadržaja stringa i dobiva dijeljenje memorije između više stringova.

U retku 3, traži se stvaranje novog stringa koji sadrži podniz znakova originalnog stringa: treba sadržavati sve znakove koji počinju od lokacije 1 uključivo (dakle, od znaka `a`), pa do lokacije 4 isključivo (tj. ukupno sadrži znakove s lokacija 1, 2 i 3) čime je sadržaj tog stringa `ark`. Stanje u memoriji prikazano je u nastavku.

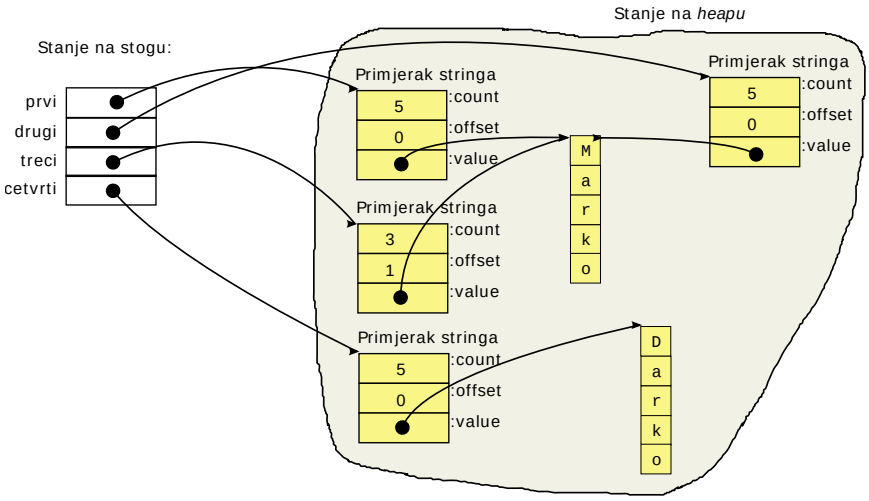
Slika 4.3. Stanje u memoriji (3)



Sa slike se opet može uočiti da se sadržaj stringa nije kopirao. Java i sve operacije generiranja podnizova provodi također u konstantnom vremenu odnosno sa složenošću O(1). Za potrebe provođenje ove operacije stvoren je novi objekt koji opet koristi isto znakovno polje. Razlika je samo u sadržaju varijabli `offset` koji je u ovom objektu postavljen na vrijednost 1 te u sadržaju varijable `count` koji je postavljen na 3 (toliko znakova polja pripada tom novom stringu). Uočimo usput, dohvat duljine stringa u Javi je također operacija koja se izvodi u konstantnom vremenu: metoda `length()` naprosto vraća sadržaj varijable `count`.

Pogledajmo sada još što će biti rezultat izvođenja četvrtog retka. U četvrtom retku zahtjeva se zamjena slova `M` slovom `D`. Kada bi se dopustilo da se ta operacija provede, pokretanjem provođenja izmjene u znakovnom polju nad stringom `prvi` promjenio bi se i sadržaj stringa `drugi` jer on koristi isto znakovno polje. U Javi, to nije moguće: jednom stvoreni, svi su stringovi nepromjenjivi. Pažljivijim uvidom u dokumentaciju uočit ćemo da sve metode koje rade neku modifikaciju nad sadržajem stringa vraćaju novi string. Ovo je jasno vidljivo na stanju koje ćemo imati u memoriji nakon izvođenja retka 4, a koje je prikazano na slici u nastavku.

Slika 4.4. Stanje u memoriji (4)



U slučaju da je potrebno raditi niz izmjena nad stringovima, preporuča se zatražiti kopiju sadržaja stringa u obliku znakovnog polja pa tamo napraviti korekcije i potom napraviti novi string temeljem tog sadržaja. Alternativno, moguće je koristiti razred `StringBuilder` ili njegovu stariju verziju `StringBuffer`. Ovi razredi koriste se i prilikom dinamičke izgradnje stringova u postupcima koji rade puno konkatenacija.



Bilješka

U Javi stringovi su nepromjenjivi. Metode koje rade modifikacije vraćaju nove stringove s modificiranim sadržajem; originalni string se ne mijenja.

Nakon što smo ovo savladali, pogledajmo sada primjer dinamičke izgradnje stringova. Program u nastavku prikazuje četiri načina.

Primjer 4.5. Konkatenacija stringova

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 /**
 4  * @author Marko Cupic
 5  * @version 1.0
 6  */
 7 public class RadSaStringovima {
 8
 9  /**
10   * Metoda koja se poziva prilikom pokretanja
11   * programa. Argumenti su objasnjeni u nastavku.
12   * @param args Argumenti iz komandne linije.
13   */
14  public static void main(String[] args) {
15   ispis1();
16   ispis2();
17   ispis3();
18   ispis4();
19  }
20
21  /**
22   * Demonstracija zbrajanja stringova.<br>
23   * Zbrajanje uporabom operatora + kroz vise naredbi.
24   * Vrlo neefikasno!
25   */
```

```
26  private static void ispis1() {
27   String tekst = null;
28
29   tekst = "The quick " + "brown ";
30   tekst += "fox jumps over ";
31   tekst += 3;
32   tekst += " lazy dogs.";
33
34   System.out.println(tekst);
35  }
36
37  /**
38   * Demonstracija zbrajanja stringova.<br>
39   * Zbrajanje operatorom + u jednoj naredbi. Efikasnije.
40   */
41  private static void ispis2() {
42   String tekst = null;
43
44   int broj = 3;
45
46  tekst = "The quick brown fox jumps over " +
47                       broj + " lazy dogs.";
48
49   System.out.println(tekst);
50  }
51
52  /**
53   * Demonstracija zbrajanja stringova.<br>
54   * Zbrajanje uporabom StringBuffer objekta. Jednako efikasno
55   * kao i primjer 2? Inicijalno se stvara spremnik
56   * velicine 16 koji se tri puta realocira kako bi se prosirio.
57   * Napomena: prije Java 5.0 koristio se StringBuffer koji je
58   * bitno sporiji (ali je višedretveno siguran).
59   */
60  private static void ispis3() {
61   String tekst = null;
62
63   StringBuilder sb = new StringBuilder();
64
65   sb.append("The quick ").append("brown ");
66   sb.append("fox jumps over ").append(3);
67   sb.append(" lazy dogs.");
68
69   tekst = sb.toString();
70
71   System.out.println(tekst);
72  }
73
74  /**
75   * Demonstracija zbrajanja stringova.<br>
76   * Zbrajanje uporabom StringBuffer objekta. Najefikasnije
77   * ako unaprijed znamo potrebnu velicinu spremnika. U
78   * primjeru se alocira spremnik velicine 50 znakova.
79   * Napomena: prije Java 5.0 koristio se StringBuffer koji
80   * je bitno sporiji (ali je višedretveno siguran).
81   */
82  private static void ispis4() {
83   String tekst = null;
```

```
84    StringBuilder sb = new StringBuilder(50);
85
86    sb.append("The quick ").append("brown ");
87    sb.append("fox jumps over ").append(3);
88    sb.append(" lazy dogs.");
89
90    tekst = sb.toString();
91
92    System.out.println(tekst);
93   }
94 }
95
```

Prvi pokušaj prikazan je kroz metodu `ispis1` koja započinje u retku 26. Postupak kreće s praznim stringom. U retku 29, traži se konkatenacija sva stringa. Stoga će se u memoriji morati zauzeti novo znakovno polje na čiji će se početak iskopirati sadržaj prvog stringa i potom u nastavku sadržaj drugog stringa. Rezultat će biti novi objekt koji pokazuje na novo polje koje sadrži rezultat konkatenacije.

Redak 30 trenutni sadržaj konkatenira s predanim stringom. Postupak će opet biti isti: u memoriji će se zauzeti novo znakovno polje dovoljno za pohranu oba stringa; potom će se prvi string iskopirati na početak (čime već po drugi puta kopiramo po memoriji tekst `"The quick "` i potom će se nadodati sadržaj drugog stringa. U retku 30 postupak se još jednom ponavlja te u retku 31 još jednom. Svaki puta, u memoriji kopiramo sve više i više podataka i operacija postaje sve sporija i sporija.

Drugi pokušaj koji je dan u metodi `ispis2` koja započinje od retka 41 je efikasniji: svo se ljepljenje obavlja u okviru jedne naredbe. Iz razloga koji će nakon sljedećeg primjera postati jasni, ovo je bolje rješenje.

Ako je string potrebno izgraditi nizom konkatenacija, tada se isplati pozabaviti problemom neprestanih alokacija sve većeg i većeg polja koje će čuvati znakove koji pripadaju stringu. Razred `StringBuilder` uveden je upravo kao rješenje tog problema, a njegova uporaba ilustrirana je u metodi `ispis3`. Razred `StringBuilder` nudi metodu `append` kojom omogućava operaciju nadodavanje. Prilikom stvaranja objekta razreda `StringBuilder`, taj objekt u memoriji stvara pomoćno polje u koje će spremati sadržaj stringa i koje je kapaciteta 16 znakova. Svi pozivi metode `append` sadržaj nadodaju u to polje sve dok u njemu ima mjesta. Tek kada polje postane premalo, objekt kreće u njegovu realokaciju: zauzima se novo polje koje je dvostruke veličine u odnosu na trenutnu veličinu, sadržaj starog polja se kopira u novo polje i potom se provode daljnje konkatenacije na isti način. Uočimo da je politika povećanja veličine polja vrlo agresivna: kapacitet polja povećava se eksponencijalno čime će uz vrlo malo realokacija i kopiranja postići vrlo velike kapacitete, u situacijama u kojima je to potrebno. Jednom kada su sve konkatenacije obavljene, konačni string može se dobiti pozivom metode `toString`.

Konačno, kako bi se maksimalno smanjio broj nepotrebnih realokacija, razred `StringBuilder` omogućava i stvaranje prilikom kojega mu se preda željena početna veličina spremnika s kojim treba raditi. Ako imamo dobru procjenu konačne veličine stringa koji gradimo, na ovaj je način moguće obaviti postupak konkatenacije bez ijedne daljnje realokacije što je svakako najbrže moguće rješenje. To je upravo prikazano u metodi `ispis4` gdje se prilikom stvaranja objekta `StringBuilder` u retku 84 predaje kao inicijalna veličina spremnika vrijednost 50.

Osvrnimo se sada još na način na koji Java implementira konkatenaciju stringova: kada korisnik napiše sljedeći kod:

```
tekst = "The quick " + "brown " + "fox jumps over " +
3 + " lazy dogs.";
```

jezični prevodilac to prepiše u sljedeće:

```
tekst = new StringBuilder().append("The quick ")
                           .append("brown ")
                           .append("fox jumps over ")
                           .append(3)
                           .append(" lazy dogs.")
                           .toString();
```

Sada je jasan i komentar -- zašto je ljepljenje koje je svo napisano kao jedna naredba efikasnije od ljepljenja koje je napisano kao serija naredbi. U ovom posljednjem slučaju, jezični prevodilac će za svaku naredbu napisati po jedan izraz sličan prethodno navedenome.

Dobro, i sada se vjerojatno pitate, znači li to da nikada ne koristite operator + za konkatenaciju stringova? Odgovor je: apsolutno ne! Operator + koristite uvijek i svugdje osim u situacijama u kojima pišete kod koji bi mogao biti potencijalno usko grlo za performanse programa. Uporaba operatora + daje kod koji je bitno čitljiviji pa eksplicitnoj uporabi `StringBuilder`a treba priskočiti samo kada za to postoji valjano opravdanje.

Primjer 5: metode s varijabilnim brojem argumenata

Programski jezik Java omogućava definiranje funkcija s proizvoljnim brojem argumenata. Uporabu ćemo ilustrirati na primjeru dviju metoda. Razred `Matematika` i demonstracijski program `Primjer1` dani su u nastavku.

Primjer 4.6. Metode s varijabilnim brojem argumenata

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 /**
 4  * Ilustracija funkcija s varijabilnim brojem argumenata.
 5  *
 6  * @author marcupic
 7  */
 8 public class Matematika {
 9
10  /**
11   * Metoda prima 0 ili više cijelih brojeva i vraća njihovu
12   * sumu.
13   *
14   * @param brojevi 0 ili više cijelih brojeva
15   * @return suma
16   */
17  public static int suma(int ... brojevi) {
18   int s = 0;
19   for(int broj : brojevi) {
20    s += broj;
21   }
22   return s;
23  }
24
25  /**
26   * Metoda predani niz brojeva pretvara u string. String
27   * započinje lijevim simbolom, slijede predani brojevi
28   * koji su razdvojeni zarezima i završava desnim simbolom.
29   *
30   * @param lijeviSimbol simbol s kojim mora započeti string
31   * @param desniSimbol simbol s kojim mora završiti string
```

```
32   * @param brojevi niz brojeva koje treba pretvoriti u string
33   * @return string koji predstavlja predani niz brojeva
34   */
35  public static String kaoTekst(char lijeviSimbol,
36                    char desniSimbol, double ... brojevi) {
37   StringBuilder sb = new StringBuilder();
38   sb.append(lijeviSimbol);
39   boolean prvi = true;
40   for(double broj : brojevi) {
41    if(prvi) {
42     prvi = false;
43    } else {
44     sb.append(", ");
45    }
46    sb.append(broj);
47   }
48   sb.append(desniSimbol);
49   return sb.toString();
50  }
51
52  /**
53   * Metoda predani niz brojeva pretvara u string. String
54   * započinje znakom '{', slijede predani brojevi koji su
55   * razdvojeni zarezima i završava znakom '}'.
56   *
57   * @param brojevi niz brojeva koje treba pretvoriti u string
58   * @return string koji predstavlja predani niz brojeva
59   */
60  public static String kaoTekst(double ... brojevi) {
61   return kaoTekst('{', '}', brojevi);
62  }
63 }
64
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class Primjer1 {
 4
 5  public static void main(String[] args) {
 6   System.out.println(
 7    Matematika.kaoTekst('[', ']', 1, 4, 8.2, 13.1)
 8   );
 9   System.out.println(Matematika.kaoTekst('[', ']', 3.3));
10   System.out.println(Matematika.kaoTekst('[', ']'));
11
12   System.out.println(Matematika.kaoTekst(1, 4, 8.2, 13.1));
13
14   System.out.println("Suma je: " + Matematika.suma(2,4,12));
15  }
16
17 }
18
```

Razred `Matematika` definira tri metode koje ilustriraju deklariranje i uporabu metoda s varijabilnim brojem argumenata.

Metoda `suma` je primjer funkcije koja prima nula, jedan ili više cijelih brojeva te računa njihovu sumu. Metoda formalno definira jedan *varijabilni* argument koji predstavlja nula, jedan ili više elemenata tipa `int`; naziv argumenta na `brojevi`. Ovako definirane argumente Java tretira

kao polje čiji su elementi upravo zadanog tipa. U našem slučaju, argument `brojevi` predstavlja polje cijelih brojeva odnosno polje tipa `int[]`. U kodu, taj se argument stoga može koristiti kao i svako drugo polje: možemo se, primjerice, pitati koliko to polje ima elemenata pozivom `brojevi.length`. Stoga se izračun sume može napisati na sljedeći način.

```
int s = 0;
for(int i = 0; i < brojevi.length; i++) {
 double broj = brojevi[i];
 s += broj;
}
return s;
```

Ako nam, kao u ovom primjeru, eksplicitni indeks kojim prolazimo kroz elemente polja nije bitan, prikazanu `for`-petlju možemo zamijeniti njezinim kraćim ekvivalentom:

```
for(double broj : brojevi) {
 s += broj;
}
```

Ovaj oblik prikazan je i u kodu metode `suma`.

Sljedeća dva primjera su primjeri metoda čija je zadaća proizvoljni niz decimalnih brojeva pretvoriti u formatirani string: string mora započeti zadanim simbolom (npr. `'{'`), treba imati navedene sve brojeve razdvojene zarezom i jednom prazninom i treba završiti drugim zadanim simbolom (npr. `'}'`). Metoda `kaoTekst` čija je definicija dana od retka 35 kao argumente prima početni simbol, završni simbol te niz decimalnih brojeva. Metoda `kaoTekst` dana od retka 60 prima samo niz brojeva i eksplicitno kao graničnike koristi simbole otvorene i zatvorene vitičaste zagrade. U obje metode, varijabilni argument naveden je kao posljednji i u ovom slučaju ekvivalentan je polju decimalnih brojeva.

Rezultat pokretanja programa `Primjer1` dan je u nastavku.

```
[1.0, 4.0, 8.2, 13.1]
[3.3]
[]
{1.0, 4.0, 8.2, 13.1}
Suma je: 18
```

Metode s varijablim brojem argumenata možemo pozivati s nula, jednim ili većim brojem argumenata.



Definiranje metode s varijabilnim brojem argumenata

Metoda s varijabilnim brojem argumenata argument koji će prihvatiti predane vrijednosti mora deklarirati kao posljednji argument koristeći sintaksu `...`. Nije moguće definirati metodu koja ima dva varijabilna argumenta. Sljedeći primjeri su ispravni:

```
void metoda1(int x, double y, float ... z);

void metoda2(int x, int ... z);

void metoda3(double ... z);
```

dok primjeri u nastavku nisu.

```
void metoda4(float ... z, int x, double y);

void metoda1(double ...  y, float ... z);
```

Pogledajmo sada još jednu vrlo korisnu mogućnost programkog jezika Java: *statička uključivanja*. Pogledaje ponovno razred `Primjer1`. Svaki poziv metoda `suma` te `kaoTekst` morali smo prefiksirati nazivom razreda kojemu metode pripadaju, pa samo tako primjerice pisali `Matematika.suma(...)`. Uporabom statičkih uključivanja moguće je u izvorni kod uključiti definicije pojedinih statičkih metoda tako da ih prilikom korištenja ne trebamo prefiksirati nazivom razreda u kojem su definirane. Primjer je dan u nastavku.

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import static hr.fer.zemris.java.primjeri.Matematika.kaoTekst;
 4
 5 public class Primjer2 {
 6
 7  public static void main(String[] args) {
 8   System.out.println(kaoTekst('[', ']', 1, 4, 8.2, 13.1));
 9   System.out.println(kaoTekst('[', ']', 3.3));
10   System.out.println(kaoTekst('[', ']'));
11
12   System.out.println(kaoTekst(1, 4, 8.2, 13.1));
13
14   System.out.println("Suma je: " + Matematika.suma(2,4,12));
15  }
16
17 }
18
```

U retku 3 definirano je statičko uključivanje metode `kaoTekst` (uočite ključnu riječ `static` nakon naredbe `import`). Zbog toga u retcima 8, 9, 10 i 12 možemo direktno pozivati metodu `kaoTekst` bez navođenja razreda u kojemu je ona definirana. Kako isto nismo napravili s metodom `suma`, nju u retku 14 i dalje pozivamo punim imenom: `Matematika.suma`. Naredbama statičkog uključivanja moguće je uključivati jednu po jednu metodu. Želimo li uključiti sve statičke metode koje su definirane na razini jednog razreda, umjesto navođenja imena metode potrebno je navesti znak `*`. Konkretno, da bismo uključili sve statičke metode razreda `Matematika`, potrebno je napisati sljedeće.

```
import static hr.fer.zemris.java.primjeri.Matematika.*;
```

U tom slučaju, svaku bismo statičku metodu razreda `Matematika` mogli pozivati samo navođenjem njezinog imena, kao da se radi o metodi koja je definirana u trenutnom razredu.

Poglavlje 5. Razredi

Java je objektno-orijentirani programski jezik. Središnji pojmovi s kojima ovaj jezik barata su *razred* (koristi se još i termin klasa, engl. *class*) te *primjerak razreda* ili *objekt* (engl. *object*). Stoga je od presudnog značaja dobro se upoznati s ovim pojmovima. Što je razred? Što je primjerak razreda? Koja je motivacija za njihovo uvođenje i kako se oni koriste? Kako uvođenje razreda u programski jezik mijenja način oblikovanja programskog sustava? Na ova i slična pitanja započet ćemo odgovarati kroz ovo i sljedeća poglavlja.

Motivacijski primjer

Motivacijski primjer koji slijedi pretpostavlja da čitatelj vlada s razumijevanjem osnovnih pojmova proceduralnog programskog jezika C; konkretno, s ključnom riječi `struct`. Pogledajmo primjer modeliranja složenog tipa podatka koji bismo u programskom jeziku C mogli koristiti za predstavljanje različitih pravokutnika (primjerice, u nekom programu za rad s vektorskom grafikom). Definicija novog tipa podatka `pravokutnik` koji je struktura sastavljena od `poz_x` i `poz_y` koordinate gornjeg lijevog ugla pravokutnika, njegove širine `sirina` i visine `visina` te naziva `ime` prikazana je u primjeru 5.1.

Primjer 5.1. Modeliranje pravokutnika u programskom jeziku C

```
1 typedef struct pravokutnik_str {
2  int poz_x;
3  int poz_y;
4  int sirina;
5  int visina;
6  char *ime;
7 } pravokutnik;
8
```

Nakon što je definiran novi tip podatka, programer može u memoriji stvarati varijable tog tipa i dalje ih proslijeđivati drugim funkcijama na obradu, što će se najčešće, zbog postizanja boljih performansi, obavljati preko adrese (tj. pokazivača). Razmotrimo svojstva ovakvog pristupa.

1. Svatko tko ima pristup varijabli ovog tipa može dohvaćati vrijednosti svih podatkovnih članova te raditi njihovu modifikaciju. Nemamo na raspolaganju mehanizam kojim bismo klijentskom kodu onemogućili pristup do pojedinih informacija.

2. Nemamo na raspolaganju mehanizme kojima bismo mogli kontrolirati postavljaju li se pojedini podatkovni članovi na valjane vrijednosti. Primjerice, bilo tko tko ima pristup do varijable koja je po tipu `pravokutnik` može postaviti njezinu širinu na vrijednost -20.

Uočenim nedostatcima možemo pokušati doskočiti na nekoliko načina. Primjerice, mogli bismo svim korisnicima strukture `pravokutnik` objasniti da tu strukturu ne smiju direktno koristiti; potom bismo mogli napisati poseban skup funkcija čija je namjena dohvat i postavljanje vrijednost podatkovnih članova ali uz provođenje odgovarajućih provjera. U ove funkcije dodat ćemo odmah i funkcije za stvaranje novih pravokutnika te funkcije za njihovo oslobađanje iz memorije. Implementacije su prikazane u primjeru 5.2.

Primjer 5.2. Funkcije za rad s pravokutnicima u programskom jeziku C

```
1 pravokutnik *pravokutnik_novi(int x, int y, int s, int v,
2                               char *ime) {
3   pravokutnik *novi = (pravokutnik*)malloc(
4     sizeof(pravokutnik)
5   );
6   if(novi == NULL) return NULL;
```

```
 7   novi->ime = (char*)malloc(strlen(ime)+1);
 8   strcpy(novi->ime, ime);
 9   novi->poz_x = x;
10   novi->poz_y = y;
11   novi->sirina = s;
12   novi->visina = v;
13   return novi;
14 }
15
16 void pravokutnik_unisti(pravokutnik *p) {
17   if(p == NULL) return;
18   free(p->ime);
19   free(p);
20   return;
21 }
22
23 void pravokutnik_postavi_sirinu(pravokutnik *p, int s) {
24   if(s < 1) return;
25   p->sirina = s;
26   return;
27 }
28
29 int pravokutnik_dohvati_sirinu(pravokutnik *p) {
30   return p->sirina;
31 }
32
```

Pogledajmo najprije funkciju za stvaranje novih pravokutnika u memoriji: `pravokutnik_novi`. Kako se posao stvaranja novog pravokutnika sastoji od nekoliko koraka (zauzimanje potrebne memorije, inicijalizacija članskih varijabli), opravdano je, i dapače, dobra praksa, potreban kod napisati na jednom mjestu i potom ga pozivati gdje god je potrebno obaviti taj posao. Na taj način dobit ćemo kod koji je lakše održavati, a u slučaju da u postupku stvaranja novog pravokutnika postoji pogreška, tu ćemo pogrešku moći ispraviti promjenom koju je potrebno napraviti samo na jednom mjestu -- upravo u metodi koju koristimo za stvaranja pravokutnika.

Metoda prima pet argumenata koji su potrebni za inicijalizaciju jednog pravokutnika: koordinate gornjeg lijevog ugla pravokutnika, njegovu širinu i visinu te naziv koji mu je potrebno pridijeliti. U retku 3 obavlja se zauzimanje potrebnog memorijskog prostora funkcijom `malloc()`. Argument ove funkcije je broj okteta koji je potrebno zauzeti: tražimo upravo onoliko okteta koliko je potrebno za pohranu jednog pravokutnika a tu informaciju pribavljamo uporabom operatora `sizeof()`. S obzirom da pokušaj zauzeća memorije može biti neuspješan, u retku 6 provjerava se je li zauzimanje uspjelo i ako nije, metoda se prekida i pozivatelju vraća vrijednost `NULL`. Retci 7 do 12 rade inicijalizaciju članskih varijabli a redak 13 vraća pozivatelju pokazivač na novostvoreni pravokutnik.

Pažljivom analizom lako ćemo se uvjeriti da prikazana funkcija nije korektno napisana. Osim što ne provjerava postavljaju li se sve varijable na valjane vrijednosti (primjerice, širina na -20), nije čak niti dodana provjera je li zauzimanje memorije koje se radi u retku 7 za potrebe kopiranja imena pravokutnika uopće uspjelo! Ova greška namjerno je ostavljena kako bi se ilustriralo koliko je lagano zaboraviti na sve potrebne provjere i time generirati kod koji nije korektan. Ako je ostatak programa pisan tako da je svaki puta kada je potrebno stvoriti novi pravokutnik pozvana ova metoda, korigiranje će biti lagano. Ako je pak ovakav kod pisan na svakom mjestu na kojem je bio potreban novi pravokutnik ispočetka, provođenje korekcija na svim takvim mjestima može postati vrlo zamorno.

Počev od retka 16 dan je i programski kod funkcije `pravokutnik_unisti` čija je namjena oslobađanje memorije koja je potrošena za pravokutnik čiju adresu funkcija dobiva kao prvi argument.

Funkcija `pravokutnik_postavi_sirinu` koja započinje u retku 23 služi za kontrolirano postavljanje širine pravokutnika. Funkcija sadrži dio koda koji provjerava je li iznos širine koji se pokušava postaviti valjan i ako nije, funkcija odbija provesti traženu promjenu. Ako je sve u redu, širina pravokutnika se ažurira. Kako bi se ovo moglo provesti, funkcija mora dobiti dva argumenta: pokazivač na pravokutnik koji je potrebno ažurirati te željeni novi iznos širine koji treba upisati.

Komentirajmo usput način na koji je rješeno čuvanje konzistentnosti podataka o pravokutniku. Prethodna metoda, ako joj se preda širina koja nije valjana, odbija provesti ažuriranje kako se pravokutnik ne bi doveo u nekonzistentno stanje. To je pozitivno. Međutim, način na koji se postupa u tom slučaju je loš. Metoda je implementirana tako pokušaj postavljanja nevaljane širine potiho ignorira. Klijentski kod koji je pozvao ovu metodu o tome neće biti obaviješten i ovaj pokušaj (koji je očito rezultat pogreške u programu) još će neko vrijeme ostati neprimijećen -- dok problem ne nastupi kasnije u nekom potpuno drugom dijelu koda koji će očekivati da je prethodno ažuriranje uspjelo. Ograničimo li se na programski jezik C, navedenu funkciju trebalo bi promijeniti tako da pozivatelju vraća informaciju je li ažuriranje provedeno ili nije (primjerice, deklariranjem da funkcija vraća vrijednost tipa `int`, pa za slučaj da se ažuriranje ne provede vrati `0` a inače bilo što različito od `0`). U modernijim programskim jezicima, opisani scenarij je tipičan primjer u kojem bi se programski izazvala iznimna situacija (takozvana *iznimka*).

Konačno, počev od retka 29 dana je implementacija funkcije `pravokutnik_dohvati_sirinu` čija je zadaća dohvaćanje trenutne vrijednosti širine pravokutnika čiju adresu u memoriji dobiva kao prvi i jedini argument.

Pretpostavimo da korisnici pravokutnika za rad s pravokutnicima koriste isključivo metode koje smo napisali. Koje su karakteristike opisanog rješenja?

1. Postoji metoda za stvaranje pravokutnika, koja prima argumente potrebne za proces stvaranja. Ta metoda zauzima u memoriji prostor za jedan pravokutnik i potom obavlja inicijalizaciju članskih varijabli.

2. Postoji metoda za uništavanje pravokutnika, koja prima pokazivač na pravokutnik. Zadaća metode je osloboditi sve resurse koje kroz članske varijable koristi pravokutnik a koje je zauzeo postupak inicijalizacije ili druge metode koje su direktno pozivali klijenti. U opisanom primjeru, potrebno je otpustiti memoriju koja je bila zauzeta za čuvanje podataka o imenu pravokutnika. Nakon što je to obavljeno, zadaća metode je i oslobađanje memorije koju su zauzimale članske varijable pravokutnika.

3. Postoje metode za dohvat informacija vezanih uz stanje pravokutnika koje primaju pokazivač na pravokutnik a vraćaju traženu informaciju. Primjer takve metode je metoda `pravokutnik_dohvati_sirinu` koja vraća širinu pravokutnika čiju adresu dobiva preko pokazivača.

4. Postoje metode za ažuriranje stanja pravokutnika koje primaju pokazivač na pravokutnik te druge dodatne argumente koji su potrebni kako bi se stanje ažuriralo. Primjer takve metode je metoda `pravokutnik_postavi_sirinu` koja dobiva adresu pravokutnika kao prvi argument te željenu širinu kao drugi argument. Ove metode omogućavaju provođenje kontrole konzistentnosti pravokutnika i osiguravaju da se pravokutnik ne može dovesti u nekonzistentno stanje.

Pogledajmo sada što smo postigli ovakvom organizacijom koda.

1. Smanjuje se mogućnost pojave pogreške u kodu i olakšava se ispravljanje pogrešaka. Primjerice, funkcija za stvaranje napisana je samo na jednom mjestu; ako ona sadrži pogrešku, ispravljanjem te pogreške automatski ćemo ispraviti problem koji su imali svi klijenti koji su pozivali tu funkciju.

2. Podatkovni članovi neće biti postavljeni na nekonzistentne vrijednosti. S obzirom da se za ažuriranje vrijednosti koriste za to zadužene metode, provjeru konzistentnosti podataka mo-

guće je ugraditi na jednom mjestu i ona automatski postaje djelotvorna za sve klijente koji pokušavaju raditi ažuriranje. Ako, kao što je često slučaj s kodom, dođe do promjene u definiciji konzistentnosti podataka, nova pravila opet je moguće lagano implementirati jer ih treba dodati na samo jedno mjesto u kodu.

3. Opisana organizacija koda u određenom smislu izolira klijente koji rade s pravokutnicima od implementacijskih detalja vezanih uz pohranu informacija u memoriji. Primjerice, ako je iz nekog razloga potrebno preimenovati varijable definirane u okviru strukture `pravokutnik`, to će se moći provesti jednostavno i bit će potrebno još modificirati samo prethodno opisane metode -- niti jedan dio klijentskog koda neće se morati mijenjati. Također, ako dođe do promjena interne strukture pravokutnika (primjerice, više ne pamtimo širinu već pamtimo koordinate donjeg desnog ugla pravokutnika), definirane metode izolirat će krajnje klijente od tih promjena (metodu `pravokutnik_postavi_sirinu` moći ćemo prilagoditi tako da izračuna `x` koordinatu donjeg desnog ugla uz koju bi pravokutnik imao traženu širinu i da potom tako ažurira tu koordinatu).

Iz prethodno navedenoga vidimo da ovakav pristup nudi mnoge prednosti. No ostaje nam neriješeni problem: kako osigurati da svatko koristi te metode?

Navedeni problem pri tome nije jedini. Pogledajmo još jedan značajan problem. Kako postići specijalizaciju strukture, ili dodavanje novih polja strukturi? Primjerice, zamislimo da u nekom trenutku želimo definirati `Kvadrat` koji je, u određenom smislu, specijalni pravokutnik -- visina i širina su mu isti. Ili, kako izreći sljedeće: "ja imam sve elemente kao i struktura `X`, i dodatno imam još elemente ..."? Ovakve operacije u nižim programskim jezicima tipično nisu podržane. Što naravno ne znači da ne možemo pokušati tako nešto ostvariti. Primjerice, zamislimo da uz obične pravokutnike želimo dodati i obojane pravokutnike. Obojani pravokutnik je pravokutnik koji dodatno ima još jednu člansku varijablu: boju. S druge strane, mogli bismo imati i rotirane pravokutnike koji su pravokutnici koji dodatno imaju još jednu člansku varijablu: kut rotacije. Jedan način koji odmah pada napamet jest pristup "kopiraj-i-zalijepi" prikazan u nastavku.

Primjer 5.3. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 1

```
 1 typedef struct pravokutnik_str {
 2  int poz_x;
 3  int poz_y;
 4  int sirina;
 5  int visina;
 6  char *ime;
 7 } pravokutnik;
 8
 9 typedef struct pravokutnik_o_str {
10  int poz_x;
11  int poz_y;
12  int sirina;
13  int visina;
14  char *ime;
15  int boja;
16 } pravokutnik_o;
17
18 typedef struct pravokutnik_r_str {
19  int poz_x;
20  int poz_y;
21  int sirina;
22  int visina;
23  char *ime;
24  double kut;
25 } pravokutnik_r;
```

```
26
```

Opisani pristup, nažalost, stvara mnoštvo problema. Teoretski, kako je početak struktura `pravokutnik_o` i `pravokutnik_r` jednak strukturi `pravokutnik`, za čitanje i ažuriranje vrijednosti definiranih na razini osnovnog pravokutnika mogli bismo koristiti već napisane funkcije, uz nešto ružnog ukalupljivanje pokazivača prilikom poziva. Međutim, što ako u osnovni pravokutnik trebamo dodati neki novi podatak koji bi potom trebao biti vidljiv i u obojanom i u rotiranom pravokutniku, poput stila kojim obrub pravokutnika treba biti nacrtan (puna linija, istočkana linija, crtkana linija i slično)? U strukturu `pravokutnik` dodali bismo odgovarajuću novu člansku varijablu. Međutim, strukture `pravokutnik_o` i `pravokutnik_r` tu promjenu ne bi automatski uvažile -- iako konceptualno povezane, u našem kodu to su samo tri različite strukture sličnih imena i sličnog početka. Vezu: "*rotirani pravokutnik* je *pravokutnik* koji dodatno ima rotaciju" morali bismo održavati sami tako da na isto mjesto dodamo potrebnu člansku varijablu i u tu strukturu i u sve druge strukture koje *jesu* pravokutnici. Ako to zaboravimo napraviti, u kod ćemo unijeti pogrešku koju će kasnije biti vrlo teško pronaći.

Opisani način nije jedini na koji smo mogli postupiti. Evo drugog pokušaja.

Primjer 5.4. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 2

```
 1 typedef struct pravokutnik_str {
 2  int poz_x;
 3  int poz_y;
 4  int sirina;
 5  int visina;
 6  char *ime;
 7 } pravokutnik;
 8
 9 typedef struct pravokutnik_o_str {
10  pravokutnik p;
11  int boja;
12 } pravokutnik_o;
13
14 typedef struct pravokutnik_r_str {
15  pravokutnik p;
16  double kut;
17 } pravokutnik_r;
18
```

Ono što smo sada postigli u određenoj mjeri rješava prethodni problem no konceptualno nije čisto. Upravo smo definirali obojani pravokutnik koji zapravo baš i nije pravokutnik već je nešto što interno sadrži pravokutnik i sadrži boju kojom treba obojati taj pravokutnik. A isto možemo reći i za rotirani pravokutnik. Također, pogledamo li kako se pristupa članskim varijablama tog pravokutnika, brzo ćemo uočiti nelogičnost: ako je `x` pokazivač na jedan obojani pravokutnik, tada boju dobivamo izrazom `x->boja` dok širinu dobivamo izrazom `x->p.sirina`. Kada bismo sada još malo poopćili priču definiranjem posebne vrste obojanih pravokutnika pa potom definiranjem još neke posebne vrste od prethodno definirane posebne vrste obojanih pravokutnika, jasno je da bi izraz za pristupanje različitim svojstvima postajao sve nepregledniji i možda još gore, programer bi neprestano morao razmišljati koje je svojstvo definirano gdje i na koji mu se način pristupa.

Dobrog rješenja za uočeni problem, u jezicima ove razine, nažalost, nemamo.



Istaknuti problemi

Ovaj motivacijski primjer trebao je ilustrirati dva najvažnija problema koji su motiv za uvođenje i prelazak na objektno-orijentiranu paradigmu:

- problem pronalaska načina kako osigurati konzistentnost podataka -- trebamo mogućnost skrivanja i zaštite podataka od korisnika te mehanizam nuđenja metoda koje klijenti legalno smiju pozivati kako bi obavili određenu zadaću te,
- uvođenje mehanizma nasljeđivanja članskih varijabli i metoda koje bi osigurale da izvedene strukture automatski postanu svjesne svih promjena načinjenih u osnovnoj strukturi i koje bi pri tome klijentima nudile jasan i unificiran način pristupanja do svih članskih varijabli, neovisno o tome jesu li one definirane u osnovnoj ili u izvedenoj strukturi.

Java je objektno-orijentirani programski jezik

Programski jezik Java ne poznaje pojam strukture kako smo ga opisali u jeziku C. Umjesto toga, Java poznaje pojam *razreda*. U određenom smislu, na razred možemo gledati kao na poopćenu strukturu programskog jezika C: to je struktura koja uz definiranje podatkovnih članova omogućava i definiranje metoda koje mogu raditi s tim podatkovnim članovima, omogućava zaštitu odnosno definiranje prava pristupa članskim varijablama i metodama te omogućava nasljeđivanje. Umjesto ključne riječi `struct` koja u Javi ne postoji, razredi se definiraju uporabom ključne riječi `class`.

Baš kao što smo u motivacijskom primjeru definirali niz korisnih metoda, tako i jezici koji podržavaju objektno-orijentiranu paradigmu omogućava da definiramo slične metode. Neke od njih imaju čak i posebna imena.

Kako bismo lakše ovladali općenitim konceptima koji nisu uska specifičnost pojedinih jezika, u nastavku su dane definicije razreda `Pravokutnik` u tri programska jezika: Java (primjer 5.5), C++ (primjer 5.6) te Python (primjer 5.7). Zbog uštede na prostoru, razredi nisu opremljeni svim potrebnim metodama ali bi iz priloženog trebalo biti jasno kako nadopuniti ono što nedostaje.

Primjer 5.5. Definiranje razreda `Pravokutnik`: Java

```
 1 class Pravokutnik {
 2
 3  private int x;
 4  private int y;
 5  private int sirina;
 6  private int visina;
 7  private String ime;
 8
 9  public Pravokutnik(int x, int y, int sirina, int visina,
10                       String ime) {
11   this.x = x;
12   this.y = y;
13   this.sirina = sirina;
14   this.visina = visina;
15   this.ime = ime;
16  }
17
18  protected void finalize() {
19   System.out.println("Izvodi se prije uništavanja objekta.");
20  }
21
22  public int dohvatiSirinu() {
23   return sirina; // ili return this.sirina;
24  }
25
26  public void postaviSirinu(int sirina) {
27   this.sirina = sirina;
```

```
28  }
29 }
30
```

Primjer 5.6. Definiranje razreda `Pravokutnik`: C++

```
 1 class Pravokutnik {
 2
 3  private:
 4
 5  int x;
 6  int y;
 7  int sirina;
 8  int visina;
 9  std::string *ime;
10
11  public:
12
13  Pravokutnik(int x, int y, int sirina, int visina,
14                  std::string ime) {
15   this->x = x;
16   this->y = y;
17   this->sirina = sirina;
18   this->visina = visina;
19   this->ime = new std::string(ime);
20  }
21
22  ~Pravokutnik() {
23   delete ime;
24   std::cout << "Izvodi se prilikom uništavanja objekta"
25             << std::endl;
26  }
27
28  int dohvatiSirinu() {
29   return sirina; // ili return this->sirina;
30  }
31
32  void postaviSirinu(int sirina) {
33   this->sirina = sirina;
34  }
35
36 };
37
```

Primjer 5.7. Definiranje razreda `Pravokutnik`: Python

```
 1 class Pravokutnik:
 2
 3  def __init__(self, x, y, sirina, visina, ime):
 4   self.x = x
 5   self.y = y
 6   self.sirina = sirina
 7   self.visina = visina
 8   self.ime = ime
 9
10  def __del__(self):
11   print "Izvodi se prilikom uništavanja objekta."
12
```

```
13  def dohvatiSirinu(self):
14   return self.sirina
15
16  def postaviSirinu(self, sirina):
17   self.sirina = sirina
18
```

Konstruktori

Posao zauzimanja memorije potrebne za primjerak razreda i posao inicijalizacije uobičajeno je razdvojen. Zauzimanje memorije za primjerak razreda (još ćemo ga zvati sinonimom *objekt*) obavlja se ili automatski na stogu (kod jezika koji to dozvoljavaju, primjerice C++) ili uporabom posebnog operatora koji će memoriju zauzeti na *heapu* i koji se tipično naziva `new`. Jednom kada je za objekt zauzeta memorija (na bilo koji način), poziva se metoda koja je zadužena za inicijalizaciju objekta. Ova metoda automatski će dobiti pokazivač na memorijski prostor koji je zauzet za objekt. U našem motivacijskom primjeru imali smo funkciju zaduženu za stvaranje pravokutnika koja je prvo zauzela potreban memorijski prostor pozivom funkcije `malloc` i potom ga inicijalizirala. U programskim jezicima poput C++ i Java, zauzimanje memorije obavlja se kao izdvojeni korak a za inicijalizaciju tako zauzete memorije zadužene su metode poznate pod nazivom *konstruktori*.

U primjeru 5.5, konstruktor je prikazan počev od retka 9. Programski jezik Java zahtjeva da se konstruktori zovu identično kao što se zove i razred te da nemaju definiran tip povratne vrijednosti. Također, konstruktor će, baš kao i sve ostale metode o kojima ćemo u nastavku pričati, implicitno dobivati referencu (u Javi ne koristimo pojam pokazivač već referenca) na prethodno zauzeti memorijski prostor. Ta se referenca prenosi kao prvi skriveni argument. Zbog toga se u popisu argumenata konstruktora eksplicitno navode samo dodatni argumenti koji su potrebni kako bi se prikupilo dovoljno informacija temeljem kojih se može provesti inicijalizacija članskih varijabli. U tijelu metode, ta je referenca dostupna preko ključne riječi `this`. Tako naredba prikazana u retku 11 članskoj varijabli `x` novozauzetog objekta pridružuje vrijednost argumenta `x`. Kada se članske varijable i argumenti ne zovu isto, tada se ključna riječ `this` ne treba niti pisati jer će biti podrazumijevana. Uočimo, u Javi se članskim varijablama preko reference pristupa uporabom znaka točka. Java za takve stvari ne koristi `->` kao što je slučaj s programskim jezicima C i C++.

U primjeru 5.6, konstruktor je prikazan počev od retka 13. U programskom jeziku C++ koristi se isti mehanizam kao i u Javi: kao prvi skriveni parametar u pozivu konstruktora prenosi se pokazivač na memorijski prostor koji je zauzet za novostvoreni objekt. Taj je pokazivač dostupan uporabom ključne riječi `this` i on je sa stajališta jezika pokazivač; stoga se preko njega do članskih varijabli dolazi na način koji je uobičajen za pokazivače: uporabom `->`.

Konačno, u primjeru 5.7 u retku 3 je prikazana metoda koja, što se Pythona tiče, konceptualno ima ulogu konstruktora: ta će metoda biti pozvana nakon što se zauzme prostor u memoriji za novi objekt kako bi se dovršile inicijalizacije tog objekta. U Pythonu, za razliku od programskih jezika Jave i C++, potrebno je eksplicitno definirati da metoda ima argumente i da je prvi argument upravo referenca na novostvoreni objekt. Prema uobičajenoj konvenciji, ta se referenca naziva `self` (iako može i drugačije) i članskim se varijablama uvijek pristupa preko nje. Pri tome se kao i u Javi koristi znak točke. Članske varijable koje će postojati definiraju se direktno u metodi `__init__` prvi puta kada im se pridruži vrijednost. Iz tog razloga, u definicijama razreda u jezicima Java i C++ postoji dio koda u kojem su članske varijable eksplicitno definirane i tipizirane dok to nije slučaj s Pythonom.

Pogledajmo sada i isječak koda koji će stvoriti dva primjerka razreda `Pravokutnik` u Javi i C++. Najprije navodimo primjer za Javu.

```
void metoda() {
  Pravokutnik p1 = new Pravokutnik(1,1,4,4,"prvi");
  Pravokutnik p2 = new Pravokutnik(2,2,8,8,"drugi");
  p1.postaviSirinu(3);
```

```
  System.out.println(p2.dohvatiSirinu());
}
```

Kako je u Javi nemoguće stvarati primjerke razreda na stogu, varijable koje su po tipu razredi automatski su reference. Stoga se u Javi nikada ne piše nešto poput `Pravokutnik *p1 = ...`; znak `*` ne koristi se za definiranje pokazivača i njegova uporaba na takav način rezultirat će sintaksno neispravnim programom. U danom primjeru definiraju se dvije lokalne varijable koje su po tipu reference. Uočite kako koristimo operator `new`: nakon navođenja operatora `new` navodi se poziv konstruktora koji će obaviti inicijalizaciju objekta. Operator `new` će pri tome odraditi posao zauzimanja memorije i potom pozvati definirani konstruktor pri čemu će uz navedene argumente koji su dani poslati i referencu na memoriju koju je zauzeo za novostvoreni objekt. U konstruktoru, ta će referenca biti vidljiva kao `this`. Konceptualno, redak:

```
  Pravokutnik p1 = new Pravokutnik(1,1,4,4,"prvi");
```

možemo gledati kao da u pseudokodu piše:

```
  Referenca r = zauzmiMemoriju(sizeof(Pravokutnik));
  Pravokutnik(r,1,1,4,4,"prvi");
  Pravokutnik p1 = ref;
```

Rezultat izvođenja prva dva retka početnog primjera jest zauzimanje memorije za dva pravokutnika i njihova inicijalizacija. U retku 3 čini se kao da pozivamo metodu `postaviSirinu` nad jednim od stvorenih pravokutnika. Iako to na konceptualnoj razini možemo gledati na taj način, istina je ipak malo drugačija. Bilo bi vrlo neefikasno kada bi svaki primjerak razreda prilikom stvaranja dobio i svoju vlastitu kopiju sveg strojnog koda koji će raditi nad njim. U stvarnosti, u memoriji računala će postojati samo jedna kopija strojnog koda metode `postaviSirinu`. Međutim, ona neće imati samo jedan argument kao što je prikazano u pozivu te metode ili kao što smo je, uostalom, i mi sami definirali. Implementacijski, ta će se metoda ponašati kao da je bila definirana ovako:

```
void postaviSirinu(Pravokutnik this, int sirina);
```

što je upravo razlog iz kojeg nam je referenca `this` dostupna u tijelu te metode. Naš poziv metode kojim metodu `postaviSirinu` zovemo nad objektom na koji pokazuje referenca `p1` prevodilac će prilikom generiranja strojnog koda pretvoriti u poziv oblika:

```
postaviSirinu(p1, 3);
```

čime će osigurati da se doista ažurira članska varijabla `sirina` pravokutnika na koji pokazuje referenca `p1`. Važno je biti svjestan ovog mehanizma jer je to način na koji se izvodi objektno-orijentirana magija: privid da metode pripadaju objektima i rade upravo nad članskim varijablama objekta samo je vješta manipulacija jezičnog prevoditelja. Nasreću, to je manipulacija od koje ćemo imati puno koristi, i jednom kada razumijemo kako ona radi, bit će nam sasvim prirodno razmišljati u okvirima programskog jezika koji nam omogućava da metode pripadaju i djeluju nad objektima.

U programskom jeziku C++ pravila su slična, posebice ona vezana uz način poziva metode nad objektom koji jezični prevoditelj prevodi u poziv odgovarajuće metode s pokazivačem na objekt kao prvim parametrom. primjer koda prikazan je u nastavku.

```
void metoda() {
  Pravokutnik *p1 = new Pravokutnik(1,1,4,4,"prvi");
  Pravokutnik p2 = Pravokutnik(2,2,8,8,"drugi");
  p1->postaviSirinu(3);
  System.out.println(p2.dohvatiSirinu());
  delete p1;
}
```

Za razliku od Jave, jezik C++ nam omogućava stvaranje objekata i na *heapu* i na stogu. Stoga su u primjeru prikazana oba načina. Prvi primjerak pravokutnika stvara se na *heapu*; memoriju zauzima operator `new` i potom se poziva konstruktor kako bi obavio inicijalizaciju. U drugom

retku prikazano je stvaranje objekta na stogu. U ovom slučaju ne postoji poziv operatora `new` jer se prostor automatski zauzima na stogu; drugi korak je opet isti: poziva se konstruktor kako bi obavio inicijalizaciju novog objekta. Metoda `postaviSirinu` poziva se sintaksom `p1->postaviSirinu(3);` jer je `p1` pokazivač. Metoda `dohvatiSirinu` poziva se preko točke jer je `p2` lokalni objekt koji živi na stogu a ne pokazivač.

Destruktori

Sljedeća vrsta metoda su takozvani *destruktori*. Destruktori su metode zadužene za oslobađanje svih resursa koje je tijekom života zauzeo primjerak razreda, osim memorije koju fizički zauzimaju članske varijable samog primjerka. Vratimo li se na naš motivacijski primjer, metoda `pravokutnik_unisti` (osim retka 19 u kojem se fizički oslobađa i memorija zauzeta za sam objekt) predstavljala bi jedan primjer destruktora. Konkretno, s obzirom da je prilikom konstrukcije objekta zauzeta memorija za čuvanje znakovnog niza koji predstavlja naziv pravokutnika (članska varijabla `ime` čuva adresu početka bloka memorije koji je zauzet u tu svrhu), ne smije se dopustiti da se prilikom uništavanja pravokutnika samo oslobodi memorija koja je zauzeta za čuvanje članskih varijabli pravokutnika jer bismo time imali problem curenja memorije -- memoriju zauzetu za čuvanje imena nitko više ne bi niti mogao osloboditi jer bismo izgubili jedini pokazivač na taj blok memorije. Zadaća destruktora je osloboditi sve zauzete resurse kako nakon oslobađanja memorije koju je trošio sam objekt ne bi došlo do curenja resursa. U primjeru destruktora pravokutnika to radi upravo redak 18.

U programskom jeziku Java destruktori ne postoje. Java nam ne omogućava direktno oslobađanje memorije jer je praksa pokazala da jezici koji nude takvu mogućnost omogućavaju programerima da rade čitav niz pogrešaka koje je vrlo teško otkriti, koje se manifestiraju kao nedeterministički rad programa i rezultiraju povremenim rušenjem programa ali na način koji je često teško reproducirati. U konačnici, osnovni uzrok takvog ponašanja je pristupanje memorijskoj lokaciji na koju i dalje imamo pokazivač ali koju smo u međuvremenu oslobodili. Java programerima onemogućava uvođenje takvih pogrešaka na način da ne nudi mogućnost direktnog oslobađanja memorije. Umjesto toga, Java koristi automatsko upravljanje memorijom uporabom *sakupljača smeća*: to je podsustav koji prati rad programa i periodički oslobađa sve resurse na koje je program izgubio reference. Zahvaljujući tome, tako dugo dok program ima referencu na objekt zauzet u memoriji, taj objekt neće biti oslobođen pa je nemoguće pristupati memoriji koja je oslobođena i time načiniti opisanu pogrešku. S druge pak strane, jednom kada program izgubi referencu na objekt, objekt postaje smeće i sakupljač smeća će ga u nekom trenutku reciklirati (odnosno osloboditi memoriju koju je on zauzimao). U slučaju da programu nikada ne usfali memorije, moguće je da se sakupljač smeća nikada ne aktivira pa tako u Javi nemamo garanciju da će se jednom zauzeti objekti ikada osloboditi tijekom života programa. A kada program završi s radom, program se terminira i njegov se adresni prostor uništava od strane operacijskog sustava.

Iz opisanih razloga, u programskom jeziku Java destruktori ne postoje. Najbliže što postoji su takozvani *finalizatori*. Finalizator je metoda čiji je prototip:

```
protected void finalize();
```

. Specifikacija programskog jezika Java garantira da će se nad svakim objektom jednom kada on postane smeće a prije no što se objekt izbriše iz memorije pozvati njegov finalizator. Pažljivim čitanjem prethodne rečenice potrebno je uočiti da jezik ne garantira da će se za svaki stvoreni objekt prije kraja programa pozvati njegov finalizator. Naime, finalizatore poziva sakupljač smeća prije no što počisti objekt. Ako program ne troši previše memorije, moguće je da se sakupljač smeća nikada ne pokrene a time i da se finalizatori nikada ne pozovu. Što opet nije problem jer će svi resursi koje je zauzimao program tijekom izvođenja ionako biti oslobođeni kada se program terminira. Izvorno, finalizatori su u Javi bili zamišljeni kao mehanizam kojim bi se omogućilo objektima da prije uništavanja oslobode resurse kojima su upravljali (primjerice, da zatvore korištene datoteke i slično). Danas se ovaj mehanizam u praksi gotovo i ne koristi i rijetko ćemo se s njime susretati. U situacijama u kojima će to biti slučaj, valja zapamtiti da se finalizatori pozivaju iz nove dretve pa njihova uporaba naš program automatski pretvara u višedretveni, sa svim posljedicama koje to povlači.

Za razliku od Jave, programski jezik C++ uvelike se oslanja na uporabu destruktora. Sve što objekt zauzme, destruktor bi morao osloboditi prije no što se njegova memorija oslobodi. U primjeru 5.6, destruktor je prikazan u retku 22. Destruktor, kao i konstruktor, nije direktno zadužen za oslobađanje memorije koju je zauzeo sam objekt. I baš kao i konstruktor, destruktor će kao skriveni parametar dobiti pokazivač na objekt koji treba *deinicijalizirati*: taj je pokazivač dostupan kroz ključni riječ `this` i ako nema kolizije s lokalno definiranim varijablama ili argumentima, podrazumijevani je. Prilikom konstrukcije pravokutnika, konstruktor je u retku 19 zauzeo memoriju za čuvanje imena pravokutnika. Stoga destruktor ima zadaću osloboditi tu memoriju što je prikazano u retku 23. Oslobađanje memorije obavlja se primjenom operatora `delete`.

Vratimo se sada na isječak koda koji je prikazao stvaranje dva pravokutnika u jeziku C++: jedan je bio stvoren uporabom operatora `new` na *heapu* (`p1`) dok je drugi bio stvoren lokalno na stogu (`p2`). Kako ne bi došlo do curenja memorije, prikazana metoda u zadnjem retku sadrži poziv `delete p1;`. Operator `delete` u jeziku C++ služi za oslobađanje objekata iz memorije. Ta naredba pokreće dva postupka: najprije se za predani objekt poziva njegov destruktor; jednom kada je destruktor oslobodio sve resurse koje je objekt koristio, operator delete oslobađa i blok memorije koji je zauzimao sam objekt. U slučaju objekta `p2` oslobađanje se ne radi uporabom operatora `delete` jer je objekt bio stvoren lokalno na stogu. Umjesto toga, pri izlasku iz metode, a prije no što se oslobodi prostor koji je na stogu bio zauzet za objekt, automatski će se pozvati destruktor objekta. Opisani scenarij upravo predstavlja razlog zbog kojeg destruktori nemaju zadaću oslobađanja memorije objekta -- objekt je mogao biti stvoren na različite načine i zbog toga njegova memorija može biti oslobođena na različite načine. Destruktor se samo mora pobrinuti da oslobađanje memorije objekta ne dovede do curenja resursa.

Konačno, primjer metode koja je po namjeni zapravo sličnija Javinom finalizatoru nego C++-ovom destruktoru u primjeru danom u jeziku Python prikazana je u retku 10: to je metoda specifičnog imena `__del__` i kao što je to inače slučaj u Pythonu, iz prototipa metode jasno je vidljivo da metoda kao prvi argument dobiva referencu na objekt koji treba deinicijalizirati.

Delegiranje zadaće konstrukcije objekta

Od ovog trenutka na dalje, u fokusu će biti programski jezik Java i primjere ćemo prikazivati isključivo u Javi. U primjeru 5.5, prikazana je definicija razreda `Pravokutnik` koja ima samo jedan konstruktor. Razred, međutim, može navesti definicije i više od jednog konstruktora. Pogledajmo primjer 5.8.

Primjer 5.8. Definiranje više konstruktora

```
 1 class Pravokutnik {
 2
 3  private int x;
 4  private int y;
 5  private int sirina;
 6  private int visina;
 7  private String ime;
 8
 9  public Pravokutnik(int x, int y, int sirina, int visina,
10                     String ime) {
11   this.x = x;
12   this.y = y;
13   this.sirina = sirina;
14   this.visina = visina;
15   this.ime = ime;
16  }
17
18  public Pravokutnik(int x, int y,
19                     String ime) {
20   this.x = x;
21   this.y = y;
```

```
 22   this.sirina = 1;
 23   this.visina = 1;
 24   this.ime = ime;
 25  }
 26
 27  public Pravokutnik(int x, int y) {
 28   this.x = x;
 29   this.y = y;
 30   this.sirina = 1;
 31   this.visina = 1;
 32   this.ime = "anonimni";
 33  }
 34 }
 35
```

U novom primjeru definirali smo čak tri konstruktora. Konstruktor prikazan u retku 9 je najopćenitiji mogući konstruktor: on preko argumenata prima vrijednosti za sve članske varijable i postavlja ih u skladu s predanim vrijednostima argumenata. Konstruktor u retku 18 prima samo koordinate gornjeg lijevog ugla pravokutnika i željeno ime; širinu i visinu postavlja na podrazumijevanu vrijednost 1. Konačno, konstruktor prikazan u retku 27 prima samo koordinate gornjeg lijevog ugla pravokutnika; širinu i visinu postavlja na vrijednost 1 a ime na vrijednost "anonimni". Ako razred ima više konstruktora, na koji se način određuje koji će biti pozvan? Problema nema jer će uz poziv operatora `new` biti predane sve potrebne informacije. Evo primjera.

```
Pravokutnik p1 = new Pravokutnik(5, 5, 10, 12, "prvi");
Pravokutnik p2 = new Pravokutnik(2, 2, "drugi");
Pravokutnik p3 = new Pravokutnik(2, 10);
```

Temeljem broja i vrste predanih argumenata, nakon poziva operatora `new` jasno je koji od definiranih konstruktora treba pozvati.

Pogledamo li sada malo detaljnije napisani kod, uočit ćemo da sva tri konstruktora dijele dosta zajedničkog koda. Kako je jedno od zlatnih pravila programiranja da dupliciranje koda treba izbjegavati, postavlja se pitanje -- možemo li konstruktore napisati bolje? Odgovor je dakako potvrdan. Java nam omogućava da jedan konstruktor posao stvaranja delegira drugom konstruktoru. Rješenje je prikazano u primjeru 5.9.

Primjer 5.9. Definiranje više konstruktora -- bolje

```
  1 class Pravokutnik {
  2
  3  private int x;
  4  private int y;
  5  private int sirina;
  6  private int visina;
  7  private String ime;
  8
  9  public Pravokutnik(int x, int y, int sirina, int visina,
 10                     String ime) {
 11   this.x = x;
 12   this.y = y;
 13   this.sirina = sirina;
 14   this.visina = visina;
 15   this.ime = ime;
 16  }
 17
 18  public Pravokutnik(int x, int y, String ime) {
 19   this(x, y, 1, 1, ime);
 20  }
```

```
21
22  public Pravokutnik(int x, int y) {
23   this(x, y, "anonimni");
24  }
25 }
26
```

U prikazanom rješenju, najopćenitiji konstruktor ostao je nepromijenjen. Najspecifičniji konstruktor iz retka 22 zadaću inicijalizacije objekta delegira konstruktoru iz retka 18, za što se koristi poziv metode specifičnog imena `this`; poziva se konstruktor koji prima dva cijela broja i jedan string pri čemu pozivatelj kao vrijednost imena predaje fiksnu vrijednost `"anonimni"`. Konstruktor iz retka 18 zadaću inicijalizacije pak delegira najopćenitijem konstruktoru: kao koordinate gornjeg lijevog ugla i kao ime predaje vrijednosti koje je dobio preko argumenata a kao širinu i visinu predaje fiksne vrijednosti 1.

Uporabom delegiranja moguće je pisati kod koji izbjegava redundantno definiranje implementacija više konstruktora. Postoji međutim jedno ograničenje: da bi se ovaj mehanizam koristio, delegirani poziv *mora biti prva naredba* u konstruktoru. Nakon te naredbe, tijelo konstruktora može sadržavati i dodatni posao koji treba obaviti.

Do sada smo razmotrili slučaj u kojem je na razini razreda definiran jedan ili čak i više alternativnih konstruktora. No je li uopće nužno da razred definira barem jedan konstruktor? Odgovor je: nije. Ako programer za razred ne definira niti jedan konstruktor, prevodilac će za takav razred prilikom generiranja *byte-koda* nadodati jedan javni konstruktor koji neće primati nikakve argumente i neće sadržavati naredbe. Ovaj konstruktor naziva se *pretpostavljeni konstruktor* (engleski termin je *default constructor*). Posljedica toga je da će svaki razred uvijek imati definiran barem jedan konstruktor. Ako programer razred opremi barem jednim konstruktorom, prevodilac neće dodavati još i pretpostavljeni konstruktor.

Prvi razredi

S objektnom paradigmom upoznavat ćemo se polagano kroz različite primjere. Krenimo s modeliranjem razreda `GeometrijskiLik`. Primjer ćemo potom razraditi tako da modeliramo različite vrste geometrijskih likova. Definirajmo geometrijski lik kao objekt koji ima jednu člansku varijablu:`ime` te dvije metode: jednu za dohvat površine lika a drugu za dohvat opsega lika. Implementacija takvog razreda prikazana je u primjeru 5.10.

Primjer 5.10. Razred `GeometrijskiLik`

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public abstract class GeometrijskiLik {
 4
 5  private String ime;
 6
 7  public GeometrijskiLik(String ime) {
 8   this.ime = ime;
 9  }
10
11  public String getIme() {
12   return ime;
13  }
14
15  public double getOpseg() {
16   return 0;
17  }
18
19  public double getPovrsina() {
```

```
20    return 0;
21   }
22 }
23
```

Razred definira člansku varijablu `ime` koja je tipa `String`, definira konstruktor koji kao argument prima ime te definira metode za dohvat površine i opsega koje obje vraćaju vrijednost 0, s obzirom da ne znamo o kojem se točno liku radi. Članska varijabla `ime` definirana je kao privatna -- to znači da joj nitko ne može pristupiti osim metoda koje su definirane direktno u razredu `GeometrijskiLik`. Metode za dohvat površine i opsega kao i konstruktor proglašene su javnima --one se mogu pozivati iz bilo kojeg dijela koda. Odluka da se članska varijabla `ime` proglasi privatnom omogućava nam zaštitu konzistentnosti samog objekta: nitko izvan ovog razreda ne može pristupiti toj varijabli pa je ne može niti promijeniti a time ne može niti narušiti konzistentnost objekta. Međutim, htjeli bismo da klijenti objekta mogu doznati za njegovo ime. Stoga smo dodali još jednu javnu metodu naziva `getIme` koja formalno ne prima nikakve argumente i koja vraća referencu na postavljeno ime objekta. Kako su stringovi u Javi nepromjenjivi, klijentima slobodno možemo vratiti referencu na string -- izvana ga nitko ne može promijeniti pa smo time sigurni. Metode čija je zadaća dohvaćanje vrijednosti neke varijable nazivaju se *getteri*. Način imenovanja ovih metoda propisan je normom *Java Beans Specification* koja nalaže da se za svako svojstvo (varijablu) koje se želi ponuditi klijentima na korištenje definira *getter* ako je svojstvo klijentima čitljivo odnosno *setter* ako klijenti smiju ažurirati svojstvo. Ako je naziv svojstva *XXX*, specifikacija definira da se je *getter* metoda čiji je naziv oblika `get`*Xxx* a *setter* metoda čiji je naziv `set`*Xxx*; prvo slovo naziva svojstva piše se veliko. Evo primjera razreda koji definira tri svojstva: *oznaku*, *brojac* i *plaću*.

```
class Primjer {
 private String oznaka;
 private int brojac;
 private double placa;

 public String getOznaka() {
   return oznaka;
 }
 public void setOznaka(String oznaka) {
   this.oznaka = oznaka;
 }

 public void setBrojac(int brojac) {
   this.brojac = brojac;
 }

 public double getPlaca() {
   return placa;
 }
}
```

Prikazani primjer definira svojstvo *oznaka* kao svojstvo koje klijent može i čitati i pisati (pa nudi *getter* i *setter*), svojstvo *brojač* koje klijent može samo mijenjati (pa nudi samo *setter*) te svojstvo *plaća* koje klijenti smiju samo čitati (pa nudi samo *getter*).

U primjeru razreda `GeometrijskiLik` odlučili smo da članska varijabla `ime` klijentima dostupna samo za čitanje pa smo razred opremili samo javnim *getterom*. Pogledajmo sada primjer uporabe ovog razreda.

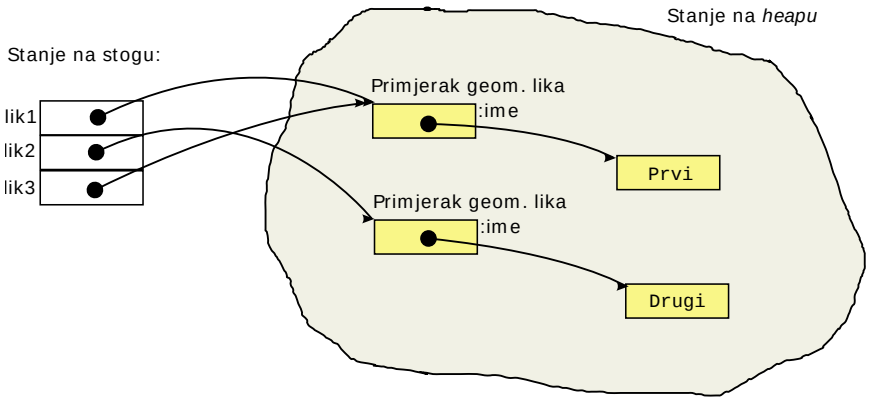
Primjer 5.11. Primjer uporabe razreda `GeometrijskiLik`

```
1 package hr.fer.zemris.java.tecaj_1.primjeri;
2
3 import hr.fer.zemris.java.tecaj_1.GeometrijskiLik;
```

```
 4
 5 public class Program1 {
 6
 7  public static void main(String[] args) {
 8   GeometrijskiLik lik1 = new GeometrijskiLik("Prvi");
 9   GeometrijskiLik lik2 = new GeometrijskiLik("Drugi");
10   GeometrijskiLik lik3 = lik1;
11
12   System.out.println("1: " + lik1.getIme());
13   System.out.println("2: " + lik2.getIme());
14   System.out.println("3: " + lik3.getIme());
15  }
16
17 }
18
```

U metodi `main` definirane su tri lokalne varijable: sve po tipu reference na objekte koji su primjerci razreda `GeometrijskiLik`. Program tijekom izvođenja stvara dva primjerka razreda pravokutnik. Reference `lik1` i `lik3` pokazuju na prvostvoreni primjerak a referenca `lik2` na drugostvoreni primjerak. Zamrznemo li program u trenutku nakon što je izveden redak 10, stanje dijela memorije programa koje nam je od interesa prikazano je na slici 5.1.

Slika 5.1. Stanje u memoriji



Lijevi dio slike prikazuje stanje na stogu. Na stogu se stvaraju lokalne varijable. Metoda `main` definirala je tri lokalne varijable: `lik1`, `lik2` i `lik3` koje su po tipu reference; stoga je na stogu zauzeto mjesto za tri reference. Primjenom operatora `new` stvorili smo dva primjerka razreda `GeometrijskiLik` koja su prikazana na *heapu*. Razred `GeometrijskiLik` deklarira jednu člansku varijablu: `ime` koja je po tipu `String`. Posljedica je da svaki primjerak razreda `GeometrijskiLik` ima svoju privatnu kopiju te varijable. Međutim, kako su u Javi Stringovi također objekti, varijabla tipa `String` zapravo je samo referenca. Stoga u memoriji kod svakog objekta imamo potrošnju memorije koliko je potrebno zauzeti za jednu referencu a ne cjelokupni znakovni niz. Sam stringa prikazan je kao novi objekt koji je također na *heapu*; objekt je prikazan kao jedan jedinstven jer nas u ovom trenutku ne zanima interni način na koji primjerci razreda `String` pamte podatke.



Nestatičke članske varijable

Članske varijable koje su deklarirane na razini razreda bez uporabe ključne riječi `static` nazivamo *nestatičkim članskim varijablama*. Nestatičke članske varijable pripadaju primjercima razreda odnosno objektima. Svaki objekt imat će svoju vlastitu kopiju takvih varijabli i promjena vrijednost takvih varijabli u jednom objektu neće imati nikakvog utjecaja na vrijednosti istoimenih varijabli u drugim objektima. Memorijsko zauzeće svakog od objekata reflektira količinu i vrstu nestatičkih varijabli koje su deklarirane u njegovom razredu.

Napravimo sada razred koji će predstavljati jedan konkretan geometrijski lik: napišimo razred čiji će primjerci predstavljati pravokutnike. Pri tome želimo ostvariti takav model koji će nam omogućiti da kažemo da je primjerak pravokutnika ujedno i primjerak geometrijskog lika (drugim riječima, ima sve što ima i geometrijski lik), koji ima još neke dodatne članske varijable. Razred `Pravokutnik` od razreda `GeometrijskiLik` trebao bi preuzeti sve što ima geometrijski lik (konkretno: člansku varijablu `ime` te metode koje su definirane u tom razredu) i trebao bi dodati članske varijable `x`, `y`, `sirina` i `visina`. Veza koju želimo uspostaviti naziva se *nasljeđivanje*. Reći ćemo da razred `Pravokutnik` nasljeđuje razred `GeometrijskiLik`. U Javi, činjenica da jedan razred nasljeđuje neki drugi razred navodi se prilikom definiranja tog novog razreda uporabom ključne riječi `extends`. Pogledajmo rješenje koje je prikazano u primjeru 5.12.

Primjer 5.12. Definiranje razreda `Pravokutnik`

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Pravokutnik extends GeometrijskiLik {
 4
 5  private int x;
 6  private int y;
 7  private int sirina;
 8  private int visina;
 9
10  public Pravokutnik(String ime, int x, int y,
11                     int sirina, int visina) {
12   super(ime);
13   this.x = x;
14   this.y = y;
15   this.sirina = sirina;
16   this.visina = visina;
17  }
18
19  public int getX() {
20   return x;
21  }
22  public int getY() {
23   return y;
24  }
25  public int getSirina() {
26   return sirina;
27  }
28  public int getVisina() {
29   return visina;
30  }
31
32 }
33
```

Redak 3 prethodnog ispisa sadrži deklaraciju koja uspostavlja vezu između razreda `Pravokutnik` i razreda `GeometrijskiLik`. Retci 5 do 8 sadrže deklaracije četiri nove članske varijable koje će dodatno imati svaki primjerak razreda `Pravokutnik`. U retku 10 prikazan je konstruktor razreda `Pravokutnik`. S obzirom da razred `Pravokutnik` nasljeđuje razred `GeometrijskiLik`, prilikom stvaranja primjerka razreda `Pravokutnik` u memoriji će se odjednom zauzeti dovoljno mjesta za sve članske varijable koje su definirane u oba razreda. Nakon toga, pozvat će se konstruktor razreda `Pravokutnik`. Prvi korak u radu tog konstruktora mora biti poziv konstruktora nadređenog razreda koji će najprije obaviti inicijalizaciju svojih članskih varijabli. Ovaj poziv obavlja se pozivom metode čije je ime `super`. Metodi je potrebno predati sve argumente koji su potrebni kako bi se mogao pozvati neki od konstruktora koji

su definirani u nadređenom razredu. S obzirom da u nadređenom razredu imamo samo jedan konstruktor koji kao argument očekuje ime, pozivamo taj konstruktor i predajemo mu ime koje smo dobili kao argument. Nakon što je taj dio obavljen, konstruktor obavlja inicijalizaciju svojih članskih varijabli.

Je li nužno da prvi redak konstruktora bude poziv konstruktora nadređenog razreda? Odgovor je: nije. Pravila su pri tome dva.

1. Ako konstruktor želi pozvati konstruktor nadređenog razreda, to mora obaviti kao prvu naredbu tijela konstruktora. Konstruktor nadređenog razreda nije moguće pozivati ako je izvedena bilo koja druga naredba.

2. Nije nužno da se u konstruktoru eksplicitno definira poziv konstruktora nadređenog razreda. U slučaju da se to ne napravi, napisani kod ekvivalentan je konstruktoru koji kao prvu naredbu ima umetnut poziv

   ```
   super();
   ```

   . Uočite da se time očekuje da nadređeni razred ima definiran konstruktor koji ne prima dodatne argumente i koji nije proglašen privatnim.

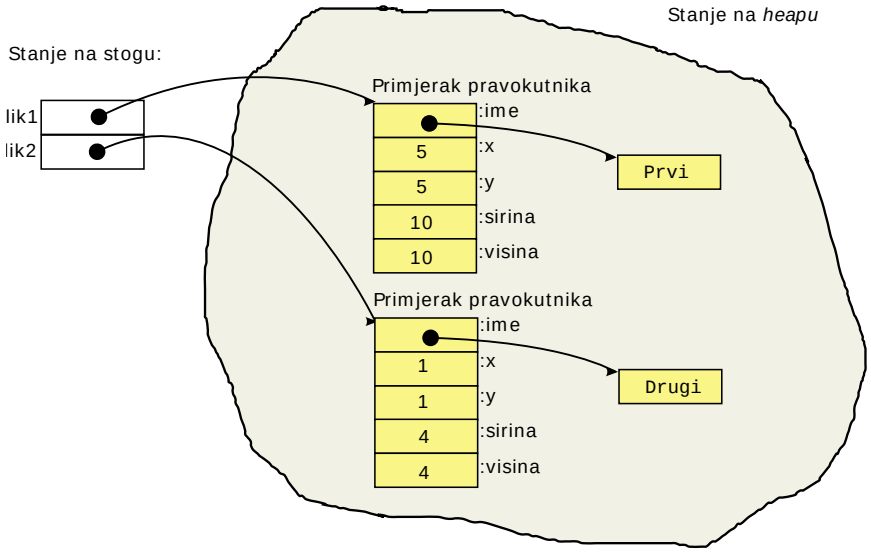
Pogledajmo sada jedan primjer programa koji koristi definirani razred `Pravokutnik`.

Primjer 5.13. Primjer uporabe razreda `Pravokutnik`

```
 1 package hr.fer.zemris.java.tecaj_1.primjeri;
 2
 3 import hr.fer.zemris.java.tecaj_1.GeometrijskiLik;
 4 import hr.fer.zemris.java.tecaj_1.Pravokutnik;
 5
 6 public class Program2 {
 7
 8  public static void main(String[] args) {
 9   GeometrijskiLik lik1 = new Pravokutnik("Prvi", 5, 5, 10, 10);
10   GeometrijskiLik lik2 = new Pravokutnik("Drugi", 1, 1, 4, 4);
11
12   System.out.format(
13    "Površina pravokutnika %s je %.1f.\n",
14    lik1.getIme(), lik1.getPovrsina());
15   System.out.format(
16    "Opseg pravokutnika %s je %.1f.\n",
17    lik2.getIme(), lik2.getOpseg());
18  }
19
20 }
21
```

Program se sastoji od metode `main` u kojoj se stvaraju dva primjerka razreda `Pravokutnik`. Zaustavimo li ovaj program prilikom izvođenja prije no što krene izvođenje retka 12, situacija koju ćemo imati u memoriji prikazana je na slici u nastavku.

Slika 5.2. Stanje u memoriji



Pokretanjem ovog programa dogodit će se, međutim, nešto neočekivano. Ispis je prikazan u nastavku.

```
Površina pravokutnika Prvi je 0.0.
Opseg pravokutnika Drugi je 0.0.
```

Naš program trenutno tvrdi da pravokutnik širine i visine 10 ima površinu 0 a ne 100 te da pravokutnik širine i visine 4 ima opseg koji je također jednak 0. Uočeni problem rezultat je činjenice da je razred `Pravokutnik` naslijedio sve metode razreda `GeometrijskiLik` a one pak vraćaju upravo dobivene rezultate.

S obzirom da nam te implementacije u razredu `Pravokutnik` ne odgovaraju, programski jezik Java omogućava nam da u razredu `Pravokutnik` ponudimo nove implementacije tih metoda koje će biti korištene od svih primjeraka ovog razreda. Jednom kada ponudimo novu implementaciju, ona će za sve objekte razreda `Pravokutnik` sakriti naslijeđenu implementaciju. Popravljena implementacija prikazana je u nastavku.

Primjer 5.14. Nadjačavanje metoda u razredu `Pravokutnik`

```
 1 public class Pravokutnik extends GeometrijskiLik {
 2
 3  // ostatak isto kao i prije ...
 4
 5  @Override
 6  public double getPovrsina() {
 7   return sirina * visina;
 8  }
 9
10  @Override
11  public double getOpseg() {
12   return 2*(sirina + visina);
13  }
14 }
15
```

Zbog uštede prostora, definicije članskih varijabli, konstruktor i prethodno definirane metode nisu prikazane već je prikazan samo novi dio koda. Počev od retka 6 dana je nova implementacija metode `getPovrsina()` koja za pravokutnik korektno računa njegovu površinu kao umnožak širine i visine pravokutnika. U retku 5 navedena je anotacija `@Override`. Uporaba ove anotacije nije razlog zbog kojeg mehanizam nadjačavanja radi -- kod bi bio jednako korektan i

bez te anotacije i radio bi jednako. Uporabom ove anotacije programer izriče jezičnom prevoditelju "ja mislim da ovime nadjačavam istoimenu metodu". Jezični prevoditelj će prilikom prevođenja provjeriti je li to istina i ako nije, prijavit će pogrešku. Ovo je zgodan kontrolni mehanizam koji će se pobrinuti da se odmah otkriju pogreške poput definiranja imena metode `getPorvsina` i sličnih koje je teško otkriti. Uočimo da definiranje tako nazvane metode nije pogrešno -- to je legalno ime i takva metoda može postojati. Problem je samo što ta metoda neće nadjačati metodu koju smo željeli i naš će kod i dalje pogrešno računati i površinu. Bez uporabe anotacije `Override` jezični prevoditelj ne bi imao nikakvog razloga na tom mjestu prijaviti pogrešku što bi potencijalno moglo dovesti do pogrešaka koje je teško pronaći u kodu i ispraviti.

Ako uz ovako modificirani razred `Pravokutnik` pokrenemo prethodni primjer, rezultat će biti korektan.

```
Površina pravokutnika Prvi je 100.0.
Opseg pravokutnika Drugi je 16.0.
```



Nadjačavanje

Mogućnost da se u prilikom nasljeđivanje u izvedenom razredu ponudi nova implementacija metode koja bi inače bila nasljeđena iz originalnog razreda predstavlja jedan od oblika *polimorfizma* i naziva se *nadjačavanje metoda*. Da bi nadjačavanje radilo, ne treba poduzimati nikakve dodatne korake. Dovoljno je da se u izvedenom razredu napiše nova implementacija i ona će automatski biti korištena za sve primjerke tog razreda ili drugih razreda koji njega nasljeđuju a sami ne nude još noviju implementaciju iste metode. Kada koristimo nadjačavanje metoda pri nasljeđivanju, kažemo da koristimo *dinamički polimorfizam*.

Prokomentirajmo sada još način na koji smo u metodi `main` pamtili reference na stvorene objekte. Pisali smo sljedeće.

```
GeometrijskiLik lik1 = new Pravokutnik("Prvi", 5, 5, 10, 10);
```

Ovaj primjer pokazuje da primjerak razreda `Pravokutnik` Java doista tretira i kao primjerak razreda `GeometrijskiLik`: s obzirom da razred `Pravokutnik` nasljeđuje razred `GeometrijskiLik`, svaki pravokutnik *je* geometrijski lik pa je takvo dodjeljivanje legalno. Mogli smo pisati i sljedeće.

```
Pravokutnik lik1 = new Pravokutnik("Prvi", 5, 5, 10, 10);
```

Program bi se ponašao na jednak način.



Kako radi dinamički polimorfizam?

Pretpostavimo da smo napisali sljedeći isječak koda.

```
public static void main(String[] args) {
  GeometrijskiLik lik1 = new GeometrijskiLik("Prvi");
  GeometrijskiLik lik2 = new Pravokutnik(
                              "Drugi", 1, 1, 4, 4);
  ispisi(lik1);
  ispisi(lik2);
}

private static void ispisi(GeometrijskiLik lik) {
  System.out.println(
    "Površina lika " + lik.getIme() +
    " je " + lik.getPovrsina()
  );
}
```

Pokretanjem ovog programa dobit ćemo sljedeći ispis.

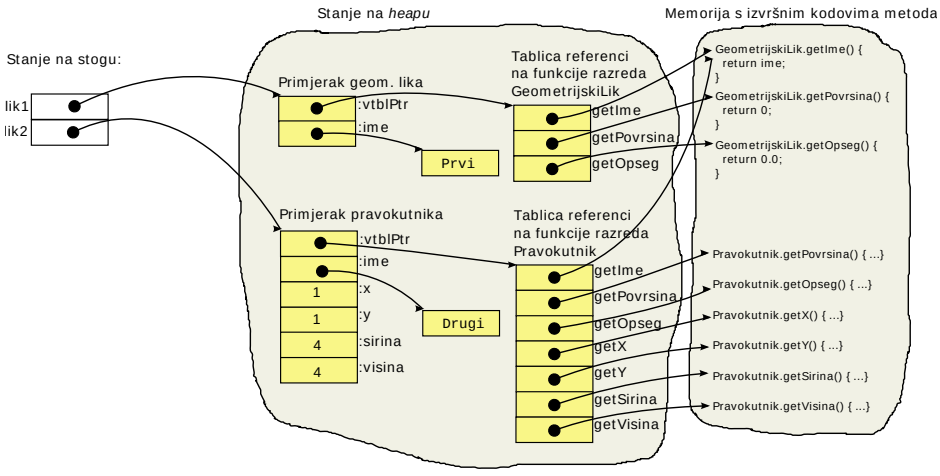
```
Površina lika Prvi je 0.0
Površina lika Drugi je 16.0
```

Zanimljivo pitanje je: kako je metoda `ispisi` znala koju implementacije metode `getPovrsina` treba pozvati -- da li onu iz razreda `GeometrijskiLik` ili onu iz razreda `Pravokutnik`? Odgovor na to pitanje je istovremeno i jednostavan (kad ga se shvati) i složen (dok ga se ne shvati). Do sada smo o konstruktorima ispričali gotovo sve što je za njih bilo važno osim jednog detalja. U Javi, jezični prevoditelj za svaki razred stvara po jednu tablicu u kojoj se nalaze reference na implementacije svih metoda koje tom razredu pripadaju direktno ili koje su nasljeđene. Tako primjerice razred `GeometrijskiLik` kako smo ga trenutno definirali ima pridruženu tablicu u kojoj se nalaze reference na implementacije metoda `getIme`, `getOpseg` te `getPovrsina`. Ta se tablica popunjava prilikom prevođenja izvornog programa. Razred `pravokutnik` ima tablicu s nešto više zapisa: osim referenci na implementacije metoda `getIme`, `getOpseg` i `getPovrsina`, tablica sadrži reference i na metode koje su po prvi puta definirane u razredu `Pravokutnik`; to su `getX`, `getY`, `getSirina` i `getVisina`. Prilikom prevođenja izvornog koda svakog od razreda moguće je utvrditi da li tajrazred nadjačava neku od metoda koje su prethodno definirane u izvornom razredu ili ne. Temeljem te informacije već je tijekom postupka prevođenja moguće pripremiti ovakve tablice za svaki definirani razred.

Jednom kada su ove tablice spremne, na scenu stupa postupak stvaranja novog objekta. Operator `new` u memoriji neće zauzeti samo onoliko mjesta koliko je potrebno za sve članske varijable -- količina zauzetog mjesta bit će nešto veća. Svaki će primjerak razreda (neovisno o konkretnom razredu) automatski dobiti još jednu skrivenu člansku varijablu koju ćemo u nastavku teksta zvati `vtblPtr`. Ta će skrivena varijabla tipično biti prva varijabla u memoriji koja je zauzeta za novi objekt. Nakon što se memorija zauzme, ažurira se vrijednost te reference tako da pokazuje na tablicu funkcija koja je pripremljena za razred čiji se primjerak stvara. Potom se poziva konstruktor razreda i kreće uobičajen proces pozivanja konstruktora nadređenog razreda iz izvedenih razreda.

U trenutku kada je potrebno obaviti poziv funkcije nad nekim objektom, dohvaća se njegova varijabla `vtblPtr` koja pokazuje na tablicu funkcija koju treba koristiti i u toj se tablici dohvaća adresa funkcije koju treba pozvati; nakon što je adresa poznata, poziva se ta funkcija. Grafički prikaz memorije za opisani slučaj prikazan je u nastavku.

Slika 5.3. Detaljno stanje u memoriji



Pogledajmo sada prethodni primjer. Pri prvom pozivu metode `ispisi` argument `lik` bit će referenca koja pokazuje na primjerak geometrijskog lika (na isti objekt na koji pokazuje i referenca `lik1`). Prilikom poziva `lik.getPovrsina()`, pogledat će se na koju tablicu funkcija pokazuje članska varijabla `vtblPtr` tog objekta što će u ovom slučaju biti tablica pripremljena za razred `GeometrijskiLik`. Potom će se pogledati na koju funkciju u toj tablici pokazuje referenca na poziciji 1 (u prikazanoj tablici, na poziciji 0 je referenca koja pokazuje na implementaciju metode `getIme`, na poziciji 1 je referenca koja pokazuje na implementaciju metode `getPovrsina` a na poziciji 2 je referenca koja pokazuje na implementaciju metode `getOpseg`) i ta će se funkcija pozvati. U ovom slučaju, to će biti implementacija koja je ponuđena u razredu `GeometrijskiLik`.

Pri sljedećem pozivu metode `ispisi` metodi se predaje referenca na objekt koji je primjerak razreda `Pravokutnik`. Metoda to, međutim, ne zna i zapravo je ne zanima. Za izvođenje poziva `lik.getPovrsina()` pogledat će se na koju tablicu funkcija pokazuje članska varijabla `vtblPtr` primljenog objekta, što će u ovom slučaju biti tablica pripremljena za razred `Pravokutnik` (provjerite to na prethodnoj slici). Potom će se pogledati na koju funkciju u toj tablici pokazuje referenca na poziciji 1 i ta će se funkcija pozvati. U ovom slučaju, to će biti implementacija koja je ponuđena u razredu `Pravokutnik`. Poziv `lik.getIme()` razriješit će se istim mehanizmom oba puta u implementaciju ponuđenu u okviru razreda `GeometrijskiLik`.

Spomenimo još jednom i cijenu koju objektno-orijentirani programski jezici plaćaju prilikom implementacije dinamičkog polimorfizma na opisani način.

- Za svaki razred koji treba podržati dinamički polimorfizam (u Javi to su svi) potrebno je u memoriji čuvati tablicu referenci na implementacije metoda.

- Za svaki objekt koji je primjerak takvog razreda potrebno je dodatno rezervirati još malo memorije kako bi se u objekt mogla pohraniti i referenca na tablicu koju treba koristiti. Za razliku od tablice koja može biti velika, referenca koja se dodaje objektima ne čini se kao veliki utrošak memorije, no treba biti svjestan da program može stvarati na tisuće ili čak i milione objekata pa se ukupni trošak brzo akumulira.

- Pozivi metoda koji se razrješavaju opisanim postupkom sporiji su od poziva metoda nad kojima dinamički polimorfizam ne djeluje. U slučaju takvih metoda nije potrebno adresu funkcije dohvaćati iz tablice već prevodilac može u trenutku prevođenja utvrditi ispravnu adresu. U jeziku poput C++-a ovo je razlika između virtualnih (tj. funkcija koje se polimorfno razrješavaju) i nevirtualnih funkcija. U programskom jeziku Java sve su nestatičke metode po definiciji virtualne (statičke metode obradit ćemo malo kasnije).

Kako ih ne bismo nepotrebno opterećivali, u nastavku ove knjige na grafičkim prikazima više nećemo prikazivati niti varijable `vtblPtr` niti tablice funkcija koje su pripremljene za svaki od razreda. Očekuje se međutim da je čitatelj svjestan ovih detalja.

Napišimo sadaza vježbu još jedan razred koji će predstavljati novu vrstu geometrijskih likova: krugove. Razred ćemo nazvati `Krug`. Definicija razreda prikazana je u primjeru 5.15.

Primjer 5.15. Definiranje razreda `Krug`

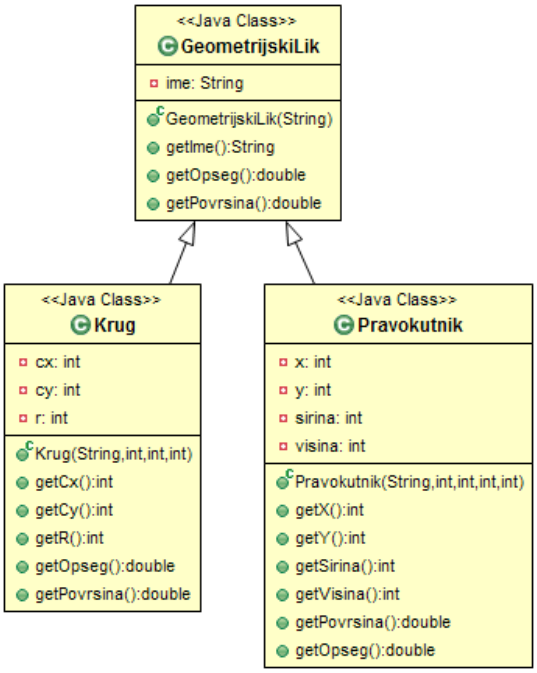
```
1 package hr.fer.zemris.java.tecaj_1;
2
3 public class Krug extends GeometrijskiLik {
4
```

```
 5  private int cx;
 6  private int cy;
 7  private int r;
 8
 9  public Krug(String ime, int cx, int cy, int r) {
10   super(ime);
11   this.cx = cx;
12   this.cy = cy;
13   this.r = r;
14  }
15
16  public int getCx() {
17   return cx;
18  }
19
20  public int getCy() {
21   return cy;
22  }
23
24  public int getR() {
25   return r;
26  }
27
28  @Override
29  public double getOpseg() {
30   return 2 * r * Math.PI;
31  }
32
33  @Override
34  public double getPovrsina() {
35   return r * r * Math.PI;
36  }
37
38 }
39
```

Razred `Krug` definira tri članske varijable: *x* i *y* koordinatu centra (članske varijable `cx` i `cy`) te radijus kruga (članska varijabla `r`). Potom su definirani javni *getteri* te nadjačanja metoda `getPovrsina` i `getOpseg` kako bi vraćale ispravne rezultate za ovu vrstu geometrijskih likova.

Dijagram razreda na kojem su prikazani osnovni razred `GeometrijskiLik` te izvedeni razredi `Pravokutnik` i `Krug` prikazan je na slici u nastavku.

Slika 5.4. Dijagram razreda



Relacija: *razred X nasljeđuje razred Y* naznačuje se strelicom s vrhom u obliku praznog trokuta, koja je usmjerena od razreda X prema razredu Y.

Razradimo sada opisani primjer. Pretpostavimo da je potrebno osigurati da se svaki od geometrijskih likova znade nacrtati na nekoj rasterskoj prikaznoj jedinici. Rasterske prikazne jedinice su jedinice koje prikazuju rasterske slike: slike sastavljene od konačnog broja slikovnih elemenata. Zaslon CRT monitora ili LCD monitora je upravo takva prikazna jedinica. Tipične su rezolucije takvih jedinica danas u rasponu od tisuću do dvije tisuće slikovnih elemenata po jednom retku te od par stotina do poprilici tisuću redaka (primjerice: 1600 puta 900). koordinatni sustav kod takve prikazne jedinice uobičajeno je smješten u gornji lijevi ugao te ima koordinate (0,0). Pozitivna *x*-os gleda u desno a pozitivna *y*-os prema dolje.

Napravimo najprije razred koji modelira takvu rastersku crno-bijelu sliku. Kod crno-bijelog prikaza, svaki slikovni element može biti upaljen (osvijetljen) ili ugašen. Stoga ćemo pamćenje slikovnih elemenata pamtiti kao dvodimenzijsko polje `boolean`-zastavica. Konačno, dodat ćemo metodu koja omogućava paljenje i gašenje slikovnog elementa te metodu koja će sliku pretvoriti u `String` koji je potom lagano ispisati za zaslon. Rješenje je prikazano u nastavku.

Primjer 5.16. Definiranje razreda `Slika` i primjer uporabe

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Slika {
 4
 5  private int sirina;
 6  private int visina;
 7  private boolean[][] elementi;
 8
 9  public Slika(int sirina, int visina) {
10   this.sirina = sirina;
11   this.visina = visina;
12   elementi = new boolean[visina][sirina];
13  }
14
```

```
15  public int getSirina() {
16   return sirina;
17  }
18
19  public int getVisina() {
20   return visina;
21  }
22
23  public void upaliElement(int x, int y) {
24   elementi[y][x] = true;
25  }
26
27  public void ugasiElement(int x, int y) {
28   elementi[y][x] = false;
29  }
30
31  public String toString() {
32   StringBuilder sb = new StringBuilder((sirina+1)*visina);
33   for(int y = 0; y < visina; y++) {
34    for(int x = 0; x < sirina; x++) {
35     sb.append(elementi[y][x] ? '*' : '.' );
36    }
37    sb.append('\n');
38   }
39   return sb.toString();
40  }
41 }
42
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class PrimjerSlike {
 4
 5  public static void main(String[] args) {
 6   Slika slika = new Slika(5, 3);
 7   slika.upaliElement(0, 0);
 8   slika.upaliElement(0, 1);
 9   slika.upaliElement(0, 2);
10   slika.upaliElement(1, 1);
11   slika.upaliElement(2, 2);
12   slika.upaliElement(3, 2);
13   slika.upaliElement(4, 2);
14   System.out.print(slika.toString());
15  }
16
17 }
18
```

Razred `Slika` u konstruktoru prima informacije o širini i visini slike odnosno o dimenzijama rastera. Podatci o svakom slikovnom elementu pamte se u dvodimenzijskom polju `elementi`. Razred `PrimjerSlike` ilustrira uporabu razreda `Slika`: stava se slika širine 5 slikovnih elemenata i visine 3 slikovna elementa. Potom se pali nekoliko slikovnih elemenata i rezultat se ispisuje na ekran. Svaki upaljeni slikovni element bit će prikazan znakom `*` dok će ugašeni slikovni elementi biti prikazani znakom `.`. Pokretanjem prethodnog programa dobit ćemo sljedeći rezultat.

```
*....
**...
*.***
```

Imajući u vidu ovakvu implementaciju slike, pogledajmo sada kako bismo mogli osigurati da se svaki od geometrijskih likova može nacrtati na slici. Evo ideje: zamislimo da svaki geometrijski lik možemo pitati za svaku točku koja nas zanima, sadrži li on tu točku ili ne. Metoda bi trebala vratiti `true` ako geometrijski lik sadrži predanu točku a `false` inače. Prototip takve metode prikazan je u nastavku.

```
public boolean sadrziTocku(int x, int y);
```

Pretpostavimo li sada da će svaki geometrijski lik znati dati korektan odgovor na postavljeno pitanje, lako je osmisliti potpuno generičku metodu koja će moći nacrtati svaki lik: za predanu sliku, metoda lik treba pitati za svaki slikovni element koji slika sadrži -- pripada li on liku ili ne. Ako pripada, slikovni element treba osvijetliti.

Što dakle trebamo napraviti?

1. U razred `GeometrijskiLik` trebamo dodati implementaciju metode `sadrziTocku`. Kako ta metoda u geometrijskom liku nema dovoljno informacija da bi utvrdila pripada li slikovni element na predanim koordinatama liku ili ne, na ovom mjestu metoda može vratiti bilo što. Razmišljajući da metodu implementiramo tako da radi najmanju štetu, odlučit ćemo se da metoda uvijek vraća `false`.

2. U svim razredima koji nasljeđuju razred `GeometrijskiLik` potrebno je nadjačati prethodnu implementaciju metode `sadrziTocku` tako da vraća korektne informacije za primjerke tog razreda. Pitanje na koje za sada nemamo odgovor jest: *kako korisnika natjerati da to doista i napravi*? Ako na ovo zaboravimo u nekom razredu, crtanje primjeraka tog razreda neće raditi korektno. Na ovaj problem osvrnut ćemo se u jednom od sljedećih poglavlja.

3. U razredu `GeometrijskiLik` potrebno je implementirati metodu `crtaj` sa sljedećim prototipom:

   ```
   public void crtaj(Slika slika);
   ```

   . Metoda će crtanje obaviti prozivanjem lika za svaki slikovni element slike i paljenjem onih slikovnih elemenata za koje se lik izjasni da mu pripadaju.

Krenimo stoga u implementaciju opisanog rješenja. Dorađene implementacije razreda prikazane su u nastavku.

Primjer 5.17. Dodavanje mogućnosti crtanja likova

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class GeometrijskiLik {
 4
 5  private String ime;
 6
 7  public GeometrijskiLik(String ime) {
 8   this.ime = ime;
 9  }
10
11  public String getIme() {
12   return ime;
13  }
14
15  public double getOpseg() {
16   return 0;
17  }
18
19  public double getPovrsina() {
20   return 0;
```

```
21  }
22
23  public boolean sadrziTocku(int x, int y) {
24   return false;
25  }
26
27  public void crtaj(Slika slika) {
28   for(int y = 0, h = slika.getVisina(); y < h; y++) {
29    for(int x = 0, w = slika.getSirina(); x < w; x++) {
30     if(this.sadrziTocku(x, y)) {
31      slika.upaliElement(x, y);
32     }
33    }
34   }
35  }
36 }
37
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Pravokutnik extends GeometrijskiLik {
 4
 5  private int x;
 6  private int y;
 7  private int sirina;
 8  private int visina;
 9
10  public Pravokutnik(String ime, int x, int y,
11           int sirina, int visina) {
12   super(ime);
13   this.x = x;
14   this.y = y;
15   this.sirina = sirina;
16   this.visina = visina;
17  }
18
19  public int getX() {
20   return x;
21  }
22  public int getY() {
23   return y;
24  }
25  public int getSirina() {
26   return sirina;
27  }
28  public int getVisina() {
29   return visina;
30  }
31
32  @Override
33  public double getPovrsina() {
34   return sirina * visina;
35  }
36
37  @Override
38  public double getOpseg() {
39   return 2*(sirina + visina);
40  }
```

```
41
42  @Override
43  public boolean sadrziTocku(int x, int y) {
44   if(x < this.x || x >= this.x + sirina) return false;
45   if(y < this.y || y >= this.y + visina) return false;
46   return true;
47  }
48 }
49
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Krug extends GeometrijskiLik {
 4  private int cx;
 5  private int cy;
 6  private int r;
 7
 8  public Krug(String ime, int cx, int cy, int r) {
 9   super(ime);
10   this.cx = cx;
11   this.cy = cy;
12   this.r = r;
13  }
14
15  public int getCx() {
16   return cx;
17  }
18
19  public int getCy() {
20   return cy;
21  }
22
23  public int getR() {
24   return r;
25  }
26
27  @Override
28  public double getOpseg() {
29   return 2 * r * Math.PI;
30  }
31
32  @Override
33  public double getPovrsina() {
34   return r * r * Math.PI;
35  }
36
37  @Override
38  public boolean sadrziTocku(int x, int y) {
39   int dx = x-cx;
40   int dy = y-cy;
41   return dx*dx + dy*dy <= r*r;
42  }
43 }
44
```

Pogledajte način na koji su implementirane navedene metode u svakom od razreda. Uočimo također na koji smo način koristili `for`-petlje u metodi `crtaj` razreda `GeometrijskiLik`. Vanjsku petlju mogli smo napisati i ovako.

```
for(int y = 0; y < slika.getVisina(); y++) {
...
}
```

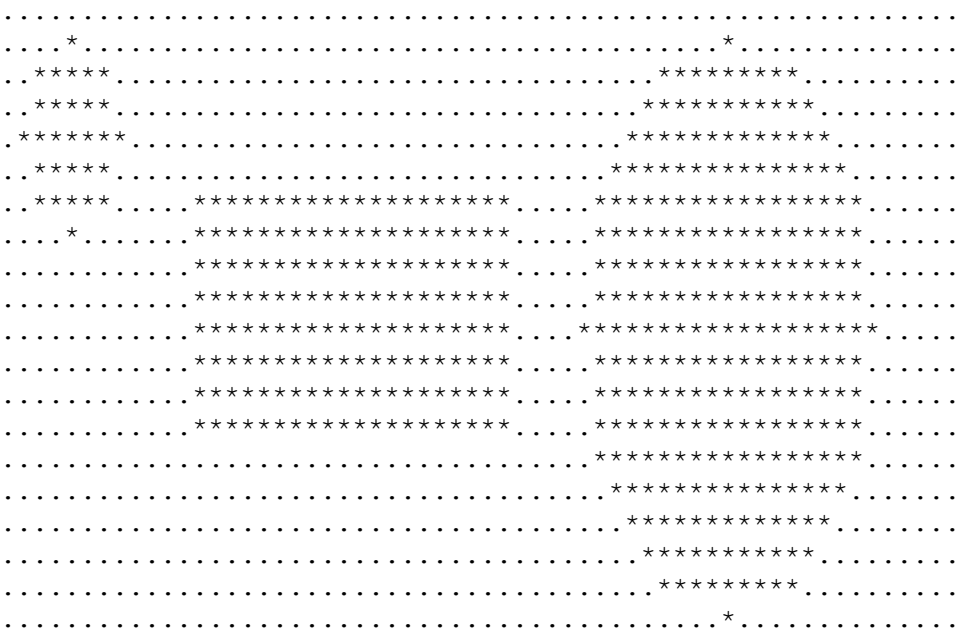
Takav način pisanja `for`-petlje treba izbjegavati: uvjet zaustavljanja petlje (središnji argument) provjeravat će se na kraju svake iteracije nanovo. To znači da će se na kraju svake iteracije nanovo pozvati metoda `getVisina` nad objektom `slika`. Ovisno o implementaciji te metode u tom objektu, ovaj poziv može biti skup i svakako ga ne želimo raditi toliko često -- posebice jer znamo da se visina slike neće promijeniti tijekom izvođenja petlje. Rješenje koje se preporuča jest podatak o visini slike dohvatiti na početku izvođenja petlje i pospremiti ga u pomoćnu varijablu s kojom se kasnije radi usporedba. To je upravo rješenje koje je napisano i u izvornom kodu a ovdje ga još jednom ponavljamo.

```
for(int y = 0, h = slika.getVisina(); y < h; y++) {
...
}
```

Pogledajmo sada kako se navedeni kod ponaša na konkretnom primjeru. Napisan je sljedeći glavni program koji stvara sliku dimenzija 60 puta 20, dva kruga i jedan pravokutnik, crta ih na slici i potom sliku prikazuje na zaslonu.

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class PrimjerSlike2 {
 4
 5  public static void main(String[] args) {
 6   Slika slika = new Slika(60, 20);
 7
 8   GeometrijskiLik[] likovi = new GeometrijskiLik[] {
 9    new Krug("k1", 4, 4, 3),
10    new Krug("k2", 45, 10, 9),
11    new Pravokutnik("p1", 12, 6, 20, 8)
12   };
13
14   for(GeometrijskiLik lik : likovi) {
15    lik.crtaj(slika);
16   }
17
18   System.out.print(slika.toString());
19  }
20
21 }
22
```

Uočite da je metodu `crtaj` sasvim legalno pozvati nad primjerkom razreda `Pravokutnik` ili `Krug` iako ona nije definirana niti u jednom od ta dva razreda. Ta je metoda definirana u razredu `GeometrijskiLik` a kako su oba razreda, `Pravokutnik` i `Krug` nasljedila razred `GeometrijskiLik`, automatski su nasljedili i tu metodu. Pokretanje programa rezultirat će sljedećim ispisom.



Preporučamo da ovaj primjer i isprobate. Napišite sve potrebne razrede i uvjerite se da prikazano rješenje radi.

Sada nam je još ostalo da se osvrnemo na performanse prikazanog rješenja. Metoda `crtaj` kako je definirana u razredu `GeometrijskiLik` ima dvije važne karakteristike.

1. Metoda je strahovito neefikasna. Ako se koristi za crtanje pravokutnika širine i visine 2 slikovna elementa na slici rezolucije 1000 puta 1000, metoda će obaviti 1 000 000 poziva metode `sadrziTocku` kako bi sve skupa na kraju upalila 4 slikovna elementa.

2. Metoda je potpuno generička i ispravno crta bilo koji geometrijski lik. Metoda od likova ne zahtjeva ništa više osim da implementiraju jednu metodu: `sadrziTocku`. Oslanjajući se na tu metodu, metoda `crtaj` primjenjiva je na crtanje svih geometrijskih likova.

Promotrimo li točke (1) i (2), uočit ćemo uobičajeni kompromis s kojim smo uvijek suočeni prilikom pisanja programa: ako imamo malo informacija o konkretnom problemu koji rješavamo, moći ćemo pisati samo generičke algoritme. Oni će biti primjenjivi na širok skup problema ali će biti neefikasni. Kada radimo u objektno-orijentiranom programskom jeziku, to ne mora biti kraj priče. Spore implementacije zahvaljujući dinamičkom polimorfizmu možemo nadjačati efikasnijim implementacijama -- u situacijama u kojima imamo dovoljno podataka da možemo napraviti efikasniju implementaciju.



Naputak za oblikovanje

Uvijek gdje je to prikladno ponudite generičke implementacije algoritama. One će biti neefikasne ali će raditi sa širokim skupom različitih objekata. Kada metode u nekom razredu možete implementirati efikasnije -- učinite to oslanjajući se na dinamički polimorfizam: neefikasnu metodu nadjačajte efikasnijom implementacijom.

Što bi to točno značilo za naš primjer geometrijskih likova? Proanalizirajmo crtanje pravokutnika. Možemo li u pravokutniku ponuditi efikasniju implementaciju crtanja od one koja je prisutna u razredu `GeometrijskiLik`? Odgovor je: svakako. Pa nema potrebe da prozivanje radimo za sve točke slike kada točno znamo koji slikovni elementi pripadaju pravokutniku. Stoga ćemo u razredu `pravokutnik` nadjačati metodu `crtaj` implementacijom koja je prikazana u nastavku.

```
1 public class Pravokutnik extends GeometrijskiLik {
2
3  private int x;
```

```
 4   private int y;
 5   private int sirina;
 6   private int visina;
 7
 8   // ...
 9   // ostatak koda nije prikazan zbog uštede u prostoru
10   // ...
11
12   @Override
13   public void crtaj(Slika slika) {
14    // Ako se niti jedan dio pravokutnika ne vidi:
15    if(x+sirina <= 0 || x >= slika.getSirina()) return;
16    if(y+visina <= 0 || y >= slika.getVisina()) return;
17
18    // Utvrdi dio pravokutnika koji se vidi na slici:
19    int odX = Math.max(0, x);
20    int doX = Math.min(x+sirina, slika.getSirina());
21    int odY = Math.max(0, y);
22    int doY = Math.min(y+visina, slika.getVisina());
23
24    // Osvijetli utvrđeno područje:
25    for(int y = odY; y < doY; y++) {
26     for(int x = odX; x < doX; x++) {
27      slika.upaliElement(x, y);
28     }
29    }
30   }
31  }
32
```

Opisana implementacija metode `crtaj` može biti efikasnija jer ima više informacija no što ih je imala istoimena metoda u razredu `GeometrijskiLik`. Na ovom mjestu točno znamo da se radi o pravokutniku, znamo gdje on započinje i znamo gdje završava. Temeljem tih informacija možemo krenuti u efikasno paljenje slikovnih elemenata koji pripadaju tom liku.

Prilikom implementacije, treba međutim biti oprezan. Sada smo suočeni s novim problemom koji prije nismo imali. Stara implementacija metode prozivala je lik za sve slikovne elemente slike. Nova implementacija želi sve slikovne elemente lika upaliti na slici -- što ako slika ne sadrži sve takve elemente? Primjerice, što ako imamo pravokutnik čiji je gornji lijevi ugao smješten u točku (-2,-2) i koji je širok 5 i visok 5 slikovnih elemenata? Slika nema slikovni element koji bi odgovarao točki (-2,-2) pa naše nadjačanje ne smije niti pozvati metodu `slika.upaliElement(-2,-2)`. Stoga se prilikom implementacije efikasnijeg popunjavanja treba povesti računa da se napravi korektno odsijecanje. Prikazana metoda najprije utvrđuje hoće li se, i koji, dio pravokutnika vidjeti na slici; ako je odgovor ne, izlazi se iz metode. Sljedeći korak je pronalazak onog dijela pravokutnika koji će se vidjeti i posljednji korak je njegovo iscrtavanje.

Zaštita privatnosti elemenata razreda

Članske varijable i metode, baš kao i sami razredi sa sobom uvijek imaju povezan *modifikator pristupa* (engl. *access modifier*) koji definira tko ima pravo pristupiti čemu. Java definira četiri modifikatora pristupa koji su navedeni u nastavku.

- *Javni pristup*: definira se uporabom ključne riječi `public`.
- *Zaštićeni pristup*: definira se uporabom ključne riječi `protected`.
- *Privatni pristup*: definira se uporabom ključne riječi `private`.

- *Privatni pristup za paket*: definira se nenavođenjem bilo koje od prethodne tri ključne riječi. Ako se dakle ne navede ništa, podrazumijeva se ovakav pristup.

Na vršnoj razini, elementi mogu kao modifikator pristupa imati definiran javan pristup ili privatni pristup za paket. primjerice, kada se u datoteci `X.java` definira razred `X`, ispred ključne riječi `class` ili može stajati `public` ili ne smije stajati ništa. Ako stoji ključna riječ `public`, taj je razred javan i može mu pristupiti bilo tko iz bilo kojeg paketa. Ako ne stoji ništa, razred je vidljiv samo drugim razredima koji su u istom paketu kao što je i promatrani razred.

Unutar razreda, na razini članskih varijabli, članskih metoda te ugniježđenih razreda moguće je koristiti bilo koji od četiri navedena modifikatora pristupa. Po zaštitnoj snazi, modifikatori su poredani ovako:

- *javni pristup* nudi najslabiju zaštitu; sve što je deklarirano javno vidljivo je svima;
- *zaštićeni pristup* nudi malo bolju zaštitu; sve što je deklarirano kao zaštićeno vidljivo je isključivo iz razreda koji se nalaze u istom paketu kao i promatrani razred ili koji se nalaze u nekom drugom paketu ali koji nasljeđuju promatrani razred; razredi koji su u drugim paketima i koji ne nasljeđuju promatrani razred ne vide njegove elemente koji su označeni kao zaštićeni;
- *privatni pristup za paket* nudi još jaču zaštitu; sve što je označeno ovim modifikatorom vidljivo je samo i isključivo iz razreda koji su smješteni u istom paketu kao i promatrani razred;
- konačno, *privatni pristup* nudi najjaču zaštitu; sve što je deklarirano privatno vidljivo je samo metodama i razredima definiranim unutar promatranog razreda; ovakve elemente ne vide drugi razredi iz istog paketa, drugi razredi iz drugih paketa ili pak razredi koji nasljeđuju promatrani razred.

Prilikom oblikovanja razreda uvijek je preporuka koristiti najjaču moguću zaštitu uz koju je moguće implementirati potrebnu funkcionalnost.

Opisano djelovanje modifikatora može se lijepo prikazati i tablicom što je načinjeno u nastavku. Pojam članski element odnosi se na bilo koji element koji može biti deklariran unutar razreda: varijabla, metoda, ugniježđeni razred i slično.

Tablica 5.1. Razine pristupa članskim elementima razreda

Modifikator	Iz istog razreda	Iz Istog paketa	Iz podrazreda	Od bilo kuda
`public`	DA	DA	DA	DA
`protected`	DA	DA	DA	NE
bez modifikatora	DA	DA	NE	NE
`private`	DA	NE	NE	NE

Na vrhu stabla: razred `Object`

Prilikom opisa pojma nasljeđivanja, rekli smo da Java nudi model jednostrukog nasljeđivanja. Nije moguće definirati razred koji nasljeđuje od više drugih razreda. A je li moguće definirati razred koji ne nasljeđuje od nikoga? Možda začuđujuće, odgovor je također ne! U programskom jeziku Java, svaki razred koji programer napiše uvijek nasljeđuje neki drugi razred. Tako primjerice, ako definiramo razred ovako:

```
class A extends B {
  ...
}
```

tada je jasno: razred `A` nasljeđuje razred `B`. No što ako napišemo sljedeće:

```
class C {
```

```
 ...
}
```

od koga sada razred `C` nasljeđuje? Odgovor će postati jasan ako kažemo da svaka deklaracija razreda u kojoj nije naveden dio `extends ...` automatski dobiva napisano sljedeće: `extends Object`. Prethodni je isječak koda stoga ekvivalentan kodu prikazanom u nastavku.

```
class C extends Object {
 ...
}
```

Razred `Object` definiran je u paketu `java.lang` i predstavlja vršni razred iz kojega direktno ili tranzitivno svi drugi razredi nasljeđuju niz korisnih metoda koje su u njemu definirane. Dapače, čak su i polja primjerci ovog razreda. Pa pogledajmo koje su to metode.

String toString();

Metoda `toString` služi za generiranje tekstovnog prikaza objekta. Tipično se koristi prilikom debugiranja za generiranje informacija o trenutnom objektu. Primjerice, tu bismo metodu u razredu `Krug` mogli nadjačati na sljedeći način.

```
@Override
public String toString() {
 return "Krug: centar=("+cx+","+cy+"), radijus="+r;
}
```

Uz tako definiranu metodu, mogli bismo napisati sljedeći primjer.

```
 1 public class Ispis1 {
 2
 3  public static void main(String[] args) {
 4   Krug k = new Krug("k1", 2, 2, 10);
 5
 6   System.out.println(k.toString());
 7   System.out.println(k);
 8
 9   String poruka = "Element je: " + k;
10   System.out.println(poruka);
11  }
12
13 }
```

Rezultat izvođenja prikazan je u nastavku.

```
Krug: centar=(2,2), radijus=10
Krug: centar=(2,2), radijus=10
Element je: Krug: centar=(2,2), radijus=10
```

Pogledajmo korak po korak kako smo došli do tog ispisa. U retku 6 zovemo metodu `println` i kao argument joj predajemo rezultat poziva upravo napisane metode što je po tipu `String`.

U retku 7, metodi `println` predajemo referencu na primjerak razreda `Krug` no rezultat je isti. Razlog tome je što postoji jedna od varijanti metode `println` koja prima referencu na bilo kakav objekt i nad tom referencom eksplicitno zove metodu `toString` i ispisuje ono što ta metoda vrati.

Konačno, u retku 9 pokušavamo zalijepiti `String` i primjerak razreda `Krug`; ono što se događa jest da se nad predanim primjerkom implicitno poziva metoda `toString` i lijepi se ono što ona vrati. Uz ovo pojašnjenje, jasno je zašto je rezultat takav.

int hasCode();

Metoda služi za generiranje *hash*-vrijednosti trenutnog objekta. Ovu vrijednost još nazivamo i *sažetak objekta*. Koristit ćemo je prilikom rada s kolekcijama i na tom mjestu ćemo dati detaljniji tretman. Ideja metode je da na temelju sadržaja objekta generira cijeli broj koji se kasnije može koristiti za određivanje pretinca u koji će se objekt pohraniti ako ga se sprema u tablicu raspršenog adresiranja. Funkcija bi trebala biti napisana na način da već i mala promjena sadržaja objekta generira veliku promjenu u generiranom broju čime se garantira dobro raspršenje adresa. Navedimo primjer kako bismo u razredu `Krug` mogli nadjačati ovu metodu: broj ćemo generirati temeljem koordinata centra i zadanog radijusa.

```
@Override
public int hashCode() {
 return (cx + 7*cy) << 5 + r;
}
```

Metoda `hashCode` u razredu `Object` implementirana je na način da vraća sažetak memorijske adrese na kojoj se nalazi objekt (odnosno, možemo se praviti da ima takvu semantiku).

boolean equals(Object obj);

Metoda služi usporedbu trenutnog objekta s objektom koji je predan kao argument. Zadaća metode jest utvrditi jesu li ta dva objekta jednaka i ako jesu, metoda treba vratiti vrijednost `true` a inače vrijednost `false`. Kada su dva objekta jednaka, stvar je definicije i ova metoda omogućava da se ta definicija implementira u kodu. Metoda se obilato koristi prilikom rada s kolekcijama. Navedimo samo primjer kako bismo u razredu `Krug` mogli nadjačati ovu metodu. Primjerice, reći ćemo da su dva kruga jednaka ako imaju isti centar i isti radijus.

```
@Override
public boolean equals(Object obj) {
 if(!(obj instanceof Krug)) return false;
 Krug drugi = (Krug)obj;
 return this.cx==drugi.cx && this.cy==drugi.cy && this.r==drugi.r;
}
```

Kako argument metode može biti bilo kakav objekt, u prvom retku metode provjeravamo može li se na predani objekt gledati kao na primjerak razreda `Krug`, što ispitujemo uoprabom operatora `instanceof`. Sintaksa kojom se koristi ovaj operator je *referenca* `instanceof` *nazivRazreda*. Operator će ispitati je li objekt na koji je predana referenca primjerak navedenog razreda ili razreda koji je izveden iz navedenog razreda (direktno ili tranzitivno). Ako je, operator će vratiti vrijednost `true` a inače vrijednost `false`. Ako utvrdimo da predani objekt nije primjerak kruga, metoda vraća `false` čime tvrdimo da trenutni objekt i predani objekt nisu semantički jednaki. Ako smo se pak uvjerili da se na predani objekt može gledati kao na krug, tada u sljedećem retku referencu na predani objekt ukalupljujemo u referencu na krug i potom ispitujemo imaju li trenutni objekt i predani krug jednak centar i radijus, pa na temelju toga donosimo odluku jesu li ta dva objekta semantički jednaka.

Metoda posebice je značajna ako se prisjetimo što radi operator `==` u Javi: on uspoređuje primitivne vrijednosti. Ako su mu argumenti reference, on će provjeriti da li te reference pokazuju na istu memorijsku lokaciju ili ne. Evo primjera.

```
String prvi = new String("Abba");
String drugi = new String("Abba");
System.out.println(prvi==drugi);
System.out.println(prvi.equals(drugi));
```

Program će na zaslon najprije ispisati `false` a potom `true`: reference `prvi` i `drugi` doista pokazuju na različite memorijske lokacije jer je svaka generirana novim pozivom operatora `new`. Međutim, mi bismo htjeli reći da su dva stringa jednaka ako su sadržajno jednaka. Razred `String` nadjačava metodu `equals` kako bi nam ponudio upravo takvu semantiku.

Uočimo da prethodno definirani način nije jedini način na koji smo mogli definirati jednakost krugova. Druga jednakovrijedna mogućnost bila bi reći da su dva kruga jednaka ako imaju definirano jednako ime. U tom slučaju bismo umjesto prethodnog nadjačanja koristili sljedeće.

```
@Override
public boolean equals(Object obj) {
 if(!(obj instanceof Krug)) return false;
 Krug drugi = (Krug)obj;
 return this.getIme().equals(drugi.getIme());
}
```

U ovoj implementaciji zadaću utvrđivanja jednakosti delegirali smo metodi `String.equals(...)` koju pozivamo kako bi usporedila imena trenutnog kruga i predanog kruga.

Koja je od ovih implementacija prikladnija ovisi o problemu koji rješavamo i ni na koji način nije unaprijed propisana.

Spomenimo još da između metoda `equals` i `hashCode` postoji ugovor koji treba poštivati: ako za objekte `a` i `b` poziv `a.equals(b)` vraća `true` odnosno ako tvrdi da su ta dva objekta jednaka, tada bi moralo vrijediti i `a.hashCode() == b.hashCode()` odnosno da su njihovi sažetci jednaki. To pak znači da se vrijednost sažetka treba računati temeljem istih vrijednosti (odnosno članskih varijabli) temeljem kojih se radi i usporedba objekata (što je preporučljivo) ili pak temeljem nekog podskupa tih varijabli (što se ne preporučuje).

U konkretnom slučaju razreda `Krug`, prethodno pravilo značilo bi da prvoj verziji metode `equals` koju smo ovdje prikazali odgovara i implementacija metode `hashCode` koju smo dali kod opisa te metode. Alternativnoj implementaciji metode `equals` koju smo naveli i koja jednakost utvrđuje temeljem imena odgovarala bi sljedeća implementacija metode `hashCode`.

```
@Override
public int hashCode() {
 return getIme().hashCode();
}
```

U ovoj implementaciji zadaću utvrđivanja sažetka delegiramo implementaciji te metode u razredu `String` tražeći je da izračuna sažetak imena koje je pridruženo trenutnom krugu.

Metoda `equals` u razredu `Object` implementirana je na način da vraća rezultat usporedbe adrese trenutnog i predanog objekta, kako je prikazano u nastavku.

```
@Override
public boolean equals(Object obj) {
 return this == obj;
}
```

Metode `wait`, `notify` i `notifyAll`

Navedene metode služe za implementaciju sinkronizacijskih mehanizama i detaljnije ćemo ih obraditi prilikom upoznavanja s višedretvenošću.

`protected void finalize()`

Metoda predstavlja finalizator. Finalizatori objekata su metode koje su konceptualno slične destruktorima i detaljnije smo ih već spomenuli u prethodnom dijelu poglavlja.

`Class<?> getClass();`

Ova metoda omogućava nam pristup do objekta koji sadrži informacije o razredu kojega je objekt primjerak. Programski jezik Java u paketu `java.lang` sadrži i razred `Class`. Za svaki

razred koji se koristi u programu, virtualni stroj će stvoriti po jedan primjerak razreda `Class` koji će napuniti s informacijama o tom razredu. Evo ilustrativnog primjera.

```
public class Ispis2 {

 public static void main(String[] args) {
  Krug k = new Krug("k1", 2, 2, 10);

  Class<?> razred = k.getClass();
  System.out.println("Naziv razreda je: " + razred.getName());
  System.out.println("Paket razreda je: " + razred.getPackage().getName());
 }

}
```

Program će rezultirati sljedećim ispisom.

```
Naziv razreda je: hr.fer.zemris.java.tecaj_1.Krug
Paket razreda je: hr.fer.zemris.java.tecaj_1
```

Dobiveni objekt može se koristiti za dobivanje još čitavog niza informacija, poput: koje su sve metode definirane u razredu, koji konstruktori postoje, je li razred anoniman i slično kao i za obavljanje niza drugih zadaća na koje se u ovom trenutku nećemo osvrtati.

Objekt koji nudi informacije o razredu možemo dobiti na dva načina. Ako imamo referencu `r` na primjerak nekog razreda, referencu na objekt koji opisuje razred čiji je on primjerak možemo dobiti pozivom `r.getClass()`. Ako pak unaprijed znamo razred za koji želimo dobiti takve informacije, možemo koristiti ključnu riječ `class` kako je to prikazano na sljedećem primjeru.

```
public class Ispis3 {

 public static void main(String[] args) {
  Class<?> razred = String.class;
  System.out.println("Naziv razreda je: " + razred.getName());
  System.out.println("Paket razreda je: " + razred.getPackage().getName());
 }

}
```

Program će rezultirati sljedećim ispisom.

```
Naziv razreda je: java.lang.String
Paket razreda je: java.lang
```

Sada se možemo osvrnuti i na prethodno uvedeni operator `instanceof`. Imamo li referencu `r` na primjerak nekog razreda, možemo se pitati sljedeće.

- Je li objekt `r` primjerak razreda `Krug` ili nekog razreda koji je direktno ili indirektno izveden iz tog razreda (odnosno je li taj objekt *barem* `Krug`)?

  ```
  boolean je = r instanceof Krug;
  ```

- Je li objekt `r` baš primjerak razreda `Krug` a ne primjerak nečega što ili uopće nije krug ili je izvedeno iz kruga (odnosno je li taj objekt *točno* `Krug`)?

  ```
  boolean je = r.getClass() == Krug.class;
  ```

  Ovo posljednje ispitivanje treba provesti tek nakon što se utvrdi da `r` nije `null`-referenca. objekti koji opisuju pojedine razrede mogu se uspoređivati i direktno operatorom `==` jer virtualni stroj jamči da će za svaki razred u memoriji postojati upravo jedan takav objekt pa jednakost možemo ispitivati usporedbom referenci odnosno bez uporabe metode `equals`.

Inicijalizacijski blokovi

Do sada smo vidjeli jedan način na koji se primjerci razreda mogu inicijalizirati: kod koji je potrebno izvršiti prilikom stvaranja novih primjeraka stavljali smo u konstruktore. Osim takvog načina, na raspolaganju nam stoje i inicijalizacijski blokovi: to su blokovi koda koji su u vitičastim zagradama napisani na razini razreda. Pogledajmo primjer.

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.util.Date;
 4
 5 public class Stoperica {
 6
 7  private Date datum;
 8  private long pocetniTrenutak;
 9
10  {
11   datum = new Date();
12  }
13
14  {
15   pocetniTrenutak = System.currentTimeMillis();
16  }
17
18  public long protekloVrijeme() {
19   return System.currentTimeMillis() - pocetniTrenutak;
20  }
21
22  public static void main(String[] args) {
23
24   Stoperica s = new Stoperica();
25   double suma = 0.0;
26   for(long l = 0; l < 1_000_000L; l++) {
27    suma += Math.sin(0.1);
28   }
29   long trajanje = s.protekloVrijeme();
30
31   System.out.println("Stoperica je stvorena na: " + s.datum);
32   System.out.println("Proteklo vrijeme u ms: " + trajanje);
33   System.out.println("Rezultat je: " + suma);
34  }
35
36 }
37
```

Program prikazuje implementaciju jednostavne štoperice: prilikom stvaranja, objekt pamti trenutno stanje sata računala. Razred nudi metodu `protekloVrijeme` čijim se pozivom dobiva vrijeme (u milisekundama) proteklo od trenutka stvaranja objekta pa da trenutka poziva te metode.

U retcima 7 i 8 definirali smo dvije članske varijable: `datum` koja pamti datum i vrijeme stvaranja objekta kao primjerak razreda `Date` te `pocetniTrenutak` koja pamti stanje na satu računala. Umjesto konstruktora, napisali smo dva inicijalizacijska bloka: prvi se proteže od retka 10 do retka 12 i inicijalizira varijablu `datum`; drugi se proteže od retka 14 do retka 16 i inicijalizira varijablu `pocetniTrenutak`. Inicijalizacijski blok može sadržavati jednu ili više naredbi (čak i petlje, pozive metoda i slično) koje je potrebno napraviti prilikom inicijalizacije objekta. Inicijalizacijski blokovi mogu biti prisutni i u slučaju kada u razredu postoje definirani konstruktori. Po definiciji, inicijalizacijski blokovi se izvode redosljedom kojim su napisani u razredu prije no

što se izvede prva linija koda konstruktora. Stoga se inicijalizacijski blokovi mogu koristiti i kao način za dijeljenje koda između više konstruktora. Sljedeći primjer ilustira redosljed izvođenja koda u prisustvu inicijalizacijskih blokova.

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class RazredA {
 4
 5  {
 6   System.out.println("A: prvi blok");
 7  }
 8
 9  public RazredA() {
10   System.out.println("A: konstruktor.");
11  }
12
13  {
14   System.out.println("A: drugi blok");
15  }
16
17 }
18
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class RazredB extends RazredA {
 4
 5  {
 6   System.out.println("B: prvi blok");
 7  }
 8
 9  public RazredB() {
10   System.out.println("B: konstruktor.");
11  }
12
13  {
14   System.out.println("B: drugi blok");
15  }
16
17 }
18
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class PrimjerIB {
 4
 5  public static void main(String[] args) {
 6   new RazredB();
 7  }
 8
 9 }
10
```

U danom primjeru, razred `RazredB` nasljeđuje razred `RazredA`. Oba imaju po dva inicijalizacijska bloka i definiran konstruktor. Glavni program stvara primjerak razreda `RazredB`. Ispis programa prikazan je u nastavku.

```
A: prvi blok
A: drugi blok
```

```
A: konstruktor.
B: prvi blok
B: drugi blok
B: konstruktor.
```

Ponovimo još jednom: inicijalizatorski blokovi izvode se onim redosljedom kojim su napisani u kodu kao da su napisani na početku svih konstruktora. Posljedica je da se prilikom stvaranja primjerka razreda `RazredB` najprije poziva konstruktor razreda `RazredA` što uzrokuje izvođenje svih inicijalizatorskih blokova napisanih za objekte razreda `RazredA`, potom se izvode naredbe napisane u njegovom konstruktoru, potom se izvode svi inicijalizatorski blokovi napisani za objekte razreda `RazredB` i potom se izvode naredbe napisane u konstruktoru razreda `RazredB`.

Osim inicijalizatorskih blokova koji služe za inicijalizaciju primjeraka razreda, moguće je pisati i inicijalizatorske blokove koji služe za inicijalizaciju samih razreda. Ti se inicijalizatorski blokovi nazivaju statički inicijalizatorski blokovi, izvode se prvi puta kada se u virtualnom stroju spomene razred kojemu pripadaju i izvode se samo jednom tijekom života programa. Virtualni stroj garantira da će se ti blokovi izvesti prije no što se kreneu stvaranje bilo kojeg primjerka razreda u kojemu su napisani. Ove blokove prepoznajemo po tome što započinju prefiksom `static` i tipično služe za inicijalizaciju statičkih varijabli razreda o kojima će biti riječi kasnije. Za sada ćemo samo dati primjer bez daljnjih pojašnjenja. Čitatelja se upućuje da još jednom pročita ovo poglavlje nakon što se upozna s pojmom statičkih varijabli i statičkih metoda. Statički inicijalizacijski blokovi ponašaju se baš kao i statičke metode.

```
 1 public class RazredC {
 2
 3   // Ovo je inicijalizacijski blok koji će se izvesti pri
 4   // svakom stvaranju primjerka razreda; u njemu je dostupna
 5   // referenca this na trenutni primjerak.
 6   {
 7     // naredbe ...
 8   }
 9
10   // Ovo je statički inicijalizacijski blok. On će se izvesti
11   // samo jednom, prije stvaranja bilo kojeg primjerka ovog
12   // razreda (dapače, prvi puta kada se ovaj razred spomene
13   // u programu)
14   static {
15     // naredbe ...
16   }
17
18 }
```

Poglavlje 6. Upravljanje pogreškama. Iznimke.

Pogreške su danas sastavni dio svakog programa. Pri tome ne mislimo na pogreške koje je unio programer -- te se pogreške dadu na odgovarajući način ispraviti. Pogreške o kojima ovdje govorimo su pogreške koje se javljaju dinamički, tijekom izvođenja programa; primjerice, pišemo program koji alocira memoriju: što učiniti ako alokacija ne uspije? Pišemo program koji čita podatke iz datoteke na disku: što učiniti ako operacijski sustav dojavi da sadržaj više nije moguće čitati jer je medij na kojem je datoteka pohranjena oštećen? Pišemo program koji podatke dohvaća protokolom TCP: što učiniti ako veza pukne prije kraja prijenosa? Ili možda najčešće -- pišemo program koji prima argumente preko naredbenog retka: što učiniti ako korisnik zada pogrešan broj argumenata ili pak zada argumente pogrešnog tipa (recimo, zada znakovni niz koji se na da pretvoriti u decimalni broj)? To su pitanja o kojima ćemo govoriti u ovom poglavlju.

U starijim programskim jezicima koji nemaju naprednijih mehanizama, pojava pogrešaka dojavljivala se putem različitih statusa, bilo u obliku povratne vrijednosti funkcije, bilo u obliku upisa oznake pogreške na neku zadanu memorijsku lokaciju. Evo primjera iz programskog jezika C.

```
double solve(double a, double b, double c, int *error) {
  double d = b * b - 4 * a * c;
  if(d<0) {
    *error = 1;
    return 0;
  } else {
    *error = 0;
    return (-b + sqrt(d)) / (2*a);
  }
}
```

Primjer ilustrira funkciju `solve` koja prima parametre kvadratne jednadžbe i vraća njezino veće rješe, uz pretpostavku da realno rješenje uopće postoji. Kako rezultat izračuna funkcija vraća preko povratne vrijednost, popis argumenata joj je proširen još s jednim argumentom: pokazivačem na mjesto u koje treba upisati status pogreške (0 ako nema pogreške, 1 ako ima pogreške). Neovisno o tome ima li pogreške ili ne, metoda nešto mora i vratiti pa je prikazana implementacija koja vraća vrijednost 0 ako je nastupila pogreška.

Klijentski kod koji koristi ovu funkciju tipično je oblika prikazanog u nastavku.

```
int status;
double rjesenje = solve(a, b, c, &status);
if(*status) {
  // Imamo pogrešku; napravi nešto...
} else {
  // Sve je OK; iskoristi dobiveno
}
```

U kodu se deklarira spremnik koji će primiti status pogreške, potom se poziva funkcija i konačno se ispituje je li pogreška nastupila. Ako je, poduzima se odgovarajuća akcija, a ako nije, rezultat se koristi.

Drugi primjer pisanja funkcija koje mogu završiti s pogreškom ilustriran je u nastavku.

```
int vratiZnak() {
  static int brojac = 20;
  static char trenutni = 'A';
  if(brojac < 1) {
```

```
    return -1;
  }
  brojac--;
  return trenutni++;
}
```

U ovom slučaju dana je funkcija koju je legalno pozvati 20 puta. Tijekom legalnih poziva, funkcija će vratiti neko veliko slovo (ova konkretna implementacija kreće od slova 'A' i redom generira njegove sljedbenike). Problem s kojim se funkcija mora znati nositi je sljedeći: što ako korisnik funkciju pozove 21. puta? Kako je funkciji dozvoljeno da tijekom legalnih poziva vraća bilo koji znak, funkcija pojavu pogreške ne može signalizirati preko povratne vrijednosti ako je ona po tipu deklarirana kao `char`. Jedno moguće rješenje je funkciji kao parametar proslijediti pokazivač na lokaciju na koju će upisati status izvođenja, što odgovara prethodno opisanom pristupu. Druga mogućnost je zlouporaba povratne vrijednost. Kako funkcija legalno može vratiti bilo koji znak, povratna vrijednost je proširena na veći komatibilan tip: odabrano je da funkcija vraća `int` kao povratnu vrijednost. Ovaj tip podataka omogućava prijenos svih legalnih vrijednosti za tip `char` a ima mogućnost prikaza i drugih vrijednosti. To je iskorišteno na način da funkcija vraća ili legalnu vrijednost ili -1 ako je nastupila pogreška. Tipičan klijentski kod koji bi koristio ovakvu funkciju prikazan je u nastavku.

```
int statusIliZnak;
char znak;
for(int i = 0; i < 5; i++) {
  statusIliZnak = vratiZnak();
  if(statusIliZnak == -1) {
    // Pogreška! Obradi pogrešku, dohvat 5 znakova nije uspio
  } else {
    znak = (char)statusIliZnak;
    // Sada iskoristi pročitani znak...
  }
}
```

Za ovakav oblik funkcija uobičajeno je da se u kodu deklarira najprije preveliki tip podatka ("trebam auto, deklariram nosač aviona"), potom se funkcija poziva pa se provjerava što je vraćeno. Ako je vraćena ilegalna vrijednost za izvorno očekivani tip podatka ("nisam dobio auto"), tada je to pogreška i ona se nekako obrađuje. Ako vraćena vrijednost nije ilegalna za izvorno očekivani tip podatka, tada se ona ukalupljuje u izvorno očekivani tip podataka ("auto") i dalje se koristi. Ako vas se ovo čini čudan pristup, možda je dovoljno spomenuti da je jedna od najčešće korištenih funkcija za čitanje znakova iz datoteke (`getChar(FILE *f)` napisana upravo na taj način. Štoviše, takva je implementacija iste funkcije ušla i u standardni Java API.

Sada kada smo se prisjetili koji su načini dojave pogreške, pogledajmo primjer koda koji radi konstrukciju malo složenijeg objekta. Primjer u nastavku prikazan je u pseudokodu. Pretpostavite da funkcije za alokaciju odgovarajućih resursa vraćaju `NULL` ako resurs nije moguće zauzeti (što predstavlja pojavu pogreške).

```
resurs * r1 = alocirajResurs1();
if(r1 == NULL) goto ERR_R1;
resurs * r2 = alocirajResurs2();
if(r2 == NULL) goto ERR_R2;
resurs * r3 = alocirajResurs3();
if(r3 == NULL) goto ERR_R3;
resurs * r4 = alocirajResurs4();
if(r1 == NULL) goto ERR_R4;
...
resurs * rN = alocirajResursN();
if(rN == NULL) goto ERR_RN;
```

```
// Ovdje ide složena obrada koja troši sve resurse
// ...

  oslobodiResurs(rN);
ERR_RN:
  oslobodiResurs(rN-1);
ERR_..:
  ...
ERR_R4:
  oslobodiResurs(r3);
ERR_R3:
  oslobodiResurs(r2);
ERR_R2:
  oslobodiResurs(r1);
ERR_R1:

  // Kraj funkcije
```

Opisani kod je čest u jezicima kod kojih je nužno eksplicitno upravljati resursima (primjerice, zauzećem memorije). Naravno, umjesto opisanog koda koji se oslanja na `goto`, ljudi znaju posezati za svačime, posebice u jezicima koji ne nude naredbu `goto`. Evo ekvivalentnog primjera izvedenog bez `goto`.

```
resurs * r1 = alocirajResurs1();
if(r1 == NULL) {
  // Povratak; posao nije obavljen
}
while(true) {
  resurs * r2 = alocirajResurs2();
  if(r2 == NULL) break;
  while(true) {
    resurs * r3 = alocirajResurs3();
    if(r3 == NULL) break;
    // Obavi posao; koristi sve resurse
    // ...
    oslobodiResurs(r3);
    break;
  }
  oslobodiResurs(r2);
  break;
}
oslobodiResurs(r1);

// Kraj funkcije
```

Primjer ilustrira situaciju u kojoj trebamo zauzeti tri resursa da bismo obavili posao, i potom ih sve osloboditi. Petlje `while` samo služe kao omotač kako bi se s `break` moglo preskočiti dio koda koji ne želimo izvesti. Takav kod se doista pisao. I takav kod je vrlo vrlo nečitljiv, nejasan i podatan za unos pogrešaka.

I konačno, kada smo prisiljeni eksplicitno raditi s pogreškama počev od mjesta na kojem je pogreška nastala, to nije uvijek jednostavno jer odgovor na pitanje: što bih sada tu trebao poduzeti često nije jasan na tom mjestu. U takvim situacijama uobičajeno je pisati funkcije koje pozivaju druge funkcije koje mogu izazvati pogrešku pa proširivati listu argumenata početne funkcije kako bi u tom slučaju ona svom pozivatelju proslijedila pogrešku koju je njoj dojavila funkcija koju je ona zvala. I tako u dubinu: ako funkcija `f1` poziva funkciju `f2` koja poziva funkciju `f3` koja poziva funkciju `f4` koja poziva funkciju `f5` koja poziva funkciju `f` koja može izazvati pogrešku, svaku od ovih funkcija morat ćemo napisati tako da provjeri je li pozvana funkcija uspjela pa da svom pozivatelju na neki način dojave što se dogodilo. Pri tome statusi pogreša-

ka tipski ne moraju biti isti. Recimo, zovemo funkciju `ucitaj` koja učitava parametre iz konfiguracijske datoteke zapisane u obliku XML. Ta funkcija poziva funkciju `izgradiXMLStablo` koja pak poziva funkciju `parsirajXML` koja poziva funkciju `procitajSljedeciTag`. Recimo da funkcija `procitajSljedeciTag` dojavi pogrešku: "Tag nije zatvoren znakom >" svom pozivatelju. Funkcija `parsirajXML` tada bi trebala svom pozivatelju javiti "XML dokument nije valjan". Funkcija `izgradiXMLStablo` tada bi svom poziovatelju trebala vratiti poruku poput "Nije moguće izgraditi XML stablo" na što bi funkcija `ucitaj` korisniku trebala dojaviti "Parametre nije moguće učitati jer je datoteka neispravna". Uočite iz ovog primjera da postoje situacije u kojima način konkretne obrade pogreške ne može biti poznat na mjestu gdje se pogreška dogodi -- funkcija za čitanje sljedećeg taga ne zna koja je namjena XML-a koji se obrađuje i korisniku ne može ponuditi odgovarajuću poruku pogreške. Umjesto toga, uvode se ovakvi lanci prijenosa statusa pogrešaka kako bi se pogreška mogla obraditi na mjestu gdje postoji dovoljno informacija za njezinu obradu. U nižim programskim jezicima pisanje ovakvih funkcija dodatno će se zakomplicirati jer će prilikom pojave pogreške trebati pisati i dio koda koji će se pobrinuti da se u svakom koraku svi zauzeti resursi oslobode prije no što se pozivatelju vrati status pogreške; u suprotnom ćemo imati curenje resursa i općenito loš program.

Kakvo nam rješenje na opisane probleme nudi Java?

Iznimke

U programskom jeziku Java definira se koncept "iznimne situacije" ili kraće, *iznimke*. Ideja je mehanizam upravljanja pogreškama i njihovom obradom razdvojiti od argumenata i povratnih vrijednosti metoda. Upareno s postojanjem automatskog upravljanja resursima (a posebice memorijom o čijem oslobađanju programer ne treba brinuti), u Javi je moguće pisati vrlo čitak i jednostavan kod koji korektno obrađuje pogreške.

Pojava pogreške u Javi se može modelirati iznimnom situacijom. Java platforma definira čitav niz iznimnih situacija i razreda čiji se primjerci koriste za njihov opis. Najopćenitiji opis iznimne situacije predstavlja razred `Throwable`. Iz tog razreda izvedeni su razredi `Error` i `Exception`. Oba razreda dalje imaju razrađeno stablo nasljeđivanja. Razredi tipa `Error` (ili iz njega izvedeni) koriste se za opisivanje iznimnih situacija koje su rezultati ozbiljnih problema koje tipična aplikacija ne bi trebala obrađivati; primjerice, postoji problem sa strojnim kodom programa koji se izvodi (primjerice: nije moguće učitati strojni kod potrebnog razreda ili je datoteka sa izvršnim kodom pokvarena), nestalo je memorije, nema više mjesta na stogu i slično. Razredi tipa `Exception` (ili iz njega izvedeni) služe za modeliranje "normalnijih" pogrešaka koje bi program mogao obraditi na prikladan način (primjerice, otvaramo datoteku sa slikom koja međutim ne postoji na disku).

Sav kod koji se za upravljanje pogreškama oslanja na iznimke, u trenutku detekcije pogreške stvara primjerak prikladnog razreda koji u semantičkom smislu predstavlja najbolji opis pogreške, puni ga odgovarajućim opisom i potom od virtualnog stroja traži pokretanje *obrade iznimke*. Svaki od ovih koraka ilustrirat ćemo primjerom prikazanim u nastavku.

Primjer 6.1. Primjer stvaranja iznimke

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.util.Scanner;
 4
 5 public class Iznimke1 {
 6
 7  public static void main(String[] args) {
 8   Scanner s = new Scanner(System.in);
 9   while(true) {
10    System.out.print("Unesite a, b i c: ");
11    double a = s.nextDouble();
12    double b = s.nextDouble();
```

```
13     double c = s.nextDouble();
14     System.out.println("Korijen je: " + solve(a,b,c));
15     System.out.print("Ako želite novi unos, upišite DA. ");
16     if(!"DA".equals(s.next())) {
17      break;
18     }
19    }
20    s.close();
21   }
22
23   public static double solve(double a, double b, double c) {
24    double d = b * b - 4 * a * c;
25    if(d<0) {
26     throw new IllegalArgumentException(
27       "Kvadratna jednadžba nema realnih korijena."
28     );
29    }
30    return (-b + Math.sqrt(d)) / (2*a);
31   }
32
33 }
34
```

Primjer koji ilustrira normalan rad programa prikazan je u nastavku.

```
Unesite a, b i c: 1 -1 -2
Korijen je: 2.0
Ako želite novi unos, upišite DA.
```

U prikazanom primjeru dana je metoda `solve` koju smo već prethodno spomenuli: ona temeljem koeficijenta kvadratne jednadžbe računa i vraća veći od dva realna korijena, ako oni postoje. Ako ne, metoda izaziva iznimku. Pogledajmo malo bolje taj drugi scenarij. U slučaju da je ustanovljeno da je diskriminanta jednadžbe negativna, dolazimo do sljedećeg bloka.

```
throw new IllegalArgumentException(
  "Kvadratna jednadžba nema realnih korijena."
);
```

`IllegalArgumentException` predstavlja naziv razreda koji koristimo za opisivanje pogreške koja je nastupila -- dobili smo parametre koji ne predstavljaju kvadratnu jednadžbu s realnim korijenima. Operatorom `new` stvaramo primjerak tog razreda. Prilikom stvaranja primjerka razreda koji predstavlja iznimku, taj će primjerak pokupiti i trenutne informacije o mjestu u kodu i stanju na stogu. Kao argument konstruktoru predajemo opis pogreške koji želimo da se zapamti u primjerku razreda `IllegalArgumentException` koji upravo stvaramo. Nakon što je objekt stvoren, referenca na taj objekt predaje se operatoru `throw`. Ovaj operator na razini virtualnog stroja započinje obradu iznimke. Pa pogledajmo najprije na primjeru što će se dogoditi. Da bismo izazvali pogrešku, trebamo predati takve koeficijente uz koje kvadratna jednadžba neće imati realne korijene. Primjerice, krenemo li od (x-2i)(x+2i) dobit ćemo x^2 + 4, odnosno koeficijente: 1, 0, 4. Rezultat izvođenja programa prikazan je u nastavku.

```
Unesite a, b i c: 1 0 4
Exception in thread "main" java.lang.IllegalArgumentException:
      Kvadratna jednadžba nema realnih korijena.
 at hr.fer.zemris.java.primjeri.Iznimke1.solve(Iznimke1.java:26)
 at hr.fer.zemris.java.primjeri.Iznimke1.main(Iznimke1.java:14)
```

Rezultat je rušenje programa uz ispisanu poruku. Iz poruke se može pročitati da je dretva `main` (dretva koja izvodi metodu `main`) naišla na iznimnu situaciju. Pogreška je opisana primjerkom

razreda `IllegalArgumentException`, dodatni opis je `Kvadratna jednadžba nema realnih korijena.`. Tijek izvođenja programa do nastanka iznimne situacije prikazuje popis poziva metoda koji je potom ispisan i koji treba čitati od dna prema vrhu: bila je pozvana metoda `main` u razredu `Iznimke1`; ona je u retku 14 pozvala metodu `solve` istog razreda. Metoda `solve` je u retku 16 izazvala iznimku `IllegalArgumentException`.

Operator `throw` je operator koji ima sličnosti s naredbom `return`: ulaskom u obradu iznimne situacije izvođenje koda se prekida i traži se blok koda koji je zadužen za obradu nastale iznimke. Ako ga se ne pronađe u trenutnoj metodi, metoda se briše sa stoga i pretražuje se njezin pozivatelj; ako se niti tamo ne nađe odgovarajući blok, postupak se nastavlja sve do trenutka kada na stogu više nema metoda -- u tom trenutku virtualni stroj zaustavlja obradu iznimke (kažemo da *hvata iznimku*), terminira trenutnu dretvu koja izvodi taj kod i ispisuje poruku na zaslon. U jednodretvenim programima terminiranje te jedine dretve za posljedicu ima i zaustavljanje čitave aplikacije.

Kako se iznimka može uhvatiti? Java nam na raspolaganje stavlja sljedeći konstrukt.

```
try {
  // dio koda koji moze baciti iznimku
} catch(RazredIznimke varijabla) {
  // dio koda koji obrađuje iznimku
}
```

Pokažimo to na primjeru. Popravit ćemo primjer `Iznimka1` tako da metoda `main`, ako ustanovi da je došlo do iznimne situacije, korisniku ispiše odgovarajuću poruku i potom nastavi s radom.

Primjer 6.2. Primjer obrade iznimke

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.util.Scanner;
 4
 5 public class Iznimke1 {
 6
 7  public static void main(String[] args) {
 8   Scanner s = new Scanner(System.in);
 9   while(true) {
10    System.out.print("Unesite a, b i c: ");
11    double a = s.nextDouble();
12    double b = s.nextDouble();
13    double c = s.nextDouble();
14    try {
15     System.out.println("Korijen je: " + solve(a,b,c));
16    } catch(IllegalArgumentException ex) {
17     System.out.println(
18       "Unijeli ste koeficijente uz koje ne postoje" +
19       "realni korijeni!"
20     );
21    }
22    System.out.print("Ako želite novi unos, upišite DA. ");
23    if(!"DA".equals(s.next())) {
24     break;
25    }
26   }
27   s.close();
28  }
29
30  public static double solve(double a, double b, double c) {
```

```
31    double d = b * b - 4 * a * c;
32    if(d<0) {
33     throw new IllegalArgumentException(
34       "Kvadratna jednadžba nema realnih korijena."
35     );
36    }
37    return (-b + Math.sqrt(d)) / (2*a);
38   }
39
40 }
41
```

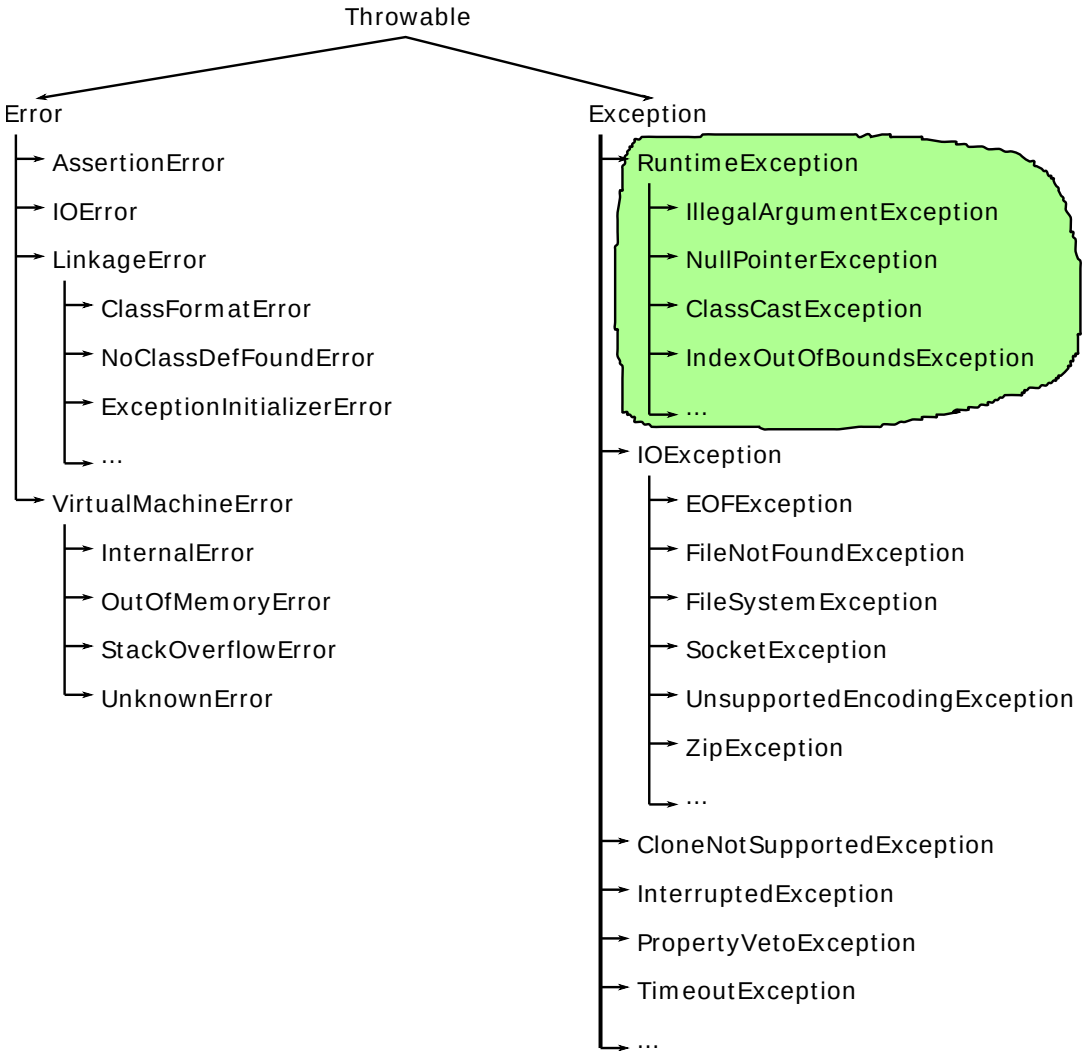
U slučaju nastanka iznimke, metoda `solve` propagirala je nastalu iznimku svojem pozivatelju. Stoga smo u metodi `main` njezin poziv okružili blokom `try { ... } catch(...)` čime smo dobili konstrukt koji se proteže od retka 14 do retka 21. Ako ne dođe do pojave iznimke, kod u dijelu `try` se normalno izvodi i potom se u potpunosti preskače sav kod koji je napisan u dijelu `catch` te se izvođenje nastavlja prvom naredbom koja slijedi čitav konstrukt. Ukoliko dođe do iznimke, jedino što očekujemo jest iznimka `IllegalArgumentException`. Stoga smo napisali `catch` blok koji hvata upravo tu iznimku (ili bilo koju koja je iz nje izvedena). Ako dođe do iznimke `IllegalArgumentException`, taj će se blok aktivirati i on će na ekran korisniku ispisati prikladnu poruku. Ulaskom u blok `catch`, izazvana iznimna situacija je obrađena. Stoga se nakon izvođenja tog bloka program nastavlja izvoditi na uobičajeni način od prve linije koja slijedi čitav konstrukt (u našem slučaju, redak 22).

Primjer izvođenja ovako korigiranog koda dan je u nastavku.

```
Unesite a, b i c: 1 0 -4
Korijen je: 2.0
Ako želite novi unos, upišite DA. DA
Unesite a, b i c: 1 0 4
Unijeli ste koeficijente uz koje ne postoje realni korijeni!
Ako želite novi unos, upišite DA.
```

Prije no što nastavimo daljnju diskusiju o iznimkama, pogledajmo kako su iznimke organizirane u Javi. Sljedeća slika može poslužiti kako bismo stekli grubu predodžbu (na slici nisu prikazane sve postojeće iznimke).

Slika 6.1. Struktura iznimki u Javi



Već smo u uvodnom dijelu spomenuli: razred `Throwable` predstavlja najopćenitiji razred čiji primjerak može biti argument operatora `throw`. Iz njega su potom izvedena dva razreda: `Error` i njegovo podstablo predstavljaju opise teških iznimnih situacija koje tipičan program neće moći niti htjeti obraditi na smisleni način. S druge strane, razred `Exception` i njegovo podstablo predstavlja iznimne situacije koje programi mogu i tipično obrađuju na prikladan način. Stoga kada govorimo o iznimkama, osim ako nije eksplicitno navedeno drugačije, podrazumijevat ćemo iznimke iz ove druge grane.

Unutar stabla nasljeđivanja razreda `Exception` ističe se jedna posebna grupa iznimaka: ona izvedena iz razreda `RuntimeException`. Iznimke koje su izvedene iz tog razreda opisuju situacije koje se mogu dogoditi praktički u svakom retku Java programa. Pogledajmo jedan ilustrativan primjer.

Primjer 6.3. Iznimke porodice `RuntimeException`

```
1 package hr.fer.zemris.java.primjeri;
2
3 public class Iznimke2 {
4
5  public static void main(String[] args) {
6   obrada();
7  }
```

```
 8
 9  private static void obrada() {
10   int broj = (int)(Math.random()*300 + 0.5) % 3;
11   switch(broj) {
12   case 0: obrada1(); break;
13   case 1: obrada2(null); break;
14   case 2: obrada3(new Integer(5)); break;
15   }
16  }
17
18  private static void obrada1() {
19   int[] polje = new int[2];
20   polje[3] = 17;
21  }
22
23  private static void obrada2(Object o) {
24   String s = o.toString();
25   System.out.println(s);
26  }
27
28  private static void obrada3(Object o) {
29   Double d = (Double)o;
30   System.out.println(d.doubleValue()+1);
31  }
32
33 }
34
```

Metoda `obradi` ovisno o slučajno generiranom broju poziva jednu od tri metode koje obavljaju različite poslove.

Metoda `obradi1` primjer je metode koja ilustrira pojavu iznimke `ArrayIndexOutOfBoundsException` koja je izvedenica od `IndexOutOfBoundsException`. U retku 19 deklarira se polje cijelih brojeva i inicijalizira na polje veličine dva elementa. To znači da su legalni indeksi koji se mogu koristiti za pristupanje elementima polja 0 i 1. Međutim, u retku 20 pokušava se postaviti vrijednost elementa čiji je indeks 3: kako polje nije dovoljno veliko da bi sadržavalo i taj element, virtualni stroj će prilikom izvođenja tog retka uočiti pogrešku, prekinuti tu naredbu, generirati iznimku `ArrayIndexOutOfBoundsException` i pokrenuti njezinu iznimku. Pozivom metode `obradi1` program će se stoga srušiti uz poruku prikazanu u nastavku.

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 3
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada1(Iznimke2.java:20)
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada(Iznimke2.java:12)
 at hr.fer.zemris.java.primjeri.Iznimke2.main(Iznimke2.java:6)
```

Metoda `obradi2` primjer je metode koja ilustrira pojavu iznimke `NullPointerException`. Metoda kao argument prima referencu na primjerak bilo kojeg razreda, pokušava za taj primljeni primjerak dohvatiti njegov tekstovni opis i potom taj opis ispisuje na ekran. Prilikom pozivanja metode `obradi2`, metoda `obrada` toj metodi umjesto reference na neki postojeći objekt predaje referencu `null`. Posljedica je da će se izvođenjem retka 24 pokušati pozvati metoda `toString` nad nepostojećim objektom. Virtualni stroj opet će u trenutku tog poziva detektirati pogrešku, prekinut će poziv, stvoriti opis pogreške kao primjerak razreda `NullPointerException` i pokrenuti obradu iznimne situacije. Pozivom metode `obradi2` program će se stoga srušiti uz poruku prikazanu u nastavku.

```
Exception in thread "main" java.lang.NullPointerException
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada2(Iznimke2.java:24)
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada(Iznimke2.java:13)
```

```
 at hr.fer.zemris.java.primjeri.Iznimke2.main(Iznimke2.java:6)
```

Metoda `obradi3` primjer je metode koja ilustrira pojavu iznimke `ClassCastException`. Metoda kao argument prima referencu na primjerak bilo kojeg razreda. Međutim, metoda očekuje da će dobiti primjerak razreda `Double` što je vidljivo iz retka 29 gdje dobiveni objekt ukalupljuje u primjerak razreda `Double`. Potom u retku 30 dohvaća pohranjeni decimalni broj, uvećava ga za 1 i rezultat ispisuje na ekran. Kako prilikom poziva ove metode metoda `obrada` kao argument prosljeđuje primjerak razreda `Integer`, izvođenjem retka 29 virtualni stroj će uočiti da se primjerak razreda `Integer` pokušava ukalupiti u primjerak razreda `Double` što je nemoguće. Stoga će na tom mjestu prekinuti izvođenje te operacije, stvoriti opis pogreške u obliku primjerka razreda `ClassCastException` i pokrenuti obradu nastale iznimne situacije. Pozivom metode `obradi2` program će se stoga srušiti uz poruku prikazanu u nastavku.

```
Exception in thread "main" java.lang.ClassCastException:
    java.lang.Integer cannot be cast to java.lang.Double
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada3(Iznimke2.java:29)
 at hr.fer.zemris.java.primjeri.Iznimke2.obrada(Iznimke2.java:14)
 at hr.fer.zemris.java.primjeri.Iznimke2.main(Iznimke2.java:6)
```

Kako se opisane iznimke mogu pojaviti praktički u svakom retku koda, Java specifikacija ne zahtjeva da metode u kojima se takve iznimke mogu pojaviti na bilo koji način to naznače. Iznimka `RuntimeException` i sve iznimke koje pripadaju u njezino stablo nasljeđivanja nazivaju se *neprovjeravane iznimke* (engleski termin je *unchecked exceptions*). Sve ostale iznimke (izuzev podstabla `Error`) nazivaju se *provjeravane iznimke*. Za provjeravane iznimke pravila igre su malo drugačija. Pogledajmo jednostavan primjer. Neka smo napisali sljedeću metodu.

```
1 private static char dohvatiZnak() {
2   if(Math.random() < 0.1) {
3     throw new java.io.IOException(
4       "Pogreška pri generiranju znaka."
5     );
6   } else {
7     return 'A';
8   }
9 }
```

Metoda u 10% slučajeva generira iznimku `IOException` a u 90% slučajeva "generira" znak koji vraća pozivatelju. Kako iznimka `IOException` ne spada u neprovjeravane iznimke, napisani kod nećemo moći prevesti: jezični prevodioc dojavit će pogrešku u retku 3:

```
Unhandled exception type IOException
```

. Ako se u metodi može dogoditi provjeravana pogreška (bilo zato što je metoda sama izaziva, bilo zato što metoda poziva neku drugu metodu koja može izazvati takvu iznimku, metoda je dužna ili iznimku uhvatiti i obraditi, ili deklarirati da je i ona sama može generirati. Kako napraviti obradu, to već znamo: poziv `throw` trebalo bi uhvatiti blokom `try-cacth`. Međutim, u našem primjeru, to nije opcija jer u toj metodi ne znamo što bismo smisleno napravili kao obradu. Stoga je potrebno prepraviti prototip metode tako da deklariramo da metoda može izazvati navedenu iznimku. Korektno rješenje prikazano je u nastavku.

```
1 private static char dohvatiZnak() throws java.io.IOException {
2   if(Math.random() < 0.1) {
3     throw new java.io.IOException(
4       "Pogreška pri generiranju znaka."
5     );
6   } else {
7     return 'A';
8   }
9 }
```

U primjeru je korišteno puno ime razreda iznimke (`java.io.IOException`) jer nismo prikazali cijeli razred niti naredbe `import`. Da smo na početku dodali `import java.io.IOException;`, dalje u kodu smo mogli koristiti samo `IOException`.



Provjeravane i neprovjeravane iznimke

Iznimka `RuntimeException` i sve iznimke koje pripadaju njezinom podstablu nasljeđivanja nazivamo *neprovjeravane* iznimke.

Ako se u metodi može dogoditi bilo koja iznimka koja ne pripada u neprovjeravane iznimke niti u iznimke porodice `Error`, metoda je dužna te iznimke ili obraditi, ili deklarirati da ih izaziva.

Deklariranje iznimaka koje metoda može izazvati radi se navođenjem ključne riječi `throws` nakon čega dolazi zarezima razdvojen popis iznimaka koje metoda izaziva; tek nakon toga dolazi tijelo metode. Npr.

```
void metoda() throws IOException, TimeoutException {
   ...
}
```

Obrada više iznimaka

Pretpostavimo da imamo metodu `obrada` koja može izazvati tri različite iznimke: `ArrayIndexOutOfBoundsException`, `NullPointerException` te `ClassCastException`; tu situaciju imali smo upravo u primjeru 6.3. Kako postići različitu obradu za pojedinu iznimku? Umjesto jednostavne strukture naredbe `try-catch` možemo koristiti složeniju inačicu koja navodi više blokova `catch`. Evo primjera u nastavku.

```
public static void main(String[] args) {
 try {
  obrada();
 } catch(ArrayIndexOutOfBoundsException ex) {
  // prvi način obrade
 } catch(NullPointerException ex) {
  // drugi način obrade
 } catch(ClassCastException ex) {
  // treći način obrade
 }
}
```

Za svaku različitu vrstu iznimke moguće je navesti jedan blok `catch` koji deklarira tu iznimku i potom sadrži kod koji će napraviti odgovarajuću obradu. Prilikom pojave iznimke, navedeni se blokovi pretražuju od prvoga prema posljednjem u potrazi za blokom koji je primjenjiv na iznimku koja je izazvana. Neki blok je primjenjiv na iznimku ako deklarira da obrađuje upravo tip iznimke koja se dogodila ili bilo koji od nadređenih razreda u stablu nasljeđivanja iznimke koja se dogodila. Konkretno, ako je izazvana iznimka tipa `ArrayIndexOutOfBoundsException`, tada će tu iznimku uhvatiti blok `catch` koji deklarira da obrađuje bilo što od sljedećega:

- `ArrayIndexOutOfBoundsException` jer je to upravo tip iznimke koji se dogodio,
- `IndexOutOfBoundsException` jer je `ArrayIndexOutOfBoundsException` njegova direktna izvedenica,
- `RuntimeException` jer je `ArrayIndexOutOfBoundsException` tranzitivno izveden iz njega,
- `Exception` jer je `ArrayIndexOutOfBoundsException` tranzitivno izveden iz njega te
- `Throwable` jer je `ArrayIndexOutOfBoundsException` tranzitivno izveden iz njega.

Da biste se uvjerili u posljednja četiri slučaja, pogledajte još jednom sliku 6.1. Na njoj ćete uočiti da je iz `Throwable` izveden `Exception`, iz njega je izveden `RuntimeException`, iz njega pak `IndexOutOfBoundsException` a iz njega `ArrayIndexOutOfBoundsException` (na slici nije prikazan samo ovaj posljednji korak). Stoga će iznimka `ArrayIndexOutOfBoundsException` biti uhvaćena blokom `catch` koji deklarira bilo koju od ovih iznimaka.

Navedeni postupak pretraživanja ima međutim važnu posljedicu na dopustiv redosljed pisanja blokova `catch`. Pogledajmo sljedeći primjer.

```
public static void main(String[] args) {
 try {
  obrada();
 } catch(RuntimeException ex) {
  // prvi način obrade
 } catch(NullPointerException ex) {
  // drugi način obrade
 } catch(ClassCastException ex) {
  // treći način obrade
 }
}
```

Navedeni primjer je neispravan i takav se kod neće moći prevesti. Jezični prevodioc ustanovit će da se blokovi `catch` koji su deklarirani da hvataju iznimke `NullPointerException` i `ClassCastException` nikada neće aktivirati: u slučaju pojave takvih iznimaka njih će uhvatiti prvi napisani blok jer on hvata bilo što što je po tipu `RuntimeException` (posljednje dvije iznimke to jesu), a blokovi se pretražuju od prvog napisanog prema zadnjem. Želimo li posebno obraditi te dvije iznimke a posebno sve ostale iznimke tipa `RuntimeException`, blokove treba presložiti tako da se najprije navedu najspecifičnije iznimke a potom one općenitije. Ispravan primjer prikazan je u nastavku.

```
public static void main(String[] args) {
 try {
  obrada();
 } catch(NullPointerException ex) {
  // drugi način obrade
 } catch(ClassCastException ex) {
  // treći način obrade
 } catch(RuntimeException ex) {
  // prvi način obrade
 }
}
```

Prilikom obrada više iznimaka, česta je situacija da je obrada za neke iznimke jednaka. Evo primjera koji to ilustrira.

```
public static void main(String[] args) {
 try {
  obrada();
 } catch(ArrayIndexOutOfBoundsException ex) {
  // prvi način obrade; zamislimo složenu obradu iznimke
 } catch(NullPointerException ex) {
  // drugi način obrade
 } catch(ClassCastException ex) {
  // na ovom mjestu programer radi copy&paste prvog načina
  // obrade jer se ova pogreška obrađuje jednako kao i prva
 }
}
```

Navedeni primjer na jednak način pokušava obraditi iznimke `ArrayIndexOutOfBoundsException` i `ClassCastException` dok iznimku `NullPointerException` obrađuje

zasebno. Prikazano idejno rješenje uključuje uporabu načela *kopiraj-i-zalijepi* koje bi u svim udžbenicima koji se bave programiranjem trebalo biti najstrože zabranjeno. Kako bi omogućila izbjegavanje dupliciranja koda, Java 7 uvela je podršku za takozvana blokove s višestrukim hvatanjem (engl. *multi-catch blocks*). Rješenje prethodnog problema uporabom takvog bloka prikazano je u nastavku.

```
public static void main(String[] args) {
 try {
  obrada();
 } catch(ArrayIndexOutOfBoundsException | ClassCastException ex) {
  // prvi način obrade; zamislimo složenu obradu iznimke
 } catch(NullPointerException ex) {
  // drugi način obrade
 }
}
```

Umjesto navođenja jednog tipa, u bloku `catch` dozvoljeno je navesti više tipova koje razdvojimo znakom `|`. Navedeni blok `catch` uhvatit će iznimku bilo kojeg od navedenih tipova.

Bezuvjetno izvođenje koda

Prilikom uporabe metoda koje mogu izazvati iznimku, nije rijetkost da se nađemo u situaciji u kojoj postoji dio koda koji želimo bezuvjetno izvesti: neovisno o tome je li prilikom izvođenja koda došlo do iznimke ili nije. Kao ilustraciju ćemo pogledati primjer 6.4.

Primjer 6.4. Primjer rada s resursima

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.IOException;
 4
 5 public class Iznimke3 {
 6
 7  static class LogFile implements AutoCloseable {
 8
 9   public LogFile() throws IOException {
10    // stvaramo dnevnik na disku ako je to moguće;
11    // ako nije, izazivamo IOException iznimku.
12   }
13
14   public void writeMessage(String message) throws IOException {
15    // kod koji zapisuje u dnevnik; pri pojavi
16    // pogreške, bacamo IOException
17   }
18
19   @Override
20   public void close() throws Exception {
21    // Kod koji zatvara dnevnik; bez ovog poziva
22    // moguće je da poruke ne budu korektno zapisane
23    // i da dnevnik ostane u nekonzistentnom stanju.
24   }
25  }
26
27  public static void main(String[] args) throws Exception {
28   primjer1();
29   primjer2();
30   primjer3();
31   primjer4();
```

```
32   }
33
34   private static void primjer1() throws Exception {
35    // ...
36   }
37
38   private static void primjer2() {
39    // ...
40   }
41
42   private static void primjer3() {
43    // ...
44   }
45
46   private static void primjer4() {
47    // ...
48   }
49
50 }
51
```

Prikazani primjer definira statički ugniježđeni razred `LogFile`. To je implementacija dnevnika poruka koji poruke zapisuje na određeno mjesto u datotečnom sustavu, koristi međuspremnike kako bi osigurao maksimalnu efikasnost i ima vrlo složenu implementaciju.

Glavni program poziva metode koje obavljaju određene poslove, periodički zapisuju podatke u dnevnik i potom zatvaraju dnevnik. Zatvaranjem dnevnika svi će se podatci koji su prethodno još zaostali u međuspremnicima prebaciti u datoteku, dnevnik će se dovesti u konzistentno stanje, datoteka će se zatvoriti i svi korišteni resursi će se otpustiti.

Pogledajmo sada prvu implementaciju klijenta koji koristi dnevnik.

```
34 private static void primjer1() throws Exception {
35   LogFile log = new LogFile();
36   // radi nešto
37   log.writeMessage(" ... poruka ...");
38   // radi nešto
39   log.writeMessage(" ... poruka ...");
40   // radi nešto
41   log.close();
42 }
```

Metoda započinje stvaranje objekta koji predstavlja dnevnik. Ukoliko to ne uspije, dogodit će se iznimka koju će obraditi pozivatelj. Ako je dnevnik stvoren, kreće obrada i povremeno zvanje metode koja zapisuje poruke. Problem navedene metode je što će, u slučaju da se dogodi iznimka, dnevnik ostati otvoren -- nitko nikada neće pozvati metodu `close` čime integritet datoteke na disku može postati narušen.

Bolje rješenje prikazano je u nastavku.

```
44 private static void primjer2() {
45   LogFile log = null;
46   try {
47    log = new LogFile();
48    // radi nešto
49    log.writeMessage(" ... poruka ...");
50    // radi nešto
51    log.writeMessage(" ... poruka ...");
52    // radi nešto
```

```
53    log.close();
54   } catch(Exception ex) {
55    System.out.println("Nastupila je pogreška!");
56    if(log!=null) {
57     try {
58      log.close();
59     } catch(Exception ignorable) {
60     }
61    }
62   }
63 }
```

Posao koji radimo okružili smo `try-catch` blokom. Neovisno o tome jesmo li uspjeli redovno obaviti posao ili je nastupila pogreška, dnevnik zatvaramo (uz zanemarivanje pogreške koja eventualno nastupi prilikom zatvaranja dnevnika jer od toga nemamo oporavak). Uočite da je zbog toga kod koji obavlja posao zatvaranja dupliciran. Dodatno, u slučaju nastupanja pogreške, korisniku na zaslon ispisujemo poruku.

Kako imamo dio koda koji želimo izvršiti neovisno o tome je li se dogodila iznima ili ne, umjesto njegovog dupliciranja možemo iskoristiti mogućnost koju do sada još nismo spominjali: blok `finally`. Rješenje je prikazano u nastavku.

```
65 private static void primjer3() {
66  LogFile log = null;
67  try {
68   log = new LogFile();
69   // radi nešto
70   log.writeMessage(" ... poruka ...");
71   // radi nešto
72   log.writeMessage(" ... poruka ...");
73   // radi nešto
74  } catch(Exception ex) {
75   System.out.println("Nastupila je pogreška!");
76  } finally {
77   if(log!=null) {
78    try {
79     log.close();
80    } catch(Exception ignorable) {
81    }
82   }
83  }
84 }
```

Java specifikacija garantira nam da će se kod u bloku `finally` izvesti neovisno o načinu napuštanja `try-catch` konstrukta. Ako se ne dogodi iznimka, nakon dijela `try` izvest će se kod u bloku `finally`. Ako se dogodi iznimka za koju postoji blok `catch` koji je na nju primjenjiv, on će se izvesti i nakon njega opet kod koji je u bloku `finall`. Ako se dogodi iznimka za koju ne postoji niti jedan primjenjiv blok `catch`, prije no što krene u uobičajenu obradu iznimne situacije izvest će se kod zadanu u bloku `finally`. Na ovaj način imamo garanciju da će dnevnik u svim mogućim scenarijima biti zatvoren.

Automatsko upravljanje resursima

Kako je prethodno dan primjer samo ilustracija obrasca koji se vrlo često pojavljuje u izradi Java-programa (primjerice, kada radimo s datotekema, bazama podataka, mrežnim aplikacijama i slično), počev od Jave 7 uvedeno je dodatno proširenje jezika koje omogućava pisanje još manje koda koje će i dalje rezultirati korektnom uporabom razreda koji predstavljaju resurse koje je potrebno zatvoriti nakon uporabe. U Java API uvedeno je sučelje `AutoCloseable`

koje svim rezredima koji ga implementiraju nalaže da moraju ponuditi implementaciju metode `close` čija je zadaća otpuštanje zauzetih resursa. To smo sučelje već implementirali u razmatranom primjeru. Novost je da se stvaranje i zatvaranje takvih objekata sada može prepustiti bloku `try`: evo primjera u nastavku.

```
86 private static void primjer4() {
87  try(LogFile log = new LogFile()) {
88   // radi nešto
89   log.writeMessage(" ... poruka ...");
90   // radi nešto
91   log.writeMessage(" ... poruka ...");
92   // radi nešto
93  } catch(Exception ex) {
94   System.out.println("Nastupila je pogreška!");
95  }
96 }
```

Nakon ključne riječi `try` može se otvoriti obla zagrada i u njoj deklarirati i inicijalizirati sve varijable (koristi se zarez za razdvajanje više deklaracija) koje je potrebno koristiti i potom automatski zatvoriti. Unutar oblih zagrada dozvoljeno je navođenje samo onih razreda koji implementiraju sučelje `AutoCloseable`. Zahvaljujući tome, iz navedenog se bloka može automatski generirati ekvivalentni blok koda koji će se pobrinuti da se svaki otvoreni resurs prije izlaska iz bloka na kraju sigurno i zatvori.

Pogledate li kod koji smo dobili, uporabom opisanog konstrukta napisali smo najmanju količinu koda uz koju i dalje imamo korektan rad metode.

Definiranje novih vrsta iznimaka

Pregled poglavlja o iznimkama ne bi bio gotov kada se još ne bismo osvrnuli i na mogućnost definiranja novih vrsta iznimaka. Iako osnovne biblioteke Jave sadrže čitav niz različitih vrsta iznimaka, prilikom pisanja programskih biblioteka javlja se potreba za definiranjem novih vrsta iznimaka. Razlog tome je jednostavan: iznimka svojim tipom donosi i određenu semantičku informaciju. Kada to ne bi bio slučaj, dovoljno bi bilo imati samo razred `java.lang.Throwable`. Stoga se prilikom razvoja programskih biblioteka uz oblikovanje potrebnih razreda i sučelja razmatra i prikladnost postojećih vrsta iznimaka te se po potrebi definiraju nove vrste iznimaka. Ovo ćemo pogledati na jednostavnom primjeru.

Zamislimo da razvijamo jednostavnu matematičku biblioteku unutar koje definiramo metodu `int prepolovi(int broj)` koja računa vrijednost predanog broja podijeljenog s dva. Metoda pri tome mora garantirati da je rezultat operacije korektan cijeli broj odnosno da se množenjem rezultata s 2 ponovno dobiva originalni broj. Prednost ove metode pred uobičajenim dijeljenjem je u brzini: metoda će rezultat izračunati bitno brže no što bi to bio slučaj kada bismo koristili operator dijeljenja: koristit ćemo operaciju aritmetičkog posmaka u desno. Da bi metoda mogla vratiti korektan rezultat, nužno je osigurati da predani broj bude paran. Stoga je u okviru razvoja ovakve biblioteke potrebno definirati novu vrstu iznimke: `NumberNotEvenException`. Kako bi se omogućilo definiranje metoda koje izazivaju ovu vrstu iznimaka a koje neće siliti programera da ih odmah obrađuje ili da svaku metodu deklarira na način metoda najavljuje bacanje takve iznimke, odlučeno je da se iznimka smjesti među neprovjeravane iznimke. Dodatno, s obzirom da će se nova iznimka izazivati samo kada metoda dobije neprihvatljiv argument (broj koji nije paran), iznimku ćemo izvesti iz razreda `IllegalArgumentException`. Kod razreda prikazan je u nastavku.

Primjer 6.5. Definiranje nove vrste iznimke

```
1 package hr.fer.zemris.java.primjeri;
2
3 public class NumberNotEvenException
```

```
 4        extends IllegalArgumentException {
 5
 6  private static final long serialVersionUID = 1L;
 7
 8  public NumberNotEvenException() {
 9  }
10
11  public NumberNotEvenException(String message) {
12   super(message);
13  }
14
15  public NumberNotEvenException(Throwable cause) {
16   super(cause);
17  }
18
19  public NumberNotEvenException(String message,
20         Throwable cause) {
21   super(message, cause);
22  }
23
24 }
25
```

Izvođenje nove vrste iznimke zapravo se svodi na izvođenje novog razreda iz nekog od prikladnih razreda (od razreda `Throwable`) na niže. Razred definira nekoliko konstruktora koji dalje pozivaju nadređene konstruktore razreda `IllegalArgumentException`.

Primjer biblioteke (odnosno jednog razreda) koja bi koristila ovu iznimku dana je u nastavku.

```
public class Biblioteka {
  public static int prepolovi(int broj) {
    if((broj & 1) == 1) {
      throw new NumberNotEvenException("Predan je broj "+broj+".");
    }
    return broj >> 1;
  }
}
```

Primjer uporabe razvijene metode dan je u nastavku.

```
public static void main(String[] args) {
 try {
  System.out.println(
   "Rez = " + Biblioteka.prepolovi(Integer.parseInt(args[0]))
  );
 } catch(NumberNotEvenException ex) {
  System.out.println(
   "Funkcija 'prepolovi' nije primjenjiva na predani broj."
  );
 }
}
```

Poglavlje 7. Apstraktni razredi. Sučelja.

U okviru ovog poglavlja nastavit ćemo upoznavanje s razredima i njima srodnim konceptima. Razmatranje ćemo započeti pojmom apstraktnog razreda nakon ćega ćemo se osvrnuti na sučelja.

Apstraktni razredi

U poglavlju 5 upoznali smo se s pojmom razreda. Primjer koji smo detaljno razradili uključivao je modeliranje različitih geometrijskih likova te omogućavanje njihovog crtanja na rasterskoj slici. Konačna inačica koda do koje smo došli izgledala je sasvim pristojno. Uspjeli smo doći do strukture rješenja kod koje je transparentno moguće dodavati nove geometrijske likove koji su se automatski mogli iscrtavati, a da pri tome nismo morali mijenjati kod koji se bavi crtanjem. Algoritam popunjavanja lika uspjeli smo napisati potpuno generički.

Prisjetimo se još jednom kako je izgledao algoritam za popunjavanje slike. U razredu `GeometrijskiLik` imali smo definiranu sljedeću metodu.

```
1 public void crtaj(Slika slika) {
2  for(int y = 0, h = slika.getVisina(); y < h; y++) {
3   for(int x = 0, w = slika.getSirina(); x < w; x++) {
4    if(this.sadrziTocku(x, y)) {
5     slika.upaliElement(x, y);
6    }
7   }
8  }
9 }
```

Algoritam se za ispravan rad oslanja na postojanje metode: `sadrziTocku(x,y)` koju smo također morali definirati u razredu `GeometrijskiLik` kako bi se napisani kod mogao prevesti. Istaknuli smo i problem s tim pristupom: kako u razredu `GeometrijskiLik` ponuditi implementaciju metode `sadrziTocku` kada u razredu `GeometrijskiLik` uopće ne znamo s kojim likom radimo te koji su njegovi parametri? Na kraju, odlučili smo se za implementaciju koja radi najmanje zlo: napisali smo implementaciju koja za svaku pitanu točku vraća vrijednost `false` čime garantira da se na slici neće ništa iscrtati. Uz takvu implementaciju definirali smo i *ugovor* koji mora vrijediti između svih razreda koji nasljeđuju razred `GeometrijskiLik`: tražili smo da *svaki razred koji naslijedi `GeometrijskiLik` mora nadjačati napisanu metodu `sadrziTocku`*. Nepridržavanje tog ugovora za posljedicu će imati neispravno iscrtavanje tih novih vrsta geometrijskih likova.

Spomenuti ugovor nadali smo se da će svaki razred koji naslijedi `GeometrijskiLik` ispoštivati. No može li bolje? Kada se prilikom programiranja moramo oslanjati na nadu da će se programer sjetiti napraviti sve što smo od njega tražili, to jasno upućuje na određene probleme do koji može doći. Što ako programer to zaboravi napraviti? Ne zato što je zločest i zao, već naprosto zato što implementira više razreda odjednom, i eto, dogodi se. Je li nužno da se ovakvi problemi otkrivaju tek kada se ustanovi da se napisani kod ne ponaša u skladu s očekivanjima koja provjeravamo (primjerice, napisanim testovima)? Svakako bi znatno bolje rješenje bilo kada bi već prevođenje koda puknulo jer je programer zaboravio ispoštivati definirani ugovor. Htjeli bismo mehanizam koji će nam omogućiti da osiguramo da svaki razred koji naslijedi razred `GeometrijskiLik` mora nadjačati metodu `sadrziTocku`.

Na sreću, u većini modernih objektno-orijentiranih programskih jezika moguće je pisati kod na način koji navedenu vrstu ugovora definira na način koji jezičnom prevodiocu potom omogućava da provjerava je li on ispoštovan. Taj isti mehanizam odmah će nas riješiti još jednog

problema: kada ne znamo kako implementirati neku primitivnu metodu (poput metode `sadrziTocke`), koju implementaciju ponuditi i zašto?

Rješenje oba problema dolazi u obliku apstraktnih metoda. Metode koje su potrebne kako bi se na nekom mjestu napisao općeniti algoritam ali koje na tom istom mjestu nismo u stanju implementirati treba proglasiti apstraktnima. Definiranjem apstraktne metode jezičnom se prevodiocu daju sve potrebne informacije kako bi provjerio koristi li ostatak koda tu metodu na ispravan način: deklaracije apstraktne metode zahtjeva da se uz ključnu riječ `abstract` ponudi samo prototip metode nakon čega dolazi znak točka-zareza, a ne vitičaste zagrade i implementacija metode. Konceptualno, apstraktne metode u ekvivalent čistih virtualnih metoda u programskom jeziku C++ (metoda koje su označene ključnom riječi `virtual` i koje za implementaciju imaju navedeno `= 0`).



Apstraktne metode

Apstraktne metode deklariraju se navođenjem ključne riječi `abstract` nakon koje dolazi ostatak prototipa metode pa znak točka-zarez. Apstraktne metode nemaju (i ne mogu imati) ponuđenu implementaciju u razredu u kojem su deklarirane -- zato ih i zovemo apstraktnima.

Deklaracijom apstraktne metode definirano je da razred sadrži takvu metodu. Dan je njezin prototip i zna se na koji se način takva metoda može pozivati. Međutim, kako razred za tu metodu ne nudi implementaciju, nužno je osigurati da programer ne može niti na koji način stvoriti primjerak tog razreda. Kada bi mogao, pitanje je koji bi se kod trebao izveo prilikom poziva te metode nad stvorenim objektom? Kako bi se osigurala jednostavnost provjere -- smije li se stvarati primjerke nekog razreda ili ne, Java specifikacijom propisuje da svaki razred koji sadrži barem jednu apstraktnu metodu (dakle, metodu koja nema ponuđene implementacije) mora biti i sam označen kao apstraktan. Da je razred apstraktan, deklarira se ključnom riječi `abstract` prije navođenja ključne riječi `class`. Specifikacija jezika također definira da je nemoguće stvarati primjerke apstraktnih razreda; takve pokušaje jezični prevodioc dužan je prijaviti kao pogreške.

Kako opisano možemo primijeniti na stablo geometrijskih likova? Evo modificiranog razreda `GeometrijskiLik`.

Primjer 7.1. Razred `GeometrijskiLik` u apstraktnom izdanju

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public abstract class GeometrijskiLik {
 4
 5  private String ime;
 6
 7  public GeometrijskiLik(String ime) {
 8   this.ime = ime;
 9  }
10
11  public String getIme() {
12   return ime;
13  }
14
15  public double getOpseg() {
16   return 0;
17  }
18
19  public double getPovrsina() {
20   return 0;
21  }
```

```
22
23  public abstract sadrziTocku(int x, int y);
24
25  public void crtaj(Slika slika) {
26   for(int y = 0, h = slika.getVisina(); y < h; y++) {
27    for(int x = 0, w = slika.getSirina(); x < w; x++) {
28     if(this.sadrziTocku(x, y)) {
29      slika.upaliElement(x, y);
30     }
31    }
32   }
33  }
34 }
35
```

Modificirana metoda `sadrziTocku` prikazana je u retku 23: tijelo metode je izbrisano i metoda je deklarirana kao apstraktna. Zbog toga je i sam razred (redak 3) morao biti deklariran kao apstraktan. Ovaj razred moguće je prevesti bez ikakvih pogrešaka. Međutim, sada više ne možemo napisati sljedeći kod.

```
GeometrijskiLik lik = new GeometrijskiLik("prvi");
```

Operator `new` više ne može stvarati primjerke razreda `GeometrijskiLik` jer takvim objektima nedostaje implementacija apstraktne metode `sadrziTocku`.

Primjerke apstraktnih razreda ne možemo stvarati ali apstraktne razrede, baš kao i one "normalne" možemo nasljeđivati. Razmotrimo razred `Krug` čija je implementacija dana u nastavku.

Primjer 7.2. Razred `Krug`

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Krug extends GeometrijskiLik {
 4  private int cx;
 5  private int cy;
 6  private int r;
 7
 8  public Krug(String ime, int cx, int cy, int r) {
 9   super(ime);
10   this.cx = cx;
11   this.cy = cy;
12   this.r = r;
13  }
14
15  public int getCx() {
16   return cx;
17  }
18
19  public int getCy() {
20   return cy;
21  }
22
23  public int getR() {
24   return r;
25  }
26
27  @Override
28  public double getOpseg() {
```

```
 29    return 2 * r * Math.PI;
 30   }
 31
 32   @Override
 33   public double getPovrsina() {
 34    return r * r * Math.PI;
 35   }
 36
 37   @Override
 38   public boolean sadrziTocku(int x, int y) {
 39    int dx = x-cx;
 40    int dy = y-cy;
 41    return dx*dx + dy*dy <= r*r;
 42   }
 43  }
 44
```

Implementacija razreda `Krug` prikazana u primjeru 7.2 jednaka je implementaciji koju smo ponudili u poglavlju 5. Razred `Krug` nije apstraktan razred -- nema razloga da bude. Razred `GeometrijskiLik` nismo smjeli stvarati jer njegovi primjerci ne bi imali definiranu metodu `sadrziTocku`. Razred `Krug` ovaj problem rješava: nudi implementaciju prethodno apstraktne metode i time ima definirane implementacije svih metoda. Stoga razred `Krug` nije apstraktan razred i operatorom `new` moguće je stvarati njegove primjerke.

Kako mehanizam apstraktnosti osigurava da razred `Krug` nadjačava metodu `sadrziTocku`? Stvar je zapravo vrlo jednostavna. Mi (programer) prilikom pisanja razreda `Krug` htjeli smo dobiti "normalni" (ne-apstraktni) razred. Stoga u deklaraciju razreda nismo pisali ključnu riječ `abstract`. Razlog je jasan: htjeli smo dopustiti da se u kodu stvaraju primjerci razreda `Krug`. Da smo kojim slučajem pokušali prevesti izvorni kod tog razreda a bez da smo u njemu nadjačali apstraktnu metodu `sadrziTocku`, jezični prevodioc dojavio bi poruku pogreške: razred `Krug` nije apstraktan a nema definirane sve metode, i postupak prevođenja tu bi puknuo. U slučaju da smo nadjačali zadanu metodu, postupak prevođenja bi uspio bez greške.

Razmotrimo sada i malo općenitiji slučaj: kada neki razred koji nasljeđuje apstraktni razred prestaje biti apstraktan? Evo primjera u nastavku.

```
abstract class C1 {
 abstract int m1();
 abstract int m2();
 abstract int m3();
}

class C2 extends C1 {
 int m1() { return 1; }
 int m2() { return 2; }
 int m3() { return 3; }
}

abstract class C3 extends C1 {
 int m1() { return 1; }
}

abstract class C4 extends C3 {
 int m3() { return 3; }
}

class C5 extends C4 {
 int m2() { return 2; }
 int m1() { return 4; }
```

```
}
```

Razred `C1` je vršni apstraktni razred koji definira tri apstraktne metode: `m1`, `m2` i `m3`. Razred `C2` nasljeđuje razred `C1` (a time i njegove tri apstraktne metode) i nudi implementaciju za svaku od njih. Kako u tom razredu više ne postoji metoda čija implementacija nije poznata, razred postaje "normalan" razred i moguće je stvarati njegove primjerke.

Razred `C3` je druga priča. Razred također nasljeđuje razred `C1` (i time dobiva njegove tri apstraktne metode). Razred potom nudi implementaciju metode `m1`. Time u razredu `C3` još uvijek ne postoje implementacije metoda `m2` i `m3` pa je razred `C3` zbog toga i sam apstraktan.

Razred `C4` nasljeđuje razred `C3`. Time preuzima njegovu implementaciju metode `m1` a sam još nudi implementaciju metode `m3`. Kako je time metoda `m2` ostala i dalje bez implementacije, razred `C4` je također apstraktan.

Konačno, razred `C5` nasljeđuje razred `C4`. Razred nudi implementaciju metode `m2` čime su sve prethodno apstraktne metode u ovom lancu nasljeđivanja definirane, pa razred `C5` više nije apstraktan i moguće je stvarati i njegove primjerke. Dodatno, razred `C5` nadjačava metodu `m1` kako bi dobio drugačije ponašanje. Primjer ilustrira da se apstraktne metode mogu nadjačavati na potpuno jednaki način kao i ne-apstraktne metode.



Apstraktni razredi

Razred koji sadrži barem jednu apstraktnu metodu (tj. metodu bez implementacije) mora biti označen kao apstraktan razred. Nije moguće stvarati primjerke apstraktnih razreda. Da bismo nasljeđivanjem koje kreće od apstraktnog razreda došli do razreda koji više nije apstraktan, nužno je da se do tog razreda ponude definicije svih apstraktnih metoda.

U kojim se situacijama koriste apstraktni razredi? Evo tipičnih scenarija.

- Kada je potrebno ponuditi osnovu za definiranje drugih razreda.
- Kako bi ponudili implementaciju svih dijeljenih algoritama temeljem apstraktnih primitiva.
- Kada želimo osigurati da neki od razreda u lancu nasljeđivanja mora ponuditi konkretne implementacije određenih metoda.

Nakon upoznavanja s apstraktnim razredima, pogledajmo još jedan sličan koncept: sučelja.

Sučelja

Postoji nekoliko načina na koje možemo opisati pojam sučelja. Mi ćemo se ovdje osvrnuti na dva pristupa koja pojmu prilaze s različitih strana.

- Sučelje je potpuno apstraktan razred.
- Sučelje je ugovor.

Jedan pogled na koncept sučelja jest poistovjetiti ga s potpuno apstraktnim razredom. Potpuno apstraktni razred je razred u kojemu su sve metode apstraktne. Konceptualno, sučelje je upravo to: popis metoda koje će razred koji ga "naslijedi" dobiti kao apstraktne metode. Želi li takav razred biti "normalan" (ne-apstraktan), morat će ponuditi implementacije za svaku od njih. U suprotnom, postat će apstraktan pa neće biti moguće stvarati njegove primjerke.

Drugi način na koji možemo gledati sučelja jest kao na ugovore. Svako sučelje predstavlja jedan ugovor: navodi popis metoda koje ga čine te daje kroz korisničku dokumentaciju sučelja i njegovih metoda semantičku informaciju o primjeni i namjeni tog sučelja te o očekivanom ponašanju implementacija metoda koje on propisuje odnosno razreda koji ga nasljeđuju.

Do sada smo već naučili da Java omogućava jednostruko nasljeđivanje. Višestruko nasljeđivanje, kao što je to slučaj kod programskog jezika C++ nije podržano. Jedan od razloga ovakve odluke jest izbjegavanje vrlo neugodnog problema koji višestruko nasljeđivanje donosi sa sobom. Pretpostavimo za tren da je sljedeći kod legalan u Javi.

```
class D1 {
 int m() { return 1; }
}
class D2 {
 int m() { return 2; }
}
class D3 extends D1, D2 {
 int n() { return m() + 1; }
}
```

U primjeru imamo tri razreda. Razredi `D1` i `D2` definiraju samo jednu metodu: `m()`. Razred `D3` nasljeđuje oba razreda i definira metodu `n()` koja se oslanja na metodu `m()`. Što će vratiti poziv metode `n()` odnosno koju od metoda `m` smo ovdje htjeli koristiti? Do sličnog problema može doći i s članskim varijablama: što ako oba razreda definiraju isto nazvane članske varijable koje su pri tome i jednakog tipa: treba li razred koji ih nasljeđuje imati po jednu kopiju takve varijable za svaki od razreda ili te definicije treba preklopiti u samo jedan primjerak varijable? A što ako su varijable u razredima deklarirane istog imena ali različitih tipova? Iz opisanog vidimo da se pri višestrukom nasljeđivanju mogu javiti vrlo nezgodni problemi i da jezik mora ponuditi odgovarajuće mehanizme za njihovo razriješavanje. Model nasljeđivanja za koji se odlučila Java ovakve probleme izbjegava.

Uporaba sučelja odnosno nasljeđivanje više od jednog sučelja ne može nas dovesti do opisanih problema. Stoga Java dozvoljava jednostruko nasljeđivanje razreda i višestruko nasljeđivanje sučelja. Kako stoga ne bi došlo do dodatnih zbrki, pojam nasljeđivanje više nećemo koristiti za sučelja: uvriježena terminologija u svijetu Jave jest da se razredi nasljeđuju a sučelja implementiraju.



Bilješka

Razredi se nasljeđuju a sučelja implementiraju. Java podržava jednostruko nasljeđivanje razreda te implementiranje proizvoljnog broja sučelja.

Već smo rekli da je sučelje popis metoda. Dajmo jedan primjer.

```
 1 interface I1 {
 2  public void m1();
 3  public int m2();
 4 }
 5
 6 interface I2 {
 7  void m1();
 8  String m3();
 9 }
10
11 class C1 implements I1 {
12  public void m1() {System.out.println("Hi!");}
13  public int m2() { return 1; }
14 }
15
16 class C2 extends C1 implements I2 {
17  String m3() { return "Sokrat"; }
18 }
19
20 class C3 implements I1, I2 {
```

```
21  public void m1() {System.out.println("Hello!");}
22  public int m2() { return 2; }
23  String m3() { return "Aristotel"; }
24 }
25
26 abstract class C4 implements I1, I2 {
27  String m3() { return "G.I. Joe"; }
28 }
```

U prikazanom primjeru najprije su definirana dva sučelja. Sučelja se definiraju slično kao i razredi. Razlika je u tome da se umjesto ključne riječi `class` koristi ključna riječ `interface`. Sučelje `I1` definira dvije javne metode. Kako je sučelje u konceptualnom smislu sinonim za potpuno apstraktan razred, sve deklarirane metode su po definiciji apstraktne i nije potrebno eksplicitno navoditi ključnu riječ `abstract` uz svaku metodu (ali nije niti pogrešno; naprosto nije uobičajeno). Sučelje `I2` definira također dvije metode. Kao i kod sučelja `I1` te su metode javne iako u deklaraciji metoda ne piše eksplicitno `public`. Po definiciji, sve metode sučelja su javne pa se pisanje ključne riječi `public` može preskočiti (i često se tako i radi).

Potom je dan primjer razreda `C1` koji implementira sučelje `I1`. Kako razred odmah i daje implementacije obiju apstraktnih metoda koje je dobio iz sučelja, razred nije apstraktan.

Razred `C2` nasljeđuje razred `C1` i implementira sučelje `I2`. Činjenicom da nasljeđuje `C1` razred `C2` također implementira sučelje `I1`. Niti `C2` nije apstraktan razred. Sučelje `I2` traži da postoje dvije metode: `void m1();` i `String m3();`. Da ne bi bio apstraktan, razred `C2` treba još samo implementirati metodu `m3`. Metoda `m1` već je implementirana od razreda `C1`. Uočite sada zašto višestruko nasljeđivanje sučelja ne stvara probleme: i sučelje `I1` i sučelje `I2` zahtjevaju da razred koji ih implementira ponudi definiciju metode `m1`, i to je za razred `C2` već ispunjeno činjenicom da je nasljedio razred `C1` koji sadrži implementaciju. Uočite da nije bitno koliko različitih sučelja kaže da razred treba imati metodu `m1`: jednom kada je on ponudi, svi njihovi zahtjevi se istovremeno zadovoljavaju.

Razred `C3` je razred koji implementira oba sučelja. Stoga ukupno mora ponuditi implementacije triju metoda i to je učinjeno. Stoga niti ovaj razred nije apstraktan.

Konačno, razred `C4` implementira oba sučelja, nudi implementacije samo dviju od ukupno tri zahtjevane metode i zbog tog razloga je apstraktan.

Proučite još jednom navedene primjere i uočite kako se deklarira nasljeđivanje a kako implementiranje. Da razred nasljeđuje neki drugi razred deklarira se ključnom riječi `extends` a da implementira jedno ili više sučelja ključnom riječi `implements` pri čemu se sučelja razdvajaju zarezima (ako ih ima više). Najopćeniti oblik tada je prikazan u nastavku.

```
class X extends Y implements I1, I2, ..., In {}
```

Pogledajmo sada zašto su sučelja potrebna. Sučelje klijentima nudi API kroz koji mogu komunicirati s objektom koji implementira to sučelje. To će biti lijepo vidljivo iz sljedećeg primjera.

```
1 public void klijent1(I1 objekt) {
2  objekt.m1();
3  System.out.println("Broj: "+ objekt.m2());
4 }
5
6 public void klijent2(I2 objekt) {
7  objekt.m1();
8  System.out.println("Ime je: "+ objekt.m3());
9 }
10
11 public static void main(String[] args) {
12  C2 c = new C2();
```

```
13  klijent1(c);
14  klijent2(c);
15 }
```

Definirane su dvije metode koje obavljaju različite zadatke. Metoda `klijent1` za posao koji radi očekuje da će dobiti referencu na objekt koji implementira sučelje `I1`. Svoj posao, ta metoda obavlja pozivajući nad dobivenim objektom upravo metode koje su definirane u tom sučelju. Slično, metoda `klijent2` očekuje da će dobiti referencu na objekt koji implementira sučelje `I2` i ona svoj posao obavlja oslanjajući se na metode tog sučelja. Metoda `main` stvara primjerak razreda `C2` i referencu na taj objekt predaje pri pozivu oba klijenta. Naime, objekt koji je primjerak razreda `C2` je primjerak razreda koji implementira i sučelje `I1` i sučelje `I2`. Stoga s takvim objektom oba klijenta mogu razgovarati kroz njima potrebno sučelje. U ovom kontekstu, zgodno je o sučelju razmišljati na još jedan način: sučelje definira jedan pogled na neki konkretan objekt. Ako je objekt primjerak razreda koji implementira neko zadano sučelje, to znači da on nudi implementacije svih metoda koje to sučelje propisuje. Dapače, on možda nudi i neke druge metode, ali to u ovom kontekstu nije bitno.



Bilješka

Uporaba sučelja omogućava izradu općenitih metoda: metoda koje ne znaju s primjerkom kojeg razreda komuniciraju i to ih nije briga. Programeru pak ovo omogućava da piše različite implementacija razreda koji implementiraju propisano sučelje te da primjerke bilo kojeg od tih razreda šalje takvim općenitim metodama. One će raditi dobro, neovisno o konkretnom razredu objekta koji su dobile.

Pogledajmo sada i što se sve smije navesti u definiciji sučelja? Legalni članovi sučelja su:

- konstante (sve se implicitno smatraju `public static final`),
- prototipovi metoda (sve se implicitno smatraju `public`) te
- deklaracije ugniježđenih tipova.

Spomenimo još i da se sučelja mogu nasljeđivati i to višestruko. Sljedeći kod je sasvim legalan.

```
interface I3 { ... }
interface I4 { ... }
interface I5 { ... }
interface I6 extends I3 { ... }
interface I7 extends I6, I4, I5 { ... }
```

Kako se oblikuje sučelje? Koliko metoda pripada u jedno sučelje? Na ova i slična pitanja pokušat ćemo odgovoriti u poglavlju 8 kroz razmatranje čitavog niza načela objektno-orijentiranog oblikovanja. Spomenimo za sada kao zanimljivost da u Javi postoje i sučelja koja nemaju niti jedne definirane metode. Takva sučelja služe isključivo kako bi se razredima koji ih implementiraju dodijelila određena semantika koja je definirana u dokumentaciji sučelja. Jedan od takvih primjera je sučelje `java.io.Serializable`. To sučelje je prazno sučelje. Međutim, zatražite li virtualni stroj da napravi serijalizaciju grafa objekata, virtualni stroj će na to pristati samo u slučaju da su svi objekti prisutni u tom grafu primjerci razreda koji implementiraju to sučelje. Činom implementiranja tog sučelja, razred poručuje da je u redu napraviti serijalizaciju primjerka tog razreda. Sličan primjer imamo sa sučeljem `java.util.RandomAccess`: razredi koji implementiraju ovo sučelje poručuju okolini da operacije pozicijskog pristupa njihovi objekti mogu raditi vrlo efikasno, odnosno u konstantnoj složenosti. U svijetu Jave, takva se sučelja nazivaju *markerska* sučelja.

Primjer uporabe sučelja

Kao ilustraciju uporabe sučelja, opet ćemo se vratiti na primjer geometrijskih likova i njihovog iscrtavanja na zaslonu. Rješenje koje smo prethodno prikazali definiralo je metodu `crtaj` kao

metodu razreda `GeometrijskiLik`. Pogledajmo sada drugačije rješenje. Najprije ćemo definirati sučelje: `SadrziocTocaka` kako je prikazano u nastavku.

```
1 package hr.fer.zemris.java.tecaj_1;
2
3 public interface SadrziocTocaka {
4  boolean sadrziTocku(int x, int y);
5 }
6
```

Radi se o jednostavnom sučelju koje garantira da će se objekt svakog razreda koji implementira ovo sučelje znati izjasniti o tome pripada li mu koja točka ili ne. Temeljeći se na ovoj informaciji, sada možemo ponuditi drugačiju implementaciju razreda `Slika`: implementaciju koja zna crtati bilo što što je po tipu `SadrziocTocaka`. Implementacija je prikazana u nastavku.

```
1 package hr.fer.zemris.java.tecaj_1;
2
3 public class Slika {
4
5  private int sirina;
6  private int visina;
7  private boolean[][] elementi;
8
9  public Slika(int sirina, int visina) {
10  this.sirina = sirina;
11  this.visina = visina;
12  elementi = new boolean[visina][sirina];
13 }
14
15 public int getSirina() {
16  return sirina;
17 }
18
19 public int getVisina() {
20  return visina;
21 }
22
23 public void upaliElement(int x, int y) {
24  elementi[y][x] = true;
25 }
26
27 public void ugasiElement(int x, int y) {
28  elementi[y][x] = false;
29 }
30
31 public String toString() {
32  StringBuilder sb = new StringBuilder((sirina+1)*visina);
33  for(int y = 0; y < visina; y++) {
34   for(int x = 0; x < sirina; x++) {
35    sb.append(elementi[y][x] ? '*' : '.' );
36   }
37   sb.append('\n');
38  }
39  return sb.toString();
40 }
41
42 public void crtaj(SadrziocTocaka lik) {
43  for(int y = 0; y < visina; y++) {
44   for(int x = 0; x < sirina; x++) {
```

```
45     if(lik.sadrziTocku(x, y)) {
46      upaliElement(x, y);
47     }
48    }
49   }
50  }
51 }
52
```

Konačno, sada možemo ponuditi jednu moguću implementaciju sadržioca točaka: razred `GeometrijskiLik` i njegovo stablo nasljeđivanja. Odgovarajući kod prikazan je u nastavku. Pokušajte pronaći razlike u odnosu na prethodnu implementaciju.

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public abstract class GeometrijskiLik
 4   implements SadrziocTocaka {
 5
 6  private String ime;
 7
 8  public GeometrijskiLik(String ime) {
 9   this.ime = ime;
10  }
11
12  public String getIme() {
13   return ime;
14  }
15
16  public double getOpseg() {
17   return 0;
18  }
19
20  public double getPovrsina() {
21   return 0;
22  }
23
24 }
25
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Pravokutnik extends GeometrijskiLik {
 4
 5  private int x;
 6  private int y;
 7  private int sirina;
 8  private int visina;
 9
10  public Pravokutnik(String ime, int x, int y,
11    int sirina, int visina) {
12   super(ime);
13   this.x = x;
14   this.y = y;
15   this.sirina = sirina;
16   this.visina = visina;
17  }
18
19  public int getX() {
20   return x;
```

```
21  }
22  public int getY() {
23   return y;
24  }
25  public int getSirina() {
26   return sirina;
27  }
28  public int getVisina() {
29   return visina;
30  }
31
32  @Override
33  public double getPovrsina() {
34   return sirina * visina;
35  }
36
37  @Override
38  public double getOpseg() {
39   return 2*(sirina + visina);
40  }
41
42  @Override
43  public boolean sadrziTocku(int x, int y) {
44   if(x < this.x || x >= this.x + sirina) return false;
45   if(y < this.y || y >= this.y + visina) return false;
46   return true;
47  }
48
49 }
50
```

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class Krug extends GeometrijskiLik {
 4  private int cx;
 5  private int cy;
 6  private int r;
 7
 8  public Krug(String ime, int cx, int cy, int r) {
 9   super(ime);
10   this.cx = cx;
11   this.cy = cy;
12   this.r = r;
13  }
14
15  public int getCx() {
16   return cx;
17  }
18
19  public int getCy() {
20   return cy;
21  }
22
23  public int getR() {
24   return r;
25  }
26
27  @Override
```

```
28  public double getOpseg() {
29   return 2 * r * Math.PI;
30  }
31
32  @Override
33  public double getPovrsina() {
34   return r * r * Math.PI;
35  }
36
37  @Override
38  public boolean sadrziTocku(int x, int y) {
39   int dx = x-cx;
40   int dy = y-cy;
41   return dx*dx + dy*dy <= r*r;
42  }
43
44 }
45
```

Definirane razrede sada koristimo u nešto drugačijem scenariju.

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 public class PrimjerSlike3 {
 4
 5  public static void main(String[] args) {
 6   Slika slika = new Slika(60, 20);
 7
 8   GeometrijskiLik[] likovi = new GeometrijskiLik[] {
 9    new Krug("k1", 4, 4, 3),
10    new Krug("k2", 45, 10, 9),
11    new Pravokutnik("p1", 12, 6, 20, 8)
12   };
13
14   for(GeometrijskiLik lik : likovi) {
15    slika.crtaj(lik);
16   }
17
18   System.out.print(slika.toString());
19  }
20
21 }
22
```

Uočavate li razliku? Više se ne crta lik na slici već slika crta lik. No, nije nužno da slika crta lik -- slika će crtati bilo što što je po tipu `SadrziocTocaka`. Ovo će nam pak omogućiti da imamo disjunktna stabla nasljeđivanja ali koja implementiraju zadano sučelje -- funkcija crtaj radit će jednako dobro s objektima koji su primjerci razreda iz bilo kojeg od tih stabala nasljeđivanja.



Apstraktni razredi vs. sučelja

Kada koristiti apstraktne razrede a kada sučelja? Sljedeće će Vam pomoći da odlučite.

Apstraktni razred je -- razred.

- Dobro rješenje za definirati osnovu za izvođenje novih razreda.
- Može ponuditi implementaciju zajedničkih algoritama.

Sučelje je -- popis.

- Dobro rješenje za dodavanje “karakteristika” postojećim razredima.
- Funkcionira neovisno o strukturi nasljeđivanja: razred koji već ima definiranu strukturu nasljeđivanja može implementirati proizvoljna sučelja.
- Izuzetno često korišteni u Javi.
- Susrest ćemo se s njima kod Collection Frameworka i Swinga.
- Omogućava razdvajanje implementacije i "obećane" funkcionalnosti.
- Može se shvatiti i kao "pogled" kroz koji se vidi neki objekt.

Modifikator `static`

Prilikom cjelokupne dosadašnje diskusije uspješno smo izbjegavali osvrnuti se modifikator `static`. Nekako smo ga implicitno pisali kada god smo govorili o "glavnom programu" u kojem su sve metode bile statičke. S druge strane, magično je nestao kada smo god govorili o "uobičajenim" razredima -- nema ga niti u razredu `GeometrijskiLik`, niti u razredu `Pravokutnik`; zapravo, nema ga nigdje gdje smo pričali o polimorfizmu, nadjačavanju i apstraktnim metodama. Vrijeme je da naučimo razlog.

Krenut ćemo s jednostavnim primjerom: u nastavku je dan izvorni kod razreda `ComplexNumber` te razreda `Primjer3` koji stvara tri primjerka kompleksnih brojeva.

Primjer 7.3. Razred `ComplexNumber` i primjer uporabe

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class ComplexNumber {
 4
 5  private double real;
 6  private double imaginary;
 7
 8  public ComplexNumber(double real, double imaginary) {
 9   super();
10   this.real = real;
11   this.imaginary = imaginary;
12  }
13
14  public double getReal() {
15   return real;
16  }
17
18  public double getImaginary() {
19   return imaginary;
20  }
21
22  public String toString() {
23   StringBuilder sb = new StringBuilder();
24   sb.append(real);
25   if(imaginary >= 0) {
26    sb.append(" + i").append(imaginary);
27   } else {
28    sb.append(" - i").append(-imaginary);
```

```
29    }
30    return sb.toString();
31   }
32  }
33
```

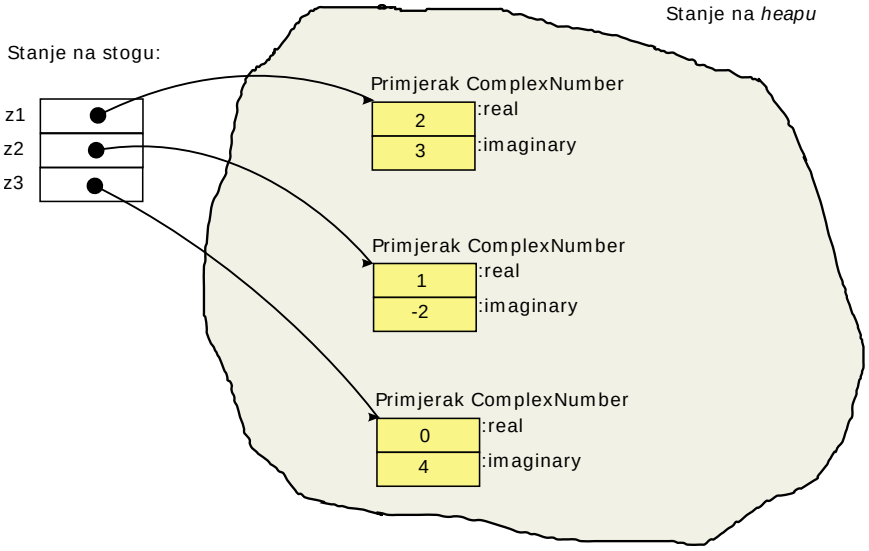
```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class Primjer3 {
 4
 5  public static void main(String[] args) {
 6   ComplexNumber z1 = new ComplexNumber(2, 3);
 7   ComplexNumber z2 = new ComplexNumber(1, -2);
 8   ComplexNumber z3 = new ComplexNumber(0, 4);
 9
10   System.out.println(z1);
11   System.out.println(z2);
12   System.out.println(z3);
13  }
14
15 }
16
```

Pokretanjem programa `Primjer3` dobit ćemo sljedeći ispis.

```
2.0 + i3.0
1.0 - i2.0
0.0 + i4.0
```

Proanalizirajmo malo detaljnije efekte izvođenja ovog programa. Kakvo je stanje u memoriji nakon izvođenja retka 9? Slika 7.1 daje odgovor na to pitanje.

Slika 7.1. Stanje u memoriji



Nakon izvođenja retka 8, na stogu imamo tri žive reference: `z1`, `z2` i `z3`. Svaka od njih pokazuje na jedan objekt u memoriji koji je primjerak razreda `ComplexNumber`. Već smo naučili da će svaki primjerak razreda u memoriji zauzeti dovoljno mjesta kako bi mogao pohraniti sve deklarirane članske varijable[1]: stoga je za svaki primjerak zauzeto dovoljno mjesta za pohranu varijable `real` i `imaginary`.

[1] pažnja, ovo nije u cijelosti točno što ćemo naučiti vrlo brzo -- pravimo se za sada da je to tako

Uporabom ključne riječi `static` prilikom deklariranja članske varijable, opisano se ponašanje mijenja. Pogledajmo najprije modifikaciju razreda `ComplexNumber` i novu inačicu demonstracijskog programa.

Primjer 7.4. Nova inačica razreda `ComplexNumber` i primjer uporabe

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class ComplexNumber {
 4
 5  private double real;
 6  private double imaginary;
 7  private static int counter = 0;
 8
 9  public ComplexNumber(double real, double imaginary) {
10   super();
11   this.real = real;
12   this.imaginary = imaginary;
13   counter++;
14  }
15
16  public double getReal() {
17   return real;
18  }
19
20  public double getImaginary() {
21   return imaginary;
22  }
23
24  public int getCounter() {
25   return counter;
26  }
27
28  public String toString() {
29   StringBuilder sb = new StringBuilder();
30   sb.append(real);
31   if(imaginary >= 0) {
32    sb.append(" + i").append(imaginary);
33   } else {
34    sb.append(" - i").append(-imaginary);
35   }
36   return sb.toString();
37  }
38 }
39
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class Primjer3 {
 4
 5  public static void main(String[] args) {
 6   ComplexNumber z1 = new ComplexNumber(2, 3);
 7   ComplexNumber z2 = new ComplexNumber(1, -2);
 8   ComplexNumber z3 = new ComplexNumber(0, 4);
 9
10   System.out.println(z1 + ", brojac = " + z1.getCounter());
11   System.out.println(z2 + ", brojac = " + z2.getCounter());
12   System.out.println(z3 + ", brojac = " + z3.getCounter());
```

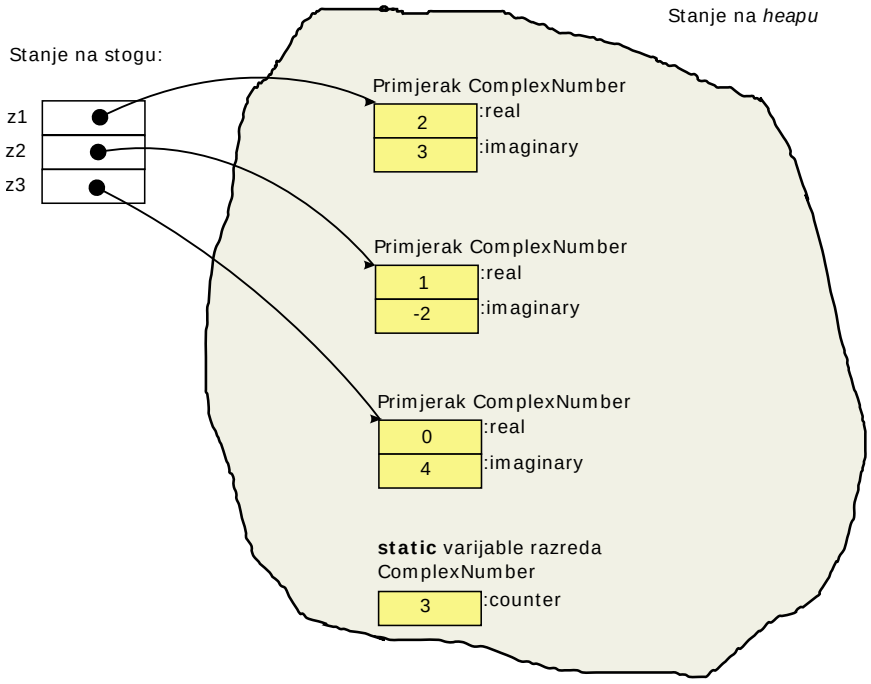
```
13   }
14
15 }
16
```

U čemu je razlika? U retku 7 dodali smo deklaraciju još jedne privatne članske varijable: `counter`. Ta je varijabla dodatno obilježena ključnom riječi `static`. Također, razredu je dodana još jedna metoda: `getCounter` koja dohvaća vrijednost varijable `counter` i vraća je pozivatelju. Pokretanjem ove inačice programa `Primjer3` dobit ćemo sljedeći ispis.

```
2.0 + i3.0, brojac = 3
1.0 - i2.0, brojac = 3
0.0 + i4.0, brojac = 3
```

Kako objasniti ovaj ispis? Stanje u memoriji nakon izvođenja retka 9 prikazano je na slici 7.2 i ono nam daje odgovor na postavljeno pitanje.

Slika 7.2. Stanje u memoriji



Kao što slika ilustrira, statičke članske varijable ne pripadaju pojedinim primjercima razreda. Statiče članske varijable pripadaju samom razredu i u memoriji postoje na samo jednom mjestu, neovisno o broju primjeraka tog razreda koji su stvoreni u memoriji.



Važno

Nestatičke članske varijable pripadaju primjercima razreda -- svaki primjerak razreda ima svoju vlastitu kopiju takvih varijabli.

Statičke članske varijable pripadaju razredu -- razred u memoriji čuva vrijednosti takvih varijabli; svi primjerci razreda stoga vide isti sadržaj pa možemo smatrati da se statičke varijable dijele između svih primjeraka razreda. Ažuriranjem vrijednosti statičke varijable nova vrijednost bit će automatski vidljiva svim primjercima tog razreda.

U opisanom primjeru, konstruktor razreda `ComplexNumber` uz inicijalizaciju vlastitih članskih varijabli inkrementira i vrijednost statičke varijable `counter`. Kako je statička varijabla `counter` inicijalizirana na vrijednost `0`, nakon konstrukcije prvog objekta njezina će vrijednost biti

postavljena na 1. Stvaranje sljedećeg primjerka razreda ComplexNumber sadržaj varijable counter još će se jednom uvećati, i konačno, nakon stvaranja i trećeg primjerka, još će se jednom uvećati. Posljedica je da će njezina vrijednost biti postavljena na 3 i tu će vrijednost vidjeti svi objekti što potvrđuje i naš ispitni program.

Spomenimo i da se za pristup statičkim članskim varijablama ne koristi referenca this: te varijable ne pripadaju primjerku objekta na koji pokazuje this. Tako primjerice, pokušate li modificirati program na način da metodu getCounter promjenite tako da bude sljedeća:

```
public int getCounter() {
 return this.counter;
}
```

i ako to radite u razvojnom okruženju Eclipse, odmah ćete dobiti poruku: *"The static field ComplexNumber.counter should be accessed in a static way"* čime vas sustav obavještava da pristup statičkoj varijabli prikazujete kao pristup nestatičkoj članskoj varijabli. Drugim riječima, zbunjujete onoga tko čita vaš kod a jezični prevodioc će zanemariti prefiks this. i generirati kod koji će pročitati vrijednost s memorijske lokacije gdje se varijabla doista nalazi a ne je tražiti među varijablama predanog primjerka razreda. Prvo rješenje koje će vam i sam Eclipse ponuditi dovoljno govori o tome: *"Change access to static using 'ComplexNumber' (declaring type)"*. Odaberete li to rješenje, sustav će kod preurediti na sljedeći način.

```
public int getCounter() {
 return ComplexNumber.counter;
}
```

Da je varijabla counter kojim slučajem bila deklarirana kao javna, iz bilo kojeg drugog dijela koda mogli bismo joj pristupiti upravo navođenjem opisane sintakse: ime razreda, točka, naziv statičke varijable. Ponovimo još jednom, razlog tome leži u činjenici da statička varijabla counter ne pripada pojedinim primjercima razreda već samom razredu. Da bismo mogli pristupiti statičkim varijablama, u memoriji ne treba postojati niti jedan primjerak tog razreda.

Sada možemo poopćiti priču oko modifikatora static. Modifikator static moguće je primijeniti na bilo koji element koji je legalno deklarirati unutar razreda. Ako se radi o članskim varijablama, posljedica je da će takve članske varijable biti u memoriji zauzete na samo jednom mjestu. Što se događa ako se radi o metodama? Deklariranjem metoda kao statičkih, poruka je ista: dotična metoda ne pripada primjercima razreda već samom razredu. Posljedica je da statičke metode prilikom poziva neće dobivati skrivenu referencu na objekt preko kojeg su pozvane; u statičkim metodama referenca this je nedostupna. Dapače, pozivanje statičkih metoda preko objekta jednako je zbunjujuće kao i pristupanje statičkim članskim varijablama navođenjem reference this: u oba slučaja, bolje je naprosto pisati naziv razreda, točku i naziv metode koja se poziva. Primjerice, u razred ComplexNumber mogli bismo dodati sljedeću statičku metodu.

```
public static int current() {
  return counter;
}
```

Metoda dohvaća trenutnu vrijednost statičke varijable counter i iz bilo kojeg drugog dijela koda mogli bismo je pozvati navođenjem ComplexNumber.current(). Primjerice, sljedeći kod napisan u metodi main razreda Primjer3 bio bi legalan.

```
int brojPrimjeraka = ComplexNumber.current();
```

Važno je razumjeti na koji način modifikator static mijenja rad s metodama koje su njime označene. Već smo rekli da jezični prevodioc prilikom pozivanja takvih metoda neće generirati kod koji bi takvim metodama kao prvi skriveni parametar predao referencu na trenutni objekt. To je dapače razumljivo, s obzirom da je metodu moguće pozvati na način poput ComplexNumber.current() pa tu niti ne postoji referenca na objekt koja bi mogla biti poslana. Ovo pak ima sljedeću važnu posljedicu: statičke metode mogu pristupati samo i isključi-

vo statičkim članskim varijablama te mogu pozivati samo i isključivo druge statičke metode. Nestatičke metode nemaju ovakvih ograničenja: s obzirom da nestatičke metode pripadaju primjercima razreda i raspolažu referencom `this`, one mogu pozivati druge nestatičke metode (i implicitno proslijediti tu referencu) te druge statičke metode (referencu `this` naprosto neće dalje proslijediti).



Važno

Nestatičke metode pripadaju primjercima razreda. Iako je njihov kod smješten na samo jednom mjestu u memoriji pa nema dupliciranja koda, nestatičke metode implicitno dobivaju referencu na primjerak razreda ("trenutni objekt") nad kojim su pozvane pa imaju pristup njegovim članskim varijablama i mogu pozivati njegove druge nestatičke i statičke metode.

Statičke metode mogu pristupati samo statičkim članskim varijablama i pozivati statičke metode. U statičkim metodama ne postoji referenca `this` pa time niti pojam "trenutnog objekta".

Vrste razreda

U ovom podpoglavlju upoznat ćemo se s još jednom mogućnošću programskog jezika Java -- Java nam dopušta gniježđenje razreda. Evo nadopunjenog primjera razreda `ComplexNumber`.

Primjer 7.5. Nadopunjena inačica razreda `ComplexNumber` i primjer uporabe

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class ComplexNumber {
 4
 5  private double real;
 6  private double imaginary;
 7  private static int counter = 0;
 8
 9  public ComplexNumber(double real, double imaginary) {
10   super();
11   this.real = real;
12   this.imaginary = imaginary;
13   counter++;
14  }
15
16  public double getReal() {
17   return real;
18  }
19
20  public double getImaginary() {
21   return imaginary;
22  }
23
24  public int getCounter() {
25   return ComplexNumber.counter;
26  }
27
28  public String toString() {
29   StringBuilder sb = new StringBuilder();
30   sb.append(real);
```

```
31    if(imaginary >= 0) {
32     sb.append(" + i").append(imaginary);
33    } else {
34     sb.append(" - i").append(-imaginary);
35    }
36    return sb.toString();
37   }
38
39   public ComplexNumberInfo getInfo() {
40    return new ComplexNumberInfo();
41   }
42
43   public class ComplexNumberInfo {
44    private int counterSnapshot;
45
46    public ComplexNumberInfo() {
47     this.counterSnapshot = counter;
48    }
49
50    public int getCounterSnapshot() {
51     return counterSnapshot;
52    }
53
54    public double calculateModule() {
55     return Math.sqrt(
56       Math.pow(ComplexNumber.this.real,2) +
57       Math.powComplexNumber.this.imaginary,2)
58     );
59    }
60
61    public double getAngle() {
62     return Math.atan2(imaginary, real);
63    }
64   }
65 }
66
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import hr.fer.zemris.java.primjeri.ComplexNumber.
 4          ComplexNumberInfo;
 5
 6 public class Primjer3 {
 7
 8  public static void main(String[] args) {
 9   ComplexNumber z1 = new ComplexNumber(2, 3);
10   ComplexNumberInfo z1info = z1.getInfo();
11   ComplexNumber z2 = new ComplexNumber(1, -2);
12   ComplexNumberInfo z2info = z2.getInfo();
13   ComplexNumber z3 = new ComplexNumber(0, 4);
14   ComplexNumberInfo z3info = z3.new ComplexNumberInfo();
15
16   System.out.println(z1 + ", brojac = " + z1.getCounter());
17   System.out.println(
18     " ==> Brojač je: " + z1info.getCounterSnapshot());
19   System.out.println(z2 + ", brojac = " + z2.getCounter());
20   System.out.println(
21     " ==> Brojač je: " + z2info.getCounterSnapshot());
```

```
22    System.out.println(z3 + ", brojac = " + z3.getCounter());
23    System.out.println(
24      " ==> Kut je: " + z3info.getAngle()/Math.PI*180);
25   }
26
27 }
28
```

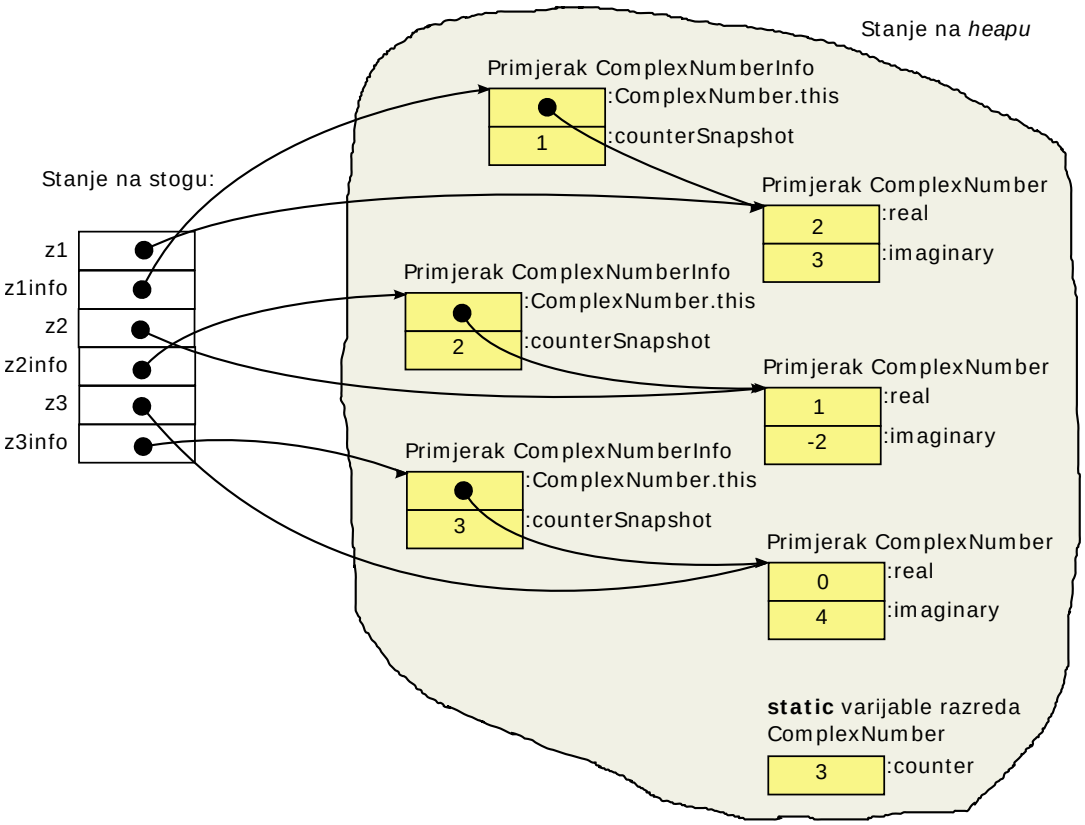
Pogledajte najprije kod razreda `ComplexNumber`. U retcima 43 do 61 dana je definicija novog razreda: razreda `ComplexNumberInfo`. Napišemo li unutar definicije jednog razreda definiciju nekog drugog razreda i to bez uporabe ključne riječi `static`, kažemo da smo napisali definiciju *unutarnjeg razreda*; engleski termin je *inner class*. Unutarnji razredi, s obzirom da nisu statički, po definiciji pripadaju primjerku vanjskog razreda. To znači da prilikom stvaranja primjerka unutarnjeg razreda taj primjerak pamti referencu na primjerak vanjskog razreda koji ga je stvorio. Zgodna posljedica je da primjerak unutarnjeg razreda ima pristup do članskih varijabli primjerka vanjskog razreda. Razred `ComplexNumberInfo` deklarira vlastitu nestatičku člansku varijablu `counterSnapshot`. Prilikom konstrukcije tog objekta, u tu se varijablu zapisuje trenutno stanje statičke varijable `counter`. Razred `ComplexNumber` u retcima 39 do 41 nudi implementaciju metode `getInfo` koja stvara primjerak razreda `ComplexNumberInfo` i vraća referencu na taj primjerak. Razred `ComplexNumberInfo` deklarira još dvije metode: jednu koja računa modul kompleksnog broja koji je stvorio taj objekt te drugu koja za isti objekt računa kut u kompleksnoj ravnini koji broj čini u odnosu na pozitivnu realnu os.

Program `Primjer3` slijedno tri puta stvara primjerak kompleksnog broja i objekt s informacijama za taj kompleksni broj. Rezultat izvođenja programa prikazan je u nastavku.

```
2.0 + i3.0, brojac = 3
 ==> Brojač je: 1
1.0 - i2.0, brojac = 3
 ==> Brojač je: 2
0.0 + i4.0, brojac = 3
 ==> Kut je: 90.0
```

Prije no što nastavimo daljnju diskusiju, pogledajmo kako će biti stanje u memoriji nakon izvođenja retka 14 prikazanog primjera.

Slika 7.3. Stanje u memoriji



Očekivano, u memoriji se nalazi 6 objekata: tri su primjerka razreda `ComplexNumber` i tri su primjerka razreda `ComplexNumberInfo`. Svaki od primjeraka razreda `ComplexNumberInfo` čuva dva podatka: referencu na primjerak razreda `ComplexNumber` koji ga je stvorio te varijablu `counterSnapshot`.

Pogledajmo sada implementaciju metode `calculateModule` čija je implementacija dana od retka 54. Kako metoda treba izračunati modul kompleksnog broja koji stvara primjerak tog razreda, metoda do članskih varijabli primjerka vanjskog razreda pristupa uporabom pune sintakse: naziv razreda, točka, ključna riječ `this`. Ovo je legalno, s obzirom da primjerak razreda `ComplexNumberInfo` ima referencu na primjerak vanjskog razreda. Međutim, uporaba takve sintakse nije nužna, što ilustrira metoda `getAngle` dana od retka 61. Ako nema kolizije u imenima između unutarnjeg razreda i vanjskog razreda, tada je dovoljno pisati samo naziv varijable: jezični prevodioc će automatski pronaći u kojem je kontekstu varijabla definirana i pristupiti joj na odgovarajući način. Stoga u metodi `getAngle` realnom dijelu kompleksnog broja pristupamo direktno navođenjem imena `real`; jezični prevodioc će ovo automatski pretvoriti u `ComplexNumber.this.real`.

Gdje se i kako mogu stvarati primjerci unutarnjih razreda? Da bismo mogli stvoriti primjerak unutarnjeg razreda, njužno je da postoji referenca na trenutni objekt kako bi je konstruktor unutarnjeg razreda mogao preuzeti. To znači da se primjerak razreda jednostavnim pozivom operatora `new` može stvarati samo iz nestatičkih metoda vanjskog objekta. Već u statičkim metodama vanjskog objekta to nije moguće. U statičkom kontekstu, operatoru `new` treba ponuditi kontekst u okviru kojega stvara primjerak unutarnjeg razreda. Ovo jasno ilustrira metoda `main` razreda `Primjer3`. Pogledajte redak 14.

```
ComplexNumberInfo z3info = z3.new ComplexNumberInfo();
```

Umjesto jednostavnog poziva operatora `new` korišten je oblik `z3.new` čime je operatoru definiran primjerak vanjskog razreda nad kojim radi. Ovaj način pozivanja iz metode `main` je moguć jer je konstruktor koji pokušavamo pozvati bio deklariran kao javan. Da smo konstruktor dek-

larirali kao privatni, tada ga ne bismo mogli pozivati iz drugih razreda; međutim, i dalje bismo mogli iz vanjskog razreda.

Definiramo li ugniježđeni razred uz uporabu ključne riječi `static`, taj bi razred pripadao vanjskom razredu a ne primjerku vanjskog razreda. Posljedica je da se prilikom stvaranja njegova primjerka ne bi automatski hvatala i pamtila referenca na primjerak vanjskog razreda. Ove razrede nazivat ćemo *statički ugniježđeni razredi*. Engleski termin koji se koristi za ovu vrstu razreda je *static nested classes*.



Važno

Statički ugniježđeni razredi su razredi definirani unutar drugih razreda i koji su označeni kao `static`. Statički ugniježđeni razredi imaju pristup samo do statičkih članskih varijabli vanjskog razreda te mogu direktno pozivati samo njegove statičke metode. Ovi razredi najčešće se koriste kao pomoćne strukture podataka.

Unutarnji razredi su ugniježđeni razredi koji nisu označeni kao `static`. Ti razredi pripadaju primjerku vanjskog razreda, prilikom stvaranja automatski pamte referencu na primjerak vanjskog razreda i mogu pristupati svim statičkim i nestatičkim članskim varijablama i metodama vanjskog razreda.

Konačno, pogledajmo još i sljedeća dva pojma: *lokalni razredi* te `anonimni razredi`. Pojam lokalnog razreda označava razrede koji su definirani u tijelu neke metode. Pogledajmo primjer. Napisan je razred koji omogućava pohranu rezultata natjecanja muške štafete 4 puta 100 metara. Konstruktor prima dva polja: prvo polje sadržava nazive timova dok drugo polje sadrži otrčana vremena u sekundama. Razred nudi metodu `ispisiSortirano` koja će rezultate ispisati sortirano od najbržeg tima do najsporijeg tima, pri čemu naziv tima može biti ispisan kako je dobiven ili pretvoren u velika slova, što ovisi o argumentu `velikimSlovima` koji funkcija dobiva prilikom poziva.

Primjer 7.6. Primjer definicije i uporabe lokalnog razreda

```
1  package hr.fer.zemris.java.primjeri;
2
3  import java.util.Arrays;
4
5  public class RezultatiNatjecanja {
6
7   private String[] timovi;
8   private double[] vremena;
9
10  public RezultatiNatjecanja(String[] timovi, double[] vremena) {
11   super();
12   this.timovi = Arrays.copyOf(timovi, timovi.length);
13   this.vremena = Arrays.copyOf(vremena, vremena.length);
14  }
15
16  public void ispisiSortirano(final boolean velikimSlovima) {
17   class Par {
18    String ime;
19    double vrijeme;
20
21    public Par(String ime, double vrijeme) {
22     super();
23     this.ime = velikimSlovima ? ime.toUpperCase() : ime;
24     this.vrijeme = vrijeme;
25    }
26
```

```
27     @Override
28     public String toString() {
29      return ime + ": " + vrijeme + "s";
30     }
31    }
32
33    // Stvori polje parova
34    Par[] parovi = new Par[timovi.length];
35    for(int i = 0; i < parovi.length; i++) {
36     parovi[i] = new Par(timovi[i], vremena[i]);
37    }
38
39    // Sortiraj polje parova
40    for(int i = 0; i < parovi.length; i++) {
41     for(int j = i+1; j < parovi.length; j++) {
42      if(parovi[i].vrijeme>parovi[j].vrijeme) {
43       Par tmp = parovi[i];
44       parovi[i] = parovi[j];
45       parovi[j] = tmp;
46      }
47     }
48    }
49
50    // Ispisi polje parova
51    for(Par p : parovi) {
52     System.out.println(p);
53    }
54   }
55
56 }
57
```

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 public class Primjer4 {
 4
 5  public static void main(String[] args) {
 6   String[] timovi = {
 7     "Tim 1", "Tim 2", "Tim 3", "Tim 4"
 8   };
 9   double[] vremena = {38.2, 37.1, 39.1, 38.5};
10
11   RezultatiNatjecanja rez = new RezultatiNatjecanja(
12    timovi, vremena
13   );
14   rez.ispisiSortirano(true);
15  }
16 }
17
```

Metoda `main` u razredu `Primjer4` stvara primjerak razreda `RezultatiNatjecanja` i predaje sve potrebne podatke kako bi konstruktor obavio svoj posao. Potom nad stvorenim objektom poziva metodu `ispisiSortirano`. Rezultat izvođenja prikazan je u nastavku.

```
TIM 2: 37.1s
TIM 1: 38.2s
TIM 4: 38.5s
TIM 3: 39.1s
```

Pogledajmo sada malo detaljnije metodu `ispisiSortirano`. Kako su izvorni podatci predani kao dva polja, prilikom sortiranja polja s vremenima trebalo bi istovremeno premještati i elemente polja s imenima kako bi se osiguralo da nakon postupka sortiranja i dalje na *i*-tom mjestu u oba polja budu ispravan naziv i vrijeme. Problem je riješen tako da je unutar metode `ispisiSortirano` definiran lokalni razred koji predstavlja uređeni par (*ime, vrijeme*). Potom su podatci iz oba polja prebačeni u polje takvih parova i zatim je to polje sortirano i ispisano.

Kako je vidljivo iz navedenog primjera, lokalni razredi su razredi koji su deklarirani unutar metode. Oni su vidljivi samo u toj metodi i mogu se koristiti od mjesta na kojem su deklarirani pa na niže (u tijelu te metode). Lokalni razredi koji su definirani u nestatičkoj funkciji ponašaju se kao unutarnji razredi -- oni također automatski hvataju referencu `this` na objekt bad kojim je metoda pozvana te zahvaljujući tome imaju pristup do svih članskih varijabli razreda kojemu metoda pripada. Lokalni razredi definirani u statičkim metodama ponašaju se kao statički ugniježđeni razredi -- oni imaju pristup samo do statičkih varijabli razreda unutar čije su statičke metode deklarirani. Spomenimo još i da lokalni razredi mogu uhvatiti i vrijednosti argumenata i lokalnih varijabli definiranih unutar metode u kojoj su definirani. U Javi 7 postoji ograničenje da su lokalnom razredu vidljive samo one varijable i argumenti koji su pretvoreni u konstante uporabom ključne riječi `final`. To je razlog zbog kojeg je argument metode `ispisiSortirano` deklariran kao `final`. U Javi 8, unutar lokalnih razreda trebali bi biti vidljivi svi argumenti i lokalne varijable koje su ili eksplicitno označene ključnom riječi `final` ili koje su *efektivno-konačne* (u smislu da ih se od promatranog mjesta na dalje više ne mijenja).

Iako zgodna mogućnost, uporaba lokalnih razreda se ne preporuča. Pisanje metoda koje deklariraju lokalne razrede koji opet sadrže metode, članske varijable i slično uvelike otežava čitljivost napisanog koda.



Važno

Lokalni razredi su razredi koji su definirani u tijelu metoda. Ako je metoda nestatička, lokalni razred ponaša se slično kao unutarnji razred. Ako je metoda statička, lokalni razred ponaša se slično kao statički ugniježđeni razred.

Konačno, ostalo nam je još razmotriti često korištenu mogućnost jezika Java -- anonimne razrede. Kao što im ime kaže, anonimni razredi su razredi koji u kodu nigdje nisu eksplicitno definirani navođenjem ključne riječi `class` iza koje bi došlo ime razreda. Sljedeći primjer pojasnit će nam o čemu se radi.

Primjer 7.7. Primjer definicije i uporabe anonimnih razreda

```
1 package hr.fer.zemris.java.primjeri;
2
3 public interface IIspisivo {
4
5  String poruka();
6
7 }
8
```

```
1 package hr.fer.zemris.java.primjeri;
2
3 public class Ispisivac {
4
5  public static void ispisi(IIspisivo ispisivo) {
6   System.out.println(ispisivo.poruka());
7  }
8
9 }
10
```

```java
package hr.fer.zemris.java.primjeri;

import java.text.SimpleDateFormat;
import java.util.Date;

public class DatumIVrijeme implements IIspisivo {

 private String format;

 public DatumIVrijeme(String format) {
  super();
  this.format = format;
 }

 @Override
 public String poruka() {
  return new SimpleDateFormat(format).format(new Date());
 }
}

```

```java
package hr.fer.zemris.java.primjeri;

public class Primjer5 {

 public static void main(String[] args) {

  Ispisivac.ispisi(new Poruka("Pozdrav!"));

  Ispisivac.ispisi(new IIspisivo() {
   @Override
   public String poruka() {
    return "Broj dostupnih procesora: " +
     Runtime.getRuntime().availableProcessors();
   }
  });

  Ispisivac.ispisi(new DatumIVrijeme("yyyy-MM-dd HH:mm:ss"));

  Ispisivac.ispisi(new DatumIVrijeme("yyyy-MM-dd HH:mm:ss") {
   @Override
   public String poruka() {
    return super.poruka() + ", dretva: " +
     Thread.currentThread().getName();
   }
  });
 }

 private static class Poruka implements IIspisivo {
  private String poruka;

  public Poruka(String poruka) {
   super();
   this.poruka = poruka;
  }

  @Override
  public String poruka() {
```

```
38      return poruka;
39    }
40   }
41 }
42
```

Za potrebe primjera definirali smo sučelje `IIspisivo` koje objektima daje svojstvo ispisivosti: bilo koji razred može implementirati ovo sučelje i potom ponuditi metodu `String poruka()` koja vraća poruku koju je potrebno ispisati. Definiran je i razred `Ispisivac` koji ima jednu statičku metodu: `void ispisi(IIspisivo ispisivo)`. Metoda prima referencu na bilo koji objekt koji je ispisiv, poziva njegovu metodu `poruka()` i tu poruku šalje na standardni izlaz. Dodatno, napisana je i jedna implementacija ovog sučelja: `DatumIVrijeme`. primjerci tog razreda u metodi `poruka` generiraju string koji predstavlja datum i vrijeme koje je bilo u trenutku poziva te metode a u skladu s formatiranjem koje je zadano preko konstruktora razreda.

Svi prethodno opisani razredi ne donose ništa novoga. Anonimne razrede pronaći ćemo u metodi `main` razreda `Primjer5`. Ova metoda sadrži nekoliko poziva metode `Ispisivac.ispisi(...)` čiji argument mora biti referenca na objekt koji je primjerak razreda koji implementira sučelje `IIspisivo`. U retcima 28 do 40, u razredu `Primjer5` definiran je statički ugniježđeni razred `Poruka`. Primjerci tog razreda dobivaju poruku preko konstruktora i nju prosljeđuju pozivatelju iz metode `poruka()`. U metodi `main` u retku 7 nalazi se prvi poziv metode `Ispisivac.ispisi`. Kao argument toj metodi predajemo referencu na primjerak razreda `Poruka` koji direktno stvaramo primjenom operatora `new`.

U retku 9 imamo sličnu situaciju. Pogledate li malo bolje, razlika u odnosu na redak je 7 je što sada imamo nakon poziva konstruktora još i blok dan vitičastim zagradama.

```
Ispisivac.ispisi(new IIspisivo() {...});
```

Na prvi pogled, čini se se poziva konstruktor od `IIspisivo`. Mi, naravno, znamo da je to nemoguće: `IIspisivo` nije razred već sučelje i nije moguće stvarati njegove primjerke. Opisani kod to ne radi. Konstrukt `new IIspisivo() {...}` kod kojeg je `IIspisivo` sučelje na tom mjestu definira jedan anonimni razred koji implementira zadano sučelje, operatorom `new` stvara jedan primjerak tog razreda i potom zove njegov konstruktor. U ovom slučaju, poziva se pretpostavljeni (*defaultni*) konstruktor. Blok s vitičastim zagradama koje slijede nužan je i on upravo sadrži definiciju metoda ovog anonimnog razreda. S obzirom da je `IIspisivo` sučelje, razred koji ga implementira po definiciji je apstraktan ako ne definira sve metode tog sučelja a tada ne bismo mogli stvarati primjerke takvog razreda. Stoga je u bloku koji je dan u nastavku ponuđena implementacija metode `poruka()` kako anonimni razred ne bi bio apstraktan.

U određenom smislu, anonimni razredi najsličniji su lokalnim razredima: ponašaju se jednako, osim što im nigdje nije dano ime i mogu se pojaviti isključivo u kontekstu gdje se odmah operatorom `new` stvara i jedan primjerak tako definiranog razreda. Važno je uočiti da se anonimnom razredu ne može eksplicitno definirati konstruktor: za to bismo morali znati njegovo ime jer sintaksa jezika Java zahtjeva da se konstruktor definira kao metoda koja se zove jednako kako se zove i razred.

U retku 17 nema ništa novoga. Razred `DatumIVrijeme` definirali smo u zasebnoj datoteci kao razred koji implementira sučelje `IIspisivo`. Stoga ovdje samo stvaramo jedan primjerak tog razreda i referencu predajemo kao argument metodi `Ispisivac.ispisi(...)`.

Redak 19 ponovno ilustrira uporabu anonimnog razreda. Na prvi pogled čini se da zovemo konstruktor razreda `DatumIVrijeme`, no to nije točno -- uočite nakon njega blok definiran vitičastim zagradama. Ovdje definiramo novi anonimni razred koji nasljeđuje razred `DatumIVrijeme`; pozivamo konstruktor tog novog anonimnog razreda koji će biti implicitno generiran od jezičnog prevodioca na način da pozove konstruktor razreda `DatumIVrijeme` sa predanim argumentima. Definicija tog razreda dana je u nastavku u vitičastim zagradama, i kao što je vidljivo, nadjačavamo metodu `poruka` tako da pozovemo nadjačanu verziju (onu iz razreda `DatumIVrijeme`) i njoj još nadodamo naziv dretve koja izvodi trenutnu metodu.

Rezultat pokretanja programa prikazan je u nastavku.

```
Pozdrav!
Broj dostupnih procesora: 2
2013-04-23 15:41:23
2013-04-23 15:41:23, dretva: main
```

Nakon prethodnih pojašnjenja, ne biste smjeli imati problema s razumijevanjem ovog ispisa.



Važno

Anonimni razredi slični su lokalnim razredima i definiraju se isključivo u paru s operacijom stvaranja primjerka takvog razreda konstruktom oblika `new RazredIliSucelje(argumenti) { definicija metoda }`. Pri tome, ako je `RazredIliSucelje` naziv sučelja, definira se anonimni razred koji implementira to sučelje a ako je `RazredIliSucelje` naziv razreda, definira se anonimni razred koji nasljeđuje taj razred. U tijelu razreda moraju biti ponuđene implementacije svih metoda koje su do tog trenutka apstraktne kako bi se operatorom `new` mogao odmah stvoriti i primjerak takvog razreda. Ovi razredi ne mogu imati eksplicitno definiran konstruktor niti mogu istovremeno nasljediti neki razred i implementirati neko sučelje ili implementirati više od jednog sučelja.

Anonimni razredi ponašaju se na jednak način kao i lokalni razredi i u pogledu hvatanja argumenata i lokalnih varijabli: ovisno o verziji Jave, u metodama anonimnog razreda dostupni su svi argumenti i lokalne varijable metode koji su eksplicitno označeni kao konačni (Java 7) ili koji su efektivno-konačni (Java 8). Ako su definirani u statičkim metodama, anonimni razredi imaju pristup samo statičkim članskim varijablama i metodama vanjskog razreda. Ako su definirani u nestatičkim metodama, ona implicitno hvataju i referencu na trenutni primjerak razreda nad kojim je metoda pozvana pa imaju dodatno još i pristup nestatičkim članskim varijablama i metodama vanjskog razreda.

Modifikator `final`

Modifikator `final` može se koristiti na nekoliko mjesta u programskom jeziku Java. Ovdje ćemo pokriti njegove najčešće uporabe.

Modifikator `final` možemo primijeniti na deklaraciju lokalne varijable, argumenta funkcije te nestatičke ili statičke članske varijable. U tom kontekstu, modifikator varijablu pretvara u konstantu. Jezični će prevodioc prilikom prevođenja koda provjeravati da li neki dio koda pokušava modificirati vrijednost i takve će pokušaje označiti kao pogreške. Evo primjera.

```
final int vrijednost = 17;
```

Statičke članske varijable uz ovaj modifikator koriste se kao globalne konstante; primjerice, u razredu `java.lang.Math` varijabla `PI` deklarirana je upravo tako.

```
public static final double PI = 3.14159265358979323846;
```

Modifikator `final` možemo primijeniti na deklaraciju funkcije. Funkcija koja je označena s `final` ne možemo više biti nadjačana -- to je njezina konačna implementacija. Apstraktne metode nije moguće proglasiti konačnima --činjenica da su apstraktne znači da im nedostaje implementacija; to svakako ne može biti njihovo konačno stanje. Evo primjera.

```
public final int smisaoZivota() {
  return 42;
}
```

Modifikator `final` možemo primijeniti na deklaraciju razreda. Razred koji je označen s `final` ne možemo više naslijediti -- to je njegova konačna implementacija. Jasno je da apstraktni razredni ne mogu istovremeno biti i konačni. Evo primjera.

```
public final class Zatvoreno {
  // ...
}
```

Jedan od primjera konačnih razreda je razred `java.lang.String` -- on je konačan kako ga nitko ne bi mogao naslijediti i tako dobiti pristup njegovim članskim varijablama.

Programiranje prema sučeljima

Sada nakon što smo se upoznali s pojmom apstraktnih razreda i sučelja, pogledajmo jedan ilustrativan primjer koji će razjasniti prednosti pisanja koda koji ovisi o sučeljima a ne o konkretnim implementacijama. Prisjetimo se ponovno primjera obilaska elemenata polja novim oblikom petlje `for` koji smo već vidjeli u poglavlju 1.

```
int[] polje = {1, 4, 7, 11, 13, 21};
for(int element : polje) {
  System.out.println("Trenutni element je: " + element);
}
```

Tada smo spomenuli da je takvim oblikom petlje moguće obilaziti bilo kakvo polje ili kolekciju. U slučaju da je predani skupni element polje, prevodilac će sam kod prekrojiti u onaj koji definira lokalnu varijablu koju će koristiti za indeksiranje elemenata i koji će potom poprimati redom sve vrijednosti od 0 pa do broj elemenata polja minus jedan. No što se događa ako je skupni element kolekcija i radi li to doista za sve kolekcije?

Kako bi se osigurala široka primjenjivost ovog oblika petlje `for`, jasno je da kod petlje mora biti napisan tako da zna vrlo malo o objektu po kojem će obilaziti. Način na koji se to postiže jest definiranjem ugovora između klijentskog koda (u ovom slučaju prevodioca koji treba generirati odgovarajući ekvivalentan kod) i razreda odnosno skupnog objekta čiji će se elementi obilaziti. A taj se ugovor definira uporabom *sučelja*.

Evo ideje. Da bi petlja `for` mogla obilaziti elemente skupnog objekta, skupni objekt mora ponuditi funkcionalnost stvaranja pomoćnog objekta koji će petlja ispitivati i koristiti za dohvat elemenata. Takav pomoćni objekt mora zadovoljavati (odnosno inplementirati) sučelje `java.util.Iterable<E>` koje je prikazano u nastavku. Ove objekte zvat ćemo *iteratorima*.

Primjer 7.8. Sučelje `java.util.Iterator<E>`

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();
  public void remove();
}
```

Slovo `E` napisano u zagradama uz naziv sučelja predstavlja zamjenu za tip elemenata koje sadrži skupni objekt nad kojim će se obavljati obilazak. Primjerice, možemo zamišljati da umjesto `E` piše `String`, `Integer` ili slično. U deklaraciji sučelja potom na svim mjestima gdje je potrebno specificirati tip elementa možemo koristiti tu zamjenu `E`.

Sučelje `Iterator` propisuje da objekt mora nuditi tri metode. Metoda `hasNext` mora vraćati `true` ako još nisu obiđeni svi elementi. Metoda `next` zadužena je za dohvat i vraćanje sljedećeg još neobiđenog elementa. Namjena metode `remove` je brisanje posljednjeg dohvaćenog elementa (kojeg još zovemo i *trenutni element*) iz skupnog objekta, ako to skupni objekt podržava. Ako ne, metoda je slobodna izazvati `UnsupportedOperationException`. Petlji `for` od opisanih metoda dovoljne su već i prve dvije kako bi se mogao napisati ekvivalentni kod koji će obaviti čitav obilazak. Evo pseudokoda.

```
SkupniObjekt obj = ...;
Iterator<Tip> iterator = stvoriIterator(skupniObjekt);
```

```
while(iterator.hasNext()) {
  Tip element = iterator.next();
  obradi(element);
}
```

Uočimo eleganciju ovog pristupa -- zahvaljujući uporabi iteratora, klijentski dio koda ne treba znati apsolutno ništa o skupnom objektu a ipak ga može koristiti na način da obilazi sve njegove elemente, neovisno o načinu na koji su elementi interno pohranjeni u skupnom objektu. Pretpostavka je, naravno, da će iterator znati više o implementacijskim detaljima skupnog objekta i da će znati kako efikasno dohvaćati njegove elemente. Ono što je još ostalo nejasno jest jednostavno pitanje: "kako do iteratora"?

Odgovor na prethodno pitanje u Javi je opet riješen uporabom dogovora -- kako bi petlja `for` (ili bilo koji drugi klijent) mogla zatražiti pristup iteratoru, skupni objekt će ponuditi univerzalnu metodu čija je namjena stvaranje iteratora koji znaju obilaziti njegove elemente. A ovaj je dogovor opet utvrđen -- sučeljem. U paketu java.lang nalazi se definicija sučelja `Iterable<E>` koja je prenesena u nastavku.

Primjer 7.9. Sučelje `java.lang.Iterable<E>`

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

Metoda `iterator` je takozvana *metoda-tvornica*: metoda čija je zadaća da na svaki poziv stvori novi primjerak objekta koji je iterator nad skupnim objektom i koji je spreman za novi obilazak elemenata skupnog objekta. Zadaća skupnog objekta jest da implementira sučelje `java.lang.Iterable<E>` i time ponudi unificiranu mogućnost klijentima da dohvaćaju iteratore koji znaju obilaziti njegove elemente. Ako je to ispunjeno, obilazak je moguće napraviti na način prikazan sljedećim pseudokodom.

```
SkupniObjekt obj = ...;
Iterator<Tip> iterator = skupniObjekt.iterator();
while(iterator.hasNext()) {
  Tip element = iterator.next();
  obradi(element);
}
```

U tom slučaju prethodni kod možemo prepisati uporabom skraćenog oblika koji koristi petlju `for` kako je to prikazano u nastavku.

Primjer 7.10. Obilazak skupnog objekta koji implementira sučelje `java.lang.Iterable<E>`

```
SkupniObjekt skupniObjekt = ...;
for(Tip element : skupniObjekt) {
  obradi(element);
}
```

Petlja `for` prikazana u primjeru 7.10 predstavlja pokratu temeljem koje prevodilac generira prethodno prikazani kod: nad skupnim objektom poziva metodu `iterator` kako bi dohvatio novi primjerak iteratora i potom taj primjerak koristi kako bi u `while`-petlji dohvaćao redom sve elemente.



Petlja `foreach`

Kratki oblik petlje `for` odnosno petlju `foreach` moguće je koristiti nad skupnim objektima koji su ili polja ili primjerci razreda koji implementiraju sučelje `java.lang.Iterable<E>`.

Motivacijski primjer

S obzirom da je ideja programiranja prema sučeljima (odnosno pisanja klijenata koji ovise o sučeljima a ne o konkretnim implementacijama) od iznimne važnosti, pogledat ćemo i konkretan primjer izrade klijenta koji može ispisati sadržaj proizvoljnog skupnog objekta čiji su elementi cijeli brojevi, izrade jednog skupnog objekta koji predstavlja kolekciju parnih brojeva i konačno izrade glavnog programa koji ilustrira njihovu uporabu.

Krenimo s prvim zadatkom. Potrebno je definirati skupni objekt koji će predstavljati skup od zadanog broja uzastopnih parnih brojeva pri čemu se pri stvaranju skupnog objekta zadaju minimalni parni broj te broj uzastopnih parnih brojeva koji pripadaju u zadani skup. Skupni objekt mora biti tako izveden da omogućava obilazak svojih elemenata uporabom skraćenog oblika petlje `for`, što znači da mora implementirati sučelje `java.lang.Iterable<E>`.

S obzirom da jedino sam skupni objekt zna svoje implementacijske detalje, iteratori će biti primjerci njegovog privatnog razreda koji će implementirati sučelje `java.util.Iterator<E>`. Rješenje je prikazano u ispisu 7.11.

Primjer 7.11. Implementacija skupnog objekta koji predstavlja zadani podniz parnih brojeva

```
 1 package hr.fer.zemris.java.tecaj_1;
 2
 3 import java.util.Iterator;
 4
 5 public class ParniBrojevi implements Iterable<Integer> {
 6
 7  private int prvi;
 8  private int n;
 9
10  public ParniBrojevi(int prvi, int n) {
11   if(prvi % 2 != 0) {
12    throw new IllegalArgumentException(
13     "Broj "+prvi+" nije paran!"
14    );
15   }
16   if(n < 0) {
17    throw new IllegalArgumentException(
18     "Broj brojeva je negativan ("+n+")!"
19    );
20   }
21   this.prvi = prvi;
22   this.n = n;
23  }
24
25  @Override
26  public Iterator<Integer> iterator() {
27   return new IteratorBrojeva();
28  }
29
30  private class IteratorBrojeva implements Iterator<Integer> {
31
32   private int trenutni;
33   private int preostaloBrojeva;
34
35   public IteratorBrojeva() {
36    this.preostaloBrojeva = n;
```

```
37     this.trenutni = prvi;
38    }
39
40    @Override
41    public boolean hasNext() {
42     return preostaloBrojeva>0;
43    }
44
45    @Override
46    public Integer next() {
47     if(preostaloBrojeva<1) {
48      throw new RuntimeException("Nema više elemenata!");
49     }
50     int vrijednost = trenutni;
51     trenutni+=2;
52     preostaloBrojeva--;
53     return vrijednost;
54    }
55
56    @Override
57    public void remove() {
58     throw new UnsupportedOperationException(
59      "Brisanje parnih brojeva nije moguće."
60     );
61    }
62   }
63 }
64
```

U ovom primjeru iterator (razred `IteratorBrojeva`) je modeliran kao unutarnji privatni razred skupnog objekta. Stoga primjerci tog razreda imaju pristup do članskih varijabli `n` i `prvi` originalnog skupnog objekta. Iterator za sebe deklarira dvije članske varijable: `trenutni` koja predstavlja trenutni element iteratora (koji će biti vraćen kada se pozove metoda `next`) te `preostaloBrojeva` koja čuva broj brojeva koje iterator još može vratiti. Obje varijable inicijaliziraju se u konstruktoru a ažuriraju u metodi `next`.

Sljedeći korak je izrada programa koji koristi prethodni skupni objekt. Kod je prikazan u ispisu 7.12 i razdvojen je u dva razreda. Privatni razred `Ispisivac` predstavlja generičkog klijenta koji radi neku konkretnu obradu. U našem slučaju, ta se obrada svodi na ispis elemenata proizvoljnog skupnog objekta koji sadrži cijele brojeve. Važno je uočiti da taj kod ni na koji način nije vezan uz naš konkretni skupni objekt: niti je svjestan njegovog imena, niti je svjestan da on uopće postoji. Bilo kakva promjena u implementaciji skupnog objekta neće imati nikakve posljedice na ovog klijenta. U izradi programskih sustava takva su rješenja upravo ona kojima treba težiti: rješenja u kojima postoje slabe veze između različitih dijelova koda i rješenja kod kojih promjene u jednom dijelu koda ne mogu uzrokovati potrebu za promjenama u drugom dijelu koda.

Konačno, metoda `main` koja se nalazi u razredu `DemonstracijaParnihBrojeva` stvara jedan primjerak objekta koji predstavlja slijedni podniz parnih brojeva te koristeći uslugu razreda `Ispisivac` obavlja dva uzastopna ispisa.

Primjer 7.12. Implementacija klijenta za skupni objekt

```
1 package hr.fer.zemris.java.tecaj_1;
2
3 public class DemonstracijaParnihBrojeva {
4
5  public static void main(String[] args) {
```

```
 6    ParniBrojevi pb = new ParniBrojevi(2, 5);
 7
 8    Ispisivac.ispisi("Obilazak prvi puta:", pb);
 9
10    Ispisivac.ispisi("Obilazak drugi puta:", pb);
11   }
12
13   private static class Ispisivac {
14    public static void ispisi(String poruka,
15                              Iterable<Integer> spremnik) {
16
17     System.out.println(poruka);
18     System.out.println("-----------------------");
19     for(Integer broj : spremnik) {
20      System.out.println(broj);
21     }
22     System.out.println();
23    }
24   }
25  }
26
```

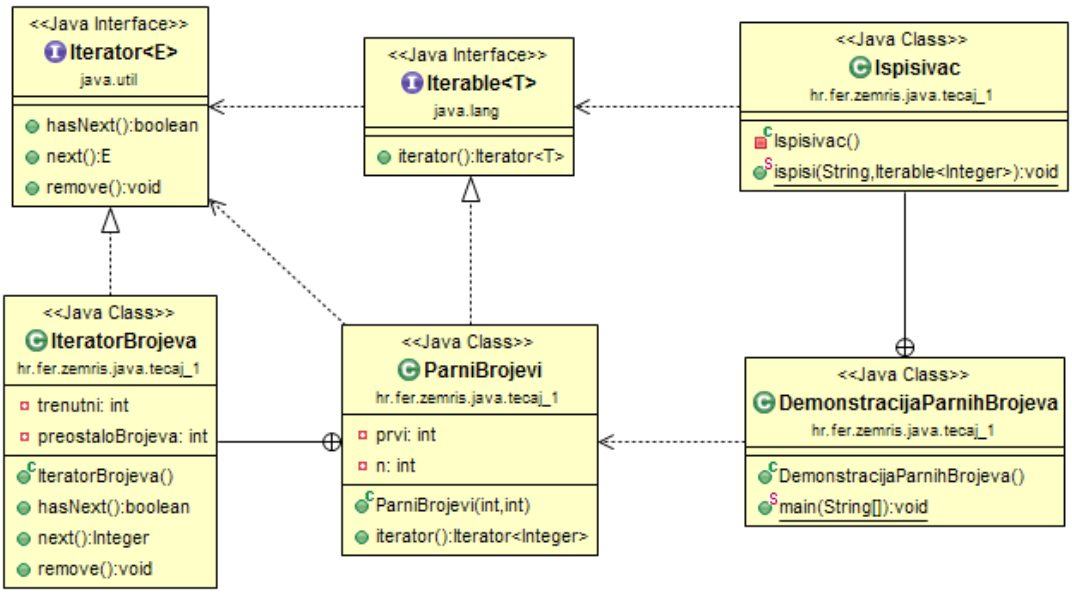
Po pokretanju, program će rezultirati ispisom prikazanim u nastavku.

```
Obilazak prvi puta:
-----------------------
2
4
6
8
10

Obilazak drugi puta:
-----------------------
2
4
6
8
10
```

UML dijagram koji prikazuje definirane razrede, korištena sučelja i njihove međuovisnosti prikazan je na slici 7.12.

Slika 7.4. Razredi i sučelja motivacijskog primjera



Prokomentirajmo malo prikazano rješenje. Postoji skupni objekt `ParniBrojevi` koji implementira sučelje `java.lang.Iterable<E>`. Taj se skupni objekt zainteresiranim klijentima nudi kroz to sučelje. Klijenti (razred `Ispisivac`) komuniciraju s bilo kojim objektom koji je takav da mu se može pristupati kroz sučelje `java.lang.Iterable<E>`. Klijenti koriste funkcionalnost sučelja `java.lang.Iterable<E>` kako bi dobili primjerke konkretnih iteratora. Iteratori su primjerci razreda koji implementiraju sučelje `java.util.Iterator<E>`. Skupni objekt definira privatni razred `IteratorBrojeva` koji implementira sučelje `java.util.Iterator<E>`. Skupni objekt po potrebi stvara primjerke tog privatnog razreda i vraća ih kao objekte koji su iteratori. Vanjski klijenti ne znaju ništa o implementacijskim detaljima tih iteratora, izuzev činjenice da to jesu iteratori i da ih se može koristiti u skladu s ugovorom koji definira sučelje `java.util.Iterator<E>` kako bi se omogućio obilazak elemenata skupnog objekta.

Opisana ideja izuzetno je moćna: osigurava razdvajanje klijenata skupnih objekata od implementacije samih skupnih objekata. Time je osigurano da isti klijent može raditi sa skupnim objektom koji elemente stvara na zahtjev (kao u našem slučaju razreda `ParniBrojevi` koji niti u jednom trenutku u memoriji doista ne sadrži čitavu kolekciju parnih brojeva koje predstavlja), sa skupnim objektom koji elemente čuva u memoriji, sa skupnim objektom koji elemente dohvaća iz datotke, iz baze podataka, preko mreže ili pak sa skupnim objektom koji koristi bilo kakvu alternativnu strategiju za rad s vlastitim elementima.

Ako se radi o skupnim objektima koji su i sami na neki način standardizirani, moguće je pisati programska rješenja kod kojih su skupni objekti i iteratori dva nezavisna elementa i u kojima vanjski klijent može odrediti koju implementaciju iteratora želi koristiti. Međutim, čest je slučaj da, s obzirom na implementacijske specifičnosti skupnih objekata upravo sami skupni objekti imaju zadaću implementiranja i nuđenja odgovarajućih iteratora. Taj slučaj prikazan je i u prethodnom primjeru gdje je implementacija iteratora skrivena unutar razreda skupnog objekta.

Oblikovni obrazac Iterator

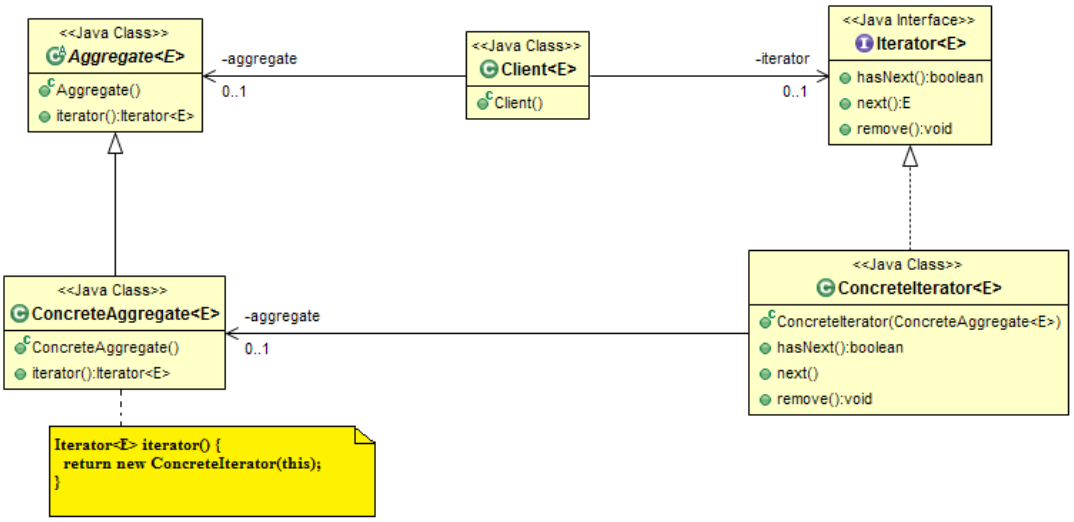
Skupni objekti izuzetno su česti u programskim sustavima a obilazak njihovih elemenata jedan je od najčešćih zadataka. Oblikovni obrazac *Iterator* naputak je koji opisuje dobar način organizacije koda koji će osigurati izradu skupnih objekata koji s kojima će moći raditi proizvoljni klijenti, a bez da su svjesni implementacijskih detalja skupnih objekata.

Prethodni motivacijski primjer već je definirao sve potrebno. Naša je zadaća na ovom mjestu izdvojiti sudionike ovog oblikovnog obrasca, dati im njihovo univerzalno ime i prikazati njihov međusobni odnos.

Namjena obrasca: ponuditi klijentima skupnih objekata koji koriste vrlo različite načine interne pohrane elemenata unificiran način obilaska kroz sve elemente skupnog objekta.

Dijagram razreda obrasca: prikazan na slici 7.5.

Slika 7.5. Dijagram razreda za oblikovni obrazac Iterator



Sudionici obrasca.

- `Aggregate`: apstraktni razred (ili sučelje) koje predstavlja skupni objekt i koje definira postojanje metode `iterator` čija je namjena dohvat iteratora. U Javi ovo je svaki razred koji implementira sučelje `java.lang.Iterable<E>`.

- `ConcreteAggregate`: implementacija apstraktnog skupnog objekta koja definira način pohrane elementa. Definira implementaciju metode `iterator` na način da stvara i vraća primjerke iteratora koji znaju obilaziti kroz njezine elemente (primjerke razreda `ConcreteIterator`).

- `Iterator`: apstraktni razred (ili sučelje) koje definira metode potrebne za iteriranje kroz proizvoljni skupni objekt. Definira postojanje metoda `hasNext`, `next` (a u Javi i `remove`) kojima se omogućava definiranje postupka obilaska.

- `ConcreteIterator`: konkretna implementacija iteratora koji zna obilaziti po elementima primjeraka skupnih objekata koji su predstavljeni razredom `ConcreteAggregate`. Ovaj razred obično ima pristup unutarnjim članskim varijablama skupnog objekta, a česta je situacija da je taj razred i definiran kao privatni razred skupnog objekta tako da nitko osim samog skupnog objekta niti ne može stvarati primjerke tog razreda.

- `Client`: generička implementacija algoritma koji s jedne strane ima referencu na skupni objekt (najčešće samo kroz apstraktni razred) te može dohvaćati primjerke iteratora pozivom metode `iterator()` nad skupnim objektom. Nije svjestan implementacijskih detalja razreda objekta koji dobiva pozivom te metode niti ga to zanima.

Napomene: u slučaju uporabe više iteratora u isto vrijeme, ovisno o implementaciji algoritma iteriranja mogu se pojaviti problemi ako se skupni objekt promijeni prilikom iteriranja. U okviru implementacija standardnih kolekcija koje su uključene u Javu, problem se rješava na način da iterator izazove iznimku ako se kolekcija promijeni izvana tijekom iteriranja. Ipak, kako bi se omogućilo sigurno mijenjanje skupnog elementa bez potrebe za prekidom iteriranja, sučelje `Iterator` donosi metodu `remove` čija je namjena provesti sigurno brisanje trenutnog elementa (onog na kojem je trenutno iterator) iz skupnog objekta. Uporabom te metode iterator će moći promijeniti skupni objekt i normalno nastaviti iteriranje. Međutim, drugi iteratori koji u isto vrijeme obavljaju iteriranje izazvat će iznimku. Ako više iteratora istovremeno radi isklju-

čivo obilazak elemenata, tu se najčešće ne očekuju nikakvi problemi. Uporaba više iteratora iz različitih dretvi u svrhu obilaska (ne i modificiranja) najčešće je sigurno (za više informacija ipak treba konzultirati dokumentaciju konkretnih razreda koji predstavljaju skupne objekte).

Naprednije mogućnosti sučelja

Počev od Jave 8, sučelja mogu ponuditi i takozvane pretpostavljene metode (engl. *default methods*) te statičke metode. Pretpostavljene metode baš kao i klasične metode deklariraju da ih primjerak razreda koji implementira to sučelje mora imati. Međutim, za razliku od klasičnih metoda sučelja, pretpostavljene metode nude i implementaciju pa iz razred koji implementira to sučelje može nanovo implementirati ako želi, a ako to ne učini, koristit će se implementacija ponuđena u sučelju. Pretpostavljene metode označavaju se ključnom riječi `default`. Evo primjera.

```
public interface I1 {

 double m1();
 default double m2() { return 42; }

}

public class C1 implements I1 {

 public double m1() {
  return 12;
 }

}

public class C2 implements I1 {

 public double m1() {
  return 12;
 }

 public double m2() {
  return 12;
 }

}
```

Definirali smo sučelje `I1` koje definira metode `m1` i `m2`, pri čemu za `m2` nudi i implementaciju. Potom smo definirali dva razreda: `C1` i `C2` i niti jedan od njih nije apstraktan. `C1` donosi implementaciju metode `m1` a iz sučelja preuzima implementaciju metode `m2`. Razred `C1` donosi implementacije obje metode.

Ako razred implementira dva sučelja koja definiraju pretpostavljene metode iste signature, onda prevoditelj neće automatizmom uzeti niti jednu od njih. U tom je slučaju programer dužan ponuditi svoju implementaciju. Ako u okviru te implementacije želi koristiti pretpostavljenu metodu iz jednog od tih sučelja, to može učiniti, ali mora napisati eksplicitni poziv uporabom ključne riječi `super` i navođenjem naziva sučelja iz kojeg preuzima implementaciju. Evo primjera koji se nastavlja na prethodni.

```
public interface I2 {

 double m3();

 default double m2() {
```

```
  return 58;
 }
}

public class C3 implements I1, I2 {

 public double m1() {
  return 12;
 }

 @Override
 public double m3() {
  return 13;
 }

 @Override
 public double m2() {
  return I1.super.m2();
 }

}
```

Razred `C3` implementira oba sučelja, i bez definiranja svoje implementacije metode `m2` prevoditelj dojavljuje pogrešku. U primjeru smo se odlučili pozvati implementaciju iz sučelja `I1`, pa je poziv `return I1.super.m2();`.

Uz pretpostavljene metode, sučelja mogu definirati i statičke metode. Statičke metode se definiraju uporabom ključne riječi `static`, i slično kao pretpostavljene metode, moraju ponuditi implementaciju. Razlika između statičkih i pretpostavljenih metoda je ista kao i kod razreda između nestatičkih i statičkih: prve pripadaju primjerku razreda (objektu) i dobivaju referencu `this` dok druge pripadaju razredu/sučelju i kod njih ne postoji referenca `this`. Također, statičke metode nije moguće nadjačati (ista je situacija kao i kod razreda). Evo primjera.

```
public interface I3 {

 double m4();

 static double m5() {
  return 58;
 }

}
```

Statičke metode moguće je "izvana" pozivati navođenjem naziva sučelja, kao što je to slučaj i kod razreda. Primjer: `double n = I3.m5();`.

Lambda izrazi

Sada nakon što smo se upoznali s vrstama razreda, za kraj ćemo se još fokusirati na jednu od tih vrsta: na *anonimne razrede*. Krenut ćemo s jednostavnim motivacijskim primjerom. Zadaća nam je napisati program koji će za primljeno polje decimalnih brojeva stvoriti novo polje jednake veličine koje je na *i*-toj lokaciji imati pohranjenu vrijednost funkcije *sinus* izračunate nad brojem koji je u originalnom polju na toj istoj poziciji. Uz pretpostavku da smo polje direktno pripremili u metodi `main`, rješenje je prikazano u primjeru 7.13.

Primjer 7.13. Osnovno rješenje

```
1 package book.java.lambde;
```

```
 2
 3 public class Primjer1 {
 4
 5  public static void main(String[] args) {
 6   double[] polje = new double[] {1,2,3,4,5,6,7,8,9,10};
 7
 8   double[] transformirano = transformiraj(polje);
 9
10   ispisi(polje, transformirano);
11  }
12
13  private static void ispisi(double[] arg, double[] f) {
14   for(int i = 0; i < arg.length; i++) {
15    System.out.format("%f -> %f%n", arg[i], f[i]);
16   }
17  }
18
19  private static double[] transformiraj(double[] polje) {
20   if(polje == null) {
21    throw new IllegalArgumentException(
22     "Predana je null referenca kao polje!"
23    );
24   }
25   double[] rezultat = new double[polje.length];
26   for(int i = 0; i < rezultat.length; i++) {
27    double vrijednost = Math.sin(polje[i]);
28    rezultat[i] = vrijednost;
29   }
30   return rezultat;
31  }
32
33 }
34
```

Metoda `main` sastoji se od samo tri naredbe: pripreme polja s podatcima koje treba tranformirati, poziva metode `transformiraj` koja obavlja traženu transformaciju te poziva metode `ispisi` koja rezultat ispisuje na zaslon. Sama metoda `transformiraj` relativno je jednostavna: provjerava je li dobila referencu na polje koja je različita od `null` te ako nije, izaziva iznimku. U suprotnom alocira novo polje za pohranu rezultata, prolazi element po element po originalnom polju, računa vrijednost funkcije sinus koju pohranjuje u novo polje i na kraju vraća novostvoreno polje koje sadrži izračunate vrijednosti funkcije sinus.

Pretpostavimo sada da Vam zatreba još i vrijednost funkcije kosinus. Kako najjednostavnije modificirati program? Prvo što pada napamet je pristup *copy&paste*: iskopirajmo metodu `transformiraj` u `transformiraj2` i promijenimo samo funkciju koju je potrebno računati. Rješenje je prikazano u primjeru 7.14.

Primjer 7.14. Nadopunjeno rješenje

```
1 package book.java.lambde;
2
3 public class Primjer2 {
4
5  public static void main(String[] args) {
6
7   double[] polje = new double[] {1,2,3,4,5,6,7,8,9,10};
8
9   double[] transformirano = transformiraj(polje);
```

```
10    double[] transformirano2 = transformiraj2(polje);
11
12    ispisi(polje, transformirano);
13    ispisi(polje, transformirano2);
14   }
15
16   private static void ispisi(double[] arg, double[] f) {
17    for(int i = 0; i < arg.length; i++) {
18     System.out.format("%f -> %f%n", arg[i], f[i]);
19    }
20   }
21
22   private static double[] transformiraj(double[] polje) {
23    if(polje == null) {
24     throw new IllegalArgumentException(
25      "Predana je null referenca kao polje!"
26     );
27    }
28    double[] rezultat = new double[polje.length];
29    for(int i = 0; i < rezultat.length; i++) {
30     double vrijednost = Math.sin(polje[i]);
31     rezultat[i] = vrijednost;
32    }
33    return rezultat;
34   }
35
36   private static double[] transformiraj2(double[] polje) {
37    if(polje == null) {
38     throw new IllegalArgumentException(
39      "Predana je null referenca kao polje!"
40     );
41    }
42    double[] rezultat = new double[polje.length];
43    for(int i = 0; i < rezultat.length; i++) {
44     double vrijednost = Math.cos(polje[i]);
45     rezultat[i] = vrijednost;
46    }
47    return rezultat;
48   }
49
50  }
51
```

Ovakvo rješenje, međutim, nije u duhu s dobrim načelima oblikovanja koda. Kopiranjem smo u kod unijeli redundanciju: i metoda `transformiraj` i metoda `transformiraj2` obje sadrže kod za provjeru je li predana vrijednost različita od `null`; sadrže odluku o konkretnoj poruci koju treba ispisati ako to nije slučaj; sadrže kod koji alocira novo polje za pohranu rezultata; sadrže kod koji obavlja iteriranje po izvornom polju; sadrže kod koji računa transformaciju izvorne vrijednosti; sadrže kod koji taj rezultat pohranjuje te konačno sadrže kod koji vraća izračunati rezultat. Što ako nam kojim slučajem još zatreba i kvadrat broja? Hoćemo li još jednom sve to kopirati? Ili ako odlučimo promijeniti tekst poruke koju treba upisati u iznimku koja se izaziva ako se preda `null` kao argument - na koliko ćemo mjesta to morati tada mijenjati?

Zapitajmo se radije: kako pripremiti bolje rješenje? Usporedimo li metode `transformiraj` i `transformiraj2` (i sve ostale slične metode koje bismo još mogli izgenerirati), pronaći ćemo samo jednu razliku između svih njih: *kako se točno ulazna vrijednost transformira u rezultat*. Stoga ćemo u našem novom rješenju sučeljem `ITransform` modelirati objekte čija je zadaća ostvarivanje neke transformacije. Sučelje će deklarirati da mora postojati metoda `double`

`transform(double value);` koja dobije broj i vrati njegovu transformiranu vrijednost. Potom ćemo pripremiti nekoliko razreda koji implementiraju ovo sučelje i svaki u metodi `transform` računa drugu transformaciju. Konačno, obrisat ćemo sve kopije metoda `transformiraj` i zamijeniti ih samo jednom: metodom koja prima polje i referencu na objekt koji radi transformaciju. Takvo rješenje prikazano je u primjeru 7.15.

Primjer 7.15. Poboljšano rješenje

```
 1 package book.java.lambde;
 2
 3 public class Primjer3 {
 4
 5  public static void main(String[] args) {
 6
 7   double[] polje = new double[] {1,2,3,4,5,6,7,8,9,10};
 8
 9   double[] transformirano = transformiraj(
10    polje, new SinTransform()
11   );
12   double[] transformirano2 = transformiraj(
13    polje, new CosTransform()
14   );
15
16   ispisi(polje, transformirano);
17   ispisi(polje, transformirano2);
18  }
19
20  private static void ispisi(double[] arg, double[] f) {
21   for(int i = 0; i < arg.length; i++) {
22    System.out.format("%f -> %f%n", arg[i], f[i]);
23   }
24  }
25
26  public interface ITransform {
27   double transform(double value);
28  }
29
30  public static class SinTransform implements ITransform {
31   public double transform(double value) {
32    return Math.sin(value);
33   };
34  }
35
36  public static class CosTransform implements ITransform {
37   public double transform(double value) {
38    return Math.cos(value);
39   };
40  }
41
42  private static double[] transformiraj(
43    double[] polje, ITransform transformer) {
44   if(polje == null) {
45    throw new IllegalArgumentException(
46     "Predana je null referenca kao polje!"
47    );
48   }
49   double[] rezultat = new double[polje.length];
50   for(int i = 0; i < rezultat.length; i++) {
```

```
51     double vrijednost = transformer.transform(polje[i]);
52     rezultat[i] = vrijednost;
53    }
54    return rezultat;
55   }
56
57 }
58
```

Spomenuto sučelje definirano je u retcima 26-28. Konkretne implementacije tog sučelja su razredi `SinTransform` (retci 30-34) i `CosTransform` (retci 36-40). Općenita metoda `transformiraj` koja sada može obavljati bilo kakvu transformaciju prikazana je u retcima 42-55. Uporaba ovakvog rješenja prikazana je u metodi `main`, retci 9-14. Uočite kako se pri svakom pozivu metode `transformiraj` predaje polje s podatcima te referenca na primjerak razreda koji zna obavljati željenu transformaciju.



Oblikovni obrazac *Strategija*

Rješenje koje smo ovdje prikazali ilustrira primjenu oblikovnog obrasca *Strategija*. Obrazac poprilici kaže sljedeće: ako imamo puno sličnih kodova koji variraju na nekom mjestu (a ostatak koda je jednak), tada to što varira treba modelirati sučeljem, za svaki različit način variranja pripremiti razred koji to implementira te potom obrisati sve kopije koda i ostaviti samo jednu koja je proširena referencom na objekt koji obavlja transformaciju i koja variranje obavlja pozivom tog objekta.

Nastavimo dalje s primjerom. Pretpostavimo da nam još zatreba i transformacija oblika `2*x-4`. Međutim, ne očekujemo da ćemo to trebati na više mjesta već isključivo ovdje u metodi `main`. Umjesto da radimo "punokrvni" razred, možemo se poslužiti anonimnim razredom - ovo je tipičan primjer njihove uporabe. Takvo rješenje prikazano je u primjeru 7.16.

Primjer 7.16. Rješenje uz uporabu anonimnog razreda

```
 1 package book.java.lambde;
 2
 3 public class Primjer4 {
 4
 5  public static void main(String[] args) {
 6
 7   double[] polje = new double[] {1,2,3,4,5,6,7,8,9,10};
 8
 9   double[] transformirano = transformiraj(
10    polje, new SinTransform()
11   );
12   double[] transformirano2 = transformiraj(
13    polje, new CosTransform()
14   );
15   double[] transformirano3 = transformiraj(
16    polje, new ITransform() {
17     @Override
18     public double transform(double value) {
19      return 2 * value - 4;
20     }
21    }
22   );
23
24   ispisi(polje, transformirano);
25   ispisi(polje, transformirano2);
26   ispisi(polje, transformirano3);
```

```
 27   }
 28
 29   // ostatak koda kao u Primjer3...
 30   // ...
 31 }
 32
```

Definiranje i stvaranje primjerka anonimnog razreda koji obavlja zadanu transformaciju prikazano je u retcima 16-21. Od Jave 8 za definiranje i stvaranje primjerka anonimnog razreda koji ima samo jednu metodu koju je potrebno implementirati možemo koristiti sintaksno kraći zapis (odnosno pokratu): trebamo napisati naziv argumenta (ili argumenata ako ih ima više, razdvojenih zarezima), potom znak minus pa veće ("->") i potom dođe tijelo te (jedne i jedine) metode. Konkretno, dio koda:

```
new ITransform() {
 @Override
 public double transform(double value) {
  return 2 * value - 4;
 }
}
```

možemo zamijeniti sintaksno kraćim zapisom:

```
(double value) -> {return 2 * value - 4;}
```

čime bi čitav poziv obavljanja treće transformacije mogli zapisati kako je prikazano u nastavku.

```
double[] transformirano3 = transformiraj(
 polje, (double value) -> {return 2 * value - 4;}
);
```

Izraz `(double value) -> {return 2 * value - 4;}` predstavlja jedan od oblika *lambda*-izraza. Prikazani izraz možemo pokratiti na još nekoliko načina koji su također ispravni. Najprije, nije potrebno pisati tip argumenata, pa onaj `double` možemo ispustiti. Nadalje, ako metoda koju implementiramo ima samo jedan argument, onda možemo ispustiti i oble zagrade. Konačno, s desne strane umjesto bloka naredbi koji je ograđen vitičastim zagradama i ima eksplicitnu naredbu `return` možemo ispustiti vitičaste zagrade i napisati samo izraz koji treba vratiti (bez `return` i bez točke-zarez na kraju). Stoga su svi sljedeći izrazi također ekvivalentni.

```
(value) -> {return 2 * value - 4;}
value -> {return 2 * value - 4;}
(double value) -> 2 * value - 4
(value) -> 2 * value - 4
value -> 2 * value - 4
```

Ponovimo još jednom: lambda-izraz je sintaksna pokrata za pisanje koda koji definira i stvara primjerak anonimnog razreda koji ima samo jednu metodu koju treba implementirati. Stoga, baš kao što možemo napisati sljedeće:

```
ITransform tr = new ITransform() {
 @Override
 public double transform(double value) {
  return 2 * value - 4;
 }
};
System.out.println(tr.transform(5));
System.out.println(tr.transform(10));
```

možemo napisati identičan ali sintaksno puno kraći oblik:

```
ITransform tr = t -> 2 * value - 4;
```

```
System.out.println(tr.transform(5));
System.out.println(tr.transform(10));
```

Postoji još nekoliko oblika lambda-izraza koje možemo koristiti. Pretpostavimo da postoji razred `Funkcije` (primjer 7.17) koji deklarira nekoliko statičkih metoda.

Primjer 7.17. Pomoćni razred Funkcije

```
 1 package book.java.lambde;
 2
 3 public class Funkcije {
 4
 5  public static double sinus(double x) {
 6   return Math.sin(x);
 7  }
 8
 9  public static double cosinus(double x) {
10   return Math.cos(x);
11  }
12
13  public static double squareRoot(double x) {
14   return Math.sqrt(x);
15  }
16
17 }
```

Stvaranje primjerka anonimnog razreda koji obavlja transformaciju uporabom statičke metode `squareRoot` razreda `Funkcije`:

```
ITransform tr1 = new ITransform() {
 @Override
 public double transform(double value) {
  return Funkcije.squareRoot(value);
 }
};
```

možemo (što smo već naučili) sintaksno pokratiti ovako:

```
ITransform tr2 = value -> Funkcije.squareRoot(value);
```

ili još kraće ovako:

```
ITransform tr3 = Funkcije::squareRoot;
```

U ovom posljednjem primjeru definira se lambda kod koje se uopće ne navode imena argumenata metode koja se implementira: prevoditelj će za metodu odabrati imena i argumente tim redosljedom poslati statičkoj metodi koju smo naveli. Uočite da se između imena razreda i imena metode ne koristi znak točke već dvostruka dvotočka. Da smo trebali implementirati metodu koja prima dva argumenta, statička metoda koju bismo ovako naveli morala bi primati dva argumenta - mi bismo i dalje naveli samo ima razreda, dvije dvotočke i ime te metode.

Sintaksa lambda-izraza s dvostrukom dvotočkom može se koristiti u još nekoliko situacija u kojima označava različite isječke koda. Pretpostavimo da imamo razred:

```
package book.java.lambde;

public class ComplexNumber {

 private double re;
 private double im;
```

```
 public ComplexNumber(double re, double im) {
  super();
  this.re = re;
  this.im = im;
 }

 public double getRe() {
  return re;
 }

 public double getIm() {
  return im;
 }

}
```

te sučelje:

```
public interface IExtractor {
 double extract(ComplexNumber n);
}
```

koje modelira objekt koji iz kompleksnog broja dohvaća jednu skalarnu vrijednost. To može biti, primjerice, njegov realni dio, njegov imaginarni dio, njegov kut, njegov modul i slično. Objekt koji računa modul kompleksnog broja prikazan je u nastavku uporabom anonimnog razreda:

```
IExtractor e1 = new IExtractor() {
 @Override
 public double extract(ComplexNumber n) {
  return n.getRe()*n.getRe() + n.getIm()*n.getIm();
 }
};
```

odnosno kraće lambdom:

```
IExtractor e2 = n -> Math.sqrt(
  n.getRe()*n.getRe() + n.getIm()*n.getIm()
);
```

Objekt koji iz kompleksnog broja dohvaća njegov realni dio možemo ostvariti anonimnim razredom:

```
IExtractor e3 = new IExtractor() {
 @Override
 public double extract(ComplexNumber n) {
  return n.getRe();
 }
};
```

lambda-izrazom:

```
IExtractor e4 = n -> n.getRe();
```

ili još kraćim lambda-izrazom:

```
IExtractor e5 = ComplexNumber::getRe;
```

U ovom posljednjem slučaju, oblik "razred dvostruka-dvotočka metoda" predstavlja pokratu za poziv navedene metode (koja ne prima niti jedan argument) nad objektom koji se dobiva kao argument metode koju implementiramo (pogledajte to u onom primjeru koji koristi punu sintaksu definiranja i stvaranja anonimnog razreda).

Ako se pitate kako prevoditelj zna na koji način protumačiti ovaj kratak izraz, odgovor je: "razmatranjem konteksta". U ovom konkretnom primjeru, prevoditelj vidi (temeljem lijeve strane znaka jednakosti) da pokušavamo definirati primjerak razreda koji implementira sučelje `IExtractor`, što znači da moramo definirati metodu koja prima jedan argument (označimo ga za potrebe ovog objašnjenja s X). Potom analizira što je točno metoda koju smo lambdom naveli. Ako je ona statička, onda mora imati jedan argument i prevoditelj će generirati kod koji poziva tu statičku metodu i predaje joj X kao argument. Ako je metoda statička i ne prima točno jedan argument, prevoditelj će dojaviti pogrešku. Ako metoda nije statička, onda (budući da naša metoda dobiva samo jedan argument) ne smije primati niti jedan argument i generirat će se kod koji nad X poziva tu metodu. Ako metoda ima jedan ili više argumenata, opet se generira pogreška. Točna interpretacije ove vrste lambda izraza ovisi dakle o kontekstu.

Pokažimo za kraj još jedan primjer. Pretpostavimo sada da imamo binarnu funkciju: objekt koji prima dva argumenta i temeljem njih nešto računa i vraća rezultat. Dajmo opet jedan konkretan primjer. Pretpostavimo da imamo razred koji modelira trokomponentni vektor realnih brojeva:

```
package book.java.lambde;

public class Vector3D {

 private double x;
 private double y;
 private double z;

 public Vector3D(double x, double y, double z) {
  super();
  this.x = x;
  this.y = y;
  this.z = z;
 }

 public double getX() {
  return x;
 }

 public double getY() {
  return y;
 }

 public double getZ() {
  return z;
 }

 public Vector3D add(Vector3D other) {
  return new Vector3D(x+other.x, y+other.y, z+other.z);
 }

 public static Vector3D sum(Vector3D v1, Vector3D v2) {
  return v1.add(v2);
 }

}
```

te da imamo sučeljem opisan objekt čija je zadaća obavljanje neke (proizvoljne) operacije nad dva takva vektora.

```
public interface IVectorOp {
 Vector3D operate(Vector3D first, Vector3D second);
}
```

Neke od mogućih operacija su, primjerice, odabir prvog vektora, odabir drugog vektora, izračun novog vektora koji predstavlja sumu dva vektora, izračun novog vektora koji predstavlja aritmetičku sredinu tih vektora i slično.

Prikažimo opet nekoliko implementacija takvog objekta. Fokusirajmo se najprije na objekt koji će računati sumu dvaju vektora uporabom nestatičke metode `add` koju razred `Vector3D` nudi. Rješenje uporabom anonimnog razreda je:

```
IVectorOp vop1 = new IVectorOp() {
 @Override
 public Vector3D operate(Vector3D first, Vector3D second) {
  return first.add(second);
 }
};
```

Rješenje koje to isto ostvaruje uporabom "klasične" lambde slijedi u nastavku. Uočite, s obzirom da metoda koju definiramo ima dva argumenta, moramo ih navesti u oblim zagradama koje su sada obavezne.

```
IVectorOp vop2 = (f,s) -> f.add(s);
```

Konačno, najkraći način da ovo ostvarimo je sljedećom lambdom:

```
IVectorOp vop3 = Vector3D::add;
```

Metoda koju implementiramo prima dva argumenta, pa oblik "razred dvostruka-dvotočka metoda" mora pogađati ili nestatičku metodu koja prima jedan argument (pa ju pozivamo nad prvim argumentom i predajemo joj drugi argument), ili statičku metodu koja prima dva argumenta. U ovom konkretnom slučaju, radi se o prvom slučaju što je vidljivo ih prethodne lambde.

Istu funkcionalnost možemo ostvariti uporabom statičke metode `sum` koju nam nudi razred `Vector3D`. Rješenje koje koristi anonimni razred je:

```
IVectorOp vop4 = new IVectorOp() {
 @Override
 public Vector3D operate(Vector3D first, Vector3D second) {
  return Vector3D.sum(first, second);
 };
};
```

Sintaksno kraće rješenje klasičnom lambdom je:

```
IVectorOp vop5 = (f,s) -> Vector3D.sum(f, s);
```

a najkraće rješenje ekvivalentnom lambdom je:

```
IVectorOp vop6 = Vector3D::sum;
```



Lambda izrazi stvaraju objekte

Lambda izraze često ćemo koristiti jer ispravno korišteni mogu značajno smanjiti količinu koda koji je potrebno pisati, čime štede vrijeme programeru ali mogu i doprinijeti povećanju čitljivosti koda. Međutim, treba boditi računa da je svaki lambda-izraz poziv operatora `new`: alociraju se novi objekti, pa ako su performanse problem, tada treba pogledati gdje se u kodu definiraju takvi izrazi i po potrebi ih definirati izvan petlji pa u petljama koristiti, čime će se izbjeći alokacija novog objekta u svakoj iteraciji petlje.

Ovo, dakako, treba raditi samo ako se uoči da su performanse programa postale problematične.

Iako smo do sada stvaranje specifičnih primjeraka anonimnih razreda (specifičnih u smislu da moraju imati samo jednu metodu koju treba implementirati) i lambde tretirali kao isto, napomenimo da između njih ipak postoji sitna razlika: prevoditelj lambda-izraz prilikom analize ne tretira kao novi doseg. Sjetite se: napišete li ključnu riječ `this` u metodi anonimnog razreda, ta referenca se odnosi na primjerak tako stvorenog anonimnog razreda. Lambda ne unosi novi doseg i tretira se kao i bilo koja druga jednostavna naredba: `this` se u lambdi odnosi na vanjski objekt u kojem je izraz definiran. Od ograničenja, sve što vrijedi za anonimne razrede, vrijedi i za lambde (izvana vide samo finalne ili efektivno-finalne lokalne varijable i slično).

Funkcijska sučelja

Uz podršku za lambda-izraze, u Javi 8 dodana je anotacija `@FunctionalInterface`. Uporabom ove anotacije nad sučeljem programer poručuje prevoditelju da misli da to sučelje ima samo jednu metodu (odnosno, da bi ga trebalo biti moguće koristiti u lambda izrazima). Prevoditelj će prilikom prevođenja provjeriti je li to istina i ako nije, generirat će poruku o pogrešci. Mehanizam je sličan kao i kod anotacije `@Override` kojom prevoditelju poručujete da mislite da nadjačavate roditeljsku metodu. Mehanizam nadjačavanja radi neovisno o toj anotaciji; svrha joj je omogućiti da Vas prevoditelj upozori ako niste postigli to što mislite da jeste. U istom duhu se tretira i anotacija `@FunctionalInterface`. Evo primjera njezine uporabe.

```
@FunctionalInterface
public interface IVectorOp {
 Vector3D operate(Vector3D first, Vector3D second);
}
```

S Javom 8 dodan je i novi paket `java.util.function` koji sadrži niz funkcijskih sučelja koja se koriste na raznim mjestima unutar Javinih biblioteka, a posebice u okviru Javinog okvira kolekcija. Spomenut ćemo samo dva sučelja koja će Vas podsjetiti na prethodne primjere. Sučelje `java.util.function.DoubleUnaryOperator` modelira unarnu funkciju nad realnim brojevima: objekt prima realni broj i transformira ga u neki drugi realni broj (primjerice, u suprotan, u sinus, u kvadrat, ...). `java.util.function.DoubleBinaryOperator` modelira binarnu funkciju nad realnim brojevima: objekt prima dva realna broja i transformira ih u neki treći realni broj (primjerice, u sumu, razliku, umnožak, ...). Oba sučelja prikazana su u nastavku.

```
@FunctionalInterface
public interface DoubleUnaryOperator {
 double applyAsDouble(double operand);
 default DoubleUnaryOperator compose(DoubleUnaryOperator before) {
  ...
 }
 default DoubleUnaryOperator andThen(DoubleUnaryOperator after) {
  ...
 }
 static DoubleUnaryOperator identity() {
  ...
 }
}

@FunctionalInterface
public interface DoubleBinaryOperator {
    double applyAsDouble(double left, double right);
}
```

Poglavlje 8. Načela objektno-orijentiranog oblikovanja

Izrada programskih sustava vrlo je složen i zahtjevan zadatak. To je posebice tako ako govorimo o izradi programskih sustava koji ispoljavaju niz pozitivnih karakteristika, odnosno koji nemaju tipične patologije lošeg koda poput:

- *otpornosti na promjene* (engl. *rigid code*),
- *krhkosti* (engl. *fragile code*) te
- *nepokretnosti* (engl. *difficult to reuse*).

U praksi se često javlja potreba da se funkcionalnost postojećeg koda prilagodi ili da se nadopuni novom funkcionalnošću. Ovisno o strukturi programskog rješenja i međuovisnosti njegovih komponenata, taj posao može biti lagano provediv ili pak gotovo nemoguć. Za kod kažemo da je *otporan na promjene* ako ga je teško mijenjati. Takav kod se zbog toga najčešće i ne mijenja — programer postaje sklon živjeti i s manjom i neadekvatnom funkcionalnošću jer je modificiranje i dodavanje nove funkcionalnosti teško, a to svakako nije poželjna osobina programskog sustava.

Postoje i programski sustavi kod kojih postoji niz komponenata koje su međusobno vrlo isprepletene i kod kojih se svaka komponenta oslanja na "intimno" poznavanje drugih komponenata. U takvim sustavima česti su dugački lanci ovisnosti komponenata a ponekad čak i cikličke ovisnosti. Posljedica toga je da promjena u bilo kojoj od komponenata koju je nužno napraviti najčešće potrga postojeći kod. Kod se prestaje prevoditi ili ako prevođenje i uspije, javlja se niz pogrešaka u radu programskog sustava i to često na nizu različitih mjesta. Kod koji ispoljava ovakve karakteristike nazivamo *krhkim* kodom.

Konačno, kada pišemo više programskih sustava, a posebice onih slične funkcionalnosti, uobičajeno je da smo u nekom od programskih sustava već napisali kod (neki modul ili biblioteku) koji nam se čini baš prikladan i za novi programski sustav te koji bismo htjeli prenijeti a ne još jednom pisati ispočetka. Nažalost, ako je kod u starom programskom sustavu čvrsto povezan s drugim komponentama tog sustava odnosno ako dijelovi tog koda imaju ovisnosti prema drugim dijelovima tog sustava, takav ćemo kod teško moći prenijeti u novi sustav koji razvijamo. Da bi to bilo moguće, trebat će uložiti puno truda kako bi se i u novom sustavu implementirale ekvivalentne komponente o kojima promatrani kod ovisi ili će pak trebati uložiti mnogo truda kako bi se takav kod preuredio. U oba slučaja, govorimo o kodu koji je *nepokretan*. Nepokretnost programskog koda loše je svojstvo koda i prilikom oblikovanja programskog sustava treba uložiti potreban napor kako bi se izbjeglo stvaranje takvog koda.

Kako onda oblikovati programski sustav koji će u što je moguće manjoj mjeri iskazivati svojstva otpornosti na promjene, krhkosti te nepokretnosti? Postoji čitav niz smjernica koje nam mogu pomoći u ostvarenju tog cilja. Evo za početak nekoliko temeljnih načela.

- Nemojte se ponavljati (engl. *Do not repeat yourself*).
- Recite stvari jednom i samo jednom (engl. *Once and only once*).
- Mora postojati jedan izvor istine (engl. *Single point of truth*).

Prilikom pisanja određenih funkcionalnosti, posebice ako su one jednostavnije, programeri su skloni svaki puta potreban kod napisati ispočetka. Banalni primjer je konstrukcija i inicijalizacija neke podatkovne strukture što je posebno često u programskim jezicima niže razine poput programskog jezika C. Gdje god je potrebno, programer će pozvati `malloc` kako bi zauzeo memoriju za objekt te potom napisati nekoliko linija koda koje će inicijalizirati elemente zauzete strukture. Nažalost, kako niti jedan kod nije otporan niti na pogreške niti na promjene, u jednom trenutku će se pojaviti potreba da se postupak zauzimanja i inicijalizacije objekata modi-

ficira. Posljedica će biti dugotrajno pretraživanje postojećeg koda u pokušaju da se pronađu sva mjesta na kojima se obavlja taj postupak i prepravljanje na svim pronađenim mjestima. Dakako, u nadi da smo uopće pronašli sva mjesta. Stoga je jedno od važnih načela — *nemojte se ponavljati*.

Prilikom pisanja koda, s vremena na vrijeme javit će se potreba za implementiranjem funkcionalnosti koja je već prisutna u kodu u istom ili ponešto drugačijem obliku. Umjesto ponovne implementacije te funkcionalnosti ili implementacije slične funkcionalnosti koja s postojećom dijeli velik dio koda, potrebno je pristupiti prekrojavanju koda (engl. *code refactoring*). Zadaća prekrojavanja jest osigurati da se eliminira višestruko ponavljanje, te da se, po potrebi, određena ponašanja apstrahiraju i riješe na jednom mjestu. Prilikom pisanja koda važno je *da stvari kažete jednom i samo jednom*.

Konačno, uočimo da je kod koji sadrži ponavljanja kod koji je teško mijenjati. Bilo kakva promjena mora biti provedena na svim mjestima koja sadrže duplikat koda. Ako se to ne provede, dolazi se do situacije u kojoj kopije koda više nisu sinkrone i svaka na malo drugačiji način obavlja svoj posao. Ovo pak može dovesti do ozbiljnih problema u ponašanju programskog sustava. Stoga je važno da u kodu za svako ponašanje postoji samo jedna implementacija — *jedan izvor istine*.

Interesantna rasprava na temu prethodna tri principa može se pronaći na Internetu — potražite *Curly's Law: Do One Thing*[1].

Sljedeća tri načela također su široko poznata u svijetu oblikovatelja programskih sustava a pripadaju u porodicu načela koja, iako jednostavna, često su teško sljediva. Radi se o načelima:

- KISS (engl. *Keep It Simple Stupid*, ili u blažoj varijanti *Keep It Simple Silly*),
- DRY (engl. *Don't Repeat Yourself*) te
- YAGNI (engl. *You Ain't Gonna Need It*).

Načelo *KISS* govori nam nešto sasvim očito: nemojmo komplicirati više no što je to stvarno potrebno. Prilikom izrade programskog rješenja, programeri često pokušavaju pisati kod koji će biti dovoljno općenit kako bi se lagano prilagodio i mnoštvu potencijalnih budućih primjena. Iako pohvalno, do takvih potencijalnih primjena najčešće ne dođe; zahtjevi na programski sustav se promijene i završavamo s hrpom napisanog koda koji nam više neće trebati (i nikada nije), na koji smo trošili vrijeme osmišljajući ga, testirajući ga i popravljajući pogreške koje smo u njega unijeli. Evo zgodnog primjera koji će to ilustrirati. Zamislimo da razvijamo neki programski sustav u kojem nam treba funkcionalnost duple dvostruko povezane liste (svaki čvor ima referencu na prethodni čvor po dva kriterija i na sljedeći čvor po dva kriterija). Recimo da našem programu treba samo funkcionalnost umetanja novih elemenata na početak liste te brisanje opet s početka liste. Slijeđenjem načela *KISS* dodali bismo kod koji radi upravo to. U praksi, dobar dio programera će zaključiti da, kad već piše taj kod, zašto se onda ograničiti samo na dodavanje i brisanje s početka kad se mogu napisati i općenite metode za dodavanje i brisanje na bilo koje mjesto (pa čak i na poziciju koja je veća od trenutnog broja elemenata)? Naravno, bilo bi zgodno imati i mogućnost umetanja na odgovarajuće mjesto tako da lista ostane sortirana, pisanje dodatnih procedura za sortiranje nesortiranih lista, enkapsuliranje takvih lista u razrede koji se korisniku predstavljaju kao stogovi ili redovi, a zgodno dođe i metoda koja bi to znala presložiti u binarno stablo. Hm, da. Bilo bi to zgodno. *KISS*.

Mane ponavljanja već smo istakli kroz prethodni blok od dana tri načela — javljaju se problemi s održavanjem i nadogradnjom, kod postaje krhak i može doći do suptilnih razlika u ponašanju programa ovisno o tome koja se kopija koda izvede ako smo prilikom promjena zaboravili ažurirati sve duplikate koda. Stoga treba zapamtiti — *DRY*.

Konačno, prethodna situacija opisana kao motivacijski primjer za načelo *KISS* ujedno je i primjer za načelo *YAGNI* — nemojte pisati kod koji Vam neće trebati. Kad Vam zatreba i ako Vam

[1] http://www.codinghorror.com/blog/2007/03/curlys-law-do-one-thing.html

zatreba, napisat ćete ga. Ako ga krenete pisati odmah, ne znate hoće li Vam trebati, hoće li Vam trebati baš u tom obliku ili u nekoj modifikaciji i badava ćete trošiti vrijeme koje možete iskoristiti bolje. Primjer ovog pristupa je razvoj programskog proizvoda tjeran testovima (engl. *Test Driven Development*) čija je osnovna mantra trodjelni ciklus: "napiši test" - "uvjeri se da je puknuo" - "napravi minimalnu izmjenu koda uz koju će taj test proći". Dakako, testovi se pri tome pišu samo za funkcionalnosti koje ste naručitelju obećali isporučiti. Posljedica je kod koji je testiran i koji sadrži samo onu funkcionalnost koja je stvarno potrebna jer ostalu nikada niste niti pisali.

Istražujući dalje, mogli bismo nabrojati još čitav niz drugih načela koja će pomoći da se dobije kod dobrih svojstava. U okviru ovog poglavlja izdvojit ćemo, međutim, još samo jedan skup načela: *SOLID*.

Na čvrstim temeljima

U knjizi "Agile Software Development: Principles, Patterns, and Practices" autor Robert Cecil Martin izdvojio je pet načela koja pomažu u izradi koda koji minimizira svojstva otpornosti na promjene, krhkosti te nepokretnosti. Zajedničko ime za tih pet načela je *SOLID* — kratica od početnih slova naziva svakog od pet načela koja su pobrojana u nastavku.

- Načelo jedinstvene odgovornosti (engl. *Single Responsibility Principle*)
- Načelo otvorenosti za nadogradnju a zatvorenosti za promjene (engl. *Open/Closed Principle*)
- Liskovino načelo supstitucije (engl. *Liskov Supstitution Principle*)
- Načelo izdvajanja sučelja (engl. *Interface Segregation Principle*)
- Načelo inverzije ovisnosti (engl. *Dependency Inversion Principle*)

Za svako od ovih načela, u nastavku ćemo dati detaljan primjer koji će pojasniti motivaciju i primjenu načela. No prije no što krenemo s detaljnim primjerima, dat ćemo kratak osvrt na svako od načela.

Načelo jedinstvene odgovornosti govori da svaki razred treba imati samo jednu odgovornost. Alternativno, mogli bismo reći da za svaki razred postoji samo jedan razlog iz kojeg taj razred treba mijenjati. Ako uočimo da postoje dva ili više razloga uslijed kojih se mogu zahtjevati promjene u razredu, tada taj razred treba razložiti na više razreda jer je očito da razred ima više odgovornosti. Problem s razredima koji imaju više odgovornosti jest povezanost njihovog koda: ako krenemo mijenjati kod razreda zbog zahtjevane promjene vezane uz jednu njegovu odgovornost, moguće je da će ta promjena imati utjecaja i na njegovu drugu odgovornost, a to je loše. Što razred ima više odgovornosti, veća je šansa da ćemo prilikom modificiranja tog razreda unijeti pogreške. Pridržavanjem načela jedinstvene odgovornosti ta se vjerojatnost minimizira. Ovo načelo ne propisuje da razred mora imati samo jednu metodu — to bi doista bilo besmisleno i suprotno duhu objektno-orijentiranog oblikovanja. Ono što načelo govori jest da razred mora imati jednu odgovornost te da sve njegove metode moraju biti konceptualno prilagođene pružanju te odgovornosti. Načelo je izvorno definirao Tom DeMarco u knjizi "Structured Analysis and Systems Specification" iz 1979. godine.

Načelo otvorenosti za nadogradnju a zatvorenosti za promjene govori nam da je razrede potrebno oblikovati na način da jedini razlog za njihovo modificiranje bude ispravljanje pogrešaka. Razred koji je jednom napisan i o kojem ovise drugi razredi trebao bi se tretirati zatvorenim za promjene. S druge pak strane, nadogradnju funkcionalnosti treba biti moguće postići nasljeđivanjem. Drugim riječima, kod treba biti takav da je moguće ostvariti nadogradnju (*otvoren za nadogradnju*) ali bez unošenja promjena u postojeće razrede o kojima već ovise i drugi klijenti (*zatvoren za promjene*).

Lijep primjer primjene ovog načela jest definiranje apstraktnih razreda koji definiraju generičke algoritme koji se ostvaruju uporabom apstraktnih metoda, te potom definiranje konkretnih

razreda koji nasljeđuju ove apstraktne razrede i donose implementacije apstraktnih metoda. Postoje i oblikovni obrasci koji su rezultat izravne primjene načela otvorenosti za nadogradnju a zatvorenosti za promjene: oblikovni obrazac *okvirna metoda* (engl. *template method pattern*) te oblikovni obrazac *strategija* (engl. *strategy pattern*).

Liskovino načelo supstitucije govori nam da se prilikom nasljeđivanja izvedeni razredi moraju moći koristiti na svim mjestima na kojima su se koristili bazni razredi. Drugim riječima, ako se postojećim klijentima umjesto primjeraka baznih razreda pošalju primjerci izvedenih razreda, klijenti i dalje moraju funkcionirati ispravno. Ovo možemo reći i drugačije: klijenti moraju moći raditi s izvedenim razreda bez da su svjesni da rade s njima; klijenta ne bi smjelo biti briga radi li s baznim razredom ili s izvedenim razredom (i ako je izveden, kako se točno zove). Načelo je uvela Barbara Liskov 1987. godine na konferenciji *Conference on Object Oriented Programming Systems, Languages and Applications* u okviru pozvanog predavanja pod nazivom *Data Abstraction and Hierarchy*.

Načelo razdvajanja sučelja govori da se klijente ne smije tjerati da ovise o sučeljima koja ne koriste. Ovo načelo predstavlja vodilju prilikom oblikovanja sučelja, kada razmišljamo o tome koje metode i koje funkcionalnosti pripadaju u sučelje. Načelo govori da sučelje mora biti takvo da omogućava obavljanje jednog konceptualnog zadatka i ništa više od toga. Lijep primjer ovog principa je sučelje `java.lang.Iterable` odnosno sučelje `java.util.Iterator` koja smo već spomenuli. Da bi petlja `for` mogla obilaziti po proizvoljnom skupnom objektu, treba imati način kako dohvatiti objekt koji će služiti za iteriranje. Sučelje `java.lang.Iterable` rješava upravo taj problem i ništa više. Sučelje `java.util.Iterator` već je konceptualno malo prljavije jer omogućava dvije stvari: i ostvarivanje iteriranja i brisanje elemenata, no opet, motivaciju odnosno opravdanje takve odluke dali smo uz opis oblikovnog obrasca *Iterator*. Problem s velikim sučeljima jest što svi razredi koji ih implementiraju moraju implementirati sve metode tog sučelja. Klijenti koji koriste objekte preko tih sučelja rijetko pak koriste sve metode sučelja već funcionalnost obavljaju oslanjajući se na samo manji broj metoda. Prilikom promjene takvog sučelja sve je razrede koji ga implementiraju potrebno promijeniti a moguće je i da svi klijenti postanu svjesni promjena u sučelju što je loše. Evo jednostavnog primjera kršenja ovog načela. Dizajniramo sučelje `Transporter` s dvije metode: `transport()` i `changeTires()`. Potom definiramo nekoliko razreda koji implementiraju ovo sučelje, poput *Automobila*, *Kombija* i *Kamiona*. Što međutim napraviti kada poželimo dodati razred *Gusjeničar* koji nema gume? Drugi primjeri kršenja ovog načela su sučelja koja enkapsuliraju raznorodne funkcionalnosti (sučelja tipa: "kuha, pere i pegla, sve u jednom").

Konačno, *načelo inverzije ovisnosti* govori nam da moduli više razine ne bi smjeli ovisiti o modulima niže razine. Apstrakcije ne bi smjele ovisi o detaljima već upravo suprotno - detalji bi trebali ovisiti o apstrakcijama. Posljedica primjene ovog principa je skraćivanje lanaca ovisnosti i postizanje labavije povezanog sustava. Umjesto da razred `A` koji treba neku funkcionalnost direktno stvara primjerak razreda `B` koji nudi tu funkcionalnost i potom čuva referencu na taj primjerak, ovo načelo govori da je bolje funkcionalnost razreda `B` definirati kroz apstraktno sučelje `BB` koje potom implementira razred `B`. Razred `A` trebao bi ovisiti samo o apstraktnom razredu `BB` tako da bilo kakva promjena u implementaciji razreda `B` nema nikakve posljedice na razred `A`. Kako bi mogao odraditi svoj posao, razred `A` bi trebao ili preko konstruktora ili preko *setter*-metode omogućiti da mu netko izvana dostavi referencu na objekt koji će koristiti za dobivanje potrebne usluge. Posljedica je da vanjski klijent može odlučiti koji će razred koristiti za nuđenje potrebne funkcionalnosti (razred `B` ili neki drugi koji također implementira zadano apstraktno sučelje) i primjerak tog razreda može proslijediti razredu `A`.

Načelo se zove "načelo inverzije ovisnosti" jer ovisnosti okreće naopačke: umjesto da modul više razine (u našem primjeru razred `A`) ovisi o modulu niže razine (u našem primjeru razred `B`), postigli smo da oba razreda, `A` i `B` ovise samo o apstraktnom razredu odnosno sučelju `BB`). Već je i sama Java lijep primjer uporabe ovog načela. Tako primjerice, umjesto da imamo posebne algoritme sortiranja koji rade nad konkretnim listama, liste (u smislu kolekcija) su specificirane apstraktnim sučeljem koje potom ima više implementacija. Tako algoritam sortiranja ovisi samo o sučelju koje definira listu, a vanjski korisnik može predati bilo koji primjerak razreda koji implementira ovo sučelje i postupak sortiranja će korektno odraditi svoj posao. Sličnih primjera ima još mnoštvo.

Načelo jedinstvene odgovornosti

Načelo jedinstvene odgovornosti govori da razred (ili metoda) ne smije imati više od jedne odgovornosti. Pojam odgovornost pri tome se često poistovjećuje s *razlogom za promjenu*. Primjenu načela najprije ćemo istražiti na primjeru razreda čija je zadaća ispis izvještaja s ocjenama studenata.

Primjer na razini metode

Pogledajmo primjer razreda koji implementira generiranje jednostavnog izvještaja na konzolu. Kod je dan u ispisu 8.1.

Primjer 8.1. Generiranje izvještaja, prvi pokušaj

```
 1 package hr.fer.zemris.principles.srp;
 2
 3 import java.text.SimpleDateFormat;
 4 import java.util.Date;
 5
 6 public class IspisIzvjestaja {
 7
 8  public void ispisi() {
 9   String formatDatuma = "yyyy-MM-dd HH:mm";
10   SimpleDateFormat sdf = new SimpleDateFormat(formatDatuma);
11   Student[] zapisi = Zapisi.svi;
12   System.out.println("Izvještaj ocjena na kolegiju");
13   System.out.println("Ispisano: "+ sdf.format(new Date()));
14   System.out.println("-----------------------------------");
15   for(Student s : zapisi) {
16    System.out.format(
17     "Student: %s %s %s\n",
18     s.getJMBAG(), s.getPrezime(), s.getIme()
19    );
20    System.out.format("   Ocjena: %d\n", s.getOcjena());
21   }
22   System.out.println("-----------------------------------");
23   System.out.format(
24    "Ukupno ispisano zapisa: %d\n", zapisi.length
25   );
26  }
27
28 }
```

Primjer ispisa dobivenog metodom `ispisi()` prikazan je u nastavku.

```
Izvještaj ocjena na kolegiju
Ispisano: 2013-04-01 18:21
-----------------------------------
Student: 0012345678 Ivić Ivana
   Ocjena: 5
Student: 0023456789 Marić Marko
   Ocjena: 5
Student: 0034567890 Petrić Jasmina
   Ocjena: 5
Student: 0045678901 Antić Stjepan
   Ocjena: 5
-----------------------------------
```

```
Ukupno ispisano zapisa: 4
```

Pogledajmo sada malo bolje napisani kod. Koji bi mogli biti razlozi za njegovu promjenu? Krenimo od vrha izvještaja element po element.

- Želimo promijeniti izvor zapisa o studentima.
- Želimo promijeniti informacije koje ispisujemo za svakog studenta (primjerice, samo JMBAG).
- Želimo promijeniti općeniti format ispisa izvještaja.
- Želimo promijeniti naslov izvještaja.
- Želimo promijeniti format prema kojem je ispisan datum.
- Želimo promijeniti način ispisa separatora.
- Želimo promijeniti kraj izvještaja.

Situacija jasno ilustrira da postoji mnoštvo razloga koji mogu tražiti modifikacije u kodu. Kako popraviti prikazani kod? Pogledajmo rješenje koje prikazuje ispis 8.2.

Primjer 8.2. Generiranje izvještaja, drugi pokušaj

```
 1 package hr.fer.zemris.principles.srp;
 2
 3 import java.text.SimpleDateFormat;
 4 import java.util.Date;
 5
 6 public class IspisIzvjestaja {
 7
 8  private Student[] zapisi;
 9  private SimpleDateFormat sdf;
10
11  public void ispisi() {
12   sdf = new SimpleDateFormat(odrediFormatDatuma());
13   zapisi = dohvatiZapise();
14   ispisiZaglavlje();
15   ispisiSeparator();
16   ispisiZapise();
17   ispisiSeparator();
18   ispisiKraj();
19  }
20
21  private String odrediFormatDatuma() {
22   return "yyyy-MM-dd HH:mm";
23  }
24
25  private Student[] dohvatiZapise() {
26   return Zapisi.svi;
27  }
28
29  private void ispisiZaglavlje() {
30   System.out.println("Izvještaj ocjena na kolegiju");
31   System.out.println("Ispisano: "+ sdf.format(new Date()));
32  }
33
34  private void ispisiSeparator() {
```

```
35   System.out.println("-----------------------------------");
36  }
37
38  private void ispisiZapise() {
39   for(Student s : zapisi) {
40    System.out.format(
41     "Student: %s %s %s\n",
42     s.getJMBAG(), s.getPrezime(), s.getIme()
43    );
44    System.out.format("   Ocjena: %d\n", s.getOcjena());
45   }
46  }
47
48  private void ispisiKraj() {
49   System.out.format(
50     "Ukupno ispisano zapisa: %d\n", zapisi.length
51    );
52  }
53 }
```

Prikazano rješenje je bolje. Razred `IspisIzvestaja` sada se sastoji od dvije vrste metoda. Metoda `ispisi()` je metoda koja se ne brine o implementacijskim detaljima ispisa izvještaja. Ta metoda definira *algoritam* — postupak koji govori od kojih se dijelova sastoji izvještaj odnosno što znači ispisati izvještaj: treba ispisati zaglavlje, treba ispisati separator, potom treba ispisati sve zapise, potom ponovno separator i na kraju treba ispisati dno izvještaja.

Implementacijski detalji svakog od tih poslova riješeni su u zasebnim metodama. Vratimo li se sada prethodno dani popis razloga za promjene, brzo ćemo uočiti da je za svako od pitanja dovoljno promijeniti samo jednu od metoda te da niti jedna metoda nije odgovor na više od jednog pitanja. Svaka metoda sada ima jasno definiranu (jednu) funkcionalnost odnosno odgovornost.

U kodu prikazanom u primjeru 8.2 ostala je još jedna metoda koja i dalje obavlja nekoliko zadaća istovremeno. Primjerice, ako poželimo promijeniti ispis na način da se ispisuju samo zapisi o studentima s ocjenom vrlo dobar ili većom, koju metodu treba promijeniti? Ako poželimo da se zapisi ispisuju sortirano prema prezimenu, koju metodu treba promijeniti? Kako želimo modificirati informacije koje se ispisuju o svakom studentu, koju metodu treba promijeniti? Odgovor na sva prethodna pitanja je isti: metodu `ispisiZapise()`. Ta metoda trenutno ima odgovornost definiranja sadržaja koji se ispisuje kao i načina na koji se on ispisuje. Da bismo situaciju raščistili do kraja, prikazanu metodu treba razlomiti u dvije: jednu koja definira što se i kojim redosljedom ispisuje te drugu koja definira kako se ispisuje pojedini zapis. Rješenje je prikazano u ispisu 8.3.

Primjer 8.3. Generiranje izvještaja, treći pokušaj

```
 1 package hr.fer.zemris.principles.srp;
 2
 3 import java.text.SimpleDateFormat;
 4 import java.util.Date;
 5
 6 public class IspisIzvjestaja {
 7
 8  private Student[] zapisi;
 9  private SimpleDateFormat sdf;
10
11  public void ispisi() {
12   sdf = new SimpleDateFormat(odrediFormatDatuma());
13   zapisi = dohvatiZapise();
14   ispisiZaglavlje();
```

```
15   ispisiSeparator();
16   ispisiZapise();
17   ispisiSeparator();
18   ispisiKraj();
19  }
20
21  private String odrediFormatDatuma() {
22   return "yyyy-MM-dd HH:mm";
23  }
24
25  private Student[] dohvatiZapise() {
26   return Zapisi.svi;
27  }
28
29  private void ispisiZaglavlje() {
30   System.out.println("Izvještaj ocjena na kolegiju");
31   System.out.println("Ispisano: "+ sdf.format(new Date()));
32  }
33
34  private void ispisiSeparator() {
35   System.out.println("------------------------------------");
36  }
37
38  private void ispisiZapise() {
39   for(Student s : zapisi) {
40    ispisiZapis(s);
41   }
42  }
43
44  private void ispisiZapis(Student s) {
45   System.out.format(
46    "Student: %s %s %s\n",
47    s.getJMBAG(), s.getPrezime(), s.getIme()
48   );
49   System.out.format("   Ocjena: %d\n", s.getOcjena());
50  }
51
52  private void ispisiKraj() {
53   System.out.format(
54     "Ukupno ispisano zapisa: %d\n", zapisi.length
55    );
56  }
57
58 }
```

Primjer na razini razreda

Pogledajmo primjer modeliranja razreda `Student` kakav bi mogao postojati u nekom sustavu koji je namijenjen upravljanju poslovanjem na fakultetu. Razred je prikazan u ispisu 8.4.

Primjer 8.4. Razred `Student`

```
1 package hr.fer.zemris.principles.srp;
2
3 public class Student {
4
5  public double calculateTuition() {
6   ...
```

```
 7    return ...;
 8   }
 9
10   private String reportPassedCourses() {
11    ...
12    return ...;
13   }
14
15   private void save() {
16    ...
17   }
18  }
```

Prikazani kod krši načelo jedinstvene odgovornosti jer sadrži metode koje čine poslovnu logiku (izračun školarine za studenta), metode za generiranje izvještaja s popisom kolegija koje je student u trenutku poziva metode položio te metode za pohranu podataka u bazu podataka. Prikazani razred primjer je složenog razreda koji ima mnoštvo odgovornosti, a time i mnoštvo razloga za promjene. Bolja organizacija koda prikazana je na ispisu 8.5.

Primjer 8.5. Razred `Student` i primjena načela jedinstvene odgovornosti

```
1  package hr.fer.zemris.principles.srp;
2
3  public class Student {
4
5   public double calculateTuition() {
6    ...
7    return ...;
8   }
9  }
```

```
1  package hr.fer.zemris.principles.srp;
2
3  public class PassedCourseReporter {
4
5   public String generateForStudent(Student student) {
6    ...
7    return ...;
8   }
9
10 }
```

```
1  package hr.fer.zemris.principles.srp;
2
3  public class StudentRepository {
4
5   public void save(Student student) {
6    ...
7   }
8  }
```

Prikazano rješenje sastoji se od tri razreda. Originalni razred `Student` zadržao je samo odgovornost za izračun školarine, novi razred `PassedCourseReporter` preuzeo je odgovornost za generiranje izvještaja o položenim kolegijima studenta dok je drugi novi razred `StudentRepository` preuzeo odgovornost za pohranu podataka o studentima u bazu podataka. U slučaju da se bilo koji aspekt sustava sada promijeni (primjerice, novi način izračuna školarine, novi format generiranja izvještaja o položenim kolegijima ili pak uporaba druge baze podataka,

prelazak s relacijske baze na objektnu ili slično), promjene će trebati provesti u samo jednom od novih razreda.

Načelo otvorenosti za nadogradnju a zatvorenosti za promjene

Načelo otvorenosti za nadogradnju a zatvorenosti za promjene pokušava osigurati oblikovanje programskog sustava na način koji minimizira potrebu za promjenom postojećeg koda čime uklanja potrebu za provođenjem korekcija koje u sustav mogu unijeti nove pogreške. Istovremeno, načelo pokušava osigurati generiranje koda koji nije krhak. U objektno orijentiranom oblikovanju temeljni alat ovog načela su apstrakcije, nasljeđivanje i polimorfizam.

Prvi primjer

Načelo ćemo ilustrirati na primjeru razreda čija je zadaća ponuditi metodu za izračun površine geometrijskih likova. Pretpostavimo da u programskom sustavu imamo dvije vrste geometrijskih likova: `Kruznica` i `Pravokutnik`. Razred `Povrsine` sadrži metodu `izracunajSumuPovrsina` koja računa ukupnu sumu površina likova koje dobiva kao argument.

Primjer 8.6. Primjer problematične metode `racunajPovrsinu`

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class Povrsina {
 4
 5  public static double izracunajPovrsinu(Object[] likovi) {
 6   double suma = 0.0;
 7   for(Object lik : likovi) {
 8    if(lik instanceof Kruznica) {
 9     Kruznica k = (Kruznica)lik;
10     suma += k.getR() * k.getR() * Math.PI;
11    } else if(lik instanceof Pravokutnik) {
12     Pravokutnik p = (Pravokutnik)lik;
13     suma += p.getA() * p.getB();
14    }
15   }
16   return suma;
17  }
18
19 }
20
```

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class Kruznica {
 4
 5  private double r;
 6
 7  public Kruznica(double r) {
 8   super();
 9   this.r = r;
10  }
11
12  public double getR() {
13   return r;
14  }
15
```

```
16 }
17
```

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class Pravokutnik {
 4
 5  private double a;
 6  private double b;
 7
 8  public Pravokutnik(double a, double b) {
 9   super();
10   this.a = a;
11   this.b = b;
12  }
13
14  public double getA() {
15   return a;
16  }
17
18  public double getB() {
19   return b;
20  }
21
22 }
23
```

Uz pretpostavku da je naš svijet doista sastavljen od samo dva lika, prethodni kod napisan je korektno. Međutim, već bi i neiskusni programer uvidom u metodu `izracunajPovrsinu` trebao postati oprezan kada vidi unutrašnjost petlje `for` — nedostaje nam posljednji `else`! Naravno, brzo ćemo se opravdati: pa nemoguće je da predani objekt ne bude niti jedan od ona dva koja prepoznajemo. Pa ipak: niti vrijedi naše opravdanje (jer metoda trenutno prima bilo kakve objekte i prevodilac ne može kontrolirati što se to predaje metodi), niti vrijedi činjenica da se programski sustavi ne mijenjaju.

Postavlja se sljedeće pitanje: što će se dogoditi ako u programski sustav moramo dodati još jedan lik, primjerice `Romb`? Je li dovoljno samo napisati novi razred koji će predstavljati taj lik i potom očekivati da će sve raditi korektno? Odgovor je, naravno, nije dovoljno s trenutnom implementacijom. Dodavanjem novog razreda koji predstavlja novi geometrijski lik napravili smo nadogradnju sustava. Ovaj kod očito jest otvoren za nadogradnju. Međutim, da bi sve radilo kako spada, moramo otići i u metodu `izracunajPovrsinu` razreda `Povrsina` i u nju moramo dodati novi dio koda koji će uzimati u obzir da postoje i rombovi čime će metoda poprimiti sljedeći oblik:

```
public static double izracunajPovrsinu(Object[] likovi) {
 double suma = 0.0;
 for(Object lik : likovi) {
  if(lik instanceof Kruznica) {
   Kruznica k = (Kruznica)lik;
   suma += k.getR() * k.getR() * Math.PI;
  } else if(lik instanceof Pravokutnik) {
   Pravokutnik p = (Pravokutnik)lik;
   suma += p.getA() * p.getB();
  } else if(lik instanceof Romb) {
   Romb r = (Romb)lik;
   suma += r.getOsnovica() * r.getVisina();
  }
 }
 return suma;
```

```
}
```

Iz primjera je jasno da napisana metoda nije zatvorena za promjene. Implementiranje nove funkcionalnosti u ovom primjeru zahtjeva pisanje i mijenjanje koda na više od jednog mjesta, a to je loše.

Bolje rješenje prikazano je u nastavku. Umjesto da metoda `izracunajPovrsinu` ovisi o konkretnim geometrijskim likovima, ideja je prepisati metodu tako da ovisi o apstraktnom geometrijskom liku. Stoga je najprije definiran apstraktni razred `Lik` koji nudi metodu `povrsina()` čija je zadaća vratiti površinu dotičnog lika. Potom su razredi svih postojećih likova modificirani tako da nasljeđuju apstraktni razred `Lik` i nude prikladnu implementaciju metode `povrsina()`. Konačno, oslanjajući se na taj apstraktni razred, napisana je nova metoda `izracunajPovrsinu` koja računa sumu površina predanih likova. Trebamo li sada dodati novi lik (primjerice prethodno spomenuti romb), sve što je potrebno napisati jest razred `Romb`. Metodu `izracunajPovrsinu` zbog dodavanja novog lika više ne treba mijenjati — sve će raditi korektno zahvaljujući nasljeđivanju i polimorfizmu. Metoda `izracunajPovrsinu` postala je time otvorena za nadogradnju a zatvorena za promjene.

Primjer 8.7. Bolji primjer metode `racunajPovrsinu`

```
1 package hr.fer.zemris.principles.ocp;
2
3 public interface Lik {
4
5  public double povrsina();
6
7 }
8
```

```
1 package hr.fer.zemris.principles.ocp;
2
3 public class Povrsina {
4
5  public static double izracunajPovrsinu(Lik[] likovi) {
6   double suma = 0.0;
7   for(Lik lik : likovi) {
8    suma += lik.povrsina();
9   }
10  return suma;
11 }
12
13 }
14
```

```
1 package hr.fer.zemris.principles.ocp;
2
3 public class Kruznica implements Lik {
4
5  private double r;
6
7  public Kruznica(double r) {
8   super();
9   this.r = r;
10 }
11
12 public double getR() {
13  return r;
14 }
15
```

```
16  @Override
17  public double povrsina() {
18   return r * r * Math.PI;
19  }
20
21 }
22
```

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class Pravokutnik implements Lik {
 4
 5  private double a;
 6  private double b;
 7
 8  public Pravokutnik(double a, double b) {
 9   super();
10   this.a = a;
11   this.b = b;
12  }
13
14  public double getA() {
15   return a;
16  }
17
18  public double getB() {
19   return b;
20  }
21
22  @Override
23  public double povrsina() {
24   return a * b;
25  }
26 }
27
```

Drugi primjer

Načelo otvorenosti za nadogradnju a zatvorenosti za promjene ilustrirat ćemo na još jednom primjeru. Pretpostavimo da razvijamo programski sustav u kojem je potrebno generirati dnevnik promjena (engl. *log*). Pretpostavimo da u trenutku inicijalnog razvoja programa dopuštamo da se dnevničke poruke zapisuju na jedno od tri moguća odredišta: na konzolu, u datoteku te da se šalju e-mailom. Rješenje je prikazano u nastavku.

Primjer 8.8. Primjer izrade dnevničkog podsustava

```
1 package hr.fer.zemris.principles.ocp;
2
3 public enum LogType {
4  CONSOLE,
5  FILE,
6  MAIL
7 }
8
```

```
1 package hr.fer.zemris.principles.ocp;
2
3 public class Logger {
```

```
 4
 5  public void log(String message, LogType whereToLog) {
 6   switch(whereToLog) {
 7   case CONSOLE:
 8    System.out.println(message);
 9    break;
10   case FILE:
11    // kod koji zapisuje u datoteku
12    break;
13   case MAIL:
14    // kod koji poruku šalje poštom
15    break;
16   }
17  }
18 }
19
```

Razmotrimo li malo bolje ovo rješenje, utvrdit ćemo brzo da to rješenje nije zatvoreno za promjene: pojavi li se potreba za slanjem dnevničkih poruka u bazu podataka, morat ćemo napisati kod koji to zna raditi, i dodatno, morat ćemo promijeniti implementaciju razreda `Logger` tako da bude svjestan novog koda. A i klijenti će nekako morati doznati da postoji i takva mogućnost jer su klijenti ti koji trenutno uz poruku predaju i željeno odredište.

Bolje rješenje prikazano je u kodu nastavku. Grafički prikaz razreda dan je na slici 8.1.

Primjer 8.9. Bolji primjer izrade dnevničkog podsustava

```
1 package hr.fer.zemris.principles.ocp;
2
3 public interface ILoggerImplementation {
4
5  public void log(String message);
6
7 }
8
```

```
1 package hr.fer.zemris.principles.ocp;
2
3 public class ConsoleLogger implements ILoggerImplementation {
4
5  @Override
6  public void log(String message) {
7   System.out.println(message);
8  }
9
10 }
11
```

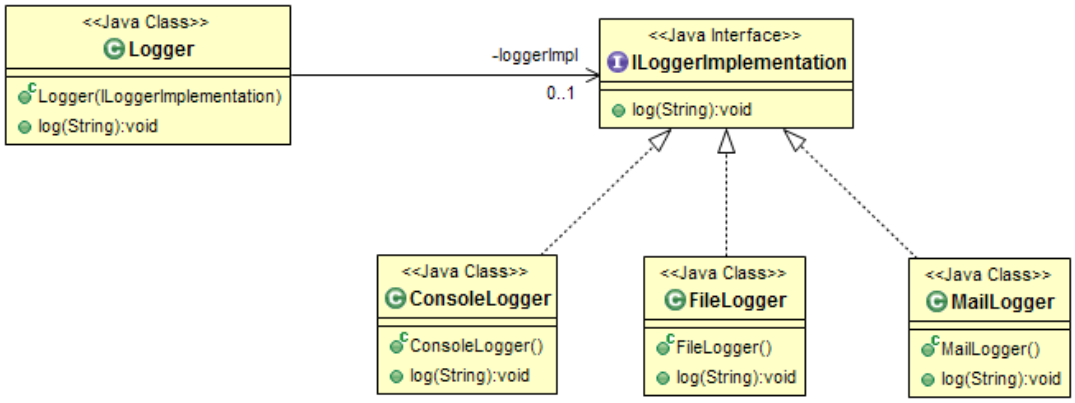
```
1 package hr.fer.zemris.principles.ocp;
2
3 public class FileLogger implements ILoggerImplementation {
4
5  @Override
6  public void log(String message) {
7   // kod koji zapisuje u datoteku
8  }
9
10 }
11
```

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class MailLogger implements ILoggerImplementation {
 4
 5  @Override
 6  public void log(String message) {
 7   // kod koji poruku šalje poštom
 8  }
 9
10 }
11
```

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class Logger {
 4
 5  private ILoggerImplementation loggerImpl;
 6
 7  public Logger(ILoggerImplementation loggerImpl) {
 8   super();
 9   this.loggerImpl = loggerImpl;
10  }
11
12  public void log(String message) {
13   loggerImpl.log(message);
14  }
15
16 }
17
```

Prikazano rješenje kreće s definiranjem potpuno apstraktnog razreda (odnosno sučelja) `ILoggerImplementation`. Namjena ovog sučelja jest osigurati labavu povezivost između klijenata koji koriste usluge i konkretnih implementacija. Temeljem tog sučelja potom su definirana tri različita razreda koja implementiraju sučelje i ostvaruju tri različita odredišta za poruke koje se zapisuju. Konačno, razred `Logger` koji koristi ostatak programa implementiran je tako da mu se prilikom stvaranja injektira neka konkretna implementacija odredišta za poruke koje on pamti i koristi u metodi `log` koju pozivaju njegovi klijenti.

Slika 8.1. Organizacija dnevničkog podsustava



Prikazano rješenje predstavlja kod koji je doista otvoren za nadogradnju a zatvoren za promjene. Primjerice, da bismo dodali mogućnost zapisivanja poruka u bazu podataka, dovoljno je napisati novi razred `DatabaseLogger` koji će implementirati sučelje `ILoggerImplementation`. Ništa osim toga u sustavu ne treba mijenjati. Dapače, sada je lagano dodavati i složenije kombinacije odredišta za poruke. Primjerice, možemo zamisliti implementaciju koja po-

ruke istovremeno šalje i putem e-maila, i koje zapisuje u datoteku. Kako bi izgledala takva implementacija? Takvo ponašanje moguće je dobili oslanjajući se na postojeće implementacije! Evo primjera u nastavku.

Primjer 8.10. Višestruko zapisivanje dnevničkih poruka

```
 1 package hr.fer.zemris.principles.ocp;
 2
 3 public class DualLogger implements ILoggerImplementation {
 4
 5  private ILoggerImplementation logger1;
 6  private ILoggerImplementation logger2;
 7
 8  public DualLogger(ILoggerImplementation logger1,
 9    ILoggerImplementation logger2) {
10   super();
11   this.logger1 = logger1;
12   this.logger2 = logger2;
13  }
14
15  @Override
16  public void log(String message) {
17   logger1.log(message);
18   logger2.log(message);
19  }
20
21 }
22
```

Korištenjem navedenog razreda primjerak razreda `Logger` moguće je inicijalizirati tako da poruke efektivno šalje na dva odredišta, kako to ilustrira kod prikazan u nastavku.

```
ILoggerImplementation dual = new DualLogger(
          new ConsoleLogger(), new MailLogger()
  );

Logger logger = new Logger(dual);

logger.log("Ovo je prva poruka.");

logger.log("Ovo je druga poruka.");
```

Liskovino načelo supstitucije

Liskovino načelo supstitucije formalno je definirala Barbara Liskov, na sljedeći način.

> Neka je *q(x)* svojstvo koje je dokazivo o objektu *x* koji je primjerak razreda *T*. Tada to isto svojstvo *q(y)* treba biti dokazivo za objekt *y* koji je primjerak razreda *S* gdje je razred *S* podtip od razreda *T*.

Ovako napisano, ovo načelo najčešće zvuči u potpunosti nerazumljivo. Što li se samo htjelo time reći? No nije tako strašno — ovo načelo izuzetno je važno ako razmišljamo o nadogradivosti koda, i zapravo kaže sljedeće: *stari kod mora moći raditi s novim kodom*. Ništa složenije od toga i sasvim razuman zahtjev. Prisjetimo se istovremeno i načela *otvorenosti za nadogradnju a zatvorenosti za promjene*: postizanje nadogradnje moguće je rješavati nasljeđivanjem i oslanjanjem na polimorfizam. Stari kod, umjesto da radi s nekom konkretnom implementacijom, obično je napisan tako da radi s nekim apstraktnim razredom (ili sučeljem). U trenutku poziva, tom se kodu predaje referenca na objekt koji predstavlja primjerak razreda koji implementira

to sučelje. Nadogradnja koda sada se postiže na način da se u kod doda još jedna, nova, implementacija istog sučelja koja obavlja neki drugi zadatak. Stari kod sada može besprijekorno raditi i s tom novom implementacijom. Nadogradnja je time uspjela. Ili nije?

Liskovino načelo supstitucije upravo nam govori nešto o načinu na koji treba pisati nadogradnju kako bismo postigli besprijekornu integraciju s postojećim kodom. Postojeći kod, kao što mu ime kaže, već je napisan i postoji. Postojeći kod prilikom korištenja starog koda od tog koda očekuje odgovarajuće ponašanje. Da bi nadogradnja novim kodom uspjela, novi kod ne smije prekršiti ta očekivanja. Evo dva jednostavna primjera.

Geometrijski likovi

Primjer 8.11. Geometrijski podsustav

```
1 package hr.fer.zemris.principles.liskov;
2
3 public class Pravokutnik {
4
5  protected int x;
6  protected int y;
7  protected int w;
8  protected int h;
9
10  public Pravokutnik(int x, int y, int w, int h) {
11   super();
12   this.x = x;
13   this.y = y;
14   this.w = w;
15   this.h = h;
16  }
17
18  public int getX() {
19   return x;
20  }
21  public void setX(int x) {
22   this.x = x;
23  }
24
25  public int getY() {
26   return y;
27  }
28  public void setY(int y) {
29   this.y = y;
30  }
31
32  public int getW() {
33   return w;
34  }
35  public void setW(int w) {
36   this.w = w;
37  }
38
39  public int getH() {
40   return h;
41  }
42  public void setH(int h) {
43   this.h = h;
44  }
```

```
45
46  public double racunajPovrsinu() {
47   return w*h;
48  }
49 }
50
```

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Klijent {
 4
 5  public static void podesi(Pravokutnik p) {
 6   p.setW(10);
 7   p.setH(20);
 8
 9   double povrsina = p.racunajPovrsinu();
10
11   if(Math.abs(povrsina - 200.0) > 1E-6) {
12    // Ups! Ovo se nije smjelo dogoditi!
13    throw new RuntimeException(
14     "Implementacija pravokutnika je strgana!"
15    );
16   }
17  }
18
19 }
20
```

Ispis 8.11 sadrži dva razreda. Razred `Pravokutnik` predstavlja model pravokutnika. Pravokutnik je pri tome zadan `x` i `y` koordinatama gornjeg lijevog vrha te širinom `w` i visinom `h`. Razred za sve članske varijable nudi metode za dohvat i postavljanje, a također nudi i metodu na izračun površine pravokutnika.

Razred `Klijent` prikazuje klijentski kod koji koristi primjerke ovog razreda. Konkretan klijent je u tom ispisu metoda `podesi` koja podešava širinu i visinu predanog pravokutnika i potom provjerava uspješnost usporedbom dohvaćene i očekivane površine (retci 12 do 15). S obzirom da klijentski kod od pravokutnika s punim pravom može očekivati da smije pozvati metodu za postavljanje širine i metodu za postavljanje visine, nakon tih poziva situacija u kojoj površina ne odgovara očekivanoj doista ukazuje na ozbiljan problem u implementaciji, pa stoga i onaj “Ups” komentar.

Primjer uporabe geometrijskog podsustava i razvijenog klijenta dan je u ispisu 8.12.

Primjer 8.12. Primjer uporabe

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Glavni {
 4
 5  public static void main(String[] args) {
 6   Pravokutnik p = new Pravokutnik(1, 1, 15, 8);
 7   Klijent.podesi(p);
 8  }
 9
10 }
11
```

Pretpostavimo sada da se nakon dvije godine uspješne primjene razvijenog programa kroz anketiranje korisnika iskristalizira potreba da se korisnicima omogući i rad s kvadratima. Svi

znamo da je kvadrat “poseban slučaj pravokutnika” — to je pravokutnik koji ima jednaku širinu i visinu. Da, činjenica je da, s obzirom da naš program već podržava pravokutnike automatski podržava i kvadrate. No stvaranje kvadrata preko pravokutnika korisnicima je nepraktično: zašto moraju dva puta predavati isti parametar umjesto da mogu predati samo jedan koji bi se tretirao i kao širina, i kao visina?

U želji da izađemo u susret korisnicima, pišemo novi razred `Kvadrat` kako je to prikazano u ispisu 8.13.

Primjer 8.13. Implementacija kvadrata

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Kvadrat extends Pravokutnik {
 4
 5  public Kvadrat(int x, int y, int w) {
 6   super(x, y, w, w);
 7  }
 8
 9 }
10
```

Razred `Kvadrat` izveli smo iz razreda `Pravokutnik` i pri tome smo prilagodili njegov konstruktor tako da umjesto pozicije lijevog gornjeg vrha, širine i visine prima samo poziciju gornjeg lijevog vrha i veličinu stranice a potom interno tu veličinu stranice predaje nadređenom konstruktoru i kao širinu, i kao visinu. Za sada — sve se čini korektno. Sada korisnik može jednostavnije stvarati kvadrate, i kod prikazan u nastavku je sasvim legalan.

```
Kvadrat k = new Kvadrat(1, 1, 10);
Pravokutnik p = k;
```

Pažljivom analizom ubrzo uočavate da tu ipak postoji problem: što ako korisnik napiše sljedeći isječak koda?

```
Kvadrat k = new Kvadrat(1, 1, 10);
Pravokutnik p = k;
p.setW(15);
p.setH(35);
```

Završit ćemo s kvadratom različitih stranica! A to po definiciji nije kvadrat. Uvođenjem ovako jednostavne modifikacije zapravo smo napravili model podataka koji se može trivijalno dovesti u nekonzistetno stanje. Koji je razlog tome, odnosno zašto je sada stanje nekonzistentno? Razlog leži u definiciji pojma kvadrat: to je lik koji ima sve stranice jednake duljine — nemamo dva stupnja slobode kao što je to bio slučaj kod pravokutnika koji je dozvoljavao nezavisno podešavanje širine i visine.

Naoružani ovom spoznajom, možemo pokušati popraviti izvor nekonzistentnosti: ne smijemo dopustiti da se u kvadratu širina mijenja nezavisno od visine (i obrnuto). Nova implementacija prikazana je u ispisu 8.14.

Primjer 8.14. Popravljena implementacija kvadrata

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Kvadrat extends Pravokutnik {
 4
 5  public Kvadrat(int x, int y, int w) {
 6   super(x, y, w, w);
```

```
 7   }
 8
 9   @Override
10   public void setW(int w) {
11    this.w = w;
12    this.h = w;
13   }
14
15   @Override
16   public void setH(int h) {
17    this.w = h;
18    this.h = h;
19   }
20 }
21
```

Ova nova implementacija problem rješava tako što nadjačava metode za postavljanje širine i visine na način da poziv bilo koje od njih odmah promijeni i širinu i visinu na istu vrijednost čime se više ne može doći u nekonzistentno stanje. Čini se — konačno je sve razriješeno. No zašto smo uopće radili ovo sve? Zato da bi stari kod mogao raditi s našim novim kodom. Modifikacija glavnog programa prikazana je u ispisu 8.15.

Primjer 8.15. Nadogradnja glavnog programa

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Glavni {
 4
 5  public static void main(String[] args) {
 6   Pravokutnik p = new Kvadrat(1, 1, 30);
 7   Klijent.podesi(p);
 8  }
 9
10 }
11
```

Nažalost, čeka nas razočaranje — stari kod odjednom počinje bacati iznimke! Nadogradnjom koda postigli smo rušenje starog koda. To svakako nije bilo željeno ponašanje. Gdje je problem? Stari kod (klijenta) pozvali smo tako da smo mu predali referencu na objekt koji po tipu jest pravokutnik (stvorili smo primjerak kvadrata, no kvadrat nasljeđuje pravokutnik pa kvadrat jest pravokutnik). Što se tiče prevoditelja, sve je korektno. Što se tiče klijenta — to više nije. Temeljni problem koji je nastao jest taj da novi kod krši očekivanja koja je stari kod imao od razreda `Pravokutnik`. To očekivanje je da se širina i visina mogu mijenjati nezavisno. Kako je to bilo u suprotnosti s definicijom kvadrata,u implementaciji smo poduzeli korake koji osiguravaju konzistentnost podataka o kvadratu. No ti koraci su direktno kršenje ponašanja koje pravokutnik obećava svojim klijentima, i zbog toga je došlo do pucanja starog koda koji više ne radi dobro ako mu se umjesto primjerka pravokutnika preda primjerak kvadrata.



Kršenje Liskovina načela supstitucije

Liskovino načelo supstitucije krši se kada novi kod ne poštuje očekivano ponašanje koje nudi stari kod. Primjeri su razredi `Pravokutnik` i `Kvadrat`, razredi `Elipsa` i `Kruznica` i drugi.

Znači li to da prirodnu vezu između pravokutnika i kvadrata nikada ne smijemo modelirati nasljeđivanjem? Odgovor je ipak malo složeniji. Zašto je došlo do problema? U našem slučaju, zbog toga što je razred `Pravokutnik` dozvoljavao naknadnu promjenu širine i visine. Da su, primjerice, razredi bili nepromjenjivi, odnosno da su primili parametre preko konstruktora i po-

tom nudili samo metode za dohvat istih, problema ne bi bilo i mogli bismo ih povezati nasljeđivanjem.

Prisjetimo se i zašto nam je nasljeđivanje važno. Nasljeđivanje pospješuje višestruku iskoristivost koda. Kada razmatramo slučaj pravokutnika i kvadrata, umjesto da u svaki dodajemo kopiju metode koja računa površinu, kopiju metode koja računa opseg, kopiju metode koja pomiče lik, kopiju metode koja crta lik, kopiju metode koja računa presjek tog lika s nekim drugim i slično, sasvim je razumno takav kod napisati na najvišem mogućem mjestu — u razredu pravokutnik, i potom dopustiti da preko nasljeđivanja razred koji modelira kvadrat koristi taj kod jer je on primjenjiv i na taj razred. S druge pak strane, treba paziti i na konzistentnost modela. Ima li onda rješenja koje će dopustiti da se kod ne duplicira, a da pravokutnici i kvadrati istovremeno budu i promjenjivi? Odgovor je, naravno, ima. Jedna mogućnost je napisati model kako je prikazano u nastavku.

Primjer 8.16. Moguće rješenje

```
1 package hr.fer.zemris.principles.liskov;
2
3 public class ApstraktniPravokutnik {
4
5  protected int x;
6  protected int y;
7  protected int w;
8  protected int h;
9
10  protected ApstraktniPravokutnik(int x, int y, int w, int h) {
11   super();
12   this.x = x;
13   this.y = y;
14   this.w = w;
15   this.h = h;
16  }
17
18  public int getX() {
19   return x;
20  }
21  public void setX(int x) {
22   this.x = x;
23  }
24
25  public int getY() {
26   return y;
27  }
28  public void setY(int y) {
29   this.y = y;
30  }
31
32  public int getW() {
33   return w;
34  }
35  public void setW(int w) {
36   this.w = w;
37  }
38
39  public int getH() {
40   return h;
41  }
42  public void setH(int h) {
```

```
43    this.h = h;
44   }
45
46  }
47
```

```
 1  package hr.fer.zemris.principles.liskov;
 2
 3  public class Pravokutnik extends ApstraktniPravokutnik {
 4
 5   public Pravokutnik(int x, int y, int w, int h) {
 6    super(x, y, w, h);
 7   }
 8
 9  }
10
```

```
 1  package hr.fer.zemris.principles.liskov;
 2
 3  public class Kvadrat extends ApstraktniPravokutnik {
 4
 5   public Kvadrat(int x, int y, int w) {
 6    super(x, y, w, w);
 7   }
 8
 9   @Override
10   public void setW(int w) {
11    this.w = w;
12    this.h = w;
13   }
14
15   @Override
16   public void setH(int h) {
17    this.w = h;
18    this.h = h;
19   }
20
21  }
22
```

Rješenje prikazano u ispisu 8.16 sastoji se od 3 razreda. Razred `ApstraktniPravokutnik` ima zadaću po prvi puta definirati članske varijable koje će čuvati poziciju pravokutnika te njegovu širinu i visinu. Ovaj razred može poslužiti i kao repozitorij za sav dijeljeni kod (površine, opsezi, kod za crtanje, kod za računanje presjeka). U dokumentaciji metoda za postavljanje širine i visine tog razreda, ponašanje treba ostaviti nespecificirano — metode ne smiju biti vezane nikakvih ugovorom na koji bi se klijent potom mogao pozvati (drugim riječima, ne smije postojati "očekivano" ponašanje); u tom smislu, ovisno što želimo, metode za postavljanje širine i visine mogli bismo čak deklarirati i zaštićenima tako da ih nitko izvana ne bi mogao pozivati. Iz tog razreda potom izvodimo razrede `Pravokutnik` i `Kvadrat`, i to na uobičajeni način. Razred `Pravokutnik` prvi je razred koji će obećati da se kod njega širina i visina mogu nezavisno mijenjati. S druge strane, razred `Kvadrat` prvi je razred koji će obećati da se kod njega širina i visina ne mogu mijenjati nezavisno. Kako oni više nisu u direktnom roditeljskom odnosu, više niti jedan od njih ne krši očekivanja (odnosno ugovor) onog drugog i prethodno opisani problemi nestaju.

Drugi primjer

Kršenje Liskovina načela supstitucije pogledat ćemo na još jednom primjeru. Pretpostavimo da radimo aplikaciju koja mora podržavati lokalizaciju poruka koje se ispisuju korisniku. Ovisno

o trenutno odabranom jeziku, korisniku primjerice treba ispisati "Bok", "Hello" ili "Hallo". Jedan od načina kako ugraditi takvu mogućnost u aplikaciju jest da se svakoj poruci pridijeli jedinstven ključ i potom se napravi baza lokalizacija ključeva. Primjerice, neka ključ `K1` označava pozdravnu poruku. Hrvatska lokalizacija tog ključa bi mu pridijelila tekst "Pozdrav", engleska tekst "Hello" i slično. Za svaki ključ ove prijevode možemo držati u datotekama ili pak u bazi podataka.

Pretpostavimo za potrebe ovog primjera da se prijevodi čuvaju u bazi podataka i da se isti u bazi mogu i povremeno mijenjati. Oblikujmo dva razreda koji će nam omogućiti rad s porukama.

Primjer 8.17. Lokalizirane poruke

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class Message {
 4
 5  private String key;
 6  protected String text;
 7
 8  public Message(String key) {
 9   text = // Učitaj iz baze
10  }
11
12  public String getText() {
13   return text;
14  }
15
16  public void update() {
17   // Provjeri u bazi, ažuriraj tekst
18  }
19 }
20
```

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 import java.util.Map;
 4
 5 public class Messages {
 6
 7  // Mapa: ključ => poruka
 8  private Map<String, Message> messages;
 9
10  public Messages() {
11   // inicijaliziraj poruke
12  }
13
14  public void updateMessages() {
15   for(Message m : messages.values()) {
16    m.update();
17   }
18  }
19
20 }
21
```

Razred `Message` predstavlja enkapsulaciju jedne poruke, To je objekt koji čuva vrijednost ključa i pridruženu vrijednost lokaliziranog teksta. Razred ima i metodu `update` koja će ažurirati vrijednost teksta s najnovijim dostupnim prijevodom. Razred `Messages` predstavlja pak

enkapsulaciju svih poruka kojima sustav raspolaže i nudi metodu `update` koja pokreće ažuriranje svih poruka.

Pretpostavimo sada da se pojavila potreba za uvođenjem poruka koje ne podržavaju osvježavanje prijevoda. Rješenje bi moglo biti uvođenje novog razreda kako je to prikazano u nastavku.

Primjer 8.18. Poruke bez mogućnosti ažuriranja

```
1 package hr.fer.zemris.principles.liskov;
2
3 public class PlainMessage extends Message {
4
5  public PlainMessage(String key, String text) {
6   super(key);
7   this.text = text;
8  }
9
10  @Override
11  public void update() {
12   throw new UnsupportedOperationException();
13  }
14
15 }
16
```

Razred u konstruktoru prima vrijednost prijevoda i brani ažuriranje na način da u metodi `update` izazove iznimku. S obzirom da sam razred predstavlja poruku koju nije moguće ažurirati, očekivanje razreda je da se ta operacija ne smije niti zatražiti — stoga odluka o izazivanju iznimke.

Uvođenje ovog razreda nažalost skršit će postojeći kod. Konkretno, skršit će izvođenje metode `update` razreda `Messages` ako se među porukama nađe barem jedan primjerak razreda `PlainMessage`. Primjerci razreda `PlainMessage` ne mogu se supstituirati na mjesto primjeraka `Message` jer razred `PlainMessage` krši očekivanja koja klijenti imaju prema primjercima razreda `Message`.

Proanalizirajmo u čemu se očituje to kršenje. Razred `Message` modelira lokaliziranu poruku koja se može ažurirati. Razred `PlainMessage` to ne čini. Razred `Message` sadrži metodu `update` koja ne izaziva iznimku `UnsupportedOperationException` pa postojeći klijenti ne računaju na njeno izazivanje. Razred `PlainMessage` krši ovo očekivanje i izaziva novu vrstu poruku na koju klijenti ne računaju čime može izazvati njihovo rušenje. Konačno, razred `Message` dolazi s obećenjem da će po pozivu metode `update` prijevod biti ažuriran, što predstavlja ugovor između tog razreda i njegovih klijenata; razred `PlainMessage` krši ovo očekivanje.

Kako bismo onda mogli riješiti ovaj problem? Jedan mogući pristup jest postaviti stvari naglavačke: postoje poruke koje imaju pridruženi prijevod za ključ koji enkapsuliraju. Potom postoji posebna podvrsta poruka — onih koje se znaju ažurirati. Ovo rješenje prikazano je u nastavku.

Primjer 8.19. Poruke bez mogućnosti ažuriranja

```
1 package hr.fer.zemris.principles.liskov;
2
3 public class Message {
4
5  private String key;
6  protected String text;
7
```

```
 8  public Message(String key) {
 9   text = null; // Učitaj iz baze
10  }
11
12  public String getText() {
13   return text;
14  }
15
16 }
17
```

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 public class UpdateableMessage extends Message {
 4
 5  public UpdateableMessage(String key) {
 6   super(key);
 7  }
 8
 9  public void update() {
10   // Provjeri u bazi, ažuriraj prijevode
11  }
12
13 }
14
```

```
 1 package hr.fer.zemris.principles.liskov;
 2
 3 import java.util.List;
 4 import java.util.Map;
 5
 6 public class Messages {
 7
 8  // Mapa: ključ => poruka
 9  private Map<String, Message> messages;
10  // Lista poruka koje se može ažurirati
11  private List<UpdateableMessage> updateableMessages;
12
13  public Messages() {
14   // inicijaliziraj poruke
15  }
16
17  public void updateMessages() {
18   for(UpdateableMessage m : updateableMessages) {
19    m.update();
20   }
21  }
22
23 }
24
```

Kod koji smo dobili na ovaj način u skladu je s Liskovinim načelom supstitucije. Razred `Messages` sada održava reference na dvije kolekcije. Prva kolekcija je i dalje kolekcija koja preslikava ključeve u poruke. Druga kolekcija je lista koja sadrži podskup poruka iz prve kolekcije — sadrži samo onaj podskup poruka koje je moguće ažurirati. Razred `Message` je pri tome razred koji nudi osnovnu funkcionalnost: povezuje ključ sa prijevodom. Razred `UpdateableMessage` na tu funkcionalnost nadodaje novu funkcionalnost — prijevod se po potrebi može ažurirati. Izvođenjem iz razreda `Message` razred `UpdateableMessage` ne narušava niti jedno očekivanje koje klijenti imaju od osnovnog razreda a ujedno nadodaje novu funkcionalnost.

Uzroci kršenja očekivanog ponašanja

Kada novi kod krši očekivano ponašanje starog koda, to može napraviti na nekoliko različitih načina. Osvrnimo se na primjer određene metode `m` koja se u nadogradnji zamjenjuje novom implementacijom. Neka je prototip takve funkcije:

```
double m(double x);
```

1. *Kršenje preduvjeta koji moraju biti zadovoljeni pri pozivu funkcije* (engl. *preconditions*).

   Funkcija `m` može očekivati da kao vrijednost arugumenata ne može dobiti bilo kakvu vrijednost već vrijednost koja zadovoljava određena ograničenja. Primjerice, neka funkcija `m` rezultat generira vađenjem drugog korijena iz `x`:

   ```
   double m(double x) {
     return sqrt(x);
   }
   ```

   U tom slučaju implicitan preduvijet koji metoda `m` postavlja je da `x` mora biti nenegativan. Imajući to u vidu, moguće je da postoji mnoštvo staroga koda koji koristi funkciju `m` i koji poštuje taj zahtjev. Ako bismo sada implementaciju funkcije htjeli zamijeniti novom implementacijom `m'`, da bismo imali garanciju da će sav postojeći kod raditi besprijekorno s novom implementacijom nužno je osigurati da funkcija `m'` također kao vrijednost argumenta `x` dozvoljava sve vrijednosti koje su barem nenegativne. Uvjeti koji na argumente postavlja funkcija `m'` moraju biti manje ili u najgorem slučaju jednako restriktivni u odnosu na uvjete koje postavlja postojeća funkcija. Ako su uvjeti restriktivniji, moguće je da će stari kod pozvati tu implementaciju s vrijednošću argumenta `x` koji će skršiti novu implementaciju a time i stari kod.

2. *Kršenje obećanja koje daje funkcija na kraju izvođenja* (engl. *postconditions*).

   Pretpostavimo da je funkcija `m` bila implementirana na sljedeći način.

   ```
   double m(double x) {
     return sqrt(x)-10;
   }
   ```

   Svi postojeći klijenti te funkcije očekivali su da će funkcija vraćati brojeve čija je vrijednost veća ili jednaka od -10. S obzirom da postojeći klijenti rade dobro, za pretpostaviti je da je to raspon rezultata s kojima se klijenti znaju nositi. Da bismo imali garanciju da nova implementacija neće skršiti postojeće klijente, nužno je osigurati da ona na povratne vrijednosti postavlja uvjete koji su barem toliko restriktivni kao što su bili i stari uvjeti, a mogu biti i restriktivniji — ne smiju biti blaži. Evo primjera nove funkcije koja ovo krši:

   ```
   double m(double x) {
     return sqrt(x)-11;
   }
   ```

   Uz ovakvu implementaciju, moguće je da funkcija vrati rezultat koji je prema prethodnoj definiciji funkcije bio nemoguć: -11. Ako takva mogućnost postoji, sasvim je moguće da se postojeći klijenti neće znati nositi s tom vrijednosti i da bi moglo doći do pucanja postojećeg klijentskog koda.

3. *Kršenje pretpostavka koje se ne bi smjele mijenjati* (engl. *invariants*).

   Prilikom implementacije bilo koje metode, često je moguće otkriti svojstva za koja metoda garantira da ih svojim izvođenjem neće narušiti. Ta svojstva nazivamo *invarijante*. Primjerice, zamislimo razred koji interno čuva pristupnu točku (engl. *socket*) preko kojeg se spaja na mrežnu uslugu i ima metodu kojom dohvaća stanje burzovnih indeksa. Invarijanta te metode može biti tvrdnja da pristupna točka prije poziva metode mora biti u zatvorenom stanju te da

će i nakon poziva te metode ostati u zatvorenom stanju. Tijekom izvođenja same metode, metoda će preko te pristupne točke uspostaviti spoj na mrežnu uslugu, dohvatiti podatke i raskinuti spoj. Svaka metoda koja će biti zamjena ove metode mora garantirati sve invarijante koje garantira i stara implementacija.

4. *Kršenje pretpostavki o mogućim iznimkama*.

   Stara implementacija metode može pod određenim uvjetima izazvati određenu iznimku. O kojim se iznimkama radi, trebalo bi biti dobro dokumentirano uz staru implementaciju (ili uz specifikaciju ako govorimo o sučelju). Nova implementacija ne bi smjela izazivati druge vrste iznimaka jer ih postojeći klijenti možda ne hvataju (a svakako ih ne očekuju). To bi pak moglo dovesti do kršenja postojećeg koda.

Načelo izdvanja sučelja

Prilikom oblikovanja pojedinih razreda nije rijetkost da se u razred krene dodavati sve više i više funkcionalnosti. Problem nastupa u trenutku kada ta funkcionalnost postane nekohezivna, odnosno kada različitim klijentima tog razreda treba različit i disjunktan podskup metoda koje razred nudi kako bi implementirali željenu funkcionalnost. Kod takvog dizajna, iako niti jedan klijent ne treba sve deklarirane funkcije, on ih je svjestan. Negativna posljedica je da se prilikom promjene prototipa bio koje od tih funkcija ili prilikom dodavanja novih funkcija *svi* postojeći klijenti moraju nanovo prevoditi.

Načelo izdvajanja sučelja govori nam *da se klijente ne smije siliti da ovise o sučelju koje ne koriste*. Imamo li takvu situaciju u kodu, nju se obično dade lagano razriješiti — treba proanalizirati koje metode koriste koji klijenti i njih treba definirati kao apstraktna sučelja. Potom, ako naš razred implementira sve te metode, definira ga se kao razred koji implementira i jedno i drugo sučelje. Svaki od klijenata tada s razredom razgovara kroz svoje specijalizirano sučelje i niti na koji način nije svjestan promjena koje se događaju na razini konkretnih razreda ako one nisu vezane direktno za sučelje koje on koristi.

Krenimo najprije s primjerom lošeg koda.

Primjer 8.20. Problematična organizacija koda

```
1 package hr.fer.zemris.principles.srp;
2
3 public class Query {
4
5  public Query(String query) {
6  }
7
8 }
9
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public interface IDatabase {
4  public int executeQuery(Query query);
5  public void updateIndexes();
6  public void compactUnusedSpace();
7  public double calcFragmentationRatio();
8 }
9
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public class ConcreteDatabase implements IDatabase {
4
```

```
 5  public ConcreteDatabase() {
 6  }
 7
 8  @Override
 9  public int executeQuery(Query query) {
10   // izvrši upit
11   return ... // rezultat
12  }
13
14  @Override
15  public void updateIndexes() {
16   // ...
17  }
18
19  @Override
20  public void compactUnusedSpace() {
21   // ...
22  }
23
24  @Override
25  public double calcFragmentationRatio() {
26   // ...
27   return ... // rezultat
28  }
29 }
30
```

```
 1 package hr.fer.zemris.principles.srp;
 2
 3 public class Client {
 4
 5  public Client(IDatabase database) {
 6   Query query = new Query("...");
 7   if(0 == database.executeQuery(query)) {
 8    throw new RuntimeException("Upit nije ništa vratio.");
 9   }
10  }
11
12 }
13
```

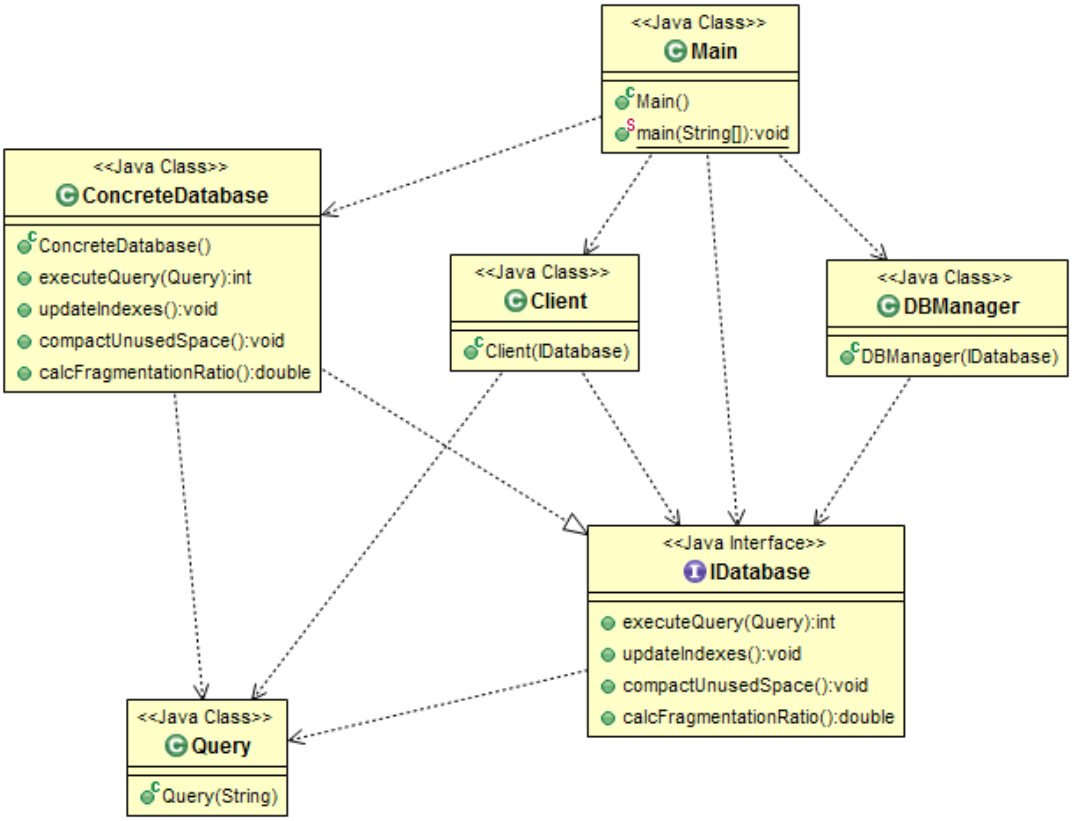
```
 1 package hr.fer.zemris.principles.srp;
 2
 3 public class DBManager {
 4
 5  public DBManager(IDatabase database) {
 6   if(database.calcFragmentationRatio() > 0.4) {
 7    database.compactUnusedSpace();
 8    database.updateIndexes();
 9   }
10  }
11
12 }
13
```

```
 1 package hr.fer.zemris.principles.srp;
 2
 3 public class Main {
 4
```

```
 5  public static void main(String[] args) {
 6   IDatabase db = new ConcreteDatabase();
 7
 8   new Client(db);
 9   new DBManager(db);
10  }
11
12 }
13
```

Organizacija i ovisnosti razreda prikazani su na slici u nastavku. Crtkane linije sa strelicom na kraju označavaju ovisnosti.

Slika 8.2. Organizacija lošeg rješenja



Prikazani kod modelira pristup jednostavnoj bazi podataka preko apstraktnog sučelja. Apstraktna baza podataka modelirana je sučeljem `IDatabase` koje klijentima nudi dvije vrste funkcionalnosti: izvođenje upita nad podatcima (razred `Query` te metoda `executeQuery()`) te izvođenja administracijskih operacija nad bazom (pokretanje defragmentacije prostora koji zauzimaju tablice s podatcima te popravljanje indeksa). Dani primjer prikazuje jednu konkretnu implementaciju baze podataka (razred `ConcreteDatabase`) te dva razreda koja predstavljaju različite klijente te baze. Razred `Client` nad bazom podataka pokreće upite vezane uz podatke a razred `DBManager` predstavlja klijenta koji se bavi održavanjem baze podataka. Konačno, razred `Main` stvara primjerke jednog i drugog klijenta i pušta ih da rade svoj posao.

Prikazano rješenje je loše jer dodavanje ili ažuriranje neke od funkcija koje su namijenjene administriranju baze podataka povlači potrebu ponovnog prevođenja svih klijenata koji nad bazom samo postavljaju upite o podatcima. Isti komentar vrijedi i za obrnuti slučaj.

Kako onda ovo rješenje popraviti? Treba uočiti da imamo dvije vrste klijenata i da nudimo dvije vrste usluga. Svaku od usluga potrebno je modelirati zasebnim sučeljem te modificirati klijente tako da ovise samo o minimalnom sučelju. Rješenje je prikazano u nastavku.

Primjer 8.21. Bolja organizacija koda

```
1 package hr.fer.zemris.principles.srp;
2
3 public class Query {
4
5  public Query(String query) {
6  }
7
8 }
9
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public interface IQueryable {
4  public int executeQuery(Query query);
5 }
6
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public interface IManageable {
4  public void updateIndexes();
5  public void compactUnusedSpace();
6  public double calcFragmentationRatio();
7 }
8
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public class ConcreteDatabase implements IQueryable, IManageable {
4
5  public ConcreteDatabase() {
6  }
7
8  @Override
9  public int executeQuery(Query query) {
10  // izvrši upit
11  return ... // vrati rezultat
12  }
13
14  @Override
15  public void updateIndexes() {
16  // ...
17  }
18
19  @Override
20  public void compactUnusedSpace() {
21  // ...
22  }
23
24  @Override
25  public double calcFragmentationRatio() {
26  // ...
27  return ... // vrati rezultat
28  }
29 }
30
```

```java
package hr.fer.zemris.principles.srp;

public class Client {

 public Client(IQueryable database) {
  Query query = new Query("...");
  if(0 == database.executeQuery(query)) {
   throw new RuntimeException("Upit nije ništa vratio.");
  }
 }

}
```

```java
package hr.fer.zemris.principles.srp;

public class DBManager {

 public DBManager(IManageable database) {
  if(database.calcFragmentationRatio() > 0.4) {
   database.compactUnusedSpace();
   database.updateIndexes();
  }
 }

}
```

```java
package hr.fer.zemris.principles.srp;

public class Main {

 public static void main(String[] args) {
  ConcreteDatabase db = new ConcreteDatabase();

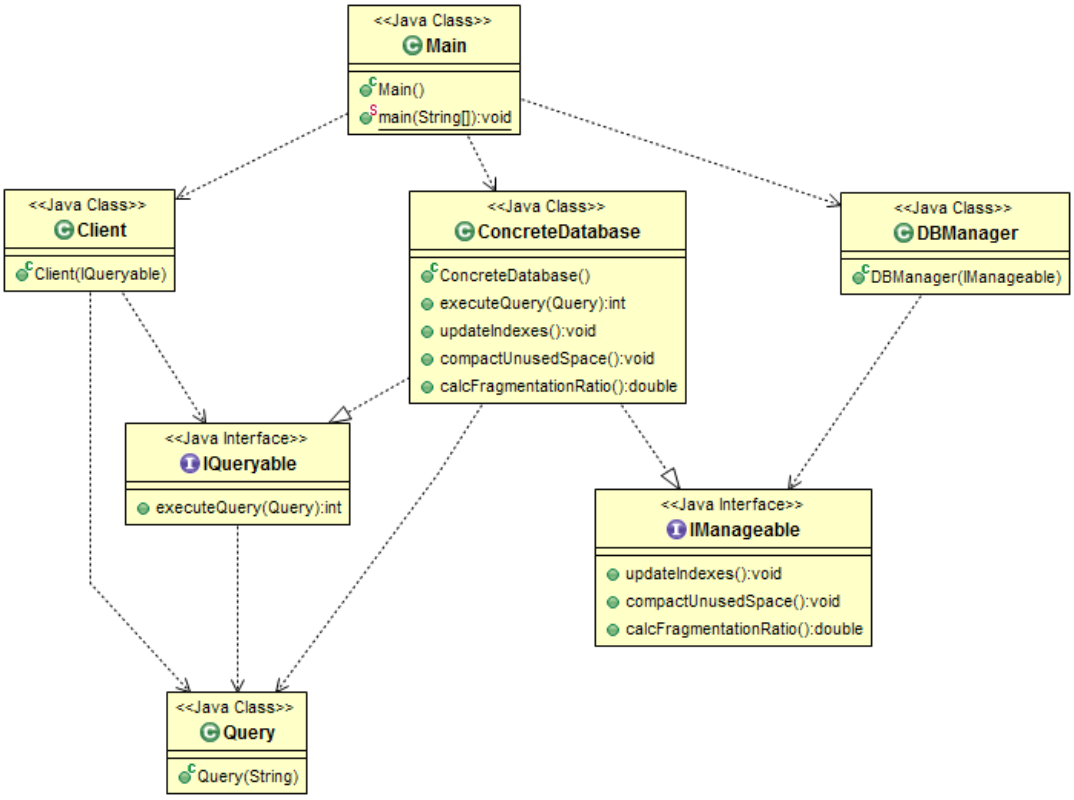
  new Client(db);
  new DBManager(db);
 }

}
```

Umjesto jednog sučelje, sada smo definirali dva sučelja. Sučelje `IQueryable` klijentima omogućava postavljanje upita nad bazom podataka. Sučelje `IManageable` omogućava izvođenje administrativnih poslova nad bazom podataka. U prikazanom rješenju baza podataka je i dalje jedna: njezina implementacija je dana u razredu `ConcreteDatabase`. Razlika je u tome da ovaj razred sada implementira oba sučelja. Zahvaljujući tome, svakome od klijenata možemo predati isti objekt kao bazu s kojom rade a klijenti pri tome ne moraju biti svjesni svih funkcionalnosti koji nudi predani objekt, već mogu s objektom raditi kroz njima dovoljno sučelje.

Organizacija i ovisnosti razreda za ovaj slučaj prikazani su na slici u nastavku.

Slika 8.3. Organizacija boljeg rješenja



Primjenom načela izdvajanja sučelja omogućili smo da svaki od klijenata radi s minimalno potrebnim sučeljem čime smo iz koda uklonili nepotrebne ovisnosti. Dobili smo kod koji je labavije povezan, koji je koncizniji i koji je lakše održavati.

Načelo inverzije ovisnosti

Načelo inverzije ovisnosti pomaže nam pri pisanju koda koji je labavije povezan. Takav kod u pravilu je jednostavnije mijenjati i nadograđivati jer su veze među komponentama rjeđe i slabije. Načelo se sastoji od dva dijela.

1. *Moduli visoke razine ne smiju ovisiti o modulima niske razine. Umjesto toga, oba trebaju ovisiti o dijeljenoj apstrakciji.*

   Pojam *modula* u ovom kontekstu možemo shvatiti i kao *razred* ili *metoda*.

2. *Apstrakcije ne smiju ovisiti o detaljima. Detalji trebaju ovisiti o apstrakcijama.*

   Apstrakcije moraju opisivati svojstva dijeljenog koncepta. Nije dobro kroz apstrakciju provlačiti koncepte koji su implementacijski detalji nekog konkretnog ostvarenja takve apstrakcije. Ovo je ključno jer će omogućiti razvoj više nezavisnih implementacija iste apstrakcije (tj. više implementacija modula niže razine) s kojom će potom modul više razine moći raditi a bez da zna implementacijske detalje ili informacije o konkretnoj odabranoj implementaciji.

Ovo načelo ilustrirat ćemo kroz primjer jednog programskog sustava u kojem se provode različite transakcije o kojima ostaje pisani trag u obliku zapisa. U programskom sustavu postoji poseban podsustav koji potom provjerava valjanost svakog od zapisa te sumnjive zapise dalje prosljeđuje ljudskom operateru na provjere.

Prvi pokušaj izgradnje ovakvog sustava prikazan je u nastavku.

Primjer 8.22. Podsustav za provjeru valjanosti transakcija, prvi pokušaj

```
1 package hr.fer.zemris.principles.dip;
2
3 /**
4  * Podatci o jednoj provedenoj transakciji
5  *
6  */
7 public class Record {
8
9  private String recordData;
10
11  public Record(String recordData) {
12   super();
13   this.recordData = recordData;
14  }
15
16  public String getRecordData() {
17   return recordData;
18  }
19
20 }
21
```

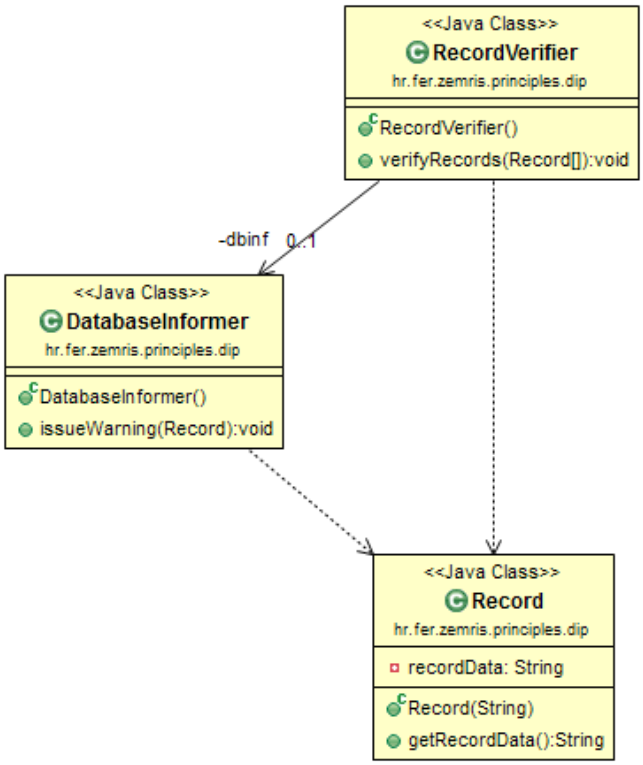
```
1 package hr.fer.zemris.principles.dip;
2
3 /**
4  * Podsustav koji provjerava valjanost provedenih
5  * transakcija i po potrebi bilježi problematične
6  * transakcije.
7  *
8  */
9 public class RecordVerifier {
10
11  private DatabaseInformer dbinf;
12
13  public RecordVerifier() {
14   dbinf = new DatabaseInformer();
15  }
16
17  public void verifyRecords(Record[] records) {
18   for(Record record : records) {
19    boolean valid = true;
20    // Napravi niz provjera nad zapisom i ako
21    // se pronađu problemi, postavi zastavicu
22    // valid na false.
23    if(!valid) {
24     dbinf.issueWarning(record);
25    }
26   }
27  }
28 }
29
```

```
1 package hr.fer.zemris.principles.dip;
2
3 /**
4  * Podsustav koji u bazu podataka zapisuje podatke
```

```
 5  * problematičnim transakcijama.
 6  *
 7  */
 8 public class DatabaseInformer {
 9
10  public DatabaseInformer() {
11  }
12
13  public void issueWarning(Record record) {
14   // Zapiši u bazu podataka podatke o problematičnom
15   // zapisu kako bi ih kasnije odgovorna osoba
16   // provjerila.
17  }
18
19 }
20
```

Razred `Record` predstavlja model jednog zapisa o provedenoj transakciji. Razred `RecordVerifier` je implementacija podsustava za provođenje provjera. Razred `DatabaseInformer` je implementacija podsustava koji poruke o zapisima dodaje u bazu podataka. Pretpostavka je da ljudski operater radi kroz klijentsku aplikaciju pomoću koje iz baze podataka dohvaća i potom obrađuje tako zapisane podatke. Dijagram razreda na kojem su prikazane i ovisnosti dan je na slici u nastavku.

Slika 8.4. Organizacija prvog predloženog rješenja



Razmotrimo li malo ovo rješenje, uočit ćemo njegove mane. Razred `RecordVerifier` ovisi o razredu `DatabaseInformer` Razred `RecordVerifier` pri tome obavlja zadaću analize valjanosti zapisa te obavještava korisnika o pronađenim problematičnim zapisima. Razred `DatabaseInformer` predstavlja implementaciju jedne specifične funkcionalnosti: omogućava pohranu problematičnih zapisa. Prvi razred je primjer razreda visoke razine dok je drugi primjer razreda niske razine. U prikazanom rješenju postoji direktna ovisnost razreda `RecordVerifier` o razredu `DatabaseInformer`.



Bilješka

Pod pojmom razreda visoke razine smatramo razrede koji implementiraju općenite postupke putem niza primitivnijih operacija (primjerice: provjera zapisa = analiza valjanosti + zapisivanje u dnevnik). Pojmom *razredi niske razine* označavamo razrede koji su specifične implementacije općenitih koncepata (primjerice: zapisivanje dnevnika u bazu podataka, zapisivanje dnevnika u datoteku na disku).

Drugi problem je što je razred `RecordVerifier` direktno zadužen za stvaranje primjerka razreda `DatabaseInformer` koji će koristiti za zapisivanje poruka. Ukoliko bi prilikom stvaranja tog primjerka konstruktoru trebalo predati i informacije poput IP adrese i mrežnog pristupa na kojem se nalazi upravitelj baze podataka, naziva baze podataka te korisničkog imena i zaporke koje je potrebno koristiti kako bi se ostvarilo pravo pristupa bazi podataka, razred `RecordVerifier` morao bi znati sve te detaljne informacije kako bi iskonfigurirao objekt koji stvara, a to je svakako loše. Posljedica ovakvog dizajna bi bila da bismo već i prilikom promjene poslužitelja na kojem se nalazi baza podataka morali mijenjati i nanovo prevoditi razred `RecordVerifier`.

Ovaj posljednje opisani problem lagano je riješiti. Sve što je potrebno napraviti jest poslužiti se *injekcijom ovisnosti*. Razredu `RecordVerifier` treba ukloniti zaduženje stvaranja primjerka razreda `DatabaseInformer` koji će koristiti. Umjesto toga, razred `RecordVerifier` treba napisati tako da očekuje da će netko drugi (izvana) stvoriti primjerak razreda `DatabaseInformer` koji je potrebno koristiti i da će ga na neki način dostaviti primjerku razreda `RecordVerifier`. Drugim riječima, primjerak razreda o kojem razred `RecordVerifier` ovisi bit će stvoren i potom injektiran razredu `RecordVerifier` kako bi ga ovaj mogao koristiti. Zaduženje stvaranja tog primjerka prebačeno je na vanjskog klijenta koji će obaviti postupak stvaranja i konfiguriranja potrebnog objekta. Od tuda naziv *injekcija ovisnosti*.



Injekcija ovisnosti

Pojam *injekcija ovisnosti* označava organizaciju koda kod koje se nekom kontekstu dostavlja referenca na primjerak razreda s kojim kontekst obavlja svoju zadaću. Injekcija ovisnosti uobičajeno se implementira predajom reference prilikom poziva konstruktora konteksta ili pak naknadnim pozivom *settera* nad kontekstom.

Da bi razred `RecordVerifier` podržao injekciju ovisnosti (odnosno injekciju objekta koji je primjerak razreda `DatabaseInformer`), treba deklarati člansku varijablu koja će po tipu biti referenca na takav objekt i potom ponuditi mehanizam primanja takvog objekta ili putem konstruktora (koji se tada proširuje jednim argumentom: referencom na taj objekt) ili putem posvećenog *settera* koji će vanjski klijent pozvati nakon što je stvorio primjerak razreda `RecordVerifier` i potreban primjerak razreda `DatabaseInformer`.

Novo rješenje koje koristi inverziju ovisnosti prikazano je u nastavku.

Primjer 8.23. Podsustav za provjeru valjanosti transakcija, drugi pokušaj

```
 1 package hr.fer.zemris.principles.dip2;
 2
 3 /**
 4  * Podatci o jednoj provedenoj transakciji
 5  *
 6  */
 7 public class Record {
 8
 9  private String recordData;
10
11  public Record(String recordData) {
```

```
12   super();
13   this.recordData = recordData;
14  }
15
16  public String getRecordData() {
17   return recordData;
18  }
19
20 }
21
```

```
 1 package hr.fer.zemris.principles.dip2;
 2
 3 /**
 4  * Podsustav koji provjerava valjanost provedenih
 5  * transakcija i po potrebi bilježi problematične
 6  * transakcije.
 7  *
 8  */
 9 public class RecordVerifier {
10
11  private DatabaseInformer dbinf;
12
13  public RecordVerifier(DatabaseInformer dbinf) {
14   this.dbinf = dbinf;
15  }
16
17  public void verifyRecords(Record[] records) {
18   for(Record record : records) {
19    boolean valid = true;
20    // Napravi niz provjera nad zapisom i ako
21    // se pronađu problemi, postavi zastavicu
22    // valid na false.
23    if(!valid) {
24     dbinf.issueWarning(record);
25    }
26   }
27  }
28 }
29
```

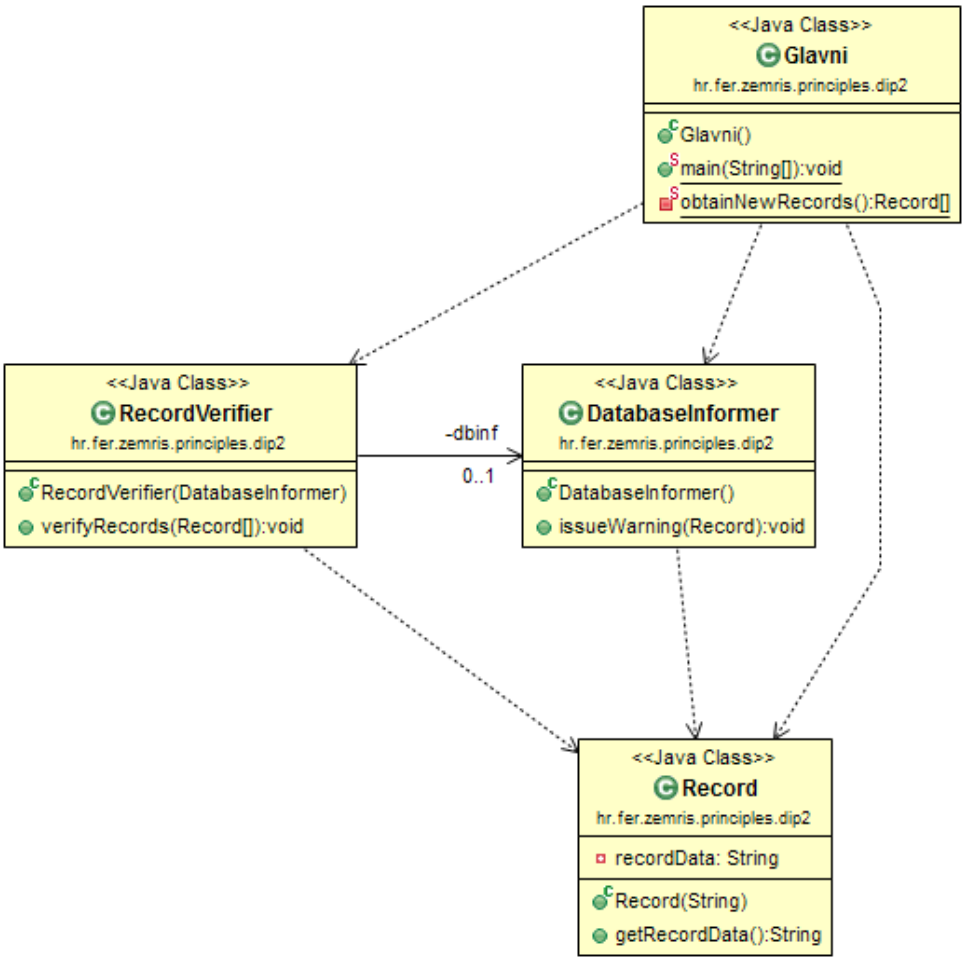
```
 1 package hr.fer.zemris.principles.dip2;
 2
 3 /**
 4  * Podsustav koji u bazu podataka zapisuje podatke
 5  * problematičnim transakcijama.
 6  *
 7  */
 8 public class DatabaseInformer {
 9
10  public DatabaseInformer() {
11  }
12
13  public void issueWarning(Record record) {
14   // Zapiši u bazu podataka podatke o problematičnom
15   // zapisu kako bi ih kasnije odgovorna osoba
16   // provjerila.
17  }
```

```
18
19 }
20
```

```
 1 package hr.fer.zemris.principles.dip2;
 2
 3 public class Glavni {
 4
 5  public static void main(String[] args) {
 6
 7   DatabaseInformer dbinf = new DatabaseInformer();
 8
 9   // Injekcija ovisnosti:
10   RecordVerifier recVer = new RecordVerifier(dbinf);
11
12   while(true) {
13    Record[] records = obtainNewRecords();
14    recVer.verifyRecords(records);
15   }
16
17  }
18
19  private static Record[] obtainNewRecords() {
20   // Aplikacijska logika koja dohvaća još neprovjerene
21   // zapise koji su u međuvremenu nastali i vraća ih
22   return null; // novi neprovjereni zapisi
23  }
24
25 }
26
```

Razred `RecordVerifier` modificiran je tako da više ne stvara potreban primjerak razreda `DatabaseInformer`. Umjesto toga, samo deklarira potrebnu člansku varijablu te referencu na objekt s kojim treba raditi sada prima putem konstruktora. Dodan je i novi razred `Glavni` koji prikazuje primjer klijentskog koda koji je zadužen za stvaranje i konfiguriranje primjerka razreda `DatabaseInformer` te potom za stvaranje primjerka razreda `RecordVerifier` i injektiranje ovisnosti. Dijagram razreda ovog novog rješenja na kojem su prikazane i ovisnosti dan je na slici u nastavku.

Slika 8.5. Organizacija drugog predloženog rješenja



Iako je prikazano rješenje korak u pravom smjeru, ono i dalje ne rješava temeljni problem na kojem smo možda trebali inzistirati već od samog početka: razred `RecordVerifier` kao razred visoke razine ovisi o razredu `DatabaseInformer` koji je razred niske razine. Koja je posljedica? Zahvaljujući prethodno uvedenoj injekciji ovisnosti, sada smo u mogućnosti barem u manjoj mjeri raditi nadogradnju bez promjene: ako trebamo modificirati ponašanje podsustava koji u bazu zapisuje poruke, možemo napraviti novi razred `DifferentDatabaseInformer` koji izvedemo iz razreda `DatabaseInformer` i u njemu implementiramo potrebnu promjenu (primjerice, da ažurira još neke informacije u nekoj drugoj tablici koje su vezane uz zapis koji se dodaje). S ovim pristupom neće biti daljnjih problema jer vanjski klijent sada može umjesto stvaranja primjerka razreda `DatabaseInformer` stvoriti primjerak razreda `DifferentDatabaseInformer` i njega injektirati razredu `RecordVerifier`: kako razred `DifferentDatabaseInformer` nasljeđuje razred `DatabaseInformer` to će biti legalno i stari kod će uredno raditi s predanim primjerkom novog razreda.

Nažalost, priča pada u vodu ako se odlučimo za malo drastičniju promjenu. Što ako želimo promijeniti mehanizam pohrane podataka o problematičnim zapisima tako da nema apsolutno nikakve veze s bazom podataka, već da informacije zapisuje u datoteku na disku, prosljeđuje XML-RPC-om drugom poslužitelju ili ih pak šalje e-mailom? To ćemo teško napraviti nasljeđivanjem razreda koji implementira logiku komuniciranja s bazom podataka. Razlog pojave ovakve nemogućnosti nadogradnje leži u činjenici da razred viske razine ovisi o razredu niske razine.

Rješenje uočenog problema je *inverzija ovisnosti*. Umjesto da razred visoke razine ovisi o razredu niske razine, nužno je pronaći dijeljenu apstrakciju čijom uporabom razred visoke razine može obaviti svoj posao a za koju razred niske razine može ponuditi implementaciju. Drugim riječima, potrebno je pronaći minimalan potreban skup primitiva koji su dovoljni kako bi razred

visoke razine njihovom uporabom mogao obaviti svoj posao i njih "opisati" kroz dijeljenu apstrakciju (sučelje ili apstraktan razred).

U našem konkretnom slučaju, kako bi izgledala ta dijeljena apstrakcija? Razred `RecordVerifier`, jednom kada pronađe problematičan zapis treba imati mogućnost da nam objektom koji će mu ponuditi uslugu zapisivanja informacija pozove metodu kojom će zapisati informacije o tom zapisu. Osim takve metode, ovom razredu ne treba ništa drugo. Stoga je u rješenju koje je prikazano u nastavku definirano novo sučelje: `IInformer` kao sučelje koje definira samo jednu metodu: `issueWarning(Record record)` čija je namjena upravo to. Cjelokupno rješenje prikazano je u nastavku.

Primjer 8.24. Podsustav za provjeru valjanosti transakcija, rješenje

```
1 package hr.fer.zemris.principles.dip3;
2
3 /**
4  * Podatci o jednoj provedenoj transakciji
5  *
6  */
7 public class Record {
8
9  private String recordData;
10
11  public Record(String recordData) {
12   super();
13   this.recordData = recordData;
14  }
15
16  public String getRecordData() {
17   return recordData;
18  }
19
20 }
21
```

```
1 package hr.fer.zemris.principles.dip3;
2
3 /**
4  * Sučelje koje opisuje proizvolju implementaciju
5  * razreda koji zna poslati upozorenje o problematičnom
6  * zapisu.
7  *
8  */
9 public interface IInformer {
10
11  public void issueWarning(Record record);
12
13 }
14
```

```
1 package hr.fer.zemris.principles.dip3;
2
3 /**
4  * Podsustav koji provjerava valjanost provedenih
5  * transakcija i po potrebi bilježi problematične
6  * transakcije.
7  *
8  */
9 public class RecordVerifier {
```

```
10
11  private IInformer inf;
12
13  public RecordVerifier(IInformer inf) {
14   this.inf = inf;
15  }
16
17  public void verifyRecords(Record[] records) {
18   for(Record record : records) {
19    boolean valid = true;
20    // Napravi niz provjera nad zapisom i ako
21    // se pronađu problemi, postavi zastavicu
22    // valid na false.
23    if(!valid) {
24     inf.issueWarning(record);
25    }
26   }
27  }
28 }
29
```

```
 1 package hr.fer.zemris.principles.dip3;
 2
 3 /**
 4  * Podsustav koji u bazu podataka zapisuje podatke o
 5  * problematičnim transakcijama.
 6  *
 7  */
 8 public class DatabaseInformer implements IInformer {
 9
10  public DatabaseInformer() {
11  }
12
13  public void issueWarning(Record record) {
14   // Zapiši u bazu podataka podatke o problematičnom
15   // zapisu kako bi ih kasnije odgovorna osoba
16   // provjerila.
17  }
18
19 }
20
```

```
 1 package hr.fer.zemris.principles.dip3;
 2
 3 /**
 4  * Podsustav koji elektroničkom poštom šalje podatke
 5  * problematičnim transakcijama.
 6  *
 7  */
 8 public class EMailInformer implements IInformer {
 9
10  public EMailInformer() {
11  }
12
13  public void issueWarning(Record record) {
14   // Pošalji e-mail o problematičnom zapisu
15   // kako bi ga kasnije odgovorna osoba
16   // provjerila.
```

```
17  }
18
19 }
20
```

```
 1 package hr.fer.zemris.principles.dip3;
 2
 3 /**
 4  * Implementacija podsustava koji sam po sebi ne zna pohranjivati
 5  * podatke o problematičnim zapisima već taj zadatak može delegirati
 6  * drugim podsustavima.
 7  *
 8  */
 9 public class CompositeInformer implements IInformer {
10
11  private IInformer[] informers;
12
13  public CompositeInformer(IInformer[] informers) {
14   // Zapamti kopiju predanih podataka
15   this.informers = new IInformer[informers.length];
16   for(int i = 0; i < informers.length; i++) {
17    this.informers[i] = informers[i];
18   }
19  }
20
21  @Override
22  public void issueWarning(Record record) {
23   for(IInformer inf : informers) {
24    inf.issueWarning(record);
25   }
26  }
27
28 }
29
```

```
 1 package hr.fer.zemris.principles.dip3;
 2
 3 public class Glavni {
 4
 5  public static void main(String[] args) {
 6
 7   // Kao implementaciju biramo kompozitni objekt
 8   // koji će obavljati delegiranje na dvije konkretne
 9   // implementacije.
10   IInformer inf = new CompositeInformer(
11    new IInformer[] {
12     new DatabaseInformer(),
13     new EMailInformer()
14    }
15   );
16
17   RecordVerifier recVer = new RecordVerifier(inf);
18
19   while(true) {
20    Record[] records = obtainNewRecords();
21    recVer.verifyRecords(records);
22   }
23
```

```
24   }
25
26   private static Record[] obtainNewRecords() {
27    // Aplikacijska logika koja dohvaća još neprovjerene
28    // zapise koji su u međuvremenu nastali i vraća ih
29    return null; // novi neprovjereni zapisi
30   }
31
32 }
33
```

U prikazanom rješenju razred `RecordVerifier` više ne ovisi o konkretnoj implementaciji podsustava za pamćenje poruka već deklarira ovisnost o objektu koji implementira sučelje `IInformer` što je direktna ovisnost o apstrakciji. Potom upravo takav objekt prima kroz konstruktor što je primjer injekcije ovisnosti.

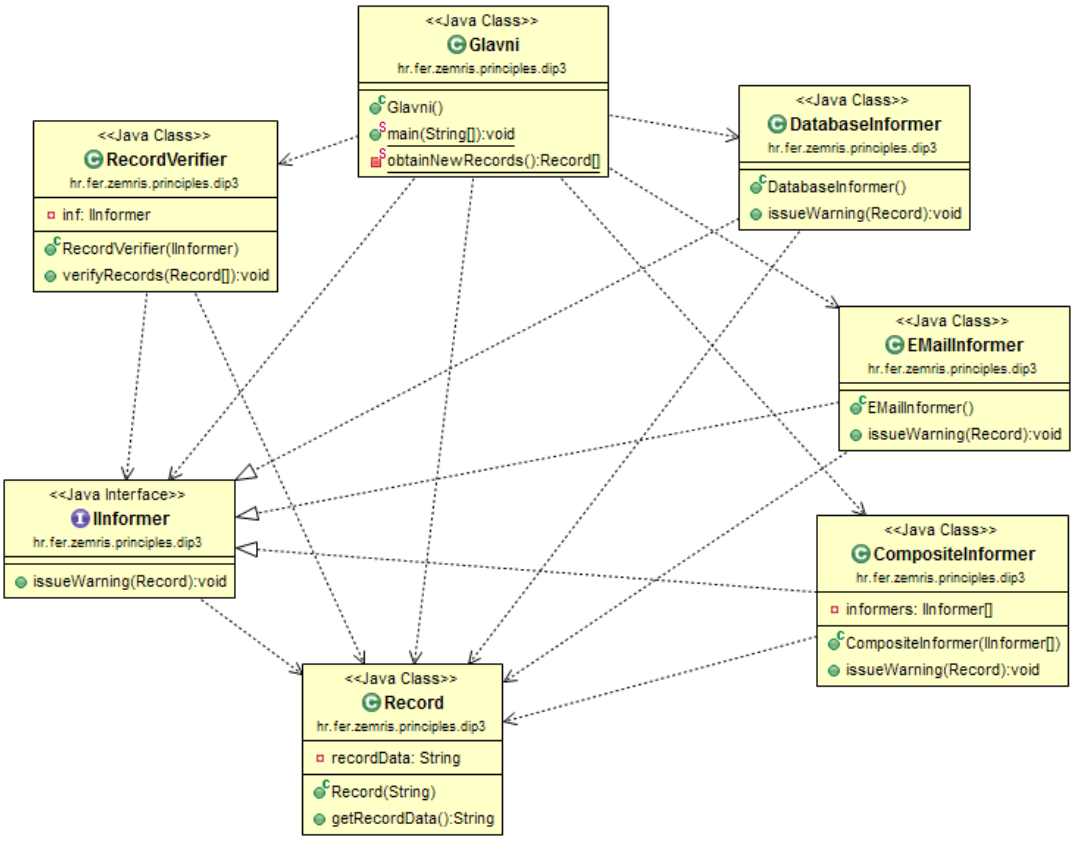
Razred `DatabaseInformer` u ovom je rješenju prilagođen tako da implementira sučelje `IInformer` kako bi se njegovi primjerci mogli slati kroz konstruktor razreda `RecordVerifier`.

Uz razred `DatabaseInformer` u prikazanom rješenju ponuđena su još dva rješenja podsustava za zapisivanje poruka: radi se o razredima `EMailInformer` te `CompositeInformer`. Razred `EMailInformer` implementira sučelje `IInformer` i sve poslove zapisivanje rješava generiranjem i slanjem e-mail poruka korisniku koji će podatke trebati ručno pregledati. Zahvaljujući činjenici da razred `EMailInformer` implementira sučelje `IInformer`, primjerke tog razreda također ćemo moći bez ikakvih problema injektirati u razred `RecordVerifier` čime smo doista osigurali nadogradnju bez promjene — sada smo u mogućnosti bilo kakvu implementaciju podsustava za zapisivanje poruka poslati starom kodu u razredu `RecordVerifier` i taj će biti u stanju raditi s novom implementacijom bez potrebe za ikakvim promjenama.

Možda najbolja ilustracija takvog pristupa je još jedna implementacija sučelja `IInformer`: razred `CompositeInformer` koji je također implementacija podsustava za zapisivanje poruka ali koja sama po sebi ništa nikamo ne zna zapisati. Umjesto toga, konstruktor ovog razreda prima polje referenci na druge podsustave za zapisivanje poruka te u implementaciji metode `issueWarning` dobiveni zapis delegira svakom od tih podsustava. Uporabom ovog razreda možemo stvoriti podsustav koji, primjerice, posao zapisivanja delegira i podsustavu koji to zapisuje u bazu podataka, i podsustavu koji šalje e-mail poruke. Upravo ovakva konfiguracija prikazana je u klijentskom kodu u razredu `Glavni` u metodi `main`. Uočite kako će sada za svaki pronađeni problematičan zapis nastati i jedan zapis u bazi podataka i bit će poslan jedna e-mail poruka — a sve bez ikakve daljnje intervencije u kod razreda `RecordVerifier`. Ovakvo ponašanje postalo je moguće jer smo postigli rješenje odnosno organizaciju koda u kojoj i modul visoke razine i i modul niske razine oba ovise o dijeljenoj apstrakciji.

Dijagram razreda za definirane razrede i sučelja prikazan je na slici u nastavku.

Slika 8.6. Organizacija rješenja uz inverziju i injekciju ovisnosti



Poglavlje 9. Tehnologija Java Generics

U ovom poglavlju ukratko ćemo se osvrnuti i upoznati s tehnologijom *Java Generics*. Ova tehnologija u Javu je uvedena od inačice 1.5 s namjerom da se pospješi pisanje višestruko iskoristivog koda uz istovremenu mogućnost provođenja provjera ispravnosti korištenih tipova podataka već tijekom samog prevođenja. Ako ste ikada radili nešto ozbiljnije u jeziku C++, tehnologija koja je (barem površinski) vrlo slična su obrasci (*C++ templates*) koji su korišteni pri izgradnji C++-ove biblioteke STL (*Standard Template Library*). Obrasci u C++ omogućavaju pisanje generičkog koda koji umjesto konkretnih tipova podataka koristi zamjenske oznake. Potom korisnik takav kod može pozvati uz specifikaciju konkretnog tipa (zadaje se tip podatka koji se prosljeđuje na mjesto zamjenske oznake). Prilikom prevođenja, prevoditelj gleda uz koje je sve različite tipove podataka taj kod pozvan pa napravi toliko novih kopija koda u kojima zamjenske oznake zamijeni tim tipom i potom tako (automatski) generiran kod prevede. Kako bismo pojasnili ideju, napisat ćemo jedan kratak C++ program koji ilustrira koncept.

Primjer 9.1. Izračun srednje vrijednosti i obrasci (jezik C++)

```
 1 #include <iostream>
 2
 3 template <typename X>
 4 X sredina(X a, X b) {
 5   X razlika = b - a;
 6   return a + razlika/2;
 7 }
 8
 9 int main() {
10   int a=2, b=8;
11   double c=4.2, d=12.4;
12   short e=6, f=20;
13
14   std::cout << sredina(a,b) << std::endl;
15   std::cout << sredina(c,d) << std::endl;
16   std::cout << sredina(e,f) << std::endl;
17
18   return 0;
19 }
20
```

Primjer 9.1 započinje definiranjem obrasca funkcije za izračun srednje vrijednosti: retci 3 do 7. Redak 3 definira, da u obrascu funkcije, *X* predstavlja zamjensku oznaku; ova se oznaka potom može koristiti gdje god se pojavljuje *tip* podatka. Potom redak 4 definira funkciju-obrazac imena *sredina* čija je povratna vrijednost tipa *X* i koja prima dva argumenta, oba tipa *X*. Redak 5 definira još pomoćnu varijablu `razlika` također tipa *X* u koju se pohranjuje razlika drugog i prvog argumenta. Konačno, redak 6 definira povratnu vrijednost metode. Razmislite sada što smo time definirali? Na prvi pogled, čini se da smo definirali funkciju. Međutim, tako definiranu funkciju ne možemo prevesti u strojni kod niti je možemo pozvati i izvršiti. Naime, iako je nama (čitatelju) namjera koda jasna, niti mi sami ne bismo znali odgovoriti na vrlo jednostavna pitanja tipa: "koliko mjesta na stogu zauzimaju argumenti metode" ili pak "koju ćemo procesorsku instrukciju koristiti za izračun razlike". Odgovori na ta pitanja ovise o konkretnom tipu: što je točno taj *X*. Stoga smo ovime zapravo definirali funkciju-obrazac: kod koji opisuje algoritam ali još uvijek ne specificira konkretne tipove podataka nad kojima radi. Razmislite li malo - dobili smo izuzetno moćan mehanizam koji omogućava da algoritam napišemo jednom i da ga potom koristimo nad različitim tipovima podataka: imamo višestruku iskoristivost napisanog koda.

Da bismo primjer razumjeli do kraja, ostalo je još odgovoriti kako se iz ovoga na kraju generira strojni kod (i tu je osnovna razlika između obrazaca u jeziku C++ te tehnologije *Java Generics* jezika Java). Za prevođenje programa danog u primjeru 9.1 koristi se višeprolazni prevoditelj koji u jednom prolazu najprije gleda i traži po kodu sva mjesta na kojima smo pozivali definiranu funkciju-obrazac (u našem primjeru to su retci 14, 15 i 16) te pamti uz koje smo to sve različite tipove podataka radili. Tom analizom danog primjera prevoditelj će ustanoviti da smo funkciju-obrazac pozivali nad argumentima tipa `int`, nad argumentima tipa `double` te nad argumentima tipa `short`. Potom će iz izvornog koda obrisati definiciju funkcije-obrasca a na to mjesto umetnuti odgovarajući broj kopija definicije funkcije pri čemu će zamjenske oznake supstituirati pronađenim tipovima podataka; svakoj tako generiranoj funkciji (u kojoj sada više nema zamjenskih oznaka) dodijelit će novo jedinstveno ime koje će također zamijeniti i na mjestu gdje je funkcija-obrazac bila pozvana s trenutnim tipom. Time će sam prevoditelj originalni kod automatski transformirati u novi kod koji je prikazan u primjeru 9.2.

Primjer 9.2. Izračun srednje vrijednosti nakon uklanjanja obrasca (jezik C++)

```
 1 #include <iostream>
 2
 3 int _Z7sredinaIiET_S0_S0_(int a, int b) {
 4   int razlika = b - a;
 5   return a + razlika/2;
 6 }
 7
 8 double _Z7sredinaIdET_S0_S0_(double a, double b) {
 9   double razlika = b - a;
10   return a + razlika/2;
11 }
12
13 short _Z7sredinaIsET_S0_S0_(short a, short b) {
14   short razlika = b - a;
15   return a + razlika/2;
16 }
17
18 int main() {
19   int a=2, b=8;
20   double c=4.2, d=12.4;
21   short e=6, f=20;
22
23   std::cout << _Z7sredinaIiET_S0_S0_(a,b) << std::endl;
24   std::cout << _Z7sredinaIdET_S0_S0_(c,d) << std::endl;
25   std::cout << _Z7sredinaIsET_S0_S0_(e,f) << std::endl;
26
27   return 0;
28 }
29
```

Nakon uklanjanja obrazaca iz koda i generiranja odgovarajućih primjeraka obrazaca (engl. *template instantitation*) dobili smo kod koji se sada može prevesti u strojni kod i izvršiti. I ne samo to; u ovom se kodu svaki konkretan primjerak funkcije-obrasca prevodi nezavisno i može se optimirati nezavisno. Naravno, naglasimo još jednom da je taj čitav postupak za programera transparentan - programer piše kod kako je prikazano u primjeru 9.1: sve ostalo (uključivo "međukod" prikazan u primjeru 9.2) je nešto što prevoditelj radi u pozadini prilikom prevođenja programa. Želite li se uvjeriti da se stvari doista događaju na taj način, možete to učiniti vrlo jednostavno. Pohranite program prikazan u primjeru 9.1 na disk u datoteku `primjer.cpp`, otvorite ljusku, pozicionirajte se u taj direktorij te (uz pretpostavku da na računalu imate instaliran prevoditelj g++ za jezik C++) zadajte naredbu:

```
g++ -S primjer.cpp
```

Dobit ćete datoteku imena `primjer.s`: tekstovnu datoteku s asembleskim instrukcijama. Kako je program kratak, lagano ćete se uvjeriti da postoji strojni kod za tri metode `sredina` i da se u svakoj od njih aritmetičke operacije obavljaju drugim instrukcijama (koje odgovaraju tipu podataka za koje je taj strojni kod generiran).



C++ obrasci i statički polimorfizam

Uporabom obrazaca u jeziku C++ je omogućeno pisanje višestruko iskoristivog koda. Prethodno prikazani primjer predstavlja ujedno i opis tzv. *statičkog polimorfizma*: možemo se praviti da imamo tri različite metode `sredina` te da prevoditelj prilikom prevođenja bira koju će od njih pozvati - stoga je poziv polimorfni a kako se ne razrješuje prilikom izvođenja već odmah prilikom prevođenja, zovemo ga *statički*. Ovaj polimorfizam "jeftiniji" je od dinamičkog i ne zahtjeva da objekti sa sobom vuku pokazivač na tablicu virtualnih funkcija.

Kako bismo pojasnili razliku između rješenja u Javi, istaknimo još pozitivne i negativne strane rješenja koje se koristi u jeziku C++. Kao negativnu stranu istaknut ćemo porast količine generiranog strojnog koda te činjenicu da statički polimorfizam temeljen na obrascima radi samo i isključivo prilikom prevođenja: prevoditelj mora znati na koje će sve načine funkcija-obrazac biti pozivana kako bi izgenerirano sve konkretne primjerke te funkcije (što u okruženju koje dozvoljava dinamičko učitavanje koda može biti problematično). Pozitivna strana je maksimum performansi: kako se za svaki različit tip uz koji je funkcija-obrazac bila pozivana generira po jedan primjerak koda te funkcije za baš taj tip, svaku je funkciju moguće nezavisno optimirati i time postići maksimalne performanse. Također, tehnologija je primjenjiva na apsolutno sve tipove podataka (i primitivne i razrede).

Parametrizirani kod u Javi

Programski jezik Java, baš kao i prethodno spomenuti C++, dozvoljava pisanje metoda-obrazaca, sučelja-obrazaca te razreda-obrazaca. U kontekstu Jave, zovemo ih *parametriziranima*: parametriziranom metodom, parametriziranim sučeljem te parametriziranim razredom. Zamjenske oznake koje ćemo u Javi zvati *parametrima* baš kao i u jeziku C++ predstavljaju zamjenu za tipove podataka. Međutim, postoji važna razlika: parametar ne može predstavljati primitivni tip (ne može predstavljati tip `int`). Najčešće, parametri predstavljaju razrede ili sučelja.

Prvi primjer koji ćemo pogledati u nastavku je pisanje parametrizirane metode (primjer 9.3). Definicija parametara piše se u zagradama <>; ako metoda treba više parametara, imena se razdvoje zarezima. Definicija parametra koji je lokalan za metodu piše se prije povratne definiranja tipa povratne vrijednosti metode. U prikazanom kodu napisane su dvije parametrizirane metode: `paranSazetak` koja vraća `boolean` i prima jedan argument čiji je tip određen parametrom (u retku 23, definiran je parametar imena `T` nakon ključnih riječi `public static`) te `veciSazetak` koja također definira parametar imena `T`, deklarira da prima dva argumenta tipa koji određen tim parametrom te da vraća referencu koja je tog istog tipa. Iako obje metode deklariraju parametar istog imena, ti parametri međusobno nisu nikako povezani; ovako definiran parametar lokalan je za samu metodu (baš kao što su i broj, vrsta te nazivi argumenata koje prima metoda).

Metoda `paranSazetak`, osim što ilustrira način definiranja parametra i činjenicu da se taj parametar potom može koristiti na mjestu gdje se očekuje tip podatka, ni na koji način dalje ne koristi taj parametar. Potpuno jednaku funkcionalnost postigli bismo da smo metodu jednostavno napisali kako je prikazano u nastavku.

```
1 public static boolean paranSazetak(Object objekt) {
2  int sazetak = objekt.hashCode();
3  return sazetak % 2 == 0;
```

```
4 }
```

I jedna i druga inačica ove metode mogu kao argument primiti bilo kakav objekt. Dapače, ako se parametar definira bez ikakvih ograničenja (što je ovdje bio slučaj), tada se prevoditelj prema njemu ponaša kao da na tom mjestu zapravo piše `java.lang.Object`. Da je tome tako, možete se uvjeriti jednostavnim eksperimentom. Prepišite primjer 9.3 u Eclipse (ili koji već IDE koristite), otiđite u redak 24, obrišite `.hashCode();` pa ponovno iza `objekt` pritisnite točku i sačekajte sa se aktivira podsustav za automatsko nadopunjavanje koda koji će Vam ponuditi popis metoda koje možete pozvati nad tim objektom. Promotrite malo popis: ustanovit ćete da Vam se nude samo metode razreda `java.lang.Object` jer, s obzirom da nema nikakvih dodatnih informacija o tom tipu, prevoditelj ga tretira kao da je tipa `java.lang.Object` i nudi samo njegove metode.

Primjer 9.3. Primjer parametriziranih metoda

```
 1 package book.java.generics;
 2
 3 public class PrimjerMetode {
 4
 5  public static void main(String[] args) {
 6   Integer broj1 = 27;
 7   Integer broj2 = -13;
 8   String ime1 = "Pero";
 9   String ime2 = "Ivo";
10
11   System.out.println(paranSazetak(broj1));
12   System.out.println(paranSazetak(ime1));
13
14   Integer broj3 = veciSazetak(broj1, broj2);
15   String ime3 = veciSazetak(ime1, ime2);
16
17   System.out.println(broj3);
18   System.out.println(ime3);
19
20   //String ime4 = veciSazetak(broj1, broj2);
21  }
22
23  public static <T> boolean paranSazetak(T objekt) {
24   int sazetak = objekt.hashCode();
25   return sazetak % 2 == 0;
26  }
27
28  public static <T> T veciSazetak(T o1, T o2) {
29   int sazetak1 = o1.hashCode();
30   int sazetak2 = o2.hashCode();
31   return sazetak1>=sazetak2 ? o1 : o2;
32  }
33 }
34
```

Metode `veciSazetak` koja je definirana u retku 28 već je zanimljivija jer ilustrira nešto što bez tehnologije *Java Generics* ne bismo mogli postići ovako jednostavno. To je primjer metode koja definira da je i njezina povratna vrijednost definirana parametrom `T`: metoda definira da prima dva objekta istog tipa te da vraća rezultat koji je tipa određenog tipom argumenata. To znači da ako metodu pozovemo nad dva objekta tipa `Integer`, rezultat će biti tipa `Integer`. Ako metodu pozovemo nad dva objekta tipa `String`, rezultat će biti tipa `String`. I doista, lagano ćemo se uvjeriti da prevoditelj metodu tretira na taj način. Promotrite retke 14 i 15: prvi metodu poziva nad brojevima (`Integer`) i rezultat pohranjuje u varijablu tipa `Integer` dok

drugi metodu poziva nad objektima tipa `String` i rezultat pohranjuje u varijablu tipa `String`. Taj se kod prevodi bez ikakvih problema i radi ispravno - prepišite ga i pokrenite. Redak 20 (zakomentiran jer nije korektan) ilustrira da provjere tipova rade u skladu s očekivanjem: ako metodu pozovemo nad dva objekta tipa `Integer`, onda ćemo dobiti i rezultat tog tipa ali njega ne možemo pohraniti u varijablu koja je tipa `String` - to je pogreška pri prevođenju. Otkomentirajte taj redak i uvjerite se da će se prevoditelj pobuniti. U tijelu metode `veciSazetak`, oba argumenta tretiramo kao primjerke razreda `java.lang.Object` i funkcionalnost metode ostvarujemo samo pozivanjem metoda razreda `Object`.

Bez uporabe tehnologije *Java Generics*, mogli bismo napisati metodu koja prima i vraća objekte: i ta bi metoda, baš kao i naša parametrizirana, mogla primiti primjerke bilo kojeg razreda. Evo definicije u nastavku.

```
1 public static Object veciSazetak(Object o1, Object o2) {
2  int sazetak1 = o1.hashCode();
3  int sazetak2 = o2.hashCode();
4  return sazetak1>=sazetak2 ? o1 : o2;
5 }
```

Sada bismo, međutim, u metodi `main` imali problema. Primjerice, uz ovako napisanu metodu, retci 14 i 15 više se ne bi mogli prevesti. Dobili bismo poruku pogreške da ne možemo u varijablu tipa `Integer` zapisati `Object`. Stvar bismo popravili tako da dodamo eksplicitno ukalupljivanje, kako je prikazano u nastavku.

```
1 Integer broj3 = (Integer)veciSazetak(broj1, broj2);
2 String ime3 = (String)veciSazetak(ime1, ime2);
```

Međutim, s obzirom da smo zbog mogućnosti pozivanja metode uz različite vrste argumenata sve tipove degradirali na `Object` (kako bismo dobili višestruku iskoristivost koda i izbjegli kopiranje i ponavljanje), izgubili smo mogućnost statičke kontrole ispravnosti tipova. Naime, i sljedeći kod će se korektno prevesti, bez ikakve dojave o pogrešci.

```
1 Integer broj3 = (Integer)veciSazetak(ime1, ime2);
2 String ime3 = (String)veciSazetak(broj1, broj2);
```

Poruku o pogrešci sada ćete dobiti tek ako takav program pokrenete i on se raspadne, jer je prvi poziv metode primio dva stringa pa vraća string koji potom pokušavate ukalupiti u `Integer`. Rezultat će biti iznimka `ClassCastException` - ali tek tijekom izvođenja. Ako kod niste pokrenuli i ako nemate običaj testirati napisani kod, mogli biste zaključiti da će kod raditi (uspjeli ste ga prevesti). Naša parametrizirana inačica metoda koju smo originalno prikazali nema takvih problema.



Bilješka

Tehnologija *Java Generics* omogućava pisanje parametriziranog koda uz očuvanje mogućnosti statičke provjere korektnosti tipova podataka; ako postoji pogreška, ona će biti otkrivena tijekom prevođenja programa.

Prije no što nastavimo dalje, usporedimo sada implementaciju ove tehnologije s implementacijom obrazaca u jeziku C++ koju smo opisali u uvodu poglavlja. Parametrizirani kod napisan u Javi ne tretira se kao obrazac već kao konačni kod koji se direktno prevodi u strojni kod (za Javin virtualni stroj). Posljedica je da u programu prikazanom u primjeru 9.3 neće nastati strojni kod za metodu `veciSazetak` čiji su argumenti primjerci razreda `Integer` te zaseban strojni kod za implementaciju metode čiji bi argumenti bili primjerci razreda `String`. Neovisno o broju različitih načina na koje smo pozivali parametriziranu metodu `veciSazetak` nastat će samo jedna inačica strojnog koda pri čemu će se parametri postaviti na najopćenitiju dopuštenu vrijednost. Konkretno, prilikom generiranja strojnog koda, prevoditelj će parametar `T` u metodi `veciSazetak` tretirati kao da je tip `java.lang.Object`. Međutim, prilikom prevođenja metode `main`, prevoditelj pamti da se radi o parametriziranoj metodi, zna u kakvoj su vezi tipovi

argumenata i tip povratne vrijednosti i to kontrolira. Tako u retku 14 zaključuje da je povratna vrijednost metode tipa `Integer`, vidi da je varijabla u koju se pohranjuje rezultat također tipa `Integer` i to dozvoljava (i ne zahtjeva eksplicitno ukalupljivanje). Primjerice, metodu `main` mogli biste proširiti još jednim pozivom:

```
1 System.out.println(veciSazetak("Jasna", "Petra").substring(1));
```

Uočite: nad povratnom vrijednosti metode `veciSazetak(...)` pozvali smo metodu `substring(...)` - i nema poruke o pogrešci. Naša metoda, s obzirom da je primila dva stringa kao rezultat vraća string i nad njime je legalno pozivati metode koje su definirane za razred `String`, i to možemo raditi bez eksplicitnog ukalupljivanja jer prevoditelj može zaključiti iz deklaracije ove parametrizirane metode da je rezultat upravo `String`.

Ponekad je potrebno ograničiti tipove podataka koje parametar može primiti. Java nudi dvije ograde (*barem nekog tipa* te *najviše nekog tipa*). Želimo li reći da parametar može poprimiti tip koji je u stablu nasljeđivanja *barem nekog tipa*, koristimo ključnu riječ `extends`. Tako definicija `<T extends Exception>` definira parametar `T` koji može biti tipa `java.lang.Exception` ili bilo kojeg tipa koji je izveden iz tog tipa (zato u čitanju koristimo *barem*). Na tip se može postaviti i više ograda koje se onda razdvajaju znakom &; primjerice: `<T extends Exception&Serializable&Comparable>` definira da `T` može biti bilo kojeg tipa koji je istovremeno svih triju navedenih tipova. U slučaju navođenja više ograničenja, samo jedno od njih (i to prvo) smije biti tip koji je razred (ostalo moraju biti sučelja). Evo jednog primjera u nastavku.

Primjer 9.4. Ograničavanje parametara

```
 1 package book.java.generics;
 2
 3 public class OgranicavanjeParametra {
 4
 5  public static void main(String[] args) {
 6   Number n = veci(Integer.valueOf(2), Integer.valueOf(5));
 7   Integer n2 = veci(Integer.valueOf(2), Integer.valueOf(5));
 8
 9   Number n3 = veci(Integer.valueOf(2), Long.valueOf(5));
10   //Integer n4 = veci(Integer.valueOf(2), Long.valueOf(5));
11   //Long n5 = veci(Integer.valueOf(2), Long.valueOf(5));
12  }
13
14  public static <T extends Number> T veci(T n1, T n2) {
15   return n1.doubleValue()>=n2.doubleValue() ? n1 : n2;
16  }
17 }
18
```

Proanalizirajmo redak po redak. Parametrizirana metoda prikazana je od retka 14. Definira jedan parametar `T` koji mora biti barem `java.lang.Number`. Prisjetimo se: iz `java.lang.Object` je izveden `java.lang.Number`, a iz njega dalje `java.lang.Integer`, `java.lang.Double`, `java.lang.Double` i drugi.

U retku 6 imamo prvi poziv te metode. Kako su oba argumenta tipa `java.lang.Integer`, prevoditelj zaključuje da se u ovom pozivu `T` veže na tip `java.lang.Integer` pa je i tip povratne vrijednosti u tom pozivu upravo taj. U retku 6, povratna se vrijednost pokušava upisati u varijablu tipa `java.lang.Number` i to je legalno jer je `Integer` izveden iz `Number`. U retku 7, parametar se veže na isti tip i zapisuje u varijablu tipa `Integer` što je legalno.

Poziv u retcima 9, 10 i 11 je interesantniji. Prvi argument je tipa `Integer` a drugi tipa `Long`. Kako je u deklaraciji naše parametrizirane metode definirano da su oba argumenta

istog tipa, prevoditelj će pronaći najveći "zajednički nazivnik" za predane tipove: najviše što je u stablu nasljeđivanja razreda `Integer` i razreda `Long` zajedničko je da su oba izvedena iz `Number` - nakon toga se stabla granaju. Stoga će za taj poziv prevoditelj parametar `T` vezati na tip `java.lang.Number` i zaključiti da je povratna vrijednost metode upravo `java.lang.Number`. Uz tu argumentaciju, rezultat se može pohraniti u varijablu tipa `Number` (redak 9) ali ne može u varijable tipa `Integer` i `Long` (retci 10 i 11). Zapravo, ono što će prevoditelj otkriti da je zajedničko tipovima `Integer` i `Long` je da su oba izvedena iz razreda `Number` i da oba implementiraju parametrizirano sučelje `java.lang.Comparable` pa će `T` biti vezan na `Number&Comparable<?>` (što u konačnici ne mijenja dani zaključak); jedino što znači je da bi i sljedeći redak također bio legalan:

```
1 Comparable<?> n6 = veci(Integer.valueOf(2), Long.valueOf(5));
```

Više o parametriziranim razredima i sučeljima bit će riječi u sljedećem podpoglavlju.

Nakon ove analize, pokušajte uočiti za što se koriste parametri: njihova stvarna vrijednost nije bitna prilikom generiranja strojnog koda parametrizirane metode. Strojni kod generirat će se samo jednom, i to uz najstrožu poznatu granicu parametra. U ovom primjeru, prevoditelj će prilikom prevođenja metode `veci` parametar `T` tretirati kao tip `Number` jer je to njegova poznata ograda. Prilikom pozivanja parametriziranih metoda, svaki se poziv analizira nezavisno i tamo se samo pokušava provjeriti je li sve tipski korektno (u skladu s definicijom metode i postavljenim ogradama).

Druga moguća ograda je ograda tipa *najviše*: postiže se uporabom ključne riječi `super`. Evo primjera. Definicija `<T super Exception>` definira da se `T` može vezati na tip podatka koji je najviše `Exception`. Prisjetite li se stabla nasljeđivanja iznimaka, od tog razreda prema vrhu prolazimo još preko razreda `Throwable` i potom dolazimo do razreda `Object`; to su jedine tri legalne mogućnosti na koje se pri pozivu parametar `T` koji je definiran uz prikazanu ogradu može vezati.

Parametrizirani razredi i sučelja

Osim parametrizacije metoda, moguće je parametrizirati i razrede i sučelja. U tom slučaju definicija parametra piše se odmah nakon naziva razreda odnosno sučelja. Evo primjera koji to ilustrira.

```
1 public class Razred<T extends Number,E> {
2   ...
3   private T varijabla;
4   ...
5   public T metoda1(T arg1, E arg2) {
6    ...
7   }
8 }
9
10 public interface Sucelje<X> {
11    ...
12    public X dohvati();
13    public void postavi(X podatak);
14    ...
15 }
```

Uočite: parametar (ili parametri, kako je ilustrirano kod razreda) definiran na razini razreda/sučelja vidljiv je unutar tijela razreda/sučelja i može se koristiti kao tip podatka pri deklaraciji članskih varijabli te pri definiranju metoda. Parametar definiran na toj razini je u određenom smislu globalan za taj razred/sučelje. Važno je napomenuti da je takav parametar vidljiv samo nestatičkim elementima razreda/sučelja jer mu se vrijednost zadaje prilikom stvaranja primjerka. Statičke metode i članske varijable ne vide ovako definirane parametre.

Primjer koji smo prikazali definira parametrizirani razred `Razred` koji definira dva parametra (`T` i `E`) pri čemu parametar `T` može predstavljati bilo koji tip podatka koji je barem `java.lang.Number`. Ti su parametri potom korišteni pri definiranju jedne članske varijable te jedne metode. Prikazano sučelje `Sucelje` parametrizirano je jednim parametrom `X` na koji nije postavljeno niti jedno ograničenje pa taj parametar može predstavljati bilo koji tip podataka (dakako uz ogradu da ne može biti primitivni tip).

Pogledajmo sada primjer koji ilustrira uporabu parametriziranih razreda i daje motivaciju zašto bismo to htjeli raditi. Prisjetite se: programski jezik Java programeru ne nudi izravnu mogućnost rada s pokazivačima, što znači da ne možemo napisati metodu koja će moći mijenjati lokalne varijable pozivatelja (što je nekada zgodna mogućnost i u C-u je postižemo tako da metodi umjesto sadržaja varijable pošaljemo adresu varijable). Evo što želimo. Potrebno je napisati metodu kojoj ćemo poslati dvije varijable tipa `Integer` i koja će posredstvom slučajnog mehanizma promijeniti ili sadržaj prve varijable ili sadržaj druge varijable (ali ne obje istovremeno). Isječak koda koji pokazuje naivni pokušaj postizanja tako nečega prikazan je u nastavku.

```
 1 public static void main(String[] args) {
 2   Integer prva = 5;
 3   Integer druga = 8;
 4   obradi(prva, druga);
 5   System.out.format("Prva: %d, druga: %d%n", prva, druga);
 6 }
 7
 8 private static void obradi(Integer a, Integer b) {
 9   if(Math.random() < 0.5) {
10     a = a + (int)(10 * Math.random());
11   } else {
12     b = b + (int)(10 * Math.random());
13   }
14 }
```

Ovaj primjer, dakako, neće ispisati ništa drugo doli 5 i 8 za varijable u pitanju. Razlog leži u činjenici da smo metodi poslali kopije referenci na dva objekta tipa `Integer`. Metoda je promijenila jednu od te dvije kopije - ali original (lokalna varijabla u metodi `main`) te promjene ne vidi; metoda `obradi` nije dobila adresu referenci `prva` i `druga` kako bi u memoriji mogla promijeniti kuda te reference pokazuju.

Uočite također da problem nismo mogli riješiti preko povratne vrijednosti metode: povratna vrijednost može vratiti samo jedan podatak.

Stoga se kao tipično praktično rješenje nameće izrada pomoćnog razreda koji će podatak koji želimo da metode mogu mijenjati enkapsulirati kao podatkovnu člansku varijablu. Takav će razred uobičajeno ponuditi mogućnost inicijalizacije te članske varijable preko konstruktora te ponuditi prikladne *gettere* i *settere* kojima je omogućen dohvat i modificiranje sadržaja te članske varijable. Takav razred uobičajeno nazivamo *omotač* (engl. *wrapper*).

Primjer 9.5 ilustrira jedan takav omotač za podatak tipa `Integer`. Razred smo prikladno nazvali `IntWrapper`.

Razred deklarira jednu člansku varijablu `value` tipa `Integer`, definira konstruktor koji prima inicijalnu vrijednost te članske varijable, definira metodu `getValue` kojom je omogućen dohvat sadržaja te varijable, metodu `setValue` kojom je omogućena promjena sadržaja te varijable te metodu `toString` koja sadržaj varijable vraća kao `String`.

Primjer 9.5. Omotač podatka tipa `Integer`

```
 1 package book.java.generics;
 2
 3 public class IntWrapper {
 4  private Integer value;
 5
 6  public IntWrapper(Integer value) {
 7   this.value = value;
 8  }
 9
10  public Integer getValue() {
11   return value;
12  }
13
14  public void setValue(Integer value) {
15   this.value = value;
16  }
17
18  @Override
19  public String toString() {
20   if (value == null)
21    return "";
22   return value.toString();
23  }
24 }
```

Program koji korektno ostvaruje opisanu obradu uporabom tog razreda prikazan je u primjeru 9.6.

Primjer 9.6. Uporaba omotača podatka tipa `Integer`

```
 1 package book.java.generics;
 2
 3 public class Zamjena {
 4
 5  public static void main(String[] args) {
 6   IntWrapper prva = new IntWrapper(5);
 7   IntWrapper druga = new IntWrapper(8);
 8   obradi(prva, druga);
 9   System.out.format(
10    "Prva: %d, druga: %d%n",
11    prva.getValue(),
12    druga.getValue());
13  }
14
15  private static void obradi(IntWrapper a, IntWrapper b) {
16   if (Math.random() < 0.5) {
17    a.setValue(a.getValue() + (int) (10 * Math.random()));
18   } else {
19    b.setValue(b.getValue() + (int) (10 * Math.random()));
20   }
21  }
22 }
```

Osnovna ideja prikazanog rješenja je da i glavni program i pomoćne metode varijabli pristupaju preko reference na omotač varijable što je zapravo objekt koji živi na gomili u memoriji i na koji i glavni program i pomoćne metode dobivaju referencu.

Ako pišemo program čije metode na prikazani način moraju mijenjati varijable različitih tipova, mogli bismo pripremiti omotače za svaki od tipova koje koristimo. Primjerice, ako trebamo omotač za varijablu tipa `Double`, napisali bismo i koristili razred prikazan u primjeru 9.7.

Primjer 9.7. Omotač podatka tipa `Double`

```
 1 package book.java.generics;
 2
 3 public class DoubleWrapper {
 4  private Double value;
 5
 6  public DoubleWrapper(Double value) {
 7   this.value = value;
 8  }
 9
10  public Double getValue() {
11   return value;
12  }
13
14  public void setValue(Double value) {
15   this.value = value;
16  }
17
18  @Override
19  public String toString() {
20   if (value == null)
21    return "";
22   return value.toString();
23  }
24 }
```

Prisjetite li se načela kojima se trebamo voditi pri oblikovanju programskog rješenja a koja smo naveli u poglavlju 8, trebali bismo se brzo složiti da takav način generiranja mnoštva razreda koji su praktički preslika jedan drugoga nije dobro rješenje. Trebamo li omotače za *n* različitih tipova podataka, morat ćemo izgenerirati upravo *n* novih razreda - a to ne želimo; uvijek moramo težiti višestrukoj iskoristivosti koda.

Moguće rješenje uočenog problema koje nudi višestruku iskoristivost koda prikazano je u primjeru 9.8.

Prikazano rješenje zahtjeva izradu samo jednog razreda omotača koji omogućava omatanje članske varijable koja je bilo kojeg tipa koji je barem `java.lang.Object`. U praksi, to znači da takav omotač može omatati primjerak bilo kojeg razreda čime smo dobili fleksibilno i višestruko iskoristivo rješenje. Međutim, cijenu ćemo ipak platiti na mjestu uporabe. Kako prikazani omotač deklarira da omata `Object`, pri svakom dohvatu sadržaja omotača (ako želimo dalje raditi nešto smisleno), morat ćemo raditi eksplicitno ukalupljivanje; u suprotnom ćemo nad sadržajem moći koristiti samo metode razreda `Object`. To pak predstavlja problem iz dva razloga. Prvo: smanjuje se čitljivost koda što u startu već treba izbjegavati. Drugo: gubimo mogućnost kontrole tipova podataka; okolo se šalju omotači objekata i možemo se samo nadati da će ukalupljivanje uspjeti odnosno da, ako metoda očekuje omotani `String`, da će ga doista i dobiti.

U takvom okruženju izuzetno je lagano raditi pogreške. metode koje očekuju omotače oko konkretnih tipova nemaju mogućnosti to deklarirati a to pak znači da niti prevoditelj to ne može provjeravati prilikom prevođenja koda. Postojanje pogrešaka moći ćemo otkriti tek pri izvođenju koda.

Primjer 9.8. Omotač podatka tipa `Object`

```
1 package book.java.generics;
2
3 public class ObjectWrapper {
4  private Object value;
5
6  public ObjectWrapper(Object value) {
7   this.value = value;
8  }
9
10  public Object getValue() {
11   return value;
12  }
13
14  public void setValue(Object value) {
15   this.value = value;
16  }
17
18  @Override
19  public String toString() {
20   if (value == null)
21    return "";
22   return value.toString();
23  }
24 }
```

Glavni program koji koristi omotač `ObjectWrapper` prikazan je u primjeru 9.9.

Primjer 9.9. Uporaba omotača podatka tipa `Object`

```
1 package book.java.generics;
2
3 public class Zamjena {
4
5  public static void main(String[] args) {
6   ObjectWrapper prva = new ObjectWrapper(Integer.valueOf(5));
7   ObjectWrapper druga = new ObjectWrapper(Integer.valueOf(8));
8   obradi(prva, druga);
9   System.out.format(
10    "Prva: %d, druga: %d%n",
11    prva.getValue(),
12    druga.getValue());
13  }
14
15  private static void obradi(ObjectWrapper a, ObjectWrapper b) {
16   if (Math.random() < 0.5) {
17    a.setValue(
18     (Integer)a.getValue() + (int) (10 * Math.random())
19    );
20   } else {
21    b.setValue(
22     (Integer)b.getValue() + (int) (10 * Math.random())
23    );
24   }
25  }
26 }
```

Kako u metodi `main` varijeble koristimo samo za ispis u metodi `format` koja prima objekte, tu smo mogli izostaviti ukalupljivanje. Međutim, u metodi `obradi` to nije bilo moguće. Da smo u metodi `main` napisali sljedeći isječak koda:

```
1 ObjectWrapper a = new ObjectWrapper(new Integer(5));
2 ObjectWrapper b = new ObjectWrapper("10");
3 obradi(a,b);
```

on bi se bez ikakvih problema preveo. Ako baš nemate sreće, kada bismo pogram pokrenuli, on bi se uredno izvršio. Ako imate sreće, program bi se raspao pa biste postali svjesni da tu nešto nije dobro. O čemu se točno radi? Onaj dio: "ako baš nemate sreće" znači da je generator slučajnih brojeva u metodi `obradi` pri prvom pozivu generirao vrijednost koja je manje od 0.5 pa smo završili u prvoj grani uvjeta gdje objekt iz prvog omotača ukalupljujemo u `Integer` - i to će sve proći OK. Nezgodno je međutim ako završimo u drugoj grani: tada ćemo `String` pokušati ukalupiti u `Integer` i tu će izvođenje programa puknuti. Ovo je klasičan primjer nedeterminističke greške - ponekad se javi i sruši program a ponekad ne, čime je testiranje takvog koda dosta nezahvalno. Grešku smo pri tome napravili u metodi `main` gdje smo metodu `obradi` pozvali uz reference na dva omotača od koji oba ne omataju `Integer`. Bilo bi krasno kada bi nas prevoditelj već na tom mjestu mogao upozoriti na pogrešku.

Možda se na prvu ruku čini da rješenje koje smo opisali, iz istaknutih razloga, nije dobro. Istina je, međutim, da smo napravili korak u pravom smjeru: umjesto generiranja mnoštva razreda omotača, prikazali smo jedan koji je višestruko iskoristiv. Sada samo moramo u rješenje vratiti mogućnost kontrole ispravnosti tipova podataka - i to je upravo idealno za parametrizaciju. Umjesto općenitog razreda `ObjectWrapper` koji eksplicitno navodi da radi s podatkom tipa `java.lang.Object`, napravit ćemo parametrizirani omotač koji će tip podatkovne članske varijable dobiti kao parametar razreda. Rješenje je prikazano u primjeru 9.10.

Primjer 9.10. Parametrizirani omotač

```
 1 package book.java.generics;
 2
 3 public class Wrapper<T> {
 4  private T value;
 5
 6  public Wrapper(T value) {
 7   this.value = value;
 8  }
 9
10  public T getValue() {
11   return value;
12  }
13
14  public void setValue(T value) {
15   this.value = value;
16  }
17
18  @Override
19  public String toString() {
20   if (value == null)
21    return "";
22   return value.toString();
23  }
24 }
```

Prikazani omotač definira jedan parametar koji ćemo koristiti kao podatkovni tip članske varijable koju omata. Prilikom stvaranja primjeraka parametriziranih razreda, koristi se sintaksa koja zahtjeva da se parametriziranost istakne i kod deklaracije članske varijable (lijeva strana u sljedećem primjeru) i pri pozivu konstruktora (desna strana u sljedećem primjeru).

```
1 Wrapper<Integer> a = new Wrapper<Integer>(new Integer(5));
```

Od Jave 8, kako bi se smanjio broj znakova koje je potrebno utipkati (a time i u određenoj mjeri povećava čitljivost), na desnoj strani se može izostaviti navođenje vrijednosti parametra (ali ne i zagrade) - prevoditelj će temeljem deklariranog tipa s lijeve strane znaka jednakosti zaključiti što je trebalo pisati u zagradama na desnoj strani. Takav poziv tada izgleda ovako:

```
1 Wrapper<Integer> a = new Wrapper<>(new Integer(5));
```

pri čemu se `<>` prikazan na desnoj strani naziva *diamond-operator*. Uporaba tehnologije *Java Generic* sada nam ponovno omogućava kontrolu ispravnosti tipova tijekom prevođenja programa. Pogledajte ponovno primjer 9.10: deklarirali smo da je razred parametriziran parametrom `T` i da konstruktor prima jedan argument koji je upravo tog tipa. Stoga u prethodnom isječku koda prevoditelj na temelju lijeve strane zaključuje da se parametar veže na tip `Integer` pa u pozivu konstruktora zapravo piše `new Wrapper<Integer>(...)` a tada ono što se predaje kao argument mora biti nešto što je `Integer`. Iz tog razloga, za sljedeći isječak prevoditelj će prijaviti pogrešku:

```
1 Wrapper<Integer> a = new Wrapper<>("10");
```

jer je kao argument očekivan `Integer` a predan je `String`; dakle, kontrola ispravnosti tipova podataka ponovno radi! I ne samo to. Pogledajte sljedeći isječak.

```
1 Wrapper<Integer> a = new Wrapper<>(5);
```

Ovo je sasvim legalno - prevoditelj može zaključiti da se kao argument konstruktora očekuje primjerak razreda `Integer`, vidi da je predan podatak koji je primitivnog tipa `int` i stoga automatski generira *autoboxing* (kao što to radi i na svim ostalim mjestima) čime napisani kod automatski prije generiranja strojnog koda prevodi u:

```
1 Wrapper<Integer> a = new Wrapper<>(Integer.valueOf(5));
```

Pogledajmo još jedan primjer.

```
1 Wrapper<Integer> broj1 = new Wrapper<>(5);
2 Wrapper<Integer> broj2 = new Wrapper<>(10);
3 Wrapper<String> grad = new Wrapper<>("Zagreb");
4 System.out.println(broj1.getValue().doubleValue());
5 Integer suma = broj1.getValue() + broj2.getValue();
6 System.out.println(grad.getValue().substring("1"));
7 //System.out.println(broj1.getValue().substring(1));
```

Primjer deklarira i inicijalizira tri omotača: dva na podatke tipa `Integer` i jedan na podatak tipa `String`. Redak 4 potom nad prvim omotačem poziva metodu `getValue()` i nad povratnom vrijednosti te metode dalje poziva metodu `doubleValue()`. Budući da prevoditelj iz deklaracije varijable `broj1` vidi da je za nju parametar vezan na tip `Integer`, te da poziv metode `getValue()` vraća objekt čiji je tip određen vezanim parametrom (u ovom primjeru na `Integer`), nad povratnom vrijednosti metode `getValue()` mogu se (bez potrebe za ukalupljivanjem) pozivati sve metode razreda `Integer`, a `doubleValue()` je jedna od njih.

Redak 5 ilustrira *deboxing* i *boxing* na djelu. Prevoditelj na desnoj strani zaključuje da `broj1.getValue()` i `broj2.getValue()` vraćaju objekte tipa `Integer` te da ih treba zbrojiti. Stoga generira dodatni kod koji iz oba objekta dohvaća pohranjenu primitivnu vrijednost (*deboxing*), prepisujući taj kod (oko operatora zbrajanja) ovako:

```
1 broj1.getValue().intValue() + broj2.getValue().intValue()
```

nakon čega je rezultat primitivni `int`. Međutim, kako se rezultat pridjeljuje varijabli tipa `Integer`, ponovno ga treba pretvoriti u objekt (*boxing*), pa se dodaje poziv `Integer.valueOf(...)` čime se cijeli kod prevodi u konačni isječak:

```
1 Integer broj3 = Integer.valueOf(
2   broj1.getValue().intValue() + broj2.getValue().intValue()
3 );
```

Redak 6 primjera ilustrira da se nad povratnom vrijednosti `grad.getValue()` mogu bez eksplicitnog ukalupljivanja pozivati metode razreda `String`. Na ovom mjestu prevoditelj zna da ono što metoda `getValue()` vrati mora biti nešto što je `String` pa dopušta takav poziv. Redak 7 je zakomentiran jer prikazuje kod koji je pogrešan (i doista, generira pogrešku koju će prevoditelj prijaviti već prilikom prevođenja, ako redak otkomentirate): prijavljena pogreška će biti da razred `Integer` (što prevoditelj zaključuje da je tip povratne vrijednosti od `broj1.getValue()`) nema metodu `substring(...)`, što je, složit ćete se, korektan zaključak.

Primjer 9.11 ilustrira uporabu parametriziranog omotača.

Primjer 9.11. Uporaba parametriziranog omotača

```
1 package book.java.generics;
2
3 public class Zamjena {
4
5  public static void main(String[] args) {
6   Wrapper<Integer> prva = new Wrapper<>(Integer.valueOf(5));
7   Wrapper<Integer> druga = new Wrapper<>(8);
8   obradi(prva, druga);
9   System.out.format(
10    "Prva: %d, druga: %d%n",
11    prva.getValue(),
12    druga.getValue());
13  }
14
15  private static void obradi(
16    Wrapper<Integer> a, Wrapper<Integer> b) {
17   if (Math.random() < 0.5) {
18    a.setValue(
19     a.getValue() + (int) (10 * Math.random())
20    );
21   } else {
22    b.setValue(
23     b.getValue() + (int) (10 * Math.random())
24    );
25   }
26  }
27 }
28
```

Ovaj primjer ujedno pokazuje i kako metoda može definirati da očekuje reference na razrede koji su parametrizirani na točno određen način. Pogledajte u retku 15 kako su deklarirani argumenti koje prima metoda: oba moraju biti primjerci razreda `Wrapper` koji su pri stvaranju bili parametrizirni tipom `Integer`; drugim riječima, metoda očekuje dva omotača oko primjeraka razreda `Integer`. Pokušate li metodu pozvati s omotačem oko nekog drugog tipa (primjerice, stringa), već će prevoditelj na mjestu takvog poziva prijaviti pogrešku.

Složeniji primjer

U nastavku ćemo pokazati još jedan primjer koji ilustrira definiranje i uporabu parametriziranog sučelja te definiranje razreda koji implementira takvo sučelje.

Primjer 9.12. Parametrizirano sučelje

```
 1 package book.java.generics;
 2
 3 public interface KeyValue<K, V> {
 4
 5  public K getKey();
 6  public V getValue();
 7  public void setValue(V value);
 8
 9 }
10
```

Primjer 9.13. Prva implementacija parametriziranog sučelja

```
 1 package book.java.generics;
 2
 3 public class Square implements KeyValue<Integer, Integer> {
 4
 5  private Integer n;
 6
 7  public Square(Integer n) {
 8   super();
 9   this.n = n;
10  }
11
12  @Override
13  public Integer getKey() {
14   return n;
15  }
16
17  @Override
18  public Integer getValue() {
19   return n*n;
20  }
21
22  @Override
23  public void setValue(Integer value) {
24   throw new UnsupportedOperationException(
25    "Square is read-only."
26   );
27  }
28 }
29
```

Primjer 9.14. Druga implementacija parametriziranog sučelja

```
 1 package book.java.generics;
 2
 3 public class Pair<K,V> implements KeyValue<K, V> {
 4
 5  private K key;
 6  private V value;
 7
 8  public Pair(K key, V value) {
 9   super();
10   this.key = key;
11   this.value = value;
12  }
13
14  @Override
15  public K getKey() {
16   return key;
17  }
18
19  @Override
20  public V getValue() {
21   return value;
22  }
23
24  public void setValue(V value) {
25   this.value = value;
26  }
27 }
28
```

Primjer 9.15. Uporaba parametriziranog sučelja

```
 1 package book.java.generics;
 2
 3 public class Primjeri {
 4
 5  public static void main(String[] args) {
 6   KeyValue<Integer, Integer> kvadrat = new Square(4);
 7   System.out.format(
 8    "%d^2 = %d%n",
 9    kvadrat.getKey(),
10    kvadrat.getValue()
11   );
12
13   KeyValue<Integer, String> posta =
14    new Pair<>(10000, "Zagreb");
15   KeyValue<String, String> suprug =
16    new Pair<>("Jasna", "Ivo");
17  }
18
19 }
20
```

Sučelje `KeyValue` (primjer 9.12) modelira jedan uređeni par. Prvi element tog para zovemo *ključ* a drugi *vrijednost*. Tip ključa i vrijednosti može biti različit pa smo sučelje parametrizirali s dva parametra: parametar `K` predstavlja tip ključa a parametar `V` tip vrijednosti.

Jednu implementaciju ovog sučelja predstavlja razred `Square` koji nije parametriziran i koji prilikom implementacije fiksira parametre sučelja (oba na `Integer`); vidi primjer 9.13). Razred kroz konstruktor prima jedan broj i tretira ga kao ključ. Metoda `getValue()` vraća broj koji je kvadrat pohranjenog ključa. Metoda `setValue(...)` izaziva iznimku `UnsupportedOperationException`. Uočite da su prilikom implementacije metoda sučelja u ovom razredu parametri već zamijenjeni fiksiranim tipovima. U deklaraciji razreda `Square` rekli smo da implementira sučelje `KeyValue<Integer,Integer>`. Stoga znamo da su povratne vrijednosti od `getKey` i od `getValue` te argument od `setValue` tipa `Integer` i to eksplicitno piše u izvornom kodu.

Razred `Pair` (primjer 9.14) predstavlja parametrizirani razred koji implementira sučelje `KeyValue`. Taj je razred parametriziran s dva parametra i deklarira da implementira sučelje `KeyValue` s ta dva ista parametra. U ovom slučaju, konkretne vrijednosti tih parametara zadat će programer pri svakom stvaranju primjeraka ovog razreda. Razred definira dvije članske varijable: jednu u kojoj pamti ključ a drugu u kojoj pamti vrijednost.

Konačno, primjer 9.15 prikazuje uporabu napisanih razreda i sučelja.

Odnosi tipova podataka uz parametrizaciju

Kroz prethodna poglavlja upoznali smo se s objektnim modelom programskog jezika Java i naučili što je stablo nasljeđivanja te koje reference se mogu automatski ukalupljivati u koje reference. Primjerice, neka imamo razred `Osnovni` te neka je razred `Izvedeni` naslijedio razred `Osnovni`. Razmotrite isječak prikazan u nastavku.

```
1 Izvedeni i = new Izvedeni();
2 Osnovni o = i;
3
4 Osnovni o2 = new Osnovni();
5 Izvedeni i2 = o2;
```

Prva dva retka su korektna. Prvi redak definira referencu tipa `Izvedeni` i postavlja je na objekt tipa `Izvedeni` koji stvara konstruktor. Drugi redak je također korektan jer ako razred `Izvedeni` nasljeđuje razred `Osnovni`, tada svaki primjerak razreda `Izvedeni` ima sve što mora imati primjerak razreda `Osnovni` pa na objekt tipa `Izvedeni` možemo gledati kroz "naočale" razreda `Osnovni`.

Redak 4 definira referencu tipa `Osnovni` u postavlja je na objekt tipa `Osnovni` koji stvara konstruktor. Redak 5 nije ispravan. Na primjerak razreda `Osnovni` ne možemo gledati kroz "naočale" razreda `Izvedeni`. To je bio kratak podsjetnik na do sada obrađena pravila. Što se mijenja kada radimo s parametriziranim tipovima?

Pravila su malo drugačija. Nasljeđivanje na razini parametara ne funkcionira. Konkretno, vratimo se na naš parametrizirani omotač. Deklaracije `Wrapper<Object>`, `Wrapper<Number>` i `Wrapper<Integer>` su tri potpuno nepovezana tipa podataka. Ne možemo reći da je tip `Wrapper<Number>` izveden iz tipa `Wrapper<Object>` - to naprosto ne vrijedi. U razloge zašto je to tako nećemo ulaziti. Međutim, ovo je važno zapamtiti kako ne biste došli u napast napisati nešto poput sljedećeg primjera.

```
1 Wrapper<Integer> i = new Wrapper<>(25);
2 Wrapper<Number> n = i;
3 Wrapper<Object> o = i;
```

Drugi i treći redak izazvat će pogrešku prilikom prevođenja.

Ponekad se javlja potreba da dozvolite da se referenca na primjerak parametriziranog razreda postavi na primjerak razreda čiji je parametar u nekom odnosu sa zadanim parametrom. U tom slučaju moguće je koristiti ograde (`extends` te `super`). Primjer koji je korektan, prikazan je u nastavku.

```
1 Wrapper<Integer> i = new Wrapper<>(25);
2 Wrapper<? extends Number> n = i;
3 Wrapper<? extends Object> o = i;
```

Kako se radi o deklaraciji varijable, na ovom mjestu ne možemo deklarirati novi parametar pa koristimo sintaksu sa znakom "?": kažemo da je `n` tipa `Wrapper` čiji je (neimenovani) parametar postavljen na nešto što je barem `Number`. Kako je s desne strane znaka jednakosti varijabla `i` koja je tipa `Wrapper<Integer>` i kako `Integer` je izveden iz `Number`, ovakvo dodjeljivanje je legalno.

Na ova ograničenja treba paziti prilikom pisanja (i korištenja) parametriziranog koda. Primjerice, napišete li metodu `obradi` ovako:

```
1 private static void obradi(
2   Wrapper<Number> a, Wrapper<Number> b) {
3   ...
4 }
```

metodu ćete moći pozivati s argumentima koji su isključivo tipa `Wrapper<Number>`; ako ste tip parametra postavili na `Number` jer ste željeli omogućiti višestruku iskoristivost koda (u smislu da metodu možete pozvati s omotačem na bilo što što je barem `Number`) - to niste postigli. Umjesto toga, deklarirajte parametar s ograničenjem:

```
1 private static void obradi(
2   Wrapper<? extends Number> a, Wrapper<? extends Number> b) {
3   ...
4 }
```

Nakon što smo ilustrirali odnos tipova koji (možda suprotno očekivanjima) ne postoji, pokažimo sada dio koji postoji. Imate li dva parametrizirana razreda koji (kada biste zanemarili parametre) jesu u odnosu, onda u tom odnosu ostaju ako su im parametri postavljeni na isti tip. Zvuči komplicirano, pa evo primjera. Vratimo se na naša dva razreda `Osnovni` i `Izvedeni`, i pretpostavimo da su parametrizirani i definirani ovako:

```
1 class Osnovni<T> { ... }
2
3 class Izvedeni<T> extends Osnovni<T> {...}
```

Uz istu parametrizaciju, očuvan je odnos između tipova: `Izvedeni<Integer>` je podtip od `Osnovni<Integer>`, ali nije podtip od `Izvedeni<Number>` ili od `Osnovni<Number>`. Evo primjera:

```
1 Izvedeni<Integer> izv = new Izvedeni<>();
2 Osnovni<Integer> ref1 = izv;
3 Izvedeni<Number> ref2 = izv;
4 Osnovni<Number> ref3 = izv;
```

Prvi i drugi redak su korektni, treći i četvrti generiraju pogrešku pri prevođenju.

Dodatna ograničenja pri radu s parametriziranim tipovima

U programskom jeziku Java dozvoljeno je deklariranje polja parametriziranih tipova ali ne i njihovo stvaranje. Konkretno, sljedeći isječak koda je korektan:

```
1 Wrapper<String>[] polje = null;
```

ali sljedeći zato nije:

```
1 Wrapper<String>[] polje2 = new Wrapper<String>[20];
```

Problem je u desnoj strani gdje operatorom `new` pokušavamo stvoriti polje referenci parametriziranog tipa što jezik ne dozvoljava. I opet, postoje razlozi zašto je tome tako ali ovdje nećemo ulaziti u te detalje. S obzirom da je nekada doista potrebno stvoriti polje parametriziranog tipa, to možemo napraviti zaobilaznim putem: stvorimo polje na način da se pravimo da razred nije parametriziran i potom koristimo eksplicitno ukalupljivanje. Evo rješenja.

```
1 Wrapper<String>[] polje3 = (Wrapper<String>[])new Wrapper[20];
```

Kako smo time najprije odustali od parametrizacije (pa prevoditelj više ne zna kojeg su tipa parametri) i potom radimo eksplicitno ukalupljivanje, prevoditelj će generirati upozorenje s obrazloženjem da na ovom mjestu on ne zna je li takvo ukalupljivanje korektno (jer ne može utvrditi vrijednost parametra za polje); radi se o upozorenju tipa *Unchecked cast from Wrapper[] to Wrapper<String>[]*. Da bismo utišali to upozorenje, naredbu možemo označiti s `@SuppressWarning("unchecked")`. Konačno rješenje je prikazano u nastavku.

```
1 @SuppressWarnings("unchecked")
2 Wrapper<String>[] polje3 = (Wrapper<String>[])new Wrapper[20];
```

Treba napomenuti da ovakva rješenja nije dobro imati posvuda po kodu. Ako je u program potrebno ponuditi ovakvu funkcionalnost, onda je treba izolirati na jedno mjesto.

Detaljnije od prethodno napisanoga u ovoj knjizi nećemo ulaziti u ovu tehnologiju. Pokrili smo njezine najvažnije elemente što je dovoljno za razumijevanje svega što slijedi u preostalim poglavljima. Za dublji tretman i više informacija čitatelj se upućuje na službenu dokumentaciju Jave.

Poglavlje 10. Kolekcije

Kroz sljedećih nekoliko kraćih poglavlja pozabavit ćemo se Javinim okvirom kolekcija (*Java Collection Framework*). U ovom uvodnom poglavlju razmotrit ćemo kako Javin okvir kolekcija definira pojam *kolekcije* te kako ga modelira. Potom ćemo se u poglavljima koja slijede osvrnuti na različite vrste kolekcija.

Motivacija

Kad pišemo računalni program, najčešće ga pišemo s namjerom da obrađuje podatke. Ovisno o složenosti obrade ali i vrsti podataka, ponekad su nam dovoljni osnovni primitivi koje nudi jezik: varijable različitih tipova te eventualno polja. Vrlo često, to međutim nije slučaj. Stoga se u okviru svih kolegija koji ljude uče programirati obrađuju i teme koje se bave različitim strukturama podataka. Spomenimo samo neke od njih: različite vrste ulančanih lista (jednostruko povezane, dvostruko povezane, cirkularne, ...), binarna (i rjeđe `n`-arna) stabla (često s dodatnom algoritmikom koja ih pokušava održavati barem balansiranima ako ne i potpunima), stogovi, redovi, gomile, ...

Razumjevanje ovakvih implementacijskih detalja važno je kako bi korisnik imao dobru predodžbu o računskim složenostima izvođenja podržanih operacija (ubacivanje novog elementa, traženje određenog elementa, uklanjanje elementa, itd). Međutim, na prvu može malo "zamagliti" pogled. Naime, u prethodnom popisu struktura podataka namjerno su pomiješani dvije vrste pojmova: implementacijski detalji (struktura podataka) te funkcionalnost koja se nudi korisniku (red, stog, ...). Pogledajmo na trenutak stog: to je vrsta kolekcije koju možemo implementirati na različite načine. Primjerice, ako znamo maksimalnu veličinu stoga, isti možemo implementirati nad običnim poljem. Ako je potrebno podržati stog varijabilne veličine, isti možemo ostvariti uporabom ulančane liste. Funkcionalno, sve su to stogovi i korisnik ih može koristiti na jednak način. Implementacijski, radi se o različitim potpornim strukturama nad kojima je izgrađena takva funkcionalnost. Ova će distinkcija biti centralna u Javinom okviru kolekcija.

Stavimo se sada u ulogu malog Javka koji pokušava definirati pojam kolekcije. Kako bi on to napravio? Javko bi sigurno zaključio da je kolekcija objekt koji grupira više elemenata - nekih drugih objekata. Stoga kolekcije još nazivamo i *spremnicima* odnosno engl. *containers*. Kolekcije vrlo često gradimo kako bismo omogućili efikasniji prijenos podataka kroz program; jednostavnije je u neki podprogram poslati referencu na kolekciju nego same pojedinačne podatke. Tipične operacije koje očekujemo od kolekcije su mogućnost pohrane podataka, dohvata podataka te manipuliranje nad pohranjenim podatcima.

Uzevši to u obzir, Javko bi kolekciju mogao definirati kao razred prikazan u nastavku.

```
abstract class Kolekcija {
      void dodajPodatak( podatak ) {...}
      podatak dohvatiPodatak( kriterij ) {...}
      void zamijeniPodatke( kriterij1, kriterij2 ) {...}
      void ukloniPodatak( kriterij ) {...}
}
```

Potom bi, oslanjajući se na mogućnost nasljeđivanja, mali Javko iz ovog razreda izveo razrede koji predstavljaju specifičnije implementacije kolekcija. Što je svakako korak u pravom smjeru. Međutim, zavirimo li u izvorni kod Javinog okvira kolekcija, vidjet ćemo da je tamo odabran drugačiji pristup.



Bilješka

U Javi, kolekcije su definirane preko sučelja.

Zašto je sučelje ovdje "bolje" od apstraktnog razreda? Aprstraktni razred je razred: ako razred naslijedi njega, neće moći naslijediti ništa drugo što bi nam moglo omogućiti postizanje bolje

organizacije koda. S druge strane, u ovom našem primjeru, mali Javko neće moći baš puno konkretnog koda napisati u potpuno apstraktnom razredu koji definira najopćenitiju moguću kolekciju; taj će razred prije biti popis metoda koje moraju postojati u bilo čemu za što tvrdimo da je kolekcija. Naoružan tom spoznajom, mali Javko sada može korigirati svoj prijedlog kolekcije, kako je prikazano u nastavku.

```
interface Kolekcija {
      void dodajPodatak( podatak ) {...}
      podatak dohvatiPodatak( kriterij ) {...}
      void zamijeniPodatke( kriterij1, kriterij2 ) {...}
      void ukloniPodatak( kriterij ) {...}
}
```

Sada je jednostavno napisati različite konkretne implementacije ovog sučelja. Primjerice, ako radimo s malom količinom podataka (odnosno imamo dovoljno memorije za njihovu pohranu), možemo napisati implementaciju koja sve podatke čuva u memoriji, kako je prikazano u nastavku.

```
class MemorijskaKolekcija implements Kolekcija {
      void dodajPodatak( podatak ) {...}
      podatak dohvatiPodatak( kriterij ) {...}
      void zamijeniPodatke( kriterij1, kriterij2 ) {...}
      void ukloniPodatak( kriterij ) {...}
}
```

Potom možemo pristupiti pisanju programa zbog kojeg smo uopće krenuli modelirati potrebne kolekcije, napisati sve potrebne metode, obradu podataka i konačno rješenje prezentirati klijentu koji je naručio program. Danas, sutra, ako količina podataka naraste, program možemo relativno jednostavno nadograditi tako da podrži rad s količinama podataka koje odjednom ne stanu u memoriju: napravit ćemo novu implementaciju sučelja kako je prikazano u nastavku. Ako smo pametno organizirali program, onda imamo na samo jednom mjestu kod koji odabire implementaciju kolekcije koju dalje šaljemo u program, i na tom mjestu dovoljno je odabrati novonapisanu implementaciju. Ako smo pisali ostatak koda tako da ovisi o definiranom sučelju, a ne o konkretnim implementacijama, sve ostalo moralo bi raditi bez ikakvih dodatnih intervencija u kod.

```
class DiskKolekcija implements Kolekcija {
      void dodajPodatak( podatak ) {...}
      podatak dohvatiPodatak( kriterij ) {...}
      void zamijeniPodatke( kriterij1, kriterij2 ) {...}
      void ukloniPodatak( kriterij ) {...}
}
```

Opisani pristup direktno je ostvarenje *načela nadogradnje bez promjene*. Svu kompleksnost ove nove implementacije od ostatka koda sakrit ćemo u razred `DiskKolekcija` pa ostatak koda toga neće biti svjestan. Naravno, performanse će sada biti nešto lošije: nije isto podatke dohvaćati s diska ili iz memorije. Program će, međutim, neovisno o implementaciji kolekcije korektno obavljati svoju funkcionalnost. Štoviše, umjesto da sami pišemo implementacije svih operacija i čitanje i pisanje s diska, možemo se poslužiti gotovim komponentama koje to rade vrlo efikasno i pouzdano. Primjerice, relacijskom bazom podataka. Pa bismo tako mogli napraviti novu implementaciju prikazanu u nastavku koja "ispod haube" koristi relacijsku bazu podataka, indekse za ubrzavanje operacija filtriranja i dohvata i slično - i to sve bez ikakvih daljnjih intervencija u ostatak programa koji se bavi konkretnom obradom podataka.

```
class RelBazaKolekcija implements Kolekcija {
      void dodajPodatak( podatak ) {...}
      podatak dohvatiPodatak( kriterij ) {...}
      void zamijeniPodatke( kriterij1, kriterij2 ) {...}
      void ukloniPodatak( kriterij ) {...}
}
```

Navedimo sada na jednom mjestu koje su prednosti definiranja kolekcija kroz sučelja.

- Korisniku se omogućava definiranje različitih implementacija istog sučelja. Svaka od implementacija može biti prikladna za različite scenarije uporabe. Različite implementacije mogu nuditi različite operacije uz različite računske složenosti.
- Omogućava se lagana reimplementacija, ako se korištena pokaže lošom. Izolira se mjesto na kojem je potrebno raditi korekcije.
- Omogućava se izrada *generičkih algoritama* koji ne ovise o konkretnoj implementaciji već mogu raditi s različitim implementacijama (čak i implementacijama koje u trenutku pisanja algoritma ne postoje već će biti naknadno napisane). To je vrlo važna ideja koja je izravno u skladu s načelom nadogradnje bez promjene.

Elementi Javinog okvira kolekcija

Javin okvir kolekcija izgrađen je od tri komponente: sučelja, implementacija te algoritama.

Sučelja se koriste za definicije svih vrsta kolekcija. Sučelja služe kao ugovori između korisnika kolekcije i implementatora konkretne kolekcije; definiraju popise metoda koje pojedine vrste kolekcija moraju ponuditi kao i njihovu semantiku.

Implementacije predstavljaju konačne razrede s kojima krajnji korisnik barata. Uobičajeno je da za većinu sučelja postoji više od jedne implementacije koje u pozadini koriste različite strukture podataka i time korisniku nude različite složenosti obavljanja određenih operacija.

Generički algoritmi koji su napisani u okviru Javinog okvira kolekcija omogućavaju korisnicima učinkovito pisanje koda: postoje gotovi algoritmi za niz često korištenih operacija koji su napisani vrlo kvalitetno. Njihovom uporabom maksimalno se potiče višestruko korištenje koda, umanjuje se mogućnost unosa pogrešaka u kod i smanjuje broj linija koda koje korisnik mora napisati kako bi riješio problem zbog kojeg piše kod. S obzirom da su ovi algoritmi pisani vrlo pažljivo, postižu odlične performanse i mala je vjerojatnost da bi korisnik sam mogao napisati nešto kvalitetnije.

Dio generičkih algoritama smješten je u razrede `java.util.Arrays` i `java.util.Collections`. Ti su razredi pravo mjesto za potražiti implementacije algoritama sortiranja, binarnog pretraživanja, generiranja slučajnih permutacija elemenata te niza drugih.

Spomenimo na ovom mjestu još jednu često korištenu operaciju nad bilo kojom kolekcijom: iteriranje nad njenim elementima. Pod pojmom *iteriranje* podrazumjevamo postupak obilaska svih elemenata kolekcije. Razmislite li, prisjetit ćete da su "školski" načini za obilaske elemenata različitih podatkovnih struktura različiti: elemente polja uobičajeno posjećujemo petljom `for` koja ažurira indeks trenutnog elemenata, elemente liste posjećujemo petljom `while` koja ažurira pokazivač na trenutni čvor i radi tako dugo dok postoji sljedeći element, a elemente stabla uobičajeno posjećujemo rekurzivnim obilaskom. U okviru Javinog okvira kolekcija jedan od postavljenih ciljeva bio je omogućiti *uniforman* način obilaska elemenata kroz bilo kakvu implementaciju kolekcije. Ideja je da korisnik ne zna ništa o implementacijskim detaljima kolekcije već da nad bilo kojom kolekcijom može zatražiti objekt *iterator* koji sam zna kako interno dohvaćati elemente a korisniku nudi kroz vrlo jednostavno sučelje mogućnost slijednog prolaza kroz sve elemente kolekcije. Konceptualno, takvo bi sučelje moralo ponuditi metodu kojom korisnik može pitati ima li sljedećeg elementa te metodu kojom dohvaća sljedeći element. U tom slučaju, za proizvoljnu kolekciju korisnik bi mogao napisati sljedeći (pseudo)kod kojim bi prošao kroz sve elemente kolekcije i obradio ih.

```
Kolekcija kolekcija = dohvatiNekuKolekciju();
Iterator iter = kolekcija.dohvatiIterator();
while(iter.postojiSljedeci()) {
  obradi( iter.dohvatiSljedeci() );
}
```

S obzirom da se elementi kolekcija dohvaćaju na različite načine (ovisno o podatkovnim strukturama koje koriste same implementacije kolekcija), jasno je da će svaka kolekcija imati drugačiju implementaciju iteratora. Stoga je koncept iteratora u Javi definiran sučeljem `java.util.Iterator`, koje je prikazano u nastavku.

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  default void remove() {...}
  default void forEachRemaining(Consumer<? super E> action) {...}
}
```

Prve dvije metode čine osnovni API koji omogućava iteriranje. Metoda `remove()` nudi mogućnost da se na siguran način obriše element kolekcije na kojem se iterator trenutno nalazi i od Jave 8 ima pretpostavljenu implementaciju koja baca iznimku `UnsupportedOperationException`. Posljednja metoda dodana je u sučelje od Jave 8, ima pretpostavljenu implementaciju i omogućava korisniku da obradu elemenata enkapsulira u objekt koji zadovoljava sučelje `Consumer` i element obrađuje u metodi `accept`; metodi se predaje referenca na takav objekt a ona iteratorom prolazi kroz sve preostale elemente kolekcije i nad svakim poziva metodu `accept` predanog objekta šaljući kao argument trenutni element. Primjer koji ovo ilustrira dat ćemo malo kasnije.

Uz ovakvu definiciju iteratora, osiguramo li mehanizam kojim bismo od svake kolekcije mogli dohvatiti iterator, imamo sve što nam treba za pisanje generičkog koda obilaženja elemenata koji ne ovisi o samoj kolekciji. Ovaj posljednji korak rješava sučelje `java.lang.Iterable` prikazano u nastavku.

```
interface Iterable<T> {
  Iterator<T> iterator();
  default void forEach(Consumer<? super T> action) { ... }
  default Spliterator<T> spliterator() { ... }
}
```

Sve Javine kolekcije bit će iterabilni objekti: primjerci razreda koji implementira sučelje `java.lang.Iterable` i koji svojim korisnicima nudi mogućnost dohvaćanja iteratora pozivom metode `iterator()`. Samo sučelje definira još dvije metode za koje daje pretpostavljene implementacije. Metoda `forEach` ima sličnu namjenu kao i prethodno spomenuta metoda `forEachRemaining` iteratora: omogućava enkapsulaciju koda koji radi obradu u zasebni objekt, pozivom metode `iterator()` nad iterabilnim objektom stvara novi iterator i njime trči kroz sve elemente pri čemu svaki obrađuje predanim objektom. Drugu metodu obradit ćemo u jednom od kasnijih poglavlja.

Jednostavne kolekcije u Javi su izvedene iz sučelja `java.util.Collection` koje nasljeđuje prethodno opisano sučelje `java.lang.Iterable`. Posljedica toga je da su sve jednostavne Javine kolekcije iterabilne. Evo primjera koji to ilustrira.

Primjer 10.1. Izračun sume brojeva

```
 1 package book.java.collections.basic;
 2
 3 import java.util.Arrays;
 4 import java.util.Collection;
 5 import java.util.Iterator;
 6
 7 public class PrimjerIteriranja {
 8
 9  public static void main(String[] args) {
10
11   Collection<Integer> brojevi = Arrays.asList(1, 2, 3, 4, 5);
12
```

```
13    int suma = 0;
14
15    Iterator<Integer> iter = brojevi.iterator();
16    while(iter.hasNext()) {
17     Integer n = iter.next();
18     suma += n;
19    }
20
21    System.out.println("Suma brojeva je: " + suma);
22   }
23 }
24
```

U retku 11 stvaramo kolekciju čiji su elementi cijeli brojevi 1, 2, 3, 4 i 5 (konkretna implementacija koju vraća pozvana metoda ovaj tren nije važna). Zadaća programa je utvrditi koliko iznosi suma svih brojeva koji su u toj kolekciji. Stoga u retku 13 deklariramo pomoćnu varijablu `suma` inicijane vrijednosti `0`, u retku 15 nad kolekcijom pozivamo metodu `iterator()` koja nam vraća referencu na jedan konkretan iterator kojim možemo obići sve elemente kolekcije, i konačno, u retcima 16-19 u petlji `while` tako dugo dok iterator javlja da ima elemenata dohvaćamo sljedeći element i pribrajamo ga u sumu. Naposlijetku, kada je iteriranje gotovo, ispisujemo utvrđenu sumu.

Umjesto da sami eksplicitno tražimo iterator i pozivamo njegove metode, možemo se poslužiti novim sintaksnim konstruktom koji je uveden od Jave 5 i koji omogućava pisanje kraćeg (jasnijeg) koda koji će prevoditelj prilikom prevođenja sam prepisati u prikazani kod koji koristi eksplicitni iterator prije no što nastavi s prevođenjem koda. Radi se o kratkom obliku petlje `for`. Ovaj oblik može se primjeniti nad bilo kojim objektom koji implementira sučelje `Iterable` (i u još nekim situacijama).

Primjer 10.2. Izračun sume brojeva kraćim oblikom petlje `for`

```
 1 package book.java.collections.basic;
 2
 3 import java.util.Arrays;
 4 import java.util.Collection;
 5
 6 public class PrimjerIteriranja2 {
 7
 8  public static void main(String[] args) {
 9
10   Collection<Integer> brojevi = Arrays.asList(1, 2, 3, 4, 5);
11
12   int suma = 0;
13
14   for(Integer n : brojevi) {
15    suma += n;
16   }
17
18   System.out.println("Suma brojeva je: " + suma);
19  }
20 }
21
```



Bilješka

Uvijek koristite kratki oblik petlje `for` jer rezultira kodom koji je pregledniji. Eksplicitni dohvat iteratora koristite tek kada je to doista nužno (jer koristite druge metode poput `remove` ili referencu na iterator šaljete u druge metode).

Osvrnimo se sada malo detaljnije na enkapsulaciju obrade u zasebne razrede. Pretpostavimo da je potrebno nad slijedom cijelih brojeva izračunati njihovu sumu te njihov minimum. Naravno, ostanimo otvoreni i za daljnje nadogradnje -- nikada ne znate što će Vam još u budućnosti zatrebati. Svaki od ovih zadataka možemo modelirati razredom izvedenim iz jednog zajedničkog sučelja: sučelja koje nudi mogućnost dojave svakog novog elementa (a mi ćemo onda ažurirati naše interne podatke). Takvo sučelje već imamo na raspolaganju u paketu `java.util.function` i zove se `Consumer`. To je parametrizirano sučelje koje je prikazano u nastavku.

```
interface Consumer<T> {
  void accept(T t);
  default Consumer<T> andThen(Consumer<? super T> after) { ... }
}
```

Ideja je obradu podataka organizirati na način da objekt elemente dobiva jedan po jedan kroz metodu `accept(t)` svaki puta napravi potrebnu obradu. Primjerice, razred koji bi računao sumu svih elemenata imao bi jednu člansku varijablu `suma` i pri svakom pozivu metode `accept(t)` sumu bi povećao za primljeni broj `t`. Metoda `andThen` koja je također definirana u sučelju omogućava kombiniranje dvije obrade u jednu novu obradu koja za svaki primljeni element slijedno poziva metode `accept(t)` originalnih obrada koje smo kombinirali.

Koristeći ovako definirane obrade, umjesto direktne uporabe iteratora ili petlji `for` odnosno `while`, obradu možemo primijeniti nad kolekcijom oslanjajući se ne metodu `foreach` sučelja `Iterable` koja će sama stvoriti iterator, protrčati kroz elemente i za svaki pronađeni element pozvati metodu `accept(t)` nad predanim objektom koji enkapsulira obradu.

U nastavku je prikazan primjer u kojem definiramo dvije obrade: jednu koja računa sumu elemenata te drugu koja utvrđuje minimalni broj među svim elementima kolekcije.

Primjer 10.3. Primjer enkapsulacije obrada u zasebne razrede

```
 1 package book.java.collections.basic;
 2
 3 import java.util.Arrays;
 4 import java.util.Collection;
 5 import java.util.Optional;
 6 import java.util.function.Consumer;
 7
 8 public class PrimjerIteriranja3 {
 9
10  private static class Sumator implements Consumer<Integer> {
11   private int suma;
12
13   @Override
14   public void accept(Integer t) {
15    suma += t;
16   }
17
18   public int getSuma() {
19    return suma;
20   }
21  }
22
23  private static class Minimum implements Consumer<Integer> {
24   private boolean minimumVazeci = false;
25   private int minimum;
26
27   @Override
28   public void accept(Integer t) {
```

```
29     if(!minimumVazeci || t<minimum) {
30      minimumVazeci = true;
31      minimum = t;
32     }
33    }
34
35    public Optional<Integer> getMinimum() {
36     if(!minimumVazeci) {
37      return Optional.empty();
38     } else {
39      return Optional.of(minimum);
40     }
41    }
42   }
43
44   public static void main(String[] args) {
45
46    Collection<Integer> brojevi = Arrays.asList(1, 2, 3, 4, 5);
47
48    Sumator sumator = new Sumator();
49    brojevi.forEach(sumator);
50
51    Minimum minimum = new Minimum();
52    brojevi.forEach(minimum);
53
54    System.out.println("Suma brojeva je: " + sumator.getSuma());
55
56    Optional<Integer> minimalni = minimum.getMinimum();
57    if(minimalni.isPresent()) {
58     System.out.println("Minimalni broj je: " + minimalni.get());
59    } else {
60     System.out.println("Minimum ne postoji.");
61    }
62   }
63
64  }
65
```

Oslanjajući se na metodu `andThen` koja gradi ulančanu obradu, dio metode `main` mogli smo napisati i kraće, uz samo jedan poziv metode `forEach`, kako je prikazano u nastavku.

```
Sumator sumator = new Sumator();
Minimum minimum = new Minimum();

brojevi.forEach(sumator.andThen(minimum));
```

Naposlijetku, upoznajmo se još s jedim razredom koji je uveden u prethodnom primjeru u okviru ugniježđenog statičkog razreda `Minimum`. Problem s kojim smo se ovdje suočili jest: što vratiti kao rezultat obrade ako metoda `accept(t)` ne bude niti jednom pozvana? Drugim riječima, koji je minimalni element u kolekciji koja je prazna?

Za modeliranje metoda koja moraju vratiti dva podatka odjednom: status postoji li rezultat ili ne, te sam rezultat (ako postoji), od Jave 8 na raspolaganju nam stoji razred `java.util.Optional` koji modelira *opcionalni rezultat*. Razred nema javnih konstruktora već nudi statičke *metode-tvornice* preko kojih možemo stvarati primjerke razreda. Pogledajmo primjer. Metoda koja vraća opcionalni cijeli broj bit će deklarirana ovako:

```
Optional<Integer> metoda() { ... }
```

Korisnik koji poziva tu metodu, zapamtit će vraćenu vrijednost, pozivom metode `isPresent()` provjeriti postoji li rezultat, i ako postoji, dohvatiti ga pozivom metode `get()`. Isječak koda koji to ilustrira prikazan je u nastavku.

```
Optional<Integer> rezultat = metoda();
if(rezultat.isPresent()) {
  Integer broj = rezultat.get();
  System.out.println("Metoda je vratila broj " + broj + ".");
} else {
  System.out.println("Metoda nije vratila broj.");
}
```

U samoj metodi `metoda()`, ako želimo vratiti rezultat, zamotat ćemo ga u primjerak razreda `Optional` statičkom metodom `Optional.of(t)`. Ako metoda ne može vratiti rezultat jer on ne postoji (nema nimimalnog broja u praznoj kolekciji), vratit ćemo primjerak razreda `Optional` dobivenog pozivom statičke metode `Optional.empty()`.



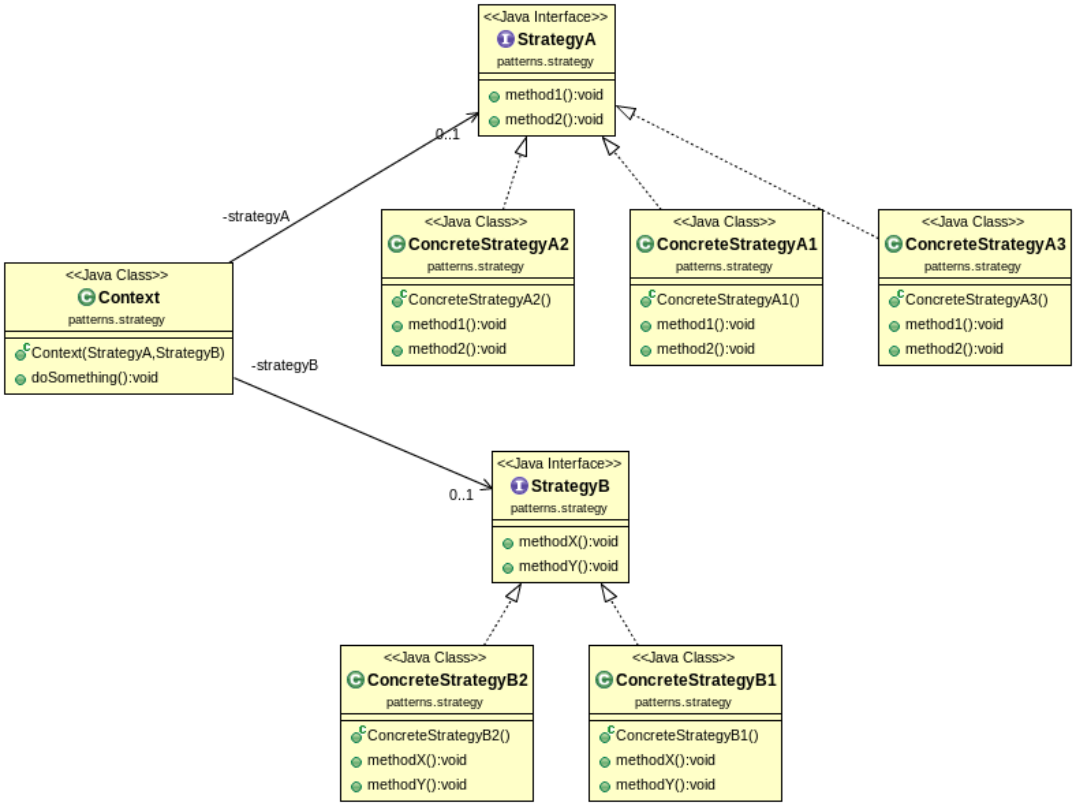
Oblikovni obrazac Strategija

Oblikovni obrazac *Strategija* (engl. *Strategy*) predstavlja jedno moguće rješenje problema izrade više primjeraka koda kod kojih je velik dio jednak a postoje razlike u samo nekim implementacijskim detaljima (koji su tipično negdje duboko u kodu). Ideja ovog obrasca je onaj dio koda koji varira enkapsulirati iza jasno definiranog sučelja, pa za svaki različit dio napisati po jedan razred koji implementira upravo takvo ponašanje. Potom se umjesto više primjeraka koda napiše samo jedan primjerak koji dobiva referencu na primjerak razreda koji implementira konkretno željeno ponašanje. Ima li kod koji pišemo više različitih dijelova koda koji variraju nezavisno, svaki se dio može enkapsulirati u nezavisnu strategiju. Primjer je prikazan na slici 10.1. Razred `Context` prikazan na slici sadrži generički dio koda (metoda `doSomething`) koja varira po dvije osi. Stoga su definirane dvije porodice strategija i za svaku određen broj implementacija. Korisnik na neki način mora kontekstu dostaviti reference na primjerak svake od strategija iz definiranih porodica. Prikazani primjer definira dvije porodice strategija i reference prenosi konstruktorom konteksta.

Sučelje koje smo definirali nazivamo *strategijom* a implementacije tog sučelja *konkretnim strategijama*.

Primjere uporabe ovakvog oblikovnog obrasca vidjeli smo već na mnoštvu mjesta. Posljednji je bila upravo generička implementacija metode `forEach(c)` koju ima svaka kolekcija; štoviše, bilo što, što je iterabilno. Ta metoda sadrži svu potrebnu logiku da bi mogla prolaziti kroz sve elemente proizvoljnog iterabilnog objekta, dohvaćati element po element i raditi obradu elementa. Jedini je problem što metoda ne zna kako da obrađuje elemente. Stoga kao argumente preko sučelja prima referencu na primjerak razreda koji enkapsulira obradu, i za svaki pronađeni element nad tim objektom poziva metodu `accept(e)` šaljući element kao argument. Želimo li mijenjati obrade, dovoljno je napisati novi razred koji implementira zadano sučelje; metodu `forEach` ne moramo mijenjati. Uočite da je ovaj oblikovni obrazac rješenje koje je u skladu s načelom nadogradnje bez promjene te načelom inverzije ovisnosti.

Slika 10.1. Oblikovni obrazac Strategija



Sučelje `java.util.Collection`

Najopćenitija (jednostavna) kolekcija u Javi je modelirana sučeljem `java.util.Collection`. Pri tome, kada kažemo "jednostavna", mislimo na kolekcije koje sadrže jednostavne elemente (objekte) a ne uređene parove i slično. Ovo sučelje prikazano je u nastavku.

```
public interface Collection<E> extends Iterable<E> {

// Osnovne operacije
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); //optional
boolean remove(Object element); //optional
Iterator<E> iterator();

// Grupne operacije
boolean containsAll(Collection <?> c);
boolean addAll(Collection<? extends E> c); //optional
boolean removeAll(Collection<?> c); //optional
boolean retainAll(Collection<?> c); //optional
void clear(); //optional

// Pretvorba u polja
Object[] toArray();
<T> T[] toArray(T[] a);

// Druge operacije (od Jave 1.8)
```

```
default boolean removeIf(Predicate<? super E> filter) { ... }

// i još neke koje preskačemo...
// ...
}
```

Metode grupe *osnovne operacije* predstavljaju temeljni skup metoda koje bilo koja kolekcija mora ponuditi svojim korisnicima. Tako su tu redom metoda koja vraća broj elemenata u kolekciji (`size()`), metoda koja provjerava je li kolekcija prazna (`isEmpty()`), metoda koja provjerava sadrži li kolekcija neki element (`contains(e)`), potom dvije opcionalne metode, jedna koja u kolekciju dodaje predani element (`add(e)`) te druga koja iz kolekcije uklanja predani element (`remove(e)`), i konačno metoda-tvornica za stvaranje iteratora. Metodama koje su označene kao opcionalne dopušteno je da odbiju napraviti svoj posao (primjerice, u slučaju da je konkretna kolekcija nepromjenjiva) ali u tom slučaju moraju izazvati iznimku `UnsupportedOperationException`.

Metode grupe *grupne operacije* omogućavaju agregiranje, ispitivanje i modifikaciju trenutne kolekcije grupom elemenata koji se predaju kao argument u obliku zasebne kolekcije. Tako imamo metodu koja može provjeriti sadrži li trenutna kolekcija sve elemente koje sadrži i predana kolekcija (`containsAll(c)`), potom metodu koja u trenutnu kolekciju dodaje sve elemente predane kolekcije (`addAll(c)`), metodu kojom iz trenutne kolekcije brišemo sve elemente koji su prisutni u predanoj kolekciji (`removeAll(e)`), metodu kojom iz trenutne kolekcije brišemo sve elemente koji nisu u predanoj kolekciji (`retainAll(e)`) te metodu kojom brišemo sve elemente iz kolekcije (`clear()`). Uočimo da su u ovoj grupi posljednje četiri metode opcionalne. Razmislite li malo, ovime je omogućen vrlo impresivan skup podržanih operacija: neovisno o tome je li konkretna implementacija prve kolekcije temeljena na binarnom stablu a druge na dvostruko povezanoj ulančanoj listi, moći ćemo na potpuno uniforman način agregirati njihove elemente i obavljati sve opisane operacije.

Metode grupe *pretvorba u polja* služe za istakanje sadržaja kolekcije u polje.

Konačno, metode grupe *druge operacije* sadrže još ponešto novih metoda dodanih u Javi 1.8. Od tih metoda, istaknut ćemo samo jednu: metodu `removeIf(p)` koja prima referencu na ispitni predikat, trči po svim elementima kolekcije, pozove ispitni predikat sa svakim pronađenim elementom i za svaki element za koji ispitni predikat vrati `true`, metoda briše element iz kolekcije. Ovo je još jedan primjer uporabe oblikovnog obrasca Strategija gdje je napisana generička metoda a odluku o brisanju korisnik izvana može definirati implementiranjem konkretne strategije.

Samo sučelje `Predicate` koje predstavlja apstraktnu strategiju metode `removeIf()` prikazano je u nastavku.

```
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) { ... }
    default Predicate<T> negate() { ... }
    default Predicate<T> or(Predicate<? super T> other) { ... }
    static <T> Predicate<T> isEqual(Object targetRef) { ... }
}
```

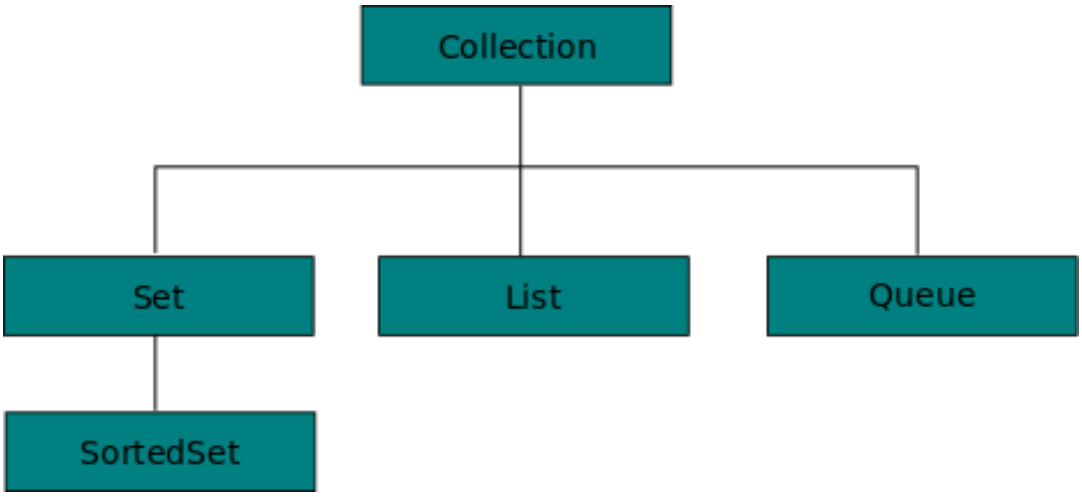
Osnovna metoda ovog sučelja je metoda `test(t)` čija implementacija treba provjeriti predani element `t` i vratiti `true` ako element zadovoljava predikat a `false` inače.

Metode `and(other)` i `or(other)` vraćaju referencu na novi predikat koji je zadovoljen samo kada su zadovoljeni trenutni i predani predikat odnosno trenutni ili predani predikat. Metoda `negate()` vraća novi predikat koji je negacija trenutnog predikata.

Konačno, statička metoda `isEqual(o)` omogućava stvaranje predikata koji će reći `true` samo ako je ispitivani objekt jednak zadanom objektu `o` koji je predan kao argument.

Za kraj ovog poglavlja prikazat ćemo još i važnija sučelja koja su izvedena iz sučelja `Collection` (slika u nastavku).

Slika 10.2. Kolekcije izvedene iz sučelja `Collection`



Sučelje `Set` modelira skupove, sučelje `List` poredane kolekcije a sučelje `Queue` redove. S pojedinim vrstama kolekcija upoznat ćemo se u poglavljima što slijede.

Poglavlje 11. Kolekcije: skup

Terminologijom Javinog okvira kolekcija, *skup* je kolekcija koja ne može sadržavati duplikate. Kolekcije koje su skupovi u Javi su modelirane sučeljem `java.util.Set`. Do Jave 7, sučelje `java.util.Set` bilo je praktički modelirano sljedećim isječkom koda:

```
interface Set extends Collection {
}
```

bez ikakvih novih dodatnih funkcionalnosti koje bi trebale biti navedene; sučelje nije navodilo niti jednu novu metodu. Jedina razika bila je u dokumentaciji ovog sučelja: to je kolekcija koja *ne dozvoljava duplikate elemenata*. Počev od Jave 8, u sučelju `java.util.Set` nalazi se dodatno još samo jedna pretpostavljena metoda.

Konkretne implementacije skupova

Paket java.util nudi tri implementacije sučelja `Set`: razred `HashSet`, razred `TreeSet` te razred `LinkedHashSet`. Svaka od njih ukratko je opisana u nastavku.

`HashSet`

Implementacija koja kao potpornu podatkovnu strukturu koristi tablicu raspršenog adresiranja: elementi se raspršuju u raspoložive pretince tablice. U slučaju da je faktor popunjenosti tablice relativno mali te uz pretpostavku da funkcija za raspršivanje elemenata koja se oslanja na metodu `hashCode()` samih elemenata dobro radi svoj posao, ova implementacija postiže računsku složenosti O(1) za operacije dodavanja, brisanja te provjere postoji li element u kolekciji. Iteratori koje vraća implementacija elemente obilaze po tablici, što znači da im je redoslijed dohvata određen pozicijom gdje su raspršeni. Ovo posljednje je, praktički, jedino negativno svojstvo ove implementacije gdje nam je često poželjno da pri obilasku elemenata imamo neki predvidivi redosljed (tipa: od manjih prema većima, redosljedom umetanja i slično). Implementacija nudi mogućnost zadavanja početnog kapaciteta tablice raspršenog adresiranja te sadrži logiku koja po potrebi dinamički povećava veličinu tablice kako bi osigurala da faktor popunjenosti ne postane prevelik i time ugrozi performanse.

`TreeSet`

Implementacija koja kao potpornu podatkovnu strukturu koristi balansirano binarno stablo: elementi se umeću u čvorove binarnog stabla koje se pokušava održavati balansiranim. Stoga su složenosti operacija umetanja, brisanja te provjeravanja sadrži li kolekcija zadani element jednake O(log N). Iterator koji vraća kolekcija osigurava obilazak elemenata skupa od najmanjeg prema najvećem. Postoje konstruktori koji omogućavaju stvaranje primjeraka kolekcije koji se oslanjaju na *prirodni poredak* elemenata, a postoje i konstruktori koji omogućavaju da korisnik definira način na koji se uspoređuju elementi.

`LinkedHashSet`

Ova implementacija nadogradnja je implementacije ponuđene u okviru razreda `HashSet`: za pohranu elemenata koristi se tablica raspršenog adresiranja integrirana s ulančanom listom koja pamti redosljed kojim su elementi umetani u kolekciju. Zahvaljujući tome, implementacija i dalje nudi složenosti operacija umetanja, brisanja te provjeravanja sadrži li

kolekcija zadani element jednake O(1), a iterator koji vraća kolekcija osigurava obilazak elemenata skupa redosljedom kojim su umetnuti (iterator jednostavno prolazi kroz stvorenu ulančanu listu).

Potrebno je naglasiti da ne podržavaju sve implementacije skupova sve vrste objekata. Primjerice, ako korisnik pokuša u skup dodati element razreda koji nije podržan od izabrane implementacije skupa, moguća je pojava iznimaka tipa `ClassCastException` ili `NullPointerException`. Metode koje rade provjeru sadrži li skup neki element, za elemente nepodržanih razreda mogu izazvati navedene iznimke ili pak mogu jednostavno vratiti `false`.

Zgodno je pogledati i dokumentaciju metode `boolean add(E element);`. Na razini sučelja `Set`, za ovu metodu definirano je sljedeće ponašanje. Ako je po pozivu metode predani element umetnut u skup, metoda će vratiti `true`. Međutim, ako element nije umetnut iz razloga jer već postoji u skupu, metoda će vratiti `false`. Ako je skup nepromjenjiv, poziv metode `add(...)` rezultirat će iznimkom `UnsupportedOperationException`. Ako se element ne može umetnuti u skup jer je nepodržanog razreda, metoda će izazvati `ClassCastException`. Ako je element `null` a kolekcija ne podržava takve elemente, metoda će izazvati `NullPointerException`. Konačno, ako neko drugo svojstvo elementa sprječava njegovo umetanje u skup (možemo zamisliti skup koji dopušta dodavanje samo parnih brojeva, a pokušamo dodati neparni broj), metoda će izazvati `IllegalArgumentException`.

Različite implementacije skupova provjeru sadrži li skup neki element obavljaju na različite načine; međutim, najčešće se postavljaju jasni uvjeti na ponašanje metode `equals(...)` elemenata koji se umeću u kolekciju. Uz ovu metodu, često je potrebno pripaziti i na korektno implementiranu metodu `hashCode()`, a ponekad i na definiranje prirodnog poretka elemenata. Više o ovom posljednjem bit će riječi malo kasnije.

Primjeri uporabe skupova

Tipične primjere uporabe skupova pokazat ćemo u nastavku. Prvi zadatak koji je potrebno riješiti je sljedeći.

> Napišite naredbeno-linijski program koji prima argumente i na zaslon ih ispisuje, ali bez duplikata.

Rješenje je prikazano u primjeru 11.1.

Primjer 11.1. Ispis argumenata programa bez duplikata

```
 1 package book.java.collections.set;
 2
 3 import java.util.Collection;
 4 import java.util.HashSet;
 5 import java.util.LinkedHashSet;
 6 import java.util.Set;
 7 import java.util.TreeSet;
 8
 9 public class IspisBezDuplikata {
10
11  public static void main(String[] args) {
12   System.out.println("Prvi pokušaj:");
13   System.out.println("--------------");
14   ispisi(ukloni1(args));
15
16   System.out.println("Drugi pokušaj:");
17   System.out.println("--------------");
18   ispisi(ukloni2(args));
19
```

```
20    System.out.println("Treći pokušaj:");
21    System.out.println("--------------");
22    ispisi(ukloni3(args));
23   }
24
25   private static Collection<String> ukloni1(String[] args) {
26    return napuniSkup(args, new HashSet<String>());
27   }
28
29   private static Collection<String> ukloni2(String[] args) {
30    return napuniSkup(args, new TreeSet<String>());
31   }
32
33   private static Collection<String> ukloni3(String[] args) {
34    return napuniSkup(args, new LinkedHashSet<String>());
35   }
36
37   private static Set<String> napuniSkup(String[] args,
38    Set<String> set) {
39    for(String s : args) {
40     set.add(s);
41    }
42    return set;
43   }
44
45   public static void ispisi(Collection<String> kolekcija) {
46    for(String s : kolekcija) {
47     System.out.println(s);
48    }
49   }
50
51  }
52
```

Prikazano rješenje organizirano je u tri glavne metode: `ukloni1(...)`, `ukloni2(...)` i `ukloni3(...)`. Svaka od te tri metode prima polje stringova (argumenti naredbenog retka), odabire jednu implementaciju sučelja `Set`, puni je elementima polja i vraća napunjeni skup. Kako ne bismo duplicirali kod, za samo punjenje skupa napisana je pomoćna metoda `napuniSkup` koju koriste sve tri prethodno navedene metode. Metoda `ukloni1(...)` kao implementaciju skupa odabite razred `HashSet`, metoda `ukloni2(...)` kao implementaciju skupa odabite razred `TreeSet` dok etoda `ukloni3(...)` kao implementaciju skupa odabite razred `LinkedHashSet`.

U metodi `main` polje argumenata se prebacuje u skup (na sva tri načina) i potom se ispisuje sadržaj tako dobivenog skupa pripremljenom metodom `ispisi(...)`.

Pokrenemo li program na sljedeći način:

```
D:\javaproj>java java.book.collections.set.IspisBezDuplikata
ovo prvo je jako vazno a i ovo drugo isto
```

dobit ćemo sljedeći ispis.

```
Prvi pokušaj:
--------------
a
isto
ovo
i
```

```
drugo
jako
je
vazno
prvo
Drugi pokušaj:
--------------
a
drugo
i
isto
jako
je
ovo
prvo
vazno
Treći pokušaj:
--------------
ovo
prvo
je
jako
vazno
a
i
drugo
isto
```

Na ispisu je lagano uočiti da je prvi ispis nema nikakav "zanimljiv" poredak: elementi su ispisani redosljedom koji je određen njihovom pozicijom u tablici raspršenog adresiranja nakon raspršivanja. Drugi ispis navodi elemente abecednim redosljedom: za stringove *prirodni poredak* je upravo onaj leksikografski. Konačno, treći ispis navodi elemente upravo redosljedom kojim su dani u naredbenom retku (jer smo ih tim redosljedom dodavali). U sva tri slučaja, naravno, skup je korektno odradio svoj posao izbacivanja duplikata, pa niti u jednom ispisu nemamo “ovo” ispisano više puta.

Osvrnimo se još i na način kako je napisana metoda za ispis elemenata. Sučelje `java.util.Collection` nasljeđuje sučelje `java.lang.Iterable`, a kako je sučelje `java.util.Set` njegova izvedenica, skupovi su iterabilni. Imamo li skup stringova, možemo ga ispisati tako da eksplicitno dohvatimo `Iterator` pozivom metode `iterator()`, i potom u petlji tako dugo dok još ima sljedećeg elementa (`.hasNext()`) dohvaćamo taj element (`.next()`) i ispišemo ga. Ovakav primjer prikazan je u isječku u nastavku.

```
public static void ispisi(Collection<String> kolekcija) {
 Iterator<String> iter = kolekcija.iterator();
 while(iter.hasNext()) {
  String s = iter.next();
  System.out.println(s);
 }
}
```

Ovakav eksplicitni rad s iteratorima ima smisla ako nam iterator treba za dodatne operacije, poput brisanja trenutnog elementa iz kolekcije ili pak ako iterator šaljemo dalje kao argument nekoj drugoj metodi. Međutim, vrlo često upravo se prikazani isječak koda može napisati sintaksno kraće (i pustiti prevoditelj da taj kraći kod prepiše u ovu dulju varijantu pri prevođenju). Taj sintaksno kraći oblik oslanja se na proširenje petlje `for` prikazanu u nastavku i koju sam prevoditelj prepisuje u prethodno prikazani kod. Treba napomenuti da nam u ovom kraćem slučaju referenca na iterator koji se koristi nije dostupna.

```
public static void ispisi(Collection<String> kolekcija) {
 for(String s : kolekcija) {
  System.out.println(s);
 }
}
```

Sintaksa kojom se uz tip podatka u zagradama navodi dodatni parametar (ili više njih) poput `Set<String> set;` uvedena je s tehnologijom *Java Generics* koja je obrađena u kasnijim poglavljima. Za sada spomenimo samo koja je njezina tipična uporaba: zahvaljujući ovoj tehnologiji prevodilac nas može upozoriti ako neku metodu pozivamo s referencom na objekt neočekivanog tipa. Naime, sve kolekcije koje su dio Java Collection Framework-a kao tip elemenata dopuštaju bilo što što je `Object`, kako bi podržale izradu kolekcija proizvoljnih tipova. Tehnologija *Java Generics* omogućava programeru da prilikom stvaranja parametriziranog razreda definira vrijednost parametra nakon čega prevodilac na argumente metoda postavlja strože uvjete: više nije dovoljno da su bilo kakvi objekti već moraju biti u skladu s definiranim parametrom. Ako na prvu zvuči zbunjujuće, evo primjera koji će ilustrirati o čemu se radi.

```
Set<String> set1 = new HashSet<>();
Set<Integer> set2 = new HashSet<>();
set1.add("Anica");
set1.add("Ivana");

set2.add(new Integer(5671));
set2.add(new Integer(2451));

// Sljedeće će rezultirati pogreškom već pri prevođenju:
set1.add(new Integer(5555));
set2.add("Jasna");
```

Primjer smo započeli stvaranjem dva skupa uz navođenje razreda objekata koji će biti pohranjivani u svaki. Java 5 zahtjevala je da se tip navodi i pri pozivu konstruktora: `Set<String> set1 = new HashSet<String>();`. Od Jave 7, dopušteno je da se u pozivu konstruktora ispusti tip (ali ostaju zagrade, pa je taj oblik poziva dobio ime *The Diamond Operator*, jer ostaje samo `<>`), i to smo koristili u prikazanom primjeru.

Prva dva dodavanja dodaju stringove u `set1`; druga dva cijele brojeve u `set2`. U posljednja dva dodavanja u `set1` se pokušava ubaciti cijeli broj a u `set2` string: oboje će od prevoditelja biti označeno kao pogreška, i takav se program neće moći prevesti.



Važno

Uvijek oblikujte kod tako da ovisnosti idu prema apstrakcijama (*Program to an interface, not an implementation*). Pokazuje se kao dobra praksa da kod zna što manje o konkretnim implementacijama; ako ste tako pisali kod, kasnije je uvijek lagano mijenjati implementacije, a kao bonus, možete pisati kod koji istovremeno radi s različitim implementacijama. Evo dobrog primjera.

```
Set<String> set1 = new HashSet<>();
void ispisi(Collection<String> kol) { ... }
```

Slijedi loš primjer.

```
HashSet<String> set1 = new HashSet<>();
void ispisi(HashSet<String> kol) { ... }
```

Pogledajmo još jedan primjer uporabe skupova.

> Napisati program koji prima imena dviju datoteka preko naredbenog retka. Svaki redak datoteka sadrži jedno ime. Program treba ispisati sva imena koja se nalaze u prvoj datoteci, a nisu u drugoj datoteci. Datoteke su relativno male.

Rješenje ovog problema prikazano je u nastavku.

Primjer 11.2. Filtrirani ispis imena

```
 1 package book.java.collections.set;
 2
 3 import java.io.IOException;
 4 import java.nio.charset.StandardCharsets;
 5 import java.nio.file.Files;
 6 import java.nio.file.Paths;
 7 import java.util.Collection;
 8 import java.util.HashSet;
 9 import java.util.Set;
10
11 public class FiltriraniIspis {
12
13  public static void main(String[] args) throws IOException {
14   if(args.length != 2) {
15    System.out.println("Očekivao sam imena dviju datoteka.");
16    return;
17   }
18
19   Collection<String> retci1 = Files.readAllLines(
20    Paths.get(args[0]),
21    StandardCharsets.UTF_8);
22
23   Collection<String> retci2 = Files.readAllLines(
24    Paths.get(args[1]),
25    StandardCharsets.UTF_8);
26
27   Set<String> prvi = new HashSet<>(retci1);
28
29   Set<String> drugi = new HashSet<>(retci2);
30
31   prvi.removeAll(drugi);
32
33   for(String ime : prvi) {
34    System.out.println(ime);
35   }
36
37  }
38
39 }
40
```

Kako bismo rješili postavljeni problem, oslanjamo se na činjenicu da se skupovi ponašaju upravo u skladu s matematičkom operacijom tog pojma te da postoje operacije koje odgovaraju uniji dvaju skupova (`addAll(...)`), presjeku (`retainAll(...)`) te razlici (`removeAll(...)`). Za potrebe ovog rješenje, želimo u prvom skupu zadržati sva imena koja nisu u drugom skupu: drugim riječima, iz prvog skupa želimo ukloniti sva imena koja su u drugom skupu - prikladna metoda je upravo `removeAll(...)`.

Poglavlje 12. Kolekcija: lista

U Javi, lista predstavlja specijalizaciju kolekcije na način da definira poredanost elemenata: točno se zna koji je prvi element kolekcije, drugi element kolekcije i tako dalje. Stoga se u dokumentaciji za listu još koristi i pojam *sekvenca* (engl. *sequence*). Kolekcija vrste lista definirana je preko sučelja `java.util.List`. Time je ostavljena mogućnost izrade različitih konkretnih implementacija koje će korisniku ponuditi usluge definirane tim sučeljem. Uzimajući u obzir činjenicu da elementi imaju pridjeljene redne brojeve (ili *poziciju*), sučelje `java.util.List` izvedeno je iz sučelja `java.util.Collection` uz definiranje novog API-ja koji omogućava operacije koje koriste tu informaciju.

Sučelje `java.util.List`

Sučelje `java.util.List` prikazano je u primjeru 12.1.

Primjer 12.1. Sučelje `java.util.List`

```
 1 package java.util;
 2
 3 import java.util.Collection;
 4 import java.util.Comparator;
 5 import java.util.ListIterator;
 6 import java.util.function.UnaryOperator;
 7
 8 public interface List<E> extends Collection<E> {
 9
10     // Izmjene
11
12     boolean addAll(int index, Collection<? extends E> c);
13     default void replaceAll(UnaryOperator<E> operator) {...}
14     default void sort(Comparator<? super E> c) {...}
15
16     // Pozicijski pristup
17
18     E get(int index);
19     E set(int index, E element);
20     void add(int index, E element);
21     E remove(int index);
22
23     // Operacije pretraživanja
24
25     int indexOf(Object o);
26     int lastIndexOf(Object o);
27
28     // Iteratori specifični za liste
29
30     ListIterator<E> listIterator();
31     ListIterator<E> listIterator(int index);
32
33     // Pogledi
34
35     List<E> subList(int fromIndex, int toIndex);
36
37 }
38
```

Pri tome je dan prikaz samo važnijih novododanih metoda koje ne postoje u sučelju java.util.Collection. Sučelje je parametrizirano i parametar prosljeđuje nadređenom sučelju. Zahvaljujući činjenici da je ovo sučelje izvedeno iz java.util.Collection, ono sadrži sve tamo definirane metode. Stoga možemo reći da je svaka lista ujedno i kolekcija. Gdje god da kod očekuje kolekciju, možemo mu proslijediti listu i taj bi kod morao uredno raditi.

Neke od metoda koje definira sučelje java.util.Collection u ovom su sučelju dodatno specificirane. Tako primjerice, u sučelju java.util.Collection postoji metoda boolean add(E element) u čijoj dokumentaciji stoji da će pozivom te metode predani element biti dodan u kolekciju, ako je to u skladu s ugovorom trenutne kolekcije. U sučelju java.util.List, dokumentacija ove metode pojašnjava gdje će taj element biti dodan - kako je lista poredana kolekcija, na ovom mjestu ugovor između te metode i pozivatelja definira da će predani element biti dodan na kraj kolekcije (njegov redni broj će biti za jedan veći od dotadašnjeg najvećeg rednog broja). Slično je i s metodom addAll(collection): ona sve elemente predane kolekcije u listu ubacuje na njezin kraj i to redosljedom kojim iterator nad predanom kolekcijom vraća elemente.

Novododane metode sučelja java.util.List po namjeni možemo podijeliti u nekoliko grupa kako je to napravljeno u primjeru 12.1.

U grupi *Izmjene* nalaze se tri nove metode. Metoda addAll(index, collection) se ponaša slično kao i metoda aaddAll(collection); razlika je što ova novododana predanu kolekciju u listu ubacuje počev od pozicije koja je predana kao prvi argument (ostatak liste najprije pomakne za potreban broj mjesta). Tako bismo, primjerice, pozivom addAll(0, collection) elemente predane kolekcije umetnuli na sam početak liste. Evo jednostavnog primjera koji kao implementaciju sučelja koristi razred java.util.ArrayList.

```
 1 List<String> lista = new ArrayList<>();
 2 lista.add("Ivo");
 3 lista.add("Ana");
 4 lista.add("Pero");
 5 lista.add("Ivana");
 6 System.out.println(lista);
 7
 8 Set<String> slova = new HashSet<>(Arrays.asList("A","B"));
 9
10 lista.addAll(slova);
11 System.out.println(lista);
12
13 lista.addAll(2, slova);
14 System.out.println(lista);
```

Tri ispisa će biti redom:

```
[Ivo, Ana, Pero, Ivana]
[Ivo, Ana, Pero, Ivana, A, B]
[Ivo, Ana, A, B, Pero, Ivana, A, B]
```

Metoda replaceAll(unaryOperator) je metoda prima referencu na unarnu funkciju (funkciju nad jednim argumentom) i koja će svaki element liste zamijeniti povratnom vrijednošću koju funkcija vrati pozvana nad tim elementom. Ova funkcija modelirana je sučeljem java.util.function.UnaryOperator<T> i zahtjeva postojanje funkcije T apply(T t);. Želimo li, primjerice, svaki element prethodne liste zamijeniti elementom u kojem su sva slova velika, to bismo mogli napraviti ovako:

```
1 // Definirajmo funkciju:
2 UnaryOperator<String> f = new UnaryOperator<String>() {
3  @Override
4  public String apply(String t) {
5   return t.toUpperCase();
```

```
 6  }
 7 };
 8 // Pozovimo provedbu zamjena:
 9 lista.replaceAll(f);
10 // I ispišimo listu:
11 System.out.println(lista);
```

što bi rezultiralo sljedećim ispisom:

```
[IVO, ANA, A, B, PERO, IVANA, A, B]
```

Prisjetimo se odmah: traženu smo funkciju mogli kraće definirati lambdom pa sve zapisati puno kraće uz jednak rezultat:

```
1 lista.replaceAll(t -> t.toUpperCase());
```

ili još kraće ovako:

```
1 lista.replaceAll(String::toUpperCase);
```

Konačno, pozivom metode `sort(comparator)` moguće je sortirati listu. Što su komparatori i kako ih pisati, bit će pojašnjeno naknadno. Ovdje ćemo ilustrirati kako listu stringova sortirati postojećim komparatorima koji se temelje na lambda izrazu koji koristi postojeće metode razreda `String`.

```
 1 Comparator<String> comp = String::compareTo;
 2
 3 lista.sort(comp);
 4 System.out.println(lista);
 5
 6 lista.sort(comp.reversed());
 7 System.out.println(lista);
 8
 9 lista.sort(null);
10 System.out.println(lista);
```

Prvi poziv (redak 3) listu sortira uzlazno a drugi (redak 6) obrnutim poretkom (dakle, silazno). Treći poziv (redak 9) ne predaje eksplicitni komparator pa metoda sortiranja očekuje da je nad elementima liste definiran njihov *prirodan poredak* koji ta metoda potom koristi. Za stringove, prirodan poredak je leksikografski (abecedno). Sva tri ispisa stoga su:

```
[A, A, ANA, B, B, IVANA, IVO, PERO]
[PERO, IVO, IVANA, B, B, ANA, A, A]
[A, A, ANA, B, B, IVANA, IVO, PERO]
```

U grupi metoda za *Pozicijski pristup* nalaze se četiri metode. Metoda `get(index)` vraća element koji se u listi nalazi na zadanoj poziciji u listi (indeksacija kreće od 0); primjerice, uz posljednje stanje naše liste iz primjera, `lista.get(2)` bi vratilo `"ANA"`. Metoda `set(index, element)` modificira listu na način da na zadanoj poziciji zamjenjuje trenutni element s predanim elementom. Metoda `add(index, element)` u listu na zadanu poziciju umeće predani element (pri tome prije toga počev od predane pozicije sve elemente "konceptualno" posmiće za jedno mjesto kako bi se na zadanoj poziciji napravilo prazno mjesto); pozivom ove metode veličina liste se povećava za jedan. Konačno, metoda `remove(index)` uklanja iz liste element sa zadane pozicije (i sve elemente iza njega "konceptualno" posmiče za jedno mjesto prema početku) čime se veličina liste smanjuje za jedan. U prethodnom opisu namjerno je korišten pojam "konceptualno posmiče" jer ovisno o konkretnoj implementaciji liste to će pomicanje trebati napraviti ili će se ono dogoditi implicitno.

U grupi *Operacije pretraživanja* nalaze se dvije metode koje listu pretražuju ili od početka (`indexOf(element)`) ili od kraja (`lastIndexOf(element)`) u potrazi za predanim ele-

mentom. Stoga će prva metoda vratiti prvi položaj a druga posljednji položaj na kojem se nalazi traženi element. Ako u listi takvog elementa nema, rezultat će biti `-1`. Ako postoji samo jedan traženi element, obje metode će vratiti isti rezultat. Ako lista na više položaja sadrži traženi element (prisjetimo se, lista može sadržavati duplikate), rezultati ovih metoda će se razlikovati.

Grupa *Iteratori specifični za liste* navodi dvije metode koje vraćaju iteratore. Metoda `listIterator()` vraća iterator koji je pozicioniran na početku liste (na lokaciji 0) a metoda `listIterator(index)` vraća iterator koji je početno pozicioniran na poziciji danoj kao argument metode. Ove obje metode vraćaju proširenu inačicu iteratora koja je modelirana sučeljem `java.util.ListIterator>E<` i koja omogućava dvosmjerno kretanje (i unaprijed i unatrag).

Konačno, grupa *Pogledi* definira metodu `subList(fromIndex,toIndex)` koja vraća pogled na dio elemenata originalne liste (od elementa na poziciji `fromIndex` do prvog elementa ispred pozicije `toIndex`). Sam pogled i sam se ponaša kao lista koja prati originalnu listu. Evo primjera koji to ilustrira.

```
1 System.out.println(lista);
2
3 List<String> dio = lista.subList(2, 6);
4 System.out.println(dio.get(0));
5 System.out.println(dio);
6
7 dio.clear();
8 System.out.println(lista);
9 System.out.println(dio);
```

Uočite, novodefinirana lista `dio` vidi četiri elementa originalne liste: one na pozicijama 2, 3, 4 i 5. Međutim, kao što ispis ilustrira, ono što je u originalnoj listi na poziciji 2 to je u našem pogledu na poziciji 0: zato taj pogled i zovemo podlista. Napravimo li modifikaciju nad podlistom, ta se modifikacija propagira i u originalnu listu: pozivom metode `clear` nad podlistom obrisali smo njezine elemente ali smo te iste elemente obrisali i u originalnoj listi. Sljedeći ispisi to ilustriraju.

```
[A, A, ANA, B, B, IVANA, IVO, PERO]
ANA
[ANA, B, B, IVANA]
[A, A, IVO, PERO]
[]
```

Ovako dobiven pogled valjan je sve dok se izravno nad originalnom listom ne napravi strukturna modifikacija (primjerice, doda novi element ili obriše neki element). U tom slučaju, pozivi metoda nad listom-pogledom izazvat će iznimku (`java.util.ConcurrentModificationException`).

Implementacije lista

U okviru Javinog okvira kolekcija, na raspolaganju nam stoje dvije često korištene implementacije ovog sučelja: `java.util.ArrayList` te `java.util.LinkedList`. Postoji još i `java.util.Vector` koja se ponaša slično kao i `java.util.ArrayList` uz razliku da su sve metode zaštićene sinkronizacijskim mehanizmima. Stoga je ta implementacija sigurna za uporabu u višedretvenog okruženju (kada više dretvi čita i piše po istoj takvoj kolekciji) ali ako se koristi u jednodretvenom okruženju, rad je sporiji. Stoga se za većinu slučajeva preporuča koristiti jednu od prethodne dvije implementacije.

`java.util.ArrayList`

Razred `java.util.ArrayList` kao potpornu podatkovnu strukturu koristi relokatibilno polje referenci. Inicijalno, zauzima se polje relativno malog kapaciteta, pa kako se elementi dodaju u listu, upisuju se u to polje. Svaki puta kada se

dosegne alocirani kapacitet polja, implementacija napravi realokaciju polja povećavajući njegov kapacitet za fiksan faktor. U Oracleovoj implementaciji Jave 8, inicijalni kapacitet je 10 elemenata a pri realokaciji polja kapacitet mu se povećava za 50% (novi kapacitet je stari kapacitet puta 1.5); kako je time porast eksponencijalan, ako lista treba držati velik broj elemenata, interno polje će postići potreban kapacitet relativno brzo.

`java.util.LinkedList` — Razred `java.util.LinkedList` kao potpornu podatkovnu strukturu koristi dvostruko povezanu ulančanu listu pri čemu objekt čuva reference na prvi čvor liste (glavu liste) te referencu na posljednji čvor, kao i pomoćnu varijablu u kojoj zapisuje broj čvorova liste (tj. broj elemenata u listi, kako bi metodu `size` mogao implementirati efikasno, bez potrebe da trči po čvorovima i broji koliko ih ima).

U tablici u nastavku dana je gruba usporedba složenosti izvođenja nekih od operacija koje su definirane sučeljem `java.util.List` za ove dvije implementacije. Slovo `n` označava veličinu (broj elemenata) liste.

Svojstvo	**Složenost**	
-	**`ArrayList`**	**`LinkedList`**
get(0)	O(1)	O(1)
get(n-1)	O(1)	O(1)
get(i)	O(1)	O(n)
indexOf(element)	O(n)	O(n)
remove(0)	O(n)	O(1)
remove(n-1)	O(1)	O(1)
remove(i)	O(n)	O(n)
remove(element)	O(n)	O(n)
add(element)	O(1)	O(1)
add(0,element)	O(n)	O(1)

Kod pojedinih operacija tablica razlikuje provođenje operacije nad prvim elementom liste (npr. `remove(0)`), nad zadnjim elementom liste (`remove(n-1)`) te prosječnu složenost kada indeksi variraju uniformno po pozicijama liste. Lijep primjer je upravo operacija `remove` kod koje je prosječna složenost O(n) kod obje implementacije, ali uklanjanje prvog elementa je kod implementacije `LinkedList` garantirano složenosti O(1) jer implementacija ne treba pomicati ostatak elemenata liste dok je kod implementacije `ArrayList` ta složenost O(n) jer se u internom potpornom polju svi elementi moraju posmaknuti za jedno mjesto prema početku. S druge strane, pri operaciji pozicijskog dohvata (metoda `get(i)`), prosječna složenost od `ArrayList` je bitno bolja (O(1)) u odnosu na `LinkedList` koja mora trčati po čvorovima liste kako bi dohvatila element na zadanoj poziciji (pa ima O(n)). Napomenimo da kod implementacije `ArrayList` nismo u obzir uzeli trošak realokacija polja za što ćemo ponuditi dva opravdanja: ako je unaprijed poznato koliko će kolekcija biti velika, ova implementacija nudi konstruktor kojemu se ta informacija može predati pa će odmah u startu biti alocirano dovoljno veliko interno polje da se realokacije neće događati; s druge strane, ako to nije poznato, interni mehanizam eksponencijalnog porasta veličine korištenog polja osigurat će da je taj trošak relativno malen u odnosu na ostatak operacija.

Iz ove kratke analize odmah je vidljivo da niti jedna od implementacija nije u svemu bolja od one druge: priliko odluke koju ćete implementaciju koristiti u kojem trenutku treba proanalizirati kako algoritam koristi listu te koje se operacije često pozivaju i na temelju toga onda odlučiti.

Pomoćne metode

Za razliku od nekih programskih jezika u kojima je liste moguće stvarati na sintaksno jednostavan način (uz malo pisanja), korištenje prikazanih razreda to ne omogućava. Primjerice, želimo li stvoriti listu koja inicijalno čuva imena 6 gradova, evo što je potrebno napisati.

```
1 List<String> gradovi = new ArrayList<>();
2
3 gradovi.add("Zagreb");
4 gradovi.add("Split");
5 gradovi.add("Varaždin");
6 gradovi.add("Vukovar");
7 gradovi.add("Rijeka");
8 gradovi.add("Dubrovnik");
9
```

Složit ćete se da je to dosta pisanja, u usporedbi s jezicima u kojima je dovoljno napisati `gradovi = ["Zagreb", "Split", "Varaždin", "Vukovar", "Rijeka", "Dubrovnik"];`. U paketu java.util postoji razred `Arrays` koji ovo može malo olakšati. Taj nam razred nudi statičku metodu `asList` koja prima varijabilni broj argumenata (ili pak jedno polje elemenata) i vraća primjerak razreda koji je implementacija liste i ujedno živi pogled na to polje. Pogledajte sljedeći primjer:

```
1 String[] elementi = {
2   "Zagreb", "Split", "Varaždin",
3   "Vukovar", "Rijeka", "Dubrovnik"
4 };
5 List<String> gradovi = Arrays.asList(elementi);
6
7 System.out.println(gradovi.get(0));
8 System.out.println(gradovi);
9
10 elementi[4] = "Pula";
11 System.out.println(gradovi);
12
13 gradovi.set(4, "Križevci");
14 System.out.println(gradovi);
15 System.out.println(Arrays.toString(elementi));
16
17 gradovi.sort(null);
18 System.out.println(gradovi);
19 System.out.println(Arrays.toString(elementi));
20
21 gradovi.remove(1);
```

Ispis će biti:

```
Zagreb
[Zagreb, Split, Varaždin, Vukovar, Rijeka, Dubrovnik]
[Zagreb, Split, Varaždin, Vukovar, Pula, Dubrovnik]
[Zagreb, Split, Varaždin, Vukovar, Križevci, Dubrovnik]
[Zagreb, Split, Varaždin, Vukovar, Križevci, Dubrovnik]
[Dubrovnik, Križevci, Split, Varaždin, Vukovar, Zagreb]
[Dubrovnik, Križevci, Split, Varaždin, Vukovar, Zagreb]
Exception in thread "main" java.lang.UnsupportedOperationException
 at java.util.AbstractList.remove(AbstractList.java:161)
 at kolekcije.Proba.main(Proba.java:78)
```

U retku 1 deklariramo polje gradova koje u retku 5 zamatamo u listu. Retci 7 i 8 ilustriraju da je lista korektno inicijalizirana: nulti element vraća "Zagreb" a ispis sadržaja liste doista ispisuje sve elemente koji su bili u polju i to upravo odgovarajućim redosljedom.

Redak 10 potom u polju mijenja element na poziciji 4. U retku 11 ispisujemo listu i u ispisu vidimo da i lista vidi novi element: u ispisu na poziciji 4 više ne piše Rijeka već sada piše Pula. Možemo zaključiti da lista koju smo dobili nije u sebe prekopirala elemente iz predanog polja: ona to polje čitavo vrijeme koristi kao svoju podatkovnu strukturu.

Redak 13 to dalje ilustrira tražeći listu da na poziciju 4 umjesto postojećeg elementa postavi element "Križevci". U retcima 14 i 15 potom ispisujemo listu i originalno polje. I doista, Križevci se vide u oba ispisa: lista je ažurirala originalno polje elemenata.

U retku 17 tražimo listu da se sortira koristeći prirodni poredak svojih elemenata nakon čega u retku 18 i 19 ponovno radimo ispis elemenata liste i originalnog polja. Iz ispisa vidimo da je sortiranje liste sortiralo originalno polje što sada više ne bi smjelo biti čudno: to polje jest podatkovna struktura koju lista koristi kao izvor svojih elemenata.

Konačno, redak 21 traži listu da iz sebe obriše element s pozicije 1. Ova operacija rezultira izazivanjem iznimke `java.lang.UnsupportedOperationException`. Naime, implementacija koju smo dobili od metode `Arrays.asList` ne može promijeniti kapacitet predanog polja - niti ga povećati, niti smanjiti. Stoga će ta implementacija baciti iznimku pri pozivu bilo koje metode koja bi trebala napraviti strukturnu modifikaciju nad listom.

Želimo li (uz malo pisanja) dobiti "normalnu" listu sa zadanim inicijalnim sadržajem, onda trebamo pisati ipak malo više: pozvati konstruktor željene implementacije liste i njemu predati listu koju stvara metoda `Arrays.asList`. Konstruktor će iz predane liste prekopirati elemente i tako se inicijalizirati. Evo isječka.

```
1 String[] elementi = {
2   "Zagreb", "Split", "Varaždin",
3   "Vukovar", "Rijeka", "Dubrovnik"
4 };
5 List<String> gradovi = new ArrayList<>(
6   Arrays.asList(elementi)
7 );
```

Koristeći činjenicu da metoda `Arrays.asList` prima varijabilni broj argumenata, ovo možemo napisati i kao jednu naredbu, kako je prikazano u nastavku.

```
1 List<String> gradovi = new ArrayList<>(Arrays.asList(
2   "Zagreb", "Split", "Varaždin",
3   "Vukovar", "Rijeka", "Dubrovnik"
4 ));
```

Ako se pitate gdje je nestalo polje - odgovor je jednostavan: nije. Prevoditelj prilikom prevođenja kada vidi poziv metode s varijabilnim brojem argumenata za Vas stvori polje, napuni ga tim argumentima i pošalje metodi; to je razlog zašto u takvoj metodi tim argumentima možete pristupati kao da su u polju - oni to doista i jesu.

Primjeri

Za kraj ćemo razmotriti dva zadatka koja je prikladno rješavati listama.

Zadatak 1

Napisati program koji s tipkovnice čita niz decimalnih brojeva (jedan broj po retku); "quit" prekida unos. Program treba ispisati samo one brojeve koji su barem za 20% veći od prosjeka svih unesenih brojeva. Ispis brojeva treba biti od manjih prema većima.

Iz teksta zadatka jasno je da brojeve moramo pamtiti do samog kraja. Naime, iako znamo da nećemo ispisati sve brojeve, što ćemo točno ispisati ovisi o prosjeku a on pak neće biti poznat dok korisnik ne unese sve brojeve. Zaključak: brojeve trebamo pamtiti. Sljedeće pitanje je tada u kojoj kolekciji? Je li skup prikladna struktura od onih koje smo do sada obradili? Skup ne dozvoljava pojavu duplikata pa nećemo moći zapamtiti da je korisnik unio više puta isti broj. Stoga ćemo koristiti listu. Rješenje je prikazano u nastavku (primjer 12.2).

Primjer 12.2. Rješenje zadatka

```
 1 package kolekcije;
 2
 3 import java.util.ArrayList;
 4 import java.util.Collections;
 5 import java.util.List;
 6 import java.util.OptionalDouble;
 7 import java.util.Scanner;
 8
 9 public class VeciOdProsjeka {
10
11  public static void main(String[] args) {
12
13   List<Double> brojevi = new ArrayList<>();
14
15   Scanner scanner = new Scanner(System.in);
16   while(true) {
17    if(scanner.hasNextDouble()) {
18     brojevi.add(scanner.nextDouble());
19    } else {
20     String uneseno = scanner.next();
21     if(uneseno.equals("quit")) {
22      break;
23     }
24     System.out.println("Nepoznat unos.");
25    }
26   }
27
28   scanner.close();
29
30   OptionalDouble avgOpt = prosjek(brojevi);
31   if(!avgOpt.isPresent()) {
32    System.out.println("Niste unijeli niti jedan broj.");
33    return;
34   }
35
36   Double limit = avgOpt.getAsDouble() * 1.2;
37
38   List<Double> zaIspis = new ArrayList<>();
39   for(Double d : brojevi) {
40    if(d >= limit) {
41     zaIspis.add(d);
42    }
43   }
44
45   zaIspis.sort(Collections.reverseOrder(Double::compareTo));
46   zaIspis.forEach(d -> System.out.println(d));
47
48  }
49
```

```
50  private static OptionalDouble prosjek(List<Double> brojevi) {
51   if(brojevi.isEmpty()) {
52    return OptionalDouble.empty();
53   }
54
55   double suma = 0;
56   for(Double d : brojevi) {
57    suma += d;
58   }
59
60   return OptionalDouble.of(suma/brojevi.size());
61  }
62
63 }
64
```

Čitavo rješenje mogli smo napisati još malo sažetije, uklanjanjem eksplicitnog "ručnog" stvaranja nove liste i njezinog filtriranja u petlji (od retka 38). Sve od retka 38 do 46 mogli smo zamijeniti sa sljedeće tri linije koda.

```
1 brojevi.removeIf(d -> d < limit);
2 brojevi.sort(Collections.reverseOrder(Double::compareTo));
3 brojevi.forEach(d -> System.out.println(d));
```

U prvom retku iz liste `brojevi` uklanjamo sve brojeve koji su manji od `limit`; u drugom retku tako filtriranu listu sortiramo i u trećem je ispisujemo.

Zadatak 2

Potrebno je napisati funkciju koja će primiti polje stringova i ispisati sve elemente ali bez duplikata (koji mogu biti prisutni u polju). Ispis treba biti obrnutim poretkom. U slučaju duplikata, ispisuje se element koji je na najranijoj poziciji.

Prvo rješenje koje se izvodi u kvadratnoj složenosti prikazano je u nastavku (primjer 12.3).

Primjer 12.3. Rješenje 1.

```
1 private static void ispisiObrnuto1(String[] polje) {
2  for(int i = polje.length-1; i >= 0; i--) {
3   String kandidat = polje[i];
4   boolean kasnijeIspisati = false;
5   for(int j = i-1; j >= 0; j--) {
6    if(polje[j].equals(kandidat)) {
7     kasnijeIspisati = true;
8     break;
9    }
10  }
11  if(kasnijeIspisati) {
12   continue;
13  }
14  System.out.println(kandidat);
15 }
16 }
17
```

Rješenje ne koristi Javin okvir kolekcija već u dvije ugniježđene petlje prolazi kroz elemente polja od kraja prema početku. *i*-ti element ispisuje samo ako takvog nema nigdje prema početku polja (što se provjerava unutarnjom petljom).

Rješenje je samo po sebi dosta teško čitljivo: imamo ugniježđene petlje, pomoćne zastavice i sve skupa je dosta teško za pratiti. Jedan od načina da kod učinimo čitljivijim je da radimo na malo višoj konceptualnoj razini. Idemo stoga postupiti ovako. U listu ćemo dodavati elemente iz polja, ali samo ako isti već nisu u listi. Tako stvorena lista neće imati duplikate. Listu ćemo potom okrenuti (što je linearna složenost) i potom ispisati (primjer 12.4).

Primjer 12.4. Rješenje 2.

```
 1 private static void ispisiObrnuto2(String[] polje) {
 2
 3  List<String> zaIspis = new ArrayList<>();
 4
 5  for(String kandidat : polje) {
 6   if(!zaIspis.contains(kandidat)) {
 7    zaIspis.add(kandidat);
 8   }
 9  }
10
11  Collections.reverse(zaIspis);
12
13  for(String elem : zaIspis) {
14   System.out.println(elem);
15  }
16 }
17
```

Za okretanje liste koristimo statičku metodu `reverse(lista)` razreda `java.util.Collections`. U ovom rješenju odmaknuli smo se direktnog rada s indeksima i ugniježđenim petljama. Kod se, barem na prvi pogled, čini čitljivijim (relativno je jasno razumjeti što se događa u kojem retku). Složenost koda je, međutim, i dalje kvadratna! Naime, u petlji pozivamo `zaIspis.contains(kandidat)` koja je već sama po sebi O(n) pa je sveukupno rješenje i dalje kvadratne složenosti.

Kvadratnu složenost možemo prebaciti u linearnu tako da zadaću otkrivanja jesmo li već vidjeli neki element ostvarimo u složenosti O(1). Naime, ono što nas zanima kada naiđemo na neki element jest je li to njegova prva pojava ili smo ga već vidjeli. Kako taj zadatak rješiti u O(1)? Odgovor leži u prikadnoj implementaciji sučelja `Set`. Sjetimo se: *skup* je kolekcija bez duplikata. Uz pretpostavku dobrog raspršenja, implementacija `HashSet` posao dodavanja elemenata i ispitivanja je li element već prisutan može ostvariti u O(1). Takvo rješenje prikazano je u nastavku (primjer 12.5).

Primjer 12.5. Rješenje 3.

```
 1 private static void ispisiObrnuto3(String[] polje) {
 2
 3  Set<String> razliciti = new HashSet<>();
 4  List<String> zaIspis = new ArrayList<>();
 5
 6  for(String kandidat : polje) {
 7   if(!razliciti.contains(kandidat)) {
 8    razliciti.add(kandidat);
 9    zaIspis.add(kandidat);
10   }
11  }
12
13  Collections.reverse(zaIspis);
14
15  for(String elem : zaIspis) {
```

```
16    System.out.println(elem);
17   }
18 }
19
```

Ovime smo došli do rješenja koje je linearne složenosti odnosno sveukupno radi u O(n). Sada, međutim, možemo ugraditi malo inženjerskog pristupa i rješenje još malo doraditi (dakako, složenost će ostati ista). Fokusirajmo se najprije na isječak:

```
1 if(!razliciti.contains(kandidat)) {
2   razliciti.add(kandidat);
```

Što se tu zapravo događa, ako je implementacija skupa `HashSet`? U metodi `razliciti.contains(kandidat)` najprije računamo sažetak kandidata, odlazimo u tablicu raspršenog adresiranja i u pogođenom slotu tražimo postoji li već takav objekt pa vraćamo rezultat pretrage. Potom u `razliciti.add(kandidat);` opet računamo sažetak kandidata, opet odlazimo u tablicu raspršenog adresiranja i u pogođenom slotu tražimo postoji li već takav objekt pa ako ga nema, ubacujemo ga tamo što nas ništa ne košta i potom pozivatelju vraćamo jesmo li ga umetnuli ili nismo. Očito je da dva puta radimo istu stvar.

Bolje rješenje prikazano je u nastavku (primjer 12.6).

Primjer 12.6. Rješenje 4.

```
1 private static void ispisiObrnuto4(String[] polje) {
2
3  Set<String> razliciti = new HashSet<>();
4  List<String> zaIspis = new ArrayList<>();
5
6  for(String kandidat : polje) {
7   if(razliciti.add(kandidat)) {
8    zaIspis.add(kandidat);
9   }
10  }
11
12  Collections.reverse(zaIspis);
13
14  for(String elem : zaIspis) {
15   System.out.println(elem);
16  }
17 }
18
```

U tom rješenju smo izbjegli lanac `razliciti.contains(...)` pa `razliciti.add(...)` i koristimo samo posljednju metodu te na temelju njezinog rezultata element ubacujemo u listu ili ne.

Konačno, uočimo da jedno vrijeme možemo u cijelosti bez liste: važno nam je izgraditi kolekciju u kojoj nema duplikata i čiji će iterator vraćati elemente onim redosljedom kojim su isti dodavani u kolekciju. Pa to upravo radi `LinkedHashSet`! Da nemamo onaj zahtjev na kraju da sve treba ispisati obrnutim redosljedom, mogli bismo u cijelosti proći bez liste. Ovako ćemo nakon konstrukcije skupa napuniti listu njegovim elementima, okrenuti je i ispisati. Rješenje je prikazano u nastavku (primjer 12.7).

Primjer 12.7. Rješenje 5.

```
1 private static void ispisiObrnuto5(String[] polje) {
2
3  Set<String> razliciti = new LinkedHashSet<>();
```

```
 4
 5  for(String kandidat : polje) {
 6   razliciti.add(kandidat);
 7  }
 8
 9  List<String> zaIspis = new ArrayList<>(razliciti);
10  Collections.reverse(zaIspis);
11
12  for(String elem : zaIspis) {
13   System.out.println(elem);
14  }
15 }
16
```

Ovo konačno rješenje vrlo je čitljivo, jasno, radi u složenosti O(n) i sve skupa ima vrlo malo redaka koda.

Poglavlje 13. Kolekcija: mapa

U različitim programskim jezicima ono što je u Javi poznato pod pojmom *mapa* može se pronaći pod nizom sinonima od koji su raširenji: riječnik (engl. *dictionary*), primjerice u jezicima C# te Python, te asocijativno polje (engl. *associative array*), primjerice u jeziku JavaScript. U svim tim programskim jezicima ideja je ista: radi se o kolekciji uređenih parova *(ključ, vrijednost)* pri čemu su ključevi jedinstveni (kolekcija ne može sadržavati više uređenih parova s istim ključem).

Ovakve kolekcije najčešće koristimo kako bismo prema zadanom ključu brzo dohvatili vrijednost koju smo pohranili uz taj ključ (ili kako bismo pohranili vrijednost). Neki od primjera uporabe ovakve kolekcije su sljedeći: za svakog studenta zapamtiti je li završio fakultet (ključevi bi mogli biti matični brojevi, vrijednosti Booleove zastavice); za svaki kolegij u akademskoj godini zapamtiti popis studenata koji ga slušaju (ključevi bi mogli biti šifre kolegija ili čak objekti koji predstavljaju kolegije, vrijednosti bi mogle biti skupovi matičnih brojeva studenata ili pak skupovi objekata koji predstavljaju studente), itd.

Mape su u Javi modelirane parametriziranim sučeljem `java.util.Map<K,V>` pri čemu prvi parametar određuje tip ključa (od tuda oznaka `K`: engl. *Key*) a drugi tip vrijednosti (oznaka `V`: engl. *Value*). Dvije tipične operacije nad mapama ilustrirane su sljedećim kodom.

```
 1 Map<Integer,Boolean> mapa = odaberiNekuImplementaciju();
 2
 3 mapa.put(0, false);
 4 mapa.put(1, false);
 5 mapa.put(2, true);
 6 mapa.put(3, true);
 7 mapa.put(4, false);
 8 mapa.put(5, true);
 9
10 System.out.format("Je li 2 prost broj? %b%n", mapa.get(2));
11 System.out.format("Je li 4 prost broj? %b%n", mapa.get(4));
12
```

Pretpostavimo za trenutak da metoda `odaberiNekuImplementaciju` vraća objekt koji je mapa (konkretna implementacija nije važna). U primjeru najprije pozivima metode `put` u mapu upisujemo podatke. Prvi argument je ključ a drugi vrijednost koju treba pohraniti uz taj ključ. U primjeru smo za šest cijelih brojeva (od 0 do 5) u mapu zapisali jesu li oni prosti brojevi ili nisu. Umjesto ručnog popunjavanja mogli smo napisati i metodu koja prima referencu na mapu i koja algoritamski provjerava je li broj prost i tu informaciju upisuje u mapu. Imamo li na raspolaganju metodu `boolean jeLiProst(int n)`, tada bismo mapu mogli koristiti kao keš (priručnu memoriju): ako nas više puta zanima je li neki broj prost, umjesto da svaki puta pozivamo metodu koja to računa, izračun bismo mogli napraviti jednom, na početku izvođenja programa, rezultate zapisati u mapu i potom ih od tamo koristiti.

Opisanu ideju možemo još malo doraditi: umjesto da unaprijed za raspon brojeva računamo jesu li prosti ili ne i to upisujemo u mapu, mogli bismo napisati metodu koja prima referencu na mapu i najprije provjerava postoji li u mapi već odgovor; ako postoji, vrati ga iz mape a ako ne postoji, provede izračun, rezultat pohrani u mapu i potom ga vrati pozivatelju. Na ovaj način izračune bismo radili samo za one brojeve za koje nas to doista i zanima i to samo prvi puta kada to zatražimo za svaki od brojeva.

Zbog ugovora između mape i korisnika koji definira da su ključevi u mapi jedinstveni, kod koji više puta u mapu upisuju vrijednosti za isti ključ rezultirat će mapom u kojoj je ključu pridružena posljednja vrijednost: svakim novim upisivanjem, prethodna vrijednost će biti zamijenjena novom.

Prethodni primjer završava s dva ispisa koji ilustriraju kako iz mape dohvaćamo prethodno pohranjenu vrijednost: radi se o metodi `get` koja prima ključ a vraća vrijednost koja bila pohranjena uz taj ključ (tj. koju smo pridružili tom ključu).

Podatkovne strukture koje će se interno koristiti za implementaciju ovakve kolekcije mogu biti različite i korisniku će nuditi različite složenosti izvođenja operacija. Ako očekujemo vrlo malo podataka, tada bismo pamćenje mogli ostvariti ulančanom listom čiji bi čvor pamtio referencu na ključ, referencu za vrijednost te referencu na sljedeći čvor. Uz vrlo malen broj zapisa, unatoč tome što su sve složenosti (upisivanje, dohvat po ključu) O(n), takva bi implementacija bila sasvim prihvatljiva.

Za specifične vrste ključeva moguće je koristiti vrlo brze podatkovne strukture. Razmotrimo naš prethodni primjer: kako su ključevi brojevi, i to od 0 do 5, a vrijednosti Booleove zastavice, podatke bismo mogli čuvati u najobičnijem polju: `boolean[] data = new boolean[6];`. Dohvat te upisivanje po ključu čitalo bi odnosno zapisivalo direktno u to polje na lokaciju koja je određena samim ključem. Složenosti tih operacija pri tome bi garantirano bile O(1) u svim slučajevima. Ovakva implementacija ipak neće biti prikladna za općenitu uporabu pa se u praksi ne pojavljuje u bibliotekama.

Posljednje dvije podatkovne strukture koje ćemo spomenuti su uređeno binarno stablo te tablica raspršenog adresiranja. Uređeno binarno stablo moglo bi biti uređeno prema ključevima, pri čemu bi svaki čvor pamtio referencu na ključ, referencu na pridruženu vrijednost te reference na lijevo i desno dijete. Takva bi implementacija mape, uz pretpostavku da se gradi potpuno balansirano binarno stablo, korisnicima ponudila složenost O(log n) i za operaciju upisivanja i za operaciju dohvata po ključu. Lijepo svojstvo takve implementacije je da je vrlo jednostavno obići sve pohranjene uređene parove sortirano prema ključevima (jer samo stablo već ostvaruje takvu strukturu). Spomenimo da je stablo korišteno u osnovnoj implementaciji mape koja je korisnicima jezika C++ dostupna u biblioteci STL.

Konačno, podatkovna struktura tablice raspršenog adresiranja također se može koristiti za ostvarivanje mape. Svaki zapis će se sastojati minimalno od reference na ključ te reference na pridruženu vrijednost. U memoriji će se čuvati polje takvih zapisa određenog kapaciteta. Raspršivanje će se raditi prema ključu. Uz pretpostavku da je kapacitet tablice (veličina polja) dovoljno velik s obzirom na broj uređenih parova koje želimo pohraniti te uz pretpostavku da imamo na raspolaganju kvalitetnu funkciju za raspršivanje ključeva, ovakva implementacija korisnicima može ponuditi operacije umetanja te pretraživanja po ključu u složenosti O(1). U praksi, da bismo imali složenost koja je tome bliska, treba doraditi još neke detalje (kako se postupa ako dođe do kolizije; što učiniti ako se broj pohranjenih parova približi kapacitetu polja i slično), što kvalitetne implementacije i rade. Evo jednog primjera kako bi ovo moglo biti riješeno. Preljevna politika rješavala bi se izgradnjom ulančane liste: zapisi bi ujedno bili i čvorovi liste s referencom na sljedeći čvor. Tablica bi tada efektivno čuvala reference na glave lista; ključ bi definirao u koju listu treba ubaciti zapis; ako više ključeva pogodi istu listu, nije problem. Pri svakom umetanju novog para, mogli bismo provjeriti omjer broja upisanih zapisa i kapaciteta tablice te kada taj omjer postane nepovoljan, mogli bismo automatski alocirati novu veću tablicu (recimo dvostruko veću) i zapise iz stare prebaciti u novu. Kako se na ovaj način kapacitet, kada je to potrebno, povećava eksponencijalno, vrlo brzo ćemo doći do tablice prikladne veličine (a korisnik o tome neće znati ništa). Još jedna mogućnost koja nam stoji na raspolaganju je umjesto lista preljeve čuvati u binarnim stablima: tablica bi tada čuvala reference na glave stabala, pa čak i uz dosta kolizija, složenosti operacija kretale bi se negdje između O(1) i O(log n) što je vrlo praktično rješenje.

Sučelje `java.util.Map`

Odabrani dio metoda sučelja `java.util.Map` prikazan je u primjeru 13.1.

Primjer 13.1. Sučelje `java.util.Map`

```
1 package java.util.Map;
2
```

```
 3 import java.io.Serializable;
 4 import java.util.Collection;
 5 import java.util.Comparator;
 6 import java.util.ConcurrentModificationException;
 7 import java.util.Objects;
 8 import java.util.Set;
 9 import java.util.function.BiConsumer;
10 import java.util.function.BiFunction;
11 import java.util.function.Function;
12
13 public interface Map<K,V> {
14
15  // Dohvat informacija
16
17  int size();
18  boolean isEmpty();
19  boolean containsKey(Object key);
20  boolean containsValue(Object value);
21  V get(Object key);
22
23  // Operacije izmjena
24
25  V put(K key, V value);
26  V remove(Object key);
27
28  // Grupne operacije
29
30  void putAll(Map<? extends K, ? extends V> m);
31  void clear();
32
33  // Dohvat pogleda
34
35  Set<K> keySet();
36  Collection<V> values();
37  Set<Map.Entry<K, V>> entrySet();
38
39  // Model uređenog para:
40
41  interface Entry<K,V> {
42     K getKey();
43     V getValue();
44     V setValue(V value);
45
46     ...
47  }
48
49  // Pretpostavljene metode
50
51  default V getOrDefault(Object key, V defaultValue) {...}
52
53  default void forEach(
54   BiConsumer<? super K, ? super V> action) {
55    ...
56  }
57
58  default void replaceAll(
59   BiFunction<? super K, ? super V, ? extends V> function) {
60    ...
```

```
61   }
62
63   default V putIfAbsent(K key, V value) {...}
64
65   default boolean remove(Object key, Object value) {...}
66
67   default boolean replace(K key, V oldValue, V newValue) {
68     ...
69   }
70
71   default V replace(K key, V value) {...}
72
73   default V computeIfAbsent(K key,
74    Function<? super K, ? extends V> mappingFunction) {
75     ...
76   }
77
78   default V computeIfPresent(K key,
79    BiFunction<? super K, ? super V, ? extends V>
80     remappingFunction) {
81     ...
82   }
83
84   default V compute(K key,
85    BiFunction<? super K, ? super V, ? extends V>
86     remappingFunction) {
87     ...
88   }
89
90   default V merge(K key, V value,
91    BiFunction<? super V, ? super V, ? extends V>
92     remappingFunction) {
93     ...
94   }
95 }
96
```

Metode sučelja smo razdijelili u nekoliko konceptualno povezanih grupa, od kojih je prva "Dohvat informacija". Tu se nalaze metode `size` za dohvat informacije o veličini mape (odnosno broju pohranjenih uređenih parova), metoda `isEmpty` koja ispituje je li mapa prazna, metoda `containsKey` koja provjerava postoji li u mapi zapis za zadani ključ, metoda `containsValue` koja provjerava je li zadana vrijednost u mapi pridružena bilo kojem od postojećih ključeva te metoda `get` koja prima ključ i vraća vrijednost koja je u mapi pohranjena za taj ključ; ako u mapi nema zapisa za zadani ključ, rezultat će biti `null`.

Grupa "Operacije izmjena" sadrži dvije metode. Metoda `put` prima ključ i vrijednost te taj uređeni par upisuje u mapu. Ako je u mapi već postojao zapis s istim ključem, on će biti prepisan a metoda će vratiti vrijednost koja je do tada bila povezana sa zadanim ključem; ako zapis nije postojao, bit će dodan a metoda će vratiti `null`. Metoda `remove` prima ključ i iz mape briše uređen par za taj ključ. Ako u mapi nije bilo uređenog para sa zadanim ključem, metoda vraća `null` a inače vraća vrijednost koja je bila pohranjena uz zadani ključ.

Grupa "Grupne izmjene" sadrži dvije metode. Metoda `putAll` prima neku drugu mapu i u sebe dodaje sve uređene parove koji su prisutni u toj drugoj mapi (koja ostaje nepromijenjena). Metoda `clear` briše sve zapise iz mape.

Grupa "Dohvat pogleda" nudi dvije tri metode koje omogućavaju stvaranje živih pogleda na postojeću mapu. Tako metoda `keySet` vraća kolekciju koja je skup (tj. tipa `java.util.Set`) i koja sadrži sve ključeve koji trenutno postoje u mapi odnosno prvi element iz svakog od ure-

đenih parova. Odabir skupa kao prikladne vrste kolekcije ne treba čuditi s obzirom da skup ne dozvoljava duplikate a mapa garantira da su ključevi jedinstveni. Brisanjem ključa iz tako dobivenog skupa iz mape će se obrisati pripadni uređeni par. Metoda `values` vraća kolekciju koja sadrži sve vrijednosti koje se nalaze u mapi (drugi element svih uređenih parova). Ova kolekcija može sadržavati duplikate (više istih vrijednosti može biti u mapu umetnuto preko različitih ključeva - primjerice, puno studenata može imati ocjenu 5), pa je kao tip odabrana najopćenitija kolekcija. Konačno, metoda `entrySet` vraća skup uređenih parova koji su modelirani sučeljem `java.util.Map.Entry<K,v>` koje je prikazano u nastavku. Kako su ključevi u uređenim parovima jedinstveni, tako su i sami uređeni parovi jedinstveni pa je kao tip kolekcije uzet skup. Brisanjem uređenog para iz ovog skupa uklanja se isti zapis i iz mape.

Posljednju i najveću grupu "Pretpostavljene metode" čine metode koje su dodane od Jave 8 a predstavljaju skup pomoćnih metoda kojima je moguće obaviti niz čestih zadataka pisanjem manje koda. Jedan od čestih idioma u kodu koji koristi mape je:

```
1 Map<String,Integer> brojaci = vratiNekuMapu();
2 Integer brojPojava;
3 if(brojaci.containsKey("Zagreb")) {
4   brojPojava = brojaci.get("Zagreb");
5 } else {
6   brojPojava = 0;
7 }
```

Posljednjih šest redaka u lokalnu varijablu `brojPojava` dohvaća broj koji zapisan za ključ "Zagreb" odnosno ako tog ključa nema u mapi, onda varijablu inicijalizira na pretpostavljenu vrijednost 0. Uporabom metode `getOrDefault` koja prima ključ i pretpostavljenu vrijednost, tih se šest redaka može pokratiti u jedan:

```
1 Map<String,Integer> brojaci = vratiNekuMapu();
2 Integer brojPojava = brojaci.getOrDefault("Zagreb", 0);
```

Metoda `forEach` prima referencu na korisničku akciju koju je potrebno obaviti nad svakim uređenim parom i zatim iterira kroz sve uređene parove i poziva tu akciju. Akcija je ovdje modelirana funkcionalnim sučeljem `java.util.function.BiConsumer` koje definira jednu metodu `accept` koja prima dva argumenta. U ovom slučaju, ta će dva argumenta biti ključ i vrijednost. Primjerice, sve uređene parove na zaslon možemo ispisati uz jednostavnu lambdu kako je prikazano u nastavku.

```
1 brojaci.forEach((k,v) -> System.out.format("%s->%d%n", k, v));
```

Metoda `replaceAll` omogućava pisanje koda koji će u svim uređenim parovima vrijednost postaviti na neku novu koja je funkcija ključa i vrijednosti uređenog para. Korisnik pri tome zadaje referencu na objekt koji računa tu funkciju i koji je opisan funkcionalnim sučeljem `java.util.function.BiFunction` koje definira jednu metodu koja prima dva argumenta (u ovom slučaju dobit će trenutni ključ i trenutnu vrijednost) i vraća rezultat (u ovom slučaju željenu novu vrijednost). Primjerice, želimo li u našoj mapi brojača vrijednosti povezane sa svim gradovima povećati za jedan, to možemo napraviti sljedećim pozivom.

```
1 brojaci.replaceAll((k,v) -> v+1);
```

Ako bismo vrijednost htjeli povećati za 1 samo onim gradovima čije je ima dulje od 5 znakova, to bismo napravili ovako:

```
1 brojaci.replaceAll((k,v) -> k.length()>5 ? v+1 : v);
```

Uporabom metode `putIfAbsent` možemo u mapu upisati za zadani zljuč zadanu vrijednost, ali samo ako taj ključ u mapi već nije povezan s nekom vrijednosti (koja je različita od `null`). Primjerice, u našem primjeru, kod:

```
1 brojaci.putIfAbsent("Zagreb", 20);
```

neće napraviti ništa ako u mapi već imamo pridružen neki broj ključu "Zagreb"; ako nemamo, onda će u mapu unijeti novi zapis ("Zagreb", 20).

Metoda remove(key,value) će iz mape obrisati uređen par koji je određen zadanim ključem i vrijednosti. Ako u mapi za zadani ključ nema zapisa ili ako zapis postoji, ali je u njemu vrijednost različita od predane, metoda ne radi ništa.

Metoda replace(key,oldValue,newValue) mijenja vrijednost koja je zapisana za zadani ključ key na vrijednost newValue samo ako u mapi za taj ključ već postoji zapis i trenutno pridružena vrijednost je upravo oldValue. Jednostavnija inačica ove metode je metoda replace(key,value) koja mijenja vrijednost prdruženu ključu key na zadanu vrijednost value ako u mapi postoji zadani ključ; u suprotnom ne radi ništa.

Slijede dvije konceptualno slične funkcije: computeIfAbsent i computeIfPresent. Prva prima ključ i referencu na objekt koji je opisan funkcionalnim sučeljem java.util.function.Function koje definira metodu apply koja prima jedan argument, transformira ga u rezultat i to vraća (model unarne funkcije). Metoda computeIfAbsent potom provjeri da zadani ključ *nije prisutan* u mapi (ili je povezan s null) i samo ako je to slučaj, u mapu pod zadanim ključem pohrani vrijednost koju dobije tako da pozove predanu funkciju i preda joj trenutni ključ. Uz pretpostavku da "Zagreb" nije u mapi, kod:

```
1 brojaci.computeIfAbsent("Zagreb", k -> k.length());
```

će pod ključem "Zagreb" upisati vrijednost 6 (što je duljina stringa "Zagreb"). Metoda computeIfPresent prima dva argumenta: ključ i referencu na objekt koji je opisan funkcionalnim sučeljem java.util.function.BiFunction koje definira jenu metodu koja prima dva argumenta i vraća rezultat (binarna funkcija). Metoda će, samo u slučaju da je u mapi predani ključ *već povezan* s nekom vrijednosti koja nije null dohvatiti tu trenutnu vrijednost, nad predanim objektom pozvati funkciju kojoj će predati ključ (kao prvi argument) i pronađenu trenutnu vrijednost (kao drugi argument) i potom promotriti dobivenu vrijednost. Ako je ona različita od null, ona će biti postavljena kao nova vrijednost, a ako je ona null, uređeni par sa zadanim ključem će biti obrisan. Evo primjera.

```
1 brojaci.computeIfPresent("Zagreb", (k,v) -> v+1);
2 brojaci.computeIfPresent("Varaždin", (k,v) -> null);
```

Prvi redak će u mapi vrijednost povezanu s ključem "Zagreb" povećati za 1 (ako je taj ključ uopće u mapi i ima pridijeljenu neku vrijednost). Drugi redak će iz mape obrisati uređen par za ključ "Varaždin" (ako u mapi postoji zapis za taj ključ i ako mu je vrijednost različita od null). Uočite da posljednji redak nije isto što i brojaci.remove("Varaždin");. Naime, mape općenito dozvoljavaju da se pod ključem pohrani vrijednost null; tada u toj mapi taj ključ postoji, ali mu je pridijeljena vrijednost null pa taj uređeni par metoda computeIfPresent neće dirati.

Posljednje dvije metode koje su preostale su compute (vrlo općenita metoda) te merge (metoda specifične primjene). Metoda compute prima referencu na ključ i na objekt koji predstavlja binarnu funkciju. Binarnu funkciju poziva s ključem i vrijednosti koja je trenutno pridružena tom ključu (ako takve nema, ovaj argument će biti null) i tako dobiva novu vrijednost. Ako je ona različita od null, u mapu se pod zadanim ključem upisuje nova vrijednost. Ako je nova vrijednost koju je funkcija vratila jednaka null, iz mape se briše zapis za zadani ključ (ako je uopće postojao). Kao svoj rezultat metoda pozivatelju ujedno vraća ono što je funkcija izračunala kao novu vrijednost. Evo primjera. Recimo da želimo u mapu za "Vinkovci" povećati pridruženi broj (ako je neki broj već pridružen) a inače želimo upisati 1. To možemo postići kako je prikazano u nastavku.

```
1 brojaci.compute("Vinkovci", (k,v) -> v==null ? 1 : v+1);
```

Metoda merge omogućava sličnu funkcionalnost kao i compute, ali ipak je malo drugačija. Metoda prima tri argumenta: ključ, vrijednost te referencu na objekt koji je binarna funkcija. Ako u mapi predani ključ ne postoji ili je povezan s null vrijednosti, metoda u mapu upisuje predanu

vrijednost. U suprotnom, metoda poziva predanu funkciju i kao argumente predaje vrijednost koja je trenutno pohranjena u mapi te predanu vrijednost. Zadaća funkcije je iskombinirati staru i predanu vrijednost u novu koja će potom biti upisana u mapu. Ako funkcija vrati `null`, uređeni par se briše iz mape. Prethodno opisani primjer s gradom "Vinkovci" uporabom ove metode možemo rješiti na sljedeći način.

```
1 brojaci.merge("Vinkovci", 1, (vOld,vNew) -> vOld+vNew);
```

Tipičniji primjer je mapa u kojoj su ključevi primjerice radnici, a pridružene vrijednosti poruke koje su poslane radniku. Metodu koja u mapu `pr` (poruke radnicima) za zadanog radnika dodaje predanu poruku mogli bismo napisati ovako.

```
1  public void dodaj(Map<String,String> pm,
2      String radnik, String poruka) {
3
4    String dosadasnjePoruke = pm.get(radnik);
5    if(dosadasnjePoruke!=null) {
6      // Kombiniramo i upisujemo:
7      String novaVrijednost = dosadasnjePoruke + poruka;
8      pm.put(radnik, novaVrijednost);
9    } else {
10     // Samo upisujemo jer je ovo prva poruka za tog radnika:
11     pm.put(radnik, poruka);
12   }
13
14 }
```

Uporabom metode `merge` ovo možemo postići puno kraće:

```
1  public void dodaj(Map<String,String> pm,
2      String radnik, String poruka) {
3
4    pm.merge(radnik, poruka, (o,n) -> o+n);
5
6  }
```

Kako su u prethodnom primjeru vrijednosti stringovi, zavirite li u Javadoc tog razreda, pronaći ćete nestatičku metodu `concat` koja stvara novi string tako da sadržaju trenutnog stringa doda sadržaj stringa poslanog kao jedini argument: idealno za još kraću lambdu:

```
1  public void dodaj(Map<String,String> pm,
2      String radnik, String poruka) {
3
4    pm.merge(radnik, poruka, String::concat);
5
6  }
```

Implementacije sučelja `java.util.Map`

Od standardnih implementacija, na raspolaganju imamo tri razreda u paketu `java.util`: `HashMap`, `TreeMap` te `LinkedHashMap`. Detaljniji opis dan je u nastavku.

`java.util.HashMap`

Razred `java.util.HashMap` kao potpornu podatkovnu strukturu koristi dinamičku tablicu raspršenog adresiranja. Inicijalno, zauzima se tablica relativno malog kapaciteta, pa kako se elementi dodaju u mapu, raspršuju se i upisuju u to polje. Svaki puta kada popunjenost tablice postane kritična s obzirom na njezin kapacitet, implementacija napravi realokaciju tablice povećavajući njezin kapacitet za fiksan faktor.

U Oracleovoj implementaciji Jave 8, inicijalni kapacitet je 16 elemenata, kritična popunjenost tablice je 75% a pri realokaciji tablice kapacitet joj se povećava za 100% (novi kapacitet je stari kapacitet puta 2); kako je time porast eksponencijalan, ako tablica treba držati velik broj elemenata, potreban će kapacitet postići relativno brzo.

`java.util.TreeMap` Razred `java.util.TreeMap` kao potpornu podatkovnu strukturu koristi vrstu uređeno balansiranog binarnog stabla. Interno još u pomoćnoj varijabli zapisuje i broj čvorova stabla (tj. broj elemenata u stablu, kako bi metodu `size` mogao implementirati efikasno, bez potrebe da trči po čvorovima i broji koliko ih ima).

`java.util.LinkedHashMap` Razred `java.util.LinkedHashMap` sličan je razredu `java.util.HashMap`; kao podatkovnu strukturu koristi stopljenu ulančanu listu u tablicu raspršenog adresiranja. Ulančana lista pamti poredak kojim su elementi umetani u tablicu, pa ova implementacija nudi iterator koji uređene parove obilazi redosljedom kojim su dodavani. Performanse ostalih metoda odgovaraju onima razreda `java.util.HashMap`.

Sada možemo usporediti i složenosti nekih od čestih operacija koje nude ove implementacije mape.

Svojstvo	**Složenost**		
-	**`HashMap`**	**`TreeMap`**	**`LinkedHashMap`**
get(k)	O(1)	O(log n)	O(1)
put(k,v)	O(1)	O(log n)	O(1)
containsKey(k)	O(1)	O(log n)	O(1)
containsValue(v)	O(n)	O(n)	O(n)
remove(k)	O(1)	O(log n)	O(1)

Iterator koji vraća razred `HashMap` (pri tome mislimo na `mapa.entrySet().iterator()` odnosno `mapa.keySet().iterator()`) uređene parove obilazi naoko slučajnim poretkom (zapravo, redosljedom kojim se fizički nalaze u tablici raspršenog adresiranja). Iterator razreda `TreeMap` obilazak radi sortirano prema ključevima. Iteratori razreda `LinkedHashMap` obilazak rade prema redosljedu umetanja uređenih parova u mapu.

Primjer uporabe

Rješavamo sljedeći zadatak. U tekstovnoj datoteci `imena.txt` nalazi se niz imena (neka i više puta). Potrebno je napisati program koji će na zaslon ispisati svako ime te koliko puta se ono nalazi u datoteci; isto ime ne smije biti ispisano više puta a ispis treba biti sortiran po imenima.

Jedno moguće rješenje prikazano je u primjeru 13.2.

Primjer 13.2. Rješenje zadatka

```
1 package book.java.collections;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.ArrayList;
```

```
 8 import java.util.HashMap;
 9 import java.util.List;
10 import java.util.Map;
11
12 public class Imena {
13
14  public static void main(String[] args) throws IOException {
15   List<String> retci = Files.readAllLines(
16    Paths.get("imena.txt"), StandardCharsets.UTF_8
17   );
18
19   Map<String, Integer> brojacImena = new HashMap<>();
20   for(String ime : retci) {
21    if(brojacImena.containsKey(ime)) {
22     Integer stari = brojacImena.get(ime);
23     Integer novi = stari + 1;
24     brojacImena.put(ime, novi);
25    } else {
26     brojacImena.put(ime, 1);
27    }
28   }
29
30   List<String> imena = new ArrayList<>(brojacImena.keySet());
31   imena.sort(null);
32
33   for(String ime : imena) {
34    System.out.format(
35     "%s => %d%n", ime, brojacImena.get(ime));
36   }
37
38  }
39
40 }
41
```

U prikazanom rješenju, u retcima 15-17 sadržaj datoteke s imenima dohvaćamo kao listu redaka. Potom kao implementaciju mape biramo `HashMap` zbog povoljnih složenosti operacija dohvata i pohrane podataka po zadanom ključu. U mapi, imena koristimo kao ključeve a broj do tada viđenih pojava tog imena kao vrijednosti.

U petlji prolazimo kroz sve retke (svaki redak je jedno ime). Za to ime provjeravamo jesmo li ga već vidjeli, pa ako jesmo, onda zapamćeni brojač povećavamo za jedan; u suprotnom smo naišli na prvu pojavu imena pa ga u mapu dodajemo s brojačem postavljenim na 1 (retci 20-28).

Kako ispis mora biti sortiran po imenima, stvaramo pomoćnu listu i punimo je svim ključevima prethodno popunjene mape (to su sva različita imena; redak 30). Tu listu potom sortiramo (redak 31). Na kraju u petlji prolazimo ime po ime iz prethodno sortirane liste, iz mape za trenutno ime dohvaćamo pridruženi brojač i to sve zajedno ispisujemo.

Malo drugačije rješenje prikazano je u nastavku (primjer 13.3).

Primjer 13.3. Malo drugačije rješenje zadatka

```
1 package book.java.collections;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
```

```
 7 import java.util.ArrayList;
 8 import java.util.HashMap;
 9 import java.util.List;
10 import java.util.Map;
11
12 public class Imena {
13
14  public static void main(String[] args) throws IOException {
15   List<String> retci = Files.readAllLines(
16    Paths.get("imena.txt"), StandardCharsets.UTF_8
17   );
18
19   Map<String, Integer> brojacImena = new HashMap<>();
20   for(String ime : retci) {
21    if(brojacImena.containsKey(ime)) {
22     Integer stari = brojacImena.get(ime);
23     Integer novi = stari + 1;
24     brojacImena.put(ime, novi);
25    } else {
26     brojacImena.put(ime, 1);
27    }
28   }
29
30   List<Map.Entry<String, Integer>> parovi =
31    new ArrayList<>(brojacImena.entrySet());
32   parovi.sort(Map.Entry.comparingByKey());
33
34   for(Map.Entry<String, Integer> par : parovi) {
35    System.out.format(
36     "%s => %d%n", par.getKey(), par.getValue());
37   }
38  }
39
40 }
```

U odnosu na prvo rješenje, ovdje smo u retku 30 stvorili listu kompletnih uređenih parova iz prethodno popunjene mape. Ako radimo s uređenim parovima, onda nema potrebe da nad mapom ponovno zovemo metodu koja za zadani ključ dohvaća pridruženu vrijednost - to već imamo na raspolaganju u uređenom paru. Međutim, uređeni par nema definiran prirodan poredak pa ih metoda `sort` ne zna sortirati bez dodatne pomoći. Stoga smo toj metodi predali referencu na objekt koji uređene parove uspoređuje prema vrijednosti ključa (redak 32, pogledajte argument metode). Nakon sortiranja, sada obilazimo listu uređenih parova i ispisujemo tražene podatke.

Konačna modifikacija ovog rješenja prikazana je u primjeru 13.4. Tijelo prve petlje skratili smo na samo jednu naredbu uporabom metode `merge` koju smo prethodno objasnili. Kao implementaciju mape odabrali smo `TreeMap` koja vraća iterator koji uređene parove obilazi sortirano po ključevima, pa ništa ne trebamo niti sortirati. U petlji koja ispisuje podatke obilazimo mapu preko pogleda koji vraća `entrySet()` i koji sadrži direktno uređene parove.

Primjer 13.4. Još drugačije rješenje zadatka

```
1 package book.java.collections;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
```

```
 7 import java.util.List;
 8 import java.util.Map;
 9 import java.util.TreeMap;
10
11 public class Imena {
12
13  public static void main(String[] args) throws IOException {
14   List<String> retci = Files.readAllLines(
15    Paths.get("imena.txt"), StandardCharsets.UTF_8
16   );
17
18   Map<String, Integer> brojacImena = new TreeMap<>();
19   for(String ime : retci) {
20    brojacImena.merge(ime, 1, Integer::sum);
21   }
22
23   for(Map.Entry<String,Integer> par : brojacImena.entrySet()) {
24    System.out.format(
25     "%s => %d%n", par.getKey(), par.getValue());
26   }
27  }
28
29 }
30
```

Poglavlje 14. Kolekcije i novi razredi

Nakon što smo se kroz poglavlja 10 do 13 upoznali s pojmom kolekcije, osnovnim vrstama kolekcija koje čine Javin okvir kolekcija te njihovim implementacijama, u ovom poglavlju istražit ćemo njihovu napredniju uporabu. Osnovno pitanje koje ovdje razmatramo jest: *što Javin okvir kolekcija očekuje od naših novih razreda, a da bi primjerci tih razreda mogli biti korišteni u kolekcijama*.

U sljedećih nekoliko primjeraka radit ćemo s modelom zaposlenika koji je prikazan u primjeru 14.1.

Primjer 14.1. Razred `Zaposlenik`

```
 1 package book.java.collections;
 2
 3 public class Zaposlenik {
 4
 5  private String sifra;
 6  private String ime;
 7  private String prezime;
 8  private double placa;
 9
10  public Zaposlenik(String sifra, String ime, String prezime) {
11   super();
12   if(sifra==null) {
13    throw new IllegalArgumentException(
14      "Šifra zaposlenika ne smije biti null!");
15   }
16   this.sifra = sifra;
17   this.ime = ime;
18   this.prezime = prezime;
19  }
20
21  public String getSifra() {
22   return sifra;
23  }
24
25  public String getIme() {
26   return ime;
27  }
28
29  public void setIme(String ime) {
30   this.ime = ime;
31  }
32
33  public String getPrezime() {
34   return prezime;
35  }
36
37  public void setPrezime(String prezime) {
38   this.prezime = prezime;
39  }
40
41  public double getPlaca() {
42   return placa;
43  }
44
```

```
45  public void setPlaca(double placa) {
46   this.placa = placa;
47  }
48
49  @Override
50  public String toString() {
51   return String.format(
52    "Zaposlenik: sifra=%s, prezime=%s, ime=%s, placa=%f",
53    sifra, prezime, ime, placa
54   );
55  }
56
57 }
```

Metoda `equals`

Prvi primjer koji ćemo istražiti odnosi se na uporabu lista. Konkretno, koristit ćemo implementaciju `java.util.ArrayList` i u takvu listu ćemo dodati nekoliko zaposlenika. Nakon toga, pokušat ćemo zaposlenike pronaći u toj listi. Demonstracijski program prikazan je u primjeru 14.2.

Primjer 14.2. `Zaposlenik` i `ArrayList`

```
 1 package book.java.collections;
 2
 3 import java.util.ArrayList;
 4 import java.util.List;
 5
 6 public class DohvatZaposlenika {
 7
 8  public static void main(String[] args) {
 9   List<Zaposlenik> lista = new ArrayList<>();
10
11   Zaposlenik prvi = new Zaposlenik("1", "Pero", "Perić");
12   lista.add(prvi);
13   lista.add(new Zaposlenik("2", "Agata", "Agić"));
14   lista.add(new Zaposlenik("3", "Ivana", "Ivić"));
15
16   // Ispiši zaposlenike:
17   for(Zaposlenik zaposlenik : lista) {
18    System.out.println(zaposlenik);
19   }
20
21   Zaposlenik trazeni = new Zaposlenik("1", "Pero", "Perić");
22
23   boolean postoji = lista.contains(prvi);
24   System.out.println(
25    "Zaposlenik "+prvi+" postoji: "+postoji
26   );
27
28   postoji = lista.contains(trazeni);
29   System.out.println(
30    "Zaposlenik "+trazeni+" postoji: "+postoji
31   );
32
33   int pozicija = lista.indexOf(prvi);
34   System.out.println(
```

```
35     "Zaposlenik "+prvi+" - pozicija: "+pozicija
36    );
37
38    pozicija = lista.indexOf(trazeni);
39    System.out.println(
40     "Zaposlenik "+trazeni+" - pozicija: "+pozicija
41    );
42   }
43
44 }
```

Program započinje stvaranjem prazne liste (redak 9). Potom se u listu dodaju tri primjerka razreda zaposlenik (retci 11 do 14), pri čemu u pomoćnoj varijabli `prvi` pamtimo referencu na prvog zaposlenika kojeg smo dodali u listu. Program nakon toga ispisuje sve zaposlenike iz liste koristeći kratki oblik petlje `for` i oslanjajući se na metodu `toString` razreda `Zaposlenik`.

U retku 21 stvaramo novi primjerak razreda zaposlenik koji predstavlja istog zaposlenika kojeg smo prethodno već dodali u listu (Peru Perića). Potom radimo četiri upita nad listom: (1) pitamo se sadrži li lista zaposlenika Peru Perića tako što pozivamo metodu `contains` i šaljemo joj referencu koju smo zapamtili u varijabli `prvi`; (2) pitamo se sadrži li lista zaposlenika Peru Perića tako što pozivamo metodu `contains` i šaljemo joj referencu koju smo zapamtili u varijabli `trazeni`; (3) pitamo se na kojoj je lokaciji u listi zaposlenika Pero Perić tako što pozivamo metodu `indexOf` i šaljemo joj referencu koju smo zapamtili u varijabli `prvi`; (4) pitamo se na kojoj je lokaciji u listi zaposlenika Pero Perić tako što pozivamo metodu `indexOf` i šaljemo joj referencu koju smo zapamtili u varijabli `trazeni`. Uočite, i varijabla `prvi` i varijabla `trazeni` pokazuju na primjerak razreda `Zaposlenik` koji predstavlja Peru Perića. Rezultat izvođenja programa prikazan je u nastavku.

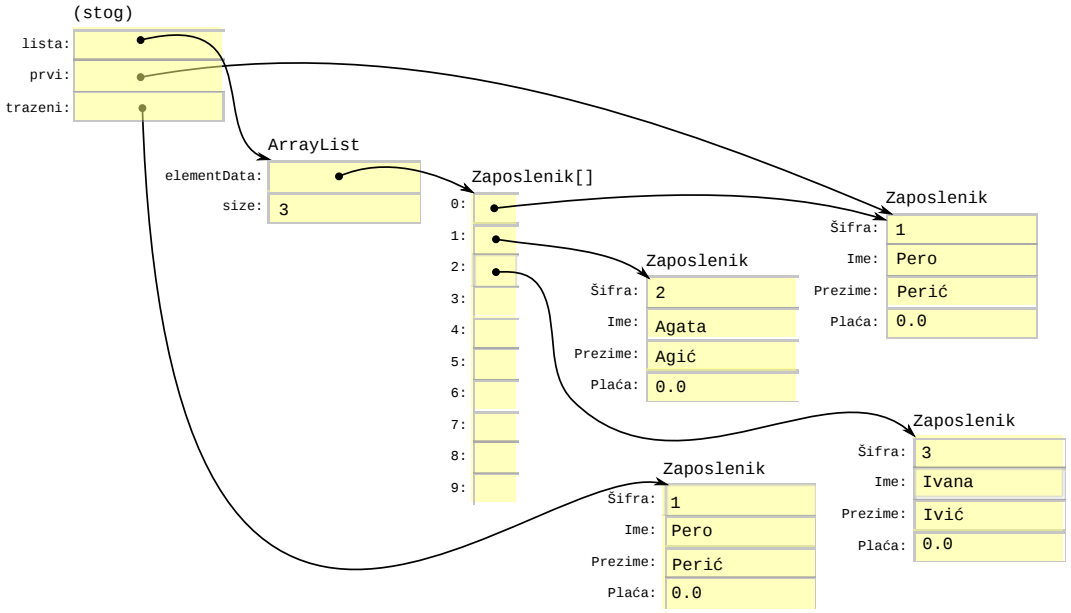
```
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: true
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: false
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 - pozicija: 0
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 - pozicija: -1
```

Posljednja četiri retka prelomljena su ručno za potrebe prikaza na ovom mjestu - program ih ispisuje kao četiri neprelomljena retka. Iz ispisa zaključujemo sljedeće. Prva tri retka nam kažu da lista na poziciji 0 sadrži Peru Perića, na poziciji 1 Agatu Agić te na poziciji 2 Ivanu Ivić. Sljedeća dva retka nam govore da Pero Perić je odnosno nije prisutan u listi. Posljednja dva retka kažu da je Pero Perić u listi prisutan na poziciji 0 odnosno da nije prisutan u listi. Na prvi pogled čini se da su rezultati kontradiktorni.

Kako bismo razjasnili što se događa, najprije ćemo skicirati stanje u memoriji za prikazani program a potom pogledati direktno u izvornom kodu razreda `ArrayList` kako su implementirane metode `contains` te `indexOf`.

Stanje u memoriji nakon stvaranja liste, punjenja zaposlenicima te nakon stvaranja traženog zaposlenika prikazano je na slici 14.1. U prikazanom programi iz primjera 14.2 to je stanje koje postoji nakon izvedenog retka 21.

Slika 14.1. Stanje u memoriji



Na stogu su do tog trenutka stvorene tri varijable: jedna referenca tipa te `List` te dvije reference tipa `Zaposlenik`. Varijabla `lista` je referenca koja pokazuje na objekt tipa `ArrayList` koji je stvoren na gomili. Taj objekt ima referencu na privatno polje referenci koje čini sadržaj liste i jednu varijablu tipa `int` u kojoj se čuva broj elemenata koji su pohranjeni u listi. U ovom konkretnom slučaju, privatno polje je kapaciteta 10 ali lista sadrži samo tri elementa na koje pokazuju prve tri reference polja. Na gomili još postoje alocirana četiri objekta koji su svi tipa `Zaposlenik` (stringovi su zbog preglednosti prikazani direktno upisani u objekte zaposlenika iako znamo da su u stvarnosti to zasebni objekti s referencama za znakovna polja koja čine njihov sadržaj).

Obratite pažnju da se u memoriji nalaze *četiri* objekta tipa `Zaposlenik` jer smo četiri puta operatorom `new` pozvali konstruktor razreda `Zaposlenik`. Dva od ta četiri objekta popunjena su identičnim sadržajem (predstavljaju zaposlenika Peru Perića).

Kako je u razredu `ArrayList` implementirana metoda `contains`? Evo isječka.

```
1 public boolean contains(Object o) {
2   return indexOf(o) >= 0;
3 }
```

Metoda se oslanja na metodu `indexOf`: nju pita da pronađe poziciju na kojoj se u listi nalazi traženi objekt, pa ako ta metoda vrati legalnu poziciju (0 ili veću), metoda vraća `true` čime signalizira da lista sadrži traženi element; u suprotnom (kada `indexOf` vrati -1 jer se traženi element ne nalazi niti na jednoj poziciji) metoda `contains` vraća `false` čime signalizira da lista ne sadrži traženi element. Vidimo da je ponašanje metode `contains` direktno određeno ponašanjem metode `indexOf` - čak im je i složenost jednaka. Budući da još uvijek nemamo dovoljno informacija da bismo otkrili zašto smo dobili onakav ispis, pogledajmo sada kako je ostvarena metoda `indexOf`.

```
1 public int indexOf(Object o) {
2     if (o == null) {
3         for (int i = 0; i < size; i++)
4             if (elementData[i]==null)
5                 return i;
6     } else {
7         for (int i = 0; i < size; i++)
8             if (o.equals(elementData[i]))
```

```
 9                  return i;
10      }
11      return -1;
12  }
```

Ova metoda razlikuje dva slučaja. Ako je traženi element jednak `null` (legalno je da lista sadrži elemente koji su `null`-reference), tada se odlazi u dio koda koji u petlji prolazi kroz privatno polje referenci i provjerava operatorom == je li koja od referenci `null`. Ako takva postoji, prva lokacija na kojoj je pronađena vraća se kao rezultat metode; ako ne postoji, vraća se -1. Zanimljiviji (i nama posebno interesantan) slučaj je ako element koji se traži nije `null`-referenca već doista neki objekt alociran u memoriji. U tom slučaju odlazi se u drugu granu u kojoj se ponovno prolazi kroz privatno polje referenci i gdje se za svaku referencu *nad traženim objektom poziva metoda `equals` kojoj se kao argument predaje referenca na trenutni element liste*. Ako metoda `equals` vrati `true`, pretraživanje se prekida i vraća se trenutna pozicija; u suprotnom, pretraživanje se nastavlja od sljedećeg elementa.

Za određivanje jednakosti, ova implementacija liste oslanja se dakle na metodu `equals` čija je zadaća utvrditi jesu li dva objekta jednaka ili nisu. Prisjetite se: metoda `equals` definirana je u razredu `java.lang.Object` i svi razredi je po nasljeđivanju također imaju definiranu. Upravo iz tog razloga, implementacija liste može se osloniti na postojanje te metode i koristiti je za izvođenje usporedbe. No zašto ta metoda kod naših zaposlenika ne radi dobro? Zavirimo direktno u izvorni kod razreda `java.lang.Object` da vidimo kako je tamo implementirana. Naime, s obzirom da je naš razred nije nadjačao, preuzeo je implementaciju koja je tamo definirana. A ta je implementacija prikazana u nastavku.

```
1 public class Object {
2      ...
3
4      public boolean equals(Object obj) {
5          return (this == obj);
6      }
7
8      ...
9
10 }
```

Osnovna metoda `equals` proglasit će dva objekta jednakima ako i samo ako se radi o istom objektu u memoriji (tj. ako obje reference, i `this` koja pokazuje na trenutni objekt i `obj` koja je dobivena kao argument metode pokazuju na istu memorijsku lokaciju).

I ovime je sve postalo jasno. Razred `Zaposlenik`, budući da nije nadjačao metodu `equals`, koristi implementaciju naslijeđenu iz razreda `java.lang.Object`. Ta implementacija dva objekta proglašava istima ako i samo ako se zapravo radi o jednom te istom objektu; čim su reference različite, metoda vraća `false`. Pogledajte sada ponovno sliku 14.1. U retku 23 demonstracijskog programa nad listom zovemo metodu `contains` kojoj predajemo referencu `prvi` a ta će metoda tu referencu proslijediti metodi `indexOf`. Ona će pak nad predanim objektom pozivati metodu `equals` s referencama koje čuva u svojem privatnom polju. Kako varijabla `prvi` pokazuje na isti objekt na koji pokazuje i prva referenca tog polja, kada se nad objektom na koji pokazuje `prvi` pozove `equals` s prvom referencom pohranjenom u polju, ta će metoda vratiti `true`: reference su iste i to je upravo taj objekt. Stoga će u demonstracijskom programu i `contains` i `indexOf` pozvani nad referencom `prvi` raditi u skladu s očekivanjem. Metode `contains` i `indexOf` pozvane nad referencom `trazeni` također će raditi u skladu s očekivanjem - ako sada razumijemo što trebamo očekivati. Doista, objekt na koji pokazuje varijabla `trazeni` ne postoji u listi.

Ponekad, umjesto ovakve semantike *jednakosti po identitetu* želimo nad objektima definirati *jednakost po sadržaju* (koju još zovemo *domenska jednakost*). Metoda `equals` upravo je prirodno mjesto za definiranje željene jednakosti. Za svaki razred, programer se može odlučiti hoće li se za primjerke tog razreda koristiti jednakost po identitetu ili jednakost po sadržaju na

način da se odluči želi li koristiti naslijeđenu implementaciju metode equals ili želi ponuditi novu implementaciju koja će definirati pravila prema kojima se utvrđuje kada je primjerak tog razreda jednak primjerku koji je dobiven kao argument.

Za potrebe primjera koji analiziramo, pretpostavimo da želimo imati domensku jednakost zaposlenika. Sada se još samo moramo dogovoriti što to točno znači. Da bi dva zaposlenika bila jednaka, je li dovoljno da imaju jednake šifre, ili moraju imati imati sve podatke jednake? Što ako imamo dva zaposlenika koja imaju istu šifru ali ostali podatci nisu jednaki? To je primjer nekonzistentnih podataka, i u ovom slučaju bi bilo bolje da taj problem ne rješavamo u metodi equals već na mjestu unosa i ažuriranja podataka. Nadalje, u našem primjeru za šifru koristimo varijablu tipa String. Ako su šifre niz znamenaka (o tome nismo do sada razmišljali), imaju li ti nizovi znamenki semantiku broja ili nemaju? Konkretno, jesu li šifre "0123" i "123" iste ili nisu? To su sve važna pitanja koja spadaju u domenu modeliranja objekata koji čine model domene a koji gradimo kako bismo riješili zadani problem za koji pišemo program. Pretpostavimo za trenutak da su dva zaposlenika ista ako su im šifre iste te da šifre nemaju semantiku broja. U tom slučaju, u razred Zaposlenik nadjačali bismo metodu equals kako je prikazano u nastavku.

```
 1 @Override
 2 public boolean equals(Object obj) {
 3  if(obj==null) return false;
 4  if(!(obj instanceof Zaposlenik)) return false;
 5  Zaposlenik drugi = (Zaposlenik)obj;
 6  if(this.sifra==null) {
 7   if(drugi.sifra==null) {
 8    return true;
 9   } else {
10    return false;
11   }
12  } else if(drugi.sifra==null) {
13   return false;
14  }
15  return this.sifra.equals(drugi.sifra);
16 }
```

U retku 3 najprije ispitujemo jesmo li dobili referencu null. Ako jesmo, s obzirom da trenutni objekt postoji (this nije null), onda this i obj sigurno nisu isti pa vraćamo false. U retku 4 provjeravamo je li objekt s kojim se moramo usporediti barem primjerak razreda Zaposlenik za što koristimo konstrukt *referenca* instanceof *razreda*. Operator instanceof vraća true ako je objekt čiju smo referencu naveli s lijeve strane primjerak razreda koji je naveden s desne strane ili primjerak razreda koji je izravno ili neizravno izveden iz tog razreda (zato kažemo: je li *barem* Zaposlenik). U retku 4 zapravo radimo obrnuti test: ako nije barem Zaposlenik, onda nismo jednaki i metoda opet vraća false. Da nas je zanimalo je li taj objekt baš Zaposlenik (i viša više od toga), mogli smo usporediti opisnike razreda: obj.getClass() == Zaposlenik.class. Za svaki razred koji je učitan u memoriju virtualnog stroja u trenutku učitavanja isti stvara opisnik tog razreda. Do reference na opisnik razreda nad objektima možemo doći pozivom nestatičke metode getClass a preko imena razreda možemo doći dodavanjem .class. Napomenimo da null referenca nije primjerak niti jednog razreda pa to možemo iskoristiti kako bismo izbacili eksplicitno ispitivanje u retku 3.

Ako je izvođenje preživjelo do retka 5, sada znamo da je objekt s kojim se moramo usporediti barem Zaposlenik pa na njega možemo gledati kroz prizmu razreda Zaposlenik; stoga stvaramo pomoćnu lokalnu varijablu koja je referenca tipa Zaposlenik i obj ukalupljujemo u taj tip, što će proći bez izazivanja iznimke jer smo provjerili da je takvo ukalupljivanje legalno.

Sada kada je this referenca na trenutnog zaposlenika a drugi referenca na predanog zaposlenika moramo ispitati jesu li jednaki i tu se moramo držati dogovorenih pravila (rekli smo da su jednaki ako su im šifre jednake). Retci 6 do 14 definiraju jednakost za slučaj da je jedna od šifri null. Ako su šifre u oba zaposlenika različite od null, onda redak 15 nad širfama po-

ziva metodu equals i jednakost zaposlenika utvrđuje na temelju rezultata koji vrati ta metoda. Kako su šifre stringovi, oni već imaju nadjačanu metodu equals koja će dva stringa proglasiti jednakima ako su im znakovni sadržaji jednaki.

Prethodna metoda uz prethodno spomenuto ponašanje operatora instanceof i reference null te uz uporabu pomoćnog razreda java.util.Objects koji je dodan od Jave 8 može se ostvariti puno kraće, kako je prikazano u nastavku.

Pokrenete li sada demonstracijski program, prva dva poziva će oba vratiti true a druga dva poziciju 0.

```
1 @Override
2 public boolean equals(Object obj) {
3  if(!(obj instanceof Zaposlenik)) return false;
4  Zaposlenik drugi = (Zaposlenik)obj;
5  return Objects.equals(this.sifra, drugi.sifra);
6 }
```

Naime, metoda Objects.equals(x,y) već sadrži logiku za usporedbu referenci null (dvije reference null se tretiraju jednakima; ako je jedna null a druga nije, onda su različite) i tek ako su obje reference različite od null, metoda usporedbu delegira metodi equals prvog objekta.

Da smo zaposlenike htjeli proglasiti jednakima samo ako su im sifre, imena i prezimena jednaki, koristili bismo metodu prikazanu u nastavku.

```
1 @Override
2 public boolean equals(Object obj) {
3  if(!(obj instanceof Zaposlenik)) return false;
4  Zaposlenik drugi = (Zaposlenik)obj;
5  return Objects.equals(this.sifra, drugi.sifra)
6    && Objects.equals(this.ime, drugi.ime)
7    && Objects.equals(this.prezime, drugi.prezime);
8 }
```



Vrste jednakosti

Kada govorimo o jednakosti objekata, možemo razmatrati jednakost po identitetu ili jednakost po sadržaju (domensku jednakost). Kakva se jednakost koristi određuje metoda equals. Svi razredi inicijalno koriste jednakost po identitetu. Ako se želi koristiti jednakost po sadržaju, razred treba nadjačati metodu equals i implementirati pravila usporedbe.

Niz gotovih implementacija Javinog okvira kolekcija oslanja se na metodu equals za potrebe utvrđivanja jednakosti. Planirate li primjerke Vaših razreda koristiti u kolekcijama, razmislite odgovara li Vam jednakost po identitetu te ako ne, nadjačajte metodu equals i definirajte pravila usporedbe.

Metoda hashCode

Naše istraživanje implementacija kolekcija nastavit ćemo s dva nova demonstracijska programa koji su prikazani u primjerima 14.3 i 14.4. Prvi program uz iste zaposlenike kao u prethodnom podpoglavlju stvara skup (implementaciju HashSet), u skup dodaje zaposlenike i potom pokušava provjeriti je li Pero Perić u tom skupu.

Primjer 14.3. Zaposlenik i HashSet

```
1 package book.java.collections;
```

```
 2
 3 import java.util.HashSet;
 4 import java.util.Set;
 5
 6 public class DohvatZaposlenika2 {
 7
 8  public static void main(String[] args) {
 9   Set<Zaposlenik> lista = new HashSet<>();
10
11   Zaposlenik prvi = new Zaposlenik("1", "Pero", "Perić");
12   lista.add(prvi);
13   lista.add(new Zaposlenik("2", "Agata", "Agić"));
14   lista.add(new Zaposlenik("3", "Ivana", "Ivić"));
15
16   for(Zaposlenik zaposlenik : lista) {
17    System.out.println(zaposlenik);
18   }
19
20   Zaposlenik trazeni = new Zaposlenik("1", "Pero", "Perić");
21
22   boolean postoji = lista.contains(prvi);
23   System.out.println(
24    "Zaposlenik "+prvi+" postoji: "+postoji
25   );
26
27   postoji = lista.contains(trazeni);
28   System.out.println(
29    "Zaposlenik "+trazeni+" postoji: "+postoji
30   );
31
32  }
33
34 }
```

Pokretanjem programa dobit ćemo sljedeći ispis.

```
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: true
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: false
```

Čini se da Pero Perić i je i nije u skupu. Sada, međutim, razlog ne može biti metoda `equals` jer smo je u razredu `Zaposlenik` korektno implementirali.

Drugi spomenuti primjer koristi mapu kako bi za svakog djelatnika zapisao bonus na plaću koji će ostvariti.

Primjer 14.4. `Zaposlenik` i `HashMap`

```
 1 package book.java.collections;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5
 6 public class Bonusi {
 7
```

```
 8  public static void main(String[] args) {
 9   Map<Zaposlenik, Double> bonusi = new HashMap<>();
10
11   Zaposlenik prvi = new Zaposlenik("1", "Pero", "Perić");
12   bonusi.put(prvi, 100.0);
13   bonusi.put(new Zaposlenik("2", "Agata", "Agić"), 200.0);
14   bonusi.put(new Zaposlenik("3", "Ivana", "Ivić"), 300.0);
15
16   for(Zaposlenik zap : bonusi.keySet()) {
17    System.out.printf(
18     "Zaposlenik %s ima bonus %f%n",
19     zap.toString(),
20     bonusi.get(zap)
21    );
22   }
23
24   System.out.println("---------------------------------");
25
26   Zaposlenik trazeni = new Zaposlenik("1", "Pero", "Perić");
27
28   System.out.printf(
29    "Zaposlenik %s ima bonus %f%n",
30    prvi.toString(),
31    bonusi.get(prvi)
32   );
33
34   System.out.printf(
35    "Zaposlenik %s ima bonus %f%n",
36    trazeni.toString(),
37    bonusi.get(trazeni)
38   );
39  }
40
41 }
42
```

Pokretanjem ovog programa dobit ćemo ispis koji je prikazan u nastavku.

```
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 ima bonus 100.000000
Zaposlenik Zaposlenik: sifra=2, prezime=Agić, ime=Agata,
  placa=0.000000 ima bonus 200.000000
Zaposlenik Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana,
  placa=0.000000 ima bonus 300.000000
---------------------------------
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 ima bonus 100.000000
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 ima bonus null
```

Čini se da i ovdje postoje problemi. Naime, Pero Perić u jednom slučaju ima pridijeljen bonus a u drugom slučaju nema, iako bonus čitamo iz iste mape. U ovom slučaju zaposlenici se u mapi koriste kao ključevi.

Što je zajedničko za obje kolekcije se koriste u primjerima 14.3 i 14.4? Odgovor je: podatkovna struktura koju obje implementacije koriste, a riječ je o tablici raspršenog adresiranja. Konceptualno, mogli bismo razmišljati ovako: u slučaju razreda `HashSet` u tablicu se direktno upisuju elementi koje dodajemo u skup i njihov položaj u tablici određen je samim elementima; u slučaju razreda `HashMap` u tablicu se umeću uređeni parovi pri čemu je položaj uređenog para

u tablici određen ključem uređenog para. Implementacijski, ova dva razreda dijele jako puno koda: razred `HashSet` ima privatni primjerak razreda `HashMap` i elemente skupa delegira toj mapi kao ključeve; sva ostala funkcionalnost ove implementacije skupa ostvaruje se pozivima odgovarajućih metoda nad tom privatnom mapom. I upravo to je razlog da smo ove dvije naoko potpuno različite vrste kolekcija odlučili istražiti zajedno.

Prisjetimo se kako se implementira tablica raspršenog adresiranja. Pretpostavimo da radimo s implementacijom koja koristi ulančane liste za rješavanje preljevne politike. To znači da je tablica raspršenog adresiranja polje referenci na glave lista: svaki slot tablice započinje listu koja čuva objekte koji su smješteni u taj slot. Kada u tablicu želimo dodati neki objekt (ili ga pronaći), za taj objekt moramo izračunati sažetak (cijeli broj) koji potom transformiramo u redni broj slota (primjerice, operatorom modulo `adresa = sazetak % tablica.length`). Tako izračunata adresa označava slot u koji taj objekt treba smjestiti odnosno ako ga tražimo, slot u kojem treba provjeriti je li objekt prisutan. Na elementarnijim kolegijima koji se bave tematikom programiranja vjerojatno ste se već upoznali s različitim metodama kako za različite tipove podataka računati sažetke; npr. kako se to radi za stringove. Kako Javin okvir kolekcija radi s proizvoljnom vrstom objekata, rješenje koje trebamo mora osigurati da izračun sažetaka radi neovisno o tipu objekta - barem što se tiče implementacije kolekcije.

Rješenje koje je implementirano u Javi oslanja se na dinamički polimorfizam i nasljeđivanje. Kako kolekcije moraju moći raditi s proizvoljnom vrstom objekata, u razredu `java.lang.Object` definirana je nestatička metoda `int hashCode()` čija je zadaća izračun sažetka objekta. U tom razredu metoda je implementirana (konceptualno) na način da se vraća sažetak adrese (reference) samog objekta. Kako su iz ovog razreda u Javi izvedeni svi ostali razredi (izravno ili neizravno), primjerak bilo kojeg razreda sada ima tu metodu i Javin okvir kolekcija je može koristiti.

Dakako, spomenuta implementacija metode nije univerzalno dobra. Ako su objekti stringovi, onda bismo htjeli da sažetak tog objekta ovisi o znakovnom sadržaju stringa a ne o memorijskoj lokaciji na kojoj je objekt pohranjen. Razred `java.lang.String` stoga nadjačava metodu `hashCode` i vraća sažetak koji ovisi isključivo o znakovnom sadržaju koji string čuva. Iz tog razloga, sljedeći isječak će u varijablu `isti` uvijek upisati vrijednost `true`.

```
1 int sazetak1 = new String("Perica").hashCode();
2 int sazetak2 = new String("Perica").hashCode();
3 boolean isti = sazetak1 == sazetak2;
```

Naši zaposlenici imaju isti problem. Kako u razredu `Zaposlenik` nismo nadjačali metodu `hashCode`, koristi se ona naslijeđena iz razreda `java.lang.Object`. Posljedica je da poziv metode `hashCode` nad objektom čiju referencu imamo u varijabli `prvi` te poziv te iste metode nad objektom čiju referencu imamo u varijabli `trazeni` ne moraju vratiti isti rezultat: iako su oba objekta konceptualno Pero Perić, to su dva objekta na različitim memorijskim adresama i sažetak im može biti različit. Uz različit sažetak izračunata adresa slota u tablici raspršenog adresiranja više ne mora biti jednaka adresi koju smo koristili prvi puta; a ako je to slučaj, onda ćemo objekt tražiti u krivom pretincu i tu nam više neće pomoći niti korektna implementacija metode `equals` koja se unutar pretinca koristi da bi se provjerilo jesmo li pronašli traženi objekt. Stoga nam ne preostaje ništa drugo doli nadjačati i tu metodu u razredu `Zaposlenik`.

Kojih se pravila pri tome moramo držati, odnosno koja svojstva ima funkcija za izračun sažetka? Pretpostavimo da su `o1` i `o2` reference ne neka dva objekta.

1. Ako vrijedi `o1.equals(o2)`, tada mora vrijediti `o1.hashCode() == o2.hashCode()`. Drugim riječima, ako su objekti jednaki (u smislu koji propisuje njihova implementacija metode `equals`), tada moraju imati i jednake sažetke.

2. Ako vrijedi `!o1.equals(o2)`, tada nema nikakvih ograničenja na odnos između `o1.hashCode()` i `o2.hashCode()`. Drugim riječima, ako objekti nisu jednaki (u smislu koji propisuje njihova implementacija metode `equals`), sažetci im mogu i ne moraju biti jednaki.

3. Ako objekti imaju različite sažetke, tj. `o1.hashCode() != o2.hashCode()`, tada ti objekti nužno moraju biti različiti. Ovo slijedi iz zahtjeva (1). Da su bili jednaki, morali bi imati i jednake sažetke.

4. Konačno, ako dva objekta imaju jednake sažetke, tj. `o1.hashCode() == o2.hashCode()`, tada ništa ne možemo zaključiti o njihovom odnosu: oni mogu biti jednaki ali i ne moraju. Ovo proizlazi iz poznatog teorema o zečevima i kavezima. Ako imamo *n* kaveza u koje moramo rasporediti *n+1* zeca, tada, kakav god raspored da radili, barem ćemo u jednom kavezu imati barem dva zeca. To je posljedica činjenice da je zečeva više no kaveza pa ne možemo osigurati da u svaki kavez stavimo najviše jednog zeca. U konkretnom primjeru, objekti su zečevi i različitih objekata može biti praktički beskonačno mnogo. Kavezi su različite vrijednosti koje može poprimiti funkcija sažetka. Kako metoda `hashCode` kao povratnu vrijednost deklarira tip `int`, slijedi da različitih sažetaka ima `2^32` odnosno oko 4.3 milijarde. Kako je različitih objekata beskonačno mnogo, tada će ih iz za svaki različit sažetak (u prosjeku) biti beskonačno mnogo.

Spomenimo na ovom mjestu da se sažetak može koristiti kao pomoć za implementaciju funkcije `equals`. Postoje situacije u kojima radimo s nepromjenjivim objektima koji su međutim bogate strukture: sadrže puno podatkovnih varijabli od kojih su neki čak i kolekcije. Ispitivanje jednakosti dvaju takvih objekata tada je vrlo skupo: treba raditi usporedbu član po član i kolekciju po kolekciju što ovisno o kolekciji može značiti čak i rekurziju. Kada su takvi objekti nepromjenjivi (podatke primaju kroz konstruktor i potom nude samo metode za čitanje), objektima se može dodati još jedna pomoćna članska varijabla u koju će se pohraniti sažetak objekta koji će se računati odmah u konstruktoru (taj će izračun također biti skup). Nadjačana metoda `hashCode` tada će biti trivijalna: vraćat će sažetak koji je zapisan u tu pomoćnu varijablu. A pravi bonus nastupa u metodi `equals`: nakon što provjerimo jesu li objekti po tipu jednaki, možemo provjeriti kakvi su im sažetci. Ako su sažetci različiti (a to se ispitivanje sada svodi doslovno na usporedbu dva cijela broja), objekti nisu jednaki i možemo van iz metode. Tek ako su sažetci jednaki, možemo nastaviti sa skupom usporedbom po članovima. Imamo li dobru funkciju za izračun sažetka, ovaj će pristup za mnoštvo parova objekata koji su različiti to odmah utvrditi a detaljno ispitivanje bit će rađeno tek za mali broj slučajeva (i naravno, za najgori slučaj kada objekti doista jesu jednaki).

Oboružani ovim spoznajama (zahtjevima 1-4), pogledajmo sada kako bismo tu metodu mogli implementirati za našeg zaposlenika. Uz pretpostavku da smo metodu `equals` implementirali tako da gleda samo šifru, pripadna metoda `hashCode` mogla bi izgledati ovako.

```
1 @Override
2 public int hashCode() {
3  return ((sifra == null) ? 0 : sifra.hashCode());
4 }
```

Uočite da pri tome izračun sažetka šifre možemo delegirati razredu `String` koji to već zna raditi. Prilikom nadjačavanja metode `hashCode` u našim razredima najčešće ćemo se samo morati pobrinuti kako iskombinirati sažetke jednostavnih članskih varijabli.

Da je metoda `equals` u obzir uzimala šifru, ime i prezime, sažetak bismo gradili iz tih elemenata. Jedan loš primjer prikazan je u nastavku.

```
1 @Override
2 public int hashCode() {
3   return ((sifra == null) ? 0 : sifra.hashCode()) +
4          ((ime == null) ? 0 : ime.hashCode()) +
5          ((prezime == null) ? 0 : prezime.hashCode());
6 }
```

Ovakvo aditivno spajanje sažetaka ima niz loših svojstava pa je bolje sažetak računati na neki složeniji način. Okruženja za razvoj programa (poput okruženja Eclipse) nudi čarobnjaka za upareno stvaranje metoda `equals` i `hashCode`: korisnik samo treba odabrati koje se članske

varijable uzimaju u obzir. Metoda `hashCode` koja bi nastala za ovaj posljednji slučaj prikazana je u nastavku.

```
@Override
public int hashCode() {
 final int prime = 31;
 int result = 1;
 result = prime * result +
  ((ime == null) ? 0 : ime.hashCode());
 result = prime * result +
  ((prezime == null) ? 0 : prezime.hashCode());
 result = prime * result +
  ((sifra == null) ? 0 : sifra.hashCode());
 return result;
}
```

Uočite da se u izračunu koriste prosti brojevi i množenje uz standardnu operaciju zbrajanja.

Pokrenete li sada programe iz primjera 14.3 i 14.4, uvjerit ćete se da sve radi za oba primjerka Pere Perića.



Metode `equals` i `hashCode`

Implementacije koje koriste tablice raspršenog adresiranja oslanjaju se na metode `equals` i `hashCode`. Radite li nove razrede čije ćete primjerke koristiti u kolekcijama, odlučite se za prikladnu definiciju jednakosti objekata te prema toj definiciji izvedite ove obje metode. Budite svjesni da postoji ugovor između te dvije metode koji morate poštivati.

Prirodan poredak i komparatori

Ostalo nam je za razmotriti još jedan demonstracijski program koji je prikazan u primjeru 14.5.

Primjer 14.5. `Zaposlenik` i `TreeSet`

```
package book.java.collections;

import java.util.Set;
import java.util.TreeSet;

public class DohvatZaposlenika3 {

 public static void main(String[] args) {
  Set<Zaposlenik> set = new TreeSet<>();

  Zaposlenik prvi = new Zaposlenik("1", "Pero", "Perić");
  set.add(prvi);
  set.add(new Zaposlenik("2", "Agata", "Agić"));
  set.add(new Zaposlenik("3", "Ivana", "Ivić"));

  for(Zaposlenik zaposlenik : set) {
   System.out.println(zaposlenik);
  }

  Zaposlenik trazeni = new Zaposlenik("1", "Pero", "Perić");

  boolean postoji = set.contains(prvi);
  System.out.println(
```

```
24     "Zaposlenik "+prvi+" postoji: "+postoji
25    );
26
27    postoji = set.contains(trazeni);
28    System.out.println(
29     "Zaposlenik "+trazeni+" postoji: "+postoji
30    );
31
32   }
33
34 }
35
```

Pokretanjem programa dobit ćemo sljedeći ispis.

```
Exception in thread "main" java.lang.ClassCastException:
  book.java.collections.Zaposlenik cannot be cast
  to java.lang.Comparable
 at java.util.TreeMap.compare(TreeMap.java:1290)
 at java.util.TreeMap.put(TreeMap.java:538)
 at java.util.TreeSet.add(TreeSet.java:255)
 at book.java.collections.DohvatZaposlenika3.main(
  DohvatZaposlenika3.java:12)
```

Upravo samo napisali program koji se srušio. O čemu se radi i što nam dobivena iznimka govori? Ovaj primjer zaposlenike stavlja u skup pri čemu smo kao implementaciju odabrali razred `TreeSet`. Ako se prisjetite kako je taj razred implementiran, morali biste razumjeti problem: ova implementacija nad zaposlenicima pokušava izgraditi uređeno binarno stablo. Da bi to mogla, mora znati usporediti dva zaposlenika i utvrditi u kakvom su odnosu: je li prvi manji od drugog, jednak drugome ili je veći od drugog. Budući da razred `TreeSet` ne zna kako bi usporedio zaposlenike, koristi određene pretpostavke koje ovdje nisu bile ispunjene i od tuda iznimka.

Kada razmatramo problem usporedbe dva objekta (u smislu manji/jednak/veći), dva su tipična pristupa: ili se objekti sami znaju usporediti ili implementaciji šaljemo referencu na treći objekt koji nudi uslugu usporedbe objekata pa implementacija može taj objekt pitati da odradi posao uspoređivanja kad god je to potrebno. Ako se objekti sami znaju međusobno uspoređivati, onda kažemo da je nad objektima definiran *prirodni poredak*. Način na koji razred definira prirodni poredak jest implementiranjem parametriziranog sučelja `java.lang.Comparable<T>` koje je prikazano u nastavku.

Primjer 14.6. Sučelje `java.lang.Comparable<T>`

```
1 public interface Comparable<T> {
2     public int compareTo(T o);
3 }
```

Metoda `compare` mora vratiti broj koji je negativan ako je `this` manji od `o`, nulu ako su `this` i `o` jednaki te broj koji je pozitivan ako je `this` veće od `o`. Sam kriterij usporedbe odgovornost je razreda koji implementira tu metodu. Primjerice, razred `java.lang.String` implementira ovo sučelje i stringove uspoređuje leksikografski; razred `java.lang.Integer` također implementira ovo sučelje i usporedbu radi prema vrijednosti broja. U slučaju zaposlenika, odgovornost je opet na nama da se odlučimo kako ćemo definirati uređaj između zaposlenika. Za potrebe ovog primjera, neka se usporedba radi prema šiframa: manja šifra znači "manjeg" zaposlenika. Razred `Zaposlenik` modificirat ćemo na sljedeći način (prikazani su samo novi dijelovi).

```
1 public class Zaposlenik implements Comparable<Zaposlenik> {
2
3  ...
4
```

```
 5   @Override
 6   public int compareTo(Zaposlenik o) {
 7    return this.sifra.compareTo(o.sifra);
 8   }
 9
10 }
```

Konkretnu provjeru odnosa dviju šifri prepustili smo razredu `String`. Uočite da bi metoda `compareTo` trebala vraćati vrijednost 0 samo su objekti jednaki u skladu s načinom na koji je to implementirano i u metodi `equals`; te dvije metode vezane su ugovorom koji treba poštivati.

Uz ovakvu modifikaciju kod iz primjera 14.5 radi korektno i generira sljedeći ispis.

```
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: true
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
  placa=0.000000 postoji: true
```

Pokušajte sada napraviti modifikaciju programa 14.5: najprije u skup dodajte Agatu, potom Ivanu i na kraju Peru pa pokrenite program. Je li se ispis (prva tri retka) promijenio? Odgovor je: nije. Opet je naprije ispisan Pero, potom Agata i na kraju Ivana, Razlog je činjenica da iterator koji dobivamo kod implementacije `TreeSet` elemente obilazi sortiranim poretkom. Kako smo definirali da je prirodan poredak zaposlenika određen šifrom, oni se u stablo dodaju sortirani upravo na taj način pa će ispis uvijek biti sortiran po šifri.

Razlog pojave iznimke pri prvom pokušaja pokretanja je pretpostavka razreda `TreeSet` da zaposlenici imaju prirodan poredak (s obzirom da konstruktoru nismo predali referencu na objekt koji zna raditi usporedbu). Da bi mogao pozvati metodu `compareTo`, `TreeSet` je primjerak zaposlenika pokušao ukalupiti u `java.lang.Comparable`, i tu je sve puklo. Nakon što smo zaposlenicima definirali prirodni poredak, više nema ovih problema.

Što učiniti ako nam prirodni poredak objekata ne odgovara? Razred `TreeSet` ima i konstruktor kojem možemo predati referencu na vanjski objekt kojeg treba pitati da uspoređuje zaposlenike. Taj objekt mora biti primjerak razreda koji implementira sučelje `java.util.Comparator` koje je prikazano u primjeru 14.7 (zbog preglednosti prikazan je samo dio pretpostavljenih metoda).

Primjer 14.7. Sučelje `java.util.Comparator<T>`

```
 1 @FunctionalInterface
 2 public interface Comparator<T> {
 3
 4     int compare(T o1, T o2);
 5
 6     default Comparator<T> reversed() {
 7       ...
 8     }
 9
10     default Comparator<T> thenComparing(
11         Comparator<? super T> other) {
12       ...
13     }
14
15     default <U> Comparator<T> thenComparing(
16             Function<? super T, ? extends U> keyExtractor,
17             Comparator<? super U> keyComparator)
18     {
```

```
19        ...
20      }
21
22      default <U extends Comparable<? super U>>
23         Comparator<T> thenComparing(
24              Function<? super T, ? extends U> keyExtractor)
25      {
26        ...
27      }
28
29      ...
30
31      public static <T extends Comparable<? super T>>
32         Comparator<T> reverseOrder() {
33        ...
34      }
35
36      public static <T extends Comparable<? super T>>
37         Comparator<T> naturalOrder() {
38        ...
39      }
40
41      public static <T, U> Comparator<T> comparing(
42              Function<? super T, ? extends U> keyExtractor,
43              Comparator<? super U> keyComparator)
44      {
45        ...
46      }
47
48      public static <T, U extends Comparable<? super U>>
49         Comparator<T> comparing(
50              Function<? super T, ? extends U> keyExtractor)
51      {
52        ...
53      }
54  }
```

Sučelje je parametrizirano tipom objekata koje uspoređuje i navodi samo jednu apstraktnu metodu: `int compare(T o1, T o2);`. Kako se ovdje radi o vanjskom objektu, on kroz tu metodu mora dobiti referencu na objekta te kroz povratnu vrijednost dojavljuje kakav je njihov odnos. Ugovor je isti kao i kod metode `compareTo` sučelja `Comparable`: povratna vrijednost mora biti negativna, nula i pozitivna ovisno o tome je li `o1` manji od `o2`, jesu li jednaki ili je `o1` veći od `o2`. Također, povratna vrijednost mora odgovarati načinu na koji je implementirana metoda `equals` (za jednake objekte `compare` mora vratiti 0).

Metodu `main` primjera sada možemo preraditi tako da konstruktoru od `TreeSet` predamo referencu na jedan vanjski komparator. Primjer 14.8 to ilustrira.

Primjer 14.8. Uporaba komparatora

```
1  package book.java.collections;
2
3  import java.util.Comparator;
4  import java.util.Set;
5  import java.util.TreeSet;
6
7  public class DohvatZaposlenika3b {
8
```

```
 9  public static class UsporedbaPoSiframaRev implements
10   Comparator<Zaposlenik> {
11   @Override
12   public int compare(Zaposlenik o1, Zaposlenik o2) {
13    return -o1.getSifra().compareTo(o2.getSifra());
14   }
15  }
16
17
18  public static void main(String[] args) {
19   Set<Zaposlenik> lista = new TreeSet<>(
20    new UsporedbaPoSiframaRev()
21   );
22
23   Zaposlenik prvi = new Zaposlenik("1", "Pero", "Perić");
24   lista.add(prvi);
25   lista.add(new Zaposlenik("2", "Agata", "Agić"));
26   lista.add(new Zaposlenik("3", "Ivana", "Ivić"));
27
28   for(Zaposlenik zaposlenik : lista) {
29    System.out.println(zaposlenik);
30   }
31
32   Zaposlenik trazeni = new Zaposlenik("1", "Pero", "Perić");
33   boolean postoji = lista.contains(trazeni);
34   System.out.println(
35    "Zaposlenik "+trazeni+" postoji: "+postoji
36   );
37
38  }
39
40 }
41
```

Počev od retka 9 definiramo razred `UsporedbaPoSiframaRev` koji implementira sučelje `Comparator<Zaposlenik>` i koji u metodi `compare` radi usporedbu zaposlenika prema šiframa i to obrnuto (reverzno): manja šifra znači većeg zaposlenika. U metodi `main` pozivamo konstruktor razreda `TreeSet` i kao argument mu predajemo primjerak upravo definiranog komparatora (retci 19-21). Posljedica je da implementacija skupa sada zaposlenike ne uspoređuje prema njihovom prirodnom poretku (iako on postoji) već koristi zadani komparator. Rezultat toga jasno je vidljiv u ispisu: iterator je sada zaposlenike dohvaćao od onog s najvećom šifrom.

Pokretanjem programa iz primjera 14.8 generirat će se sljedeći ispis.

```
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000
Zaposlenik Zaposlenik: sifra=1, prezime=Perić, ime=Pero,
   placa=0.000000 postoji: true
```



Implementacije kolekcija koje koriste stabla

Implementacije kolekcija koje koriste stabla trebaju način za usporedbu objekata u smislu manji/jednak/veći. U Javi, dva su uobičajena načina kako se to postiže.

Prvi način je oslanjajući se na *prirodan poredak*. Razred definira prirodan poredak implementiranjem sučelja `java.lang.Comparable<T>`. Drugi način je predavanjem reference na vanjski komparator. Komparator je objekt s kojim se može razgovarati kroz sučelje `java.util.Comparator<T>`.

Konstruktori razreda implementacija kolekcija koji ne primaju referencu na vanjski komparator (a trebaju funkcionalnost usporedbe) pretpostavljaju da objekti imaju definiran prirodan poredak (što ako nije ispunjeno, može rezultirati pojavom iznimke).

Razred `java.util.TreeMap` mogućnost usporedbe zahtjeva nad ključevima i uređene parove sortira prema uređaju ključeva.

Još malo o komparatorima

Primjer komparatora vrlo je plodan za ilustraciju još nekoliko koncepata pa ćemo ih razraditi u nastavku.

Izravna i reverzna usporedba

Razmotrite ponovno razred `Zaposlenik`. Imamo četiri članske varijable: ime, prezime, plaću te šifru. Ako za svaku od tih varijabli želimo biti u mogućnosti raditi usporedbu (po imenu od 'A' do 'Ž', po imenu od 'Ž' do 'A'; po prezimenu od ...), završit ćemo s 8 razreda: za svaku člansku varijablu imat ćemo dva razreda - komparator koji uspoređuje "u jednom smjeru" i komparator koji to radi u obrnutom smjeru. Kako su 4 varijable, imamo 8 razreda. Trebamo li ih doista toliko? Odgovor je, dakako, ne. Umjesto da za svaku varijablu radimo po dva komparatora, možemo napraviti samo jedan (koji radi "normalnu" usporedbu) te jedan zajednički razred *koji laže*: dobije referencu na jedan komparator, pita ga za rezultat usporedbe i potom vrati negiranu vrijednost; ako je izvorni rekao *manje*, mi ćemo reći *veće*. Time ćemo zadatak riješiti sa samo 5 razreda, odnosno općenito *n+1* gdje je *n* broj članskih varijabli.

Program koji ovo ilustrira prikazan je u nastavku (primjer 14.9).

Primjer 14.9. Reverzni komparator

```
 1 package book.java.collections;
 2
 3 import java.util.Arrays;
 4 import java.util.Comparator;
 5 import java.util.List;
 6 import java.util.Set;
 7 import java.util.TreeSet;
 8
 9 public class PrimjerK1 {
10
11  public static class UsporedbaPoSiframa
12    implements Comparator<Zaposlenik> {
13   @Override
14   public int compare(Zaposlenik o1, Zaposlenik o2) {
15    return o1.getSifra().compareTo(o2.getSifra());
16   }
17  }
18
19  public static class UsporedbaPoImenu
20    implements Comparator<Zaposlenik> {
21   @Override
22   public int compare(Zaposlenik o1, Zaposlenik o2) {
23    return o1.getIme().compareTo(o2.getIme());
24   }
25  }
26
27  public static class Reverzni<T> implements Comparator<T> {
```

```
28   private Comparator<T> originalni;
29
30   public Reverzni(Comparator<T> originalni) {
31    super();
32    this.originalni = originalni;
33   }
34
35   @Override
36   public int compare(T o1, T o2) {
37    int rez = originalni.compare(o1, o2);
38    return -rez;
39   }
40  }
41
42  public static void main(String[] args) {
43   List<Zaposlenik> lista = Arrays.asList(
44     new Zaposlenik("2", "Agata", "Agić"),
45     new Zaposlenik("1", "Pero", "Perić"),
46     new Zaposlenik("3", "Ivana", "Ivić")
47   );
48
49   Set<Zaposlenik> set1 = new TreeSet<Zaposlenik>(
50    new UsporedbaPoImenu()
51   );
52   Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
53    new Reverzni<>(new UsporedbaPoImenu())
54   );
55
56   set1.addAll(lista);
57   set2.addAll(lista);
58
59   for(Zaposlenik z : set1) {
60    System.out.println(z);
61   }
62   System.out.println();
63   for(Zaposlenik z : set2) {
64    System.out.println(z);
65   }
66  }
67 }
68
```

Program u retcima 11-17 odnosno 19-25 definira dva komparatora: jedan koji radi usporedbu zaposlenika po šiframa te drugi koji ih uspoređuje po imenima. U retcima 27-40 nalazi se implementacija našeg "lažućeg" komparatora. Radi se o parametriziranom komparatoru koji deklarira jednu privatnu člansku varijablu (referencu na neki drugi komparator) i nju inicijalizira kroz konstruktor. Potom u metodi `compare` (retci 36-39) najprije pitamo originalni komparator da napravi usporedbu i zatim vratimo negirani rezultat. Uočimo ovaj komparator univerzalno je primjenjiv na bilo koji drugi komparator (neovisno o kriteriju usporedbe koji taj drugi komparator koristi); dapače, možemo ga i više puta koristiti (doduše, uz upitnu svrhu) kako je ilustrirano u nastavku.

```
1 Comparator<Zaposlenik> c = new Reverzni<>(
2   new Reverzni<>(new UsporedbaPoImenu())
3 );
```

Ovako dobiveni komparator laže o rezultatu usporedbe provedene komparatorom koji laže o rezultatu usporedbe provedene komparatorom `UsporedbaPoImenu`: posljedica je da ćemo dobiti jednako ponašanje kao da uopće nismo koristili reverzne dekoratore.



Bilješka

Razred `Reverzni` primjer je uporabe oblikovnog obrasca *Dekorator*: definiramo razred koji prema klijentu nudi jednako sučelje kao drugi razredi čije primjerke koristi za ostvarivanje svoje (modificirane) funkcionalnosti. Kako prema klijentu nudi jednako sučelje, omogućeno je rekurzivno omatanje takvih objekata.

Rezultat izvođenja programa prikazan je u nastavku.

```
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000

Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=0.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=0.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=0.000000
```

Prvi ispis je sortiran abecedno po imenu od 'A' do 'Ž', drugi reverzno.

Kako je opisana ideja izuzetno praktična i često korištena, Java u svojim standardnim bibliotekama već nudi tu funkcionalnost. U razredu `java.util.Collections` postoji statička metoda `reverseOrder(original)` koja prima referencu na komparator i vraća referencu na reverzni komparator koji je ostvaren upravno na opisani način koristeći pomoćni privatni statički razred koji je definiran unutar razreda `java.util.Collections`. Stoga zapamtite ovu ideju i koristite je u sličnim scenarijima, ali ako Vam treba reverzni komparator, sjetite se da to već imate gotovo. Umjesto da pišete Vaš vlastiti razred za laganje i potom pozivate:

```
1 new Reverzni<>(new UsporedbaPoImenu())
```

dovoljno je da to napišete ovako:

```
1 Collections.reverseOrder(new UsporedbaPoImenu())
```

a od Jave 8 ne morate više raditi niti to. Naime, kako je Java 8 omogućila uporabu pretpostavljenih metoda, jedna takva dodana je i u samo sučelje `Comparator`. Radi se o metodi `reversed()` koja je za Vas poziva `reverseOrder` razreda `Collections` nad trenutnim komparatorom: `return Collections.reverseOrder(this);`. Stoga možete pisati i ovo:

```
1 new UsporedbaPoImenu().reversed()
```

Usporedba po više kriterija

Evo još jednog poučnog primjera. Radite grafičku aplikaciju koja u tablici korisniku prikazuje sve zaposlenike. Korisnik naklikavanjem po zaglavlju tablice želi biti u mogućnosti sortirati zaposlenike. Primjerice, morate mu dopustiti da zatraži sortiranje po bilo kojem od stupaca (ime, prezime, šifra, plaća) i to izravno ili reverzno. Međutim, morate mu dopustiti i da definira da zaposlenike želi sortirati primjerice po plaći silazno, a u svakoj grupi koja ima istu plaću, po prezimenu uzlazno. Također, korisnik bi mogao poželjeti sortirati po prezimenu uzlazno, unutar istih prezimena po imenu silazno, a ako ima više ljudi s istim imenom i prezimenom, onda unutar takve grupe još po plaći silazno. U našem konkretnom slučaju, kako imamo četiri varijable, to znači da korisnik može zadati kriterij usporedbe koji u obzir uzima samo jednu, dvije, tri ili sve četiri varijable. Ukupan broj takvih mogućih kriterija je 4*2 + 4*3*2^2 + 4*3*2*2^3 + 4*3*2*1*2^4 = 8 + 48 + 192 + 384 što je 632 različita kriterija. Treba li eksplicitno napomenuti da ne želimo pisati sve te razrede?

Koje je sada rješenje? Imamo dvije mogućnosti: kompozitni komparator ili pak ulančavanje komparatora. Ideja kompozitnog komparatora je da napišemo jedan općeniti razred koji prima polje (ili listu) komparatora koje treba koristiti pri usporedbi. Usporedbu radi komparator po komparator i čim neki od komparatora utvrdi razliku, taj se rezultat vraća. Samo ako trenutni

komparator kaže da su objekti po njegovom kriteriju jednaki, prelazi se na sljedeći komparator. Takvo rješenje prikazano je u nastavku.

Primjer 14.10. Kompozitni komparator

```
 1 package book.java.collections;
 2
 3 import java.util.Arrays;
 4 import java.util.Collections;
 5 import java.util.Comparator;
 6 import java.util.List;
 7 import java.util.Set;
 8 import java.util.TreeSet;
 9
10 public class PrimjerK2 {
11
12  public static class UsporedbaPoSiframa ...
13
14  public static class UsporedbaPoImenu ...
15
16  public static class UsporedbaPoPrezimenu ...
17
18  public static class Reverzni<T> ...
19
20  public static class Kompozitni<T> implements Comparator<T> {
21   private Comparator<T>[] originalni;
22
23   @SafeVarargs
24   public Kompozitni(Comparator<T> ... originalni) {
25    super();
26    this.originalni = originalni;
27   }
28
29   @Override
30   public int compare(T o1, T o2) {
31    for(Comparator<T> c : originalni) {
32     int rez = c.compare(o1, o2);
33     if(rez != 0) return rez;
34    }
35    return 0;
36   }
37  }
38
39  public static void main(String[] args) {
40   List<Zaposlenik> lista = Arrays.asList(
41     new Zaposlenik("2", "Agata", "Agić", 10),
42     new Zaposlenik("1", "Pero", "Perić", 10),
43     new Zaposlenik("3", "Ivana", "Ivić", 10),
44     new Zaposlenik("4", "Agata", "Brajić", 10),
45     new Zaposlenik("5", "Agata", "Ada", 10),
46     new Zaposlenik("6", "Agata", "Agić", 5)
47   );
48
49   Set<Zaposlenik> set1 = new TreeSet<Zaposlenik>(
50    new Kompozitni<>(
51     new UsporedbaPoImenu(),
52     new Reverzni<>(new UsporedbaPoPrezimenu()),
53     new Reverzni<>(new UsporedbaPoSiframa())
```

```
54     )
55    );
56
57    set1.addAll(lista);
58
59    for(Zaposlenik z : set1) {
60     System.out.println(z);
61    }
62   }
63 }
64
```

Pri tome su u prikazu izostavljeni dijelovi definicija razreda koji su već dani i 14.10. Kompozitni komparator prikazan je u retcima 20 do 37. Čuva privatno polje originalnih komparatora koje puni kroz konstruktor i koje u metodi `compare` proziva redom. Primjer uporabe je prikazan u retcima 50 do 54: htjeli smo komparator koji će zaposlenike najprije usporediti po imenima; unutar istih imena po prezimenima ali reverzno, a unutar istih imena i prezimena po šiframa reverzno. Popis zaposlenika malo je proširen da se vidi korektan rad ovog komparatora. Pokretanjem se generira sljedeći ispis.

```
Zaposlenik: sifra=4, prezime=Brajić, ime=Agata, placa=10.000000
Zaposlenik: sifra=6, prezime=Agić, ime=Agata, placa=5.000000
Zaposlenik: sifra=2, prezime=Agić, ime=Agata, placa=10.000000
Zaposlenik: sifra=5, prezime=Ada, ime=Agata, placa=10.000000
Zaposlenik: sifra=3, prezime=Ivić, ime=Ivana, placa=10.000000
Zaposlenik: sifra=1, prezime=Perić, ime=Pero, placa=10.000000
```

Uvjerite se: ispis je sortiran po imenima (Agata, pa Ivana pa Pero); četiri Agate sortirane su po prezimenima padajuće (Brajić, Agić, Ada), a dvije Agate Agić sortirane su po šifri padajuće (6 pa 2).

Drugi način ostvarivanje iste funkcionalnosti je ulančavanjem komparatora: mogli bismo definirati jedan komparator koji prima reference na dva komparatora (k1 i k2). Najprije pita k1 da napravi usporedbu i ako tu postoji odluka (veće/manje), ona se vraća. Ako prvi komparator nema odluke (vraća 0), vraćamo ono što vraća drugi komparator. Višestrukom uporabom ovakvog komparatora opet možemo izgraditi komparator s proizvoljnim brojem komponenata. Evo tog rješenja.

Primjer 14.11. Ulančani komparator

```
 1 package book.java.collections;
 2
 3 import java.util.Arrays;
 4 import java.util.Comparator;
 5 import java.util.List;
 6 import java.util.Set;
 7 import java.util.TreeSet;
 8
 9 public class PrimjerK3 {
10
11  public static class UsporedbaPoSiframa ...
12
13  public static class UsporedbaPoImenu ...
14
15  public static class UsporedbaPoPrezimenu ...
16
17  public static class Reverzni<T> ...
18
19  public static class Ulancani<T> implements Comparator<T> {
```

```
20   private Comparator<T> k1;
21   private Comparator<T> k2;
22
23   public Ulancani(Comparator<T> k1, Comparator<T> k2) {
24    super();
25    this.k1 = k1;
26    this.k2 = k2;
27   }
28
29   @Override
30   public int compare(T o1, T o2) {
31    int rez = k1.compare(o1, o2);
32    if(rez != 0) return rez;
33    return k2.compare(o1, o2);
34   }
35  }
36
37  public static void main(String[] args) {
38   List<Zaposlenik> lista = Arrays.asList(
39     new Zaposlenik("2", "Agata", "Agić", 10),
40     new Zaposlenik("1", "Pero", "Perić", 10),
41     new Zaposlenik("3", "Ivana", "Ivić", 10),
42     new Zaposlenik("4", "Agata", "Brajić", 10),
43     new Zaposlenik("5", "Agata", "Ada", 10),
44     new Zaposlenik("6", "Agata", "Agić", 5)
45   );
46
47   Set<Zaposlenik> set1 = new TreeSet<Zaposlenik>(
48    new Ulancani<>(
49     new Ulancani<>(
50      new UsporedbaPoImenu(),
51      new Reverzni<>(new UsporedbaPoPrezimenu())
52     ),
53     new Reverzni<>(new UsporedbaPoSiframa())
54    )
55   );
56
57   set1.addAll(lista);
58
59   for(Zaposlenik z : set1) {
60    System.out.println(z);
61   }
62  }
63 }
64
```

Implementacija ulančanog komparatora prikazana je u retcima 19-35. Primjena je prikazana u retcima 48-54. Pokretanjem programa dobit ćemo identičan ispis kao što smo to dobili i u prethodnom primjeru s kompozitnim komparatorom.

Za razliku od kompozitnih komparatora, ulančavanje komparatora izravno je podržano kroz pretpostavljene metode sučelja `Comparator` koje posao stvaranja ovog pomoćnog razreda odrađuju za nas. Tako se prethodni poziv:

```
1 Set<Zaposlenik> set1 = new TreeSet<Zaposlenik>(
2  new Ulancani<>(
3   new Ulancani<>(
4    new UsporedbaPoImenu(),
5    new Reverzni<>(new UsporedbaPoPrezimenu())
```

```
 6    ),
 7     new Reverzni<>(new UsporedbaPoSiframa())
 8    )
 9  );
10
```

može korištenjem raspoloživih metoda napisati na sljedeći način:

```
1 Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
2  new UsporedbaPoImenu()
3    .thenComparing(new UsporedbaPoPrezimenu().reversed())
4    .thenComparing(new UsporedbaPoSiframa().reversed())
5 );
6
```

Izrada jednostavnih komparatora

Za kraj ovog poglavlja razmotrit ćemo još jedan scenarij. Vratite se na primjer 14.5 i usporedite razrede `UsporedbaPoSiframa` i `UsporedbaPoImenu`; i zamislite još da imamo i usporedbu po prezimenu i još niz drugih (ali takvih jednostavnih) članskih varijabli. Svi ti razredi su gotovo identični. Jedina je razlika u tome koju metodu pozivamo nad oba objekta da bismo došli do članske varijable nad kojom ćemo potom pozvati metodu `compareTo`. U prvom razredu radi se o `.getSifra()` a u drugom o `.getIme()`. Sve ostalo je masovna redundancija koda. Možemo li onda to nekako popraviti?

Odgovor je: naravno. Imamo li niz razreda koji su gotovo identični, samo neki dijelovi u unutrašnjosti variraju - to je recept za oblikovani obrazac *Strategija*: ono što varira modeliraj sučeljem, ukloni redundantne razrede i zamijeni ih jednim općenitijem koji je proširen referencom na objekt koji implementira konkretnu varijaciju.

Što bi to u našem slučaju značilo? Ono što varira je način na koji se dohvaća članska varijabla (koja je tipa koji implementira sučelje `Comparable`). Da bismo rješili problem, zamislimo da imamo jedan objekt kojemu kroz neku njegovu metodu pošaljemo objekt koji nas interesira a on nam vrati sadržaj članske varijable tog objekta koju je on odabrao. Tada bismo naš komparator mogli ostvariti tako da tom objektu predamo prvi objekt koji uspoređujemo i on nam vrati pripadnu varijablu, potom mu predamo drugi objekt koji trebamo usporediti i on nam opet vrati njegovu pripadnu varijablu; na kraju te dvije varijable usporedimo i vratimo rezultat.

Za modeliranje opisanog objekta ne trebamo izmišljati svoje sučelje: opisani objekt je određena implementacija unarne funkcije: primi nešto, vrati nešto izračunato na temelju onoga što je primio. Takvo funkcionalno sučelje već imamo u paketu java.util.function: radi se o parametriziranom sučelju `Function<T,R>` čiji su parametri tip argumenta i tip rezultata a metoda u pitanju je `R apply(T t);`. Oboružani time, sada možemo napisati takav općeniti komparator kako je prikazano u nastavku.

```
 1 public static class Jednostavni<T,
 2   U extends Comparable<? super U>> implements Comparator<T> {
 3  private Function<T,U> vadiKljuc;
 4
 5  public Jednostavni(Function<T, U> vadiKljuc) {
 6   super();
 7   this.vadiKljuc = vadiKljuc;
 8  }
 9
10  @Override
11  public int compare(T o1, T o2) {
12   U var1 = vadiKljuc.apply(o1);
13   U var2 = vadiKljuc.apply(o2);
14   return var1.compareTo(var2);
```

```
15   }
16 }
```

Uz takav komparator više nam uopće ne trebaju razredi `UsporedbaPoImenu`, `UsporedbaPoPrezimenu` i `UsporedbaPoSiframa`. Sada primjer:

```
1 Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
2  new UsporedbaPoImenu()
3   .thenComparing(new UsporedbaPoPrezimenu().reversed())
4   .thenComparing(new UsporedbaPoSiframa().reversed())
5 );
```

možemo dalje napisati ovako:

```
1 Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
2  new Jednostavni<Zaposlenik,String>(o -> o.getIme())
3   .thenComparing(
4    new Jednostavni<Zaposlenik,String>(
5     o -> o.getPrezime()
6    ).reversed())
7   .thenComparing(
8    new Jednostavni<Zaposlenik,String>(
9     o -> o.getSifra()
10   ).reversed())
11 );
```

a prikazane lambda izraze možemo dodatno pojednostavniti sintaksom *razred :: metoda* čime dolazimo do:

```
1 Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
2  new Jednostavni<Zaposlenik,String>(Zaposlenik::getIme)
3   .thenComparing(
4    new Jednostavni<Zaposlenik,String>(
5     Zaposlenik::getPrezime
6    ).reversed())
7   .thenComparing(
8    new Jednostavni<Zaposlenik,String>(
9     Zaposlenik::getSifra
10   ).reversed())
11 );
```

I da, pogodili ste. S obzirom da je opisani razred izuzetno koristan, upravo ovakav model dostupan je na uporabu ali ne preko razreda nego preko statičke metode `comparing` sučelja `Comparator` koja za Vas odradi posao stvaranja primjerka razreda. Metoda očekuje samo referencu na objekt koji predstavlja prethodno opisanu funkciju koja će dobiti objekt koji se uspoređuje i vratiti sadržaj varijable po kojoj se radi usporedba. Time dolazimo do konačnog oblika prethodnog koda koji biste sada trebali u cijelosti razumjeti (i pisati):

```
1 Set<Zaposlenik> set2 = new TreeSet<Zaposlenik>(
2  Comparator.comparing(Zaposlenik::getIme)
3   .thenComparing(
4    Comparator.comparing(Zaposlenik::getPrezime).reversed()
5   )
6   .thenComparing(
7    Comparator.comparing(Zaposlenik::getSifra).reversed()
8   )
9 );
```

Poglavlje 15. Obrada podataka i tokovni API

Kroz prethodna poglavlja upoznali smo se s nizom tema koje su nas pripremile za ovo poglavlje: naučili smo osnove objektno-orijentirane paradigme, naučili smo koristiti razrede i upoznali se s njihovim vrstama, upoznali smo se sa sučeljima i konačno, upoznali smo se s najčešće korištenim kolekcijama i njihovim implementacijama koje su ponuđene u okviru Javinih standardnih biblioteka.

U ovom poglavlju pažnju ćemo posvetiti obradi podataka. Podataka koje obrađujemo može biti konačno ili beskonačno (u posljednjem slučaju pretpostavka je da ih obrada ne treba sve), mogu biti pohranjeni u neku od kolekcija a i ne moraju. Jedan od razloga zašto o ovome govorimo tek sada jest činjenica da podatci najčešće jesu u kolekcijama i kolekcije će biti njihov dominantan izvor.

Za potrebe ovog poglavlja razmatrat ćemo malu tvrtku i njezine zaposlenike. Zaposlenici će biti modelirani razredom `Zaposlenik` kojemu smo dodali nešto više podataka u odnosu na prethodna poglavlja (vidi primjer 15.1); tako uz osnovne podatke pamtimo i broj djece zaposlenika zbog obračuna dječjeg doplatka, godinu rođenja zaposlenika te šifru odjela u kojem zaposlenik radi.

Primjer 15.1. Razred `Zaposlenik`

```
 1 package book.java.collections;
 2
 3 public class Zaposlenik {
 4
 5  private String sifra;
 6  private String ime;
 7  private String prezime;
 8  private double placa;
 9  private String sifraOdjela;
10  private int godinaRodenja;
11  private int brojDjece;
12
13  public Zaposlenik(String sifra, String ime, String prezime,
14    int godinaRodenja, String sifraOdjela, double placa,
15    int brojDjece) {
16   super();
17   this.sifra = sifra;
18   this.ime = ime;
19   this.prezime = prezime;
20   this.godinaRodenja = godinaRodenja;
21   this.sifraOdjela = sifraOdjela;
22   this.placa = placa;
23   this.brojDjece = brojDjece;
24  }
25
26  public String getSifra() {
27   return sifra;
28  }
29  public String getIme() {
30   return ime;
31  }
32  public String getPrezime() {
```

```
33   return prezime;
34  }
35  public double getPlaca() {
36   return placa;
37  }
38  public String getSifraOdjela() {
39   return sifraOdjela;
40  }
41  public int getGodinaRodenja() {
42   return godinaRodenja;
43  }
44  public int getBrojDjece() {
45   return brojDjece;
46  }
47
48  @Override
49  public String toString() {
50   return String.format("(%s) %s %s", sifra, ime, prezime);
51  }
52 }
53
```

Kako će nam za demonstraciju različitih primjera obrade trebati i malo programsko okruženje koje će pripremiti podatke i potom pozvati obradu, pripremljena je osnovna inačica takvog programa koja je prikazana u primjeru 15.2. U taj kod na za to predviđeno mjesto potrebno je umetnuti programske isječke koje ćemo prikazati u nastavku.

Primjer 15.2. Struktura demonstracijskog programa

```
 1 package book.java.collections;
 2
 3 import java.util.Arrays;
 4 import java.util.List;
 5
 6 public class PrimjeriObrade {
 7
 8  public static void main(String[] args) {
 9   List<Zaposlenik> popis = dohvatiPopis();
10
11   // Ovdje slijede primjeri...
12   // ...
13
14  }
15
16  public static List<Zaposlenik> dohvatiPopis() {
17   return Arrays.asList(
18    new Zaposlenik("1", "Pero", "Perić", 1980, "1", 5800, 2),
19    new Zaposlenik("2", "Ivana", "Anić", 1991, "1", 4900, 1),
20    new Zaposlenik("3", "Jakov", "Jak", 1975, "2", 4900, 0),
21    new Zaposlenik("4", "Karla", "Ivić", 1985, "1", 5800, 2),
22    new Zaposlenik("5", "Ivan", "Zak", 1988, "2", 5000, 0),
23    new Zaposlenik("6", "Jasna", "Pek", 1986, "2", 5000, 1),
24    new Zaposlenik("7", "Ante", "Car", 1988, "1", 5300, 1),
25    new Zaposlenik("8", "Petra", "Pik", 1986, "1", 5300, 0)
26   );
27  }
28 }
29
```

Kroz prethodnih nekoliko poglavlja veliku smo pažnju posvetili vrstama kolekcija: kakvu uslugu od njih želimo, koje su podatkovne strukture prikadne za ostvarivanje te usluge i uz koju računsku složenost, na koji način možemo ažurirati i ispitivati takvu kolekciju i slično.

Kada govorimo o obradi podataka, naglasak je potpuno drugačiji: želimo primjerice iz kolekcije zaposlenika na zaslon ispisati tablicu koja prikazuje ime, prezime i plaću zaposlenika; možda želimo izračunati prosječnu plaću svih zaposlenika; možda želimo za zaposlenike koji su u odjelu čija je šifra "2" odrediti i ispisati prosječan broj djece; možda želimo na temelju kolekcije zaposlenika generirati kolekciju prezimena koja je sortirana po prezimenu; možda želimo izgraditi mapu čiji su ključevi šifre odjela a vrijednosti kolekcije zaposlenika koji rade u tom odjelu. Ovakvih primjera je beskonačno, a svima im je zajedničko da na kolekciju koja čuva osnovni popis podataka gledaju kao na običan izvor podataka koji se potom obrađuju; interna implementacija kolekcije nije bitna.

Razmotrimo jedan konkretan primjer. Na temelju kolekcije zaposlenika želimo generirati sortiranu listu zaposlenika koji su rođeni nakon 1980. godine i koji rade u odjelu čija je šifra "2".

```
 1 List<Zaposlenik> radeUOdjelu2 = new ArrayList<>();
 2 for(Zaposlenik z : popis) {
 3  if(z.getSifraOdjela().equals("2")) {
 4   radeUOdjelu2.add(z);
 5  }
 6 }
 7
 8 List<Zaposlenik> rodeniNakon = new ArrayList<>();
 9 for(Zaposlenik z : radeUOdjelu2) {
10  if(z.getGodinaRodenja() > 1980) {
11   rodeniNakon.add(z);
12  }
13 }
14
15 List<String> prezimena = new ArrayList<>();
16 for(Zaposlenik z : rodeniNakon) {
17  prezimena.add(z.getPrezime());
18 }
19
20 List<String> sortiranaPrezimena = new ArrayList<>(prezimena);
21 sortiranaPrezimena.sort(null);
22
23 for(String prezime : sortiranaPrezimena) {
24  System.out.println(prezime);
25 }
```

Razmotrimo malo strukturu rješenja. U retku 1 stvaramo pomoćnu kolekciju (listu) koja bi trebala sadržavati samo one zaposlenike koji su zaposleni u odjelu čija je šifra jednaka "2". U retcima 2-6 nalazi se kod obavlja takvo *filtriranje*: on prolazi po izvornoj listi svih zaposlenika, svakog zaposlenika testira kako bi utvrdio radi li u odjelu "2" i ako je uvjet testiranja ispunjen, zaposlenik se dodaje u listu `radeUOdjelu2`.

Jednom kad imamo filtriranu listu po odjelu, želimo izgraditi novu listu koja će od zaposlenika koji su ušli u listu `radeUOdjelu2` sadržavati samo one koji su rođeni nakon 1980. godine. Stoga u retku 8 najprije definiramo novu praznu listu `rodeniNakon` te u retcima 9-13 prolazimo kroz sve elemente liste `radeUOdjelu2`, svakog zaposlenika testiramo i one zaposlenike za koje utvrdimo da su rođeni nakon 1980. godine dodajemo u listu `radeUOdjelu2`.

Po završetku izvođenja ova dva koraka izgradili smo kolekciju zaposlenika koja od svih zaposlenika tvrtke sadrži samo one zaposlenike koji su zaposleni u odjelu čija je šifra "2" i koji su rođeni nakon 1980. godine.

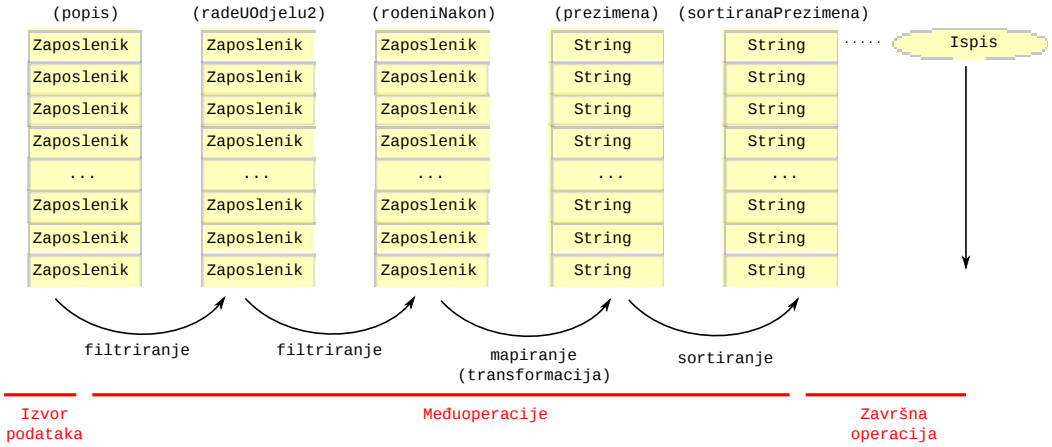
Sljedeći korak je na temelju kolekcije tih zaposlenika izgraditi kolekciju njihovih prezimena. Ovime mijenjamo tip elemenata kolekcije: kolekcija koju ćemo tako dobiti više neće biti kolekcija objekata tipa `Zaposlenik` već će biti kolekcija objekata tipa `String`. To je prikazano u retcima 15-18. U retku 15 stvaramo novu praznu listu stringova `prezimena` a u retcima 16-18 obavljamo tranformaciju (kažemo još: *mapiranje*) elemenata liste `radeUOdjelu2`.

Kako je zadatak ispisati prezimena sortirano, u retcima 20-21 rješavamo taj dio zadatka. Stvaramo novu kolekciju prezimena `sortiranaPrezimena` (redak 20) koju punimo prethodno utvrđenim popisom prezimena i koju potom (redak 21) sortiramo.

Konačno, u retcima 23-25 prolazimo kroz svako prezime kolekcije `sortiranaPrezimena` i nad njime pozivamo željenu *akciju* što u našem slučaju predstavlja ispis na zaslon.

Jednostavan dijagram koji prikazuje korake u rješavanju ovog zadatka prikazan je na slici 15.1.

Slika 15.1. Dijagram obrade podataka



Što je s kvalitetom prikazanog rješenja? Idemo najprije riješiti ono trivijalno: imamo dupliciranje koda. Pogledajte retke 2-6 te 9-13: to je gotovo identičan kod; jedina je razlika u primijenjenom uvjetu ispitivanja. Na bolje rješenje odvest će nas već uobičajeni oblikovni obrazac *Strategija*: ono što varira treba modelirati sučeljem. A što kod nas varira? Radi se o kodu koji dobije neki objekt, treba provjeriti zadovoljava li taj objekt uvjet prihvaćanja i vratiti `true` ili `false`. Takvo sučelje u Javi već imamo. U paketu java.util.function nalazi se parametrizirano sučelje `Predicate<T>`:

```
1 @FunctionalInterface
2 public interface Predicate<T> {
3     boolean test(T t);
4
5     // te pretpostavljene metode "and", "negate", "or", ...
6
7 }
```

Sučelje deklarira jednu apstraktnu metodu: `test` koja prima testirani objekt i mora vratiti `true` ili `false`. Uz pretpostavku da ćemo testove pisati uporabom tog sučelja, u kod bismo mogli dodati jednu statičku metodu koja će obavljati filtriranje kolekcije proizvoljne kolekcije. Kod ćemo dopuniti još jednom metodom koja će primiti ulaznu listu i željeni ispitni predikat (odnosno filter), stvorit će izlaznu listu i posao njenog punjenja prepustiti prvoj općenitijoj metodi. Cjelovito rješenje u kojem su filtri ostvareni uporabom lambda-izraza prikazano je u nastavku (primjer 15.3).

Primjer 15.3. Poopćena usluga filtriranja

```
1 package book.java.collections;
```

```
 2
 3 import java.util.ArrayList;
 4 import java.util.Arrays;
 5 import java.util.Collection;
 6 import java.util.List;
 7 import java.util.function.Predicate;
 8
 9 public class PrimjeriObrade {
10
11  public static void main(String[] args) {
12   List<Zaposlenik> popis = dohvatiPopis();
13
14   List<Zaposlenik> radeUOdjelu2 = filtrirajListu(
15    popis,
16    z -> z.getSifraOdjela().equals("2"));
17
18   List<Zaposlenik> rodeniNakon = filtrirajListu(
19    radeUOdjelu2,
20    z -> z.getGodinaRodenja() > 1980);
21
22   // ostatak...
23  }
24
25  public static <T> void filtriraj(Collection<T> ulazna,
26   Collection<T> izlazna, Predicate<T> uvjet) {
27
28   for(T element : ulazna) {
29    if(uvjet.test(element)) {
30     izlazna.add(element);
31    }
32   }
33  }
34
35  public static <T> List<T> filtrirajListu(
36   Collection<T> ulazna,
37   Predicate<T> uvjet) {
38
39   List<T> izlazna = new ArrayList<>();
40   filtriraj(ulazna, izlazna, uvjet);
41   return izlazna;
42  }
43
44  // ...
45 }
46
```

Spomenuta općenita funkcija filtriranja prikazana je u retcima 25-33. Prima tri argumenta: ulaznu kolekciju (najopćenitije vrste), izlaznu kolekciju u koju treba upisati rezultat filtriranja (također najopćenitije vrste) te referencu na ispitni predikat. Metoda potom iterira po ulaznoj kolekciji element po element; za svaki element pita ispitni predikat je li element prihvatljiv i ako ispitni predikat kaže da je, dodaje ga u izlaznu kolekciju.

Pomoćna metoda prikazana je u retcima 35-42: ona prima samo dva argumenta: ulaznu kolekciju te ispitni predikat. Metoda sama odlučuje da će se kao izlazna kolekcija koristiti lista pa je stvara i potom poziva prethodnu metodu kako bi se proveo postupak filtriranja. Nakon toga, metoda vraća referencu na popunjenu izlaznu listu.

Pojednostavljenje koje se time dobije vidljivo je u retcima 14 i 18: svaku od pomoćnih kolekcija stvaramo jednim pozivom metode koja prima ulaznu kolekciju i ispitni predikat koji smo ostvarili

uporabom lambda-izraza (sjetite se - to je samo pokrata za istovjetni kod koji stvara primjerak anonimnog razreda i koji bi nam potrošio nekoliko redaka).

Ovime smo malo popravili prva dva koraka prikazane obrade (filtriranja). Vratimo se sada na izvorni kod gdje smo prvi puta prikazali cjelovitu obradu i pogledajmo retke 15-18. Za očekivati je da će nam i takva funkcionalnost trebati pri različitim obradama i kod ne želimo pisati više puta. Proanalizirajmo stoga taj kod: što će se razlikovati od jednog do drugog konkretnog slučaja primjene? Svaki puta ćemo prolaziti kroz petlju i na temelju trenutnog objekta dohvatiti ili stvoriti neki novi objekt koji ćemo dodati u kolekciju koju stvaramo. Opet ćemo primijeniti oblikovni obrazac *Strategiju* i uočiti nam treba sučelje koje modelira operaciju y = f(x). I takvo sučelje postoji u paketu java.util.function pod nazivom `Function<T,R>` gdje je prvi parametar tip argumenta a drugi parametar tip rezultata. Sučelje je prikazano u nastavku.

```
1 @FunctionalInterface
2 public interface Function<T, R> {
3     R apply(T t);
4
5     // plus pretpostavljene metode...
6 }
```

Konkretna funkcija koja će nama trebati primat će zaposlenika a vraćat njegovo prezime. Možemo je modelirati lambda-izrazom `z -> z.getPrezime()` ili kraće `Zaposlenik::getPrezime`. Napisat ćemo općenitu metodu koja obavlja mapiranje a prima referencu na ulaznu kolekciju tipa `T`, na izlaznu kolekciju tipa `R` te na funkciju mapiranja koja objekte tipa `T` prevodi u objekte tipa `R`. Potom ćemo nadodati pomoćnu metodu koja će za nas stvoriti listu za izlaznu kolekciju i nju napuniti pozivanjem prethodne općenitije metode. Dvije potrebne metode prikazane su u nastavku.

```
1 public static <T,R> void mapiraj(Collection<T> ulazna,
2   Collection<R> izlazna, Function<T,R> maper) {
3
4  for(T element : ulazna) {
5   izlazna.add(maper.apply(element));
6  }
7 }
8
9 public static <T,R> List<R> mapirajListu(Collection<T> ulazna,
10   Function<T,R> maper) {
11
12  List<R> izlazna = new ArrayList<>();
13  for(T element : ulazna) {
14   izlazna.add(maper.apply(element));
15  }
16  return izlazna;
17 }
```

a njihovom uporabom dio izvornog koda koji je radio taj posao:

```
1 List<String> prezimena = new ArrayList<>();
2 for(Zaposlenik z : rodeniNakon) {
3  prezimena.add(z.getPrezime());
4 }
```

možemo nadomjetiti jednim pozivom metode kako je prikazano u nastavku.

```
1 List<String> prezimena = mapirajListu(
2   rodeniNakon,
3   z -> z.getPrezime());
```

Mogli bismo ostvariti i pomoćnu metodu koja za nas obavlja sortiranje. Njezin izvorni kod prikazan je u nastavku.

```
1 public static <T> List<T> sortirajListu(Collection<T> ulazna,
2   Comparator<T> komparator) {
3
4  List<T> izlazna = new ArrayList<>(ulazna);
5  izlazna.sort(komparator);
6  return izlazna;
7 }
```

čime bi isječak koda u demonstracijskom programu:

```
1 List<String> sortiranaPrezimena = new ArrayList<>(prezimena);
2 sortiranaPrezimena.sort(null);
```

mogli zamijeniti jednom naredbom:

```
1 List<String> sortiranaPrezimena = sortirajListu(
2   prezimena,
3   null);
```

Na kraju, još samo jedan dio koda izvornog demonstracijskog programa nismo dotakli - onaj koji obavlja knačnu akciju nad svakim elementom pripremljene kolekcije. Da ne bismo svaki puta morali kopirati kod koji obavlja obilazak, u skladu s oblikovnim obrascem *Strategija* uočimo da samu akciju možemo modelirati odgovarajućim sučeljem i potom napisati općenitu metodu koja prima kolekciju i referencu na akciju te implementira posao obilaska i pozivanja akcije nad svakim od elemenata. Kako modelirati općenitu akciju? Treba nam sučelje koje definira metodu koja prima objekt nad kojim se obavlja akcija i ništa ne vraća. Takvo sučelje na raspolaganju nam je u paketu java.util.Function pod imenom `Consumer<T>` i prikazano je u nastavku.

```
1 @FunctionalInterface
2 public interface Consumer<T> {
3
4     void accept(T t);
5
6     // plus pretpostavljene metode...
7 }
```

Uporabom ovog sučelja općenita metoda koja nam treba prikazana je u nastavku.

```
1 public static <T> void obaviAkciju(Collection<T> ulazna,
2   Consumer<T> akcija) {
3
4  for(T element : ulazna) {
5   akcija.accept(element);
6  }
7 }
```

Njezinom uporabom posljednji dio originalnog programa koji je obavljao ispis možemo zamijeniti s:

```
1 obaviAkciju(sortiranaPrezimena, System.out::println);
```

Akcija je ovdje zadana kao lambda-izraz `p -> System.out.println(p)` što se može kraće zapisati kao `System.out::println`.

Kako smo kroz prethodni tekst radili više izmjena, evo još jednom konačnog oblika programa (primjer 15.4). Lijepa stvar je da smo razvili nekoliko općenitih metoda koje možemo premjetiti u izolirani razred i potom ih koristiti u svim programima i na svim mjestima gdje nam to zatreba. Kod koji ćemo pri tome pisati na mjestu uporabe bit će jednostavan, koncizan i jasan.

Primjer 15.4. Poboljšana inačica programa

```
1 package book.java.collections;
```

```
 2
 3 import java.util.ArrayList;
 4 import java.util.Arrays;
 5 import java.util.Collection;
 6 import java.util.Comparator;
 7 import java.util.List;
 8 import java.util.function.Consumer;
 9 import java.util.function.Function;
10 import java.util.function.Predicate;
11
12 public class PrimjeriObrade {
13
14  public static void main(String[] args) {
15   List<Zaposlenik> popis = dohvatiPopis();
16
17   // Ovdje slijede primjeri...
18   // ...
19
20   List<Zaposlenik> radeUOdjelu2 = filtrirajListu(
21    popis,
22    z -> z.getSifraOdjela().equals("2"));
23
24   List<Zaposlenik> rodeniNakon = filtrirajListu(
25    radeUOdjelu2,
26    z -> z.getGodinaRodenja() > 1980);
27
28   List<String> prezimena = mapirajListu(
29    rodeniNakon,
30    z -> z.getPrezime());
31
32   List<String> sortiranaPrezimena = sortirajListu(
33    prezimena,
34    null);
35
36   obaviAkciju(sortiranaPrezimena, System.out::println);
37
38  }
39
40  public static <T> void filtriraj(Collection<T> ulazna,
41   Collection<T> izlazna, Predicate<T> uvjet) {
42
43   for(T element : ulazna) {
44    if(uvjet.test(element)) {
45     izlazna.add(element);
46    }
47   }
48  }
49
50  public static <T> List<T> filtrirajListu(Collection<T> ulazna,
51   Predicate<T> uvjet) {
52
53   List<T> izlazna = new ArrayList<>();
54   filtriraj(ulazna, izlazna, uvjet);
55   return izlazna;
56  }
57
58  public static <T,R> void mapiraj(Collection<T> ulazna,
59   Collection<R> izlazna, Function<T,R> maper) {
```

```
60
61    for(T element : ulazna) {
62     izlazna.add(maper.apply(element));
63    }
64   }
65
66   public static <T,R> List<R> mapirajListu(
67    Collection<T> ulazna, Function<T,R> maper) {
68
69    List<R> izlazna = new ArrayList<>();
70    for(T element : ulazna) {
71     izlazna.add(maper.apply(element));
72    }
73    return izlazna;
74   }
75
76   public static <T> List<T> sortirajListu(
77    Collection<T> ulazna, Comparator<T> komparator) {
78
79    List<T> izlazna = new ArrayList<>(ulazna);
80    izlazna.sort(komparator);
81    return izlazna;
82   }
83
84   public static <T> void obaviAkciju(Collection<T> ulazna,
85    Consumer<T> akcija) {
86
87    for(T element : ulazna) {
88     akcija.accept(element);
89    }
90   }
91
92   public static List<Zaposlenik> dohvatiPopis() {
93    return Arrays.asList(
94     new Zaposlenik("1", "Pero", "Perić", 1980, "1", 5800, 2),
95     new Zaposlenik("2", "Ivana", "Anić", 1991, "1", 4900, 1),
96     new Zaposlenik("3", "Jakov", "Jak", 1975, "2", 4900, 0),
97     new Zaposlenik("4", "Karla", "Ivić", 1985, "1", 5800, 2),
98     new Zaposlenik("5", "Ivan", "Zak", 1988, "2", 5000, 0),
99     new Zaposlenik("6", "Jasna", "Pek", 1986, "2", 5000, 1),
100    new Zaposlenik("7", "Ante", "Car", 1988, "1", 5300, 1),
101    new Zaposlenik("8", "Petra", "Pik", 1986, "1", 5300, 0)
102   );
103  }
104 }
105
```

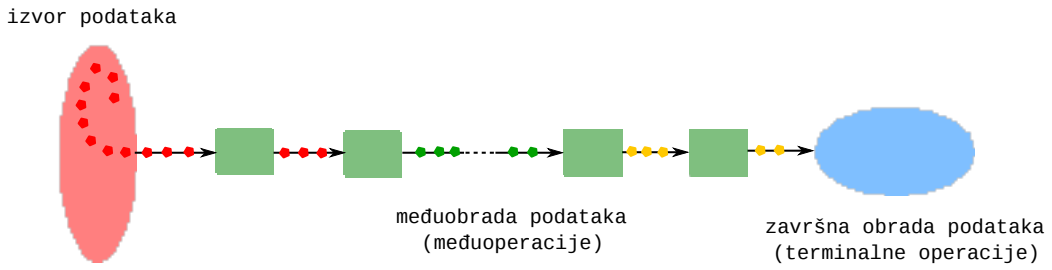
Sjećate li se da smo na početku ovog poglavlja utvrdili da s našim rješenjem postoje problemi i da ćemo krenuti na rješavanje trivijalnih? Dobra vijest je da smo taj dio odradili. Sada nam je ostalo riješiti one prave probleme: prikazano rješenje generira veliko memorijsko zauzeće stvarajući niz međukolekcija koje nam - zapravo - ne trebaju. Dodatno, prikazano rješenje ne može baš na trivijalan način koristiti dobrobiti višeprocesorskih računalnih sustava koji su danas sasvim uobičajeni; za to bismo trebali pisati još dosta koda.

Počev od Jave 8, u okviru standardnih biblioteka dostupan nam je API koji je spomenute teške probleme već riješio za nas: radi se o paketu java.util.stream koji je namijenjen ostvarivanju učinkovite obrade podataka nadogradnjom prethodno opisanih ideja koje smo do sada ilustrirali u ovom poglavlju.

Obrada modelom toka

Da bi se omogućilo pisanje koda koji na učinkovit način može obavljati slijednu ili paralelnu obradu podataka, Java 8 dolazi s paketom java.util.stream i novom apstrakcijom obrade koja se temelji na tokovima podataka koji su ostvareni cjevovodima (bez panike - razmišljajte o cjevovodu samo na konceptualnoj razini). Sličica prikazana u nastavku to ilustrira. Sljedeća ilustracija to lijepo prikazuje.

Slika 15.2. Model obrade podataka



Model konkretne obrade podataka odnosno (odnosno cjevovod) sastavlja se od tri vrste objekata:

1. jednog izvora podataka (engl. *source*) što je na slici prikazano kao crveni oval lijevo,
2. nula, jedne ili više međuoperacije (engl. *intermediate operations*) koje su na slici prikazane kao zeleni kvadratići te
3. jedne komponente za završnu obradu podataka (engl. *terminalne operacije*) koja je na slici prikazana kao plavi oval desno.

Podatci koji se obrađuju putuju iz izvora kroz međuoperacije (koje ih mogu filtrirati pa dalje propustiti samo neke od podataka ili ih mogu transformirati u neki drugi tip ili ih mogu sortirati itd) sve do završne obrade. Završna obrada prekida cjevovod.

Izvori

S obzirom da podatci koje je potrebno obraditi mogu biti objekti ili pak primitivne vrijednosti, definirano je nekoliko sučelja koja modeliraju različite vrste izvora:

- `java.util.stream.Stream`,
- `java.util.stream.IntStream`,
- `java.util.stream.LongStream` te
- `java.util.stream.DoubleStream`.

Izvori su jednokratni objekti koji isporučuju podatke koji se obrađuju. Jednom kada izvor isporuči sve podatke, ne može se ponovno koristiti za sljedeću obradu; ako je to potrebno, moramo stvoriti novi izvor podataka.

Izvor *objekata* modeliran je sučeljem `java.util.stream.Stream`. Izvor možemo izgraditi na različite načine. Izgradnja uporabom statičkih metoda ovog sučelja prikazana je u nastavku na nekoliko primjera.

```
1 // Izvor koji sadrži tri stringa:
```

```
2 Stream<String> s1 = Stream.of("Zagreb", "Osijek", "Križevci");
3
```

Statička metoda `Stream.of(...)` prima varijabilni broj elemenata i vraća izvor koji će na zahtjev isporučiti upravo te elemente. Prazan izvor možemo izgraditi statičkom metodom `Stream.empty(...)` kako je prikazano u sljedećem primjeru:

```
1 // Prazan izvor:
2 Stream<String> s2 = Stream.empty();
```

Ako nam svi podatci potrebni za definiranje izvora nisu odjednom dostupni, izgradnju izvora možemo razdijeli u dva koraka. U prvom koraku stvaramo pomoćni objekt koji služi za postupnu izgradnju izvora i modeliran je sučeljem `Stream.Builder<T>`. Sučelje `Stream` nudi statičku metodu koja vraća jednu implementaciju takvog objekta. Elemente koje izvor mora isporučiti potom možemo dodavati ili metodom `accept(T element)` koja ništa ne vraća, ili metodom `add(T element)` koja vraća referencu na taj isti pomoćni objekt pa pozive možemo lijepo ulančati kako je prikazano u nastavku. Jednom kad smo gotovi s izgradnjom, izvor dobivamo pozivom metode `build()`. Evo primjera.

```
1 // Izgradnja izvora korak po korak:
2 Stream.Builder<String> b3 = Stream.builder();
3
4 b3.add("Zagreb").add("Osijek");
5 b3.add("Križevci");
6
7 Stream<String> s3 = b3.build();
```

Imamo li sve elemente koje izvor treba isporučiti u polju, razredu `java.util.Arrays` dodana je prikladna statička metoda koja prima polje i vraća izvor koji vraća te elemente:

```
1 // Izgradnja uporabom polja elemenata:
2 Stream<String> s4 = Arrays.stream(new String[] {
3   "Zagreb", "Osijek", "Križevci"
4 });
```

Moguće je definirati i izvore koji sadrže beskonačno mnogo elemenata. Za dva tipična načina već je u Javinim bibliotekama pripremljena potrebna infrastruktura. Prvi od načina je uporabom metode `Stream.iterate(T prvi, UnaryOperator operator)` pri čemu je funkcionalno sučelje `java.util.function.UnaryOperator` specijalizacija općenite funkcije na funkciju koja prima jedan argument i vraća rezultat koji je istog tipa kao i argument. Na ovaj način uporabom metode `iterate` stvara se izvor čiji su elementi redom: *prvi*, *operator.apply(prvi)*, *operator.apply(operator.apply(prvi))*, itd. Drugim riječima, operator se iterativno primjenjuje na vrijednost koju je vratio u prethodnom pozivu. Primjerice, želimo li izvor objekata tipa `Integer` koji sadrži sve parne brojeve počev od 0, možemo pisati:

```
1 // Izgradnja izvora s beskonačno mnogo elemenata (1):
2 Stream<Integer> s5 = Stream.iterate(0, v -> v+2);
```

pri čemu smo kao operator definirali lambda izraz koji primljeni broj uveća za 2. Na taj način dobit ćemo elemente 0, 0+2=2, 2+2=4, 4+2=6, 6+2=8, ... Drugi način izgradnje izvora s beskonačno mnogo elemenata jest oslanjanjem na objekt za "proizvodnju" objekata koji je modeliran sučeljem `java.util.function.Supplier<T>`. To sučelje traži postojanje metode `T get()` koja na svaki poziv "proizvede" i vrati jedan element. Evo primjera koji na ovaj način definira izvor koji sadrži parne brojeve počevši od 0.

```
1 // Izgradnja izvora s beskonačno mnogo elemenata (2):
2 Stream<Integer> s6 = Stream.generate(
3  new Supplier<Integer>() {
4   private Integer current = 0;
```

```
 5   public Integer get() {
 6    Integer element = current;
 7    current += 2;
 8    return element;
 9   }
10  }
11 );
```

Osnovno sučelje jednostavnih kolekcija `java.util.Collection` također je s Javom 8 dopunjeno tako da deklarira metode `stream()` i `parallelStream()` za stvaranje izvora na temelju elemenata kolekcije. Posljedica je da bilo koju kolekciju možemo koristiti kao izvor podataka za obradu. Evo primjera.

```
1 // Izgradnja izvora iz kolekcije
2 Set<String> gradovi = new HashSet<>();
3 gradovi.add("Zagreb");
4 gradovi.add("Osijek");
5 gradovi.add("Vinkovci");
6
7 Stream<String> s7 = gradovi.stream();
8 Stream<String> s8 = gradovi.parallelStream();
```

Od izvora primitivnih tipova podataka, na raspolaganju nam stoje samo implementacije za tipove `int`, `long` i `double` i ti su izvori modelirani sučeljima `IntStream`, `LongStream` i `DoubleStream`. Kako se radi o izvorima puno specifičnije semantike, ta sučelja nude i prikladne metode koje su smislene pri radu s brojevima. Evo nekoliko primjera.

```
1 IntStream s9 = IntStream.of(1, 1, 2, 3, 5, 8);
2 IntStream s10 = IntStream.range(0, 10);
3 IntStream s11 = IntStream.rangeClosed(0, 10);
```

U prvom retku stvaramo izvor primitivnih cijelih brojeva koji su dani kao argument. U drugom retku stvaramo izvor koji sadrži cijele brojeve od 0 do 9 (gornja granica u metodi `range` nije uključiva) dok u trećem retku koristimo metodu `rangeClosed` koja stvara izvor cijelih brojeva od 0 do 10 jer su obje granice uključene. Sučelje, dakako, podržava i uporabu inačica metoda `iterate(prvi, IntUnaryOperator operator)`, `generate(IntSupplier s)` te postupnu izgradnju dohvaćanjem objekta za izgradnju metodom `IntStream.builder()`. Slične metode podržane su i u preostala dva sučelja.

Prije no što nastavimo analizirati ostale komponente ovakvog modela obrade podataka, evo jednog primjera.

```
1 Stream.of("Zagreb", "Osijek", "Križevci")
2  .filter(grad -> grad.length() <= 6)
3  .sorted()
4  .forEach(System.out::println);
```

Primjer opisuje obradu koja se sastoji od izvora čiji su podatci tri stringa: "Zagreb", "Osijek" i "Križevci" (redak 1: stvaranje izvora), potom dvije međuobrade podataka (redak 2: filtriranje i redak 3: sortiranje) te završne obrade (redak 4: ispis na zaslon). Pokretanjem tog isječka dobit ćemo sljedeći ispis.

```
Osijek
Zagreb
```

Uočite kako je čitav postupak obrade elegantno zapisan kao jedna (ulančana) naredba.

Izvor podataka može biti namijenjen za slijednu ili za paralelnu obradu. U primjeru kod kolekcija pokazali smo da nad kolekcijom možemo zatražiti `.stream()` ili pak `.parallelStream()`. Jednom kada imamo izvor podataka, u onaj drugi ga možemo pretvoriti jednim pozivom meto-

de. Naime, sučelje `Stream` definira dvije metode koje su namijenjene upravo tome: `parallel()` te `sequential()`. Nad paralelnim izvorom obrada podataka obavljat će se višedretveno bez ikakve dodatne intervencije od korisnika.

Međuobrada podataka

Operacije koje obavljaju međuobradu podataka i same se ponašaju kao novi izvor podataka. Ovu vrstu operacija dijelimo na operacije bez stanja te operacije sa stanjem. Operacije bez stanja izlazni podatak generiraju samo na temelju trenutnog ulaznog podatka; takva je, primjerice, operacija filtriranja. Operacije sa stanjem (ovisno o vrsti) mogu zatražiti dohvat svih elemenata iz ulaznog niza prije no što krenu generirati izlazne elemente. Primjer takve operacije je sortiranje gdje nije moguće na izlaz poslati niti jedan podatak dok svi podatci s ulaza nisu viđeni.

Za razliku od klasične obrade podataka kod koje korisnik eksplicitno iterira po kolekciji podataka koje obrađuje, model koji ovdje opisujemo koristi takozvano *unutarnje iteriranje*: implementacije samih operacija biraju kako će i kada dohvaćati podatke iz izvora nad kojim su pozvane. Posljedica je da se sama operacija brine o tome hoće li obradu raditi slijedno ili paralelno te ako će raditi paralelno, kako će to postići; ti su detalji skriveni od programera.

Same operacije u pravilu su izvedene na način da ne grade međukolekcije podataka (sjetite se - mi to jesmo radili u motivacijskom primjeru i to je bio jedan od problema) i podatke uopće ne obrađuju osim ako baš ne moraju. Ovo ćemo ilustrirati sljedećim primjerom. Operacija `filter` prima referencu na ispitni predikat (baš kao u našem motivacijskom primjeru i uz jednako sučelje) te se ponaša kao novi izvor koji sadrži samo one elemente koji zadovolje ispitni predikat. Što mislite, što će ispisati sljedeći programski isječak?

```
1 Stream.of("Zagreb", "Osijek", "Križevci")
2  .filter(grad -> {
3   System.out.println("Ispitujem grad "+grad);
4   return grad.length() <= 6;
5  });
```

Uočite što smo napravili: u ispitni predikat smo dodali još jedan redak koji će ispisati na zaslon da je taj kod pozvan i uz koji grad. Pokretanjem se na zaslon neće ispisati ništa! Naime, definirali smo izvor; definirali smo međuoperaciju koja djeluje nad tim izvorom; i tu smo stali. Tu međuoperaciju nitko nikada nije pitao niti za jedan podatak pa ona nikada nije krenula s izračunom. Tek kada dodamo završnu obradu podataka pokrenut će se lanac obrade: završna obrada će svoj izvor pitati za podatak; taj će svoj izvor pitati za podatak; i tako sve do originalnog izvora koji će isporučiti prvi podatak. Zahtjev za obradom, dakle, kreće od kraja lanca i ide prema početku lanca; podatci tada kreću od početka lanca prema kraju lanca.

Operacija kojom se radi mapiranje objekata je `.map(Function mapper)` koja kao argument očekuje referencu na funkciju. Operacija nad objektima svojeg izvora podataka poziva funkciju te se sama ponaša kao novi izvor u kojem se nalaze rezultati koje su vratili ti pozivi funkcije. Pokušajte sada proanalizirati što će ispisati sljedeći program.

```
1 Stream.of("1000", "50", "504", "125", "17", "217", "603")
2  .filter(s -> {
3   System.out.println("Prvi filter: "+s);
4   return s.length()<4; }
5  )
6  .map(s -> {
7   System.out.println("Mapiranje: "+s);
8   return Integer.valueOf(s); }
9  )
10 .filter(b -> {
11  System.out.println("Drugi filter: "+b);
12  return b < 200; }
13 )
```

```
14   .forEach(System.out::println);
```

Prvi filter prima stringove (pozvan je nad izvorom stringova) i dalje propušta samo one stringove čija je duljina manja od 4. Funkcija mapiranja na ulazu dobiva stringove koje je filter propustio dalje i konvertira ih u objekte tipa `Integer`; za svoje korisnike, ta se funkcija dalje ponaša kao izvor objekata tipa `Integer`. Drugi filter na ulaz dobiva objekte tipa `Integer` i dalje propušta samo one koji su manji od 200. Konačno, završna obrada na ulaz dobiva također objekte tipa `Integer` i ispisuje ih na zaslon. Pokretanjem ovog programa dobiva se sljedeći ispis.

```
Prvi filter: 1000
Prvi filter: 50
Mapiranje: 50
Drugi filter: 50
50
Prvi filter: 504
Mapiranje: 504
Drugi filter: 504
Prvi filter: 125
Mapiranje: 125
Drugi filter: 125
125
Prvi filter: 17
Mapiranje: 17
Drugi filter: 17
17
Prvi filter: 217
Mapiranje: 217
Drugi filter: 217
Prvi filter: 603
Mapiranje: 603
Drugi filter: 603
```

Koje je objašnjenje za ovakav ispis? Završna operacija od svojeg izvora (što je filter 2) zatražila je jedan podatak. Zbog toga je filter 2 od svojeg izvora (a to je metoda mapiranja) zatražio jedan podatak. Zbog toga je metoda mapiranja od svojeg izvora (a to je filter 1) zatražila jedan podatak. A zbog toga je filter 1 od svojeg izvora zatražio jedan podatak i dobiva string "1000". Filter 1 sada poziva ispitni predikat s kojim smo ga parametrizirali da vidi smije li to isporučiti dalje. Pokretanjem ispitnog predikata na zaslonu generiramo ispis "Prvi filter: 1000" a sam predikat vraća `false` pozivatelju. Zbog toga filter 1 od svojeg izvora traži sljedeći podatak i dobiva "50". Poziva ispitni predikat s tim objektom što rezultira ispisom "Prvi filter: 50" a predikat vraća `true`. Prvi filter stoga taj objekt isporučuje dalje svojem pozivatelju a to je bila metoda mapiranja. Metoda mapiranja dobiva, dakle, string "50" i poziva funkciju mapiranja s kojom smo je parametrizirali. To rezultira ispisom "Mapiranje: 50" i konverzijom tog stringa u objekt tipa `Integer`. Taj objekt metoda mapiranja vraća svojem pozivatelju, a to je filter 2. Filter 2 nad tim objektom poziva ispitni predikat s kojim smo njega parametrizirali i tako nastaje ispis "Drugi filter: 50" a sam predikat vraća `true`. Zbog toga filter 2 taj broj predaje svojem pozivatelju a to je metoda `forEach` koja taj objekt predaje objektu s kojim smo je parametrizirali (lambda-izrazom tipa `Consumer`) i to rezultira ispisom "50". Sada završna operacija traži sljedeći podatak, i čitava priča se ponavlja.

Koliko je trivijalno paralelizirati obradu ilustrira sljedeći isječak. Dodali smo samo poziv `.parallel()` nad izvorom i proširili pomoćne ispise tako da ispisuju naziv dretve koja izvodi taj dio koda.

```
1 Stream.of("1000", "50", "504", "125", "17", "217", "603")
2  .parallel()
3  .filter(s -> {
4   System.out.println(
5    "Prvi filter: "+s+" "+Thread.currentThread().getName());
```

```
 6    return s.length()<4; }
 7   )
 8   .map(s -> {
 9    System.out.println(
10     "Mapiranje: "+s+" "+Thread.currentThread().getName());
11    return Integer.valueOf(s); }
12   )
13   .filter(b -> {
14    System.out.println(
15     "Drugi filter: "+b+" "+Thread.currentThread().getName());
16    return b < 200; }
17   )
18   .forEach(b -> {
19    System.out.println(b+" "+Thread.currentThread().getName());
20   });
```

što svakim pokretanjem daje malo drugačiji ispis; primjer je prikazan u nastavku.

```
Prvi filter: 17 main
Prvi filter: 603 ForkJoinPool.commonPool-worker-2
Prvi filter: 504 ForkJoinPool.commonPool-worker-5
Mapiranje: 504 ForkJoinPool.commonPool-worker-5
Prvi filter: 217 ForkJoinPool.commonPool-worker-4
Mapiranje: 217 ForkJoinPool.commonPool-worker-4
Drugi filter: 504 ForkJoinPool.commonPool-worker-5
Prvi filter: 1000 ForkJoinPool.commonPool-worker-3
Prvi filter: 125 ForkJoinPool.commonPool-worker-5
Mapiranje: 125 ForkJoinPool.commonPool-worker-5
Drugi filter: 217 ForkJoinPool.commonPool-worker-4
Prvi filter: 50 ForkJoinPool.commonPool-worker-1
Mapiranje: 50 ForkJoinPool.commonPool-worker-1
Drugi filter: 50 ForkJoinPool.commonPool-worker-1
50 ForkJoinPool.commonPool-worker-1
Mapiranje: 17 main
Mapiranje: 603 ForkJoinPool.commonPool-worker-2
Drugi filter: 17 main
Drugi filter: 125 ForkJoinPool.commonPool-worker-5
17 main
Drugi filter: 603 ForkJoinPool.commonPool-worker-2
125 ForkJoinPool.commonPool-worker-5
```

U ispisu možete primjetiti da se pojavljuje nekoliko različitih dretvi koje paralelno obavljaju zadani posao. Pri tome, kako nismo eksplicitno tražili da se rezultat generira onim redosljedom kojim su elementi u izvoru, u ovom smo konkretnom pokretanju dobili smo ispis 50, 17, 125.

Što se događa ako u obradu ubacimo operaciju sortiranja? Evo primjera. Želimo da je ispisani popis brojeva sortiran.

```
 1 Stream.of("1000", "50", "504", "125", "17", "217", "603")
 2  .filter(s -> {
 3   System.out.println("Prvi filter: "+s);
 4   return s.length()<4; }
 5  )
 6  .map(s -> {
 7   System.out.println("Mapiranje: "+s);
 8   return Integer.valueOf(s); }
 9  )
10  .sorted((n1,n2) -> {
11   System.out.format("Komparator: (%d,%d)%n", n1, n2);
```

```
12    return n1.compareTo(n2);
13   })
14   .filter(b -> {
15    System.out.println("Drugi filter: "+b);
16    return b < 200; }
17   )
18   .forEachOrdered(System.out::println);
19
```

Operaciju sortiranja umetnuli smo nakon operacije mapiranja i kao argument predali referencu na komparator (koji usput na zaslon ispisuje svaki poziv). Ispis je sljedeći.

```
Prvi filter: 1000
Prvi filter: 50
Mapiranje: 50
Prvi filter: 504
Mapiranje: 504
Prvi filter: 125
Mapiranje: 125
Prvi filter: 17
Mapiranje: 17
Prvi filter: 217
Mapiranje: 217
Prvi filter: 603
Mapiranje: 603
Komparator: (504,50)
Komparator: (125,504)
Komparator: (125,504)
Komparator: (125,50)
Komparator: (17,125)
Komparator: (17,50)
Komparator: (217,125)
Komparator: (217,504)
Komparator: (603,125)
Komparator: (603,504)
Drugi filter: 17
17
Drugi filter: 50
50
Drugi filter: 125
125
Drugi filter: 217
Drugi filter: 504
Drugi filter: 603
```

Što se dogodilo? Izvođenje je krenulo od završne obrade preko drugog filtera na operaciju sortiranja - i tu se događa nešto interesantno. Da bi operacija sortiranja mogla generirati sortirane elemente, ona ih najprije mora sve dohvatiti. Stoga operacija sortiranja od svojeg izvora zahtjeva podatke tako dugo dok ih ima; i to se u ispisu lijepo vidi kroz niz ispisa "Prvi filter"+"Mapiranje". Tek kada je operacija sortiranja dohvatila sve podatke, kreće ih sortirati što je vidljivo kroz niz poziva komparatora. Jednom kad je postupak sortiranja gotov, operacija otpušta prvi element koji odlazi u drugi filter i konačno na završnu obradu; potom završna obrada traži sljedeći element od filtera 2 koji to traži od operacije sortiranja; ona pak ima elemente spremne i sada otpušta sljedeći element (50), itd.

Ovaj odjeljak zaključit ćemo sažetim pregledom često korištenih međuoperacija.

- `.filter(Predicate p)`: elemente izvora filtrira predanim predikatom i dalje prosljeđuje samo one koji zadovoljavaju predikat.

- `.map(Function f)`: elemente izvora transformira predanom funkcijom i dalje prosljeđuje objekte koje vrati funkcija.
- `.sort()`: dalje prosljeđuje elemente sortiranim redosljedom u skladu s *prirodnim poretkom elemenata*.
- `.sort(Comparator c)`: dalje prosljeđuje elemente sortiranim redosljedom u skladu s predanim komparatorom.
- `.limit(n)`: dalje prosljeđuje najviše *n* elemenata iz svojeg izvora.
- `.mapToInt(ToIntFunction f)`: elemente izvora transformira u primitivne cijele brojeve tipa `int` i vraća odgovarajući tip izvora (`IntStream`). Transformaciju obavlja uporabom predane funkcije `f`.
- `.mapToLong(ToLongFunction f)`: elemente izvora transformira u primitivne cijele brojeve tipa `long` i vraća odgovarajući tip izvora (`LongStream`). Transformaciju obavlja uporabom predane funkcije `f`.
- `.mapToDouble(ToDoubleFunction f)`: elemente izvora transformira u primitivne decimalne brojeve tipa `double` i vraća odgovarajući tip izvora (`DoubleStream`). Transformaciju obavlja uporabom predane funkcije `f`.

Završna obrada podataka

Završnu obradu podataka obavljaju terminalne operacije. Terminalne operacije djeluju nad izvorom ali se same više ne ponašaju kao izvor. Umjesto toga, ove operacije ili ne vraćaju ništa (poput `.forEach(Consumer c)`), ili vraćaju jednostavan podatak (neki od elemenata toka, neki broj i slično), ili vraćaju složene tipove podataka (poput novih kolekcija).

Završne operacije koje na temelju podataka toka računaju i vraćaju rezultat nazivaju se *redukcije*. Redukcije mogu vraćati podatak jednostavnog tipa (poput srednje vrijednosti, minimalne vrijednosti, maksimalne vrijednosti, sume, broja elemenata i slično) ili mogu vraćati rezultate složenijeg tipa (poput kolekcija).

Za potrebe primjera koji slijede koristit ćemo razred `Zaposlenik` (primjer 15.1) te osnovni demonstracijski program dan u primjeru 15.2.

Postojeće jednostavne redukcije

Izvori primitivnih tipova korisnicima nude niz gotovih metoda za provedbu redukcije. Primjerice, sučelje `DoubleStream` nudi sljedeće metode ovog tipa.

- `double sum();` - računa i vraća sumu svih elemenata izvora.
- `long count();` - računa i vraća broj elemenata u izvoru.
- `OptionalDouble average();` - računa i vraća prosjek svih elemenata izvora (obratite pažnju na tip rezultata).
- `OptionalDouble min();` - računa i vraća minimalni elemenata izvora (obratite pažnju na tip rezultata).
- `OptionalDouble max();` - računa i vraća maksimalni elemenata izvora (obratite pažnju na tip rezultata).

Slično nude i sučelja `IntStream` i `DoubleStream`. Općenito sučelje `Stream` koje modelira izvor objekata proizvoljnog tipa nešto je siromašnije ovakvim metodama; nudi `count` te metode `min` i `max` pri čemu ove posljednje dvije kao argument očekuju komparator kojim će se elementi uspoređivati. Međutim, ovo sučelje nudi niz drugih korisnih redukcija poput ispitivanja postoji li

ikoji element koji zadovoljava ispitni predikat (metoda `anyMatch`), zadovoljavaju li svi elementi ispitni predikat (metoda `allMatch`) i slično.

Recimo da nas zanima ukupan trošak plaće za sve zaposlenike naše tvrtke. Kako bismo ga izračunali?

```
1 Double trosakPlace = popis.stream()
2   .mapToDouble(Zaposlenik::getPlaca)
3   .sum();
```

Nad popisom zaposlenika (sjetite se, to je bila kolekcija `List<Zaposlenik>`) izgradimo izvor na koji zakvačimo operaciju mapiranja koja zaposlenika preslika u njegovu plaću (čime dobivamo `DoubleStream` jer smo kao metodu mapiranja koristili `mapToDouble` a ne općenitu `map`) te redukciju koja računa i vraća sumu. Uz podatke koje imamo, rezultat će biti 42000 kuna.

Općenita redukcija

Kao podršku za pisanje redukcija koje nisu direktno implementirane u standardnim bibliotekama na raspolaganju nam stoje tri inačice metode `reduce` koje su deklarirane u sučelju `Stream`.

Najjednostavnija inačica ove metode tok objekata tipa `T` reducira u objekt tipa `T` uporabom binarnog operatora opisanog sučeljem `BinaryOperator` (to je funkcija dvaju argumenata pri čemu su i argumenti i rezultat istog tipa). Prototip ove metode je:

```
1 Optional<T> reduce(BinaryOperator<T> accumulator);
```

Predani binarni operator mora biti asocijativan, ne-interferirajući (ako je izvor kolekcija, operator ne smije modificirati tu kolekciju tijekom izvođenja) i bez stanja. Uz to ograničenje, metoda se ponaša na sljedeći način. Ako je tok prazan, metoda vraća `Optional.empty()`. Ako tok ima samo jedan element, metoda vraća taj element. Ako tok ima više elemenata (označimo ih s e1, e2, e3, e4, ..., en), konceptualni izračun rezultata prikazan je u nastavku.

partial = e1;
partial = accumulator.apply(partial, e2);
partial = accumulator.apply(partial, e3);
partial = accumulator.apply(partial, e4);
...
partial = accumulator.apply(partial, en);
result = partial

Kako bismo uporabom ove metode napisali izračun ukupnog troška plaće? Evo rješenja.

```
1 Optional<Double> trosakPlace = popis.stream()
2   .map(Zaposlenik::getPlaca)
3   .reduce((suma,placa) -> suma+placa);
```

ili još kraće:

```
1 Optional<Double> trosakPlace = popis.stream()
2   .map(Zaposlenik::getPlaca)
3   .reduce(Double::sum);
```

Pokretanjem ćemo dobiti rezultat od 42000 kuna. Detaljni trag izvođenja možemo prikazati ako dodamo pomoćne ispise, kako je napravljeno u nastavku.

```
1 Optional<Double> trosakPlace = popis.stream()
2   .map(z -> {
3    System.out.println("Plaća: " + z.getPlaca());
4    return z.getPlaca();
5   })
```

```
 6    .reduce((suma,placa) -> {
 7     double tmp = suma+placa;
 8     System.out.format("%f+%f=%f%n", suma, placa, tmp);
 9     return tmp;
10    });
```

Dobit ćemo sljedeći ispis iz kojega je jasno vidljivo kako se ukupna suma malo po malo akumulira do ukupnog iznosa.

```
Plaća: 5800.0
Plaća: 4900.0
5800.000000+4900.000000=10700.000000
Plaća: 4900.0
10700.000000+4900.000000=15600.000000
Plaća: 5800.0
15600.000000+5800.000000=21400.000000
Plaća: 5000.0
21400.000000+5000.000000=26400.000000
Plaća: 5000.0
26400.000000+5000.000000=31400.000000
Plaća: 5300.0
31400.000000+5300.000000=36700.000000
Plaća: 5300.0
36700.000000+5300.000000=42000.000000
Optional[42000.0]
```

Ograničenje prethodne najjednostavnije metode redukcije je što su tipovi elemenata ulaznog toka i generiranog rezultata fiksirani na isto. Inačica redukcije koja ne vraća rezultat tipa `Optional<T>` omogućena je metodom čiji je prototip sljedeći.

```
1 T reduce(T identity, BinaryOperator<T> accumulator);
```

Prvi argument predstavlja neutralni element za predani operator, odnosno element za koji vrijedi da je `accumulator.apply(identity,t)` opet jednako `t`. Pod tim uvjetima, ako je tok prazan, ova metoda vraća predani neutralni element a u suprotnom vraća rezultat izračunat na jednak način kao i prethodna inačica metode. Kako je za zbrajanje 0 neutralni element, izračun ukupnog troška plaće ovom ćemo metodom izračunati na sljedeći način.

```
1 Double trosakPlace = popis.stream()
2   .map(Zaposlenik::getPlaca)
3   .reduce(0.0, Double::sum);
```

Konačno, najopćenitiji oblik redukcije je onaj koji dozvoljava da se tip računatog podatka i tip podataka toka razlikuju. To je podržano sljedećom inačicom metode `reduce` čiji je prototip prikazan u nastavku.

```
1 <U> U reduce(U identity,
2               BiFunction<U, ? super T, U> accumulator,
3               BinaryOperator<U> combiner);
```

Metoda prima tri argumenta: neutralni element, akumulatorsku funkciju koja je sada modelirana kao `BiFunction` i koja prima kao prvi argument parcijalni rezultat (tipa U) i sljedeći element toka (tipa T) i generira novi parcijalni rezultat (tipa U) te kombinator (uskoro ćemo ga pojasniti). Prije toga samo mala digresija. Zamislimo da moramo napraviti redukciju toka čiji su elementi e1, e2, e3, e4, e5 i e6 (oni su svi tipa T). Pretpostavimo da imate višejezgreno računalo. Zadaću redukcije mogli bismo razložiti na redukciju prva tri elementa čime dobivamo parcijalni rezultat u1 (tipa U) i na redukciju druga tri elementa čime dobivamo parcijalni rezultat u2 (tipa U). Prednost ovakve podjele je što te podzadatke možemo računati paralelno (a podjele ne moraju biti samo dvije već ih se može generirati i više, ovisno o ukupnom broju elemenata te raspoloživim sklopovskim resursima). Međutim, jednom kada su izračuni parcijalnih redukcija

gotovi, završili smo s objektima tipa U koje sada nekako treba spojiti u jedan konačan rezultat tipa U. Uočite da akumulatorska funkcija to ne može: ona očekuje argument tipa U i drugi tipa T. Upravo to je razlog što ova posljednja inačica metode `reduce` prima i treći argument: kombinator koji dobiva dva parcijalna rezultata i mora ih spojiti u zajednički rezultat. Kako on prima dva argumenta tipa U i vraća rezultat tipa U, modeliran je sučeljem `BinaryOperator`.

Kao dokaz koncepta, kako bismo ovom metodom napisali kod koji će izgenerirati jedan string u kojem su sve plaće razdvojene razmacima? Evo mogućeg rješenja. Neutralni element je prazan string; akumulatorska funkcija prima string i `Double` i vraća novi string u kojem je broj pretvoren u string i nadodan na kraj primljenog stringa; kombinator je binrni operator koji prima dva stringa i vraća njihovu konkatenaciju. Rješenje je prikazano u nastavku.

```
1 String ispis = popis.stream()
2  .map(Zaposlenik::getPlaca)
3  .reduce(
4   "",
5   (str,placa)-> str + placa + " ",
6   (str1,str2) -> str1+str2
7  );
8
9 System.out.println(ispis);
```

Kombinator smo kraće mogli zapisati kao `String::concat`. Program će ispisati:

```
5800.0 4900.0 4900.0 5800.0 5000.0 5000.0 5300.0 5300.0
```

Prilikom pisanja ovih funkcija treba se držati sljedećih ograničenja. Akumulatorska funkcija i kombinator moraju biti asocijativni. Predani neutralni element mora biti neutralni element za kombinator. Kombinator i akumulatorska funkcija moraju biti upareni, što znači da `kombinator.apply(u, akumulator.apply(identity,t))` mora davati isti rezultat koji daje `akumulator.apply(u,t)`.

Izračun ukupnog troška plaće ovom ćemo metodom izračunati na sljedeći način.

```
1 Double trosakPlace = popis.stream()
2  .map(Zaposlenik::getPlaca)
3  .reduce(0.0, Double::sum, Double::sum);
```

Kako iz decimalnih brojeva računamo opet decimalni broj, i akumulator i kombinator su jednake funkcije.

Kolektori

Osnovni mehanizam reduktora koji smo upravo obradili u nekim situacijama može onemogućiti jednostavno pisanje efikasnog koda. Razmotrimo to na dva primjera. Prvi primjer na temelju niza imena (elementi izvora su stringovi) gradi jedan string u kojem su sva imena spojena zajedno i razdvojena razmakom. Slično smo maloprije vidjeli s gradovima. Evo koda.

```
1 List<String> imena = Arrays.asList(
2     "Ivo", "Ana", "Jasmin", "Đurđa", "Pero", "Štefanija"
3 );
4
5 String spojenaImena = imena.stream()
6  .reduce(
7   "",
8   (spojeno,ime) -> spojeno+ime+" ",
9   (spoj1,spoj2) -> spoj1+spoj2
10  );
```

Kao drugi primjer prikažimo kako iz popisa zaposlenika izgraditi listu njihovih plaća. Evo koda.

```
List<Double> place = popis.stream()
 .map(Zaposlenik::getPlaca)
 .reduce(
  (List<Double>)new ArrayList<Double>(),
  (lista, placa)->{
   List<Double> nova = new ArrayList<>();
   nova.addAll(lista);
   nova.add(placa);
   return nova;
  },
  (lista1,lista2) -> {
   List<Double> nova = new ArrayList<>();
   nova.addAll(lista1);
   nova.addAll(lista2);
   return nova;
  }
 );
```

Vidite li zašto su oba rješenja neefikasna? Kako bi kod bio višedretveno siguran, akumulatorska funkcija na temelju primljenih argumenata *mora vratiti novi objekt* koji predstavlja rezultat. Posljedica je da primjer koji stvara string u kojem su sva imena spojena i razdvojena razmacima gradi izuzetno puno međurezultata. Za naš konkretan primjer, kod se izvodi kao da smo napisali:

```
String rez = "";
rez = rez + "Ivo" + " ";
rez = rez + "Ana" + " ";
rez = rez + "Jasmin" + " ";
rez = rez + "Đurđa" + " ";
rez = rez + "Pero" + " ";
rez = rez + "Štefanija" + " ";
```

Negdje na samom početku ove knjige objasnili smo zašto je to problematično (sa stajališta efikasnosti izvođenja).

U drugom primjeru gdje gradimo listu plaća, krećemo s praznom listom pa potom stvaramo novu listu u koju dodajemo prethodnu (praznu) i prvu plaću. U sljedećem koraku stvaramo novu listu u koju nadodajemo prethodnu listu (koja sadrži prvu plaću) i potom drugu plaću. U sljedećem koraku potom stvaramo novu listu u koju nadodajemo prethodnu listu (koja sadrži prve dvije plaće) i potom treću plaću. I tako redom. Broj lista koje stvaramo linearan je s brojem plaća koje trebamo nadodati i u svakom sljedećem koraku u novostvorenu praznu listu nadodajemo sve više i više elemenata, čime dobivamo kvadratnu složenost izvođenja.

Za ovakvu vrstu zadataka prikladno je rješenje uporaba promjenjivih (engl. *mutable*) akumulatora. Za spajanje stringova takav je akumulator razred `StringBuilder`. Kada bismo se ograničili na slijednu implementaciju redukcije, jedan primjerak takvog akumulatora bio bi nam sasvim dovoljan. Međutim, čitav API koji ovdje analiziramo zamišljen je tako da se može izvoditi bilo slijedno, bilo paralelno. Primjerice, pri paralelnom izvođenju redukcije implementacija bi mogla posao spajanja šest imena razbiti u dva podzadatka koji će se izvesti paralelno (spojiti prva tri imena u jedan akumulator te druga tri imena u drugi akumulator) i potom ta dva akumulatora iskombinirati u konačni rezultat. Alternativno, implementacija bi mogla odlučiti iskoristiti tri dretve: u jedan akumulator spojiti prva dva imena, u drugi sljedeća dva imena, i konačno u treći sljedeća dva imena; potom bi iskombinirala te akumulatore (recimo prvi i drugi u novi, pa taj s trećim u konačni).

Kako će posao točno biti izveden implementacijski je detalj i mi ne bismo htjeli o tome razmišljati. Međutim, ako implementacija sama može odlučivati kako će razlomiti posao, onda joj ne možemo kao kod prethodno obrađene metode `reduce` proslijediti jedan primjerak akumulatora (što je tamo bio onaj neutralni element). Umjesto toga, metodi moramo uz akumulatorsku

funkciju i kombinator proslijediti i referencu na objekt-tvornicu novih akumulatora. Taj objekt mora implementirati sučelje `Supplier<R>` (s kojim smo se već upoznali) te na svaki poziv metode `.get()` vratiti novi primjerak praznog promjenjivog akumulatora. Uporabom tog objekta implementacija redukcije može stvoriti po želji potreban broj akumulatora, svaki od njih iskoristiti za izgradnju djelomičnog rezultata redukcije (pozivima akumulatorske funkcije) i potom te akumulatore spojiti u konačan rezultat redukcije pozivima kombinatora.

Za izvođenje ovakve vrste redukcija više ne koristimo inačice metode `reduce(...)` već jednu od dvije inačice metode `collect(...)` koje su također definirane u sučelju `Stream`. Prva od tih metoda direktno kao argumente prima spomenuta tri objekta i njezin prototip prikazan je u nastavku.

```
1 <R> R collect(Supplier<R> supplier,
2             BiConsumer<R, ? super T> accumulator,
3             BiConsumer<R, R> combiner);
```

Obratite pažnju na tipove (parametrizaciju). Metodu pozivamo nad izvorom čiji su elementi tipa T. Objekt `supplier` je tvornica promjenjivih akumulatora tipa R. Akumulatorska funkcija modelirana je sučeljem `BiConsumer<R, ? super T>` koje definira metodu `accept` s dva argumenta (prvi tipa R, drugi tipa T) i koja ništa ne vraća; njezin je zadatak (u ovom slučaju) u akumulator primljen kao prvi argument ugraditi element toka primljen kao drugi argument; još jednom skrećem pažnju da metoda ništa ne vraća: njezina zadaća nije stvaranje novog akumulatora (kojeg bi onda nekako trebala vratiti) već modifikacija primljenog. Konačno, kombinator je modeliran istim sučeljem uz parametre `<R,R>`: njegova je zadaća u akumulator primljen kao prvi argument ugraditi podatke iz akumulatora primljenog kao drugi argument (metoda time modificira akumulator primljen kao prvi argument i ništa ne vraća).

S obzirom da je dosta važno shvatiti što se tu događa, prikazat ćemo oba primjera najprije uporabom anonimnih razreda (tako de se točno istaknu korištena sučelja i metode) a potom ćemo to pokratiti uporabom lambda-izraza i zatim to još malo pokratiti uporabom kraćih lambda-izraza (takozvanih referenci na metode).

Prvi primjer riješen uporabom anonimnih razreda prikazan je u nastavku.

```
1 String spojenaImena = imena.stream()
2  .collect(
3   new Supplier<StringBuilder>() {
4    @Override
5    public StringBuilder get() {
6     return new StringBuilder();
7    }
8   },
9   new BiConsumer<StringBuilder, String>() {
10   @Override
11   public void accept(StringBuilder aku, String ime) {
12    aku.append(ime);
13   }
14  },
15  new BiConsumer<StringBuilder, StringBuilder>() {
16   public void accept(StringBuilder aku1, StringBuilder aku2) {
17    aku1.append(aku2);
18   };
19  }
20 )
21  .toString();
```

Metoda `collect` pri tome kao konačan rezultat vraća završni akumulator koji je po tipu `StringBuilder`. Stoga je u nastavku ulančan poziv `.toString()` koji vraća konačni string.

Drugi primjer riješen uporabom anonimnih razreda prikazan je u nastavku.

```
List<Double> listaPlaca = popis.stream()
 .map(Zaposlenik::getPlaca)
 .collect(
  new Supplier<List<Double>>() {
   @Override
   public List<Double> get() {
    return new ArrayList<>();
   }
  },
  new BiConsumer<List<Double>, Double>() {
   @Override
   public void accept(List<Double> aku, Double placa) {
    aku.add(placa);
   }
  },
  new BiConsumer<List<Double>, List<Double>>() {
   @Override
   public void accept(List<Double> aku1, List<Double> aku2) {
    aku1.addAll(aku2);
   }
  }
 );
```

Prvi primjer riješen uporabom lambda-izraza prikazan je u nastavku.

```
String spojenaImena = imena.stream()
 .collect(
  () -> new StringBuilder(),
  (aku,ime) -> aku.append(ime),
  (aku1,aku2) -> aku1.append(aku2)
 )
 .toString();
```

Uočite kako je lambdom zamijenjena metoda-tvornica (metoda `get` sučelja `Supplier`): to je lambda koja ne prima argumente i vraća rezultat operatora `new`.

Drugi primjer riješen uporabom lambda-izraza prikazan je u nastavku.

```
List<Double> listaPlaca = popis.stream()
 .map(Zaposlenik::getPlaca)
 .collect(
  () -> new ArrayList<>(),
  (aku,placa) -> aku.add(placa),
  (aku1,aku2) -> aku1.addAll(aku2)
 );
```

Prvi primjer riješen uporabom kraćih lambda-izraza (referenci na metode) prikazan je u nastavku.

```
String spojenaImena = imena.stream()
 .collect(
  StringBuilder::new,
  StringBuilder::append,
  StringBuilder::append
 )
 .toString();
```

Uočite kako smo lambdu `() -> new StringBuilder()` zamijenili kraćom inačicom koja koristi referencu na konstruktor: `StringBuilder::new`.

Drugi primjer riješen uporabom kraćih lambda-izraza (referenci na metode) prikazan je u nastavku.

```
List<Double> listaPlaca4 = popis.stream()
 .map(Zaposlenik::getPlaca)
 .collect(
  ArrayList::new,
  List::add,
  List::addAll
 );
```

Treba napomenuti da smo ovdje za potrebe ilustracije ovih pokrata malo odstupili od početnog primjera kod spajanja imena: izostavili smo separator pa će imena završiti doslovno zaljepljena jedno za drugo. Želimo li nadodati i separator, uporabom razreda `StringBuilder` morat ćemo stati na malo složenijem lambda-izrazu koji ne možemo pokratiti u poziv jedne metode:

```
String spojenaImena = imena.stream()
 .collect(
  () -> new StringBuilder(),
  (aku,ime) -> aku.append(ime).append(" "),
  (aku1,aku2) -> aku1.append(aku2)
 )
 .toString();
```

što će skoro pa dobro odraditi posao (ostat će nam završni razmak nakon zadnjeg imena, ali to je problem svih prethodnih primjera).

Za jednostavno rješavanje problema spajanja stringova u Javu 8 u paket java.util dodan je još jedan pomoćni razred: `StringJoiner`. Primjerak tog razreda može se konfigurirati tako da se zada koji separator treba koristiti (a moguće je čak definirati i globalni prefiks i sufiks koji treba dodati). Pri tome će sam razred voditi računa o tome da delimiter ubacuje samo između elemenata a definirani prefiks i sufiks (ako su definirani) doda na početak i kraj. Evo ilustracija uporabe.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
sj.add("Ivana").add("Pero");
sj.add("Jasna").add("Krešo");
String spojeni = sj.toString();

System.out.println(spojeni);
```

Ovaj isječak rezultirat će ispisom:

```
[Ivana, Pero, Jasna, Krešo]
```

jer smo kao delimiter definirali string zarez-razmak, kao prefiks uglatu otvorenu zagradu a kao sufiks uglatu zatvorenu zagradu.

Objektu tipa `StringJoiner` moguće je nadodati elemente drugog objekta tipa `StringJoiner` pozivom metode `merge` (čime se modificira prvi objekt); pri tome se preuzimaju samo elementi koji su bili dodani a ne i eventualno konfigurirani prefiks i sufiks. Razred nudi dva konstruktora: jedan koji prima samo željeni delimiter te drugi koji prima delimiter, prefiks i sufiks. Uz ovaj razred, spajanje imena možemo riješiti vrlo elegantno, što ilustrira isječak koda u nastavku.

```
String spojenaImena5 = imena.stream()
 .collect(
  () -> new StringJoiner(", "),
  StringJoiner::add,
  StringJoiner::merge
 )
```

```
  7   .toString();
```

Prethodno opisani mehanizam redukcije uporabom prikazane inačice metode `collect` izuzetno je ekspresivan. No postoje još neka ograničenja koja u određenim situacijama mogu otežati pisanje kod. Vratimo se opet na spajanje stringova: prikazani kolektor ne može vratiti string - on vraća `StringJoiner` (ili `StringBuilder` ili što smo već koristili kao akumulator) a mi sami moramo ulančati završni poziv kojim ćemo iz akumulatora dobiti konačni podatak.

Drugi primjer koji ilustrira ograničenje pristupa je izračun srednje vrijednosti brojeva iz toka. Kako biste to napravili? U najboljem slučaju mogli biste napisati kolektor koji zbraja ukupnu sumu brojeva i njihov broj i taj par informacija vraća kao rezultat; za izračun prosjeka sami trebate napisati još nekoliko linija koda koje će iskoristiti prikupljene sumarne podatke i vratiti prosjek. Evo koda koji to ilustrira.

```
 1 double[] sumaBroj = Stream.of(1.0, 5.0, 8.0, 4.0, 12.0, 6.0)
 2  .collect(
 3   () -> new double[] {0.0, 0.0},
 4   (stat, broj) -> {
 5    stat[0] += broj;
 6    stat[1]++;
 7   },
 8   (stat1, stat2) -> {
 9    stat1[0] += stat2[0];
10    stat1[1] += stat2[1];
11   }
12  );
13
14 OptionalDouble prosjek = sumaBroj[1] == 0 ?
15   OptionalDouble.empty() :
16   OptionalDouble.of(sumaBroj[0]/sumaBroj[1]);
17
18 System.out.println("Prosjek je: " + prosjek);
```

Kao akumulatore koristimo polja decimalnih brojeva od dva elemenata: prvi element čuva parcijalnu sumu a drugi element broj brojeva koji su sumirani. Implementacija akumulatorske funkcije tada akumulatoru na nultu lokaciju pribroji broj (poveća sumu) te na prvu lokaciju doda 1 (poveća broj brojeva). Kombinator je trivijalan: prvi akumulator u sebe nadoda podatke drugog akumulatora. Za izračun prosjeka nakon što je završena operacija redukcije potrebno je pogledati stanje vraćenog akumulatora i podijeliti sumu s brojem elemenata (ili signalizirati da je tok bio prazan i da prosjek ne postoji). Ovo je svakako nešto što ne bismo htjeli raditi svaki puta. Bilo bi odlično kada bi se u okviru redukcije nekako mogao odraditi i ovaj posljednji korak.

Kako bi se podržalo pisanje takvih reduktora, definirano je sučelje `java.util.stream.Collector<T,A,R>`. Sučelje predstavlja opis kolektora koji radi redukciju elemenata tipa T koristeći akumulatore tipa A i na kraju iz završnog akumulatora proizvodi rezultat tipa R. Sučelje je prikazano u nastavku.

```
1 public interface Collector<T, A, R> {
2     Supplier<A> supplier();
3     BiConsumer<A, T> accumulator();
4     BinaryOperator<A> combiner();
5     Function<A, R> finisher();
6     Set<Characteristics> characteristics();
7
8     // plus dvije statičke metode (za sada preskačemo)
9 }
```

Sučelje definira po jednu metodu za dohvat objekata koje smo inače slali kao argumente metodi `collect`, te definira dvije dodatne metode. Uočite da je u sučelju promijenjen tip kombi-

natora iz `BiConsumer` u `BinaryOperator` čime je omogućeno da kombinator ili modificira jedan od akumulatora sadržajem drugog ili stvori novi akumulator koji će napuniti sadržajem oba; što god da napravio, preko povratne vrijednosti vraća referencu na akumulator koji čuva rezultat kombiniranja. Od spomenutih novih metoda, prva je `finisher()` koja vraća referencu na funkciju koja prima završni akumulator (stvoren kao rezultat kompletne redukcije) i koja ne temelju njega stvara konačni rezultat koji vraća kolektor. Druga metoda je `characteristics()` čija je zadaća vratiti skup svojstava koja opisuju kolektor i na temelju kojih će podsustav koji će izvoditi ovu operaciju (bilo slijedno, bilo paraleno) doznati uz koja ograničenja mora raditi. Svojstva su definirane ugniježđenom enumeracijom `Collector.Characteristics` koja definira tri vrijednosti pojašnjene u nastavku.

- `IDENTITY_FINISH`: ako kolektor deklarira ovo svojstvo, tada se nad njim na kraju ne treba pozvati metoda `finisher()` već se kao rezultat može vratiti direktno zadnji stvoreni akumulator, što može povećati učinkovitost izvođenja. Posljedica je da se tip A mora moći ukalupiti u tip R.

- `UNORDERED`: ako kolektor deklarira ovo svojstvo, tada ne garantira da će redoslijed obrade elemenata izvora (a time i generiranje rezultata) odgovarati redoslijedu kojim su elementi dani u izvoru (ako izvor uopće definira da redoslijed postoji).

- `CONCURRENT`: ako kolektor deklarira ovo svojstvo, tada poručuje da je implementacija akumulatora koju vraća metoda-tvornica višedretveno sigurna, što znači da čak i u slučaju paralelne provedbe redukcije, nema potrebe stvarati više akumulatora pa ih potom kombinirati već je dozvoljeno stvoriti samo jedan akumulator i njega koristiti paralelno.

Oboružani ovim znanjem, spremni smo za pisanje našeg prvog kolektora koji će biti modeliran sučeljem `Collector`. Prikazat ćemo kolektor koji spaja stringove i koji ovaj puta, bez ikakvog dodatnog truda od korisnika, vraća string. Evo koda.

```
 1 public class SpojiStringove
 2    implements Collector<String, StringJoiner, String> {
 3
 4  private String delimiter;
 5
 6  public SpojiStringove(String delimiter) {
 7   super();
 8   this.delimiter = delimiter;
 9  }
10
11  @Override
12  public Supplier<StringJoiner> supplier() {
13   return () -> new StringJoiner(delimiter);
14  }
15
16  @Override
17  public BiConsumer<StringJoiner, String> accumulator() {
18   return StringJoiner::add;
19  }
20
21  @Override
22  public BinaryOperator<StringJoiner> combiner() {
23   return StringJoiner::merge;
24  }
25
26  @Override
27  public Function<StringJoiner, String> finisher() {
28   return StringJoiner::toString;
29  }
30
```

```
31  @Override
32  public Set<Collector.Characteristics> characteristics() {
33   return Collections.emptySet();
34  }
35 };
```

Za uporabu kolektora koji su opisani na ovaj način na raspolaganju nam stoji druga inačica metode `collect` koja prima samo jedan argument: referncu na primjerak kolektora. Uporabu ilustrira sljedeći isječak koda.

```
1 String spojenaImena = imena.stream()
2   .collect(new SpojiStringove(", "));
3
4 System.out.println(spojenaImena);
```

Uočite da pojedine metode u napisanom razredu `SpojiStringove` vraćaju ono što smo inače pisali kao argumente: i dalje nam trebaju primjerci anonimnih razreda ili lambde.

Sljedeći primjer ilustrira kolektor koji računa srednju vrijednost i koji doista vraća `OptionalDouble` a ne djelomično rješenje koje još treba doraditi.

```
1 public class SrednjaVrijednost
2    implements Collector<Double, double[], OptionalDouble> {
3
4  @Override
5  public Supplier<double[]> supplier() {
6   return () -> new double[] {0.0, 0.0};
7  }
8
9  @Override
10 public BiConsumer<double[], Double> accumulator() {
11  return (stat, broj) -> {
12    stat[0] += broj;
13    stat[1]++;
14  };
15 }
16
17 @Override
18 public BinaryOperator<double[]> combiner() {
19  return (stat1, stat2) -> {
20    stat1[0] += stat2[0];
21    stat1[1] += stat2[1];
22    return stat1;
23  };
24 }
25
26 @Override
27 public Function<double[], OptionalDouble> finisher() {
28  return sumaBroj -> sumaBroj[1] == 0 ?
29    OptionalDouble.empty() :
30    OptionalDouble.of(sumaBroj[0]/sumaBroj[1]);
31 }
32
33 @Override
34 public Set<Collector.Characteristics> characteristics() {
35  return Collections.unmodifiableSet(
36   EnumSet.of(Characteristics.UNORDERED)
37  );
38 }
```

```
39
40 }
```

Ovaj kolektor koristili bismo na sljedeći način.

```
1 OptionalDouble prosjek  =
2  Stream.of(1.0, 5.0, 8.0, 4.0, 12.0, 6.0)
3   .collect(new SrednjaVrijednost());
4
5 System.out.println("Prosjek je: " + prosjek);
```

Puna snaga novog API-ja koji nam stoji na raspolaganju ogleda se u još jednom razredu. U paket java.util.stream dodan je razred `Collectors` koji nudi niz statičkih metoda koje vraćaju unaprijed pripremljene kolektore za rješavanje niza zadataka koji se često javljaju. Za cjelovit pregled čitatelj se upućuje na dokumentaciju tog razreda. Mi ćemo u nastavku dati samo primjer nekoliko raspoloživih kolektora.

Isječak koda koji popis zaposlenika pretvara u listu njihovih plaća:

```
1 List<Double> place = popis.stream()
2   .map(Zaposlenik::getPlaca)
3   .collect(Collectors.toList());
```

Isječak koda koji popis zaposlenika pretvara u skup njihovih plaća:

```
1 Set<Double> places = popis.stream()
2   .map(Zaposlenik::getPlaca)
3   .collect(Collectors.toCollection(TreeSet::new));
```

Pri tome metoda `Collectors.toCollection` prima referencu na tvornicu kolekcija - koju god da ovdje pošaljem, takve će kolekcije biti stvarane i vraćene. Sada je jasno da je metoda `Collectors.toList` korištena u prethodnom isječku samo pokrata za `Collectors.toCollection(ArrayList::new)`.

Isječak koda koji niz stringova spaja u jedan koristeći zadani separator:

```
1 String spojenaImena = imena.stream()
2   .collect(Collectors.joining(", "));
```

Isječak koda koji za popis zaposlenika računa ukupni trošak plaće te prosječnu plaću:

```
1 double trosakPlace11 = popis.stream()
2   .collect(Collectors.summingDouble(Zaposlenik::getPlaca));
3
4 double prosjecnaPlaca11 = popis.stream()
5   .collect(Collectors.averagingDouble(Zaposlenik::getPlaca));
```

Oba prikazana kolektora kao argument primaju referencu na funkciju koja element toka mapira u decimalni broj (pa ne trebamo eksplicitno najprije pozivati mapiranje). Postoje inačice koje izračun rade nad cijelim brojevima tipa `int` i `long`.

Isječak koda koji na temelju popisa zaposlenika gradi mapu čiji su ključevi šifra zaposlenika a vrijednosti zaposlenici:

```
1 Map<String,Zaposlenik> zapPoSifri = popis.stream()
2   .collect(Collectors.toMap(Zaposlenik::getSifra, z->z));
```

Prvi argument kolektora je funkcija koja element toka pretvara u ključ (mi iz zaposlenika dohvaćamo i vraćamo njegovu šifru) a drugi argument je funkcija koja element toka pretvara u vrijednost koju pridružujemo ključu. U našem slučaju želimo uz šifru povezati trenutnog zapos-

lenika što znači da je ova funkcija zapravo funkcija identiteta. To smo mogi i eksplicitno reći koristeći već pripremljenu funkciju identiteta:

```
1 Map<String,Zaposlenik> zapPoSifri = popis.stream()
2   .collect(Collectors.toMap(
3     Zaposlenik::getSifra,
4     Function.identity()
5   ));
```

Da smo htjeli izgraditi mapu koja primjerice širfu zaposlenika mapira na plaću zaposlenika, koristili bismo sljedeći kod:

```
1 Map<String,Double> sifraPlaca = popis.stream()
2   .collect(Collectors.toMap(
3     Zaposlenik::getSifra,
4     Zaposlenik::getPlaca
5   ));
```

Ako želimo izgraditi mapu kojoj su ključevi šifre odjela a vrijednosti liste zaposlenike koji rade u tim odjelima (što je poznato kao operacija grupiranja), koristit ćemo sljedeći kod.

```
1 Map<String,List<Zaposlenik>> poOdjelima = popis.stream()
2   .collect(Collectors.groupingBy(Zaposlenik::getSifraOdjela));
```

Metoda `groupingBy` prima referencu na funkciju koja iz elementa toka vraća vrijednost prema kojoj se određuje u koju grupu upada taj element. U našem primjeru, iz zaposlenika dohvaćamo šifru odjela.

Posljednji primjer ilustrira provođenje binarne particije: zadamo ispitni predikat koji ispituje elemente toka i te elemente želimo podijeliti u dvije kolekcije: jednu u koju idu svi elementi koji ne zadovoljavaju ispitni predikat te drugu u koju idu svi elementi koji zadovoljavaju ispitni predikat. Za tako našto na raspolaganju nam stoji metoda `Collectors.partitioningBy` koja prima referencu na ispitni predikat a vraća mapu čiji su ključevi `Boolean.TRUE` i `Boolean.FALSE` a vrijednosti pripadne kolekcije. Primjerice, idemo izračunati particiju zaposlenika na one koji zarađuju više od 5000 kuna i one ostale.

```
1 Map<Boolean,List<Zaposlenik>> razinaPlace = popis.stream()
2   .collect(
3     Collectors.partitioningBy(z -> z.getPlaca() > 5000.0)
4   );
```

Izvori i poredak elemenata

U prethodnim razmatranjima nismo ništa govorili poretku elemenata u toku. Tok može i ne mora imati definiran poredak elemenata. Izvori koji nastaju iz kolekcija koje imaju definiran poredak generiraju tokove koji imaju definiran poredak (primjer su liste i polja). Izvori koji nastaju iz kolekcija koje nemaju definiran poredak elemenata (poput razreda `HashSet`) generiraju tokove koji nemaju definiran poredak. Primjerice, ako je izvor lista `["Zagreb", "Split", "Vukovar"]` i radimo mapiranje koje elementu pridijeli njegovu duljinu, tj. `e -> e.length()`, ako nam je poredak bitan, jedini prihvatljivi rezultat je lista `[6, 5, 7]`. Ako poredak elemenata nije bitan, tada je i lista `[7, 5, 6]` te bilo koja druga lista koje je njezina permutacija prihvatljivo rješenje.

Ima li tok definiran poredak ili ne, može ograničiti način na koji će pojedine operacije koje su definirane nad tokom biti izvođene. Pri paralelnom izvođenju česta je situacija da je izvođenje operacija nad tokovima koji nemaju poredak učinkovitija (ali postoje dokumentirane iznimke).

Tok se iz poredanog u neporedani može prevesti pozivom metode `.unordered()`. Neke među-operacije mogu tokove bez poretka pretvoriti u tok u kojem elementi imaju poredak; primjer je operacija sortiranja.

Poglavlje 16. Rad s datotečnim sustavom

Datotečni sustav danas je temeljni dio svakog operacijskog sustava. Datotečni sustav korisnicima nudi mjesto za pohranu podataka -- programa, dokumenata te višemedijskih sadržaja. U svakodnevnoj uporabi je mnoštvo programa koji rade s datotečnim sustavom: programi koji korisnicima omogućavaju pregled datotečnog sustava (primjerice kroz hijerarhijski prikaz datoteka i direktorija), programi koji nude usluge pretraživanja datotečnog sustava, programi koji korisnicima nude dohvat informacija o datotečnom sustavu (primjerice, koliko mjesta zauzima određen direktorij i svi njegovi poddirektoriji), programi koji nude mogućnost kopiranja datoteka i čitavih direktorijskih stabala, programi koji omogućavaju izradu sažetih arhiva dijelova datotečnog sustava i rekonstrukciju iz tih arhiva (arhive tipa `tar`, `zip`, `rar`, `gz`, `7z`) te mnogi drugi. U okviru ovog poglavlja pogledat ćemo što nam u Javi stoji na raspolaganju za rad s datotečnim sustavom.

Datotečni sustav konceptualno je izgrađen od hijerarhije objekata datotečnog sustava. Pri tome se pod pojmom *objekt* podrazumijeva bilo kakav entitet koji može postojati na razini datotečnog sustava. Primjerice, ti objekti mogu biti datoteke, direktoriji, te poveznice (linkovi; meke i čvrste). Potpora za rad s datotečnim sustavom u Javi je doživjela dva izdanja. U okviru paketa java.io dostupan je razred `File` koji je u Javi najduže, i mnoštvo drugih biblioteka se oslanja na funkcionalnosti koje on nudi. Stoga ćemo najprije pogledati što nam on nudi. Počev s Javom 7, definirana su nova sučelja te je dodana nova implementacija razreda koji također omogućavaju rad s datotečnim sustavom i pri tome nude niz novih mogućnosti. Oni su smješteni u paket java.nio.file, a radi se o `Path`, `Paths`, `FileSystem`, `FileSystems`, `FileStore`, `Files` te drugima.

Razred `java.io.File`

Razred `java.io.File` ima dvojaku namjenu. Kroz svoje statičke metode i članske varijable nudi korisnicima općenite informacije o datotečnom sustavu i njegovim specifičnostima. Od statičkih članskih varijabli, razred sadrži četiri koje zapravo nude dvije različite informacije na dva različita načina. Evo o čemu se radi.

Svaki datotečni sustav definira način na koji se serijalizira (odnosno zapisuje u obliku teksta) apsolutna staza do nekog objekta tog datotečnog sustava. Primjerice, ako je datotečni sustav NTFS i ako se na disku `C` u direktoriju `Windows` nalazi datoteka `Notepad.exe`, apsolutna staza do tog objekta bila bi `C:\Windows\Notepad.exe`. Ako je pak datotečni sustav Ext3, te ako u korijenskom direktoriju postoji direktorij `etc` u kojem je datoteka `hosts`, apsolutna staza do tog objekta bila bi `/etc/hosts`. Stoga prva informacija koja nas može zanimati je: kako se razdvajaju imena objekata datotečnog sustava prilikom izgradnje apsolutne staze? Odgovor nam nude dvije statičke članske varijable razreda `File`.

```
static String separator;
static char separatorChar;
```

Članska varijabla `File.separator` po tipu je String dok je članska varijabla `File.separatorChar` po tipu `char`. Međutim, obje će sadržavati istu informaciju: simbol koji se na datotečnom sustavu koristi za izgradnju apsolutnih staza. Za datotečni sustav NTFS to će biti simbol `\` dok će za datotečni sustav Ext3 to biti `/`. Kako danas različiti operacijski sustavi dolaze s pretpostavljenim datotečnim sustavom (Windowsi s NTFS-om a Linux s nekom varijantom Ext-a), možemo pričati i o simbolu koji se pretpostavljeno koristi na razini operacijskog sustava.

Svaki datotečni sustav također definira na koji se način u tekst zapisuju skupine apsolutnih staza. Lijep primjer ovoga je varijabla okruženja `PATH` koja i na Windowsima i na Linuxu sadrži popis direktorija koje treba pretražiti kada korisnik upiše ime programa koji želi pokrenuti kako

bi se taj program pronašao. Na neki način potrebno je "zalijepiti" više staza u jedan tekst. Simbol koji se koristi u tu svrhu dostupan je kroz preostale dvije statičke članske varijable razreda File.

```
static String pathSeparator;
static char pathSeparatorChar;
```

Opet, razlika je samo u deklariranom tipu varijable. Na operacijskom sustavu Windows, sadržaj će biti postavljen na ; dok će na operacijskom sustavu Linux sadržaj biti postavljen na :. Primjerice, sadržaj varijable PATH na operacijskom sustavu Linux mogao bi biti /usr/bin:/bin:/usr/sbin.

Pogledajmo jednostavan primjer. Program koji ispisuje sadržaj separatora te separatora staza prikazan je u nastavku.

Primjer 16.1. Ispis separatora

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.io.File;
 4
 5 public class Primjer1 {
 6
 7  public static void main(String[] args) {
 8   System.out.println(
 9     "Separator direktorija je: "+File.separator
10   );
11   System.out.println(
12     "Separator staza je: "+File.pathSeparator
13   );
14  }
15
16 }
17
```

Pokretanjem ovog programa na zaslon ćemo dobiti sljedeći ispis, uz pretpostavku da je program pokrenut na operacijskom sustavu Windows.

```
Separator direktorija je: \
Separator staza je: ;
```

Uz navedene statičke članske varijable, razred File sadrži i tri statičke metode. Dvije od te tri su metode createTempFile uz različiti broj argumenata. Uporabom tih metoda korisniku je omogućeno stvaranje privremenih datoteka bilo u direktoriju koji se na operacijskom sustavu koristi za pohranu privremenih datoteka, bilo u direktoriju koji zada sam korisnik. Metode pri tome garantiraju da će stvoriti datoteku jedinstvenog imena koja u trenutku poziva metode sigurno ne postoji.

Treća statička metoda je metoda File[] listRoots() koja kao rezultat izvođenja vraća polje objekata tipa File. Svaki objekt tog polja predstavlja jedan vršni datotečni objekt koji postoji na razini operacijskog sustava. Radi li se o operacijskom sustavu Windows, dobit ćemo po jedan objekt za svaki disk koji Windowsi prepoznaju. Na operacijskom sustavu Linux dobit ćemo samo jedan objekt -- onaj koji predstavlja korijenski direktorij (/) s obzirom da Linux ima unificirani datotečni sustav. Primjer programa koji ispisuje sve vršne datotečne objekte prikazan je u nastavku.

Primjer 16.2. Ispis vršnih datotečnih objekata

```
1 package hr.fer.zemris.tecaj;
2
```

```
 3 import java.io.File;
 4
 5 public class Primjer2 {
 6
 7  public static void main(String[] args) {
 8   File[] korijeni = File.listRoots();
 9   for(File f : korijeni) {
10    System.out.println(f);
11   }
12  }
13
14 }
15
```

Pokretanjem ovog programa na zaslon ćemo dobiti sljedeći ispis, uz pretpostavku da je program pokrenut na operacijskom sustavu Windows.

```
C:\
D:\
E:\
```

Pogledajmo sada i nestatičke metode razreda `File`. Primjerci razreda `File` predstavljaju apstraktnu reprezentaciju nekog objekta datotečnog sustava koji u datotečnom sustavu možda postoji a možda ne postoji. Razred nudi četiri konstruktora od kojih su dva najčešće korištena:

```
public File(String pathname);
public File(File parent, String child);
```

Prva se inačica konstruktora koristi kada na raspolaganju imamo stazu do objekta datotečnog sustava u obliku stringa. Ta staza pri tome ne treba biti apsolutna: primjerice, navedemo li kao stazu `.`, stvoreni objekt predstavljat će radni direktorij pokrenutog programa. Evo tri primjera.

```
File notepad = new File("C:\\Windows\\Notepad.exe");
File hosts = new File("/etc/hosts");
File trenutni = new File(".");
```

U prvom primjeru svi su separatori navedeni dva puta za redom: radi se o tome da je korektna staza `C:\Windows\Notepad.exe`, no kada to napišemo kao string u izvornom Java-kodu, znak `\` tretira se kao početak escape-sekvence, pa da bismo dobili doista znak `\`, potrebno ga je napisati dva puta: `\\`. Da bi se izbjegao ovaj problem, Java definira znak `/` kao univerzalni separator pa je prvi primjer i na Windowsima moguće napisati na način prikazan u nastavku.

```
File notepad = new File("C:/Windows/Notepad.exe");
```

Jednom kada imamo stvoren primjerak razreda `File`, nad njime možemo pozivati čitav niz različitih metoda. Ovdje ćemo navesti samo neke od zanimljivijih, a čitatelja se upućuje da prouči dokumentaciju ovog razreda.

- `boolean exists();`

  Metoda provjerava postoji li u datotečnom sustavu objekt čija je ovo apstraktna reprezentacija.

- `boolean canRead();`

  `boolean canWrite();`

  `boolean canExecute();`

  Metode provjeravaju može li aplikacija čitati, zapisivati odnosno izvršiti objekt čija je ovo apstraktna reprezentacija.

- `boolean delete();`

  Metoda briše iz datotečnog sustava objekt čija je ovo apstraktna reprezentacija.

- `boolean isDirectory();`

  `boolean isFile();`

  `boolean isHidden();`

  Metode za objekt čija je ovo apstraktna reprezentacija provjeravaju je li on u datotečnom sustavu direktorij, regularna datoteka te je li skriven.

- `long lastModified();`

  `long length();`

  Metode redom vraćaju vrijeme zadnje modifikacije te veličinu objekta datotečnog sustava čija je ovo apstraktna reprezentacija. Ako je trenutni objekt direktorij, poziv metode `length` neće vratiti ukupno zauzeće prostora svih objekata koji su smješteni u tom direktoriju; metoda uvijek vraća koliko je sam objekt velik pa u slučaju da se radi o direktoriju, informacija predstavlja količinu prostora koji datotečni sustav troši za održavanje podataka o samom direktoriju. U to su uključeni, primjerice, troškovi zapisa koji čuvaju podatke o izravnoj djeci tog direktorija.

- `long getTotalSpace();`

  `long getFreeSpace();`

  `long getUsableSpace();`

  Metode vraćaju ukupnu količinu prostora, ukupnu količinu slobodnog prostora te ukupnu količinu iskoristivnog prostora na particiji na kojoj se nalazi i objekt čija je ovo apstraktna reprezentacija.

- `File getAbsoluteFile();`

  `String getAbsolutePath();`

  Obje metode razrješavaju objekt čija je ovo apstraktna reprezentacija u objekt koji u toj reprezentaciji neće imati relativnih segmenata. Primjerice, objekt `new File(".")` bit će razriješen u objekt čije komponente neće biti `.` ili `..` i slične. Međutim, ova metoda ne garantira da će razriješiti simboličke linkove.

- `File getCanonicalFile();`

  `String getCanonicalPath();`

  Obje metode razrješavaju objekt čija je ovo apstraktna reprezentacija u objekt koji u toj reprezentaciji neće imati relativnih segmenata niti simboličkih linkova (u određenom smislu, metode rade jedan korak više od metoda koje računaju apsolutne staze).

- `boolean mkdir();`

  `boolean mkdirs();`

  Obje metode u datotečnom sustavu pokušavaju stvoriti direktorij za koji je ovo apstraktna reprezentacija. Prva metoda pri tome očekuje da roditeljski direktorij postoji; druga metoda će napraviti i sve nedostajuće roditeljske direktorije.

- `boolean renameTo(File dest);`

Metoda mijenja ime objektu datotečnog sustava čija je ovo apstraktna reprezentacija u ime predano kao argument. Kako se kao argument predaje referenca na neku drugu apstraktnu reprezentaciju imena, moguće je da se od metode zatraži čak i seljenje objekta iz jednog direktorija u drugi ili s jedne particije na drugu. Hoće li takva operacija uspjeti, bit će indicirano preko povratne vrijednosti koju svakako treba provjeriti.

- `File[] listFiles();`

  `String[] list();`

  Obje metode za objekt čija je ovo apstraktna reprezentacija dohvaćaju njegovu djecu. Očekuje se da je trenutni objekt direktorij. Ako nije ili ako dođe do pogreške, metoda vraća `null`. Prva navedena metoda prikladnija je za izvođenje algoritama koji dalje rekurzivno obilaze podstablo s obzirom da vraća polje objekata tipa `File`.

- `File[] listFiles(FilenameFilter filter);`

  `String[] list(FilenameFilter filter);`

  Obje metode za objekt čija je ovo apstraktna reprezentacija dohvaćaju njegovu djecu koja zadovolje predani filterski objekt. `FilenameFilter` je sučelje koje definira samo jednu metodu: `boolean accept(File dir, String name);`. Ovoj varijanti metoda `list`/`listFiles` predaje se referenca na objekt koji implementira ovo sučelje. Metoda `list`/`listFiles` za svaki će pronađeni datotečni objekt pozvati metodu `accept` nad predanim filterom i u rezultat uključiti samo one objekte za koje je filterska metoda `accept` vratila vrijednost `true`.

Pogledajmo sada nekoliko primjera. Najprije ćemo napisati program koji će na zaslon izlistati sadržaj zadanog direktorija i svih njegovih poddirektorija.

Primjer 16.3. Rekurzivni ispis sadržaja direktorija

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.io.File;
 4
 5 public class Listaj {
 6
 7  public static void main(String[] args) {
 8   if(args.length != 1) {
 9    System.err.println("Krivi poziv!");
10    System.exit(0);
11   }
12   File dir = new File(args[0]);
13   rekurzivnoListaj(dir);
14  }
15
16  private static void rekurzivnoListaj(File dir) {
17   System.out.println(dir);
18   File[] djeca = dir.listFiles();
19   if(djeca==null) return;
20   for(File f : djeca) {
21    if(f.isFile()) {
22     System.out.println(f);
23    } else if(f.isDirectory()) {
24     rekurzivnoListaj(f);
25    }
26   }
27  }
```

```
28
29 }
30
```

Pokretanjem ovog programa na zaslon ćemo dobiti sljedeći ispis, uz pretpostavku da je program pokrenut na operacijskom sustavu Windows.

```
C:\> java -cp bin hr.fer.zemris.tecaj.Listaj C:\Intel
C:\Intel
C:\Intel\Logs
C:\Intel\Logs\IntelChipset.log
C:\Intel\Logs\IntelStor.log
```

Metoda `main` kao argument očekuje naziv direktorija koji treba izlistati. Za samo listanje zadužena je rekurzivna metoda `rekurzivnoListaj`.

Napišimo sada modificiranu verziju programa koja će generirati ispis u obliku stabla -- ispis naziva svake datoteke te svakog direktorija bit će uvučen to više što je datoteka/direktorij dublja u hijerarhiji koja se ispisuje. Evo rješenja.

Primjer 16.4. Ispis stabla direktorija

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.io.File;
 4
 5 public class Listaj2 {
 6
 7  public static void main(String[] args) {
 8   if(args.length != 1) {
 9    System.err.println("Krivi poziv!");
10    System.exit(0);
11   }
12   File dir = new File(args[0]);
13   rekurzivnoListaj(dir, 0);
14  }
15
16  private static void rekurzivnoListaj(File dir,
17       int indentacija) {
18   if(indentacija==0) {
19    System.out.println(dir);
20   } else {
21    System.out.format(
22      "%"+indentacija+"s%s%n", "", dir.getName()
23    );
24   }
25   File[] djeca = dir.listFiles();
26   if(djeca==null) return;
27   indentacija += 2;
28   for(File f : djeca) {
29    if(f.isFile()) {
30     System.out.format(
31       "%"+indentacija+"s%s%n", "", f.getName()
32     );
33    } else if(f.isDirectory()) {
34     rekurzivnoListaj(f, indentacija);
35    }
36   }
37  }
```

```
38
39 }
40
```

Pokretanjem ovog programa na zaslon ćemo dobiti sljedeći ispis, uz pretpostavku da je program pokrenut na operacijskom sustavu Windows.

```
C:\> java -cp bin hr.fer.zemris.tecaj.Listaj2 C:\Intel
C:\intel
  Logs
    IntelChipset.log
    IntelStor.log
```

I u ovom rješenju metoda `main` kao argument očekuje naziv direktorija koji treba izlistati. Za samo listanje zadužena je rekurzivna metoda `rekurzivnoListaj` koja u ovoj inačici prima dva argumenta: trenutni direktorij te trenutnu razinu uvlačenja. Sa svakim rezkurzivnim pozivom razina uvlačenja se povećava.

Konačno, pogledajmo još jedan primjer: napišimo program koji će omogućiti da preko argumenata naredbenog retka zadamo direktorij te željenu ekstenziju. Program treba ispisati sve datoteke (i direktorije, ako takvi postoje) čije ime sadrži zadanu ekstenziju. Pretpostavimo da rješenje treba temeljiti na uporabi metode `File.listFiles(FilenameFilter filter)`.

Rješenje zadatka prikazano je u nastavku.

Primjer 16.5. Filtrirani ispis sadržaja direktorija

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.io.File;
 4 import java.io.FilenameFilter;
 5
 6 public class Primjer3 {
 7
 8  public static void main(String[] args) {
 9   if(args.length != 2) {
10    System.err.println("Krivi poziv!");
11    System.exit(0);
12   }
13   File dir = new File(args[0]);
14   ispisiFormatirano(dir, new FilterPoEkstenziji(args[1]));
15  }
16
17  public static void ispisiFormatirano(File dir,
18     FilenameFilter filter) {
19   File[] djeca = dir.listFiles(filter);
20   if(djeca==null) return;
21   for(File f : djeca) {
22    if(f.isDirectory()) {
23     System.out.format("[DIR]            %s%n", f.getName());
24    } else {
25     System.out.format(
26       "[FIL] %10d %s%n", f.length(), f.getName()
27     );
28    }
29   }
30  }
31
32  static class FilterPoEkstenziji implements FilenameFilter {
```

```
33
34   private String ekstenzija;
35
36   public FilterPoEkstenziji(String ekstenzija) {
37    this.ekstenzija = "." + ekstenzija;
38   }
39
40   @Override
41   public boolean accept(File dir, String name) {
42    if(name.length() < ekstenzija.length()) {
43     return false;
44    }
45    int pocetak = name.length()-ekstenzija.length();
46    String ocitanaEkstenzija = name.substring(pocetak);
47    return ocitanaEkstenzija.equalsIgnoreCase(ekstenzija);
48   }
49  }
50 }
51
```

Pokretanjem ovog programa na zaslon ćemo dobiti sljedeći ispis, uz pretpostavku da je program pokrenut na operacijskom sustavu Windows.

```
C:\> java -cp bin hr.fer.zemris.tecaj.Primjer3 C:\Windows exe
[FIL]      30208 agrdel64.exe
[FIL]      55816 agrsmdel.exe
[FIL]      65536 bfsvc.exe
[FIL]    3079168 explorer.exe
[FIL]      14848 fveupdate.exe
[FIL]     734720 HelpPane.exe
[FIL]      15872 hh.exe
[FIL]     306688 IsUninst.exe
[FIL]     169472 notepad.exe
[FIL]     134656 regedit.exe
[FIL]      27184 snuvcdsm.exe
[FIL]      39936 splwow64.exe
[FIL]      49680 twunk_16.exe
[FIL]      31232 twunk_32.exe
[FIL]       9216 winhlp32.exe
```

U prikazanom rješenju metoda `main` očekuje dva argumenta: naziv direktorija koji treba izlistati te ekstenziju koju trebaju imati prikazane datoteke. Zadaću listanja i ispisa implementirali smo u metodi `ispisiFormatirano`. Metoda kao argument dobiva direktorij koji treba listati te filter koji treba primijeniti.

Filter imena prema zadanoj ekstenziji implementiran je kao novi privatni statički razred čiji je kod dan od retka 32. Razred definira konstruktor koji prihvaća željenu ekstenziju te metodu `accept` koju je prema ugovoru sa sučeljem `FilenameFilter` dužan ponuditi.

Paket java.nio.file

Paket java.nio.file sadrži reimplementaciju i nadogradnju podrške za rad s datotečnim sustavima i njihovim objektima. Apstraktna reprezentacija objekata datotečnog sustava sada je modelirana na najapstraktniji mogući način: uporabom sučelja `Path`. Gledano sa stajališta tog sučelja, svaka se staza sastoji od opcionalnog korijena te nula, jednog ili više elemenata koji čine imena objekata. Korijen, ako je prisutan, određuje disk (particiju) na kojoj se nalazi zadana staza. Za stvaranje primjeraka razreda koji implementiraju ovo sučelje na raspolaganju nam je razred `Paths` koji nudi prikladne statičke metode. Pogledajmo konkretan primjer.

Primjer 16.6. Modeliranje staze sučeljem `Path`

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.nio.file.Path;
 4 import java.nio.file.Paths;
 5
 6 public class Primjer4 {
 7
 8  public static void main(String[] args) {
 9   Path p = Paths.get(
10     "C:\\", "Windows", "System32", "calc.exe"
11   );
12
13   System.out.println("Informacije o korijenu:");
14   System.out.println("----------------------");
15   System.out.println(p.getRoot());
16   System.out.println(p.getRoot().getNameCount());
17   System.out.println();
18
19   System.out.println("Informacije o elementima:");
20   System.out.println("-------------------------");
21   int brojElemenata = p.getNameCount();
22   for(int i = 0; i < brojElemenata; i++) {
23    Path pocetni = p.getName(i);
24    System.out.println(pocetni);
25   }
26   System.out.println();
27
28   System.out.println("Različite druge operacije:");
29   System.out.println("--------------------------");
30   System.out.println(p.toAbsolutePath());
31   System.out.println(p.subpath(0, 2));
32   System.out.println();
33
34   info(p.getParent());
35   info(p.getParent().getParent());
36   info(p.getParent().getParent().getParent());
37   info(p.getParent().getParent().getParent().getParent());
38  }
39
40  private static void info(Path p) {
41   if(p==null) {
42    System.out.println("null");
43    return;
44   }
45   System.out.println(
46     p + ", broj elemenata: " + p.getNameCount() +
47     ", korijen: " + p.getRoot()
48   );
49  }
50 }
51
```

Pokretanjem ovog programa dobit ćemo sljedeći ispis (na Windowsima). Redni brojevi redaka dodani su u prikaz kako bismo se mogli lakše kasnije pozvati na pojedine retke.

```
1 Informacije o korijenu:
```

```
 2 ----------------------
 3 C:\
 4 0
 5
 6 Informacije o elementima:
 7 -------------------------
 8 Windows
 9 System32
10 calc.exe
11
12 Različite druge operacije:
13 --------------------------
14 C:\Windows\System32\calc.exe
15 Windows\System32
16
17 C:\Windows\System32, broj elemenata: 2, korijen: C:\
18 C:\Windows, broj elemenata: 1, korijen: C:\
19 C:\, broj elemenata: 0, korijen: C:\
20 null
```

Prokomentirajmo malo dobiveni rezultat.

U retku 9 programa stvaramo stazu koja ima zadan korijen (`C:\`) te tri elementa. Pozivom metode `getRoot()` nad stazom dobiva se nova staza koja ima identičan korijen kao i originalna staza nad kojom smo pozvali metodu te nema niti jednog elementa. Ispisi koji se rade u retku 15 i 16 programa to potvrđuju (ispisan redak 3 prikazuje samo `C:\` a ispisan redak 4 potvrđuje da je broj elemenata u toj novoj stazi jednak 0).

Petlja `for` u retku 22 programa obilazi sve elemente staze. U promatranom slučaju, poziv `p.getNamesCount()` vraća `3`: staza sadrži slijed od tri imena (`Windows`, `System32` te `calc.exe`). Metoda `getName(int index)` pozvana nad stazom vraća novu stazu koja sadrži samo jedan element (upravo indeksirani) i ne sadrži korijen. Takva staza je po definiciji postala relativna: više se niti ne zna kojem disku pripada, niti se zna tko joj je roditelj. Ispisani retci 8, 9 i 10 to ilustriraju.

Pozivom metode `toAbsolutePath()` dobiva se nova staza koja predstavlja trenutnu stazu razriješenu oko trenutnog direktorija programa. Ako je početna staza već bila apsolutna, razrješavanje će je vratiti nepromijenjenu.

Pozivom metode `subpath(startIndex,endIndex)` dobiva se nova staza koja sadrži elemente originalne staze počev od onog određenog s `startIndex` (koji je uključen) pa do zadnjeg elementa koji je prije elementa određenog indeksom `endIndex`; drugim riječima, `endIndex` je isključiv. Tako dobivena staza ne preuzima korijen originalne staze: korijen u toj stazi nije postavljen i staza je opet relativna. U konkretnom slučaju, pozivom `subpath(0,2)` dobiva se staza koja od originalne staze preuzima slijed elemenata s lokacija 0 i 1. To je vidljivo u ispisu u retku 15.

Konačno, trebamo li za zadanu stazu dohvatiti roditeljsku stazu, ona je dostupna pozivom metode `getParent()`. Ovako dobivena staza imat će jedan element manje no što ga ima originalna staza (posljednji element neće biti prisutan) i preuzet će korijen od originalne staze, kako je u njoj definiran. Rezultat je vidljiv u ispisu u retcima 17 do 20. Staza u kojoj nema zadanih elemenata više nema roditeljske staze pa u tom slučaju metoda `getParent` vraća `null` (ispis, redak 20).

Sučelje `Path` uz navedene metode definira još čitav niz drugih metoda koje djeluju nad stazama: postoje metode za razrješavanje relativnih staza, za relativizaciju jedne staze s obzirom na drugu stazu, metoda za dohvat iteratora kojim se može obilaziti po elementima koji čine stazu, metode za dohvat imena objekta koji staza predstavlja (što je zapravo posljednji element u stazi koji je zbog prikladnosti dostupan i direktno pozivom metode `getFileName`), metoda za dohvat objekta koji predstavlja datotečni sustav kojemu ova staza pripada (`getFileSystem()`)

i slično. Međutim, ne postoje metode koje bi vraćale bilo kakve informacije o objektima datotečnog sustava koje staza opisuje. Za razliku od razreda `java.io.File` koji je objedinjavao heterogen niz različitih funkcionalnosti, u novom Javinom API-ju različite dužnosti implementirane su u različitim dijelovima paketa.

Ponovimo što smo do sada obradili.



Bilješka

Sučelje `Path` predstavlja apstraktnu reprezentaciju staze koja je sastavljena od opcionalnog korijena te nula, jednog ili više elemenata. Sučelje definira niz korisnih metoda za manipuliranje trenutnom stazom te za kombiniranje više staza. Uobičajen način stvaranja primjeraka razreda koji implementiraju ovo sučelje je statičkim metodama pomoćnog razreda `Paths`.

Uz prikazani primjer dobivanja staze pozivom metode `Paths.get(...)` kojoj smo dali element po element, metodi je moguće predati i cjelokupnu stazu kao jedan argument kako to pokazuje sljedeća dva primjera u nastavku.

```
Path p1 = Paths.get("C:\\Windows\\System32\\calc.exe");
Path p1 = Paths.get("C:/Windows/System32/calc.exe");
```

Pogledajmo još jedan primjer. Potrebno je napisati program koji će pratiti stanje nekog direktorija i svaki puta kada se u direktoriju dogodi neki od unaprijed specificiranih događaja (primjerice, kada se stvori nova datoteka ili kada se modificira postojeća), program treba na zaslon ispisati informaciju o tome. Ovakvi programi imaju vrlo praktičnu primjenu: uređivači teksta mogu na taj način pratiti je li netko drugi izvana promijenio datoteku koju korisnik upravo uređuje i o tome mogu obavijestiti korisnika ("Želite li učitati novu verziju ili želite zadržati postojeću?"); programi za konverziju formata datoteka mogu pratiti stanje u zadanom direktoriju i svaki puta kada korisnik u direktorij ubaci datoteku određenog formata, oni mogu automatski krenuti u pretvorbu datoteke u drugi format (primjerice, iz `.ps` datoteka u letu generirati `.pdf` datoteke) i slično.

Loš način implementacije ovakve funkcionalnosti bio bi napisati program koji prilikom pokretanja u memoriju učita sve podatke o svim postojećim datotekama u zadanom direktoriju i koji potom svakih primjerice 10 sekundi nanovo lista sadržaj direktorija i uspoređuje zapamćeno stanje s trenutnim stanjem. Kako moderni operacijski sustavi već odavno nude ovakvu funkcionalnost, od Jave 7 ona je dostupna i korisnicima Jave.

Primjer 16.7. Nadzor direktorija uporabom `WatchService`

```
 1 package hr.fer.zemris.tecaj;
 2
 3 import java.io.IOException;
 4 import java.nio.file.Path;
 5 import java.nio.file.Paths;
 6 import java.nio.file.StandardWatchEventKinds;
 7 import java.nio.file.WatchEvent;
 8 import java.nio.file.WatchKey;
 9 import java.nio.file.WatchService;
10
11 public class NadzorDirektorija {
12
13  public static void main(String[] args) throws IOException {
14   Path p = Paths.get("C:\\", "Temp");
15
16   try(WatchService service =
17       p.getFileSystem().newWatchService()) {
18    WatchKey key = null;
19    try {
```

```
20      key = p.register(
21       service,
22       StandardWatchEventKinds.ENTRY_CREATE,
23       StandardWatchEventKinds.ENTRY_MODIFY);
24     } catch (IOException e) {
25      e.printStackTrace();
26      return;
27     }
28
29     while(true) {
30      try {
31       key = service.take();
32      } catch (InterruptedException e) {
33       return;
34      }
35
36      for(WatchEvent<?> event : key.pollEvents()) {
37       WatchEvent.Kind<?> kind = event.kind();
38       if(kind == StandardWatchEventKinds.OVERFLOW) {
39        continue;
40       }
41
42       WatchEvent<Path> pathEvent = (WatchEvent<Path>)event;
43       Path path = pathEvent.context();
44       System.out.println(
45         "Zabilježen događaj " + kind +
46         " nad stazom " + path
47       );
48      }
49
50      boolean valid = key.reset();
51      if(!valid) {
52       break;
53      }
54     }
55    }
56   }
57
58 }
59
```

Pokretanjem ovog programa dobit ćemo sljedeći ispis (na Windowsima). Nakon pokretanja programa programom `Notepad` stvorena je datoteka `ocjene.txt` koja je potom još jednom modificirana. Zatim je u direktorij iskopirana i jedna `PDF` datoteka.

```
Zabilježen događaj ENTRY_CREATE nad stazom Ocjene.txt
Zabilježen događaj ENTRY_MODIFY nad stazom Ocjene.txt
Zabilježen događaj ENTRY_CREATE nad stazom ui_salabahter.pdf
Zabilježen događaj ENTRY_MODIFY nad stazom ui_salabahter.pdf
Zabilježen događaj ENTRY_MODIFY nad stazom ui_salabahter.pdf
```

Kako program radi? Program započinje stvaranjem staze koja predstavlja direktorij koji će biti nadziran (redak 14). Potom se za tu stazu dohvaća objekt koji predstavlja pripadni datotečni sustav i traži se stvaranje objekta tipa `WatchService` koji može raditi s tim datotečnim sustavom. Stvaranje ovog objekta riješeno je u bloku `try` na način koji će osigurati da se po izlasku iz tog bloka stvoreni objekt automatski zatvori.

Nakon stvaranja objekta `service` obavlja se registracija staze `p` nad tim objektom (redak 20) pri čemu se definira koje sve događaje objekt `service` treba dojavljivati. U primjeru smo

odabrali da nas zanimaju dvije vrste događaja: stvaranje objekata te modificiranje objekata u nadziranom direktoriju.

Po uspješno obavljenoj registraciji, ulazi se u beskonačnu petlju (redak 29). Poziva se metoda `service.take()` koja će zablokirati tako dugo dok nema novih događaja -- pozivom te metode naš program prelazi u fazu spavanja. Kada se dogodi promjena u nadziranom direktoriju, program će se odblokirati; tada ulazimo u petlju `for` koja obilazi po kolekciji svih prikupljenih događaja (moguće je da se prilikom buđenja ustanovi da postoji nekoliko registriranih događaja). Osim događaja za koje smo se registrirali, moguće je da se u popisu zabilježenih događaja pojavi događaj tipa `StandardWatchEventKinds.OVERFLOW`: tim nas događajem sustav obavještava da se u međuvremeno dogodio jedan ili više događaja koje nismo dobili. Ako je trenutni događaj takav, zanemarujemo ga i prelazimo na sljedeći. Ako događaj nije `OVERFLOW`, tada je jedan od registriranih pa događaj ukalupljujemo u `WatchEvent<Path>` i potom objekt koji predstavlja stazu do objekta datotečnog sustava koji je uzrokovao događaj dohvaćamo pozivom metode `context()`, redak 43. Jednom kada znamo o kojem se objektu datotečnog sustava radi, na zaslon ispisujemo traženu informaciju i prelazimo na sljedeći događaj.

Nakon prolaska kroz sve zabilježene događaje, nad ključem se poziva metoda `reset()` što je nužno kako bi se moglo nastaviti primanje informacija o novim događajima. Po resetu dobivenog ključa, program se vraća u metodu `service.take()` koja će ga ponovno zablokirati sve do generiranja novih događaja.

Umjesto blokirajuće metode `take()`, na raspolaganju su nam i metode koje ne blokiraju izvođenje ili ga blokiraju samo na unaprijed zadani vremenski interval; ovisno o vrsti programa koji se piše, takve metode ponekad znaju biti praktičnije.

Razred `java.nio.file.Files`

Usporedimo li funkcionalnost koju nam nudi sučelje `java.nio.file.Path` s funkcionalnošću razreda `java.io.File`, uočit ćemo da nam nedostaje jedan važan dio: dio koji na temelju apstraktne reprezentacije staze nudi informacije o objektu datotečnog sustava čija je to staza. U novoj verziji Javinog API-ja, ta je funkcionalnost izdvojena u razred `java.nio.file.Files`. Ovaj razred predstavlja repozitorij statičkih metoda koje kao argumente primaju objekte tipa `Path` te nude odgovarajuće usluge. U nastavku ovog poglavlja osvrnut ćemo se na dio sadržane funkcionalnosti.

Kako je razred `Files` konceptualno nadomjestak za nedostajući dio funkcionalnosti koju je nudi razred `File` a koji sučelje `Path` ne nudi, za očekivati je da ćemo u njemu pronaći odgovarajuće metode. I doista, metode su tamo i dapače, nude i više opcija no što je to nudio razred `File`. Tako primjerice metodom:

```
boolean exists(Path path, LinkOption ... options);
```

možemo ispitati postoji li na datotečnom sustavu objekt za koji je dana staza kao prvi argument. Dapače, ako kao drugi argument predamo element `NOFOLLOW_LINKS` enumeracije `LinkOption`, rezultat će biti `true` samo ako staza na disku doista postoji i ako niti jedan njezin dio nije simbolički link. Slično, metodom:

```
boolean notExists(Path path, LinkOption ... options);
```

možemo provjeriti da objekt datotečnog sustava ne postoji.

Za ispitivanje vrste objekta datotečnog sustava na raspolaganju su nam sljedeće metode.

```
boolean isDirectory(Path path, LinkOption ... options);
boolean isRegularFile(Path path, LinkOption ... options);
boolean isSymbolicLink(Path path);
```

Prva metoda provjerava predstavlja li predana staza direktorij, druga metoda provjerava predstavlja li predana staza "normalnu" datoteku dok treća metoda provjerava predstavlja li predana staza simbolički link.

Uz navedeno, dostupne su i sljedeće informacije.

```
boolean isExecutable(Path path);
boolean isHidden(Path path);
boolean isReadable(Path path);
boolean isWritable(Path path);
```

Metode redom ispituju za objekt datotečnog sustava čiju stazu dobivaju kao argument: je li on izvršiv, je li skriven, može li ga korisnik čitati te može li korisnik u njega zapisivati.

Na raspolaganju imamo i metode kojima možemo doznati više informacija o objektima datotečnog sustava.

```
<V extends FileAttributeView> V getFileAttributeView(
  Path path, Class<V> type, LinkOption... options
);
<A extends BasicFileAttributes> A readAttributes(
  Path path, Class<A> type, LinkOption... options
);
Set<PosixFilePermission> getPosixFilePermissions(
  Path path, LinkOption... options
);
```

Navedene metode nude mogućnost dohvata objekta koji enkapsulira čitav niz informacija o objektu datotečnog sustava, pa nakon dohvata takvog objekta informacije možemo dohvaćati vrlo efikasno. Evo najprije primjer uporabe metode `getFileAttributeView`.

```
Path p = Paths.get(" ... neka staza ...");
  BasicFileAttributeView view = Files.getFileAttributeView(
    p, BasicFileAttributeView.class
  );
  if(view != null) {
    BasicFileAttributes a = view.readAttributes();
    System.out.println(a.creationTime());
    System.out.println(a.lastAccessTime());
    System.out.println(a.lastModifiedTime());
    System.out.println(a.size());
    System.out.println(a.isRegularFile());
    System.out.println(a.isDirectory());
    System.out.println(a.isSymbolicLink());
    System.out.println(a.isOther());
  }
```

Uz pogled `BasicFileAttributeView` koji omogućava dohvat osnovnih atributa objekta datotečnog sustava (modeliranih s `BasicFileAttributes`), definiran je još niz drugih pogleda koji omogućavaju dohvat još mnoštva dodatnih informacija; ovdje ćemo samo nabrojati poglede a čitatelja se upućuje da detaljnije istraži iste u službenoj dokumentaciji. Pogledi su: `AclFileAttributeView`, `BasicFileAttributeView`, `DosFileAttributeView`, `FileOwnerAttributeView`, `PosixFileAttributeView` te `UserDefinedFileAttributeView`.

Uporabom metode `readAttributes` mogli smo doći do istih podataka, samo na malo drugačiji način, kako je prikazano u nastavku.

```
Path p = Paths.get(" ... neka staza ...");
  BasicFileAttributes a = Files.readAttributes(
    p, BasicFileAttributes.class
  );
  System.out.println(a.creationTime());
  ...
```

```
  System.out.println(a.isOther());
```

Kao i u prethodnom slučaju, metodu možemo koristiti za dohvat niza različitih vrsta atributa; ovdje ćemo ih samo nabrojati: `DosFileAttributes`, `PosixFileAttributes`.

Konačno, metodom `getPosixFilePermissions` možemo direktno dohvatiti dozvole koje su nad objektom podešene a koje poznaje POSIX specifikacija (radi se o zastavicama: čitljivo, zapisivo te izvršivo i to posebno za vlasnika objekta, posebno za grupu korisnika kojoj je pridružen i objekt te posebno za sve ostale korisnike). Evo primjera.

```
import static java.nio.file.attribute.PosixFilePermission.*;
...
Set<PosixFilePermission> perm = Files.getPosixFilePermissions(p);
System.out.println(set.contains(OWNER_READ));
System.out.println(set.contains(OWNER_WRITE));
System.out.println(set.contains(OWNER_EXECUTE));
System.out.println(set.contains(GROUP_READ));
System.out.println(set.contains(GROUP_WRITE));
System.out.println(set.contains(GROUP_EXECUTE));
System.out.println(set.contains(OTHERS_READ));
System.out.println(set.contains(OTHERS_WRITE));
System.out.println(set.contains(OTHERS_EXECUTE));
```

U nastavku ćemo navesti još nekoliko preostalih metoda koje nude informacije o objektima datotečnog sustava.

```
FileTime getLastModifiedTime(Path path, LinkOption... options);
String probeContentType(Path path);
long size(Path path);
```

Metode redom vraćaju: datum i vrijeme posljednje izmjene objekta datotečnog sustava, *mime-tip* objekta (što je posebno zgodna informacija ako programirate vlastiti web-poslužitelj) te veličinu objekta.

Od operacija, na raspolaganju nasm stoje metode za stvaranje strukture direktorija temeljem predane staze, za stvaranje datoteka, brisanje datoteka, premještanje datoteka, kopiranje datoteka, stvaranje linkova, privremenih datoteka i direktorija i niz drugih. Ovdje ćemo navesti samo najčešće korištene metode a čitatelja se upućuje na detaljniji pregled ponuđenih metoda u službenoj dokumentaciji. Evo prototipova zanimljivijih metoda.

```
Path copy(Path source, Path target, CopyOption... options);
void delete(Path path);
boolean deleteIfExists(Path path);
Path move(Path source, Path target, CopyOption... options);
Path createDirectories(Path dir, FileAttribute<?>... attrs);
Path createDirectory(Path dir, FileAttribute<?>... attrs);
Path createFile(Path path, FileAttribute<?>... attrs);
Path createLink(Path link, Path existing);
Path createSymbolicLink(
  Path link, Path target, FileAttribute<?>... attrs
);
```

Prije kraja opisa metoda ovog razreda izdvojit ćemo još tri vrste metoda: metode koje nude mogućnost obilaska djece direktorija (varijante metode `newDirectoryStream`), metode koje omogućavaju učitavanje sadržaja datoteke u memoriju (poput `readAllBytes` i `readAllLines`) i zapisivanje sadržaja memorije u datoteku (varijante metode `write`) te metode koje nude mogućnost obilaska cjelokupnog podstabla datotečnog sustava (varijante metode `walkFileTree`).

Krenimo stoga redom kroz svaku od vrsta i ilustrirajmo to jednostavnim primjerima.

Ispis sadržaja direktorija uporabom DirectoryStream

Objekt tipa DirectoryStream predstavlja pogled na objekte datotečnog sustava koji su smješteni neposredno u nekom direktoriju. Taj pogled ne mora biti cjelovit: moguće je da pogled sadrži samo one objekte koji zadovoljavaju neki definirani kriterij. Ako pretpostavimo da je vršni direktorij definiram stazom dir, objekt tipa DirectoryStream za zadani direktorij dir možemo dobiti pozivom jedne od triju statičkih metoda Files.newDirectoryStream(dir, ...). Metode se razlikuju samo u (ne)postojanju i vrsti drugog argumenta koji predstavlja filter. Dobiveni objekt implementira sučelje Iterable<Path> tako je nad njim moguće pozvati metodu .iterator() kojom ćemo dohvatiti iterator i potom njime proći kroz sve elemente. Alternativno, objekt se može direktno koristiti u skraćenom obliku petlje for.

Konkretan primjer prikazan je u nastavku.

Primjer 16.8. Ispis sadržaja direktorija

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.IOException;
 4 import java.nio.file.DirectoryStream;
 5 import java.nio.file.Files;
 6 import java.nio.file.Path;
 7 import java.nio.file.Paths;
 8
 9 public class IspisDirektorija {
10
11  public static void main(String[] args) {
12   if(args.length != 1) {
13    System.out.println("Pogrešan poziv!");
14    return;
15   }
16   Path dir = Paths.get(args[0]);
17   try(DirectoryStream<Path> stream =
18         Files.newDirectoryStream(dir, "*.exe")) {
19    for(Path p : stream) {
20     System.out.println(p.getFileName());
21    }
22   } catch(IOException ex) {
23    System.err.println(ex.getMessage());
24   }
25  }
26
27 }
28
```

Pokretanjem ovog primjera dobit ćemo sljedeći ispis.

```
java -cp bin hr.fer.zemris.java.primjeri.IspisDirektorija
    C:\Windows\System32
ACW.exe
AdapterTroubleshooter.exe
agrsmdel.exe
ARP.EXE
at.exe
...
```

```
wuapp.exe
wusa.exe
xcopy.exe
```

U primjeru je korištena varijanta metode `newDirectoryStream` koja kao drugi argument prima jednostavan filter zadan kao string: konkretno, tražili smo sve objekte koji imaju ekstenziju `exe` pa smo zadali filter `"*.exe"`. Metoda `newDirectoryStream` koja ne prima drugi argument vratila bi `DirectoryStream` kojim bismo dobili sve sadržane objekte zadanog direktorija. Treća varijanta metode kao drugi argument prima bilo koji objekt koji implementira sučelje `DirectoryStream.Filter<? super Path>`: to sučelje definira samo jednu metodu (`accept`) koja se koristi za filtriranje objekata koje treba proslijediti u `DirectoryStream` koji će biti vraćen.

Neovisno o načinu stvaranja objekta `DirectoryStream`, nakon uporabe objekt treba zatvoriti kako bi se otpustili svi resursi koje je objekt zauzeo. Iz tog razloga smo u prikazanom primjeru umjesto eksplicitnog zatvaranja objekta isti stvoriti u proširenoj verziji bloka `try` koji će sam na kraju zatvoriti taj objekt.

Čitanje i pisanje sadržaja datoteke

Za čitanje i pisanje datoteka, razred `Files` nudi nam dvije vrste pomoćnih metoda na koje ćemo se u ovom trenutku osvrnuti. Postoje metode za učitavanje i zapisivanje binarnih datoteka te metode za učitavanje i zapisivanje tekstovnih datoteka. Evo primjera.

Primjer 16.9. Čitanje iz i zapisivanje u datoteku

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.IOException;
 4 import java.nio.charset.StandardCharsets;
 5 import java.nio.file.Files;
 6 import java.nio.file.Path;
 7 import java.nio.file.Paths;
 8 import java.util.Arrays;
 9 import java.util.List;
10
11 public class PisiCitaj {
12
13  public static void main(String[] args) throws IOException {
14
15   primjerBinarni();
16   primjerTekstovni();
17
18  }
19
20  private static void primjerBinarni() throws IOException {
21   // Binarni sadržaj za datoteku
22   byte[] sadrzaj = {1, 2, 5, 7, 11};
23
24   // Staza do datoteke:
25   Path binarna = Paths.get("binarna.bin");
26
27   // Zapiši sadržaj u datoteku:
28   Files.write(binarna, sadrzaj);
29
30   // Pročitaj sadržaj iz datoteke:
31   byte[] novi = Files.readAllBytes(binarna);
32
```

```
33    // Usporedi zapisano i pročitano:
34    System.out.println(
35      "Jednaki: " + Arrays.equals(sadrzaj, novi)
36    );
37   }
38
39   private static void primjerTekstovni() throws IOException {
40    // Lista redaka koje treba pohraniti u datoteku:
41    List<String> lista = Arrays.asList(
42      "Zagreb", "Bjelovar", "Šibenik", "Split"
43    );
44
45    // Staza do datoteke:
46    Path tekstovna = Paths.get("gradovi.txt");
47
48    // Zapiši sadržaj u datoteku:
49    Files.write(tekstovna, lista, StandardCharsets.UTF_8);
50
51    // Pročitaj sadržaj iz datoteke:
52    List<String> ucitana = Files.readAllLines(
53      tekstovna, StandardCharsets.UTF_8
54    );
55
56    // Usporedi zapisano i pročitano:
57    System.out.println("Jednako: " + lista.equals(ucitana));
58   }
59
60  }
61
```

Uz metodu `main`, prikazani primjer sadrži još dvije metode. Metoda `primjerBinarni` radi s binarnim sadržaj. U retku 22 definirano je polje binarnih podataka. U retku 28 to se polje zapisuje u datoteku. Način na koji je upotrijebljena metoda `write` osigurava da u slučaju da je datoteka prethodno postojala, da će njezin sadržaj biti uklonjen i zamijenjen predanim. Ako nije postojala, nastat će i opet preuzeti predani sadržaj. Ova metoda u pozadini otvara datoteku, zapisuje sadržaj i potom zatvara datoteku. U retku 31 koristi se metoda `readAllBytes` koja radi upravo suprotno: otvara datoteku za čitanje, čita njezin cjelokupni sadržaj i potom zatvara datoteku; pročitani sadržaj vraća kao novo polje.

Metoda `primjerTekstovni` radi slično što i opisana metoda, samo što podatci nisu binarni već su tekst. Da bismo tekst zapisali u datoteku (koja je po definiciji binarni medij), nužno je definirati koja će se kodna stranica koristiti prilikom zapisivanja odnosno prilikom čitanja. U ovom primjeru korištena je konda stranica `UTF-8`. Metoda `write` koju sada koristimo kao drugi argument prihvaća bilo koji iterabilni objekt čiji su elementi znakovni slijedovi (modelirani sučeljem `CharSequence`): stringovi to jesu pa smo kao sadržaj pripremili listu stringova koja predstavlja retke datoteke. Metoda `readAllLines` prima stazu do datoteke te referencu na kodnu stranicu; čita sadržaj datoteke i vraća ga kao listu stringova.

Izračun statističkih informacija o zadanom podstablu

Kao posljednji primjer razmotrit ćemo pisanje programa koji za zadani direktorij i njegovo podstablo računa općenite podatke: broj prisutnih datoteka, ukupna veličina prisutnih datoteka, prosječna veličina datoteka i broj direktorija, te dodatno prikuplja informacije za svaku vrstu datoteka pri čemu vrste razlikuje po ekstenziji (opet se traži broj, ukupna veličina te prosječna veličina). Program na zaslon treba ispisati opće podatke te potom za svaku pronađenu ekstenziju podatke prikupljene o datotekama s tom ekstenzijom. Podatci o ekstenzijama trebaju biti ispisani sortirano prema ekstenziji od leksikografski posljednje ekstenzije.

Jedno moguće rješenje prikazano je u nastavku. Obilazak podstabla riješen je rekurzivno.

Primjer 16.10. Izračun statističkih podataka, 1. pokušaj

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.File;
 4 import java.util.ArrayList;
 5 import java.util.Collections;
 6 import java.util.Comparator;
 7 import java.util.HashMap;
 8 import java.util.List;
 9 import java.util.Map;
10
11 public class Statistika {
12
13  public static void main(String[] args) {
14   if(args.length != 1) {
15    System.err.println("Krivi poziv!");
16    System.exit(0);
17   }
18
19   File dir = new File(args[0]);
20
21   Informacije info = obradi(dir);
22
23   System.out.println(
24     "Broj datoteka je: " + info.brojPojavaDatoteka);
25   System.out.println(
26     "Ukupna velicina datoteka je: " +
27     info.ukupnaVelicinaDatoteka);
28   System.out.println(
29     "Prosjecna velicina datoteka je: " +
30     info.prosjekDatoteka());
31   System.out.println(
32     "Broj direktorija je: " + info.brojPojavaDirektorija);
33
34   List<InfoOEkstenziji> lista = new ArrayList<>(
35     info.mapa.values()
36   );
37
38   Comparator<InfoOEkstenziji> komparator =
39     new Comparator<InfoOEkstenziji>() {
40    @Override
41    public int compare(InfoOEkstenziji o1,
42                       InfoOEkstenziji o2) {
43     return o1.ekstenzija.compareTo(o2.ekstenzija);
44    }
45   };
46
47   Collections.sort(
48     lista, Collections.reverseOrder(komparator)
49   );
50
51   for(InfoOEkstenziji i : lista) {
52    System.out.println(
53      i.ekstenzija + " " + i.brojPojava + " " +
54      i.ukupnaVelicina + " " + i.prosjek()
55    );
56   }
```

```
57
58  }
59
60  private static Informacije obradi(File dir) {
61   Informacije info = new Informacije();
62   rekurzivnoListaj(dir, info);
63   return info;
64  }
65
66  private static void rekurzivnoListaj(File dir,
67     Informacije info) {
68   info.obradiDirektorij(dir);
69   File[] djeca = dir.listFiles();
70   if(djeca==null) return;
71   for(File f : djeca) {
72    if(f.isFile()) {
73     info.obradiDatoteku(f);
74    } else if(f.isDirectory()) {
75     rekurzivnoListaj(f, info);
76    }
77   }
78  }
79
80  // Use the Class, Luke!
81
82  /**
83   * Razred modelira informacije koje pamtimo
84   * o jednoj ekstenziji.
85   *
86   */
87  static class InfoOEkstenziji {
88   String ekstenzija;
89   int brojPojava;
90   long ukupnaVelicina;
91
92   public InfoOEkstenziji(String ekstenzija) {
93    super();
94    this.ekstenzija = ekstenzija;
95   }
96
97   double prosjek() {
98    if(brojPojava==0) {
99     return 0;
100    }
101    return ukupnaVelicina / brojPojava;
102   }
103  }
104
105  /**
106   * Razred modelira sve informacije koje pamtimo
107   * tijekom analize podstabla.
108   */
109  static class Informacije {
110   int brojPojavaDirektorija;
111   int brojPojavaDatoteka;
112   long ukupnaVelicinaDatoteka;
113   Map<String,InfoOEkstenziji> mapa =
114       new HashMap<String, InfoOEkstenziji>();
```

```
115
116   double prosjekDatoteka() {
117    if(brojPojavaDatoteka==0) {
118     return 0;
119    }
120    return ukupnaVelicinaDatoteka / brojPojavaDatoteka;
121   }
122
123   public void obradiDirektorij(File dir) {
124    brojPojavaDirektorija++;
125   }
126
127   public void obradiDatoteku(File f) {
128    brojPojavaDatoteka++;
129    ukupnaVelicinaDatoteka += f.length();
130
131    String ekst = izracunajEkstenziju(f.getName());
132    if(ekst==null) {
133     return;
134    }
135    InfoOEkstenziji info = mapa.get(ekst);
136    if(info==null) {
137     info = new InfoOEkstenziji(ekst);
138     mapa.put(ekst, info);
139    }
140    info.brojPojava++;
141    info.ukupnaVelicina += f.length();
142   }
143
144   private String izracunajEkstenziju(String name) {
145    int pozicijaTocke = name.lastIndexOf('.');
146    if(pozicijaTocke<0) return null;
147    String ekst = name.substring(pozicijaTocke+1);
148    if(ekst.isEmpty()) return null;
149    return ekst.toUpperCase();
150   }
151
152  }
153 }
154
```

Za potrebe rješavanja problema napisana su dva pomoćna razreda. Razred `InfoOEkstenziji` sadrži statističke podatke vezane uz jednu konkretnu ekstenziju. Razred `Informacije` predstavlja razred koji enkapsulira općenite statističke podatke, čuva kolekciju informacija o pronađenim ekstenzijama (koristi se mapa) te sadrži još dvije metode: `obradiDirektorij` i `obradiDatoteku`. Pozivom svake od njih ažuriraju se dotadašnje prikupljene informacije. Ideja ovog pristupa je omogućiti da se na jednom mjestu izolirano napiše metoda koja će raditi obilazak čitavog podstabla (bez ikakvog razumijevanja o konkretnoj obradi obiđenih elemenata) te da za svaki pronađeni element pozove metodu koja će raditi odgovarajuće ažuriranje. Taj pristup upravo je i ostvaren kroz metode `obradi` (počinje u retku 60) i `rekurzivnoListaj` (počinje u retku 66).

Glavni program odnosno metoda `main` pozove metodu `obradi` kako bi dobila tražene informacije, ispisuje opće statističke podatke, dohvaća i sortira podatke o ekstenzijama i na kraju i njih ispisuje. Za sortiranje je zadužena metoda `Collections.sort` kojoj kao drugi argument predajemo primjerak napisanog komparatora.

Koje su karakteristike prikazanog rješenja? Za početak: radi. To je uvijek vrlo važno. Međutim, kod nije lagano proširiv. Zamislimo da trebamo napisati program tako da u podstablu traži

datoteke prema zadanom kriteriju ili pak da kopira jedno podstablo u drugo podstablo, ili pak da obriše sve exe datoteke. Svim tim programima bile bi zajedničke dvije stvari: svi bi trebali metodu koja zna obilaziti podstablo, i svi bi trebali imati objekt tipa onog kojeg nudi razred Informacije koji sadrži metode odradiDirektorij i obradiDatoteku koje će znati što je točno potrebno napraviti.

Uočimo da su navedeni postupci konceptualno *dva različita postupka*. Dapače, u svim ćemo problemima prvi zadatak rješavati na gotovo identičan način. Da bismo izbjegli ovo dupliciranje koda, rješenje možemo malo preraditi. Umjesto razreda Informacije trebamo jedno općenito sučelje koje će svaka konkretna obrada implementirati. Prikladno sučelje koje nam nudi Java 7 je java.nio.file.FileVisitor<V>. Sučelje je prikazano u nastavku.

Primjer 16.11. Sučelje `FileVisitor`

```
 1 public interface FileVisitor<T> {
 2
 3  FileVisitResult preVisitDirectory(
 4    T dir, BasicFileAttributes attrs
 5  ) throws IOException;
 6
 7  FileVisitResult visitFile(
 8    T file, BasicFileAttributes attrs
 9  ) throws IOException;
10
11  FileVisitResult visitFileFailed(
12    T file, IOException exc
13  ) throws IOException;
14
15  FileVisitResult postVisitDirectory(
16    T dir, IOException exc
17  ) throws IOException;
18 }
19
20 public enum FileVisitResult {
21     CONTINUE,
22     TERMINATE,
23     SKIP_SUBTREE,
24     SKIP_SIBLINGS;
25 }
26
```

Sučelje sadrži četiri metode i svaka od njih vraća element enumeracije FileVisitResult. Ova enumeracija definira četiri vrijednosti: CONTINUE (obilazak se treba nastaviti), TERMINATE (obilazak se treba prekinuti), SKIP_SUBTREE (preskoči obilazak podstabla trenutnog direktorija) te SKIP_SIBLINGS (preskoči obilazak preostalih direktorija koji su u istom direktoriju kao i trenutni; za detalje pogledati službenu dokumentaciju).

Opišimo sada i četiri metode sučelja FileVisitor. Razmišljajući o nekom općenitom algoritmu obilaska podstabla, u svakom trenutku algoritam će pronaći ili datoteku ili direktorij ili će naići na pogrešku prilikom čitanja informacija o datoteci odnosno direktoriju. Ako je pronađeni element direktorij, obilazak će se tipično nastaviti rekurzivno. Sučelje FileVisitor za ovakav algoritam stoga nudi metode opisane u nastavku. Za iscrpno pojašnjenje čitatelj se upućuje na službenu dokumentaciju.

- visitFile

  Metoda za obilazak pozvat će je za svaku pronađenu datoteku. Nastavak obilaska ovisit će o rezultatu koji metoda vrati. Primjerice, ako vrati TERMINATE, cjelokupni obilazak će se prekinuti; ako vrati SKIP_SIBLINGS, metoda za obilazak će preskočiti sve preostale

datoteke koje su smještene u istom roditeljskom direktoriju; ako vrati CONTINUE, obilazak će se nastaviti.

- preVisitDirectory

  Metoda za obilazak će je pozvati kada naiđe na direktorij. Ako metoda vrati CONTINUE, procedura obilaska krenut će u rekurzivni obilazak članova tog direktorija. Ako metoda vrati SKIP_SUBTREE, obilazak direktorija će se preskočiti.

- postVisitDirectory

  Metoda se poziva po izlasku iz obilaska direktorija. Daljnji postupak obilaska ovisi o vraćenoj vrijednosti.

- visitFileFailed

  Metoda se poziva kada nastupi pogreška prilikom obilaska datoteke ili direktorija. Daljnji postupak obilaska ovisi o vraćenoj vrijednosti.

Oslanjajući se na opisano sučelje, sada bismo lagano mogli napisati (ili prilagoditi našu postojeću) implementaciju rekurzivnog obilaska podstabla koja uz referencu na vršni direktorij od kojeg kreće obilazak prima i referencu na objekt koji implementira sučelje FileVisitor; za svaki pronađeni element pozvali bismo odgovarajuću metodu primljenog objekta i nastavili u skladu s dobivenom povratnom vrijednosti. Međutim, to ne trebamo raditi: razred Files već nudi gotovu implementaciju ove metode (dapače porodicu metoda). Radi se o metodama walkFileTree koje se međusobno razlikuju samo po argumentima koje primaju. Sve su te metode upravo općenita implementacija algoritma obilaska stabla i svaka od njih minimalno prima referencu na direktorij od kojeg kreće obilazak te referencu na objekt koji implementira sučelje FileVisitor.

Rješenje prethodnog zadatka koje za obilazak koristi gotove metode prikazano je u nastavku.

Primjer 16.12. Izračun statističkih podataka, bolje

```
 1 package hr.fer.zemris.java.primjeri;
 2
 3 import java.io.IOException;
 4 import java.nio.file.FileVisitResult;
 5 import java.nio.file.FileVisitor;
 6 import java.nio.file.Files;
 7 import java.nio.file.Path;
 8 import java.nio.file.Paths;
 9 import java.nio.file.attribute.BasicFileAttributes;
10 import java.util.ArrayList;
11 import java.util.Collections;
12 import java.util.Comparator;
13 import java.util.HashMap;
14 import java.util.List;
15 import java.util.Map;
16
17 public class Statistika2 {
18
19  public static void main(String[] args) {
20   if(args.length != 1) {
21    System.err.println("Krivi poziv!");
22    System.exit(0);
23   }
24
25   Path dir = Paths.get(args[0]);
26
```

```
27   Analiza analiza = new Analiza();
28   try {
29    Files.walkFileTree(dir, analiza);
30   } catch (IOException e) {
31    System.err.println(e.getMessage());
32    return;
33   }
34   Informacije info = analiza.info;
35
36   System.out.println(
37     "Broj datoteka je: " + info.brojPojavaDatoteka);
38   System.out.println(
39     "Ukupna velicina datoteka je: " +
40     info.ukupnaVelicinaDatoteka);
41   System.out.println(
42     "Prosjecna velicina datoteka je: " +
43     info.prosjekDatoteka());
44   System.out.println(
45     "Broj direktorija je: " + info.brojPojavaDirektorija);
46
47   List<InfoOEkstenziji> lista =
48     new ArrayList<>(info.mapa.values());
49
50   Comparator<InfoOEkstenziji> komparator =
51     new Comparator<InfoOEkstenziji>() {
52    @Override
53    public int compare(InfoOEkstenziji o1,
54                       InfoOEkstenziji o2) {
55     return o1.ekstenzija.compareTo(o2.ekstenzija);
56    }
57   };
58
59   Collections.sort(
60     lista, Collections.reverseOrder(komparator)
61   );
62
63   for(InfoOEkstenziji i : lista) {
64    System.out.println(
65      i.ekstenzija + " " + i.brojPojava + " " +
66      i.ukupnaVelicina + " " + i.prosjek()
67    );
68   }
69
70  }
71
72  private static class Analiza implements FileVisitor<Path> {
73
74   private Informacije info = new Informacije();
75
76   @Override
77   public FileVisitResult postVisitDirectory(Path dir,
78                                             IOException exc)
79     throws IOException {
80    return FileVisitResult.CONTINUE;
81   }
82
83   @Override
84   public FileVisitResult preVisitDirectory(Path dir,
```

```
85          BasicFileAttributes attrs) throws IOException {
86       info.obradiDirektorij(dir);
87       return FileVisitResult.CONTINUE;
88     }
89     @Override
90     public FileVisitResult visitFile(Path file,
91          BasicFileAttributes attrs) throws IOException {
92       info.obradiDatoteku(file);
93       return FileVisitResult.CONTINUE;
94     }
95     public FileVisitResult visitFileFailed(Path file,
96          IOException exc) throws IOException {
97       return FileVisitResult.CONTINUE;
98     };
99   }
100
101  // Use the Class, Luke!
102
103  /**
104   * Razred modelira informacije koje pamtimo
105   * o jednoj ekstenziji.
106   *
107   */
108  static class InfoOEkstenziji {
109   String ekstenzija;
110   int brojPojava;
111   long ukupnaVelicina;
112
113   public InfoOEkstenziji(String ekstenzija) {
114     super();
115     this.ekstenzija = ekstenzija;
116   }
117
118   double prosjek() {
119     if(brojPojava==0) {
120       return 0;
121     }
122     return ukupnaVelicina / brojPojava;
123   }
124  }
125
126  /**
127   * Razred modelira sve informacije koje pamtimo
128   * tijekom analize podstabla.
129   */
130  static class Informacije {
131   int brojPojavaDirektorija;
132   int brojPojavaDatoteka;
133   long ukupnaVelicinaDatoteka;
134   Map<String,InfoOEkstenziji> mapa =
135     new HashMap<String, InfoOEkstenziji>();
136
137   double prosjekDatoteka() {
138     if(brojPojavaDatoteka==0) {
139       return 0;
140     }
141     return ukupnaVelicinaDatoteka / brojPojavaDatoteka;
142   }
```

```
143
144   public void obradiDirektorij(Path dir) {
145    brojPojavaDirektorija++;
146   }
147
148   public void obradiDatoteku(Path f) {
149    brojPojavaDatoteka++;
150
151    long velicina = 0;
152    try {
153     velicina = Files.size(f);
154    } catch (IOException ignorable) {
155    }
156
157    ukupnaVelicinaDatoteka += velicina;
158
159    String ekst = izracunajEkstenziju(
160      f.getFileName().toString()
161    );
162    if(ekst==null) {
163     return;
164    }
165    InfoOEkstenziji info = mapa.get(ekst);
166    if(info==null) {
167     info = new InfoOEkstenziji(ekst);
168     mapa.put(ekst, info);
169    }
170    info.brojPojava++;
171    info.ukupnaVelicina += velicina;
172   }
173
174   private String izracunajEkstenziju(String name) {
175    int pozicijaTocke = name.lastIndexOf('.');
176    if(pozicijaTocke<0) return null;
177    String ekst = name.substring(pozicijaTocke+1);
178    if(ekst.isEmpty()) return null;
179    return ekst.toUpperCase();
180   }
181
182  }
183 }
184
```

Cjelokupni posao prikupljanja statistike zatvorili smo u razred `Analiza` koji implementira sučelje `FileVisitor`. Prilikom inicijalizacije objekta, stvara se prazan objekt s prikupljenim informacijama; svaka od metoda sučelja `FileVisitor` potom ažurira te podatke.

U metodi `main` stvaramo primjerak razreda `Analiza`, predajemo ga kao argument metodi `Files.walkFileTree` i nakon toga dohvaćamo objekt koji sadrži konačne statističke podatke. Iste potom ispisujemo na jednak način kao i u prethodnom rješenju.

Usporedite li na konceptualnoj razini oba prikazana rješenja, razliku ćete naći na samo jednom mjestu: drugo rješenje sadrži implementaciju samo onoga što ga čini različitim od drugih sličnih programa; za sve dijeljene operacije koriste se općenite metode.

Spomenimo i da na raspolaganju imamo i razred `java.nio.file.SimpleFileVisitor` koji je implementacija sučelja `FileVisitor` kod kojeg sve metode vraćaju vrijednost `CONTINUE`. Imamo li obradu kojoj su interesantne samo informacije o pronađenim datotekama, umjesto da pišemo razred koji implementira sučelje `FileVisitor` mogli bismo napisati raz-

red koji nasljeđuje `SimpleFileVisitor` i koji samo nadjačava metodu `visitFile`; sve ostale metode preuzet će podrazumijevane implementacije čime ćemo ukupno pisati najmanju potrebnu količinu koda.

Preostali važniji razredi

Još nam je preostalo pogledati na koji su način modelirani datotečni sustavi i konkretne particije. Razred `FileSystem` predstavlja model datotečnog sustava. Datotečni sustav korisnicima nudi:

- informaciju koji se separator elemenata koristi za izgradnju staze (metoda `String getSeparator();`),
- metodu koja temeljem predanih elemenata gradi stazu (metoda `Path getPath(String first, String... more);`; upravo nju poziva metoda `Paths.get()` koju smo prethodno koristili),
- kolekciju vršnih datotečnih objekata tog datotečnog sustava (metoda `Iterable<Path> getRootDirectories()`),
- kolekciju particija koje pripadaju tom datotečnom sustavu (metoda `Iterable<FileStore> getFileStores()`) te druge.

Razred `FileSystems` nudi statičku metodu `getDefault()` kojom se dohvaća referenca na pretpostavljeni datotečni sustav te još nekoliko naprednijih metoda kojima je na razlini programa moguće stvarati vlastite implementacije datotečnih sustava (što nećemo dalje obrađivati).

Konkretne particije (ono što su u Windowsima particija `C:`, particija `D:` i slično) modelirane su razredom `FileStore`. Primjerci razreda `FileStore` nude nam više metoda od kojih ćemo izdvojiti samo neke.

- Metoda `long getTotalSpace();` vraća ukupnu veličinu particije u oktetima.
- Metoda `long getUnallocatedSpace();` vraća količinu slobodnog prostora na particiji.
- Metoda `long getUsableSpace();` vraća količinu prostora koju na particiji može koristiti naš program.
- Metoda `boolean isReadOnly();` provjerava je li particija dostupna samo za čitanje.

Za svaku stazu (objekt tipa `Path`) pozivom metode `getFileSystem()` moguće je dobiti referencu na datotečni sustav u skladu s kojim je staza izgrađena. Dodatno, pozivom statičke metode `getFileStore(path p)` razreda `Files` moguće je za stazu pribaviti referencu na objekt koji opisuje particiju na kojoj se nalazi datotečni objekt čija je to staza.

Poglavlje 17. Tokovi okteta i znakova

U prethodnom poglavlju upoznali smo se s programskim sučeljem koje nam stoji na raspolaganju za dobivanje informacija o datotečnom sustavu računala te objektima datotečnog sustava, bilo da se radi o particijama, direktorijima, datotekama ili simboličkim linkovima. U ovom poglavlju fokusirat ćemo se na rad s jednom istaknutom vrstom objekata datotečnog sustava: s *datotekama*. U datotečnom sustavu datoteke imaju centralno mjesto: particije omogućavaju njihovo fizičko grupiranje unutar jednog ili pak između više različitih fizičkih naprava za pohranu podataka; direktoriji omogućavaju njihovo logičko grupiranje u hijerarhijske strukture. U konačnici, na dnu svega su upravo datoteke -- spremnici za pohranu svih vrsta informacija s kojima rade naši programi. Te informacije mogu biti svakojake: dokumenti, slike, video, muzika, arhive i još niz drugoga.

Kada govorimo o datotekama, treba uobičajeno razlikujemo dvije vrste datoteka:

- *binarne datoteke* čija je namjena pohrana podataka koji nisu nužno samo tekst; prilikom izrade ovakvih datoteka podatci se zapisuju u skladu sa specifikacijom koja opisuje format datoteke (primjerice, specifikacije MP3, ZIP, DOC, AVI);

- *tekstovne datoteke* čija je namjena pohrana podataka uporabom teksta; pojam tekstovna datoteka ne označava samo datoteke čija je namjena pohrana teksta (eseja, pjesme, sastavka i slično); pojam se odnosi na sve datoteke koje uporabom teksta pohranjuju bilo kakve informacije; primjerice, datoteka izrađena u skladu sa specifikacijom SVG je tekstovna datoteka i sadrži naputak kako nacrtati sliku: konceptualno, informacija koju ona predstavlja je slika; XML dokumenti su tekstovne datoteke; HTML dokumenti su tekstovne datoteke koje specificiraju sadržaj ali i strukturu web-stranice.

Prilikom razmatranja datoteka treba stoga razlikovati tekstovne datoteke od binarnih. Tekstovne datoteke izgrađene su od simbola (odnosno znakova); konceptualna minimalna jedinica podatka u takvoj datoteci je jedan znak koji se ne mora nužno preslikati u jedan oktet pri pohrani datoteke na disk (potreban broj okteta bit će određen kodnom stranicom koja se koristi za preslikavanje znakova u oktete). Kod binarnih datoteka situacija je jasnija: minimalna jedinica podatka je uvijek oktet.

Osim ove razlike, treba razlikovati datoteke i po načinu na koji je zamišljeno njihovo korištenje. Ovdje opet možemo razmatrati dvije vrste datoteka.

- *Slijedne datoteke* su datoteke čiji se sadržaj treba čitati od prvog elementa prema posljednjem. Kod slijednih datoteka operacije poput pomaka na lokaciju 375. okteta pa čitanja ili pisanja od tog mjesta nemaju smisla jer informacija koja tu piše (ili se zapisuje) ovisi o svemu što piše ispred a moguće i iza tog mjesta. Tipični primjeri su datoteke čiji je sadržaj upravo tekst (primjerice, neki dopis) te datoteke koje kroz tekst opisuju druge vrste dokumenata (primjerice, SVG, HTML); postoje naravno i binarne datoteke ovog tipa.

- *Direktne datoteke* su datoteke čiji je sadržaj moguće čitati počev od različitih lokacija. To su uobičajeno datoteke koje pohranjuju zapise fiksne (iste) veličine pa se direktnim pomakom na određene lokacije dolazi do podataka koji se mogu jasno interpretirati iako nisu pročitani svi prethodni podatci. U direktne datoteke mogli bismo ubrojati i datoteke koje na određenom mjestu sadrže "kazalo" koje se mora pročitati i temeljem kojega se potom može direktno "skakati" na pojedine lokacije u datoteci. Direktne datoteke su najčešće binarne datoteke.

Pažljivijim čitanjem prethodnih podjela lagano će se doći do zaključka da se u praksi najčešće susreću tri tipa datoteka:

- slijedne binarne datoteke,

- slijedne tekstovne datoteke te

- direktne binarne datoteke.

Programska sučelja dostupna u Javi relativno dobro prate takvu podjelu: imamo na raspolaganju biblioteke koje nude rad s tokovima okteta, biblioteke koje nude rad s tokovima znakova te biblioteke koje nude rad s direktnim binarnim datotekama. Naš pregled dostupnoga stoga ćemo obaviti upravo tim redosljedom.

Tokovi okteta: motivacija

Rad sa slijednim binarnim datotekama (odnosno bilo kakvim slijednim binarnim izvorom ili ponorom okteta) apstrahiran je uporabom pojma *tok okteta*. Sa strane korisnika, tok može biti *ulazni* ili *izlazni*. Ulazni tokovi okteta modeliraju binarne slijedne izvore (iz kojih korisnik može čitati redom dostupne oktete, sve dok ne pročita i posljednji oktet, tj. dok ne *iscrpi* izvor). Izlazni tokovi okteta modeliraju slijedne binarne ponore okteta u koje korisnik redom zapisuje željeni slijed binarnih podataka (okteta). Primjer jednostavnog modela ulaznog i izlaznog toka prikazan je u nastavku.

```
interface UlazniTokOkteta {
  public byte procitajOktet() throws IOException;
  public void close() throws IOException;
  public int preskoci(int n) throws IOException;
}

interface IzlazniTokOkteta {
  public void zapisiOktet(byte oktet) throws IOException;
  public void close() throws IOException;
  public void flush() throws IOException;
}
```

U oba prikazana sučelja, sve metode deklariraju da u slučaju pogreške izazivaju iznimku `IOException`.

Ulazni tokovi okteta

Ulazni tok modelirali smo sučeljem `UlazniTokOkteta`. Ulazni tok definira samo tri metode: metoda `procitajOktet` iz toka dohvaća (i uklanja) prvi nepročitani oktet. Kako je ovo model slijednog pristupa, sadržaj se čita oktet po oktet, od prvog do posljednjeg raspoloživog okteta. Kada su svi oktet pročitani, čitanje više nije moguće i metoda generira `EOFException`. Metoda `close` služi zatvaranju toka čime se otpuštaju svi resursi koje je tok zauzimao. Metoda `preskoci` preskače zadani niz okteta (ili manje ako ih toliko više nema) i vraća broj okteta koji su doista preskočeni. Konceptualno, ponaša se kao da je zatraženo čitanje `n` okteta koji se pročitaju i potom odbace. Prikažimo kao primjer dvije implementacije ovakvog toka. Prvi tok će podatke čitati iz polja koje mu se preda prilikom poziva konstruktora; drugi tok će svakim pozivom metode vratiti jedan slučajno generirani broj. Ovaj tok nikada neće doći do kraja pa nikada neće niti izazvati `EOFException`.

Primjer 17.1. Primjer ulaznih tokova okteta.

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Model ulaznog toka okteta. Ulazni tok je tok iz
 7  * kojeg klijenti mogu čitati podatke, slijedno,
 8  * počev od prvog pa do posljednjeg raspoloživog
 9  * okteta.
10  *
11  * @author marcupic
```

```
 */
public interface UlazniTokOkteta {

 /**
  * Metoda dohvaća sljedeći nepročitani oktet i
  * vraća ga. Ako iz toka više nije moguće pročitati
  * niti jedan oktet jer su svi podatci iscrpljeni,
  * metoda izaziva iznimku {@link IOException}.
  *
  * @return sljedeći nepročitani oktet
  * @throws IOException ako više nema podataka
  */
 public byte procitajOktet() throws IOException;

 /**
  * Metoda zatvara trenutni tok i otpušta sve njegove
  * resurse. Tok koji je zatvoren ne može se opet
  * zatvoriti (u tom slučaju bit će izazvana iznimka
  * {@link IOException}).
  *
  * @throws IOException ako je tok već zatvoren
  */
 public void close() throws IOException;

 /**
  * Metoda preskače zadan broj okteta. Efekt je isti
  * kao da je <code>n</code> puta pozvana metoda
  * {@link #procitajOktet()} čiji se rezultati potom
  * zanemare. Ako se tok prije iscrpi, broj preskočenih
  * okteta bit će manji.
  *
  * @param n željeni broj okteta koje treba preskočiti
  * @return stvarno preskočeni broj okteta
  * @throws IOException ako nastupi pogreška prilikom
  *         preskakanja
  * @throws IllegalArgumentException ako je <code>n</code>
  *         negativan
  */
 public int preskoci(int n) throws IOException;
}
```

```
package book.java.iostreams.motivation;

import java.io.EOFException;
import java.io.IOException;

/**
 * Ulazni tok koji čita podatke iz polja okteta koje dobiva
 * prilikom konstrukcije. Čitanje kreće od prvog okteta
 * u predanom polju i napreduje prema posljednjem oktetu.
 *
 * @author marcupic
 */
public class UlazniTokOktetaIzPolja
  implements UlazniTokOkteta {

 // Spremnik okteta iz kojeg se čitaju podatci
```

```
17  private byte[] polje;
18  // Pozicija okteta koji će biti sljedeći vraćen
19  private int trenutnaPozicija;
20  // Jesmo li pročitali sve oktete?
21  private boolean gotovo;
22
23  /**
24   * Konstruktor. Prima se polje okteta koje će tok čitati.
25   * Predano polje ne smije biti <code>null</code>.
26   *
27   * @param polje polje okteta iz kojeg će se čitati podatci
28   * @throws IllegalArgumentException ako je predan
29   *         <code>null</code>
30   */
31  public UlazniTokOktetaIzPolja(byte[] polje) {
32   super();
33   this.polje = polje;
34   if(this.polje==null) {
35    throw new IllegalArgumentException(
36      "Polje ne smije biti null.");
37   }
38   this.trenutnaPozicija = 0;
39   this.gotovo = false;
40  }
41
42  @Override
43  public byte procitajOktet() throws IOException {
44   if(trenutnaPozicija >= polje.length) {
45    gotovo = true;
46   }
47   if(gotovo) {
48    throw new EOFException("Sadržaj je pročitan.");
49   }
50   byte oktet = polje[trenutnaPozicija];
51   trenutnaPozicija++;
52   return oktet;
53  }
54
55  @Override
56  public void close() throws IOException {
57   if(polje==null) {
58    throw new IOException("Tok je već zatvoren.");
59   }
60   // Otpusti resurse...
61   polje = null;
62   gotovo = true;
63  }
64
65  @Override
66  public int preskoci(int n) throws IOException {
67   for(int i = 0; i < n; i++) {
68    try {
69     procitajOktet();
70    } catch(EOFException ex) {
71     return i;
72    }
73   }
74   return n;
```

```
 }
}

```

```
package book.java.iostreams.motivation;

import java.io.EOFException;
import java.io.IOException;
import java.util.Random;

/**
 * Implementacija ulaznog toka koji predstavlja
 * neograničen izvor slučajno generiranih
 * okteta.
 *
 * @author marcupic
 *
 */
public class UlazniTokOktetaSlucajni
  implements UlazniTokOkteta {

 // Zastavica koja govori je li tok zatvoren
 private boolean zatvoreno;
 // Privatni generator slučajnih brojeva
 private Random generator;

 /**
  * Konstruktor.
  */
 public UlazniTokOktetaSlucajni() {
  generator = new Random();
 }

 @Override
 public byte procitajOktet() throws IOException {
  if(zatvoreno) {
   throw new EOFException(
     "Čitanje zatvorenog toka nije moguće.");
  }
  return (byte)(generator.nextInt() & 0xFF);
 }

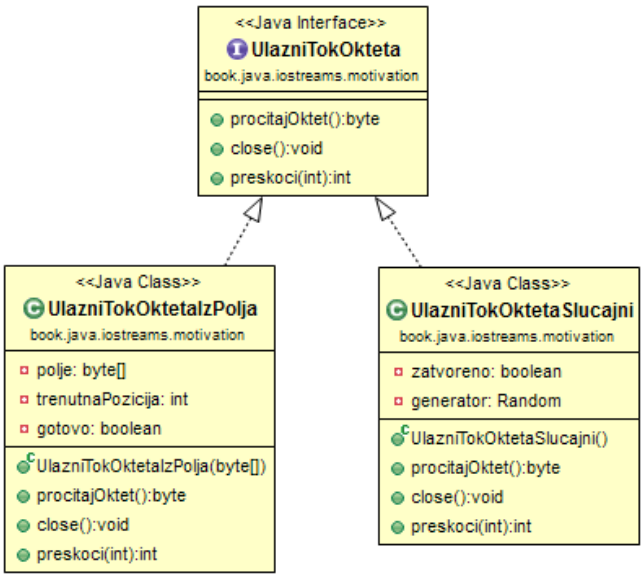
 @Override
 public void close() throws IOException {
  if(zatvoreno) {
   throw new IOException("Tok je već zatvoren.");
  }
  // Otpusti resurse...
  generator = null;
  zatvoreno = true;
 }

 @Override
 public int preskoci(int n) throws IOException {
  // ne treba ništa raditi; rezultate ionako
  // generiramo slučajno.
  return n;
 }
```

```
55 }
56
```

Strukturni dijagram ovog rješenja prikazuje slika 17.1.

Slika 17.1. Primjer ulaznih tokova okteta.



Proučite pažljivije dane implementacije i uočite na koji su način ostvarene pojedine metode.

Uporabu danih implementacija ilustrira primjer 17.2.

Primjer 17.2. Primjer uporabe ulaznih tokova okteta.

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.EOFException;
 4 import java.io.IOException;
 5
 6 /**
 7  * Primjer uporabe ulaznih tokova.
 8  *
 9  * @author marcupic
10  */
11 public class Primjer1 {
12
13  /**
14   * Metoda od koje počinje izvođenje programa.
15   *
16   * @param args ne koristi se
17   * @throws IOException pri pogrešci u radu s tokovima
18   */
19  public static void main(String[] args) throws IOException {
20   UlazniTokOkteta utok1 = new UlazniTokOktetaIzPolja(
21    new byte[] {1, -5, 28, -45, 37}
22   );
23
24   UlazniTokOkteta utok2 = new UlazniTokOktetaSlucajni();
25
26   System.out.println("Ispis iz prvog toka:");
```

```
27    ispisi(utok1, 8);
28    System.out.println();
29    utok1.close();
30
31    System.out.println("Ispis iz drugog toka:");
32    ispisi(utok2, 8);
33    System.out.println();
34    utok2.close();
35   }
36
37   /**
38    * Pomoćna metoda koja prima ulazni binarni tok te broj
39    * okteta koje je potrebno iz toka pročitati. Metoda čita
40    * toliko okteta i formatirano ih ispisuje na zaslon. Ako
41    * tok nema toliko okteta, metoda će ih pročitati i
42    * ispisati onoliko koliko ih tok sadrži te potom ispisati
43    * dodatnu poruku.
44    *
45    * @param utok ulazni tok iz kojeg se čitaju okteti
46    * @param n broj okteta koje je potrebno pročitati
47    * @throws IOException ako dođe do pogreške pri čitanju
48    */
49   private static void ispisi(UlazniTokOkteta utok, int n)
50    throws IOException {
51
52    try {
53     for(int i = 0; i < n; i++) {
54      byte podatak = utok.procitajOktet();
55      System.out.printf("%d. podatak: %d%n", i+1, podatak);
56     }
57    } catch(EOFException ex) {
58     System.out.println("Tok više nema podataka.");
59    }
60   }
61  }
62
```

U metodi `main` stvaramo dva ulazna binarna toka. Tok `utok1` je primjerak razreda `UlazniTokOktetaIzPolja` kojemu u konstruktoru dajemo referencu na polje od pet okteta koje će tok vraćati kao svoj sadržaj. Tok `utok2` je primjerak razreda `UlazniTokOktetaSlucajni` i on predstavlja tok okteta neograničene duljine koji pri svakom pozivu vraća slučajno generirani oktet.

Pripremljena je i pomoćna metoda `ispisi(utok, n)` koja kao argumente dobiva referencu na ulazni binarni tok okteta (prvi argument) te broj okteta koje iz toka treba pročitati (drugi argument). Metoda potom čita toliko okteta i ispisuje ih. U slučaju da tok nema dovoljno okteta, metoda nakon iscrpljivanja toka prestaje s čitanjem ali ispisuje odgovarajuću poruku. Metoda `main` poziva ovu metodu dva puta: najprije nad tokom `utok1` a potom nad tokom `utok2`. Primjer ispisa koji ćemo dobiti prikazan je u nastavku.

```
Ispis iz prvog toka:
1. podatak: 1
2. podatak: -5
3. podatak: 28
4. podatak: -45
5. podatak: 37
Tok više nema podataka.

Ispis iz drugog toka:
```

```
1. podatak: -76
2. podatak: -127
3. podatak: -50
4. podatak: -114
5. podatak: -113
6. podatak: 95
7. podatak: 87
8. podatak: 109
```

Obratite pažnju na redoslijed okteta koje je prvi tok isporučio klijentu (metodi `ispisi`) i kako smo ga inicijalizirali.

Uočavate li da smo u ovom programu iskoristili i jedan oblikovni obrazac? Primijetite: imamo samo jednu (zajedničku) metodu koja jednako uspješno čita podatke iz pripremljenog polja okteta te generira oktete koristeći generator slučajnih brojeva - a bez da zna išta o tome. Radi se, naravno, o oblikovnom obrascu *Strategija*: apstraktno ponašanje toka modelirano je sučeljem `UlazniTokOkteta` (sudionik obrasca: *Apstraktna strategija*) i potom su napravljene dvije različite implementacije, razredi `UlazniTokOktetaIzPolja` i `UlazniTokOktetaSlucajni` (sudionici obrasca: *Konkretne strategije*). Potom je pripremljena općenita metoda `ispisi` (sudionik obrasca: *Kontekst*) koja dobiva referencu na strategiju koju treba koristiti.

Zahvaljujući ovakvoj organizaciji koda, sada je lagano zamisliti da smo pripremili još niz drugih konkretnih strategija koje podatke dohvaćaju iz različitih izvora; primjerice, mogli bismo imati jedan razred koji implementira sučelje `UlazniTokOkteta` i koji podatke dohvaća iz datoteke pohranjene na lokalnom datotečnom sustavu; mogli bismo imati drugi razred koji implementira isto sučelje i podatke dohvaća preko Interneta; treći koji podatke preuzima od lokalnog procesa koji je pokrenut na istom računalu i koji s našim procesom komunicira putem cjevovoda ili nekog drugog mehanizma za međuprocesnu komunikaciju koje danas nude moderni operacijski sustavi; itd.

Sve opisane implementacije, kao i niz drugih kojih se nismo dosjetili mogle bi biti korištene od istog klijenta (naše metode `ispisi`) potpuno transparentno. Štoviše, ako tri mjeseca nakon što smo napisali čitav program uočimo da nam je potreban još jedan (novi) izvor podataka, bit će dovoljno samo napisati novi razred koji implementira sučelje `UlazniTokOkteta` i koji podatke dohvaća iz tog izvora. Napisani klijentski kod automatski će raditi i s tom implementacijom što pokazuje da imamo rješenje koje je u skladu s *Načelom nadogradnje bez promjene*.

Mala korekcija modela

Sučelje `UlazniTokOkteta` vrlo je jednostavno i semantički čisto. U praksi, uporaba takvog sučelja za programera može biti nepraktična iz dva razloga koja ćemo u nastavku istaknuti.

Pogledajte što kao povratni tip vraća metoda `procitajOktet`: metodu smo deklarirali tako da vraća `byte` što je razumljivo s obzirom da je njezina zadaća dohvat sljedećeg raspoloživog okteta. Zbog te odluke, tok nema jednostavnog načina da nas obavijesti da je iscrpljen. Stoga će u slučaju da pozovemo čitanje okteta iz iscrpljenog toka isti izazvati iznimku `EOFException` što će nas prisiliti da pišemo dodatne konstrukte koji se bave hvatanjem i obradom iznimaka. Kako je za očekivati da ćemo pri čitanju svakog konačnog ulaznog toka podataka u jednom trenutku doći do njegova kraja, mogli bismo argumentirati da se tu ne radi o iznimnoj situaciji već o normalnom i očekivanom stanju pa za to ne želimo koristiti mehanizam iznimaka.

Dva su tipična načina kako bismo mogli modificirati sučelje `UlazniTokOkteta` da bismo izbjegli dojavu iscrpljivanja izvora uporabom iznimaka.

1. Mogli bismo u sučelje dodati novu metodu: `boolean imaPodataka()` koju bismo morali pozivati prije svakog čitanja da provjerimo postoji li još barem jedan oktet ili je tok izvor iscrpljen. U tom slučaju, tipičan način uporabe prikazan je u nastavku.

   ```
   UlazniTokOkteta utok = ...;
   ```

```
while(utok.imaPodataka()) {
  byte oktet = utok.procitaj();
  //obradi oktet
}
```

Opisano rješenje semantički je vrlo čisto. U slučaju da korisnik pozove metodu `procitajOktet` nakon što metoda `imaPodataka` dojavi da podataka više nema, odgovor bi bio isti kao i prije: `procitajOktet` bi izazvala iznimku `EOFException`. Razlika u odnosu na prvotno rješenje je da korisnik sada ima na raspolaganju API kojim može dobiti potrebne informacije i bez izazivanja iznimke; kvalitetno napisan kod ne mora nikada doći u situaciju da se ta iznimka izazove. Nezgodna karakteristika ovog rješenja je da čitanje svakog okteta zahtjeva pozivanje dviju metoda: najprije metodu `imaPodataka` pa potom metodu `procitajOktet`. U programu koji obrađuje veliku količinu okteta to može predstavljati neprihvatljivu neučinkovitost i dovesti do značajnog usporenja rada programa.

2. Alternativno rješenje je dopustiti metodi `procitajOktet` da preko povratne vrijednosti pozivatelju dojavi je li izvor iscrpljen ili je vraćen pročitani oktet. Kako oktet može poprimiti bilo koju vrijednost od -128 do +127 (ako 8 bitova tumačimo u skadu s dvojnim komplementom) i kako je to upravo cijeli raspon definiran za tip `byte` u Javinoj specifikaciji tipova, da bismo postigli željeno ponašanje morat ćemo promijeniti povratni tip metode `procitajOktet` u neki koji obuhvaća veći raspon. Moguće rješenje je metodu deklarirati tako da umjesto podatka tipa `byte` vraća podatak tipa `int` i da ista u slučaju iscrpljivanja izvora pozivatelju vrati neku vrijednost koja nije iz raspona koje pokriva podatak tipa `byte` (primjerice, 25 miliona). U tom slučaju, pozivatelj poziva samo metodu `procitajOktet`, povratnu vrijednost pohranjuje u pomoćnu varijablu šireg tipa i jednostavnom naredbom `if` provjerava je li vraćena vrijednost ona "specijalna" koja označava da nije vraćen novi podatak već da je tok iscrpljen te ako nije, onda vraćeni podatak može ukalupiti u tip `byte` jer se tada stvarno radi o vraćenom oktetu.

   U praksi, metoda `procitajOktet` se modificira tako da, u slučaju da je uspjela pročitati novi oktet, vrati njegovu vrijednost kao `int` ali dobiven interpretacijom bitova okteta u zapisu pozitivnih brojeva (pa će vraćene vrijednosti biti od 0 do 255) te vratiti vrijednosti -1 ako je tok iscrpljen pa metoda nije mogla pročitati novi podatak. U tom slučaju, tipična uporaba prikazana je u nastavku.

   ```
   UlazniTokOkteta utok = ...;
   while(true) {
     int rezultat = utok.procitaj();
     if(rezultat==-1) {
       break;
     }
     byte oktet = (byte)rezultat;
     //obradi oktet
   }
   ```

   Negativna strana ovog pristupa je zlouporaba povratne vrijednosti: metoda je semantički manje jasna a interpretacija rezultata ovisi o vrijednosti rezultata. Zbog veće učinkovitosti, ovaj pristup čest je izbor u odnosu na pristup s dvije metode.

Drugi razlog nepraktičnosti izvornog sučelja je činjenica da puno puta programer iz toka ne treba samo jedan oktet već ih treba više pa svaki puta mora pisati petlje kojima se obavlja to čitanje. Kako bi se korisnicima ponudila mogućnost uporabe najprikladnijeg API-ja, sučelje ćemo proširiti s još jednom metodom za čitanje. Takvo sučelje u kojem smo prilagodili povratni tip metode `procitajOktet` i u koje smo dodali drugu inačicu metode `procitajOktet` prikazano je u nastavku.

```
interface UlazniTokOkteta {
  public int procitajOktet() throws IOException;
  public int procitajOktet(byte[] spremnik, int odmak, int broj)
```

```
                throws IOException;
  public void close() throws IOException;
  public int preskoci(int n) throws IOException;
}
```

Dodatna inačica metode `procitajOktet` prima tri argumenta: spremnik okteta koji je pripremio korisnik i u koji je potrebno pohraniti pročitane oktete iz toka (polje), poziciju u predanom polju od koje će se pročitani okteti popunjavati (popunjavanje polja ne treba nužno krenuti od pozicije 0) te maksimalni broj okteta koje je potrebno pročitati iz toka i upisati ih u polje; ako tok pri pozivu nije već iscrpljen, metoda bi morala iz toka pročitati minimalno 1 oktet odnosno maksimalno onoliko okteta koliko je određeno trećim argumentom. Koliko će ih točno pročitati, to će biti implementacijski detalj samog toka. Metoda će kroz povratnu vrijednost vratiti *broj* okteta koje je pročitala i spremila u predano polje odnosno -1 ako je tok već pri pozivu metode bio iscrpljen pa iz njega nije bilo moguće pročitati niti jedan oktet. Uz ovakav dogovor, metoda nikada neće vratiti vrijednost 0.

Prije daljnje rasprave, dogovorimo se još o ponašanju metode `close`. Pretpostavimo da je ta metoda *idempotentna* odnosno da višestruki pozivi te metode nemaju nikakve dodatne posljedice u odnosu na samo jedan poziv (drugim riječima, tok koji je zatvoren, je zatvoren; novi pokušaji zatvaranja će se u implementaciji jednostavno zanemarivati).

Uz prethodne dogovoreno, došlo je vrijeme da razmislimo o razredima koji implementiraju ovo sučelje. Neki tokovi su primarno takvog karaktera da podatke mogu generirati isključivo jedan po jedan. Svi razredi koji predstavljaju implementacije takvih tokova imat će prirodnu implementaciju metode `procitajOktet()` a metodu `procitajOktet(spremnik, odmak, broj)` će implementirati višekratnim pozivanjem metode `procitajOktet()` u nekoj petlji. S druge strane, neke implementacije doista mogu efikasnije odraditi čitanje većeg broja okteta u odnosnu na čitanje pojedinačnih okteta - lijep primjer je čitanje iz datoteke koje je učinkovitije ako se radi u većim blokovima podataka. Naš API htjeli bismo tako oblikovati da implementacije koje prirodno rade s pojedinačnim oktetima ne silimo da dupliciraju kod koji obavlja čitanje u polje, a s druge strane, htjeli bismo omogućiti implementacijama koje čitanje u polje mogu napraviti učinkovitije, da to naprave.

Nadalje, kod nekih implementacija tokova, tok neće koristiti nikakve resurse pa će kod takvih tokova metoda `close` moći biti prazna. Htjeli bismo omogućiti implementacijama koje trebaju tu metodu da je i ponude, a ostale implementacije da je uopće ne spominju u kodu (čime se smanjuje broj linija koda koje je potrebno napisati i povećava njegova čitljivost).

Na koji bismo način opisane želje mogli ostvariti? Objektno orijentirana paradigma nudi rješenje: umjesto da ulazni tok modeliramo sučeljem, modelirajmo ga apstraktnim razredom. Metode koje ne znamo implementirati, ostavit ćemo apstraktnima a za metode koje možemo ostvariti uporabom apstraktnih primitiva ponudit ćemo implementaciju. Zahvaljujući mehanizmima nasljeđivanja i dinamičkog polimorfizma, razredi koji mogu ponuditi učinkovitije implementacije metoda moći će nadjačati osnovne implementacije ponuđene u našem vršnom razredu.

Novi model ulaznog toka podataka koji je ovaj puta ostvaren kroz apstraktni razred prikazan je u primjeru 17.3.

Primjer 17.3. Ulazni tok modeliran apstraktnim razredom.

```
1 package book.java.iostreams.motivation;
2
3 import java.io.IOException;
4
5 public abstract class UlazniTokOkteta {
6
7  public abstract int procitajOktet() throws IOException;
8
```

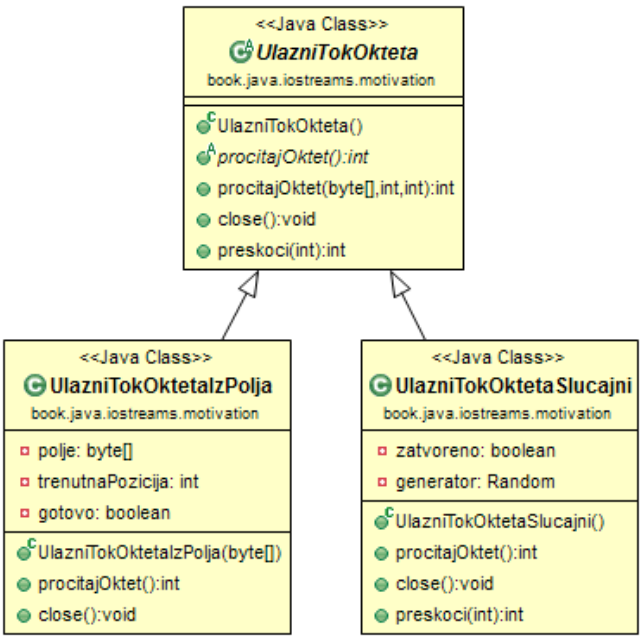
```
 9  public int procitajOktet(byte[] spremnik, int odmak,
10   int broj) throws IOException {
11
12   if(odmak < 0 || odmak >= spremnik.length) {
13    throw new IllegalArgumentException(
14     "Odmak nije legalan."
15    );
16   }
17   if(broj < 1 || odmak+broj > spremnik.length) {
18    throw new IllegalArgumentException(
19     "Kombinacija odmak/broj nije legalna."
20    );
21   }
22   for(int i = 0; i < broj; i++) {
23    int podatak = procitajOktet();
24    if(podatak==-1) {
25     return i==0 ? -1 : i;
26    }
27    spremnik[odmak+i] = (byte)podatak;
28   }
29   return broj;
30  }
31
32  public void close() throws IOException {
33  }
34
35  public int preskoci(int n) throws IOException {
36   for(int i = 0; i < n; i++) {
37    if(-1 == procitajOktet()) {
38     return i;
39    }
40   }
41   return n;
42  }
43 }
44
```

Prikazani razred apstraktan je samo zbog metode `int procitajOktet()` (redak 7). Sve ostale metode imaju svoje implementacije. Tako je metoda `procitajOktet(spremnik, odmak, broj)` ostvarena pozivima apstraktne metode `int procitajOktet()`, metoda `close` ima praznu implementaciju a metoda `preskoci` također je ostvarena pozivanjem apstraktne metode `int procitajOktet()` i zanemarivanjem rezultata.

Uz ovakav model ulaznog toka okteta razrede `UlazniTokOktetaIzPolja` i `UlazniTokOktetaSlucajni` možemo napisati nešto jednostavnije. U odnosu na prethodno dani model potrebno je promijeniti deklaraciju razreda (`implements UlazniTokOkteta` u `extends UlazniTokOkteta`). Metodu koja podatke čita u polje nije potrebno nadodavati u ove implementacije - one će preuzeti ponuđenu implementaciju koju smo dali u apstraktnom razredu `UlazniTokOkteta` i koja je za naše potrebe zadovoljavajuća.

Time ćemo doći do sljedeće strukture razreda (slika 17.2). Čitatelju se ostavlja za vježbu da napiše i isproba ovu prilagodbu.

Slika 17.2. Primjer ulaznih tokova okteta.



Prilagodba demonstracijskog programa danog u primjeru 17.2 tako da radi s ovim razredima prikazana je u primjeru 17.4. Pokrenite ga i uvjerite se da je izlaz programa i dalje u skladu s očekivanim.

Primjer 17.4. Primjer uporabe ulaznih tokova okteta.

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 public class Primjer1 {
 6
 7  public static void main(String[] args) throws IOException {
 8   UlazniTokOkteta utok1 = new UlazniTokOktetaIzPolja(
 9    new byte[] {1, -5, 28, -45, 37}
10   );
11
12   UlazniTokOkteta utok2 = new UlazniTokOktetaSlucajni();
13
14   System.out.println("Ispis iz prvog toka:");
15   ispisi(utok1, 8);
16   System.out.println();
17   utok1.close();
18
19   System.out.println("Ispis iz drugog toka:");
20   ispisi(utok2, 8);
21   System.out.println();
22   utok2.close();
23  }
24
25  private static void ispisi(UlazniTokOkteta utok, int n)
26   throws IOException {
27
28   for(int i = 0; i < n; i++) {
29    int podatak = utok.procitajOktet();
```

```
30     if(podatak==-1) {
31      System.out.println("Tok više nema podataka.");
32      break;
33     }
34     System.out.printf("%d. podatak: %d%n", i+1, podatak);
35    }
36   }
37 }
38
```

Uočite da je u odnosu na izvorni primjer došlo do izmjene u klijentskom kodu zbog načina na koji smo modificirali ponašanje metode `procitajOktet` tako da više ne izaziva iznimku `EOFException`.



Što smo do sada napravili

1. Definirali smo model ulaznog toka okteta kroz sučelje.
2. To smo sučelje promijenili tako što smo metodi `procitajOktet()` proširili tip vrijednosti koju vraća kako bismo preko povratne vrijednosti mogli dojavljivati i informaciju o kraju toka i pročitani podatak.
3. Sučelje smo proširili i novom inačicom metode `procitajOktet` koja prima referencu na polje i koja može pročitati više okteta u jednom pozivu. Motivacija je bila otkloniti od programera potrebu da višekratno sam piše takav kod te omogućiti tokovima koji učinkovito mogu generirati veću količinu okteta da to učine.
4. Kako implementatori razreda koji su ulazni tokovi okteta ne bi morali kopirati kod, umjesto definicije ulaznog toka sučeljem, isti smo modelirali apstraktnim razredom. U tom razredu definirali smo samo jednu apstraktnu metodu: onu koja dohvaća jedan oktet a za sve ostale metode ponudili smo gotove implementacije koje svoju funkcionalnost ostvaruju pozivanjem te metode.

Nadogradnja funkcionalnosti

Ostalo nam je za pogledati još jedan vrlo praktičan problem te način njegova rješavanja. Ilustrirat ćemo to na dva primjera.

Primjer 1. Pretpostavite da ste otvorili ulazni tok okteta koji vraća oktete koji su kriptirani. Za potrebe ovog primjera pretpostavit ćemo da je korišten jednostavan algoritam kriptiranja koji sve bitove u binarnom prikazu okteta invertira (pa tako od broja 0 dobijemo broj 255; od broja 15 dobijemo broj 240 i slično; okteti {1, -5, 28, -45, 37} time se preslikavaju u {-2, 4, -29, 44, -38}). Tok bismo htjeli koristiti s postojećim klijentima bez potrebe da mijenjamo klijente. U našem konkretnom primjeru, klijent je metoda `ispisi`. Njoj bismo htjeli predati referencu na takav tok a ona bi opet korektno ispisala dekriptirane oktete.

Primjer 2. Pretpostavimo da raspolažemo implementacijom ulaznog toka okteta koja čitanje jednog okteta (za to predviđenom metodom) obavlja za 2 milisekunde, a čitanje 100 okteta (za to predviđenom metodom kojoj predajemo polje) obavlja u 5 milisekundi. Očito, čitamo li 100 okteta prvom metodom oktet po oktet, bit ćemo gotovi za 100x2 = 200 milisekundi što je 40 puta dulje no da smo unaprijed zauzeli pomoćni spremnik od 100 okteta i pozvali čitanje s tim spremnikom. Sljedeća pretpostavka: programer piše relativno kompleksan kod u kojem na više mjesta obavlja čitanje i to jedan po jedan oktet - čitanje više okteta mu je nepraktično jer ostatak onda mora nekako dalje slati kroz kod na druga mjesta gdje su podatci potrebni (dok ovako može slati samo referencu na tok). U ovom slučaju htjeli bismo postići da čitanje pojedinačnih okteta na neki način koristi činjenicu da se čitanje više okteta odjednom može obaviti učinkovitije.

Što je zajedničko ovim primjerima? U oba slučaja imamo ulazni tok okteta koji klijenti mogu koristiti ali na nezadovoljavajući način. U prvom primjeru tok isporučuje kriptirane oktete a mi

bismo htjeli dobiti izvorne; u drugom slučaju želimo koristiti pojedinačno čitanje okteta jer nam je to praktično za problem koji riješavamo ali to kod toka s kojim radimo nije učinkovito. Bolje bi bilo čitati više okteta u pomoćni spremnik no nama je to nepraktično.

Oba problema možemo riješiti na jednak način. Napisat ćemo nove razrede koji će korigirati uočene "defekte" postojećih, i to na način da budu primjenjivi na bilo koju konkretnu implementaciju ulaznog toka okteta. Ako imamo tri razreda koji učinkovitije čitaju više okteta, ne želimo pisati tri nova razreda od kojih će svaki "popraviti" jednog od njih. Želimo jedan razred koji obavlja tu vrstu popravka. Ovaj posljednji zahtjev već jednim dijelom uvjetuje novi razred: njega ne možemo naslijediti iz ovih razreda jer Java ne podržava višestruko nasljeđivanje (niti bi to imalo smisla). Umjesto toga, novi će razred morati kroz konstruktor primiti referencu na primjerak razreda čije ponašanje treba korigirati i tu će referencu zapamtiti kroz člansku varijablu. Za potrebe isporuke dekriptiranih podataka postupit ćemo na jednak način: napisat ćemo novi razred koji će kroz konstruktor primiti referencu na primjerak razreda koji isporučuje kriptirane podatke.

Sljedeće pitanje je: "imamo li kakvih zahtjeva na sučelje novih razreda"? U činjenici da primjerke novih razreda moramo moći transparentno podmetnuti klijentima postojećih ulaznih tokova okteta nalazi se odgovor: naši novi razredi prema klijentima se moraju ponašati kao ulazni tokovi okteta što znači da ih moramo naslijediti iz osnovnog razreda `UlazniTokOkteta`.

U glavnom dijelu programa stvorit ćemo primjerak izvornog razreda nakon čega ćemo stvoriti i primjerak našeg novog razreda kojemu ćemo u kontruktor poslati referencu na stvoreni primjerak izvornog razreda. Klijentu ćemo poslati referencu na ovaj posljednji (primjerak novog razreda) i on će pozivati njegove metode. U njegovim metodama podatke ćemo čitati od izvornog primjerka, po potrebi ih modificirati i potom vratiti klijentu. U određenom smislu, primjerci takvih novih razreda ponašaju se kao transparentni posrednici između klijentskog koda i (iz nekog razloga neodgovarajućih) implementacija postojećih tokova.

U nastavku su prikazani izvorni kodovi za oba razreda. Primjer 17.5 prikazuje razred `KriptiraniUlazniTokOkteta`.

Primjer 17.5. Ulazni tok koji transparentno dekriptira podatke.

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Ulazni tok koji prilikom čitanja automatski dekriptira
 7  * podatke. Tok pretpostavlja da su podatci kriptirani
 8  * invertiranjem bitova.
 9  *
10  * @author marcupic
11  */
12 public class KriptiraniUlazniTokOkteta
13   extends UlazniTokOkteta {
14
15  private UlazniTokOkteta originalni;
16
17  /**
18   * Konstruktor.
19   * @param originalni originalni izvor kriptiranih podataka
20   */
21  public KriptiraniUlazniTokOkteta(UlazniTokOkteta originalni) {
22   this.originalni = originalni;
23  }
24
25  @Override
```

```
26  public int procitajOktet() throws IOException {
27   int procitano = originalni.procitajOktet();
28   if(procitano==-1) return procitano;
29   return (byte)((~procitano) & 0xFF);
30  }
31
32  @Override
33  public int preskoci(int n) throws IOException {
34   return originalni.preskoci(n);
35  }
36
37  @Override
38  public void close() throws IOException {
39   originalni.close();
40  }
41 }
42
```

Uočite: radi se o razredu koji smo naslijedili iz `UlazniTokOkteta` - pa to je implementacija ulaznog toka okteta i referencu na primjerak tog razreda možemo poslati bilo kojem klijentu koji očekuje ulazni tok okteta. U retku 15 deklarirana je referenca na originalni tok koji je stvarni izvor okteta. Konstruktor je prikazan u retcima 21-23; prima referencu na originalni tok i pamti je.

Metoda `procitajOktet()` najprije originalni tok traži čitanje jednog podatka (kažemo da mu delegira tu zadaću). Ako je originalni tok iscrpljen, tada i naša metoda vraća -1. Ako nam je originalni tok vratio podatak, mi ga dekriptiramo i vraćamo pozivatelju koji nije svjestan ovih operacija. Slično smo riješili i ostale metode.

Primjer 17.6 prikazuje razred `MeduspremnikUlazniTokOkteta`.

Primjer 17.6. Ulazni tok koji koristi međuspremnik okteta.

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Ulazni tok koji koristi međuspremnik za učinkovito
 7  * čitanje veće količine okteta.
 8  *
 9  * @author marcupic
10  */
11 public class MeduspremnikUlazniTokOkteta
12   extends UlazniTokOkteta {
13
14  private UlazniTokOkteta originalni;
15  private byte[] spremnik;
16  private int pocetak;
17  private int preostalo;
18  /**
19   * Konstruktor.
20   * @param originalni originalni izvor kriptiranih podataka
21   * @param velicina željena veličina međuspremnika
22   */
23  public MeduspremnikUlazniTokOkteta(UlazniTokOkteta originalni,
24   int velicina) {
25
26   if(velicina<1) {
27    throw new IllegalArgumentException(
```

```
28      "Veličina spremnika mora biti barem 1."
29     );
30    }
31    this.originalni = originalni;
32    this.spremnik = new byte[velicina];
33    this.preostalo = 0;
34   }
35
36   private void napuniSpremnik() throws IOException {
37    if(preostalo > 0) return;
38    int procitano = originalni.procitajOktet(
39     spremnik, 0, spremnik.length
40    );
41    if(procitano != -1) {
42     preostalo = procitano;
43     pocetak = 0;
44    }
45   }
46
47   @Override
48   public int procitajOktet() throws IOException {
49    if(preostalo<1) {
50     napuniSpremnik();
51     if(preostalo<1) {
52      return -1;
53     }
54    }
55    byte podatak = spremnik[pocetak];
56    pocetak++;
57    preostalo--;
58    return podatak;
59   }
60
61   @Override
62   public int procitajOktet(byte[] spremnik, int odmak, int broj)
63     throws IOException {
64
65    if(odmak < 0 || odmak >= spremnik.length) {
66     throw new IllegalArgumentException("Odmak nije legalan.");
67    }
68    if(broj < 1 || odmak+broj > spremnik.length) {
69     throw new IllegalArgumentException(
70      "Kombinacija odmak/broj nije legalna."
71     );
72    }
73
74    if(preostalo<1) {
75     napuniSpremnik();
76     if(preostalo<1) {
77      return -1;
78     }
79    }
80
81    int zaKopirati = Math.min(broj, preostalo);
82    for(int i = 0; i < zaKopirati; i++) {
83     spremnik[i+odmak] = this.spremnik[pocetak+i];
84    }
85    pocetak += zaKopirati;
```

```
 86    preostalo -= zaKopirati;
 87    return zaKopirati;
 88   }
 89
 90   @Override
 91   public void close() throws IOException {
 92    preostalo = 0;
 93    pocetak = 0;
 94    originalni.close();
 95   }
 96  }
 97
```

Ideja ovog razreda je vrlo jednostavna. Pri stvaranju primjerka zauzet ćemo i pomoćni memorijski spremnik. Kad god se isprazni, taj ćemo spremnik puniti podatcima iz originalnog toka koji smo primili kroz konstruktor i zapamtili, pri čemu ćemo punjenje raditi inačicom metode `procitajOktet(spremnik, odmak, broj)` koja je po pretpostavci vrlo učinkovita; pozvat ćemo je i zatražiti punjenje čitavog spremnika s kojim raspolažemo (a metoda će nam reći koliko je stvarno podataka unutra upisala). Za samo punjenje spremnika napisali smo pomoćnu privatnu metodu `napuniSpremnik` (retci 36-45) koja ujedno upravlja s dvije pomoćne članske varijable: `pocetak` čuva poziciju u međuspremniku na kojoj se nalazi prvi sljedeći oktet koji treba isporučiti klijentu a `preostalo` čuva broj okteta koji su upisani u spremnik i koji još nisu isporučeni klijentu.

Metoda `procitajOktet()` prikazana je u retcima 47-59. Metoda najprije provjerava ima li još podataka u spremniku te ako nema, poziva punjenje spremnika. Potom čita oktet u spremniku s pozicije `pocetak`, tu poziciju povećava za 1, `preostalo` umanjuje za jedan i pozivatelju vraća pročitani oktet. Uočite, ako se u međuspremniku nalazi 100 okteta, klijent će moći 100 puta pozvati `procitajOktet()` (koja će ga izuzetno brzo poslužiti jer podatak čita direktno iz spremnika), prije no što se od originalnog toka zatraže novi podatci (a i u tom slučaju, od originalnog toka će se opet zatražiti jedno čitanje većeg broja novih okteta koji će biti pohranjeni u spremnik).

Metoda `procitajOktet(spremnik, odmak, broj)` samo je malo složenija. Ideja je iz spremnika odjednom isporučiti onoliko podataka koliko ih ima u spremniku a da još uvijek stanu u polje koje je pripremio pozivatelj (retci 61-88).

Kako ćemo koristiti ove razrede? Neka je osnovni demonstracijski program prikazan u nastavku.

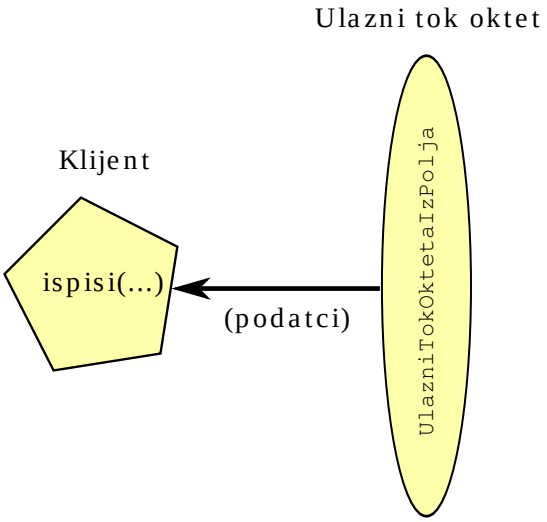
```
1 UlazniTokOkteta utok = new UlazniTokOktetaIzPolja(
2  // kriptirane vrijednosti okteta iz originalnog primjera
3  new byte[] {-2, 4, -29, 44, -38}
4 );
5
6 System.out.println("Ispis iz toka:");
7 ispisi(utok, 8);
8 System.out.println();
9 utok.close();
```

Kao rezultat ćemo dobiti ispis prikazan u nastavku.

```
Ispis iz toka:
1. podatak: -2
2. podatak: 4
3. podatak: -29
4. podatak: 44
5. podatak: -38
Tok više nema podataka.
```

Slika 17.3 ilustrira odnos između metode-klijenta (metoda `ispisi`) i ulaznog toka okteta (`UlazniTokOktetaIzPolja`). Klijent poziva metode za čitanje i izravno dobiva podatke izvornog toka.

Slika 17.3. Komunikacija klijent - izvorni tok.



Primjer koji pokazuje kako ostvariti transparentno dekriptiranje prikazan je u nastavku.

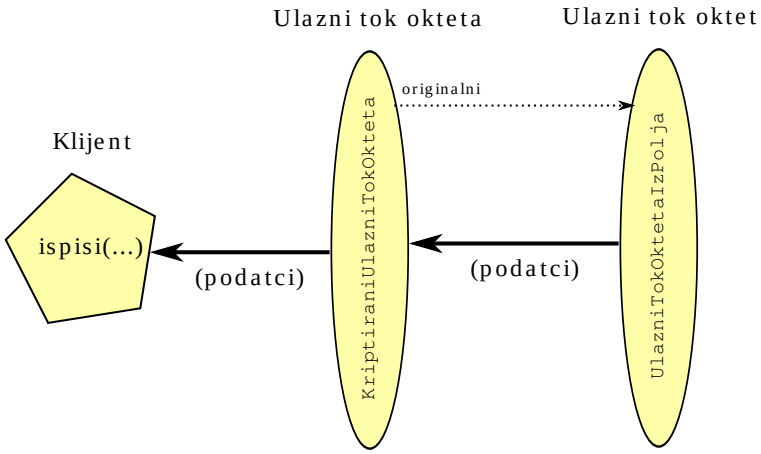
```
 1 UlazniTokOkteta utok = new KriptiraniUlazniTokOkteta(
 2   new UlazniTokOktetaIzPolja(
 3    // kriptirane vrijednosti okteta iz originalnog primjera
 4    new byte[] {-2, 4, -29, 44, -38}
 5   )
 6 );
 7
 8 System.out.println("Ispis iz toka:");
 9 ispisi(utok, 8);
10 System.out.println();
11 utok.close();
```

Kao rezultat ćemo dobiti ispis prikazan u nastavku.

```
Ispis iz toka:
1. podatak: 1
2. podatak: -5
3. podatak: 28
4. podatak: -45
5. podatak: 37
Tok više nema podataka.
```

Slika 17.4 ilustrira odnos između metode-klijenta (metoda `ispisi`) i ulaznog toka okteta (`UlazniTokOktetaIzPolja`) koji postoji u ovom slučaju. Klijent nema referencu na izvorni ulazni tok već na primjerak razreda `KriptiraniUlazniTokOkteta` i komunicira s njime; on pak ima referencu na izvorni tok i od njega traži podatke, obrađuje ih i potom vraća pozivatelju.

Slika 17.4. Komunikacija klijent - izvorni tok.



Konačno, primjer koji ilustrira da je moguće "zamotati" više razvijenih pomoćnih razreda oko originalnog, prikazan je u nastavku.

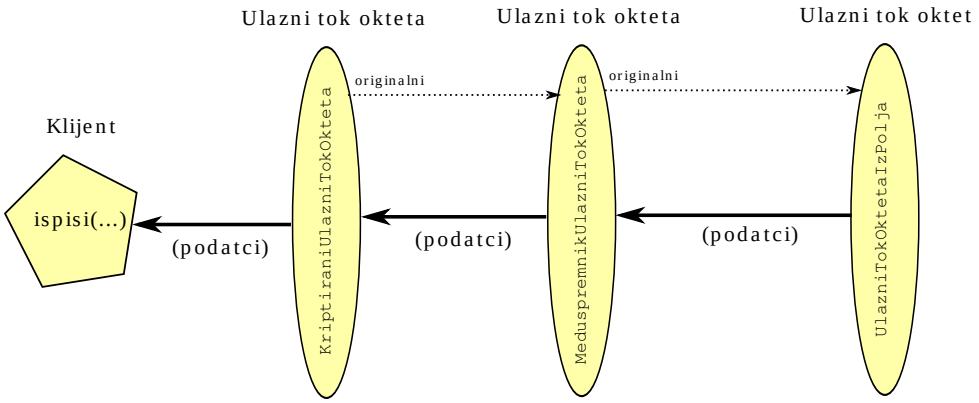
```
1 UlazniTokOkteta utok = new KriptiraniUlazniTokOkteta(
2   new MeduspremnikUlazniTokOkteta(
3    new UlazniTokOktetaIzPolja(
4     // kriptirane vrijednosti okteta iz originalnog primjera
5     new byte[] {-2, 4, -29, 44, -38}
6    ), 4
7   )
8 );
9
10 System.out.println("Ispis iz toka:");
11 ispisi(utok, 8);
12 System.out.println();
13 utok.close();
```

Kao rezultat ćemo dobiti ispis prikazan u nastavku.

```
Ispis iz toka:
1. podatak: 1
2. podatak: -5
3. podatak: 28
4. podatak: -45
5. podatak: 37
Tok više nema podataka.
```

Slika 17.5 ilustrira odnos između metode-klijenta (metoda `ispisi`) i ulaznog toka okteta (`UlazniTokOktetaIzPolja`) koji postoji u ovom slučaju. Klijent nema referencu na izvorni ulazni tok već na primjerak razreda `KriptiraniUlazniTokOkteta` i komunicira s njime; on pak ima referencu primjerak razreda `MeduspremnikUlazniTokOkteta` i od njega traži podatke; ovaj posljednji ima referencu na izvorni tok. Uočite kako smo time stvorili lanac: zahtjev za podatcima od klijenta putuje kroz pomoćne objekte prema izvornim toku nakon čega se podatak vraća korak po korak prema klijentu. U svakom međukoraku može se dogoditi odgovarajuća obrada podataka.

Slika 17.5. Komunikacija klijent - izvorni tok.



Pomoćnih razreda koji nisu generatori podataka već koji trebaju referencu na neki drugi izvor i koji podatke samo modificiraju ili nude neku drugu korisnu uslugu (poput povećanja učinkovitosti čitanja uporabom međuspremnika) možemo napisati još mnogo (spomenimo primjerice implementaciju koja bi u skladu sa specifikacijom ZIP transparentno otpakiravala izvorne podatke koji su komprimirani). Svim tim implementacijama zajedničko je da se radi o razredima koji su izvedeni iz općeg sučelja/razreda ulaznog toka (kako bi se i sami mogli klijentima predstavljati kao tokovi) te da primaju referencu na neki drugi tok od kojeg čitaju podatke. Takve implementacije nazivamo *dekoratorima*.

Krenete li i sami pisati takve dekoratore, uočit ćete da radite kopiranje dijelova koda. Primjerice, svi dekoratori trebat će deklarirati člansku varijablu koja pamti referencu na originalni (kažemo još *dekorirani*) tok. Da bismo izbjegli dupliciranje, uobičajeni je pristup napraviti razred koji predstavlja apstraktnog dekoratora (ne mora biti *apstraktan* u smislu OOP-a) koji deklarira člansku varijablu te potrebne metode i predstavlja bazu za izvođenje, i nakon toga iz njega izvesti konkretne dekoratore. Takvo je rješenje prikazano u nastavku.

Primjer 17.7. Apstraktni dekorator i konkretni dekoratori

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Razred predstavlja osnovu za izvođenje svih
 7  * dekoriranih ulaznih tokova okteta.
 8  *
 9  * @author marcupic
10  */
11 public abstract class DekoriraniUlazniTokOkteta
12   extends UlazniTokOkteta {
13
14  // Referenca na originalni tok čiji je ovo dekorator
15  protected UlazniTokOkteta originalni;
16
17  /**
18   * Konstruktor.
19   * @param originalni tok koji se dekorira
20   */
21  public DekoriraniUlazniTokOkteta(UlazniTokOkteta originalni) {
22   this.originalni = originalni;
23  }
24
25  @Override
```

```
26  public int procitajOktet() throws IOException {
27   return originalni.procitajOktet();
28  }
29
30  @Override
31  public void close() throws IOException {
32   originalni.close();
33  }
34 }
35
```

```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Ulazni tok koji prilikom čitanja automatski dekriptira
 7  * podatke. Tok pretpostavlja da su podatci kriptirani
 8  * invertiranjem bitova.
 9  *
10  * @author marcupic
11  */
12 public class KriptiraniUlazniTokOkteta
13   extends DekoriraniUlazniTokOkteta {
14
15  /**
16   * Konstruktor.
17   * @param originalni originalni izvor kriptiranih podataka
18   */
19  public KriptiraniUlazniTokOkteta(UlazniTokOkteta originalni) {
20   super(originalni);
21  }
22
23  @Override
24  public int procitajOktet() throws IOException {
25   int procitano = super.procitajOktet();
26   if(procitano==-1) return procitano;
27   return (byte)((~procitano) & 0xFF);
28  }
29 }
30
```

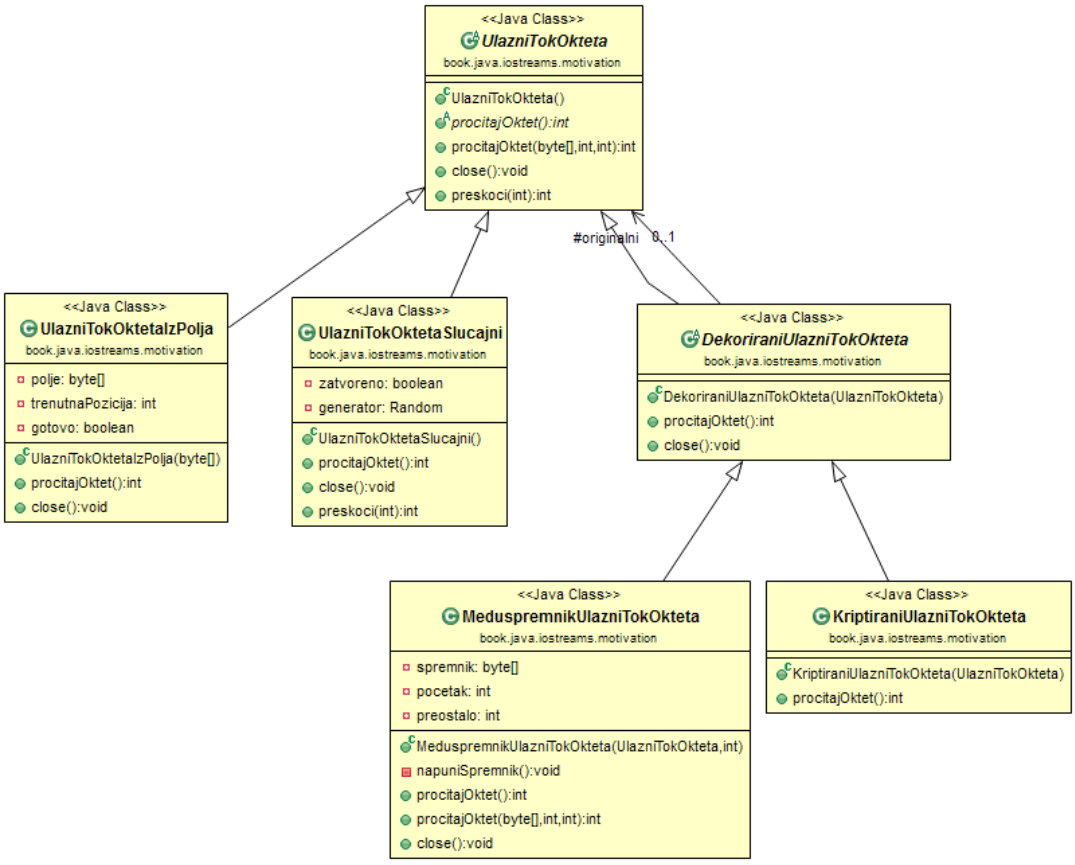
```
 1 package book.java.iostreams.motivation;
 2
 3 import java.io.IOException;
 4
 5 /**
 6  * Ulazni tok koji koristi međuspremnik za učinkovito
 7  * čitanje veće količine okteta.
 8  *
 9  * @author marcupic
10  */
11 public class MeduspremnikUlazniTokOkteta
12   extends DekoriraniUlazniTokOkteta {
13
14  private byte[] spremnik;
15  private int pocetak;
16  private int preostalo;
```

```
17  /**
18   * Konstruktor.
19   * @param originalni originalni izvor kriptiranih podataka
20   * @param velicina željena veličina međuspremnika
21   */
22  public MeduspremnikUlazniTokOkteta(UlazniTokOkteta originalni,
23   int velicina) {
24
25   super(originalni);
26   if(velicina<1) {
27    throw new IllegalArgumentException(
28     "Veličina spremnika mora biti barem 1."
29    );
30   }
31   this.spremnik = new byte[velicina];
32   this.preostalo = 0;
33  }
34
35  private void napuniSpremnik() throws IOException {
36   if(preostalo > 0) return;
37   int procitano = originalni.procitajOktet(
38    spremnik, 0, spremnik.length
39   );
40   if(procitano != -1) {
41    preostalo = procitano;
42    pocetak = 0;
43   }
44  }
45
46  @Override
47  public int procitajOktet() throws IOException {
48   if(preostalo<1) {
49    napuniSpremnik();
50    if(preostalo<1) {
51     return -1;
52    }
53   }
54   byte podatak = spremnik[pocetak];
55   pocetak++;
56   preostalo--;
57   return podatak;
58  }
59
60  @Override
61  public int procitajOktet(byte[] spremnik, int odmak, int broj)
62    throws IOException {
63
64   if(odmak < 0 || odmak >= spremnik.length) {
65    throw new IllegalArgumentException("Odmak nije legalan.");
66   }
67   if(broj < 1 || odmak+broj > spremnik.length) {
68    throw new IllegalArgumentException(
69     "Kombinacija odmak/broj nije legalna."
70    );
71   }
72
73   if(preostalo<1) {
74    napuniSpremnik();
```

```
75     if(preostalo<1) {
76      return -1;
77     }
78    }
79
80    int zaKopirati = Math.min(broj, preostalo);
81    for(int i = 0; i < zaKopirati; i++) {
82     spremnik[i+odmak] = this.spremnik[pocetak+i];
83    }
84    pocetak += zaKopirati;
85    preostalo -= zaKopirati;
86    return zaKopirati;
87   }
88
89   @Override
90   public void close() throws IOException {
91    preostalo = 0;
92    pocetak = 0;
93    originalni.close();
94   }
95  }
96
```

Obratite pažnju koliko je jednostavan dekorator koji obavlja dekriptiranje. Strukturni dijagram svih uključenih razreda prikazan je na slici 17.6.

Slika 17.6. Struktura razreda razvijenih u prethodnom razmatranju



Karakterističnost ove strukture je sljedeća. Postoji općenito sučelje (ili razred) koje definira osnovnu funkcionalnost. Postoji niz razreda koji su konkretne implementacije te funkcionalnosti

(na slici to su dva lijeva razreda izvedena iz osnovnog razreda). Postoji razred koji također izvodi osnovni razred i sadrži referencu na primjerak tog osnovnog razreda (taj je strukturni idiom poznat pod nazivom *rekurzivna kompozicija* i omogućava izgradnju lanaca objekata u memoriji, što smo ilustrirali na slikama 17.3, 17.4 i 17.5); konačno, iz tog je razreda izveden niz drugih koji sami za sebe ne nude nikakvu funkcionalnost već se koriste za modificiranje funkcionalnosti drugih razreda istog osnovnog tipa. Opisana struktura razreda te takav način rješavanja problema predstavlja oblikovni obrazac *Dekorator* koji je opisan u nastavku.

Oblikovni obrazac Dekorator

Pretpostavimo da trebate riješiti sljedeći problem. Imat ćete 10 komponenata (označimo ih OK1, OK2 do OK10) koje nude odgovarajuću osnovnu funkcionalnost, svaka na svoj jedinstveni način. Primjer 1: program treba dobivati oktete koje obrađuje a kao izvor okteta trebaju biti podržane datoteke, cjevovodi, dohvat preko mreže protokolom TCP, dohvat preko mreže protokolom UDP, *generiranje-u-letu prema nekom pravilu*, itd. Primjer 2: program obrađuje slike (obrada ne ovisi o izvoru slika) i treba ih dohvaćati s različitih izvora: iz datoteke, čitanjem preko kamere, uzimanjem snimke korisnikove radne površine (engl. *screenshot*), dohvatom preko mreže, itd.

Prvi zahtjev koji ćemo postaviti jest da klijent (primjer 1: metoda koja dohvaća i obrađuje oktete; primjer 2: metoda koja dohvaća i obrađuje slike) ne smije ovisiti o konkretnoj komponenti koja pruža osnovnu funkcionalnost. Rješenje nudi oblikovni obrazac *Strategija*: prisjetimo se: osnovnu funkcionalnost potrebno je apstrahirati kroz zajedničko sučelje i potom sve konkretne komponente izvesti iz tog sučelja. Klijent treba primati referencu na komponentu preko zajedničkog sučelja. U tom slučaju, klijentu ćemo moći poslati referencu na bilo koji osnovnu komponentu i klijent će s njom moći raditi potpuno transparentno. Štoviše, ako tri godine nakon što smo napisali program i klijenta utvrdimo da nam treba nova inačica osnovne komponente, dovoljno je napisati novi razred koji se opet izvodi iz zajedničkog sučelja. Time će stari klijent bez potrebe da ga modificiramo moći raditi i s najnovijom osnovnom komponentom čime je rješenje oblikovano u skladu s *Načelom nadogradnje bez promjene*.

Pretpostavimo sada da imamo još jedan zahtjev: naše rješenje treba biti takvo da omogućava da se svim osnovnim komponentama po potrebi doda bilo kakava kombinacija nekoliko novih funkcionalnosti (ili odgovornosti). Uz primjer 1, recimo da neovisno o konkretnoj korištenoj komponenti koja predstavlja izvor želimo klijentu slati oktet `(o+1) modulo 2` gdje je `o` oktet kojeg daje osnovna komponenta pri čitanju; ili, uz pretpostavku da je čitanje pojedinačnog okteta vremenski skupo, želimo između klijenta i stvarne osnovne komponente dodati međuspremnik (kako smo to učinili u prethodnom dijelu ovog poglavlja); ili, uz pretpostavku da okteti koje isporučuje osnovna komponenta odgovaraju sadržaju koji je komprimiran, želimo klijentu isporučivati oktete koji čine dekomprimirani sadržaj; ili, uz prepostavku da okteti koje isporučuje osnovna komponenta odgovaraju sadržaju koji je kriptiran, želimo klijentu isporučivati oktete koji čine dekriptirani sadržaj, itd. Uz primjer 2, neovisno o konkretnom izvoru slike, kao dodatne funkcionalnosti mogli bismo željeti klijentu isporučiti sliku na kojoj je "u letu" u gornji desni ugao dodan tekstovni zapis s trenutnim datumom i vremenom; mogli bismo poželjeti klijentu proslijediti sliku koja nad kojom je primijenjen morfološki operator erozije; mogli bismo klijentu proslijediti sliku s koje je odgovarajućim filterom uklonjena određena vrsta šuma; itd.

Ovakve su vrste zadataka u praksi dosta česte i želimo ih naučiti rješavati na kvalitetan način. Imamo li osnovne komponente O1 do Om, te zahtjevane dodatne funkcionalnosti F1 do Fn, te s obzirom da je svaku "obogaćenu" komponentu potrebno moći proslijediti klijentu preko zajedničkog sučelja, mogli bismo se izravno osloniti na naslijeđivanje i generirati razrede O1F1, O1F2, O1Fn, O1F1F2, O1F1F3, ..., O1F1F2...Fn, pa tako isto za komponente O2, O3 i sve do Om. Takvo rješenje ima dva problema zbog kojih ćemo ga odmah odbaciti: generira eksploziju broja novih razreda te uvodi masovnu redundanciju u koda (pogledajte samo u koliko smo razreda kopirali kod koji nudi funkcionalnost F1).

Sljedeći pristup koji pada napamet jest modificirati razrede O1 do Om tako da u svaki dodamo kod koji obavlja sve funkcionalnost te da dodamo skup zastavica te metode za njihovo postavljanje kojima korisnik može podesiti koja je od funkcionalnosti uključena a koja se preskače.

Prednost rješenja je što ne stvara nove razrede. Mana je opet uvođenje redundancija u kod (u svaki razred ćemo dopisivati/integrirati kod za svaku od funkcionalnosti) te, možda još važnije, kršenje načela nadogradnje bez promjene. Poželimo li implementirati novu funkcionalnost, morat ćemo ući u svih `m` osnovnih razreda i mijenjati njihov izvorni kod - a to ne želimo.

Razmotrimo treće rješenje. Idemo za svaku od dodatnih funkcionalnosti napisati jedan razred koji izvodimo iz zajedničkog sučelja i koji kroz konstruktor dobiva referencu neki drugi primjerak razreda izvedenog iz istog zajedničkog sučelja. Imamo li `m` dodatnih funkionalnosti, imat ćemo samo `m` novih razreda. Razred koji ostvaruje funkcionalnost Fi će u metodi zajedničkog sučelja koju koristi klijent najprije nad objektom na koji je dobio referencu u konstruktoru pozvati istu metodu, dobiveni će rezultat prilagoditi u skladu sa svojom funkcionalnošću te će tako modificirani rezultat vratiti svome pozivatelju (koji, uočite, može biti klijent ali i ne mora - to može biti neki treći objekt koji ostvaruje funkcionalnost Fj).

Na opisani način programeru stoji na raspolaganju mogućnost da stvori osnovnu komponentu `o` te da nju "zamota" u komponentu koja nudi funkcionalnost Fi (čime je dobio referencu na novi objekt `fi` koji u sebi čuva referencu na `o`). Tako dobivenu referencu na novi objekt potom je moguće "zamotati" u komponentu koja nudi funkcionalnost Fj (čime je dobio referencu na novi objekt `fj` koji u sebi čuva referencu na objekt `fi` koji u sebi čuva referencu na `o`), i tako dalje, po potrebi. Klijentu se šalje referenca na posljednje stvoreni objekt čime jedan poziv klijenta započinje lanac poziva kroz objekte koji nude pojedine dodatne funkcionalnosti od vanjskog (posljednje-stvorenog) prema unutarnjem i tako sve do osnovne komponente koja generira rezultat; taj se rezultat potom vraća u suprotnom smjeru od najdubljeg objekta prema vanjskom objektu s kojim komunicira klijent i svaki od tih objekata rezultat može po potrebi prilagoditi.

Važna karakteristika opisanog rješenja je da se dodatne funkcionalnosti ne definiraju na razini osnovnih komponenata već se konfiguriraju na razini pojedinih primjeraka osnovnih komponenata (drugim riječima, ne mora svaki izvor koji podatke čita iz datoteke biti obogaćen uporabom međuspremnika; možemo složiti čitanje iz dvije datoteke koje koriste međuspremnike te iz tri koje ne koriste međuspremnike).

Uočimo li sada da će svaki razred koji implementira dodatnu funkcionalnost trebati privatnu člansku varijablu za pohranu reference na objekt zajedničkog sučelja, dupliciranje koda možemo ukloniti još jednom izmjenom opisanog trećeg rješenja: iz zajedničkog sučelja izvest ćemo pomoćni razred koji deklarira referencu na neki drugi objekt zajedničkog sučelja, nudi konstruktor u kojem prima tu referencu te nudi implementacije svojih metoda tako da ih delegira objektu na koji je primio referencu. Potom ćemo sve razrede koji implementiraju dodatne funkcionalnosti izvesti iz tog novog pomoćnog razreda. Uz ovu modifikaciju upravo smo dali "recept" za rješavanje početno istaknutog problema koji je poznati pod nazivom *Oblikovni obrazac Dekorator*.

Oblikovni obrazac *Dekorator* identificira 4 sudionika.

1. *Apstraktna komponenta*: predstavlja zajedničko sučelje koje definira način na koji klijent komunicira s razmatranim objektima.

2. *Konkretne komponente*: predstavljaju implementacije osnovnih komponenata.

3. *Apstraktni dekorator*: predstavlja pomoćni razred koji smo uveli kako bismo na jednom mjestu deklarirali referencu na primjerak nekog drugog razreda preko zajedničkog sučelja i gdje smo riješili inicijalizaciju te reference te ponudili implementacije metoda delegiranjem dobivenom objektu.

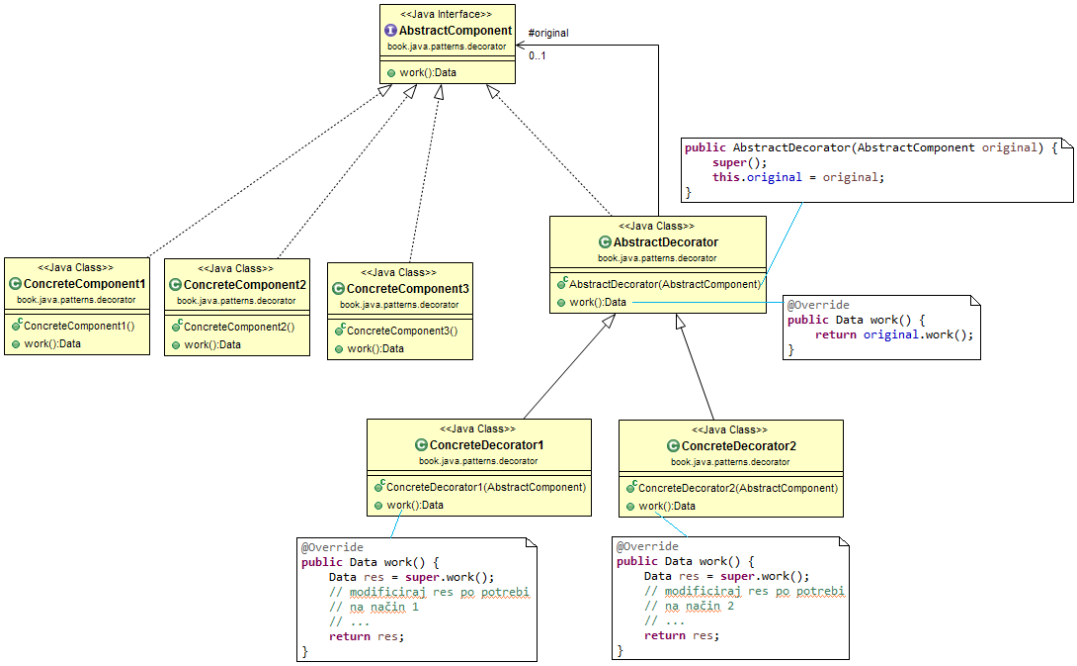
4. *Konkretni dekoratori*: predstavljaju razrede koji su izvedeni iz apstraktnog dekoratora i koji predstavljaju implementacija dodatnih funkcionalnosti.

Prikazano rješenje ima niz pogodnih svojstava. Dodatne funkcionalnosti mogu se po želji dodavati konkretnim komponentama neovisno o vrsti konkretne komponente. Nove funkcionalnosti mogu se dodavati u skladu s Načelom nadogradnje bez promjene: dovoljno je napisati novi

razred koji je izveden iz apstraktnog dekoratora i koji nudi implementaciju te nove funkcionalnosti. Bez ikakvih daljnjih promjena u kodu stari klijenti mogu koristiti tu novu funkcionalnost. Konačno, klijentski kod nije svjestan niti konkretnih komponenata niti dodatnih funkcionalnosti: za njega, čitavo rješenje izgleda kao obična *Strategija*.

Strukturni dijagram koji odgovara oblikovnom obrascu *Dekorator* prikazan je na slici 17.7.

Slika 17.7. Oblikovni obrazac Dekorator: strukturni dijagram



Uočite na slici: razred `AbstractDecorator` nasljeđuje razred/sučelje `AbstractComponent` i također ima referencu na primjerak razreda/sučelja `AbstractComponent`. Takav idiom nazivamo *rekurzivna kompozicija*. Rekurzivna kompozicija omogućava "zamatanje" objekata različitih komponenata i ona je osnovni dio oblikovnog obrasca Dekorator.



Važno

Rekurzivna kompozicija osnovni je gradbeni element oblikovnog obrasca Dekorator.

Izlazni tokovi okteta

Izlazni tok okteta na samom početku ovog poglavlja modelirali smo sučeljem `IzlazniTokOkteta`. Izlazni tokovi modeliraju ponore podataka: aplikacija u izlazne tokove zapisuje podatke. Očiti primjeri izlaznih tokova okteta bi bili objekti koji nude pohranu podataka u datoteku, prijenos podataka na drugo računalo putem računalne mreže i slično.

U sučelju `IzlazniTokOkteta` naveli smo tri metode. Zapisivanje okteta obavlja se pozivom metode `zapisiOktet(oktet)`. Radi se o metodi koji prima oktet i upisuje ga u tok. Ako iz bilo kojeg razloga u tok nije moguće taj tren upisati novi oktet (a da se ne radi o pogrešci), od metode očekujemo za pozivatelja zablokira sve dok se podatak ne uspije zapisati u tok. Ugovor prema pozivatelju jamči da, nakon što je izvođenje metode završilo (regularno), predani oktet je upisan u tok. U slučaju pogreške metoda će izazvati iznimku `IOException`.

Kako bi se omogućila implementacija izlaznih tokova koji učinkovito prenose podatke predvidjeli smo i metodu `flush`. Ideja je sljedeća: dopustit ćemo implementaciji da nakon što zaprimi oktet kroz metodu `zapisiOktet`, obradu okteta odgodi za neki kasniji trenutak. Primjerice,

radi li se o izlaznom toku koji oktete šalje preko mreže drugom računalu, umjesto da šalje oktet po oktet (što je u tom slučaju doista neučinkovito), implementacija bi mogla sačekati da korisnik preda nekoliko okteta pa ih potom sve prenese u jednom paketu preko mreže. Slično bismo mogli razmišljati i o implementaciji koja oktete zapisuje u datoteku na korisnikov disk. Metoda `flush` u takvom će scenariju omogućiti programeru da implementaciji dojavi da taj tren više nema nikakvih podataka i da je do tada zaprimljene oktete potrebno obraditi bez daljnjih čekanja (ako to već nije učinjeno).

Konačno, sučelje predviđa i metodu `close` čija je zadaća oslobađanje resursa koje je koristio tok (a time i prevođenje toka u stanje u kojem se s njime više ne može raditi).

Analizom programa koji rade s izlaznim tokovima brzo bismo uočili da nam je ovakvo sučelje prekruto - bilo bi lijepo kada bismo imali još poneku pomoćnu metodu. Jednakim razmišljanjem kao što smo to učinili pri analizi sučelja `UlazniTokOkteta` došli bismo do sličnog rješenja koje ćemo opisati u nastavku.

- Umjesto sučeljem, izlazni tok ćemo modelirati *apstraktnim razredom* `IzlazniTokOkteta`.
- U razredu `IzlazniTokOkteta` metoda `void zapisiOktet(byte oktet)` bit će apstraktna.
- U razredu `IzlazniTokOkteta` metode `void flush()` i `void close()` ima će praznu implementaciju.
- U razred `IzlazniTokOkteta` dodat ćemo metodu `void zapisiOktet(byte[] spremnik, int odmak, int broj)` koja u tok upisuje `broj` okteta iz predanog spremnika `spremnik` počev od pozicije `odmak`. Metoda će imati implementaciju koja to upisivanje ostvaruje iterativnim pozivanjem apstraktne metode `void zapisiOktet(byte oktet)`. Posljedica je da metoda pozivatelju garantira da će svi podatci koje je metoda trebala upisati u tok u trenutku povratka iz metode doista i biti upisani u tok (naravno, osim ako metoda završi s iznimkom a ne regularnim povratkom).

Razmotrimo još jedan zahtjev. Klijentima izlaznih tokova htjeli bismo ponuditi da transparentno rade i s tokovima koji imaju dodatne funkcionalnosti. Primjerice: klijent zapisuje oktete a implementacija toka ih transparentno kriptira algoritmom AES ili ih sažima algoritmom ZIP ili pak koristi međuspremnik kako bi povećala učinkovitost zapisivanja. Ovo sve htjeli bismo ponuditi neovisno o konkretnim implementacijama tokova (zapisivanje u datoteku, slanje premo mreže, ...).

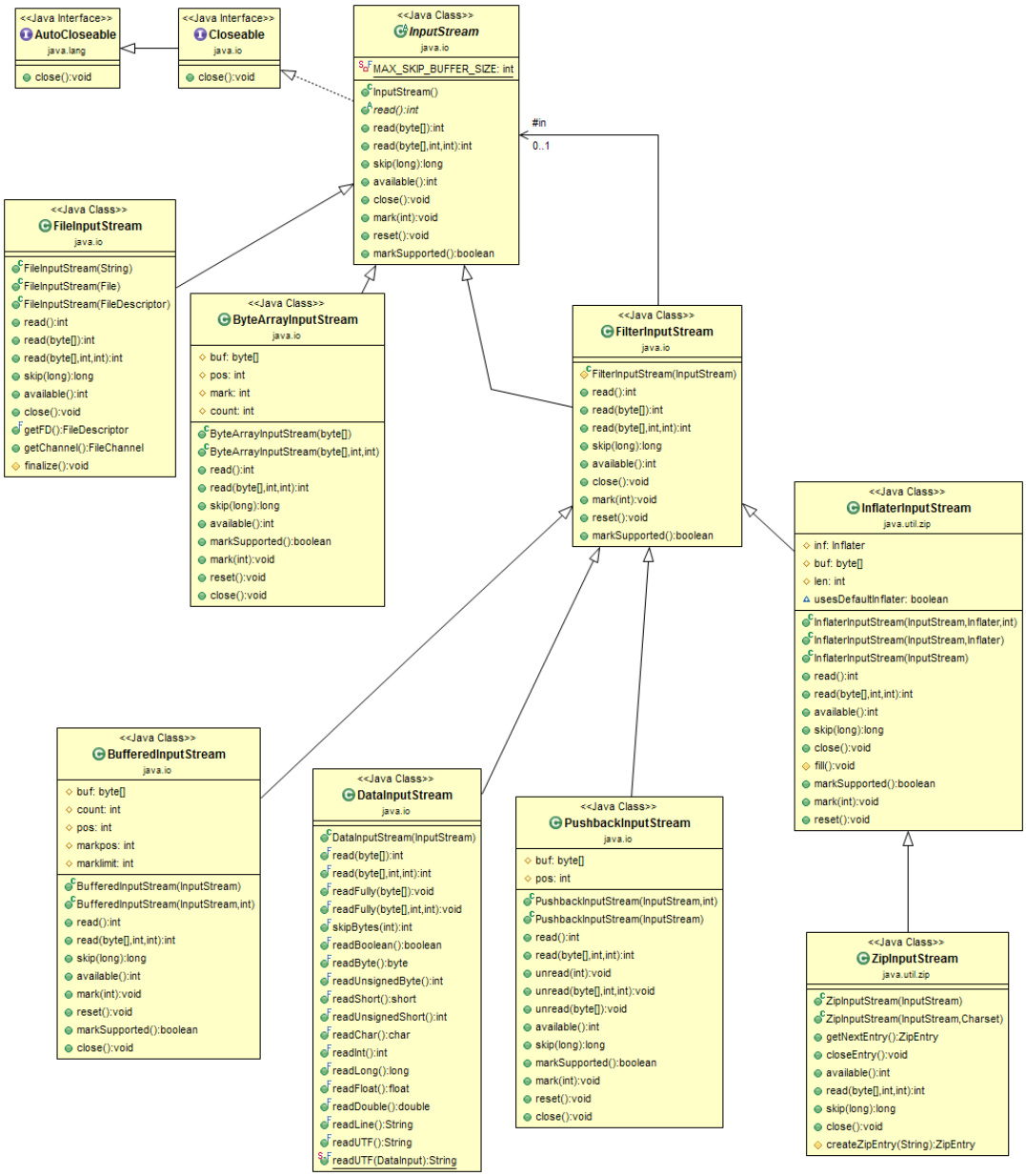
Rješenje je jednako kao i kod ulaznih tokova, a na ovom mjestu rješenju možemo dati i ime: razrede treba oblikovati u skladu s oblikovnim obrascem Dekorator, što znači da trebamo apstraktnu komponentu (u ovom primjeru to će biti apstraktni razred `IzlazniTokOkteta`), konkretne komponente (razredi `IzlazniTokUDatoteku`, `IzlazniTokTCP`, i slični), apstraktnog dekoratora (razred `DekoriraniIzlazniTokOkteta`) te konkretne dekoratore (razredi `MeduspremnikIzlazniTokOkteta`, `KriptiraniIzlazniTokOkteta`, `ZIPIzlazniTokOkteta`, itd).

Ako ste pažljivo prošli kroz prethodno napisani tekst, u ovom bi Vam trenutku trebala biti jasna ideja toka, razlika između ulaznog i izlaznog toka te načini oblikovanja razreda koji nude podršku za rad s tokovima okteta, a koji su oblikovani u skladu s oblikovnim obrascem Dekorator. Stoga ćemo se u nastavku uvjeriti da API koji nam u Javi stoji na raspolaganju izravno prati ovaj način razmišljanja.

Tokovi okteta i Javine biblioteke

Implementacije ulaznih i izlaznih tokova okteta u Javi su dostupne u paketu java.io. Ulazni tokovi okteta modelirani su apstraktnim razredom `java.io.InputStream` a izlazni tokovi okteta apstraktnim razredom `java.io.OutputStream`. Slika 17.8 prikazuje dijagram razreda za nekoliko odabranih ulaznih tokova.

Slika 17.8. Odabrani ulazni tokovi okteta u Javi



Sučelje `java.lang.AutoCloseable` (vidi primjer 17.8) modelira objekte koje je moguće zatvoriti, za što je predviđena metoda `void close()`. Ovo sučelje ne odnosi se isključivo na objekte koji predstavljaju tokove već na bilo koju vrstu objekata koja upravlja određenim resursima. Stoga je i sama metoda `void close()` deklarirana kao metoda koja pri pogrešci izaziva najopćenitiju vrstu iznimke (modeliranu razredom `java.lang.Exception`).

Primjer 17.8. Sučelje `java.lang.AutoCloseable`

```
1 package java.lang;
2
3 public interface AutoCloseable {
4     void close() throws Exception;
5 }
6
```

Specijalizacijom tog sučelja izvedeno je sučelje `java.io.Closeable` (vidi primjer 17.9) koje pobliže specificira semantiku metode `close()` i zahtjeva da višestruko pozivanje te metode

nema nikakvih posljedica u odnosu na samo jedan poziv (na tragu ideje: pokušaj zatvaranja nečega što je već zatvoreno ne radi ništa). Pretpostavlja se da će objekti modelirani ovim sučeljem biti tokovi što je vidljivo i iz iznimke koju će metoda izazvati u slučaju pogreške, a koja je modelirana razredom `java.io.IOException`. Istaknimo da metoda `close()` sučelja `java.lang.AutoCloseable` dozvoljava da se implementacija ponaša drugačije pri drugom i naknadnim pozivima, odnosno nema zahtjeva *idempotentnosti*.

Primjer 17.9. Sučelje `java.io.Closeable`

```
1 package java.io;
2
3 import java.io.IOException;
4
5 public interface Closeable extends AutoCloseable {
6     public void close() throws IOException;
7 }
8
```

Ulazni tok okteta modeliran je apstraktnim razredom `java.io.InputStream` koji implementira sučelje `java.io.Closeable`. Razred je apstraktan samo zbog jedne apstraktne metode: `int read()` koja čita i vraća jedan oktet (vrijednosti 0 do 255) ili pak -1 ako u toku više nema podataka (tj. ako je tok iscrpljen). Uočite kako je povratni tip metode proširen na `int` kako bi ovo bilo moguće. Uz tu metodu, razred specificira i nudi implementacije još dvije metode za čitanje: prva prima referencu na spremnik, odmak u spremniku te broj okteta koje je potrebno pročitati dok je druga jednostavnija inačica koja prima samo referencu na spremnik i ponaša se kao ova općenitija uz odmak postavljen na 0 te broj postavljen na veličinu predanog polja. Obje garantiraju da će pročitati barem jedan oktet a najviše traženi broj okteta; koliko ih je točno pročitano, metoda vraća kao povratnu vrijednost. U slučaju da je tok iscrpljen, obje metode će vratiti -1.

Osim metoda za čitanje, na raspolaganju su i metoda za preskakanje zadanog broja okteta (`skip`), metoda kojom možemo pitati koliko je okteta taj tren dostupno u toku (`available`; može biti manje od ukupnog broja preostalih okteta u toku), metoda za zatvaranje toka (`close`) te tri pomoćne metode: `markSupported`, `mark` i `reset`. Kako je ulazni tok inherentno jednosmjeran, ideja ovih posljednjih triju metoda je korisniku ponuditi jednostavan mehanizam kojim je stanje toka može restaurirati na neko prethodno zapamćeno blisko stanje. Konkretno, korisnik u određenom trenutku pozove metodu `mark()` i u tom trenutku tok "zapamti" svoju trenutnu poziciju. Korisnik tada može pročitati nekoliko okteta i nakon toga pozvati metodu `reset` - u tom trenutku tok se vraća u stanje u kojem je bio pri pozivu metode `mark` (kao da korisnik nije pročitao onih nekoliko okteta; kada krene čitanje, tok će opet isporučiti najprije te oktete). Ovaj mehanizam nije općenito podržan od svih ulaznih tokova i prije njegova korištenja potrebno je pozvati metodu `markSupported` koja će vratiti `true` ako je ovaj mehanizam podržan od implementacije toka; u suprotnom se isti ne smije koristiti (i izazvat će iznimku `java.io.IOException`). Maksimalni broj okteta koje nakon poziva metode `mark` tok mora biti u stanju vratiti pri pozivu metode `reset` zadaje se kao argument metode `mark`. Kako je ovaj mehanizam kod nekih tokova podržan a kod nekih nije, algoritmi koji rade s ulaznim tokovima okteta ga izbjegavaju.

Prije no što nastavimo dalje s pregledom, vratite se na sliku 17.8 i obratite pažnju *na strukturu prikazanih razreda*. Zanemarite sučelja `java.lang.AutoCloseable` i `java.io.Closeable`. Na što Vas ta organizacija podsjeća?

Radi se, dakako, o oblikovnom Dekorator u skladu s kojim je oblikovana Javina podrška za rad s tokovima - ovu strukturu vidjet ćemo i kod organizacije izlaznih tokova okteta baš kao i kod organizacije ulaznih i izlaznih tokova znakova. Konkretno, terminologijom tog oblikovnog obrasca na prikazanoj slici apstraktni razred `java.io.InputStream` je *Apstraktna komponenta*, razredi `java.io.FileInputStream` i `java.io.ByteArrayInputStream` su *Konkretne komponente*, razred `java.io.FilterInputStream` (uočavate li na dijagramu rekurzivnu kompoziciju?) je *Apstraktni dekorator* dok su razredi `java.io.DataInputStream`,

`java.io.PushbackInputStream` te `java.util.zip.InflaterInputStream` (odnosno razred `java.util.zip.ZipInputStream`) *Konkretni dekoratori*.

U standardnim Javinim bibliotekama na raspolaganju nam stoji niz konkretnih implementacija ulaznih tokova okteta od kojih ćemo u ovom trenutku spomenuti dvije. Razred `java.io.FileInputStream` predstavlja ulazni tok okteta koji iste dohvaća iz datoteke. Ima nekoliko konstruktora a dva najčešće korištena su konstruktor koji prima putanju do datoteke zadanu kao `java.lang.String` odnosno konstruktor koji prima putanju do datoteke zadanu kao primjerak razreda `java.io.File`. U slučaju da datoteka ne postoji, konstruktori će izazvati iznimku `java.io.FileNotFoundException`.

Primjer 17.10 ilustrira uporabu ovog razreda za izračun zaštitne sume zadane datoteke. Zaštitnu sumu datoteke definirat ćemo kao sumu svih okteta koji čine sadržaj datoteke. Znamo li zaštitnu sumu datoteke, tada ćemo, u slučaju da se sadržaj datoteke promijeni u jednom oktetu to moći i ustanoviti jer nanovo izračunata zaštitna suma više neće odgovarati prethodno izračunatoj.

Primjer 17.10. Izračun zaštitne sume zadane datoteke (1)

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.FileInputStream;
 4 import java.io.IOException;
 5
 6 public class RacunajZastitnuSumu {
 7
 8  public static void main(String[] args) {
 9
10   if(args.length != 1) {
11    System.out.println("Morate zadati putanju do datoteke.");
12    return;
13   }
14
15   try {
16    int suma = racunajSumu(args[0]);
17    System.out.println("Zaštitna suma iznosi: " + suma + ".");
18   } catch(IOException ex) {
19    System.out.println("Pogreška: " + ex.getMessage());
20   }
21  }
22
23  private static int racunajSumu(String putanja)
24   throws IOException {
25
26   FileInputStream fis = null;
27   int suma = 0;
28
29   try {
30    fis = new FileInputStream(putanja);
31    while(true) {
32     int oktet = fis.read();
33     if(oktet==-1) break;
34     suma += oktet;
35    }
36   } finally {
37    if(fis!=null) {
38     try { fis.close(); } catch(IOException ignorable) {}
39    }
40   }
```

```
41
42    return suma;
43   }
44
45 }
46
```

Metoda `racunajSumu` za izračun zaštitne sume prikazana je u retcima 23-45. Tijelo metode podijeljeno je u tri dijela: stvaranje ulaznog toka (redak 30: poziv konstruktora razreda `FileInputStream`), čitanje okteta iz toka i njihova obrada (petlja u retcima 31-35) te zatvaranje stvorenog toka (redak 38).

Pogreške se pri tome mogu javiti na tri mjesta: pri otvaranju ulaznog toka, pri čitanju okteta iz toka te pri zatvaranju toka. Što je potrebno učiniti ako dođe po pogreške, nije odgovornost ove metode; hoće li zbog toga na zaslon biti ispisana poruka o pogrešci, hoće li ista biti zanemarena, hoće li korisniku biti prikazan dijalog s porukom ili će pak poruka biti zapisana u dnevnik programa, to ova metoda ne može znati. Stoga je metoda deklarirana tako da i sama prosljeđuje pogrešku tipa `IOException` pozivatelju. Međutim, s obzirom da smo metodu napisali na način da ona sama otvara ulazni tok, u slučaju pogreške pozivatelj nema referencu na taj tok i ne može ga sam zatvoriti. Stoga smo iskoristili konstrukt `try { ... } finally { ... }` kojim osiguravamo da će, neovisno o načinu na koji se izvede kod koji otvara i obrađuje datoteku, ista na kraju biti zatvorena. Sam poziv metode `close()` također može izazvati iznimku (koja bi u `finally`-dijelu maskirala prethodno izazvanu iznimku u slučaju da je iznimka nastala pri otvaranju ili obradi datoteke). Stoga smo pokušaj zatvaranja datoteke okružili u novi `try {...} catch(...) {...}` blok u kojem eventualno izazvanu iznimku metode `close()` gušimo. Uočite također da ulaskom u blok `finally` ne znamo jesmo li u njega došli zbog toga što datoteku uopće nismo mogli otvoriti (jer ista ne postoji, pa nemamo što zatvarati) ili nakon uspješnog otvaranja datoteke te normalne obrade ili obrade prekinute iznimkom (u kojem slučaju tok jest stvoren). Koji je od ovih slučajeva nastupio provjeravamo usporedom reference na tok s vrijednošću `null` i tok pokušavamo zatvoriti samo ako je isti doista i bio stvoren.

Da bismo referencu na tok vidjeli u bloku `finally`, istu nismo mogli deklarirati u bloku `try` već smo deklaraciju morali napraviti prije čitavog konstrukta `try - finally`. No takvim rješenjem ova referenca nije vidljiva samo kroz blokove `try - finally` već i u ostatku tijela metode, čime je zagađen prostor imena a sama referenca je vidljiva i u kodu koji je ne bi trebao biti svjestan (a kamoli koristiti).

Sumirajmo uočene nedostatke. Prikazano rješenje ima dva problema: (i) na neki način moramo osigurati da će otvoreni tok (ili općenito resurs) biti na kraju zatvoren, neovisno o tome je li obrada završila regularno ili s iznimkom te (ii) referenca na tok vidljiva je i nakon koda koji radi obradu podataka iz toka.

Od Jave 7, u jezik je dodana podrška za *automatsko upravljanje resursima* u obliku proširene sintakse konstrukta `try` koji će osigurati da je vidljivost resursa vidljiva samo unutar bloka koji radi njegovu obradu i koji će osigurati da će resurs pri napuštanju tog bloka biti zatvoren, neovisno o tome je li obrada završila regularno ili s iznimkom. Da bismo koristili ovu mogućnost, resurs nad kojim želimo automatsko upravljanje moramo deklarirati u oblim zagradama odmah nakon ključne riječi `try` a prije otvorene vitičaste zagrade u kojoj se nalazi kod koji ga obrađuje. Metoda `racunajSumu` koja koristi ovu mogućnost prikazana je u nastavku.

```
1 private static int racunajSumu(String putanja)
2   throws IOException {
3
4  int suma = 0;
5
6  try(FileInputStream fis = new FileInputStream(putanja)) {
7   while(true) {
8    int oktet = fis.read();
9    if(oktet==-1) break;
10   suma += oktet;
```

```
11    }
12   }
13
14   // na ovom mjestu referenca fis više nije vidljiva
15   // a tok je sigurno zatvoren
16
17   return suma;
18  }
```

Kako u prikazanom bloku `try` nismo dodavali blokove koji se bave obradom iznimki, sve eventualne iznimke koje nastanu bit će proslijeđene pozivatelju. Želimo li sami raditi obradu iznimaka, to možemo učiniti na uobičajen način (bez ikakvog utjecaja na automatsko zatvaranje toka koje će se i dalje dogoditi).



Važno

Automatsko ipravljanje resursima moguće je samo nad primjercima razreda koji implementiraju sučelje `java.lang.AutoCloseable`.

Metoda `racunajSumu` kako smo je trenutno zamislili ima više odgovornosti: ona određuje vrstu toka čiji se sadržaj obrađuje te način obrade. Jasna posljedica toga je da ćemo pri nadogradnji (recimo, želimo računati zaštitnu sumu toka koji podatke čita iz pripremljenog polja u memoriji - funkcionalnost nudi razred `java.io.ByteArrayInputStream`) morati uvoditi dupliciranje koda. Napravimo stoga reorganizaciju koda u kojoj ćemo odgovornost određivanja vrste toka ostaviti pozivatelju: neka metoda `racunajSumu` prima referencu na apstraktni tok (što je primjena oblikovnog obrasca Strategija). Tada istu metodu možemo pozivati nad različitim vrstama tokova, kako je prikazuje primjer 17.11.

Primjer 17.11. Izračun zaštitne sume ulaznog toka okteta

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.ByteArrayInputStream;
 4 import java.io.FileInputStream;
 5 import java.io.IOException;
 6 import java.io.InputStream;
 7
 8 public class RacunajZastitnuSumu3 {
 9
10  public static void main(String[] args) {
11
12   if(args.length != 1) {
13    System.out.println("Morate zadati putanju do datoteke.");
14    return;
15   }
16
17   byte[] spremnik = {1, 5, 17, 23, -48, 21, -52};
18
19   try(InputStream fis = new FileInputStream(args[0])) {
20    int suma = racunajSumu(fis);
21    System.out.println("Zaštitna suma iznosi: " + suma + ".");
22   } catch(IOException ex) {
23    System.out.println("Pogreška: " + ex.getMessage());
24   }
25
26   try(InputStream bis = new ByteArrayInputStream(spremnik)) {
27    int suma = racunajSumu(bis);
28    System.out.println("Zaštitna suma iznosi: " + suma + ".");
29   } catch(IOException ex) {
```

```
30     System.out.println("Pogreška: " + ex.getMessage());
31    }
32   }
33
34   private static int racunajSumu(InputStream is)
35     throws IOException {
36
37    int suma = 0;
38
39    while(true) {
40     int oktet = is.read();
41     if(oktet==-1) break;
42     suma += (byte)oktet;
43    }
44
45    return suma;
46   }
47
48 }
49
```

U ovom primjeru automatsko upravljanje resursima podigli smo u metodu `main` koja sada stvara tokove i brine za njihovo zatvaranje. Primjer ilustrira računanje zaštitne sume nad dva toka: jedan koji oktete dohvaća iz datoteke na disku te drugi koji oktete dohvaća iz unaprijed pripremljenog polja; za ovaj posljednji, očekivana zaštitna suma, s obzirom na pripremljene oktete bit će `-33`. Izbacite li u metodi `racunajSumu` prije pribrajanja ukalupljivanje u tip `byte`, izračunati rezultat bit će `479`: možete li objasniti zašto?

Prikazani kod mogli smo reorganizirati na još jedan način. Razmislite o reorganizaciji koju smo maloprije napravili. Iz rješenja u kojem metoda `racunajSumu` ima tri odgovornosti (odabir i stvaranje toka, obrada toka, zatvaranje toka) otišli smo u rješenje u kojem ima samo jednu odgovornost (obrada toka) dok smo odgovornosti odabira i stvaranje toka, te zatvaranje toka prepustili pozivatelju. Kako se radi o dvije odgovornosti, radi li pozivatelj s više tokova, radit će i dupliciranje koda. Rješenje koje je prikazano u nastavku pozivatelju omogućava da definira što će biti odabrano i stvoreno kao resurs (ali ne i kada) dok metodi `racunajSumu` vraća sve tri odgovornosti ali u skladu s Načelom nadogradnje bez promjene: metoda će dobiti referencu na strategiju koja stvara tok i sve odraditi uporabom automatskog upravljanja resursima. Takvo rješenje prikazuje primjer 17.12.

Primjer 17.12. Izračun zaštitne sume ulaznog toka okteta

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.ByteArrayInputStream;
 4 import java.io.FileInputStream;
 5 import java.io.IOException;
 6 import java.io.InputStream;
 7
 8 public class RacunajZastitnuSumu4 {
 9
10  @FunctionalInterface
11  public interface InputStreamSupplier {
12   InputStream get() throws IOException;
13  }
14
15  public static void main(String[] args) {
16
17   if(args.length != 1) {
18    System.out.println("Morate zadati putanju do datoteke.");
```

```
19     return;
20    }
21
22    byte[] spremnik = {1, 5, 17, 23, -48, 21, -52};
23
24    try {
25     int suma = racunajSumu(
26      ()->new FileInputStream(args[0])
27     );
28     System.out.println("Zaštitna suma iznosi: " + suma + ".");
29    } catch(IOException ex) {
30     System.out.println("Pogreška: " + ex.getMessage());
31    }
32
33    try {
34     int suma = racunajSumu(
35      ()->new ByteArrayInputStream(spremnik)
36     );
37     System.out.println("Zaštitna suma iznosi: " + suma + ".");
38    } catch(IOException ex) {
39     System.out.println("Pogreška: " + ex.getMessage());
40    }
41   }
42
43   private static int racunajSumu(InputStreamSupplier supplier)
44     throws IOException {
45
46    int suma = 0;
47
48    try(InputStream is = supplier.get()) {
49     while(true) {
50      int oktet = is.read();
51      if(oktet==-1) break;
52      suma += oktet;
53     }
54    }
55
56    return suma;
57   }
58
59  }
60
```

Kako bi korisnik mogao definirati način stvaranja toka koji će biti korišten za izvođenje izračuna zaštitne sume, pripremili smo funkcijsko sučelje `InputStreamSupplier` (retci 10-13) koje definira metodu `get()` čija je zadaća stvoriti i vratiti ulazni tok ili izazvati iznimku `java.io.IOException` u slučaju pogreške. U paketu java.util.function postoji parametrizirano sučelje `Supplier<T>` slične namjene ali koje nam nije iskoristivo jer njegova metoda `get` ne deklarira izazivanje iznimke `java.io.IOException`.

Nova inačica metode `racunajSumu` prikazana je u retcima 43-59. Metoda prima referencu na objekt tipa `InputStreamSupplier` te uporabom automatskog upravljanja resursima traži stvaranje toka, definira obradu i potom zatvara tok. U metodi `main` sada se ponovno bavimo samo odgovarajućom obradom pogrešaka te pri pozivima metode `racunajSumu` objekt koji stvara i vraća tok pri pozivu metode `get` definiramo uporabom lambda izraza.

Vratimo se opet na sliku 17.8 gdje su prikazana i četiri dekoratora. Dekorator `BufferedInputStream` omogućava dodavanje međuspremnika u rad s ulaznim tokom okteta. Tako primjerice umjesto toka:

```
1 InputStream is = new FileInputStream(args[0]);
```

možemo klijentima predavati tok s međuspremnikom:

```
1 InputStream is = new BufferedInputStream(
2   new FileInputStream(args[0])
3 );
```

što će omogućiti klijentskom kodu da koristi bilo metode pojedinačnog dohvata okteta bilo metode koje čitaju u pomoćno polje na jednako učinkovit način.

Dekorator `DataInputStream` omogućava čitanje i konverziju više okteta u Javine primitivne tipove. Primjerice, ako su sljedeća dva okteta u toku `is` viši i niži oktet nastao pri zapisivanju varijable tipa `short` uporabom konvencije *Big-Endian*, sadržaj te varijable možemo rekonstruirati izravno sljedećim isječkom:

```
1 short var = 0;
2 int oktetVisi = is.read();
3 if(oktetVisi!=-1) {
4  int oktetNizi = is.read();
5  if(oktetNizi!=-1) {
6   var = (short)((oktetVisi << 8) + oktetNizi);
7  } else {
8   throw new EOFException();
9  }
10 } else {
11  throw new EOFException();
12 }
13 // Sada imamo broj:
14 System.out.println(var);
```

što je doista puno koda. Želimo li na sličan način riz niza okteta ekonstruirati podatak tipa `int`, `long`, `float`, `double` i slično, kod će se još više zakomplicirati. Dekorator `DataInputStream` sav ovaj kod već ima implementiran u prikladnim metodama. Stoga čitanje broja tipa `short` možemo puno jednostavnije postići njegovom uporabom:

```
1 InputStream is = ...;
2 // dekoriraj izvorni tok:
3 DataInputStream dis = new DataInputStream(is);
4 // čitaj preko dekoratora:
5 short var = dis.readShort();
6 System.out.println(var);
```

Razred `java.io.PushbackInputStream` jer dekorator koji *garantirano* omogućava da se pročitani okteti vrate natrag u tok kako bi pri pozivima metoda za čitanje ponovno bili pročitani. Postoje problemi koji se uz takav tok mogu rješavati nešto elegantnije. Konačno, dekorator `java.util.zip.ZipInputStream` omogućava transparentnu dekompresiju podataka koji su bili komprimirani algoritmom ZIP.

U nastavku ćemo dati još jedan primjer: želimo modificirati metodu `racunajSumu` tako da umjesto pojedinačnog čitanja okteta koristi vlastito pomoćno polje okteta. Evo rješenja.

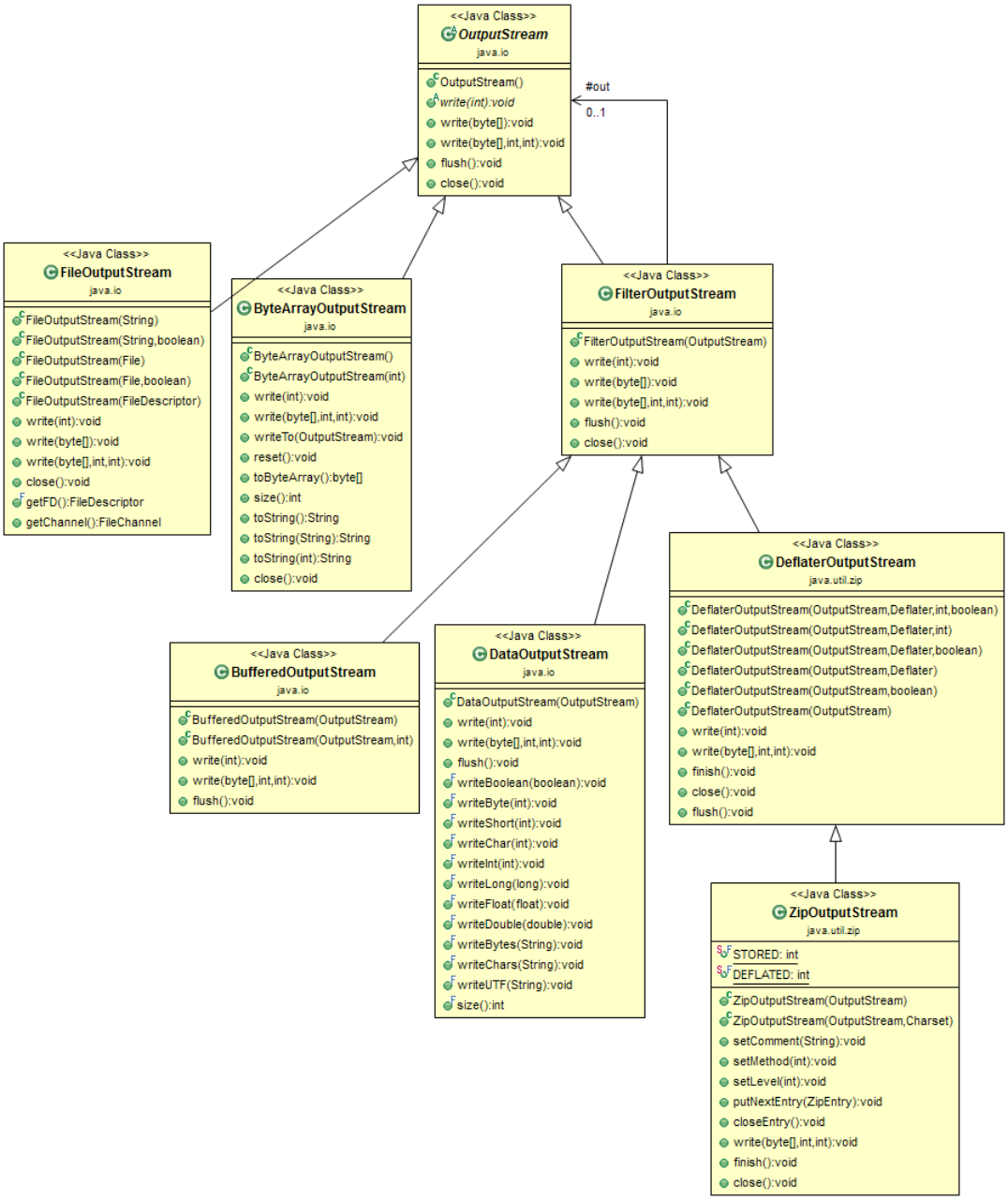
```
1 private static int racunajSumu(InputStreamSupplier supplier)
2   throws IOException {
3
4  int suma = 0;
5  byte[] spremnik = new byte[1024];
6
7  try(InputStream is = supplier.get()) {
```

```
 8   while(true) {
 9    int procitano = is.read(spremnik);
10    if(procitano < 1) break;
11    for(int i = 0; i < procitano; i++) {
12     suma += spremnik[i];
13    }
14   }
15  }
16
17  return suma;
18 }
```

U retku 9 obavljamo čitanje okteta u pripremljeni spremnik koji smo zauzeli prije ulaska u tijelo petlje (redak 5). Ako je tok iscrpljen, metoda `read` će vratiti `-1` pa tada izlazimo iz petlje. U suprotnom, metoda vraća koliko je okteta doista pročitala i upisala u pripremljeno polje pa u petlji prolazimo samo kroz te oktete i radimo zbrajanje.

Dijagram razreda odabranih izlaznih tokova okteta koji se nude u Javinim standardnim bibliotekama prikazan je na slici 17.9.

Slika 17.9. Odabrani izlazni tokovi okteta u Javi



Prikazali smo "konceptualno" jednak odabir razreda kao i kod ulaznih tokova okteta. Uočite primjenu oblikovnog obrasca Dekorator: apstraktna komponenta je apstraktni razred `java.io.OutputStream` kojom su modelirani izlazni tokovi okteta; konkretne komponente su razredi `java.io.FileOutputStream` (koji omogućava zapisivanje okteta u datoteku) te `java.io.ByteArrayOutputStream` (koja omogućava zapisivanje u interni spremnik koji čijom veličinom samostalno upravlja i čiji se sadržaj na kraju može dobiti pozivom metode `toByteArray()`; apstraktni dekorator je `java.io.FilterOutputStream` a konkretni dekoratori su `java.io.BufferedOutputStream` (dodaje međuspremnik u postupak zapisivanja), `java.io.DataOutputStream` (pandan razredu `java.io.DataInputStream` - obavlja serijalizaciju primitivnih tipova u slijed okteta) te `java.util.zip.DeflaterOutputStream` i iz njega izveden razred `java.util.zip.ZipOutputStream` koji omogućava transparentnu kompresiju podataka.

Razred `java.io.OutputStream` je model osnovnog izlaznog toka okteta. Nudi tri inačice metode `write`: apstraktnu inačicu koja prima jedan oktet koji treba zapisati te dvije inačice

koje imaju implementaciju koja se temelji na prvoj inačici: jedna prima polje okteta koje čitavo treba zapisati dok druga prima polje, odmak i broj okteta. Sve ove inačice nemaju povratne vrijednosti jer garantiraju (osim ako dođe do iznimke) da će zaprimljene podatke upisati u tok. Postoji još i metoda `flush` koja nalaže toku da svo odgođeno zapisivanje obavi (ako je korišten međuspremnik, da se podatci iz njega zapišu u tok) te metoda `close` koja zatvara tok.

Pogledajmo nekoliko primjera. Primjer 17.13 prikazuje metodu koja kopira datoteku. Međuspremnik se koristi i za ulazni tok kojim se čita izvorna datoteka i za izlazni tok kojim se okteti zapisuju u novu datoteku.

Primjer 17.13. Kopiranje datoteke

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.BufferedInputStream;
 4 import java.io.BufferedOutputStream;
 5 import java.io.FileInputStream;
 6 import java.io.FileOutputStream;
 7 import java.io.IOException;
 8 import java.io.InputStream;
 9 import java.io.OutputStream;
10
11 public class Kopiranje {
12
13  public static void main(String[] args) {
14
15   try {
16    kopiraj(args[0], args[1]);
17   } catch(IOException ex) {
18    System.out.println("Pogreška: " + ex.getMessage());
19   }
20
21  }
22
23  private static void kopiraj(String origDatoteka,
24    String novaDatoteka) throws IOException {
25   try(
26    InputStream is = new BufferedInputStream(
27     new FileInputStream(origDatoteka)
28    );
29    OutputStream os = new BufferedOutputStream(
30     new FileOutputStream(novaDatoteka)
31    );
32   ) {
33    byte[] spremnik = new byte[1024];
34    while(true) {
35     int procitano = is.read(spremnik);
36     if(procitano < 1) break;
37     os.write(spremnik, 0, procitano);
38    }
39   }
40  }
41
42 }
43
```

Koristeći automatsko upravljanje resursima definirali smo dva resursa: ulazni tok i izlazni tok. Pojedine deklaracije razdvojili smo znakom točka-zarez. Na isti način mogli smo po potrebi definirati i više resursa. U retku 33 zauzimamo pomoćni spremnik veličine 1024 okteta i potom

ulazimo u petlju. U petlji tražimo ulazni tok da napuni predani spremnik podatcima iz toka. Ako nije pročitan niti jedan oktet, ulazni tok je iscrpljen i izlazimo iz petlje. U suprotnom, u izlazni tok zapisujemo sadržaj spremnika počev od pozicije 0 i to samo onoliko okteta koliko je metoda čitanja upisala u spremnik.



Bilješka

Za više detalja o automatskom upravljanju resursima (te diskusiju da li resurse stvarati ulančavanjem konstruktora ili pojedinčano, te koje su opasnosti) čitatelj se upućuje na web-mjesto http://www.oracle.com/technetwork/articles/java/trywithresources-401775.html.

Drugi primjer koji ćemo prikazati koristi ZIP algoritam za kompresiju i dekompresiju podataka. Program je prikazan u primjeru 17.14.

Primjer 17.14. Sažimanje podataka

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.ByteArrayInputStream;
 4 import java.io.ByteArrayOutputStream;
 5 import java.io.IOException;
 6 import java.util.Arrays;
 7 import java.util.zip.ZipEntry;
 8 import java.util.zip.ZipInputStream;
 9 import java.util.zip.ZipOutputStream;
10
11 public class Kompresija {
12
13  public static void main(String[] args) throws IOException {
14
15   byte[] podatci = new byte[10240];
16
17   for(int i = 0; i < podatci.length; i++) {
18    podatci[i] = (byte)(i % 16);
19   }
20
21   byte[] komprimirano = null;
22   ByteArrayOutputStream bos = new ByteArrayOutputStream();
23   try (ByteArrayOutputStream bos1 = bos;
24    ZipOutputStream zos = new ZipOutputStream(bos1)) {
25    zos.putNextEntry(new ZipEntry("podatci.bin"));
26    zos.write(podatci);
27    zos.closeEntry();
28   }
29   komprimirano = bos.toByteArray();
30
31   System.out.format(
32    "Duljina izvornih podataka: %d.%n",
33    podatci.length);
34   System.out.format(
35    "Duljina komprimiranih podataka: %d.%n",
36    komprimirano.length);
37
38   byte[] dekomprimirano = null;
39   bos = new ByteArrayOutputStream();
40   try (ByteArrayOutputStream bos1 = bos;
41    ZipInputStream zis = new ZipInputStream(
```

```
42      new ByteArrayInputStream(komprimirano)
43     )) {
44
45     zis.getNextEntry();
46
47     byte[] spremnik = new byte[1024];
48     while(true) {
49      int procitano = zis.read(spremnik);
50      if(procitano < 1) break;
51      bos.write(spremnik, 0, procitano);
52     }
53
54    }
55    dekomprimirano = bos.toByteArray();
56
57    System.out.format(
58     "Duljina dekomprimiranih podataka: %d.%n",
59     dekomprimirano.length);
60
61    System.out.println(
62     "Podatci su jednaki: " +
63     Arrays.equals(podatci, dekomprimirano));
64   }
65
66 }
67
```

U programu najprije stvaramo polje s izvornim podatcima koje punimo u petlji. Radi se o 10240 okteta dobivenih cikličkim ponavljanjem podataka 0 do 15.

Kod u retcima 21-29 obavlja komprimiranje pripremljenih podataka. U retku 22 stvara se tok `bos` koji oktete zapisuje u relokatibilno polje koje kasnije može isporučiti pozivatelju; u retku 24 taj je tok dekoriran uporabom dekoratora `ZipOutputStream`. Ovaj dekorator omogućava stvaranje ZIP arhiva koje mogu sadržavati više datoteka. Stoga je u tok najprije potrebno upisati naziv datoteke koji će biti zapisan uz podatke koji slijede, što se radi pozivom metode `putNextEntry` (redak 25; kao ime smo odabrali `"podatci.bin"`). Potom u tok zapisujemo podatke kroz metodu `write` dekoratora (redak 26) i zatvaramo "logičku" datoteku s podatcima (redak 27). Izlaskom iz bloka `try` otvoreni tokovi se automatski zatvaraju. U retku 29 tok `bos` tražimo da nam vrati polje okteta koje odgovara svemu što je bilo zapisano u tok.

Slijede dva ispisa. U retku 31 ispisujemo veličinu izvornog polja s podatcima a u retku 34 veličinu polja koje čuva komprimirane podatke (zapravo oktete nastale ZIP arhive). Po pokretanju, programa, ispis koji nastaje zbog ove dvije naredbe bit će:

```
Duljina izvornih podataka: 10240.
Duljina komprimiranih podataka: 188.
```

Da smo kao konačni ponor okteta umjesto primjerka razreda `ByteArrayOutputStream` odabrali `FileOutputStream`, na disku bismo dobili stvorenu ZIP arhivu koja bi bila čitljiva bilo kojim klasičnim programom za rad s tom vrstom arhiva.

Retci 38-55 prikazuju dekompresiju podataka. Kao izvor podataka koristimo `ByteArrayInputStream` koji oktete čita iz polja koje čuva komprimirane podatke. Taj ulazni tok potom je dekoriran uporabom dekoratora `ZipInputStream` koji obavlja dekompresiju i preko kojega čitamo podatke (retci 41-43). Još stvaramo i pomoćni izlazni tok (redak 39) u koji ćemo upisati pročitane podatke kako bismo ih mogli opet dohvatiti kao polje okteta. U retku 45 ZIP-dekorator tražimo da se pozicionira na prvu datoteku arhive (i jedinu u našem slučaju) nakon čega imamo petlju koja obavlja klasično kopiranje: čita iz ulaznog toga i zapisuje u izlazni. Kad je kopiranje gotovo, izlazimo iz petlje i bloka `try` i dohvaćamo polje koje čuva zapisane oktete.

U retku 57 ispisujemo veličinu tog polja (ispisat će se 10240) a u retku 61 ispisujemo rezultat usporedbe izvornog polja s podatcima i polja dobivenog dekompresijom (ispisat će se `true`).

Na temelju ovih primjera ne biste više smijeli imati problema s konceptom tokova okteta te njihovom uporabom.

Znakovi i okteti

Prije no što se upoznamo sa znakovnim tokovima istaknut ćemo razliku između znaka i okteta. Znak (u Javi podatak modeliran tipom `char`) predstavlja simbol (veliko ili malo slovo, znamenke, interpunkciju, grafičke oznake, slova grčke abecede, kineske znakove, znakove drugih pisama te još niz drugih vrsta simbola). Razmislimo li malo, brzo ćemo doći do spoznaje da interesantnih simbola ima mnogo. Svakako puno više od 256 koliko ih je moguće prikazati jednim oktetom.

Kako bi se riješio taj problem, osmišljen je koncept *kodnih stranica*. Rane inačice kodnih stranica za kodiranje simbola koristile su jedan oktet. To su bile specifikacije koje su napravile odabir 256 "interesantnih" znakova i propisale koji se od tih odabranih simbola predstavlja kojom vrijednosti okteta. Jedna od najpoznatijih i najranijih kodnih stranica je US-ASCII kodna stranica. Osnovna inačica te kodne stranice specificirala je samo 128 simbola i njima pridružene vrijednosti okteta. U tih 128 simbola nalazila su se velika i mala slova, često korišteni znakovi interpunkcije, znamenke, podvlake i slično. Kasnije je napravljena proširena US-ASCII kodna stranica koja je odabrala još 128 grafičkih simbola čime je istrošen prostor od 256 brojeva koje može poprimiti jedan oktet. Slika 17.10 prikazuje simbole i dodijeljene 8-bitne numeričke vrijednosti okteta za osnovnu kodnu stranicu US-ASCII.

Slika 17.10. ASCII tablica

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Pogledajmo samo neke od primjera iz ove tablice. Okteti čije su vrijednosti 0 do 31 ne predstavljaju simbole koji imaju grafičku reprezentaciju (radi se o upravljačkim simbolima). Oktet vrijednosti 32 predstavlja razmaknicu, oktet vrijednosti 33 znak uskličnik, oktet vrijednosti 34 znak dvostrukog navodnika, itd. Pronaći ćete i velika i mala slova, znamenke te niz drugih simbola. Primjerice, slijed okteta 86, 111, 108, 105, 109, 32, 74, 97, 118, 117 bismo uporabom kodne stranice US-ASCII protumačili kao niz simbola *Volim Javu*.

Kako bismo tom istom kodnom stranicom u oktete pretvorili niz simbola *Čevapčić šeće šumom a ja želim džem*? Promotrimo li koje simbole predviđa kodna stranica US-ASCII, brzo ćemo uočiti problem. Nema dijakritičkih znakova. Problem možemo riješiti tako da simbole kodiramo uporabom kodne stranice Windows-1250 ili pak ISO-8859-2; te su kodne stranice među 256 simbola odabrale i dijakritičke znakove (a izbacile su neke druge znakove). Ovo ćemo demonstrirati jednostavnim primjerom.

Kodna stranica u Javi je predstavljena razredom `java.nio.charset.Charset`. Do objekata koji predstavljaju konkretne kodne stranice koje su podržane na platformi na kojoj je program pokrenut može se doći pozivom statičke metode `Charset.forName(ime)`. Razred `java.lang.String` nudi podršku za konverziju u polje okteta uporabom predane kodne stranice (metoda `String.getBytes(charset)`) te za stvaranje iz polja okteta i predane kodne stranice (konstruktor `String(data,charset)`).

```
 1 String poruka = "Čevapčić šeće šumom a ja želim džem";
 2
 3 Charset win1250 = Charset.forName("Windows-1250");
 4 Charset iso88592 = Charset.forName("ISO-8859-2");
 5
 6 byte[] okteti = poruka.getBytes(win1250);
 7 System.out.println(
 8  "Rekonstrukcija: " + new String(okteti, win1250)
 9 );
10
11 byte[] okteti2 = poruka.getBytes(iso88592);
12 System.out.println(
13  "Rekonstrukcija: " + new String(okteti2, iso88592)
14 );
15
16 System.out.println(
17  "Rekonstrukcija: " + new String(okteti2, win1250)
18 );
```

U retcima 3 i 4 dohvaćamo dvije kodne stranice. Potom u retku 6 tekst pohranjen u stringu `poruka` pretvaramo u oktete uporabom kodne stranice Windows-1250. U retku 7 iz tog polja okteta rekonstruiramo znakove (i poruku) što ispisujemo na zaslon. U retcima 11 i 12 ponavljamo postupak ali ovaj puta uporabom kodne stranice ISO-8859-2. Konačno, u retku 16 oktete dobivene pretvorbom znakova uz kodnu stranicu ISO-8859-2 pretvaramo u znakove uporabom kodne stranice Windows-1250 i to ispisujemo na zaslon. Ispis će biti:

```
Rekonstrukcija: Čevapčić šeće šumom a ja želim džem
Rekonstrukcija: Čevapčić šeće šumom a ja želim džem
Rekonstrukcija: Čevapčić ąeće ąumom a ja ľelim dľem
```

Prva dva ispisa su korektna: ako znakove pretvorimo u oktete i potom te oktete interpretiramo kao znakove uporabom *iste* kodne stranice, nema problema. Problem nastaje u trećem slučaju gdje smo znakove pretvorili u oktete uporabom jedne kodne stranice a njih natrag vratili u znakove uporabom druge kodne stranice. Interesantno je za primjetiti da je osnovni skup slova ostao očuvan: obje kodne stranice isti broj pridjeljuju znaku 'a', isti broj znaku 'b' itd. Razlika je brojevima koji se pridjeljuju slovima s dijakriticima: 'Č', 'č' i 'ć' obje kodne stranice kodiraju jednako ali 'š' i 'ž' ne.

Imamo li tekst koji koristi više od 256 različitih simbola ili pak tekst koji kombinira latinično i grčko pismo, ovakve kodne stranice koje definiraju kodiranje za neki podskup veličine 256 simbola nam ne mogu pomoći.

Posljednji ispis programa ilustrira još jedan problem: kada tekst spremimo na disk u datoteku, zapisujemo niz okteta. Prekopiramo li tu datoteku na drugo računalo i otvorimo u uređivaču teksta, može nas dočekati iznenađenje: tekst može izgledati nečitljivo jer se na tom računalu

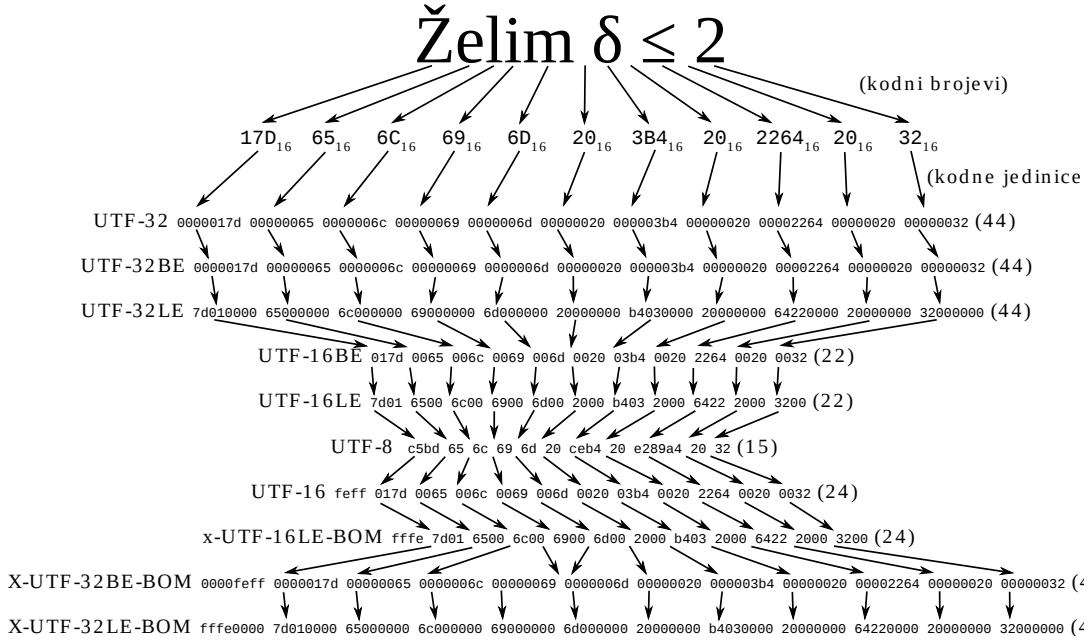
koristi neka druga kodna stranica. Ako se pitate možemo li znati kojom je kodnom stranicom nastao sadržaj datoteke, odgovor je: ne. Ta se informacija nigdje ne zapisuje uz tekstovne datoteke.

Opisane probleme u određenoj mjeri danas rješava nova kodna stranica: *Unicode*. Ta specifikacija predviđa rad sa skupom od preko milijun simbola (uključivo simbole svih češće korištenih pisama). Specifikacija svakom simbolu pridjeljuje jedinstven broj i ti se brojevi kreću u rasponu od 0_{16} do $10FFFF_{16}$ (brojevi su dani u heksadekadskoj bazi; raspon u dekadskom sustavu glasi 0 do 1114112). Svaki takav broj naziva se jednim brojem u kodu (engl. *code point*).

Bilo koji niz simbola (primjerice, neki tekst) konceptualno možemo zamisliti kao niz pripadnih kodnih brojeva (ovo, dakako, vrijedi i za prethodno spomenute kodne stranice). Pitanje koje treba riješiti je kako taj niz brojeva pretvoriti u skupine bitova u svrhu pohrane istih u memoriju računala ili u datoteku na disk. Niz bitova kojim se kodira jedan kodni broj naziva se *kodna jedinica* (engl. *code unit*). Kod svih prethodno spomenutih kodnih stranica ovo je bilo jednostavno: svaka je kodna stranica bila u stanju radi isključivo s podskupom od odabranih 256 simbola pa se pri postupku kodiranja koristila reprezentacija koja je za svaku kodnu jedinicu generirala jedan oktet (8 bitova). Time smo dobili da je broj potrošenih okteta odgovarao broju znakova u znakovnom nizu (uz ograničenje da smo mogli koristiti samo one znakove koje kodna stranica definira).

Sa specifikacijom Unicode situacija je nešto složenija jer se vrijednosti kodnih brojeva kreću u puno širem intervalu. Iz tog razloga, specifikacija definira nekoliko različitih načina generiranja kodnih jedinica za kodne brojeve. Slika 17.11 to lijepo ilustrira.

Slika 17.11. Znakovni niz i Unicode



Razmatramo tekst "Želim δ ≤ 2" koji se sastoji od 11 znakova: 'Ž', 'e', 'l', 'i', 'm', razmak, 'δ', razmak, '≤', razmak, '2'. Kodni brojevi pridruženi svakom od tih znakova na slici su prikazani u prvom redu uporabom heksadekadskog brojevnog sustava. Ovaj brojevni sustav prikladan je zbog činjenice da svaka znamenka u računalu predstavlja grupu od 4 bita pa dvije znamenke čine (odnosno za pohranu trebaju) jedan oktet. Znakovi 'e', 'l', 'i', 'm', razmak te '2' definirani su u grupi znakova *Basic Latin*, znak 'Ž' u grupi znakova *Latin Extended-A*, znak 'δ' u grupi znakova *Greek and Coptic* te znak '≤' u grupi znakova *Mathematical Operators*[1]

[1]Znakovi koji su definirani u spomenutim grupama (kao i sve preostale grupe) mogu se pogledati na http://jrgraphix.net/research/unicode_blocks.php. Tablice po pismima (i drugim podjelama) mogu se pogledati na http://unicode.org/charts/.

Unicode specifikacija definira niz načina na koje se za kodne brojeve generiraju kodne jedinice (koje se pohranjuju u memoriju ili u datoteku). U Javi, implementacija svake od tih specifikacija dostupne su kao jedna kodna stranica odnosno kao primjerak razreda `Charset`.

UTF-32	Svaki se kodni broj zapisuje uporabom 32 bita (4 okteta) koristeći poredak Big Endian.
UTF-32BE	Svaki se kodni broj zapisuje uporabom 32 bita (4 okteta) koristeći poredak Big Endian.
UTF-32LE	Svaki se kodni broj zapisuje uporabom 32 bita (4 okteta) koristeći poredak Little Endian.
UTF-16BE	Koristi se varijabilno kodiranje koje najvećem dijelu kodnih brojeva dodjeljuje 16-bitne kodne jedinice, no dio 16-bitnih brojeva koristi se za uvođenje escape-sekvenci kojima se stvaraju i dulje kodne jedinice (ali za rijeđe korištene simbole). Koristi se poredak Big Endian.
UTF-16LE	Koristi se varijabilno kodiranje koje najvećem dijelu kodnih brojeva dodjeljuje 16-bitne kodne jedinice, no dio 16-bitnih brojeva koristi se za uvođenje escape-sekvenci kojima se stvaraju i dulje kodne jedinice (ali za rijeđe korištene simbole). Koristi se poredak Little Endian.
UTF-8	Koristi se varijabilno kodiranje koje najčešće korištene kodne brojeve predstavlja 8-bitnim kodnim jedinicama, no postoji mehanizam escapeanja kojim je moguće tvoriti 16-bitne kodne jedinice (za rjeđe korištene simbole), 32-bitne kodne jedinice (za još rjeđe korištene simbole) itd.
UTF-16	Kao UTF-16BE, samo što na početku dodaje 16-bita (takozvani BOM) čija vrijednost govori o kojem se načinu kodiranja radi.
x-UTF-16LE-BOM	Kao UTF-16LE, samo što na početku dodaje 16-bita (takozvani BOM) čija vrijednost govori o kojem se načinu kodiranja radi.
x-UTF-32BE-BOM	Kao UTF-32BE, samo što na početku dodaje 32-bita (takozvani BOM) čija vrijednost govori o kojem se načinu kodiranja radi.
x-UTF-32LE-BOM	Kao UTF-32LE, samo što na početku dodaje 32-bita (takozvani BOM) čija vrijednost govori o kojem se načinu kodiranja radi.

Pogledajte u prethodnom primjeru rezultat nastao primjenom kodne stranice UTF-8. Slova poput 'e' i 'l' te znamenka '2' kodirana su samo jednim oktetom. Za kodiranje slova 'Ž' i 'δ' potrošena su po 2 okteta a za kodiranje simbola '≤' čak četiri okteta.

Kako bi se omogućilo da se uz bitove koji predstavljaju kodne jedinice za neki tekst prenese i informacija o korištenoj kodnoj stranici, specifikacija Unicode dozvoljava uporabu specifičnog niza bitova koji se postavljaju na početku (prije bitova kodnih jedinica) i koji određuju korištenu kodnu stranicu. Pronađite te oktete (takozvane markere BOM) za posljednja četiri kodiranja i uvjerite se da su svi različiti. Ovo nažalost danas još uvijek ne podržavaju svi programi koji rade s tekstom pa često taj dio pogrešno protumače kao znakove.

Pokažimo sada nekoliko isječaka koda koji rade s kodnim stranicama. Sve kodne stranice koje su dostupne Javinom virtualnom stroju moguće je ispisati za sljedeći način.

```
1 Charset.availableCharsets()
2   .forEach((k,v)->System.out.println(v.name()));
```

Isječak koda koji za tekst s prethodne slike ispisuje oktete nastale kodiranjem uz navedene kodne stranice dan je u nastavku.

```
1 public static void main(String[] args) {
2
3  String poruka = "Želim \u03b4 \u2264 2";
4
5  Charset[] charsets = {
6   StandardCharsets.UTF_8,
7   StandardCharsets.UTF_16,
8   StandardCharsets.UTF_16BE,
9   StandardCharsets.UTF_16LE,
10  Charset.forName("UTF-32"),
11  Charset.forName("UTF-32BE"),
12  Charset.forName("UTF-32LE"),
13  Charset.forName("x-UTF-16LE-BOM"),
14  Charset.forName("x-UTF-32BE-BOM"),
15  Charset.forName("x-UTF-32LE-BOM")
16 };
17 for(Charset cs : charsets) {
18  ispisi(cs, poruka);
19 }
20 }
21
22 private static void ispisi(Charset cs, String poruka) {
23  System.out.println("Način kodiranja: " + cs.name());
24  byte[] octets = poruka.getBytes(cs);
25  StringBuilder sb = new StringBuilder();
26  for(byte c : octets) {
27   sb.append(String.format("%02x", (int)c & 0xFF));
28  }
29  System.out.println("  " + sb);
30  System.out.println("  Ukupni broj okteta:" + octets.length);
31
32 }
```

Uočite da je dostupan razred `java.nio.charset.StandardCharsets` koji kao javne statičke konstante nudi reference na nekoliko često korištenih kodnih stranica.

Razred `java.nio.charset.Charset` omogućava dohvat objekata koji rade kodiranje znakova u oktete te dekodiranje okteta u znakove i upravo tu funkcionalnog za nas koristi metoda `String.getBytes(charset)` te konstruktor `String(byte[], charset)`, kako mi ne bismo morali direktno raditi s njima. Međutim, zgodno je upoznati se i izravno s tim mogućnostima pa ćemo u nastavku dati primjere.

Razred `CharsetEncoder` omogućava kodiranje znakova u kodne jedinice (slijed okteta). Radi se o objektu koji se može koristiti više puta pa ga pri svakoj uporabi treba resetirati, potom jednom ili više puta pozivati metodu koja dostavlja znakove koje treba kodirati i kad je sve gotovo treba pozvati metodu `flush` koja će završiti postupak. Razred nudi i metodu koja odjednom prima kompletan niz znakova i koja interno radi sva tri navedena koraka te vraća konačan slijed okteta. Umjesto s poljima, ovaj razred radi sa spremnicima znakova i okteta koji su modelirani razredima `java.nio.ByteBuffer` i `java.nio.CharBuffer`. Primjerak spremnika može se iz polja dobiti pozivom statičke metode `wrap`; za prebacivanje sadržaja spremnika u polje potrebno je pripremiti polje prikladne veličine i pozvati nad psremnikom metodu `get` koja će obaviti prijenos podataka iz spremnika u polje. Evo ilustrativnog primjera.

```
1 char[] znakovi = "Želim δ ≤ 2".toCharArray();
2
3 Charset charset = StandardCharsets.UTF_8;
4 CharsetEncoder encoder = charset.newEncoder();
5
6 CharBuffer cb = CharBuffer.wrap(znakovi);
```

```
 7 ByteBuffer bb = encoder.encode(cb);
 8
 9 byte[] okteti = new byte[bb.remaining()];
10 bb.get(okteti);
11
12 // sada je polje okteta napunjeno oktetima...
```

U retku 1 pripremili smo polje znakova koje želimo kodirati u niz okteta kako bismo oktete pohranili u datoteku (ili poslali preko mreže, ili ...). Redak 3 bira kodnu stranicu a u retku 4 tražimo od kodne stranice novi primjerak objekta za kodiranje. U retku 6 polje znakova zamatamo u spremnik koji u retku 7 predajemo objektu za kodiranje; on obavlja kodiranje te stvara i vraća spremnik okteta u koji je pohranjen slijed kodnih jedinica za svaki od znakova koje smo predali. U retku 9 stvaramo polje okteta potrebne veličine te u retku 10 tražimo spremnik da oktete prepiše u to polje. Nakon toga, polje je popunjeno i možemo ga koristiti.

Primjer koji iz polja okteta obavlja dekodiranje u znakove prikazan je u nastavku.

```
 1 byte[] okteti = dohvatiOktete();
 2
 3 Charset charset = StandardCharsets.UTF_8;
 4 CharsetDecoder decoder = charset.newDecoder();
 5
 6 ByteBuffer bb = ByteBuffer.wrap(okteti);
 7 CharBuffer cb = decoder.decode(bb);
 8
 9 char[] znakovi = new char[cb.remaining()];
10 cb.get(znakovi);
11
12 // sada je polje znakova napunjeno znakovima...
```

Ako ne želite koristiti funkcionalnost inkrementalnog kodiranja odnosno dekodiranja već želite napraviti točno ovo što smo pokazali, prethodni kod je moguće napisati kraće jer razred `Charset` nudi i izravno metode `encode(charbuffer)` i `decode(bytebuffer)` koje će odraditi posao dohvata objekta za kodiranje/dekodiranje i odgovarajući poziv. U tom slučaju prednost je što će ova metoda samo pri prvom pozivu unutar trenutne dretve stvoriti potreban objekt za kodiranje/dekodiranje koji će koristiti i pri kasnijim pozivima.

Simboli, Java i Unicode

Sjećate li se metode `String.length()`? Što ona vraća? Ako je vaš odgovor "broj znakova u stringu" (u smislu broj slova tj. općenito simbola) - razmislite još jednom jer je taj odgovor pogrešan. Do ovog trenutka bilo je prerano ulaziti u takve detalje ali jednom moramo i to obraditi; u ovom trenutku biste za to trebali biti spremni.

Sjećate li se da smo na početku upoznavanja s Javom prošli kroz primitivne tipove podataka te da smo istaknuli da u Javi oktet (`byte`) i znak (`char`) nisu isto? Rekli smo da specifikacija znak tretira kao 16-bitnu (odnosno dvooktetnu) vrijednost. Također, istaknuli smo da Java nativno radi sa specifikacijom Unicode. Nobroj simbola koje predviđa Unicode mjeri se u milijunima dok različitih 16-bitnih vrijednosti ima samo 65536; očito, u Javi ne može vrijediti jednakost da je jedan simbol jednak jednom znaku - i nije.

Vratimo se ponovno na ideju specifikacije Unicode. Svakom simbolu pridružen je jedinstven kodni broj. Java interno za pohranu tih kodnih brojeva u memoriju koristi sljedeće pravilo. Najveći dio kodnih brojeva koji su iz raspona 0_{16} do $FFFF_{16}$ pohranjuje se u 16 bitova odnosno u jedan znak (podatak tipa `char`). Za pohranu preostalih kodnih brojeva Java koristi slijed od dva znaka i sve kodne brojeve koji spadaju u taj raspon naziva *dodatnim kodnim brojevima* (engl. *supplementary code points*). Razred `java.lang.Character` nudi dvije vrste metoda: metode koje rade nad znakovima te metode koje rade nad kodnim brojevima (u Javi - podatcima

tipa `int`). Evo prvog primjera. Metoda `Character.isSupplementaryCodePoint` prima kodni broj i provjerava pripada li on u grupu dodatnih kodnih brojeva.

```
1 System.out.println(Character.isSupplementaryCodePoint(0x00041));
2 System.out.println(Character.isSupplementaryCodePoint(0x20021));
```

Prvi redak će ispisati `false` (radi se o broju koji predstavlja veliko slovo A) a drugi `true` (radi se o broju koji predstavlja jedan simbol kineskog pisma).

Koliko nam znakova treba za pohranu simbola čiji znamo kodni broj možemo doznati pozivom metode `Character.charCount`:

```
1 System.out.println(Character.charCount(0x41));
2 System.out.println(Character.charCount(0x20021));
```

Ispis će biti:

```
1
2
```

Za kodne brojeve koji pripadaju u grupu dodatnih kodnih brojeva uporabom metoda `Character.highSurrogate` i `Character.lowSurrogate` možemo dohvatiti znakove koji kao par predstavljaju taj jedan simbol.

```
1 char hi = Character.highSurrogate(0x20021);
2 char lo = Character.lowSurrogate(0x20021);
3 System.out.printf("%04X %04X%n", (int)hi, (int)lo);
```

Ispis će biti:

```
D840 DC21
```

Razmotrimo sada sljedeći primjer koji kreće od polja znakova gdje smo svaki od znakova definirali unicode-konstantom.

```
1 char[] znakovi4 = {
2   '\u005a', '\u006e', '\u0061', '\u006b',
3   '\u0020', '\ud840', '\udc21', '\u0021'
4 };
5
6 System.out.println(znakovi4.length);
7 System.out.println(
8  Character.codePointCount(znakovi4, 0, znakovi4.length)
9 );
10
11 String str = new String(znakovi4);
12 System.out.println(str);
13 System.out.println(str.length());
14 System.out.println(str.codePointCount(0, str.length()));
```

Ispis će biti:

```
8
7
Znak 𠀡!
8
7
```

Ispis veličine polja rezultirat će ispisom "8": doista, polje ima 8 znakova. Sljedeći redak zahtjeva da se u tom polju izbroji koliko to polje predstavlja simbola: rezultat je 7.

Nakon toga kod iz polja stvara string te ga ispisuje: uočite, ispis doista sadrži 7 simbola. Posljednja dva retka daju odgovor na početno pitanje ovog podpoglavlja: metoda `String.length` vraća broj znakova koji definiraj sadržaj stringa što je 8 i veće je od broja simbola koje predstavlja string. Tek pozivom metode `String.codePointCount` dobivamo korektnu informaciju: broj simbola koje predstavlja ovaj string je 7.

Tokovi znakova

Tokovi znakova konceptualno su slični tokovima okteta: jedinu razliku čini tip elementarnih podataka koji prolaze tokom i koji su kod tokova okteta bili okteti (podatci tipa `byte`) dok su kod tokova znakova to znakovi (podatci tipa `char`).

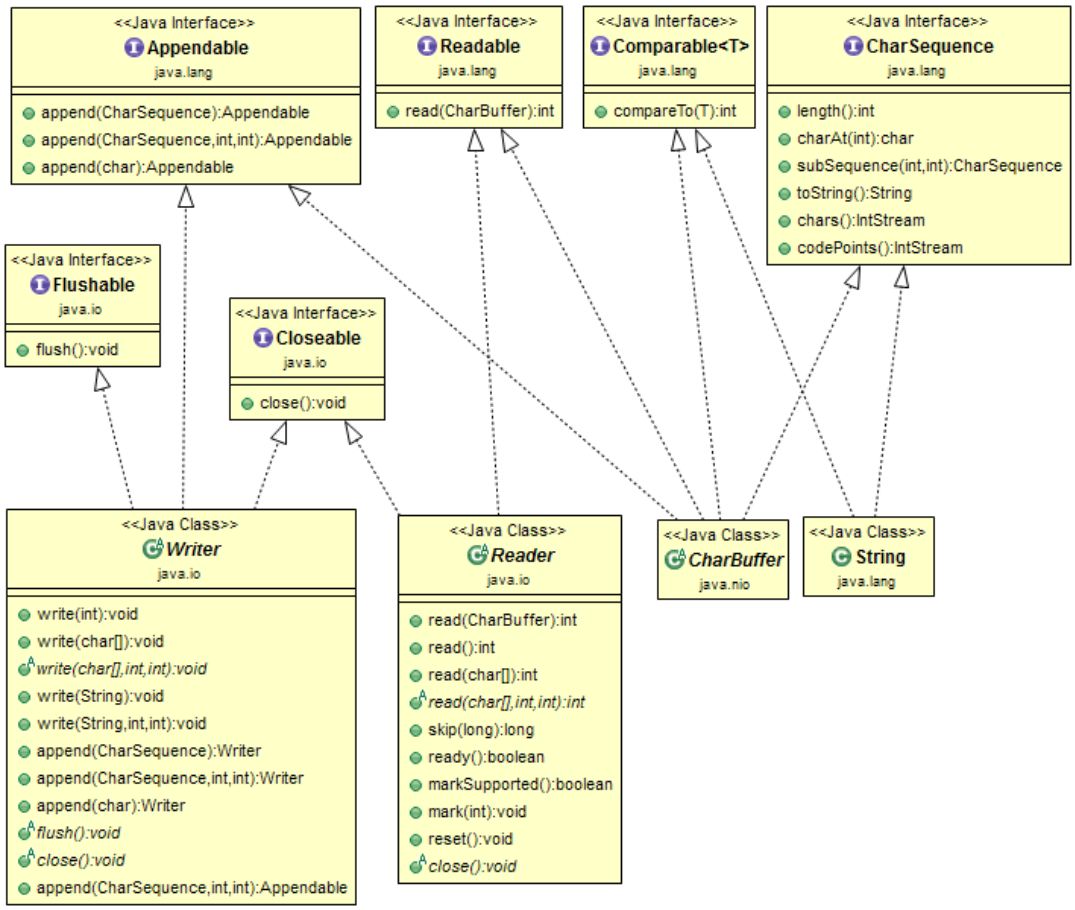
U osnovnom paketu java.lang od Jave inačice 1.5 dodana su dva sučelja koja nemaju direktne veze s tokovima znakova već modeliraju općeniti izvor znakova te općeniti ponor znakova: radi se o sučeljima `java.lang.Readable` (izvor znakova) te `java.lang.Appendable` (ponor). Ova sučelja prikazana su u nastavku.

```
 1 public interface Readable {
 2     public int read(java.nio.CharBuffer cb) throws IOException;
 3 }
 4
 5 public interface Appendable {
 6     Appendable append(CharSequence csq) throws IOException;
 7     Appendable append(CharSequence csq, int start, int end)
 8                       throws IOException;
 9     Appendable append(char c) throws IOException;
10 }
```

Sučelje `Readable` deklarira samo jednu metodu: `read` koja prima referencu na spremnik znakova i u njega upisuje određen broj znakova iz izvora koji modelira; metoda vraća broj znakova koji su upisani u spremnik ili -1 ako je izvor iscrpljen pa nije bilo moguće dohvatiti i upisati niti jedan znak.

Sučelje `Appendable` deklarira tri metode koje specificiraju znakove koje treba zapisati u modelirani ponor znakova. Prve dvije metode znakove koje treba zapisati u ponor nude kroz objekt tipa `CharSequence` (prva metoda nudi čitav sadržaj tog objekta a druga samo dio sadržaja tog objekta); treća metoda izravno prima jedan znak koji treba upisati u ponor. Sučelje `CharSequence` predstavlja model nepromjenjivog niza znakova te nudi metode za dohvat broja znakova, znaka na određenoj poziciji, stvaranje podniza te pretvorbu u objekt tipa `String`. Razred `String` jedan je od razreda koji implementiraju ovo sučelje. Spremnik znakova također implementira ovo sučelje, kao i sučelja `Comparable`, `Readable` i `Appendable`. Odnose spomenutih razreda i sučelja prikazuje slika 17.12. Radi preglednosti, sa slike su uklonjeni strukturni elementi razreda `String` i `CharBuffer`.

Slika 17.12. Spremnici znakova, stringovi, izvori i ponori, tokovi



U ovu strukturu prirodno se uklapaju i ulazni tokovi znakova (vrsta izvora) te izlazni tokovi znakova (vrsta ponora), kako ih definira paket java.io, a što je također prikazano na prethodnoj slici. Ulazni tokovi znakova modelirani su apstraktnim razredom `java.io.Reader` a izlazni tokovi znakova apstraktnim razredom `java.io.Writer`. Primjer 17.15 prikazuje odabrane detalje razreda `java.io.Reader` a primjer 17.16 odabrane detalje razreda `java.io.Writer`. Pri tome treba napomenuti da su veće metode zamijenjene s "..." te da prikazani kod nije maksimalno efikasan već prikazuje načelnu izvedbu. Primjerice, Oracleova implementacija razreda `java.io.Writer` umjesto neprestanog stvaranja novih znakovnih polja (što je prikazano primjeru 17.16) koristi interno privatno polje koje zauzme samo jednom i dalje koristi po potrebi. Razlog zbog kojeg prikazujemo navedene isječke koda jest kako bismo pogledali koji je podskup metoda apstraktan te kako su sve ostale metode riješene delegiranjem općenitijim metodama ili pak ovim apstraktnim metodama.

Primjer 17.15. Apstraktni razred `java.io.Reader`.

```
 1 package java.io;
 2
 3 public abstract class Reader implements Readable, Closeable {
 4     protected Object lock;
 5
 6     protected Reader() {
 7         this.lock = this;
 8     }
 9
10     protected Reader(Object lock) {
11         if (lock == null) {
12             throw new NullPointerException();
```

```
13          }
14          this.lock = lock;
15      }
16
17      public int read(java.nio.CharBuffer target)
18          throws IOException {
19          ...
20      }
21
22      public int read() throws IOException {
23          char cb[] = new char[1];
24          if (read(cb, 0, 1) == -1)
25              return -1;
26          else
27              return cb[0];
28      }
29
30      public int read(char cbuf[]) throws IOException {
31          return read(cbuf, 0, cbuf.length);
32      }
33
34      abstract public int read(char cbuf[], int off, int len)
35          throws IOException;
36
37      public long skip(long n) throws IOException {
38          ...
39      }
40
41      public boolean ready() throws IOException {
42          return false;
43      }
44
45      public boolean markSupported() {
46          return false;
47      }
48
49      public void mark(int readAheadLimit) throws IOException {
50          throw new IOException("mark() not supported");
51      }
52
53      public void reset() throws IOException {
54          throw new IOException("reset() not supported");
55      }
56
57      abstract public void close() throws IOException;
58  }
59
```

Primjer 17.16. Apstraktni razred `java.io.Writer`.

```
1  package java.io;
2
3  public abstract class Writer
4      implements Appendable, Closeable, Flushable {
5
6      protected Object lock;
7
8      protected Writer() {
```

```
 9          this.lock = this;
10      }
11
12      protected Writer(Object lock) {
13          if (lock == null) {
14              throw new NullPointerException();
15          }
16          this.lock = lock;
17      }
18
19      public void write(int c) throws IOException {
20          synchronized (lock) {
21              write(new char[] {c}, 0, 1);
22          }
23      }
24
25      public void write(char cbuf[]) throws IOException {
26          write(cbuf, 0, cbuf.length);
27      }
28
29      abstract public void write(char cbuf[], int off, int len)
30        throws IOException;
31
32      public void write(String str) throws IOException {
33          write(str, 0, str.length());
34      }
35
36      public void write(String str, int off, int len)
37        throws IOException {
38          synchronized (lock) {
39              char cbuf[] = new char[len];
40              str.getChars(off, (off + len), cbuf, 0);
41              write(cbuf, 0, len);
42          }
43      }
44
45      public Writer append(CharSequence csq) throws IOException {
46          if (csq == null)
47              write("null");
48          else
49              write(csq.toString());
50          return this;
51      }
52
53      public Writer append(CharSequence csq, int start, int end)
54        throws IOException {
55          CharSequence cs = (csq == null ? "null" : csq);
56          write(cs.subSequence(start, end).toString());
57          return this;
58      }
59
60      public Writer append(char c) throws IOException {
61          write(c);
62          return this;
63      }
64
65      abstract public void flush() throws IOException;
66
```

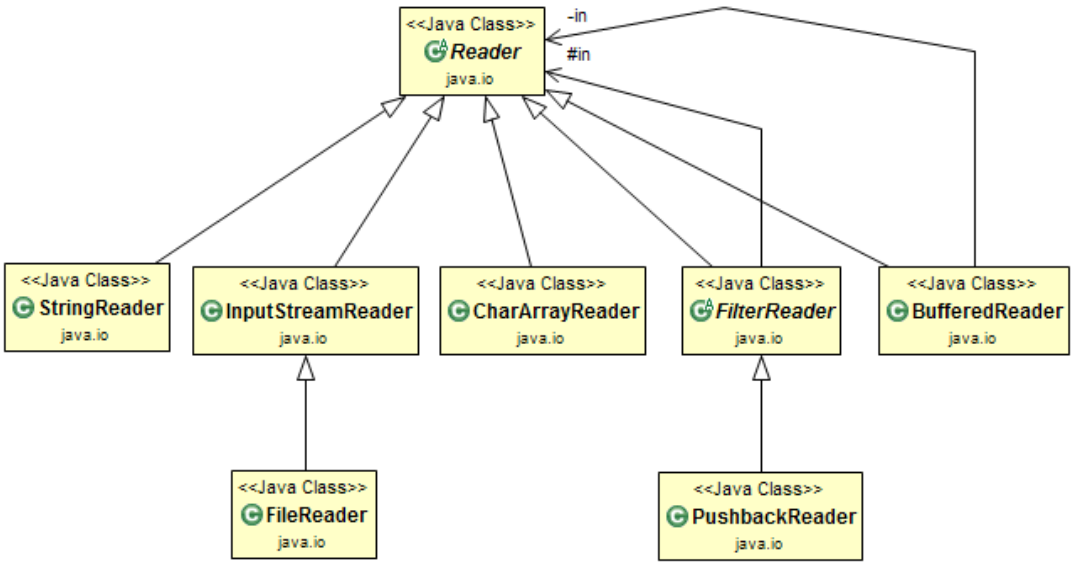
```
67      abstract public void close() throws IOException;
68  }
69
```

Razred `java.io.Reader` definira metodu za dohvat jednog znaka (koja vraća znak ukalupljen u `int` ili pak `-1` ako je tok iscrpljen) te metode za dohvat više znakova (inačice koje primaju polje znakova ili pak spremnik znakova). Kao i kod tokova okteta, postoji podrška za (opcionalno) vraćanje toka u neko prethodno zapamćeno stanje (`markSupported`, `mark` te `reset`) i zatvaranje toka (`close`). Novitet je metoda `ready` čija je zadaća provjeriti ima li u toku taj tren dostupan barem jedan znak za čitanje čime se garantira da pozivatelj neće biti zablokiran ako zatraži čitanje znaka. Napomenimo da se metoda u određenom smislu ponaša kao implikacija: ako vrati `true`, pozivatelj sigurno neće biti zablokiran pri čitanju sljedećeg znaka; međutim, ako vrati `false`, to ne znači da će sigurno biti zablokiran.

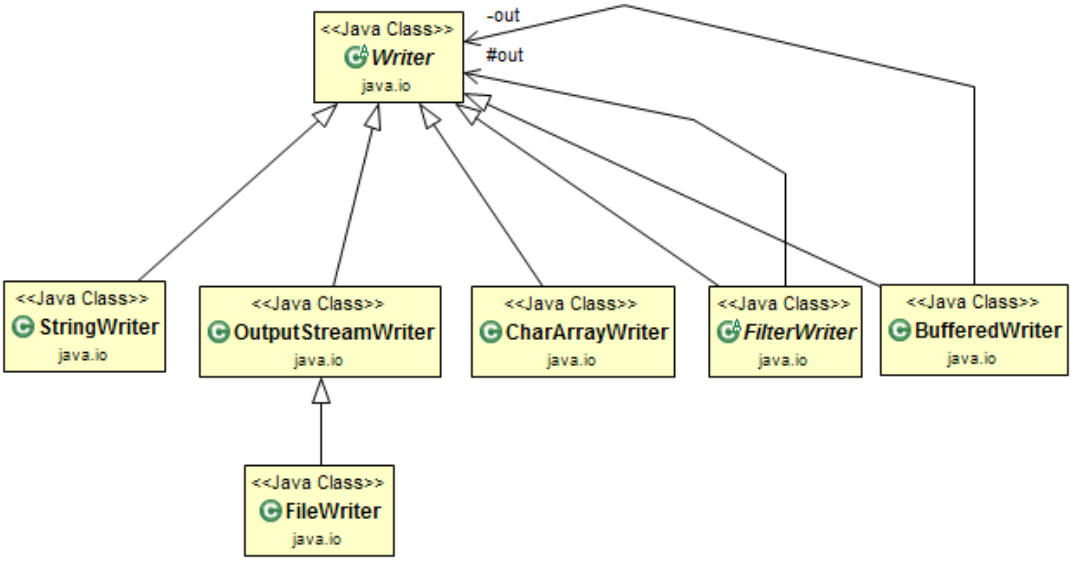
Razred `java.io.Writer` definira metode `write` i `append` koje omogućavaju zapisivanje pojedinačnih znakova (kao argument primaju jedan znak) te metode koje omogućavaju zadavanje više znakova koje treba zapisati (inačice metode `write` koje primaju polje znakova, inačice metode `write` koje primaju `String` te inačice metode `append` koje primaju `CharSequence`). Uz navedeno, razred definira i metodu za zatvaranje toka `close` te metodu kojom se od toka traži da obavi sve eventualno odgođene operacije zapisivanja (`flush`).

Iz vršnih razreda `java.io.Reader` i `java.io.Writer` kojima su modelirani apstraktni ulazni i izlazni tokovi znakova, a u skladu s oblikovnim obrascem *Dekorator*, izvedene su porodice razreda koji predstavljaju konkretne izvore i ponore te niz pomoćnih dekoratora (poput dekoratora koji dodaje međuspremnik: `BufferedReader` i `BufferedWriter`). Ilustrativan podskup razreda za svaku od ove dvije strukture prikazan je na slikama 17.13 i 17.14.

Slika 17.13. Dio podstabla od `java.io.Reader`



Slika 17.14. Dio podstabla od `java.io.Writer`



Kod tokova znakova dekoratora nema mnogo. Dapače, pogledate li slike 17.13 i 17.14 uočit ćete da postoji apstraktni dekorator (`FilterReader` i `FilterWriter`) i samo jedan konkretan dekorator (`PushbackReader`). Međutim, na obje slike postoji još jedan razred koji je istovremeno drugi apstraktni i konkretan dekorator: radi se o razredima `BufferedReader` i `BufferedWriter` koji omogućavaju uporabu međuspremnika znakova za povećavanje učinkovitosti operacija nad znakovnim tokom.

Ono što su kod tokova okteta bili razredi `ByteArrayInputStream` i `ByteArrayOutputStream`, kod tokova znakova su razredi `CharArrayReader` i `CharArrayWriter`. Razredi `StringReader` i `StringWriter` omogućavaju dobivanje ulaznog toka znakova čiji je sadržaj određen predanim stringom te izlaznog toka znakova koji pamti sve u njega zapisane znakove te nudi mogućnost dohvata stringa čiji će sadržaj odgovarati tim zapisanim znakovima. Evo primjera.

```
 1 Reader r = new StringReader("Java je najbolja!");
 2 Writer w = new StringWriter();
 3
 4 while(true) {
 5  int podatak = r.read();
 6  if(podatak == -1) break;
 7  char znak = (char)podatak;
 8  znak = Character.toUpperCase(znak);
 9  w.write(znak);
10 }
11
12 w.close();
13 r.close();
14
15 String sadrzaj = w.toString();
16
17 System.out.println(sadrzaj);
```

Primjer će rezultirati ispisom:

```
JAVA JE NAJBOLJA!
```

Uočite: retci 4 do 10 čine općenitu metodu kopiranja znak po znak iz ulaznog toka znakova u izlazni tok znakova. Učinkovitiju inačicu mogli bismo napisati uporabom pomoćnog polja zna-

kova koje bi omogućilo da odjednom čitamo više znakova i pročitane znakove jednim pozivom zapišemo u izlazni tok znakova.

Čitanje znakova iz datoteke te pisanje znakova u datoteku izravno je moguće uporabom razreda `FileReader` te `FileWriter`. Međutim, uporaba tih razreda *se ne preporuča*.



Važno

Nemojte za čitanje i pisanje znakova u/iz datoteke koristiti razrede `FileReader` i `FileWriter`.

Problem s navedenim razredima leži u činjenici da su datoteke slijedovi okteta - ne znakova - te sav API koji radi sa znakovima i datotekama mora znati koju kodnu stranicu koristiti za pretvorbu. Java kao platforma ne definira pretpostavljenu kodnu stranicu već će ovi razredi koristiti kodnu stranicu koja je u trenutku pokretanja bila podešena na operacijskom sustavu (ili koja je eksplicitno zadana prilikom pokretanja programa). Vidite li zašto je to loše? Isti program na različitim operacijskim sustavima može koristiti različite kodne stranice. I ne samo to, program pokrenut na istom operacijskom sustavu danas i sutra može koristiti različite kodne stranice ako je korisnik u međuvremenu promijenio lokalizacijske postavke operacijskog sustava. To će pak za posljedicu imati pogrešno tumačenje prethodno zapisanih podataka i gubitak interoperabilnosti (ako radimo s raspodijeljenom aplikacijom koja između računala razmjenjuje datoteke).

Da bi se riješio ovaj problem, napravljena su i dva mostna razreda: `InputStreamReader` i `OutputStreamWriter`. Razred `InputStreamReader` je ulazni tok znakova; on u konstruktoru očekuje dva argumenta: referencu na ulazni tok okteta iz kojeg će čitati oktete (pa to može biti primjerice datoteka) te referencu na kodnu stranicu koju treba koristiti pri dekodiranju okteta u znakove. Razred `OutputStreamWriter` je izlazni tok znakova koji u konstruktoru također očekuje dva argumenta: referencu na izlazni tok okteta u koji će zapisivati oktete (pa to može biti primjerice datoteka) te referencu na kodnu stranicu koju treba koristiti pri kodiranju znakova koje korisnik zapisuje u oktete koje će ova implementacija poslati dalje u izlazni tok okteta.

Uporabom ovih razreda programer je unaprijed odgovoran za odluku o konkretnoj kodnoj stranici koju želi koristiti, i tako piše program. Tako napisan program neće imati problema ako korisnik odluči promijeniti lokalizacijske postavke operacijskog sustava ili pak s razmjenom podataka između više računala. U današnje doba preporuka je koristiti kodnu stranicu UTF-8. U nastavku je dan primjer čitanja odnosno pisanja znakova iz/u datoteku.

```
 1 char[] poruka = "Šeće čevapčić šumicom.".toCharArray();
 2
 3 try(Writer w = new BufferedWriter(
 4  new OutputStreamWriter(
 5   new BufferedOutputStream(
 6    new FileOutputStream("datoteka.txt")
 7   ),StandardCharsets.UTF_8
 8  ))) {
 9  w.write(poruka);
10 }
11
12 StringBuilder sb = new StringBuilder();
13 try(Reader r = new BufferedReader(
14  new InputStreamReader(
15   new BufferedInputStream(
16    new FileInputStream("datoteka.txt")
17   ),StandardCharsets.UTF_8
18  ))) {
19
20  char[] buf = new char[1024];
21  while(true) {
22   int procitano = r.read(buf);
```

```
23    if(procitano==-1) break;
24    sb.append(buf, 0, procitano);
25   }
26 }
27 String procitanaPoruka = sb.toString();
28 System.out.println(procitanaPoruka);
```

Primjer najprije definira polje znakova koje je potrebno zapisati u datoteku (redak 1). Potom u retcima 3-10 uporabom bloka `try` uz automatsko upravljanje resursima stvara izlazni tok okteta, njega omata u izlazni tok okteta s međuspremnikom, njega pak omata u `OutputStreamWriter` (kojemu kao kodnu stranicu predaje UTF-8) a njega omata u izlazni tok znakova s međuspremnikom. U tako stvoreni izlazni tok znakova program upisuje znakove iz pripremljenog polja a automatsko upravljanje resursima se pobrine za zatvaranje toka.

Retci 12-27 ilustriraju čitanje znakova iz datoteke uporabom kodne stranice UTF-8. Program u petlji malo po malo čita znakove i zapisuje ih u `StringBuilder`; kada više nema raspoloživih znakova čitanje se prekida i konstruira se konačni string na temelju sadržaja `StringBuilder`-a. Redak 28 taj string ispisuje na zaslon. Uočite da samo umjesto razreda `StringBuilder` mogli koristiti i `StringWriter` ili pak `CharArrayWriter` (kojim bismo mogli dohvatiti čitav pročitani sadržaj kao jedno polje znakova).

Razred `BufferedReader` uz uvođenje međuspremnika proširuje i sučelje ulaznog toka znakova dodavanjem (i implementiranjem) metode `String readLine()`. Metoda čita znakove iz toka sve dok ne pročita i terminator retka (ASCII 10, ASCII 13 ili pak slijed ASCII 13, ASCII 10) i vraća slijed pročitanih znakova (bez oznake za kraj retka) kao string. Ako je tok iscrpljen, metoda će vratiti `null`. Da bismo imali pristup ovoj metodi, referencu ne smijemo ukalupiti u osnovni tip `Reader` jer iz tog tipa ova metoda nije vidljiva.

Ako za rad s objektima datotečnog sustava koristimo sučelja iz novog paketa java.nio.file (konkretno, `Path` a ne `File` iz paketa java.io), tada tokove možemo otvarati uporabom statičkih metoda razreda `java.nio.file.Files` čiji su prototipovi prikazani u nastavku.

```
1 public static InputStream newInputStream(Path path,
2     OpenOption... options) throws IOException;
3
4 public static OutputStream newOutputStream(Path path,
5     OpenOption... options) throws IOException;
6
7 public static BufferedReader newBufferedReader(Path path,
8     Charset cs) throws IOException;
9
10 public static BufferedReader newBufferedReader(Path path)
11     throws IOException;
12
13 public static BufferedWriter newBufferedWriter(Path path,
14     Charset cs, OpenOption... options) throws IOException;
15
16 public static BufferedWriter newBufferedWriter(Path path,
17     OpenOption... options) throws IOException;
```

Metode za stvaranje tokova znakova koje ne primaju kodnu stranicu uvijek koriste istu kodnu stranicu (UTF-8) čime iste nemaju probleme koje ima implementacija razreda `FileReader` i `FileWriter`.

Spomenimo da razred `java.nio.file.Files` nudi statičke metode kojima je cjelokupni sadržaj datoteke moguće jednim pozivom dohvatiti kao polje okteta ili pak kao listu redaka. Njihovi prototipovi prikazani su u nastavku.

```
1 public static byte[] readAllBytes(Path path)
2     throws IOException;
```

```
3
4 public static List<String> readAllLines(Path path, Charset cs)
5     throws IOException;
6
7 public static List<String> readAllLines(Path path)
8     throws IOException;
```

Primjer uporabe

U nastavku ovog poglavlja razmotrit ćemo opet jedan konkretan zadatak. Najprije ćemo dati *"sjedi i nakucaj da radi"*-rješenje koje ćemo potom pokušati preoblikovati u nešto kvalitetnije.

Zadatak je sljedeći. Na raspolaganju su tri tekstovne datoteke. Primjer prve datoteke (`data/studenti.txt`) dan je u nastavku:

```
0012345678 Perić Perica
0023456789 Anić Anica
0034567890 Marić Martina
0045678901 Stavić Jasen
0056789012 Dadić Ivona
```

Datoteka čuva informacije o studentima (JMBAG, prezime te ime; jedan redak - jedan student). Primjer druge datoteke (`data/predmeti.txt`) slijedi.

```
12345 Programiranje u Javi
23456 Mreže računala
34567 Mrežna administracija
45678 Baze podataka
```

Datoteka čuva informacije o kolegijima (šifra te naziv; jedan redak - jedan kolegij). Konačno, primjer treće datoteke (`data/upisi.txt`) dan je u nastavku:

```
0012345678 12345
0023456789 12345
0034567890 12345
0012345678 23456
0023456789 23456
0056789012 23456
0012345678 34567
0023456789 34567
0056789012 34567
0023456789 45678
```

Datoteka čuva informacije o upisima (JMBAG studenta te šifra kolegija; jedan redak - jedan upis).

Sve su datoteke stvorene uporabom UTF-8 kodne stranice. Unutar retka, pojedini su elementi razdvojeni znakom TAB.

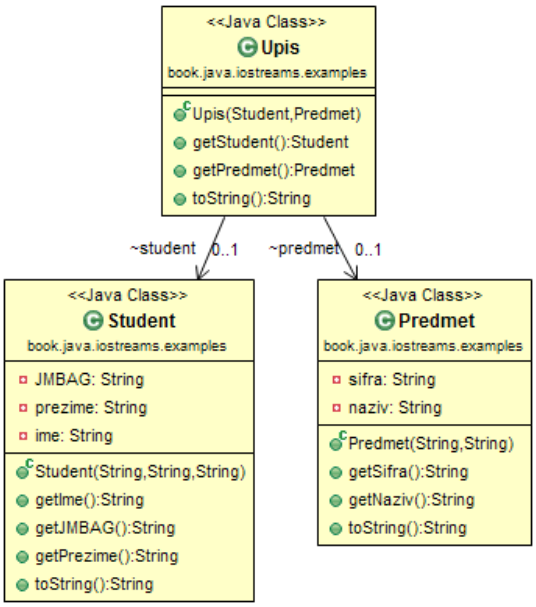
Naš je zadatak napisati program koji će na temelju sadržaja tih datoteka na zaslon za svaki upis ispisati potpunu informaciju; evo primjer željenog ispisa.

```
Perica Perić (0012345678) je upisao Programiranje u Javi (12345)
Anica Anić (0023456789) je upisao Programiranje u Javi (12345)
```

Model podataka

U rješavanje problema krenut ćemo modeliranjem domene u kojoj radimo. Radimo s tri entiteta: studentom, kolegijem i upisom. Svaki od ta tri entiteta modelirat ćemo razredom, čime dolazimo do modela prikazanog na slici 17.15.

Slika 17.15. Model domene



Implementaciju ovih razreda prikazuju primjeri 17.17, 17.18 i 17.19.

Primjer 17.17. Razred `Student`.

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.Objects;
 4
 5 public class Student {
 6
 7  private String JMBAG;
 8  private String prezime;
 9  private String ime;
10
11  public Student(String jMBAG, String prezime, String ime) {
12   super();
13   JMBAG = Objects.requireNonNull(
14    jMBAG, "JMBAG ne može biti null.");
15   this.prezime = Objects.requireNonNull(
16    prezime, "Prezime ne može biti null.");
17   this.ime = Objects.requireNonNull(
18    ime, "Ime ne može biti null.");
19  }
20
21  public String getIme() {
22   return ime;
23  }
24
25  public String getJMBAG() {
26   return JMBAG;
27  }
28
29  public String getPrezime() {
30   return prezime;
31  }
32
```

```
33  @Override
34  public String toString() {
35   return ime+" "+prezime+" ("+JMBAG+")";
36  }
37 }
38
```

Primjer 17.18. Razred `Predmet`.

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.Objects;
 4
 5 public class Predmet {
 6
 7  private String sifra;
 8  private String naziv;
 9
10  public Predmet(String sifra, String naziv) {
11   super();
12   this.sifra = Objects.requireNonNull(
13    sifra, "Šifra ne može biti null.");
14   this.naziv = Objects.requireNonNull(
15    naziv, "Naziv ne može biti null.");
16  }
17
18  public String getSifra() {
19   return sifra;
20  }
21
22  public String getNaziv() {
23   return naziv;
24  }
25
26  @Override
27  public String toString() {
28   return naziv+" ("+sifra+")";
29  }
30 }
31
```

Primjer 17.19. Razred `Upis`.

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.Objects;
 4
 5 public class Upis {
 6
 7  Student student;
 8  Predmet predmet;
 9
10  public Upis(Student student, Predmet predmet) {
11   super();
12   this.student = Objects.requireNonNull(
13    student, "Student ne može biti null.");
14   this.predmet = Objects.requireNonNull(
15     predmet, "Predmet ne može biti null.");
```

```
16  }
17
18  public Student getStudent() {
19   return student;
20  }
21
22  public Predmet getPredmet() {
23   return predmet;
24  }
25
26  @Override
27  public String toString() {
28   return student.toString() + " -> " + predmet.toString();
29  }
30 }
31
```

Prva inačica rješenja

Sada smo spremni prikazati prvu inačicu rješenja. Rješenje ćemo oblikovati na sljedeći način.

1. Pripremit ćemo mapu studenata i nju ćemo napuniti temeljem sadržaja datoteke `data/studenti.txt`. Ključ će biti JMBAG.
2. Pripremit ćemo mapu predmeta i nju ćemo napuniti temeljem sadržaja datoteke `data/predmeti.txt`. Ključ će biti šifra.
3. Pripremit ćemo listu upisa i nju ćemo napuniti temeljem sadržaja datoteke `data/upisi.txt` te prethodno stvorenih mapa.

Prilikom obrade datoteka prazne retke ćemo ignorirati a za retke u kojima pronađemo pogrešku izazvat ćemo iznimku (primjerice, ako u retku očekujemo tri elementa a pronađemo ih pet). Takvo rješenje prikazano je u primjeru 17.20.

Primjer 17.20. Program `Upisi`.

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.BufferedInputStream;
 4 import java.io.BufferedReader;
 5 import java.io.IOException;
 6 import java.io.InputStreamReader;
 7 import java.nio.charset.StandardCharsets;
 8 import java.nio.file.Files;
 9 import java.nio.file.Paths;
10 import java.nio.file.StandardOpenOption;
11 import java.util.ArrayList;
12 import java.util.HashMap;
13 import java.util.List;
14 import java.util.Map;
15
16 public class Upisi {
17
18  public static void main(String[] args) throws IOException {
19   Map<String, Student> studenti = ucitajStudente();
20   Map<String, Predmet> predmeti = ucitajPredmete();
21   List<Upis> upisi = ucitajUpise(studenti, predmeti);
22
23   for(Upis u : upisi) {
24    System.out.println(u.student +" je upisao "+u.predmet);
```

```
25    }
26   }
27
28   private static Map<String, Student> ucitajStudente()
29    throws IOException {
30
31    Map<String, Student> mapa = new HashMap<>();
32    try(BufferedReader reader =
33     new BufferedReader(
34      new InputStreamReader(
35       new BufferedInputStream(
36        Files.newInputStream(
37         Paths.get("data/studenti.txt"), StandardOpenOption.READ
38        )
39       ), StandardCharsets.UTF_8)) ) {
40     while(true) {
41      String redak = reader.readLine();
42      if(redak==null) break;
43      redak = redak.trim();
44      if(redak.isEmpty()) continue;
45      String[] elementi = redak.split("\t");
46      if(elementi.length != 3) {
47       throw new IOException(
48        "Pronaden redak s pogrešnim brojem stupaca.");
49      }
50      mapa.put(
51       elementi[0],
52       new Student(elementi[0], elementi[1], elementi[2]));
53     }
54    }
55    return mapa;
56   }
57
58   private static Map<String, Predmet> ucitajPredmete()
59    throws IOException {
60
61    Map<String, Predmet> mapa = new HashMap<>();
62    try(BufferedReader reader =
63     new BufferedReader(
64      new InputStreamReader(
65       new BufferedInputStream(
66        Files.newInputStream(
67         Paths.get("data/predmeti.txt"), StandardOpenOption.READ
68        )
69       ), StandardCharsets.UTF_8)) ) {
70     while(true) {
71      String redak = reader.readLine();
72      if(redak==null) break;
73      redak = redak.trim();
74      if(redak.isEmpty()) continue;
75      String[] elementi = redak.split("\t");
76      if(elementi.length != 2) {
77       throw new IOException(
78        "Pronaden redak s pogrešnim brojem stupaca.");
79      }
80      mapa.put(
81       elementi[0],
82       new Predmet(elementi[0], elementi[1])
```

```
 83      );
 84     }
 85    }
 86    return mapa;
 87   }
 88
 89   private static List<Upis> ucitajUpise(
 90    Map<String, Student> studenti,
 91    Map<String, Predmet> predmeti)
 92    throws IOException {
 93
 94    List<Upis> lista = new ArrayList<>();
 95    try(BufferedReader reader =
 96     new BufferedReader(
 97      new InputStreamReader(
 98       new BufferedInputStream(
 99        Files.newInputStream(
100         Paths.get("data/upisi.txt"), StandardOpenOption.READ
101        )
102       ), StandardCharsets.UTF_8)) ) {
103     while(true) {
104      String redak = reader.readLine();
105      if(redak==null) break;
106      redak = redak.trim();
107      if(redak.isEmpty()) continue;
108      String[] elementi = redak.split("\t");
109      if(elementi.length != 2) {
110       throw new IOException(
111        "Pronađen redak s pogrešnim brojem stupaca.");
112      }
113      Student student = studenti.get(elementi[0]);
114      Predmet predmet = predmeti.get(elementi[1]);
115      if(student==null) {
116       throw new IOException(
117        "Pronađen nepostojeći student ("+
118        elementi[0]+") u upisima!");
119      }
120      if(predmet==null) {
121       throw new IOException(
122        "Pronađen nepostojeći predmet ("+
123        elementi[1]+") u upisima!");
124      }
125      lista.add(new Upis(student, predmet));
126     }
127    }
128    return lista;
129   }
130
131  }
132
```

Proučite dobro prikazano rješenje. Kakve su karakteristike tog rješenja? Prvo i osnovno - rješenje ima jedno odlično svojstvo, a to je da radi. Međutim, ako ćemo ikada trebati bilo što promijeniti u tom rješenju, jao si ga nama (primjerice, želimo promjeniti da separator nije TAB već znak ljestvi; gdje to sve moramo mijenjati?). Već na prvi pogled možemo uočiti masovnu redundanciju koda. Imamo tri pomoćne metoda i svaka je gotovo pa kopija preostalih dviju. Pažljivom analizom tih metoda uočit ćemo da između njih postoje razlike navedene u nastavku.

1. *Tip rezultata koji metoda vraća.* Jedna vraća mapu studenata, druga mapu predmeta a treća listu upisa.

2. *Očekivani broj elemenata u retku.* Dvije metode očekuju dva elementa; treća očekuje tri elementa.

3. *Način obrade elemenata retka.* Što točno treba napraviti s elementima retka? Različite metode stvaraju primjerke različitih razreda i ubacuju ih u različite kolekcije.

Osim različitosti, gledano sa strane pozivatelja, sve tri metode imaju i jedno zajedničko svojstvo: proizvode i vraćaju objekt koji je zadanog tipa (dva puta mapa, jednom lista).

Sva ostala logika u ovim je metodama ista: otvaranje datoteka, odluka o kodnoj stranici, odluka o uporabi međuspremnika, čitanje radaka, razdvajanje elemenata retka, provjera broja elemenata u retku te odluka o dojavi pogreške i tekstu koji treba generirati, zatvaranje datoteke, povrat generiranog rezultata.

S obzirom da imamo toliko toga zajedničkoga, prirodno bi bilo pokušati taj kod napisati jednom i potom ga višestruko koristiti. Koji nam oblikovni obrazac nudi jedno moguće prikladno rješenje? Razmislite!

Druga inačica rješenja

Ako ste se kod posljednjeg pitanja sjetili *Strategije*, bravo! Za potrebe preoblikovanja rješenja strategijom moramo najprije utvrditi što sve varira od primjera do primjera i zatim to modelirati sučeljem. Zatim ćemo napisati jednu općenitu metodu koja će još dobiti i referencu na primjerak strategije i delegacijom pozivati potreban kod. Sve elemente koji ulaze u sučelje strategije maloprije smo utvrdili. Stoga ćemo definirati apstraktno sučelje strategije kako je prikazano u primjeru 17.21.

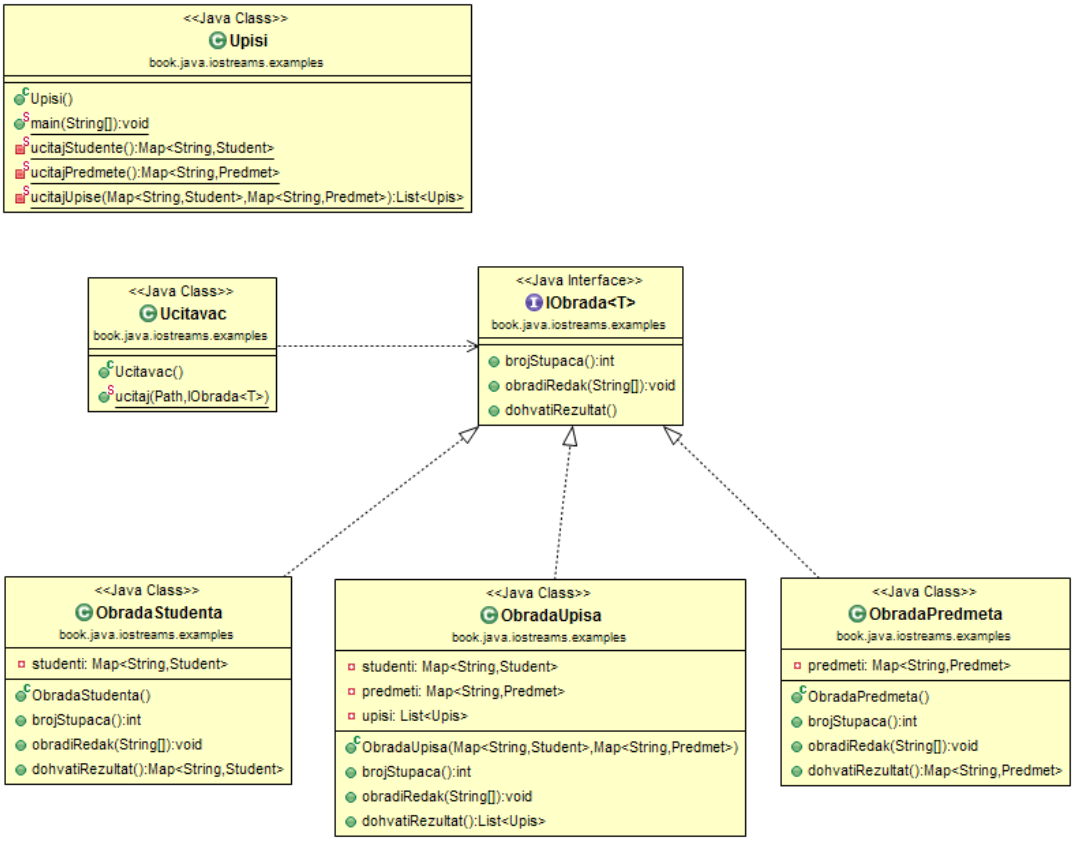
Primjer 17.21. Apstraktno sučelje strategije (`IObrada`)

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.IOException;
 4
 5 public interface IObrada<T> {
 6
 7  int brojStupaca();
 8
 9  void obradiRedak(String[] elems) throws IOException;
10
11  T dohvatiRezultat();
12
13 }
14
```

Sučelje je parametrizirano tako da može raditi s bilo kojim tipom "proizvoda" koji gradimo. Nudi se metode `brojStupaca` kako bi generički kod imao informaciju koliko elemenata svaki redak mora sadržavati, potom metoda `obradiRedak` koja prima polje elemenata (generička metoda će ga pripremiti) i obrađuje ga (generičku metodu ne zanima kako), i konačno metoda `dohvatiRezultat` koja dohvaća izgrađeni objekt koji na kraju treba vratiti pozivatelju.

Pripremit ćemo i tri implementacije ovog sučelja: jednu za obradu redaka s informacijama o studentima, drugu za obradu redaka s informacijama o predmetima te treću za obradu redaka s informacijama o upisima. Potom ćemo pripremiti pomoćni razred `Ucitavac` sa statičkom metodom `T ucitaj(Path,IObrada)` koja predstavlja implementaciju spomenute generičke metode. Naš glavni program smješten u razred `Upisi` koristit će ovu funkcionalnost. Dijagram razreda koji opisuje rješenje prikazan je na slici 17.16.

Slika 17.16. Dijagram razreda rješenja strategijom



Programski kod svih razreda i sučelja ovog rješenja dan je u primjeru 17.22.

Primjer 17.22. Programski kod rješenja strategijom

```
1 package book.java.iostreams.examples;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class ObradaStudenta
7  implements IObrada<Map<String,Student>> {
8
9  private Map<String, Student> studenti = new HashMap<>();
10
11  @Override
12  public int brojStupaca() {
13   return 3;
14  }
15
16  @Override
17  public void obradiRedak(String[] elems) {
18   studenti.put(
19    elems[0],
20    new Student(elems[0], elems[1], elems[2])
21   );
22  }
23
24  @Override
25  public Map<String,Student> dohvatiRezultat() {
```

```
26   return studenti;
27  }
28
29 }
30
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5
 6 public class ObradaPredmeta
 7  implements IObrada<Map<String,Predmet>> {
 8
 9  private Map<String, Predmet> predmeti = new HashMap<>();
10
11  @Override
12  public int brojStupaca() {
13   return 2;
14  }
15
16  @Override
17  public void obradiRedak(String[] elems) {
18   predmeti.put(elems[0], new Predmet(elems[0], elems[1]));
19  }
20
21  @Override
22  public Map<String,Predmet> dohvatiRezultat() {
23   return predmeti;
24  }
25
26 }
27
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.IOException;
 4 import java.util.ArrayList;
 5 import java.util.List;
 6 import java.util.Map;
 7
 8 public class ObradaUpisa implements IObrada<List<Upis>> {
 9
10  private Map<String, Student> studenti;
11  private Map<String, Predmet> predmeti;
12  private List<Upis> upisi = new ArrayList<>();
13
14  public ObradaUpisa(Map<String, Student> studenti,
15    Map<String, Predmet> predmeti) {
16   super();
17   this.studenti = studenti;
18   this.predmeti = predmeti;
19  }
20
21  @Override
22  public int brojStupaca() {
23   return 2;
24  }
```

```
25
26  @Override
27  public void obradiRedak(String[] elems) throws IOException {
28   Student student = studenti.get(elems[0]);
29   Predmet predmet = predmeti.get(elems[1]);
30   if(student==null) {
31    throw new IOException("Pronađen nepostojeći student ("+
32     elems[0]+") u upisima!");
33   }
34   if(predmet==null) {
35    throw new IOException("Pronađen nepostojeći predmet ("+
36     elems[1]+") u upisima!");
37   }
38   upisi.add(new Upis(student, predmet));
39  }
40
41  @Override
42  public List<Upis> dohvatiRezultat() {
43   return upisi;
44  }
45
46 }
47
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.BufferedInputStream;
 4 import java.io.BufferedReader;
 5 import java.io.IOException;
 6 import java.io.InputStreamReader;
 7 import java.nio.charset.StandardCharsets;
 8 import java.nio.file.Files;
 9 import java.nio.file.Path;
10 import java.nio.file.StandardOpenOption;
11 import java.util.Objects;
12
13 public class Ucitavac {
14
15  public static <T> T ucitaj(Path path, IObrada<T> obrada)
16   throws IOException {
17
18   Objects.requireNonNull(obrada, "Obrada ne smije biti null.");
19   int brojStupaca = obrada.brojStupaca();
20
21   try(BufferedReader reader =
22    new BufferedReader(
23     new InputStreamReader(
24      new BufferedInputStream(
25       Files.newInputStream(
26        path, StandardOpenOption.READ
27       )
28      ), StandardCharsets.UTF_8)) ) {
29    while(true) {
30     String redak = reader.readLine();
31     if(redak==null) break;
32     redak = redak.trim();
33     if(redak.isEmpty()) continue;
34     String[] elementi = redak.split("\t");
```

```
35     if(elementi.length != brojStupaca) {
36      throw new IOException(
37       "Pronaden redak s pogrešnim brojem stupaca."
38      );
39     }
40     obrada.obradiRedak(elementi);
41    }
42   }
43   return obrada.dohvatiRezultat();
44  }
45 }
46
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.IOException;
 4 import java.nio.file.Paths;
 5 import java.util.List;
 6 import java.util.Map;
 7
 8 public class Upisi {
 9
10  public static void main(String[] args) throws IOException {
11   Map<String, Student> studenti = ucitajStudente();
12   Map<String, Predmet> predmeti = ucitajPredmete();
13   List<Upis> upisi = ucitajUpise(studenti, predmeti);
14
15   for(Upis u : upisi) {
16    System.out.println(u.student +" je upisao "+u.predmet);
17   }
18  }
19
20  private static Map<String, Student> ucitajStudente()
21   throws IOException {
22
23   return Ucitavac.ucitaj(
24    Paths.get("data/studenti.txt"),
25    new ObradaStudenta());
26  }
27
28  private static Map<String, Predmet> ucitajPredmete()
29   throws IOException {
30
31   return Ucitavac.ucitaj(
32    Paths.get("data/predmeti.txt"),
33    new ObradaPredmeta());
34  }
35
36  private static List<Upis> ucitajUpise(
37   Map<String, Student> studenti, Map<String, Predmet> predmeti)
38   throws IOException {
39
40   return Ucitavac.ucitaj(
41    Paths.get("data/upisi.txt"),
42    new ObradaUpisa(studenti, predmeti));
43  }
44
45 }
```

```
46
```

Umjesto pisanja "punokrvnih" razreda, implementacije strategija mogli smo ponuditi i u samom razredu `Upisi` uporabom anonimnih razreda.

Sjećate li se pitanja koje smo postavili nakon inicijalne implementacije? Je li se odgovor promijenio? Konkretno, razmotrite sljedeće potencijalne promjene koje se mogu javiti. Pronađite mjesto/mjesta u kodu u kojima bismo trebali mijenjati kod kako bismo implementirali tu promjenu.

- Želimo promjeniti da se kao separator u datotekama koristi znak ljestvi, a ne znak TAB.
- U datoteci sa studentima došlo je do promjene pa je novi redosljed elemenata prezime, ime, JMBAG.
- Želimo podržati komentare u datotekama. Sve što počinje s // treba tretirati kao komentar i preskočiti.

Svaka od opisanih promjena sada zahtjeva točno jedno mjesto intervencije. Za prvu točku moramo intervenirati u metodu `Ucitavac.ucitaj`, za drugu promjenu moramo intervenirati u konkretnu strategiju `ObradaStudenta` a za treću točku moramo intervenirati ponovno u metodu `Ucitavac.ucitaj`.



Bilješka

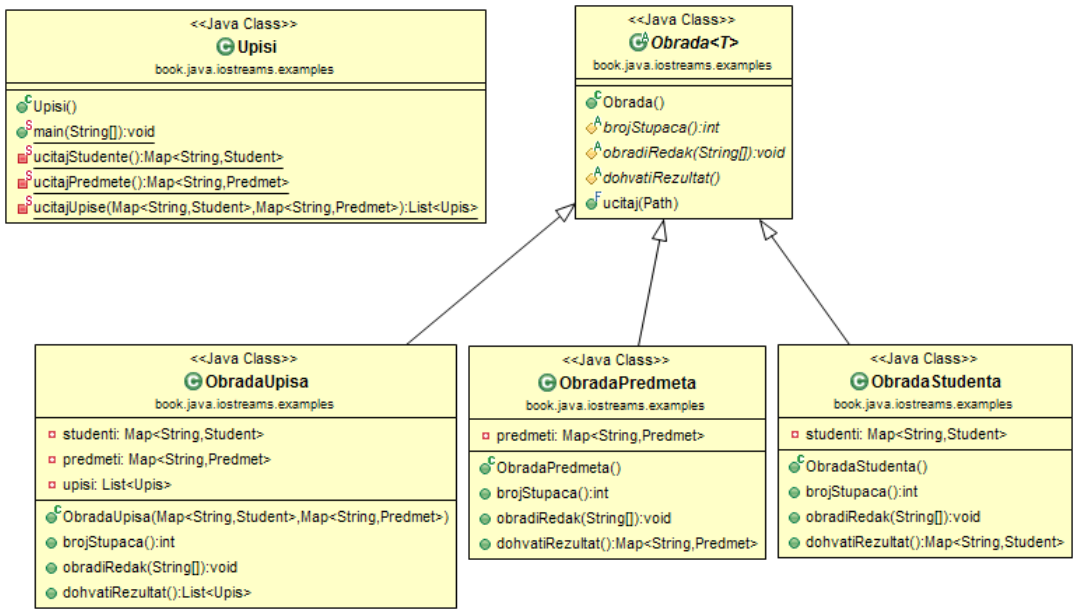
Kako biste dodatno preoblikovali kod da podržite različite vrste komentara u različitim datotekama? Primjerice, da komentari u datoteci sa studentima započinju znakom %; da u datoteci s predmetima čitav redak koji bilo gdje sadrži slijed REM bude komentar a da u datoteci u upisima dio retka nakon // bude komentar (ali što je bilo ispred toga nije komentar)?

Treća inačica rješenja

Druga inačica rješenja za injekciju koda koristila je delegaciju: rješenje koje lijepo opisuje oblikovni obrazac Strategija. Delegacija, međutim, nije jedini način kako u stari kod injektirati novi kod - moguće rješenje je i nasljeđivanje. Ovu ideju lijepo formulira oblikovni obrazac *Okvirna meotda*. Ideja je sljedeća. Napišimo razred koji implementira općeniti algoritam obrade te deklarira apstraktne metode koje definiraju primitive čija implementacija nije dostupna u tom razredu (primitivi ovdje će biti iste metode koje smo deklarirali u sučelju strategije). Potom se oslonimo na nasljeđivanje i tražimo da korisnik za svaku različitu obradu napiše novi razred koji nasljeđuje početni i nudi implementacije apstraktnih primitiva. Naš klijent i dalje će moći s primjercima konkretnih obrada komunicirati kroz sučelje početnog razreda tako da krajnji korisnik čak i ne mora biti svjestan konkretne obrade (osim ako je on sam ujedno i ne stvara).

Predložena organizacija koda prikazana je na slici 17.17.

Slika 17.17. Dijagram razreda rješenja okvirnom metodom



Pogledajte malo pažljivije razred `Obrada` na slici 17.17. Metode `brojStupaca`, `obradiRedak` i `dohvatiRezultat` su apstraktne i zaštićene metode; u ovom razredu za njih nemamo smislenu implementaciju (stoga su apstraktne). Te su metode zaštićene jer nisu namijenjene pozivanju od strane klijenta - njihova namjena je omogućiti mehanizam preko kojeg će novi naknadno napisani razred na to mjesto podmetnuti svoj kod; da bi to bilo moguće, taj razred mora naslijediti trenutni razred i ponuditi implementaciju tih metoda pa je prikladan modifikator vidljivosti `protected`. Ovu vrstu metoda nazivamo *primitivima* (u kontekstu oblikovnog obrasca Okvirna metoda).

Razred `Obrada` definira i jednu konkretnu metodu: `T ucitaj(Path)` koja ima implementaciju. Ta metoda sadrži "okvirni algoritam": kod je napisan i može ga se prevesti ali kako dio svoje funkcionalnosti ostvaruje pozivanjem primitiva, što on točno radi nije jasno dok nije poznato što točno primitivi rade. Zbog toga se ova metoda (u kontekstu oblikovnog obrasca Okvirna metoda) naziva *okvirnom metodom*. Ova je metoda ujedno označena i kao `final` kako je razred koji nasljeđuje trenutni ne bi mogao nadjačati - to je okvirna metoda koja sadrži implementaciju algoritma i ona nije predviđena za nadjačavanje.

Konačno, oblikovni obrazac Okvirna metoda predviđa još jednu vrstu sudionika a to su *nadomjesne metode* (engl. *hooks*). Nadomjesne metode su poput primitiva ali imaju početnu implementaciju. Primjerice, kada bismo analizom utvrdili da većina datoteka koje obrađujemo ima dva stupca, umjesto da metodu `brojStupaca` deklariramo kao apstraktnu, mogli bismo joj dati implementaciju: `int brojStupaca() { return 2; }`. Time bismo iz najvećeg dijela razreda koji nasljeđuju ovaj razred mogli samo preuzeti ponuđenu implementaciju a tek one obrade koje rade s brojem stupaca koji je različit od 2 morale bi metodu nadjačati i ponuditi drugu implementaciju. Takve metode stoga nazivamo nadomjesnim metodama.

Programski kod svih razreda i sučelja ovog rješenja dan je u primjeru 17.23.

Primjer 17.23. Programski kod rješenja okvirnom metodom

```
1 package book.java.iostreams.examples;
2
3 import java.io.BufferedInputStream;
4 import java.io.BufferedReader;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.nio.charset.StandardCharsets;
```

```
 8 import java.nio.file.Files;
 9 import java.nio.file.Path;
10 import java.nio.file.StandardOpenOption;
11
12 public abstract class Obrada<T> {
13
14  protected abstract int brojStupaca();
15
16  protected abstract void obradiRedak(String[] elems)
17    throws IOException;
18
19  protected abstract T dohvatiRezultat();
20
21  public final T ucitaj(Path path) throws IOException {
22   int brojStupaca = brojStupaca();
23
24   try(BufferedReader reader =
25    new BufferedReader(
26     new InputStreamReader(
27      new BufferedInputStream(
28       Files.newInputStream(
29        path, StandardOpenOption.READ
30       )
31      ), StandardCharsets.UTF_8)) ) {
32    while(true) {
33     String redak = reader.readLine();
34     if(redak==null) break;
35     redak = redak.trim();
36     if(redak.isEmpty()) continue;
37     String[] elementi = redak.split("\t");
38     if(elementi.length != brojStupaca) {
39      throw new IOException(
40       "Pronađen redak s pogrešnim brojem stupaca."
41      );
42     }
43     obradiRedak(elementi);
44    }
45   }
46   return dohvatiRezultat();
47  }
48 }
49
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5
 6 public class ObradaStudenta
 7   extends Obrada<Map<String,Student>> {
 8
 9  private Map<String, Student> studenti = new HashMap<>();
10
11  @Override
12  public int brojStupaca() {
13   return 3;
14  }
15
```

```
16  @Override
17  public void obradiRedak(String[] elems) {
18   studenti.put(
19    elems[0],
20    new Student(elems[0], elems[1], elems[2])
21   );
22  }
23
24  @Override
25  public Map<String,Student> dohvatiRezultat() {
26   return studenti;
27  }
28
29 }
30
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5
 6 public class ObradaPredmeta
 7   extends Obrada<Map<String,Predmet>> {
 8
 9  private Map<String, Predmet> predmeti = new HashMap<>();
10
11  @Override
12  public int brojStupaca() {
13   return 2;
14  }
15
16  @Override
17  public void obradiRedak(String[] elems) {
18   predmeti.put(elems[0], new Predmet(elems[0], elems[1]));
19  }
20
21  @Override
22  public Map<String,Predmet> dohvatiRezultat() {
23   return predmeti;
24  }
25
26 }
27
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.IOException;
 4 import java.util.ArrayList;
 5 import java.util.List;
 6 import java.util.Map;
 7
 8 public class ObradaUpisa extends Obrada<List<Upis>> {
 9
10  private Map<String, Student> studenti;
11  private Map<String, Predmet> predmeti;
12  private List<Upis> upisi = new ArrayList<>();
13
14  public ObradaUpisa(Map<String, Student> studenti,
```

```
15     Map<String, Predmet> predmeti) {
16    super();
17    this.studenti = studenti;
18    this.predmeti = predmeti;
19   }
20
21   @Override
22   public int brojStupaca() {
23    return 2;
24   }
25
26   @Override
27   public void obradiRedak(String[] elems) throws IOException {
28    Student student = studenti.get(elems[0]);
29    Predmet predmet = predmeti.get(elems[1]);
30    if(student==null) {
31     throw new IOException("Pronaden nepostojeći student ("+
32      elems[0]+") u upisima!");
33    }
34    if(predmet==null) {
35     throw new IOException("Pronaden nepostojeći predmet ("+
36      elems[1]+") u upisima!");
37    }
38    upisi.add(new Upis(student, predmet));
39   }
40
41   @Override
42   public List<Upis> dohvatiRezultat() {
43    return upisi;
44   }
45
46 }
47
```

```
 1 package book.java.iostreams.examples;
 2
 3 import java.io.IOException;
 4 import java.nio.file.Paths;
 5 import java.util.List;
 6 import java.util.Map;
 7
 8 public class Upisi {
 9
10  public static void main(String[] args) throws IOException {
11   Map<String, Student> studenti = ucitajStudente();
12   Map<String, Predmet> predmeti = ucitajPredmete();
13   List<Upis> upisi = ucitajUpise(studenti, predmeti);
14
15   for(Upis u : upisi) {
16    System.out.println(u.student +" je upisao "+u.predmet);
17   }
18  }
19
20  private static Map<String, Student> ucitajStudente()
21   throws IOException {
22
23   return new ObradaStudenta().ucitaj(
24    Paths.get("data/studenti.txt")
```

```
25    );
26   }
27
28   private static Map<String, Predmet> ucitajPredmete()
29    throws IOException {
30
31    return new ObradaPredmeta().ucitaj(
32     Paths.get("data/predmeti.txt")
33    );
34   }
35
36   private static List<Upis> ucitajUpise(
37    Map<String, Student> studenti, Map<String, Predmet> predmeti)
38    throws IOException {
39
40    return new ObradaUpisa(studenti, predmeti).ucitaj(
41     Paths.get("data/upisi.txt")
42    );
43   }
44
45 }
46
```

Usporedite razrede `ObradaStudenta` iz primjera 17.22 i iz primjera 17.23. Vidite li u čemu je razlika?

Usporedite sada metodu `Upisi.ucitajStudente()` u tim istim primjerima. Uočavate li razliku? Evo tih implementacija još jednom u nastavku. Klijentska metoda `Upisi.ucitajStudente()` kod rješenja sa strategijom izgleda ovako:

```
1 return Ucitavac.ucitaj(
2  Paths.get("data/studenti.txt"),
3  new ObradaStudenta());
```

Ista metoda kod rješenja s okvirnom metodom izgleda ovako:

```
1 return new ObradaStudenta().ucitaj(
2  Paths.get("data/studenti.txt")
3 );
```

U prvom slučaju (strategija) metoda `Ucitavac.ucitaj` je statička metoda koja sadrži implementaciju koda i treba referencu na drugi objekt koji nudi implementacije "primitiva"; u drugom slučaju stvaramo novi objekt (primjerak razreda koji nasljeđuje osnovni razred okvirne metode i tako donira implementacije apstraktnih primitiva iz osnovnog razreda) i nad njim pozivamo općenitu metodu iz osnovnog razreda.



Bilješka

Iz razloga što strategija odvaja u različite hijerarhije klijenta i implementacije potrebnih primitiva (pa je iste implementacije moguće koristiti od više nesrodnih klijenata), u praksi se strategija češće koristi no okvirna metoda.

Oblikovni obrazac Okvirna metoda

Kroz prethodni primjer upoznali smo se s oblikovnim obrasce Okvirna metoda. Ovdje ćemo dati formalan opis tog obrasca.

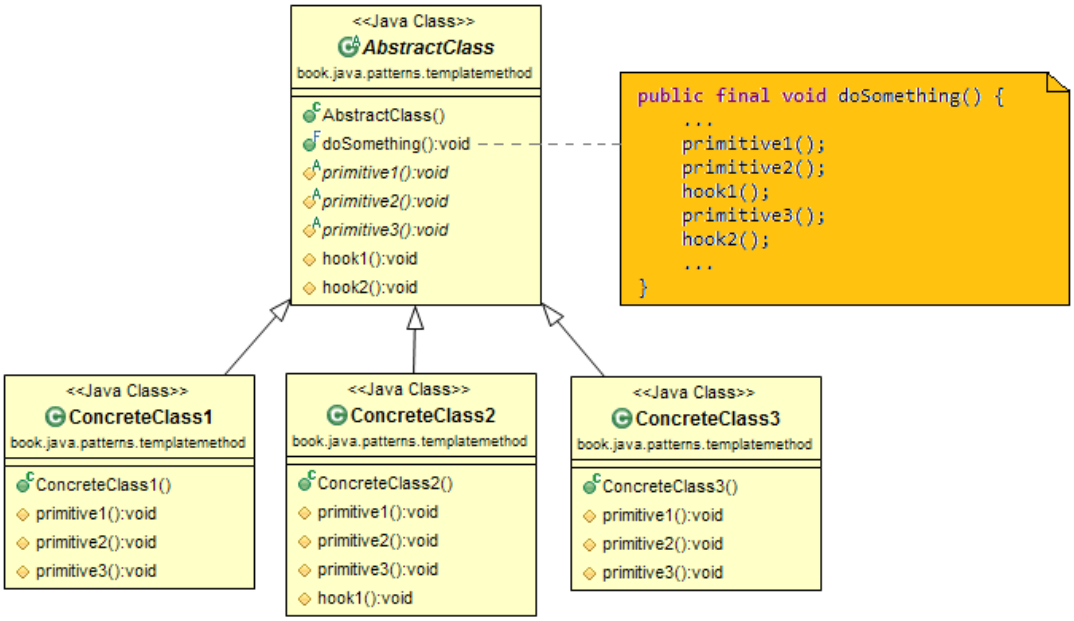
Oblikovni obrazac Okvirna metoda pripada u porodicu *ponašajnih oblikovnih obrazaca*. Ideja obrasca je ponuditi strukturu algoritma i dopustiti razredima koji ga nasljeđuju da definiraju ili izmijene pojedine korake algoritma ali bez mogućnosti da mijenjaju samu strukturu.

Struktura algoritma dana je u *okvirnoj metodi*: članskoj metodi apstraktnog razreda koja je javna i konačna (zaštićena od nadjačavanja). Metoda je javna kako bi je klijent mogao pozivati. Algoritam u toj metodi ostvaren je pozivanjem metoda-primitiva te nadomjesnih metoda koje ostvaruju pojedine korake algoritma.

Metode primitivi u apstraktnom su razredu deklarirane kao zaštićene apstraktne metode (nemaju implementaciju a vidljive su samo razredima koji nasljeđuju taj apstraktan razred). Nadomjesne metode su zaštićene metode koje imaju ponuđenu implementaciju koju razred koji nasljeđuje ovaj apstraktni razred može nadjačati ako je to potrebno, ali i ne mora ako mu je implementacija prihvatljiva.

Oblikovni obrazac razlikuje dva sudionika: apstraktni razred te konkretne razrede. Strukturni dijagram koji prikazuje sudionike obrasca dan je na slici 17.18.

Slika 17.18. Strukturni dijagram oblikovnog obrasca Okvirna metoda



Sudionik *Apstraktni razred* (na slici `AbstractClass`) nudi implementaciju okvirne metode (na slici `doSomething`) kojom definira općeniti algoritam obrade te deklarira primitive (na slici apstraktne metode `primitive1`, `primitive2` i `primitive3`) te nadomjesne metode (na slici metode `hook1` i `hook2`). Budući da je ovaj razred apstraktan, klijent ne može stvarati njegove primjerke.

Sudionik *Konkretni razred* (na slici razredi `ConcreteClass1`, `ConcreteClass2` i `ConcreteClass3`) nasljeđuje apstraktni razred, nudi implementacije apstraktnih primitiva te po potrebi nadjačava nadomjesne metode prikladnijim implementacijama. Time do kraja definira korake algoritma. Klijent može stvoriti primjerak konkretnog razreda i nad njim pozvati okvirnu metodu. U prikazanom dijagrami, samo je razred `ConcreteClass2` iskoristio mogućnost nadjačavanja nadomjesnih metoda (nadjačao je metodu `hook1`).

Datoteke sa slučajnim pristupom

Osim kroz apstrakciju rada s tokovima, datotekama je ponekad potrebno pristupati i izravno; primjerice, kada u datoteku želimo istovremeno pisati i čitati ili kada se po datoteci želimo izravno pozicionirati. U tu svrhu Java nudi razred `java.io.RandomAccessFile`.

Razred nudi konstruktore koji primaju dva argumenta. Prvi je objekt tipa `File` ili tipa `String` koji predstavlja putanju do datoteke koju treba otvoriti. Drugi argument određuje način na koji

datoteku treba otvoriti (zadaje se kao string). Najčešće se koristi `"r"` ako se datoteku želi otvoriti samo za čitanje ili pak `"rw"` ako se datoteku želi otvoriti i za čitanje i za pisanje.

Razred nudi metodu `seek(position)` kojom je moguće postaviti trenutnu poziciju unutar datoteke od koje će se obavljati sljedeće naredbe kao i metodu `getFilePointer()` koja vraća trenutnu poziciju u datoteci. Uz ove, razred nudi i uobičajene metode za čitanje i pisanje jednog okteta, za čitanje i pisanje polja okteta kao i niz pomoćnih metoda kojima je moguće zapisivati i čitati javine primitivne tipove (sjetite se razreda `DataInputStream` i `DataOutputStream`). Evo jednostavnog primjera.

```
 1 RandomAccessFile raf = new RandomAccessFile("datoteka.bin", "rw");
 2
 3 raf.seek(50);
 4 System.out.println("Prije pisanja: " + raf.getFilePointer());
 5 raf.writeInt(0x3E5B4211);
 6 System.out.println("Nakon pisanja: " + raf.getFilePointer());
 7
 8 raf.seek(50);
 9 System.out.format(
10  "%02X %02X %02X %02X%n",
11  raf.read(),
12  raf.read(),
13  raf.read(),
14  raf.read()
15 );
16
17 raf.close();
```

Program u tekućem direktoriju stvara datoteku `datoteka.bin` (ako ista ne postoji) odnosno ako postoji, samo je otvara (redak 1). Potom se pozicionira u datoteci na poziciju 50 (redak 3) i na zaslon ispisuje trenutnu poziciju (redak 4). Potom u datoteku zapisuje 32-bitnu vrijednost 0x3E5B4211 po formatu BigEndian (poziv će rezultirati zapisivanjem 4 okteta, redom: 0x3E, 0x5B, 0x42, 0x11; redak 5). U retku 6 na zaslon se ispisuje trenutna pozicija u datoteci (očekujemo 54). U retku 8 poziciju ponovno vraćamo na 50 nakon čega čitamo četiti okteta i ispisujemo ih. Kako specifikacija jezika Java garantira da je poredak evaluacije argumenata pri pozivu metode s lijeva na desno, možemo biti sigurni da će se najprije izvršiti prvi `raf.read()` koji će pročitati podatak s pozicije 50, potom drugi koji će pročitati podatak s pozicije 51, i tako redom. Program će, očekivano, ispisati:

```
Prije pisanja: 50
Nakon pisanja: 54
3E 5B 42 11
```

Za detaljnije upoznavanje s podržanim metodama čitatelj se upućuje na dokumentaciju ovog razreda.

Poglavlje 18. Studija slučaja: programski jezik *vlang*

U okviru ovog poglavlja napravit ćemo mali prekid u upoznavanju različitih tehnologija Jave i umjesto toga razraditi jedan složeniji programski zadatak. Prilikom izrade kompleksnijih programskih sustava nije rijetkost da se u svrhu ostvarivanja fleksibilnog rješenja korisniku nudi upravljanje implementiranom funkcionalnošću kroz vlastiti skriptni jezik koji je razvijen upravo za potrebe navedenog programskog sustava.

Razvoj i programsko ostvarenje prevoditelja za složenije programske jezike te omogućavanje izvođenja tako napisanih programa vrlo je kompleksan zadatak iza kojeg stoji puno teorije i različitih formalizama. U okviru ovog poglavlja stoga ćemo razmotriti jedan jednostavniji programski jezik koji ćemo nazvati *vlang*: jezik za računanje s vektorima. Naš će jezik podržavati definiranje varijabli, pridruživanje algebarskih izraza varijablama te ispisivanje algebarskih izraza. Primjer programa napisanog u ovom programskom jeziku prikazan je u nastavku.

Primjer 18.1. Primjer programa napisanog jezikom *vlang*

```
def a, b: vector;
def c: vector;
let a = [1.5, 2.8];
let b = [2, 5];
let c = a - (b + [1.2,1]);
print a, b, c;
```

Jezik *vlang* definira tri naredbe. Naredba **def** omogućava definiranje varijabli. Naredba **let** je nadredba pridruživanja kojom se vrijednost izraza specificiranog s desne strane operatora = pridružuje varijabli čije je ime navedeno s lijeve strane operatora =. Konačno, naredba **print** omogućava ispis niza algebarskih izraza. Elementi niza se međusobno odvajaju znakom zarez i svaki se ispisuje u zasebnom retku.

Obradu programa napisanog jezikom *vlang* rastavit ćemo u tri koraka (slika 18.1 daje grafički prikaz koraka):

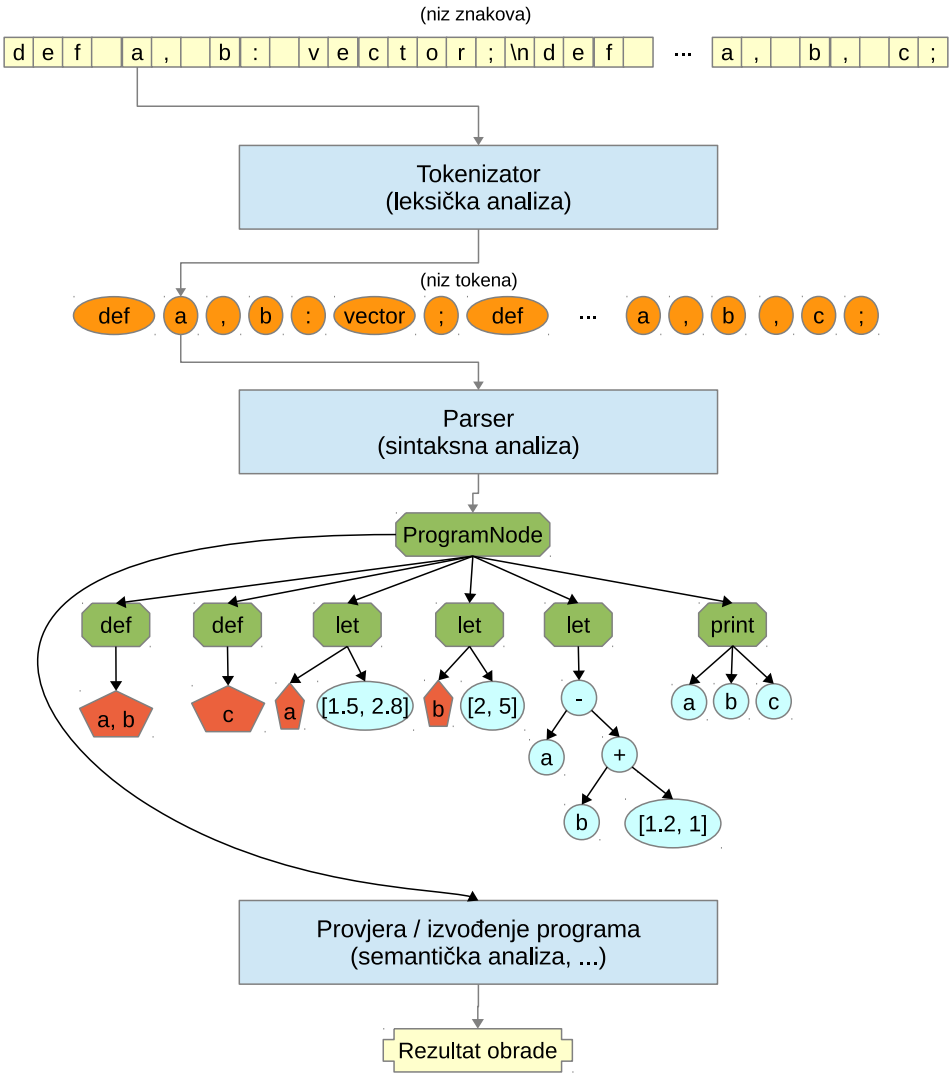
- *leksička analiza* ima kao zadatak iz toka znakova koje čini izvorni kod programa generirati tok *tokena* - temeljnih objekata od kojih je sastavljen izvorni kod (to su ključne riječi, identifikatori, brojevi, operatori, ...),

- *sintaksna analiza* ima kao zadatak iz toka tokena izgraditi sintaksno stablo - podatkovnu strukturu koja predstavlja izvorni kod programa uporabom jezičnih konstrukata (u našem slučaju to su pojedine naredbe)

- *semantička analiza / obrada programa* ima kao zadatak iz generiranog sintaksnog stabla ispitati je li program korektan te pokrenuti odgovarajuću obradu (primjerice, izvesti program kako bismo doznali rezultat izvođenja).

Treba primjetiti da navedeni koraci nisu posve identični koracima koje obavlja klasični jezični procesor. Klasičnom obradom izvornog koda programa obično se generira međukod ili strojni kod koji se pohranjuje u neki oblik izvršive datoteke, a samo izvođenje obavlja se u nekom kasnijem trenutku. Kako ovdje razmatramo uporabu jednostavnog jezika koji koristimo unutar nekog drugog programskog sustava, ono što će nam biti interesantno jest program odmah i pokrenuti pa su prethodni koraci prilagođeni tom cilju.

Slika 18.1. Dijagram toka obrade programa



Osnovni razredi

Da bismo započeli s izgradnjom programskog ostvarenja sustava opisanog slikom 18.1, najprije moramo definirati nekoliko pomoćnih razreda. Jezik koji smo zamislili namjenjen je pisanju programa čija je zadaća manipuliranje vektorima. Stoga prvo što moramo definirati je upravo model vektora realnih brojeva. Implementacija razreda `Vector` prikazana je u izvornom kodu 18.2 dok je sučelje tog razreda prikazano na slici 18.2.

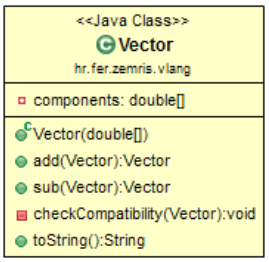
Primjer 18.2. Model nepromjenjivog vektora realnih brojeva.

```
 1 package hr.fer.zemris.vlang;
 2
 3 import java.util.Arrays;
 4
 5 /**
 6  * Razred predstavlja implementaciju vektora čije su komponente
 7  * decimalni brojevi.
 8  *
 9  * @author marcupic
10  */
11 public class Vector {
```

```
12
13  /**
14   * Komponente vektora.
15   */
16  private double[] components;
17
18  /**
19   * Konstruktor.
20   * @param components komponente vektora
21   */
22  public Vector(double...components) {
23   this.components = Arrays.copyOf(components, components.length);
24  }
25
26  /**
27   * Zbrajanje vektora. Metoda vraća novi vektor koji je
28   * jednak zbroju trenutnog vektora i predanog vektora.
29   * @param other vektor s kojim treba zbrojiti
30   * @return rezultat zbrajanja
31   */
32  public Vector add(Vector other) {
33   checkCompatibility(other);
34   double[] res = new double[components.length];
35   for(int i = 0; i < res.length; i++) {
36    res[i] = components[i] + other.components[i];
37   }
38   return new Vector(res);
39  }
40
41  /**
42   * Oduzimanje vektora. Metoda vraća novi vektor koji je
43   * jednak vektoru koji se dobije kada se od trenutnog vektora
44   * oduzme predani vektor.
45   * @param other vektor koji treba oduzeti
46   * @return rezultat oduzimanja
47   */
48  public Vector sub(Vector other) {
49   checkCompatibility(other);
50   double[] res = new double[components.length];
51   for(int i = 0; i < res.length; i++) {
52    res[i] = components[i] - other.components[i];
53   }
54   return new Vector(res);
55  }
56
57  /**
58   * Pomoćna metoda koja provjerava jesu li trenutni i predani
59   * vektor kompatibilni po dimenzijama.
60   * @param other vektor s kojim se treba usporediti
61   */
62  private void checkCompatibility(Vector other) {
63   if(components.length != other.components.length) {
64    throw new RuntimeException("Incompatible vectors.");
65   }
66  }
67
68   @Override
69  public String toString() {
```

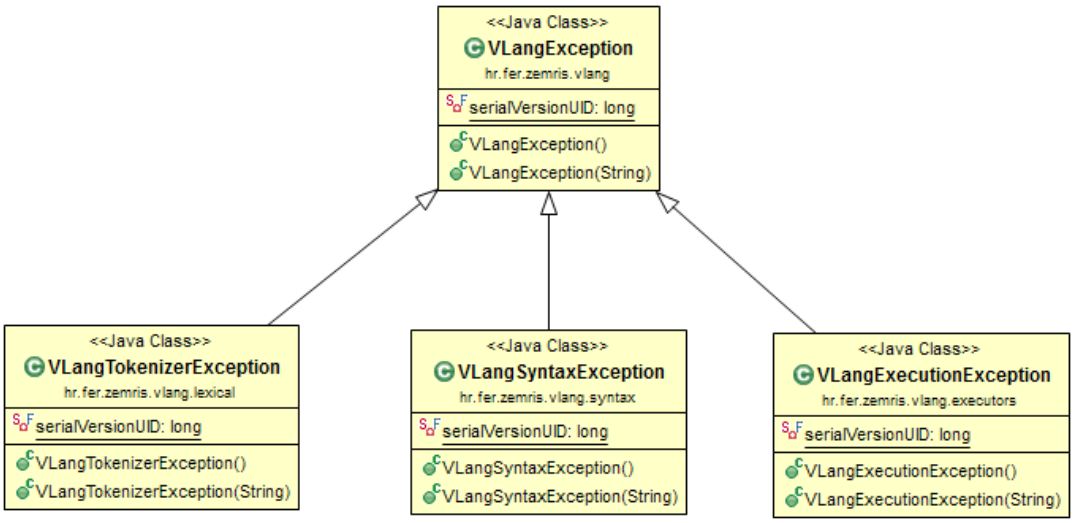
```
70     StringBuilder sb = new StringBuilder();
71     sb.append('[');
72     for(int i = 0; i < components.length; i++) {
73      if(i>0) sb.append(", ");
74      sb.append(components[i]);
75     }
76     sb.append(']');
77    return sb.toString();
78   }
79 }
80
```

Slika 18.2. Razred Vector



Prilikom različitih faza obrade i izvođenja programa napisanog jezikom *vlang* moguće su i pojave različitih pogrešaka. Stoga ćemo napraviti podstablo iznimaka koje ćemo izvesti iz `java.lang.RuntimeException`. Vršna iznimka koju ćemo izvesti bit će `hr.fer.zemris.vlang.VLangException` i ona će enkapsulirati bilo koju od iznimaka koje se događaju prilikom prevođenja i obrade programa. Iz nje ćemo izvesti tri nove iznimke, pri čemu će svaka odgovarati pogreškama koje se javljaju u različitim fazama. Slika 18.3 prikazuje ovo podstablo.

Slika 18.3. Podstablo iznimaka



Provedba leksičke analize

Prvi korak u obradi izvornog koda je njegova tokenizacija. Razmatramo li općenito kako se provodi proces tokenizacije, najopćenitije implementacije kreću od konstrukcije nedeterminističkog konačnog automata s epsilon-prijelazima koji se dobije iz regularnih izraza koji opisuju strukturu svakog od tokena. U okviru ove studije slučaja mi ćemo krenuti nešto jednostavnijim

pristupom: tokeni od kojih se sastoji izvorni kod programskog jezika *vlang* dovoljno su jednostavni da ih se može razlikovati već po prvom slovu. Zahvaljujući tome moći ćemo napraviti puno jednostavniju implementaciju.

U programskom jeziku *vlang* definirane su različite vrste tokena. Stoga ćemo definirati enumeraciju `VLangTokenType` koja definira sve moguće vrste tokena (18.3). Tako razlikujemo identifikatore (poput naziva varijabli), ključne riječi (naredbe **def**, **let** i **print** te tip **vector**), operatore `+` i `-`, znakove interpunkcije (`.`, `:`, `;`), operator pridruživanja `=`, oble zagrade i konačno vektorske konstante (započinju otvorenom uglatom zagradom, slijedi jedan ili više realnih brojeva razdvojenih zagradama i završavaju zatvorenom uglatom zagradom). Konačno, postojat će i posebna vrsta tokena (`EOF`) koja će označavati da su svi znakovi izvornog koda pročitani i da je tokenizacija izvornog koda gotova.

Primjer 18.3. Enumeracija različitih vrsta tokena.

```
 1 package hr.fer.zemris.vlang.lexical;
 2
 3 /**
 4  * Enumeracija vrsta tokena od kojih se sastoji <i>vlang</i> program.
 5  *
 6  * @author marcupic
 7  */
 8 public enum VLangTokenType {
 9  /** Označava da više nema tokena **/
10  EOF,
11  /** Identifikator **/
12  IDENT,
13  /** Ključna riječ **/
14  KEYWORD,
15  /** Dvotočka **/
16  COLON,
17  /** Točka-zarez **/
18  SEMICOLON,
19  /** Zarez **/
20  COMMA,
21  /** Vektorska konstanta **/
22  VECTOR_CONSTANT,
23  /** Operator plus **/
24  OP_PLUS,
25  /** Operator minus **/
26  OP_MINUS,
27  /** Znak pridruživanja **/
28  ASSIGN,
29  /** Otvorena obla zagrada **/
30  OPEN_PARENTHESES,
31  /** Zatvorena obla zagrada **/
32  CLOSED_PARENTHESES
33 }
34
```

Potom je potrebno definirati model jednog tokena. Tokeni će biti primjerci razreda `VLangToken`, i pri tome će pamtiti kojeg su tipa te koja im je vrijednost. Tokeni koji su identifikatori ili ključne riječi vrijednost će imati postavljenu upravo na odgovarajući string koji predstavlja sam identifikator odnosno ključnu riječ. Tokeni koji su vektorske konstante vrijednost će imati postavljenu na primjerak razreda `Vector` koji odgovara definiranoj konstanti (slijedi da će tokenizator iz izvornog koda morati pročitati o kojem se vektoru radi i stvoriti odgovarajući primjerak razreda `Vector`). Preostali tokeni vrijednost će imati postavljenu na `null` jer je već iz njihovog tipa jasna interpretacija. Razred koji modelira token prikazan je u izvornom kodu 18.4.

Primjer 18.4. Model jednog tokena.

```
 1 package hr.fer.zemris.vlang.lexical;
 2
 3 /**
 4  * Razred modelira jedan token ulaznog programa.
 5  *
 6  * @author marcupic
 7  */
 8 public class VLangToken {
 9
10  /**
11   * Vrsta tokena.
12   */
13  private VLangTokenType tokenType;
14  /**
15   * Vrijednost tokena.
16   */
17  private Object value;
18
19  /**
20   * Konstruktor.
21   * @param tokenType vrsta tokena
22   * @param value vrijednost tokena
23   */
24  public VLangToken(VLangTokenType tokenType, Object value) {
25   super();
26   if(tokenType==null) throw new IllegalArgumentException(
27    "Token type can not be null.");
28   this.tokenType = tokenType;
29   this.value = value;
30  }
31
32  /**
33   * Dohvat vrste tokena.
34   * @return vrsta tokena
35   */
36  public VLangTokenType getTokenType() {
37   return tokenType;
38  }
39
40  /**
41   * Dohvat vrijednosti tokena.
42   * @return vrijednost tokena ili <code>null</code> ako token
43   *         ove vrste nema pridruženu vrijednost
44   */
45  public Object getValue() {
46   return value;
47  }
48 }
49
```

Konačno, potrebno je napisati i implementaciju samog tokenizatora. Ideja je jednostavna. Tokenizator je zadužen za grupiranje jednog ili više znakova u tokene. U konstruktoru, tokenizator dobiva izvorni kod programa zapisan kao jedan string. Taj string pretvara u polje znakova koje potom čuva u privatnoj članskoj varijabli `data`. U svakom trenutku, tokenizator ima kazaljku na prvi neobrađeni znak (kazaljka `curPos`) koja se u konstruktoru postavlja na `0`. Implementacija tokenizatora prikazana je u izvornom kodu 18.5. Sam konstruktor poziva metodu koja izlučuje

prvi token i njega pohranjuje kao trenutni token u privatnu člansku varijablu `currentToken`. Razred nudi metodu `getCurrentToken` za dohvaćanje trenutnog tokena koju je moguće pozivati proizvoljan broj puta. Tokenizator nudi i metodu čije pozivanje izlučuje sljedeći token (ako takav postoji): `nextToken`.

Razred `VLangTokenizer` koristi i dvije statičke mape. S obzirom da postoji više tokena koji se stvaraju temeljem samo jednog znaka iz izvornog koda, mapa `mapper` čuva parove *znak-vrsta tokena* za sve definirane tokene za koje je to moguće. Skup `keywords` služi kako bi se nakon pronalaska identifikatora provjerilo je li identifikator ključna riječ te se vratio ispravan tip tokena (strukturno, sve ključne riječi jezika *vlang* jesu identifikatori). Implementacija tokenizatora dana je u izvornom kodu 18.5.

Primjer 18.5. Implementacija tokenizatora.

```
 1 package hr.fer.zemris.vlang.lexical;
 2
 3 import hr.fer.zemris.vlang.Vector;
 4
 5 import java.util.ArrayList;
 6 import java.util.HashMap;
 7 import java.util.HashSet;
 8 import java.util.List;
 9 import java.util.Map;
10 import java.util.Set;
11
12 /**
13  * Tokenizator izvornog koda programa napisanog jezikom
14  * <i>vlang</i>.
15  *
16  * @author marcupic
17  */
18 public class VLangTokenizer {
19
20  /**
21   * Polje znakova koji čine izvorni kod programa koji se
22   * obrađuje.
23   */
24  private char[] data;
25  /**
26   * Kazaljka na prvi neobrađeni znak u polju <code>data</code>.
27   */
28  private int curPos;
29  /**
30   * Posljednji token koji je stvoren analizom izvornog koda
31   * programa.
32   */
33  private VLangToken currentToken;
34
35  /**
36   * Statička mapa tipova tokena koje je moguće utvrditi prema
37   * samo jednom znaku (znak == token).
38   */
39  private static final Map<Character, VLangTokenType> mapper;
40
41  // Inicijalizacija mape mapper
42  static {
43   mapper = new HashMap<>();
44   mapper.put(Character.valueOf(':'), VLangTokenType.COLON);
```

```
45   mapper.put(Character.valueOf(';'), VLangTokenType.SEMICOLON);
46   mapper.put(Character.valueOf('+'), VLangTokenType.OP_PLUS);
47   mapper.put(Character.valueOf('-'), VLangTokenType.OP_MINUS);
48   mapper.put(Character.valueOf(','), VLangTokenType.COMMA);
49   mapper.put(Character.valueOf('='), VLangTokenType.ASSIGN);
50   mapper.put(Character.valueOf('('),
51    VLangTokenType.OPEN_PARENTHESES);
52   mapper.put(Character.valueOf(')'),
53    VLangTokenType.CLOSED_PARENTHESES);
54  }
55
56  /**
57   * Skup svih ključnih riječi programa.
58   */
59  private static final Set<String> keywords;
60
61  // Inicijalizacija skupa ključnih riječi
62  static {
63   keywords = new HashSet<>();
64   keywords.add("def");
65   keywords.add("vector");
66   keywords.add("let");
67   keywords.add("print");
68  }
69
70  /**
71   * Konstruktor. Prima izvorni kod programa kao
72   * <code>String</code>.
73   * @param program izvorni kod programa
74   * @throws VLangTokenizerException ako dođe do pogreške pri
75   *         tokenizaciji
76   */
77  public VLangTokenizer(String program) {
78   data = program.toCharArray();
79   curPos = 0;
80   extractNextToken();
81  }
82
83  /**
84   * Metoda dohvaća trenutni token. Metoda se može zvati više
85   * puta i uvijek će vratiti isti token, sve dok se ne
86   * zatraži izlučivanje sljedećeg tokena.
87   * @return trenutni token
88   */
89  public VLangToken getCurrentToken() {
90   return currentToken;
91  }
92
93  /**
94   * Metoda izlučuje sljedeći token, postavlja ga kao trenutnog
95   * i odmah ga i vraća.
96   * @throws VLangTokenizerException ako dođe do problema pri
97   *         tokenizaciji
98   */
99  public VLangToken nextToken() {
100  extractNextToken();
101  return getCurrentToken();
102 }
```

```
103
104  /**
105   * Metoda izlučuje sljedeći token iz izvornog koda.
106   * @throws VLangTokenizerException ako dođe do pogreške pri
107   *         tokenizaciji
108   */
109  private void extractNextToken() {
110   // Ako je već prije utvrđen kraj, ponovni poziv metode je greška:
111   if(currentToken!=null &&
112    currentToken.getTokenType()==VLangTokenType.EOF) {
113    throw new VLangTokenizerException("No tokens available.");
114   }
115
116   // Inače preskoči praznine:
117   skipBlanks();
118
119   // Ako više nema znakova, generiraj token za kraj izvornog
120   // koda programa
121   if(curPos>=data.length) {
122    currentToken = new VLangToken(VLangTokenType.EOF, null);
123    return;
124   }
125
126   // Vidi je li trenutni znak neki od onih koji direktno
127   // generiraju token:
128   VLangTokenType mappedType =
129    mapper.get(Character.valueOf(data[curPos]));
130   if(mappedType != null) {
131    // Stvori takav token:
132    currentToken = new VLangToken(mappedType, null);
133    // Postavi trenutnu poziciju na sljedeći znak:
134    curPos++;
135    return;
136   }
137
138   // Ako znak direktno ne generira token, provjeri što je.
139   if(Character.isLetter(data[curPos])) {
140    int startIndex = curPos;
141    curPos++;
142    while(curPos<data.length &&
143     Character.isLetter(data[curPos])) {
144     curPos++;
145    }
146    int endIndex = curPos;
147    String value = new String(data,
148     startIndex, endIndex-startIndex);
149    if(keywords.contains(value)) {
150     currentToken = new VLangToken(
151      VLangTokenType.KEYWORD, value);
152     return;
153    }
154    currentToken = new VLangToken(VLangTokenType.IDENT, value);
155    return;
156   }
157
158   // Ako pak imamo početak vektorske konstante:
159   if(data[curPos]=='[') {
160    curPos++;
```

```
161    skipBlanks();
162    List<Double> components = new ArrayList<>();
163    components.add(extractNumber());
164    while(true) {
165     skipBlanks();
166     if(curPos>=data.length)
167      throw new VLangTokenizerException("Invalid vector constant.");
168     if(data[curPos]==']') {
169      curPos++;
170      break;
171     }
172     if(data[curPos]!=',')
173      throw new VLangTokenizerException("Invalid vector constant.");
174     curPos++;
175     skipBlanks();
176     components.add(extractNumber());
177    }
178    double[] values = new double[components.size()];
179    for(int i = 0, n = components.size(); i < n; i++) {
180     values[i] = components.get(i).doubleValue();
181    }
182    currentToken = new VLangToken(
183     VLangTokenType.VECTOR_CONSTANT, new Vector(values));
184    return;
185   }
186
187   // Inače nije ništa što razumijemo:
188   throw new VLangTokenizerException(
189    "Invalid character found: '"+data[curPos]+"'.");
190  }
191
192  /**
193   * Metoda izlučuje decimalni broj u formatu predznak, cijeli
194   * dio, opcionalna točka i decimalni dio.
195   * @return decimalni broj zapisan u izvornom kodu programa
196   *         na trenutnoj poziciji
197   */
198  private Double extractNumber() {
199   if(curPos>=data.length)
200    throw new VLangTokenizerException("Invalid vector constant.");
201
202   // Je li broj negativan?
203   boolean negative = false;
204   if(data[curPos]=='+') {
205    curPos++;
206   } else if(data[curPos]=='-') {
207    negative = true;
208    curPos++;
209   }
210
211   // Zapamti početak:
212   int startIndex = curPos;
213
214   // Pređi preko cijelobrojnog dijela:
215   while(curPos<data.length &&
216    Character.isDigit(data[curPos])) {
217    curPos++;
218   }
```

```
219
220   // Ako smo došli do decimalne točke:
221   if(curPos<data.length && data[curPos]=='.') {
222    curPos++;
223    // Pređi preko decimalnog dijela:
224    while(curPos<data.length &&
225     Character.isDigit(data[curPos])) {
226     curPos++;
227    }
228   }
229
230   // Zapamti kraj i dohvati prihvaćeni dio kao string:
231   int endIndex = curPos;
232   String value = new String(data,
233    startIndex, endIndex - startIndex);
234
235   // Ako broj nema znamenaka ili ima samo točku, baci iznimku:
236   if(endIndex==startIndex ||
237    (endIndex==startIndex+1 && data[startIndex]=='.')) {
238    throw new VLangTokenizerException(
239     "Invalid decimal number: '"+value+"'.");
240   }
241
242   // Inače konvertiraj i vrati broj:
243   double d = Double.parseDouble(value);
244   if(negative) d = -d;
245   return Double.valueOf(d);
246  }
247
248  /**
249   * Metoda pomiče kazaljku trenutnog znaka tako da preskače
250   * sve prazne znakove (razmaci, prelasci u novi red,
251   * tabulatori).
252   */
253  private void skipBlanks() {
254   while(curPos<data.length) {
255    char c = data[curPos];
256    if(c==' ' || c=='\t' || c=='\r' || c=='\n') {
257     curPos++;
258     continue;
259    }
260    break;
261   }
262  }
263
264 }
265
```

Metoda koja čini glavni dio tokenizatora je`extractNextToken`. Metoda rad započinje provjerom je li već prethodno utvrđeno da su svi znakovi obrađeni; ako je, izaziva se iznimka. U suprotnom, kazaljka na trenutni znak se pomiče tako dugo dok je trenutni znak neki od praznih znakova (razmak, enter, tabulator). Jednom kada se utvrdi da kazaljka pokazuje na neprazni znak, analizira se što je trenutni znak i temeljem toga izlučuje odgovarajući token. Obratite pažnju kako je metoda napisana: po završetku rada, kazaljka na sljedeći znak mora pokazivati na prvi neobrađeni znak nakon izlučenog tokena.

Uz ovako napisan tokenizator spremni smo napisati i jednostavan demonstracijski program - pogledajte izvorni kod 18.6.

Primjer 18.6. Demonstracija rada tokenizatora.

```
 1 package test;
 2
 3 import hr.fer.zemris.vlang.lexical.VLangTokenType;
 4 import hr.fer.zemris.vlang.lexical.VLangTokenizer;
 5
 6 /**
 7  * Demonstracija rada tokenizatora.
 8  *
 9  * @author marcupic
10  */
11 public class TestTokenizer {
12
13  /**
14   * Metoda s kojom započinje izvođenje programa.
15   * Argumenti se ignoriraju.
16   * @param args argumenti naredbenog retka
17   */
18  public static void main(String[] args) {
19   String program = "def a, b,c: vector;\r\n" +
20     "let a = [1.5, 2.8];\r\n" +
21     "let b = [2, 5];\r\n" +
22     "let c = a - (b + [1.2,1]);\r\n" +
23     "print a, b, c;"
24     ;
25   VLangTokenizer tokenizer = new VLangTokenizer(program);
26
27   while(true) {
28    VLangToken token = tokenizer.getCurrentToken();
29    System.out.println(
30     "Trenutni token: " + token.getTokenType() +
31     ", vrijednost '" + token.getValue()+"'"
32    );
33    if(token.getTokenType() == VLangTokenType.EOF) break;
34    tokenizer.nextToken();
35   };
36  }
37
38 }
39
```

Pokretanjem programa dobit ćemo sljedeći izlaz na ekranu (središnji dio ispisa je uklonjen zbog uštede na zauzeću).

```
Trenutni token: KEYWORD, vrijednost 'def'
Trenutni token: IDENT, vrijednost 'a'
Trenutni token: COMMA, vrijednost 'null'
Trenutni token: IDENT, vrijednost 'b'
Trenutni token: COMMA, vrijednost 'null'
Trenutni token: IDENT, vrijednost 'c'
Trenutni token: COLON, vrijednost 'null'
Trenutni token: KEYWORD, vrijednost 'vector'
Trenutni token: SEMICOLON, vrijednost 'null'
Trenutni token: KEYWORD, vrijednost 'let'
Trenutni token: IDENT, vrijednost 'a'
Trenutni token: ASSIGN, vrijednost 'null'
Trenutni token: VECTOR_CONSTANT, vrijednost '[1.5, 2.8]'
```

```
Trenutni token: SEMICOLON, vrijednost 'null'
Trenutni token: KEYWORD, vrijednost 'let'
...
Trenutni token: KEYWORD, vrijednost 'print'
Trenutni token: IDENT, vrijednost 'a'
Trenutni token: COMMA, vrijednost 'null'
Trenutni token: IDENT, vrijednost 'b'
Trenutni token: COMMA, vrijednost 'null'
Trenutni token: IDENT, vrijednost 'c'
Trenutni token: SEMICOLON, vrijednost 'null'
Trenutni token: EOF, vrijednost 'null'
```

Izrada parsera i izgradnja sintaksnog stabla

Jednom kada smo napravili tokenizator, možemo krenuti u izgradnju parsera čija je zadaća provjeriti jesu li tokeni u izvornom kodu dani dopuštenim redosljedom, te ako jesu, izgraditi sintaksno stablo koje predstavlja napisani program. Prvi korak u tom smjeru je definiranje gramatike. Gramatika se sastoji od skupa pravila te skupa završnih i nezavršnih simbola. Završni simboli predstavljaju tokene jezika; nezavršni simboli predstavljaju predloške koje dalje treba proširiti u skladu s pravilima gramatike.

Prije no što krenemo s primjerom, važno je napomenuti da postoji više vrsta gramatika (različitih ekspresivnosti) i više načina kako se temeljem pojedine vrste gramatike može konstruirati parser. Ovo područje izuzetno je kompleksno i mi ćemo se ovdje ograničiti na vrlo jednostavan slučaj.

U pravilima gramatike koju ćemo definirati velikim početnim slovom te pisanjem u zagradama <> označit ćemo nezavršne znakove. Konkretno, imat ćemo sljedeće nezavršne znakove: `<Start>`, `<DefStmt>`, `<LetStmt>`, `<PrintStmt>`, `<Expr>`, `<Atom>`. Izgradnja sintaksnog stabla započet će od proširivanja nezavršnog znaka `<Start>`.

Završne znakove pisat ćemo ili pod jednostrukim navodnicima (ključne riječi te pojedine simbole), ili malim slovima (*id* će označavati identifikator, *vect* će označavati vektorsku konstantu).

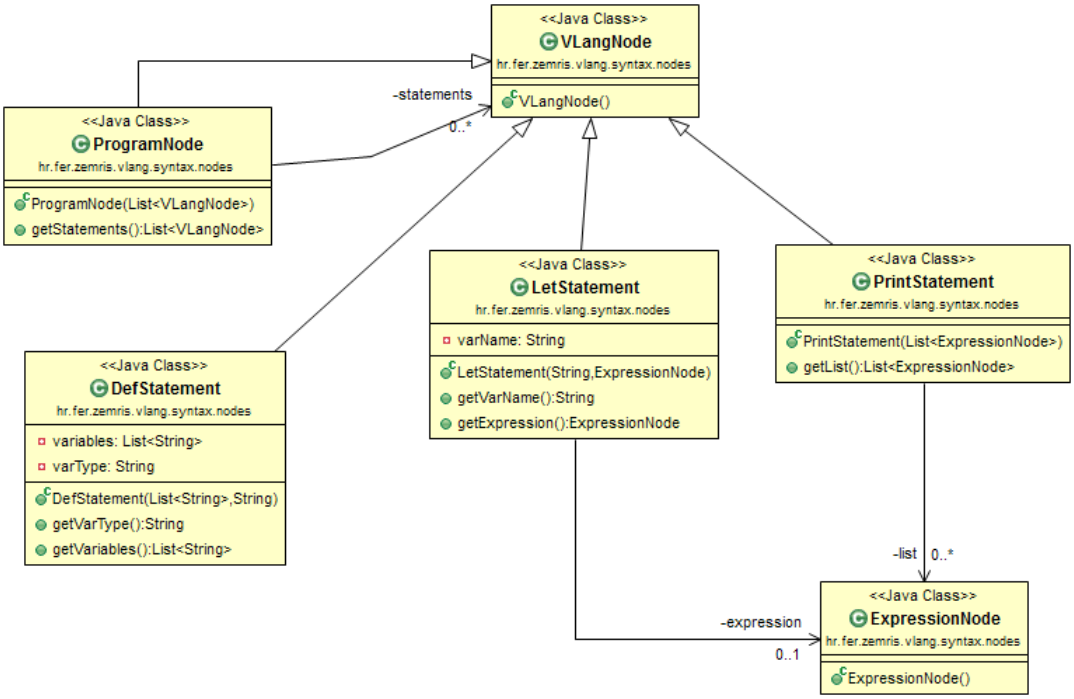
Želimo li grupirati neki dio podizraza, okružit ćemo ga vitičastim zagradama. Ako se neki dio pravila može uzastopno ponoviti nula, jednom ili više puta, taj će dio biti napisan unutar vitičastih zagrada nakon kojih ćemo pisati zvjezdicu, što znači da se ono što je definirano unutar vitičastih zagrada može ponavljati. Ako u okviru pravila nudimo alternative (ili-ili), međusobno ćemo ih odvojiti znakom |. Koristeći ova pravila, u nastavku je pikazana gramatika koju ćemo koristiti.

Primjer 18.7. Gramatika jezika *vlang*

```
[1]        <Start> : := { <DefStmt> | <LetStmt> | <PrintStmt> }*
[2]    <DefStmt> : := 'def' id { ',' id }* ':' 'vector' ';'
[3]    <LetStmt> : := 'let' id '=' <Expr> ';'
[4]  <PrintStmt> : := 'print' <Expr> { ',' <Expr> }* ';'
[5]  <ExprStmt> : := <Atom> { {'+' <Atom>} | {'-' <Atom>} }*
[6]       <Atom> : := id | vect | {'(' <Expr> ')'}
```

Što smo rekli ovim pravilima? Pravilo za nezavršni znak <Start> koji je ujedno i početni kaže da se svaki program sastoji od slijeda duljine nula ili više čiji su elementi naredbe **def**, **let** i **print**. Za izgradnju stabla definirat ćemo apstraktni razred `VLangNode` koji predstavlja bilo slijed naredbi, bilo neku konkretnu naredbu. Iz njega ćemo izvesti četiri konkretna razreda: razred `ProgramNode` koji predstavlja slijed naredbi (sadrži privatnu listu u koju pohranjuje reference na primjerke razreda `VLangNode`) te razrede `DefStatement`, `LetStatement` i `PrintStatement` koji predstavljaju redom naredbe **def**, **let** i **print** te imaju odgovarajuće podatkovne članove za pamćenje svih potrebnih dijelova naredbe. Odnosi između ovih razreda vidljivi su na slici 18.4.

Slika 18.4. Podstablo za pamćenje naredbi



Pravilo za <DefStmt> opisuje strukturu naredbe **def**: naredba započinje ključnom riječi **def** nakon čega slijedi jedan identifikator te još nula, jednom ili više puta slijed *zarez* pa identifikator; potom dolazi dvotočka, ključna riječ *vector* pa točka-zarez.

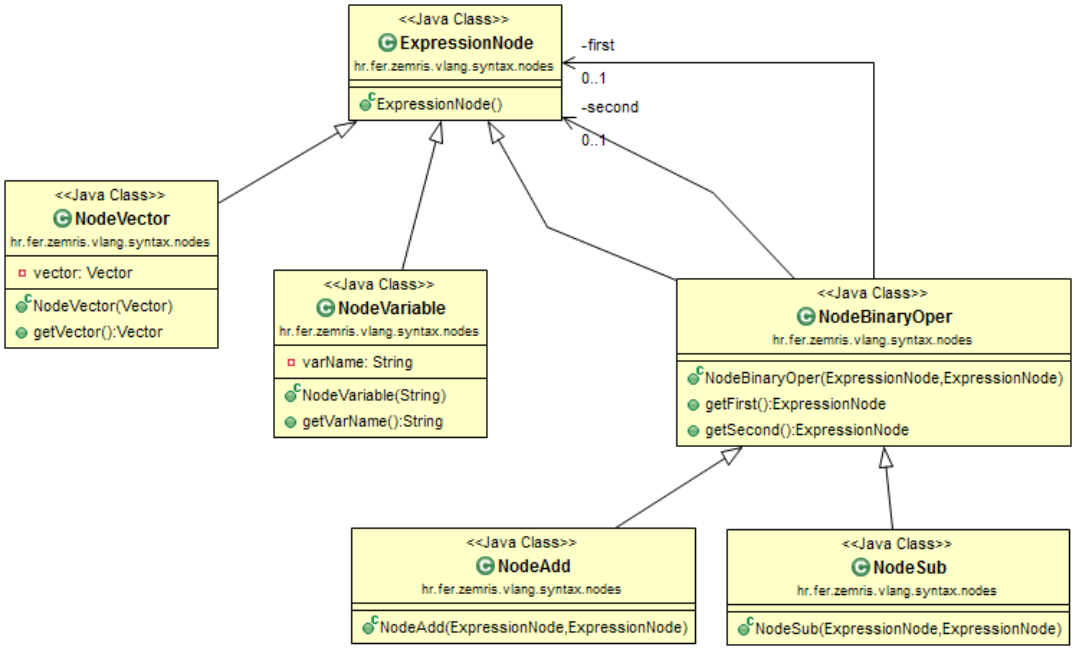
Pravilo za <LetStmt> opisuje strukturu naredbe **let**: naredba započinje ključnom riječi **let** nakon čega slijedi jedan identifikator, znak pridruživanja, algebarski izraz i konačno znak točka-zarez.

Pravilo za <PrintStmt> opisuje strukturu naredbe **print**: naredba započinje ključnom riječi **print** nakon čega slijedi jedan identifikator, potom nula ili više puta još slijed *zarez* i algebarski izraz, te na kraju znak točka-zarez.

Apstraktni algebarski izraz modelirat ćemo apstraktnim razredom `ExpressionNode`, iz kojeg ćemo izvesti još `NodeVector` čiji će primjerci predstavljati konkretne vektorske konstante, `NodeVariable` čiji će primjerci predstavljati pojave varijabli u izrazima te apstraktni razred `NodeBinaryOper` čiji će primjerci predstavljati binarne operatore. Razred `NodeBinaryOper` ima konstruktor koji prima reference na neka druga dva apstraktna izraza te sadrži privatne podatkovne članove preko kojih pamti dobivene reference te javne metode za njihov dohvat. Iz razreda `NodeBinaryOper` izvest ćemo dva konkretna razreda: `NodeAdd` te `NodeSub` čiji će primjerci predstavljati pojavu operatora zbrajanja odnosno oduzimanja. Međusobni odnos ovih razreda prikazan je na slici 18.5.

Umjesto izvođenja zasebnih razreda za svaki od konkretnih operatora mogli smo u konstruktoru razreda `NodeBinaryOper` predati i nekakav identifikator operatora koji bismo mogli zapamtiti u tom razredu. Međutim, jednom kada krenemo razmatrati kako izvoditi program napisan jezikom *vlang*, došli bismo u situaciju da bismo dodavanjem novih operatora morali dodatno intervenirati i u već prethodno napisani stari kod - a to ne želimo. Bolje rješenje je omogućiti da se dodavanje nove funkcionalnosti može obaviti naprosto dodavanjem novih razreda i bez potrebe za modificiranjem prethodno napisanog postojećeg koda.

Slika 18.5. Podstablo za pamćenje izraza



Razred `VLangParser` koji predstavlja parser za izgradnju sintaksnog stabla tehnikom rekurzivnim spustom prikazan je izvornim kodom 18.8.

Primjer 18.8. Implementacija parsera.

```
 1 package hr.fer.zemris.vlang.syntax;
 2
 3 import java.util.ArrayList;
 4 import java.util.List;
 5
 6 import hr.fer.zemris.vlang.Vector;
 7 import hr.fer.zemris.vlang.lexical.VLangTokenType;
 8 import hr.fer.zemris.vlang.lexical.VLangTokenizer;
 9 import hr.fer.zemris.vlang.lexical.VLangTokenizerException;
10 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
11 import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
12 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
13 import hr.fer.zemris.vlang.syntax.nodes.NodeAdd;
14 import hr.fer.zemris.vlang.syntax.nodes.NodeSub;
15 import hr.fer.zemris.vlang.syntax.nodes.NodeVariable;
16 import hr.fer.zemris.vlang.syntax.nodes.NodeVector;
17 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
18 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
19 import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
20
21 /**
22  * Implementacija parsera za jezik <i>vlang</i> rekurzivnim
23  * spustom.
24  *
25  * @author marcupic
26  */
27 public class VLangParser {
28
29  /**
30   * Tokenizator izvornog koda.
```

```
31   */
32  private VLangTokenizer tokenizer;
33  /**
34   * Stablo koje predstavlja parsirani program.
35   */
36  private ProgramNode programNode;
37
38  /**
39   * Konstruktor.
40   * @param tokenizer tokenizator izvornog koda
41   * @throws VLangSyntaxException u slučaju pogreške pri
42   *         parsiranju
43   * @throws VLangTokenizerException u slučaju pogreške
44   *         pri tokenizaciji
45   */
46  public VLangParser(VLangTokenizer tokenizer) {
47   this.tokenizer = tokenizer;
48   programNode = parse();
49  }
50
51  /**
52   * Dohvat stabla nastalog parsiranjem izvornog koda.
53   * @return stablo programa
54   */
55  public ProgramNode getProgramNode() {
56   return programNode;
57  }
58
59  /**
60   * Pomoćna metoda koja provjerava je li trenutni
61   * token danog tipa. Vraća <code>true</code>
62   * ako je, <code>false</code> inače.
63   * @param type tip s kojim se uspoređuje
64   * @return <code>true</code> ako je
65   */
66  private boolean isTokenOfType(VLangTokenType type) {
67   return tokenizer.getCurrentToken().getTokenType()==type;
68  }
69
70  /**
71   * Pomoćna metoda koja predstavlja implementaciju parsera
72   * rekurzivnim spustom. Ova metoda predstavlja vršnu metodu
73   * tako implementiranog parsera.
74   *
75   * @return stablo programa
76   */
77  private ProgramNode parse() {
78   List<VLangNode> statements = new ArrayList<>();
79   while(true) {
80    // Ako je kraj programa, gotovi smo:
81    if(isTokenOfType(VLangTokenType.EOF)) {
82     break;
83    }
84    // Inače prema sintaksi mora doći ključna riječ:
85    if(!isTokenOfType(VLangTokenType.KEYWORD)) {
86     throw new VLangSyntaxException("Keyword expected.");
87    }
88    // Ako gledam naredbu "def":
```

```
 89     if("def".equals(tokenizer.getCurrentToken().getValue())) {
 90      tokenizer.nextToken();
 91      statements.add(parseDef());
 92      continue;
 93     }
 94     // Ako gledam naredbu "let":
 95     if("let".equals(tokenizer.getCurrentToken().getValue())) {
 96      tokenizer.nextToken();
 97      statements.add(parseLet());
 98      continue;
 99     }
100     // Ako gledam naredbu "print":
101     if("print".equals(tokenizer.getCurrentToken().getValue())) {
102      tokenizer.nextToken();
103      statements.add(parsePrint());
104      continue;
105     }
106     // Inače imam nepoznatu naredbu:
107     throw new VLangSyntaxException("Unexpected keyword found.");
108    }
109    // Obradili smo čitav program:
110    return new ProgramNode(statements);
111   }
112
113   /**
114    * Pomoćna metoda koja predstavlja implementaciju parsera
115    * naredbe "def".
116    *
117    * @return čvor koji predstavlja ovu naredbu
118    */
119   private VLangNode parseDef() {
120    List<String> variables = new ArrayList<>();
121    while(true) {
122     if(!isTokenOfType(VLangTokenType.IDENT)) {
123      throw new VLangSyntaxException("Identifier was expected.");
124     }
125     variables.add(
126      (String)tokenizer.getCurrentToken().getValue()
127     );
128     tokenizer.nextToken();
129     if(isTokenOfType(VLangTokenType.COMMA)) {
130      tokenizer.nextToken();
131      continue;
132     }
133     break;
134    }
135    if(!isTokenOfType(VLangTokenType.COLON)) {
136     throw new VLangSyntaxException(
137      "Colon was expected after variable(s).");
138    }
139    tokenizer.nextToken();
140    if(!isTokenOfType(VLangTokenType.KEYWORD)) {
141     throw new VLangSyntaxException("A keyword was expected.");
142    }
143    String varType =
144     (String)tokenizer.getCurrentToken().getValue();
145    if(!"vector".equals(varType)) {
146     throw new VLangSyntaxException(
```

```
147     "Keyword 'vector' was expected.");
148    }
149    tokenizer.nextToken();
150    if(!isTokenOfType(VLangTokenType.SEMICOLON)) {
151     throw new VLangSyntaxException(
152      "A semicolon was expected.");
153    }
154    tokenizer.nextToken();
155    return new DefStatement(variables, varType);
156   }
157
158   /**
159    * Pomoćna metoda koja predstavlja implementaciju parsera
160    * naredbe "let".
161    *
162    * @return čvor koji predstavlja ovu naredbu
163    */
164   private VLangNode parseLet() {
165    if(!isTokenOfType(VLangTokenType.IDENT)) {
166     throw new VLangSyntaxException("Identifier was expected.");
167    }
168    String varName =
169     (String)tokenizer.getCurrentToken().getValue();
170    tokenizer.nextToken();
171    if(!isTokenOfType(VLangTokenType.ASSIGN)) {
172     throw new VLangSyntaxException("Assignment was expected.");
173    }
174    tokenizer.nextToken();
175    ExpressionNode expr = parseExpression();
176    if(!isTokenOfType(VLangTokenType.SEMICOLON)) {
177     throw new VLangSyntaxException("Semicolon was expected.");
178    }
179    tokenizer.nextToken();
180    return new LetStatement(varName, expr);
181   }
182
183   /**
184    * Pomoćna metoda koja predstavlja implementaciju parsera
185    * izraza (ono što se nalazi s desne strane u naredbi
186    * pridruživanja ili pak u naredbi print).
187    *
188    * @return čvor koji predstavlja čitav izraz
189    */
190   private ExpressionNode parseExpression() {
191    ExpressionNode first = parseAtomicValue();
192    while(true) {
193     if(isTokenOfType(VLangTokenType.OP_PLUS)) {
194      tokenizer.nextToken();
195      ExpressionNode second = parseAtomicValue();
196      first = new NodeAdd(first, second);
197      continue;
198     }
199     if(isTokenOfType(VLangTokenType.OP_MINUS)) {
200      tokenizer.nextToken();
201      ExpressionNode second = parseAtomicValue();
202      first = new NodeSub(first, second);
203      continue;
204     }
```

```
205     break;
206    }
207    return first;
208   }
209
210   /**
211    * Metoda koja parsira atomički izraz: to je vektorska
212    * konstanta, varijabla ili pak podizraz u oblim
213    * zagradama.
214    * @return čvor koji predstavlja ovaj izraz
215    */
216   private ExpressionNode parseAtomicValue() {
217    if(isTokenOfType(VLangTokenType.IDENT)) {
218     String varName =
219      (String)tokenizer.getCurrentToken().getValue();
220     tokenizer.nextToken();
221     return new NodeVariable(varName);
222    }
223    if(isTokenOfType(VLangTokenType.VECTOR_CONSTANT)) {
224     Vector vector =
225      (Vector)tokenizer.getCurrentToken().getValue();
226     tokenizer.nextToken();
227     return new NodeVector(vector);
228    }
229    if(isTokenOfType(VLangTokenType.OPEN_PARENTHESES)) {
230     tokenizer.nextToken();
231     ExpressionNode expression = parseExpression();
232     if(!isTokenOfType(VLangTokenType.CLOSED_PARENTHESES)) {
233      throw new VLangSyntaxException(
234       "Closed parentheses was expected.");
235     }
236     tokenizer.nextToken();
237     return expression;
238    }
239    throw new VLangSyntaxException("Unexpeced token type.");
240   }
241
242   /**
243    * Implementacija parsera naredbe "print".
244    * @return čvor koji predstavlja ovu naredbu.
245    */
246   private VLangNode parsePrint() {
247    List<ExpressionNode> list = new ArrayList<>();
248    list.add(parseExpression());
249    while(true) {
250     if(!isTokenOfType(VLangTokenType.COMMA)) {
251      break;
252     }
253     tokenizer.nextToken();
254     list.add(parseExpression());
255    }
256    if(!isTokenOfType(VLangTokenType.SEMICOLON)) {
257     throw new VLangSyntaxException("Semicolon was expected.");
258    }
259    tokenizer.nextToken();
260    return new PrintStatement(list);
261   }
262  }
```

```
263
```

Jednostavan demonstracijski program koji konstruira tokenizator i potom gradi sintaksno stablo uporabom razvijenog parsera prikazan je izvornim kodom 18.9.

Primjer 18.9. Demonstracija uporabe parsera.

```
 1 package test;
 2
 3 import hr.fer.zemris.vlang.lexical.VLangTokenizer;
 4 import hr.fer.zemris.vlang.syntax.VLangParser;
 5
 6 /**
 7  * Demonstracija rada parsera.
 8  *
 9  * @author marcupic
10  */
11 public class TestParser {
12
13  /**
14   * Metoda s kojom započinje izvođenje programa. Argumenti
15   * se ignoriraju.
16   * @param args argumenti naredbenog retka
17   */
18  public static void main(String[] args) {
19   String program = "def a, b: vector;\r\n" +
20     "def c: vector;\r\n" +
21     "let a = [1.5, 2.8];\r\n" +
22     "let b = [2, 5];\r\n" +
23     "let c = a - (b + [1.2,1]);\r\n" +
24     "print a, b, c;"
25     ;
26   VLangTokenizer tokenizer = new VLangTokenizer(program);
27   new VLangParser(tokenizer);
28  }
29
30 }
31
```

Sada kada znamo na koji način možemo generirati sintaksno stablo, pogledat ćemo kako napisati kod koji će napisani program i izvesti. Krenut ćemo s najjednostavnijim mogućim rješenjem, pokazat ćemo da ono radi ali i da ima svojih mana. Nakon toga pokušat ćemo popraviti svojstva prikazanog rješenja.

Izvođenje programa: pokušaj prvi

Naš prvi pokušaj pisanja koda koji zna izvesti program čije smo generirali stablo najjednostavniji je mogući: imamo jedan razred i u njemu niz pitalica: gledam li naredbu ovog tipa ili gledam naredbu onog tipa; gledam li izraz ovog tipa ili gledam izraz onog tipa. Za svaku moguću naredbu i svaki mogući tip imat ćemo dio koda koji određuje što treba napraviti. Rješenje je prikazano izvornim kodom 18.10.

Primjer 18.10. Izvođenje programa (1)

```
1 package hr.fer.zemris.vlang.executors.simple;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import hr.fer.zemris.vlang.Vector;
```

```
 6 import hr.fer.zemris.vlang.executors.VLangExecutionException;
 7 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
 8 import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
 9 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
10 import hr.fer.zemris.vlang.syntax.nodes.NodeAdd;
11 import hr.fer.zemris.vlang.syntax.nodes.NodeSub;
12 import hr.fer.zemris.vlang.syntax.nodes.NodeVariable;
13 import hr.fer.zemris.vlang.syntax.nodes.NodeVector;
14 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
15 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
16 import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
17
18 /**
19  * Jednostavna inačica stroja za izvođenje programa napisanog
20  * jezikom <i>vlang</i>.
21  *
22  * @author marcupic
23  */
24 public class ExecutorSimple {
25
26  /**
27   * Vršni čvor programa.
28   */
29  private ProgramNode programNode;
30  /**
31   * Mapa definiranih varijabli.
32   */
33  private Map<String, Vector> variables;
34
35  /**
36   * Konstruktor.
37   * @param programNode vršni čvor programa koji treba izvesti
38   */
39  public ExecutorSimple(ProgramNode programNode) {
40   this.programNode = programNode;
41  }
42
43  /**
44   * Metoda izvodi program predan u konstruktoru.
45   *
46   * @throws VLangExecutionException ako dođe do pogreške
47   *         pri izvođenju programa
48   */
49  public void execute() {
50   // Obriši mapu definiranih varijabli
51   variables = new HashMap<>();
52   // Izvedi svaku naredbu:
53   for(VLangNode node : programNode.getStatements()) {
54    // Ako je trenutna naredba naredba "def":
55    if(node instanceof DefStatement) {
56     DefStatement def = (DefStatement)node;
57     for(String varName : def.getVariables()) {
58      if(variables.containsKey(varName)) {
59       throw new VLangExecutionException(
60        "Variable "+varName+" already defined.");
61      } else {
62       variables.put(varName, null);
63      }
```

```
64      }
65      continue;
66     }
67     // Ako je trenutna naredba naredba "let":
68     if(node instanceof LetStatement) {
69      LetStatement let = (LetStatement)node;
70      if(!variables.containsKey(let.getVarName())) {
71       throw new VLangExecutionException(
72        "Undeclared variable " + let.getVarName() +
73        " in left side of let statement.");
74      }
75      Vector v = calculateExpression(let.getExpression());
76      variables.put(let.getVarName(), v);
77      continue;
78     }
79     // Ako je trenutna naredba naredba "print":
80     if(node instanceof PrintStatement) {
81      PrintStatement print = (PrintStatement)node;
82      for(ExpressionNode exp : print.getList()) {
83       System.out.println(calculateExpression(exp));
84      }
85     }
86    }
87   }
88
89   /**
90    * Pomoćna rekurzivna metoda za izračun vrijednosti izraza.
91    * @param node čvor koji predstavlja trenutni izraz
92    * @return izračunata vrijednost izraza
93    */
94   private Vector calculateExpression(ExpressionNode node) {
95    // Ako je trenutna čvor vektorska konstanta:
96    if(node instanceof NodeVector) {
97     return ((NodeVector)node).getVector();
98    }
99    // Ako je trenutna čvor varijabla:
100   if(node instanceof NodeVariable) {
101    NodeVariable varNode = (NodeVariable)node;
102    return getVariable(varNode.getVarName());
103   }
104   // Ako je trenutna čvor operator zbrajanja:
105   if(node instanceof NodeAdd) {
106    NodeAdd addOper = (NodeAdd)node;
107    Vector left = calculateExpression(addOper.getFirst());
108    Vector right = calculateExpression(addOper.getSecond());
109    return left.add(right);
110   }
111   // Ako je trenutna čvor operator oduzimanja:
112   if(node instanceof NodeSub) {
113    NodeSub subOper = (NodeSub)node;
114    Vector left = calculateExpression(subOper.getFirst());
115    Vector right = calculateExpression(subOper.getSecond());
116    return left.sub(right);
117   }
118   // Inače imamo pogrešku -- ovdje nismo smjeli doći:
119   throw new VLangExecutionException(
120    "Unknown expression: "+node.getClass().getName()+".");
121  }
```

```
122
123  /**
124   * Pomoćna metoda za dohvat vrijednosti varijable.
125   * @param varName ime varijable
126   * @return vrijednost varijable
127   * @throws VLangExecutionException ako varijabla
128   *         nije definirana ili nije inicijalizirana
129   */
130  private Vector getVariable(String varName) {
131   Vector v = variables.get(varName);
132   if(v==null) {
133    if(!variables.containsKey(varName)) {
134     throw new VLangExecutionException(
135      "Undeclared variable "+varName+" in expression.");
136    } else {
137     throw new VLangExecutionException(
138      "Uninitialized variable "+varName+" in expression.");
139    }
140   }
141   return v;
142  }
143 }
144
```

Primjer uporabe ovog razreda za izvođenje programa dan je izvornim kodom 18.11.

Primjer 18.11. Izvođenje programa (1)

```
 1 package test;
 2
 3 import hr.fer.zemris.vlang.executors.simple.ExecutorSimple;
 4 import hr.fer.zemris.vlang.lexical.VLangTokenizer;
 5 import hr.fer.zemris.vlang.syntax.VLangParser;
 6 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
 7
 8 /**
 9  * Program koji predstavlja demonstraciju izvođenja programa
10  * napisanog jezikom <i>vlang</i>. Izvođenje je ostvareno
11  * implementacijom {@link ExecutorSimple}.
12  *
13  * @author marcupic
14  */
15 public class TestExecutorSimple {
16
17  /**
18   * Metoda s kojom započinje izvođenje programa. Argumenti
19   * se ignoriraju.
20   * @param args argumenti naredbenog retka
21   */
22  public static void main(String[] args) {
23   String program = "def a, b: vector;\r\n" +
24     "def c: vector;\r\n" +
25     "let a = [1.5, 2.8];\r\n" +
26     "let b = [2, 5];\r\n" +
27     "let c = a - (b + [1.2,1]);\r\n" +
28     "print a, b, c;"
29     ;
30
```

```
31    VLangTokenizer tokenizer = new VLangTokenizer(program);
32    VLangParser parser = new VLangParser(tokenizer);
33
34    ProgramNode programNode = parser.getProgramNode();
35
36    new ExecutorSimple(programNode).execute();
37   }
38
39 }
40
```

Pokretanje ovog programa rezultirat će sljedećim ispisom na zaslonu.

```
[1.5, 2.8]
[2.0, 5.0]
[-1.7000000000000002, -3.2]
```

Sada kada smo se uvjerili da kod radi, pogledajmo malo detaljniju implementaciju prikazanu izvornim kodom 18.10. Razred `ExecutorSimple` kroz konstruktor dobiva referencu na sintaksno stablo programa, i tu referencu pamti u privatnoj članskoj varijabli `programNode`. Razred definira i člansku varijablu `variables` koja je po tipu mapa: ključevi te mape su imena definiranih varijabli, vrijednosti su trenutne vrijednosti svake od varijabli. Početno, ova mapa je prazna.

Izvođenje programa pokreće se pozivom javne metode `execute`. U metodi se najprije resetira mapa definiranih varijabli (za slučaj da korisnik izvana opetovano poziva ovu metodu). Nakon toga se u petlji `for` prolazi kroz sve naredbe primljenog slijeda naredbi. Za svaku se naredbu provjerava uporabom operatora `instanceof` o kojoj se točno naredbi radi. Tako primjerice, ako je trenutna naredba primjerak razreda `DefStatement`, stvaramo pomoćnu referencu upravo tog tipa i naredbu ukalupljujemo u `DefStatement`. Potom dohvaćamo listu imena varijabli koje se definiraju, i redom provjeravamo postoje li već takvi ključevi u mapi varijabli; ako postoje, imamo semantičku pogrešku u programu: korisnik je više puta pokušao definirati istu varijablu pa bacamo iznimku. U suprotnom, u mapu dodajemo prazno povezivanje s imenom varijable. Na sličan način rješene su i ostale naredbe.

Kod naredbi **let** i **print** potrebno je obaviti i izračun izraza na koji naredbe imaju referencu. Za to je definirana pomoćna rekurzivna metoda `calculateExpression` čija se izvedba opet temelji na prozivanje različitih vrsta izraza uporabom operatora **instanceof**.

Što možemo zaključiti o napisanom rješenju? Velika prednost rješenja je da ono *radi*. Dano rješenje nažalost iskazuje svojstvo krhkosti: primjerice, želimo li dodati novi binarni operator, ne samo da moramo napraviti novi razred koji ga modelira i prilagoditi parser da ga bude svjestan (koji se simbol koristi, kojeg je prioriteta u odnosu na druge operatore, je li lijevo ili desno asocijativan i slično), već se moramo sjetiti tražiti svugdje po kodu gdje smo imali `instanceof` pitalice te raditi korekcije i u takvim metodama. A mijenjenje postojećeg koda zbog ubacivanja nove funkcionalnosti direktno je kršenje načela nadogradnje bez promjene - i to želimo izbjeći kada je god to moguće.

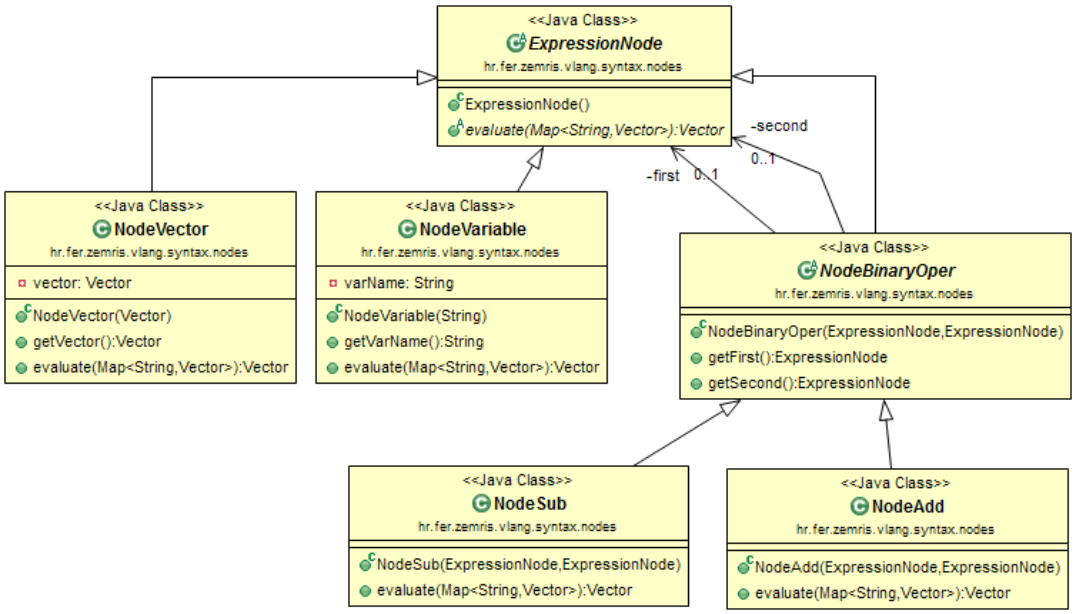
Izvođenje programa: pokušaj drugi

U ovom poglavlju pokušat ćemo popraviti jednu manu prethodnog rješenja: krhkost implementacije izračuna aritmetičkih izraza. S obzirom da želimo osigurati da dodavanjem novih vrsta izraza ne moramo mijenjati postojeći kod koji izvodi napisani program, prirodno rješenje je definirati na apstraktnoj razini algebarskog izraza apstraktnu metodu za izračun vrijednosti izraza i potom pisati kod koji se bavi izvođenjem programa na način da ovisi samo o toj metodi, odnosno da poziva isključivo nju. Potom je zadaća konkretnih implementacija izraza da na korektan način izračunaju svoju vrijednost. Primjerice, izraz koji predstavlja varijablu sam bi trebao pogledati u mapu postojećih varijabli i dohvatiti i vratiti svoju vrijednost. Izraz koji predstavlja binarni operator zbrajanja sam bi trebao pozvati izračun vrijednosti lijevog djeteta, izračun vri-

jednosti desnog djeteta i potom izračunati sumu tih rezultata i to vratiti kao svoj rezultat. Ovakvo rješenje prikazano je dijagramom razreda na slici 18.6. Apstraktni binarni operator modeliran je razredom `NodeBinaryOper` koji nasljeđuje razred `ExpressionNode` (jer primjerci binarnih operatora jesu izrazi) i istovremeno ima reference na dva druga izraza (reference označene strelicama uz koje piše `first` i `second`; brojnost prikazana uz te veze je `0..1` čime se sugerira da referenca ili može biti `null` ili pokazuje na jedan objekt; pri stvaranju objekata ovog tipa konstruktor bi se trebao pobrinuti da u slučaju predavanja `null` referenci baci iznimku tako da prikazane veze možemo tretirati kao da uvijek pokazuju na točno jedan drugi izraz).

Svim je izrazima dodana metoda `evaluate` koja prima referencu na mapu vrijednosti definiranih varijabli a vraća vektor. Implementacije kojima ta mapa ne treba ignorirat će je; implementacije kojima je mapa potrebna koristit će je. U određenom smislu, ova mapa predstavlja kontekstu unutar kojeg se obavlja računanje i zato je dostavljamo svima. U slučaju da je kontekst složeniji, opravdano bismo ga mogli enkapsulirati novim razredom i referencu na primjerak tog razreda davati kao jedini argument metode `evaluate`.

Slika 18.6. Modificirano podstablo za pamćenje izraza



Implementacije razreda koji predstavljaju različite vrste izraza prikazane su u izvornom kodu 18.12. Pri tome su ostavljeni samo dijelovi koda koji su novi u odnosu na početnu implementaciju.

Primjer 18.12. Izmjene razreda koji predstavljaju izraze

```
 1 public abstract class ExpressionNode {
 2
 3  // ostatak koda ...
 4
 5  /**
 6   * Apstraktna metoda za izračun vrijednosti trenutnog
 7   * izraza.
 8   * @param variables mapa definiranih varijabli
 9   * @return izračunatu vrijednost izraza
10   * @throws VLangExecutionException ako dode do
11   *         pogreške pri izračunu
12   */
13  public abstract Vector evaluate(Map<String, Vector> variables);
14 }
15
```

```
1 public class NodeVector extends ExpressionNode {
2
3  // ostatak koda ...
4
5  @Override
6  public Vector evaluate(Map<String, Vector> variables) {
7   return getVector();
8  }
9 }
10
```

```
1 public class NodeVariable extends ExpressionNode {
2
3  // ostatak koda ...
4
5  @Override
6  public Vector evaluate(Map<String, Vector> variables) {
7   Vector v = variables.get(varName);
8   if(v==null) {
9    if(!variables.containsKey(varName)) {
10    throw new VLangExecutionException(
11     "Undeclared variable "+varName+" in expression.");
12    } else {
13     throw new VLangExecutionException(
14      "Uninitialized variable "+varName+" in expression.");
15    }
16   }
17   return v;
18  }
19 }
20
```

```
1 public abstract class NodeBinaryOper extends ExpressionNode {
2
3  // ostatak koda ...
4
5 }
6
```

```
1 public class NodeAdd extends NodeBinaryOper {
2
3  // ostatak koda ...
4
5  @Override
6  public Vector evaluate(Map<String, Vector> variables) {
7   Vector left = getFirst().evaluate(variables);
8   Vector right = getSecond().evaluate(variables);
9   return left.add(right);
10  }
11 }
12
```

```
1 public class NodeSub extends NodeBinaryOper {
2
3  // ostatak koda ...
4
5  @Override
6  public Vector evaluate(Map<String, Vector> variables) {
7   Vector left = getFirst().evaluate(variables);
```

```
 8    Vector right = getSecond().evaluate(variables);
 9    return left.sub(right);
10   }
11
12 }
13
```

Koje smo izmjene napravili u kodu?

- Razred `ExpressionNode` učinili smo apstraktnim. Razlog tomu je dodavanje apstraktne metode `evaluate` za koju na ovom mjestu nemamo prikladne implementacije.
- Razred `NodeVector` dobio je implementaciju metode `evaluate` koja vraća pohranjenu vektorsku konstantu.
- Razred `NodeVariable` dobio je implementaciju metode `evaluate` koja vraća vrijednost koja je u mapi varijabli pridružena varijabli koju predstavlja primjerak ovog razreda odnosno koja baca iznimku ako takva varijabla još ne postoji ili ako nije inicijalizirana.
- Razred `NodeBinaryOper` postao je apstraktan jer ne nudi nikakvu implementaciju metode `evaluate` a nasljeđuje razred `ExpressionNode` koji je upravo zbog te metode apstraktan.
- Razredi `NodeAdd` i `NodeSub` sadrže odgovarajuću implementaciju metode `evaluate` vraća sumu ili razliku rekurzivno izračunatih podizraza.



Bilješka

Ideja enkapsulacije jednostavnih i složenih objekata kroz jedno unificirano sučelje koje od klijenata skriva informaciju s čime klijenti točno rade i koja omogućava da klijent na jednak način obavlja operacije s jednostavnim i složenim objektima centralna je ideja oblikovnog obrasca *Kompozit*. Kod ovog oblikovnog obrasca definira se jedinstveno sučelje koje sadrži definicije svih klijentima interesantnih metoda i potom se iz tog sučelja izvode implementacije jednostavnih objekata te implementacije složenih objekata koje sadrže reference na druge objekte istog apstraktnog tipa te u svojim implementacijama metoda obavljaju rekurzivno delegiranje sadržanim objektima. Pojednostavljenje koje nudi ovaj oblikovni obrazac temelji se na *rekurzivnoj kompoziciji* koja je sastavni dio i nekih drugih oblikovnih obrazaca (spomenimo primjerice *Dekorator*).

U našem primjeru apstraktno sučelje je `ExpressionNode` i ono definira metodu `evaluate`. Jednostavni objekti su primjerci razreda `NodeVector` i `NodeVariable` koji direktno ostvaruju metodu `evaluate`; složeni objekti su primjerci razreda izvedenih iz `NodeBinaryOper` koji sadrže reference na druge izraze preko istog apstraktnog sučelja te koji u metodi `evaluate` rekurzivno pozivaju evaluiranje tih objekata, kombiniraju rezultate u konačni rezultat i njega vraćaju.

Uz ovakvu implementaciju razreda koji modeliraju izraze sada možemo napisati jednostavniju verziju koda koji se bavi izvođenjem programa i koja je prikazana izvornim kodom 18.13.

Primjer 18.13. Poboljšana izvedba izvođenja programa

```
 1 package hr.fer.zemris.vlang.executors.improved;
 2
 3 import hr.fer.zemris.vlang.VLangException;
 4 import hr.fer.zemris.vlang.Vector;
 5 import hr.fer.zemris.vlang.executors.VLangExecutionException;
 6 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
 7 import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
 8 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
 9 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
10 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
```

```
11 import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
12
13 import java.util.HashMap;
14 import java.util.Map;
15
16 /**
17  * Poboljšana inačica stroja za izvođenje programa napisanog
18  * jezikom <i>vlang</i> koja ne zahtjeva promjene u kodu ako
19  * se mijenja broj i vrsta operatora koji se mogu koristiti
20  * u izrazima.
21  *
22  * @author marcupic
23  */
24 public class ExecutorImproved {
25
26  /**
27   * Vršni čvor programa.
28   */
29  private ProgramNode programNode;
30  /**
31   * Mapa definiranih varijabli.
32   */
33  private Map<String, Vector> variables;
34
35  /**
36   * Konstruktor.
37   * @param programNode vršni čvor programa
38   *        koji treba izvesti
39   */
40  public ExecutorImproved(ProgramNode programNode) {
41   this.programNode = programNode;
42  }
43
44  /**
45   * Metoda izvodi program predan u konstruktoru.
46   *
47   * @throws VLangExecutionException ako dođe do pogreške
48   *         pri izvođenju programa
49   */
50  public void execute() {
51   // Obriši mapu definiranih varijabli
52   variables = new HashMap<>();
53   // Izvedi svaku naredbu:
54   for(VLangNode node : programNode.getStatements()) {
55    // Ako je trenutna naredba naredba "def":
56    if(node instanceof DefStatement) {
57     DefStatement def = (DefStatement)node;
58     for(String varName : def.getVariables()) {
59      if(variables.containsKey(varName)) {
60       throw new VLangException(
61        "Variable "+varName+" alredy defined.");
62      } else {
63       variables.put(varName, null);
64      }
65     }
66     continue;
67    }
68    // Ako je trenutna naredba naredba "let":
```

```
69     if(node instanceof LetStatement) {
70      LetStatement let = (LetStatement)node;
71      if(!variables.containsKey(let.getVarName())) {
72       throw new VLangException(
73        "Undeclared variable "+let.getVarName() +
74        " in left side of let statement.");
75      }
76      variables.put(
77       let.getVarName(),
78       let.getExpression().evaluate(variables));
79      continue;
80     }
81     // Ako je trenutna naredba naredba "print":
82     if(node instanceof PrintStatement) {
83      PrintStatement print = (PrintStatement)node;
84      for(ExpressionNode exp : print.getList()) {
85       System.out.println(exp.evaluate(variables));
86      }
87     }
88    }
89   }
90
91 }
92
```

Uočite kako je dobiveni kod postao bitno kraći i kako u njemu više niti na jednom mjestu nema niti spomena bilo kakve konkretne implementacije izraza: umjesto toga, svaka naredba koja računa vrijednost izraza ima referencu na apstraktni izraz i nad njim jednostavno poziva `Vector result = exp.evaluate(variables);`.

Koje su prednosti a koje mane uporabe oblikovnog obrasca *Kompozit* (i načina kako smo ga primijenili) nad izrazima? Osnovna prednost je mogućnost pisanja koda koji se bavi izvršavanjem programa na način koji ne zahtjeva nikakve promjene u njemu, ako se danas-sutra odlučimo dodati još jedan ili dva binarna operatora. Detalje o načinu izračuna tih operatora znat će same implementacije operatora a kako naš kod s njima razgovara kroz jasno definirano sučelje, nikakve promjene na toj strani neće biti potrebne.

Rješenje, međutim, ima i lošu stranu: naš domenski model podataka postao je opterećen kodom koji obavlja konkretnu operaciju nad tim modelom. Želimo li na isti način dodati operaciju konverzije proizvoljnog izraza u string, uz apstraktnu metodu `evaluate` dodali bismo i apstraktnu metodu `serializeToString`, i potom nju implementirali u svim jednostavnim i složenim objektima. Za svaku novu operaciju koju bismo htjeli dodati nad modelom, trebali bismo u svaki razred modela dodati po odgovarajuću apstraktnu metodu. A time direktno kršimo načelo nadogradnje bez promjene. Dodatno poboljšanje i razrada ovog pristupa koja rješava navedeni problem prikazana je u sljedećem odjeljku.

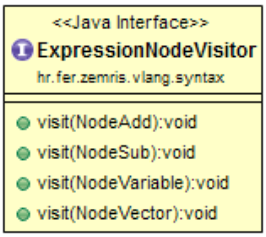
Izvođenje programa: pokušaj treći

Rezimirajmo što smo uspjeli postići prethodnim pokušajem: dobili smo implementaciju izraza koja omogućava transparentno dodavanje novih vrsta izraza bez potrebe da mijenjamo kod koji se bavi izvršavanjem (odnosno izračunavanjem) izraza. To smo postigli primjenom oblikovnog obrasca *Kompozit* čiji je temelj rekurzivna kompozicija. Loša strana našeg načina implementacije tog rješenja je čvrsto povezivanje konkretnih operacija koje želimo obavljati nad modelom sa samim modelom.

Pitanje koje se postavlja je sljedeće: je li moguće koristiti benefite koje dobivamo uporabom *Kompozita* a istovremeno ne povezati model podataka s jednom (konkretnom) operacijom? Odgovor na to pitanje je potvrdan, i jedna od modifikacija prikazanog rješenja dovest će nas do oblikovnog obrasca *Posjetitelj*. No krenimo redom.

Prvi korak u razdvajanju konkretne operacije od modela je apstrahiranje operacije. Definirajmo stoga sučelje `ExpressionNodeVisitor` koje za svaku vrstu izraza definira jednu apstraktnu metodu koja zna obaviti traženu operaciju nad tim izrazom. Slika 18.7 odnosno izvorni kod 18.14 prikazuje jednu moguću definiciju takvog sučelja.

Slika 18.7. Apstraktna operacija



Primjer 18.14. Definicija apstraktne operacije nad izrazom

```
 1 package hr.fer.zemris.vlang.syntax;
 2
 3 import hr.fer.zemris.vlang.syntax.nodes.NodeAdd;
 4 import hr.fer.zemris.vlang.syntax.nodes.NodeSub;
 5 import hr.fer.zemris.vlang.syntax.nodes.NodeVariable;
 6 import hr.fer.zemris.vlang.syntax.nodes.NodeVector;
 7
 8 /**
 9  * Apstraktni posjetitelj izraza.
10  *
11  * @author marcupic
12  */
13 public interface ExpressionNodeVisitor {
14
15  /**
16   * Operacija posjećivanja čvora {@link NodeAdd}.
17   * @param add čvor
18   */
19  public void visit(NodeAdd add);
20  /**
21   * Operacija posjećivanja čvora {@link NodeSub}.
22   * @param add čvor
23   */
24  public void visit(NodeSub sub);
25  /**
26   * Operacija posjećivanja čvora {@link NodeVariable}.
27   * @param add čvor
28   */
29  public void visit(NodeVariable var);
30  /**
31   * Operacija posjećivanja čvora {@link NodeVector}.
32   * @param add čvor
33   */
34  public void visit(NodeVector vector);
35
36 }
37
```

Pretpostavimo sada da imamo konkretnu implementaciju ovog sučelja u kojem je svaka metoda implementirana tako da zna obaviti traženu operaciju nad svakim od podržanih konkret-

nih razreda. Pretpostavimo također da imamo listu stvorenih izraza. Tada bismo mogli napisati sljedeći kod koji bi korektno za svaki izraz pozvao korektnu inačicu metode nad primjerkom razreda koji implementira zadanu operaciju.

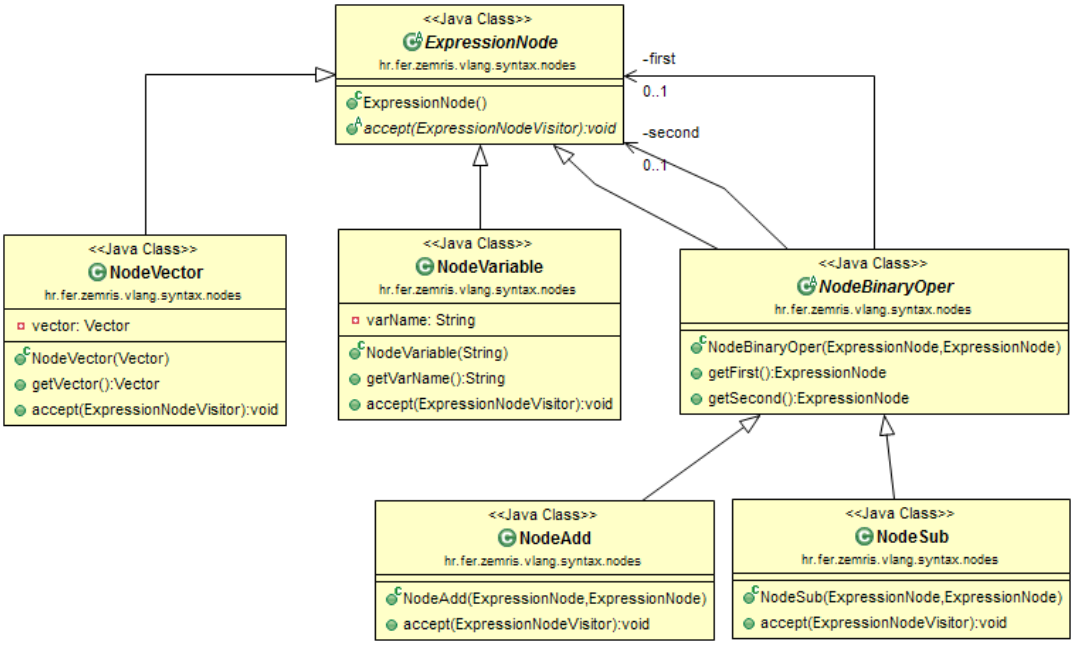
```
List<ExpressionNode> nodes = getExpressions();
ExpressionNodeVisitor visitor = getConcreteVisitor();

for(ExpressionNode expr : nodes) {
 if(expr instanceof NodeVector) {
  NodeVector vect = (NodeVector)expr;
  visitor.visit(vect);
 } else if(expr instanceof NodeVariable) {
  NodeVariable var = (NodeVariable)expr;
  visitor.visit(var);
 } else if(expr instanceof NodeAdd) {
  NodeAdd add = (NodeAdd)expr;
  visitor.visit(add);
 } else if(expr instanceof NodeSub) {
  NodeSub sub = (NodeSub)expr;
  visitor.visit(sub);
 }
}
```

Konkretnu operaciju stvara i vraća metoda `getConcreteVisitor` dok listu izraza konstruira i vraća metoda `getExpressions` (njihova implementacija ovdje nije bitna).

Imamo li sve operacije riješene na opisani način, prethodni generički kod može proizvoljnu operaciju (zadanu kroz apstraktno sučelje) primijeniti na proizvoljni konretan izraz. Jedini preostali problem prethodnog koda je uvođenje `instanceof`-pitalice, kako bismo znali pozvati korektnu operaciju za korektan tip izraza. Ovaj problem, međutim, znamo riješiti: oblikovni obrazac *Kompozit* omogućava da svaki konkretan tip sam definira dio koda koji je za njega specifičan, a koji klijent poziva kroz apstraktno sučelje zahvaljujući polimorfizmu. Ono što bismo htjeli jest pojedine grane gornje `instanceof`-pitalice razmjestiti u pripadne razrede modela (kao što smo to radili s implementacijama metode `evaluate` u drugom pokušaju reorganizacije koda). Stoga ćemo ovom prilikom početni model (podstablo) izraza korigirati tako da u vršni apstraktni razred dodamo apstraktnu metodu `void acceptVisitor(ExpressionNodeVisitor visitor);` te u izvedene razrede dodamo konkretne implementacije koje u apstraktnoj operaciji pozivaju upravo pravu metodu koja odgovara tipu samog izraza. Potrebne korekcije u razredima prikazane su u dijagramu razreda na slici 18.8.

Slika 18.8. Apstraktna operacija



Izvorni kod 18.15 prikazuje provedene korekcije u kodu.

Primjer 18.15. Uvođenje potpore za primjenu oblikovnog obrasca *Posjetitelj* u model izraza

```
1 public abstract class ExpressionNode {
2
3  // Ostatak koda ...
4
5  /**
6   * Apstraktna metoda koja prihvaća posjetitelja izraza.
7   *
8   * @param visitor posjetitelj
9   */
10  public abstract void accept(ExpressionNodeVisitor visitor);
11
12 }
13
```

```
1 public class NodeVector extends ExpressionNode {
2
3  // Ostatak koda ...
4
5  @Override
6  public void accept(ExpressionNodeVisitor visitor) {
7   visitor.visit(this);
8  }
9 }
10
```

```
1 public class NodeVariable extends ExpressionNode {
2
3  // Ostatak koda ...
4
5  @Override
```

```
 6  public void accept(ExpressionNodeVisitor visitor) {
 7   visitor.visit(this);
 8  }
 9 }
10
```

```
 1 public abstract class NodeBinaryOper extends ExpressionNode {
 2
 3  // Ostatak koda ...
 4
 5 }
 6
```

```
 1 public class NodeAdd extends NodeBinaryOper {
 2
 3  // Ostatak koda ...
 4
 5  @Override
 6  public void accept(ExpressionNodeVisitor visitor) {
 7   visitor.visit(this);
 8  }
 9 }
10
```

```
 1 public class NodeSub extends NodeBinaryOper {
 2
 3  // Ostatak koda ...
 4
 5  @Override
 6  public void accept(ExpressionNodeVisitor visitor) {
 7   visitor.visit(this);
 8  }
 9 }
10
```

Zašto ovo radi korektno, kada smo u sve razrede dodali praktički isti kod: `visitor.visit(this);`? Radi se o tome da prilikom prevođenja metode `accept` napisane u razredu `NodeAdd` prevoditelj referencu `this` tretira kao referencu na objekt tipa `NodeAdd`. Stoga generira kod koji nad predanim objektom `visitor` poziva metodu čiji je prototip upravo `void visit(NodeAdd add);`, što je i željena metoda za obradu primjeraka tog razreda.

Jednom kada smo u model ugradili ovu potporu, prethodni isječak koda koji je sadržavao `instanceof`-pitalicu sada možemo napisati puno jednostavnije, što je prikazano u nastavku.

```
List<ExpressionNode> nodes = getExpressions();
ExpressionNodeVisitor visitor = getConcreteVisitor();

for(ExpressionNode expr : nodes) {
 expr.accept(visitor);
}
```

Da bismo razumjeli što se ovdje događa, važno je uočiti mehanizam poznat pod nazivom *dvostruko odašiljanje* (engl. *double-dispatch*): najprije naš klijentski kod polimorfnim pozivom nad apstraktnim izrazom poziva nadjačanu metodu `accept` koja je definirana upravo nad konkretnim izrazom (ako je stvarni tip izraza primjerice `NodeAdd`, onda se poziva upravo metoda `accept` definirana u razredu `NodeAdd`). Potom se iz tog koda novim polimorfnim pozivom poziva metoda operacije (posjetitelja) koja je namjenjena obradi baš tog konkretnog tipa izraza (za to se je pobrinuo prevodilac prilikom prevođenja koda metode `accept`). Zahvaljujući ovoj međuigri konkretnih izraza i posjetitelja odnosno povezivanju oblikovnih obrazaca *Kompozit* i *Posjetitelj* moguće je dobiti domenski model koji ne ovisi o konkretnim operacijama.

Da bismo ilustrirali primjenu ova dva obrasca na izraze, pogledajmo kako bismo napisali dvije operacije: jednu koja računa vrijednost izraza te drugu koja izraz serijalizira (pretvara) u tekst. Implementacija prve operacije dana je u izvornom kodu 18.16 a druge u izvornom kodu 18.17.

Primjer 18.16. Posjetitelj koji računa vrijednost izraza

```
 1 package hr.fer.zemris.vlang.syntax.nodes.visitors;
 2
 3 import java.util.Map;
 4 import java.util.Stack;
 5
 6 import hr.fer.zemris.vlang.Vector;
 7 import hr.fer.zemris.vlang.executors.VLangExecutionException;
 8 import hr.fer.zemris.vlang.syntax.ExpressionNodeVisitor;
 9 import hr.fer.zemris.vlang.syntax.nodes.NodeAdd;
10 import hr.fer.zemris.vlang.syntax.nodes.NodeSub;
11 import hr.fer.zemris.vlang.syntax.nodes.NodeVariable;
12 import hr.fer.zemris.vlang.syntax.nodes.NodeVector;
13
14 public class ExpressionEvalVisitor
15  implements ExpressionNodeVisitor {
16
17  private Stack<Vector> stack = new Stack<>();
18  private Map<String, Vector> variables;
19
20  public ExpressionEvalVisitor(Map<String, Vector> variables) {
21   this.variables = variables;
22  }
23
24  @Override
25  public void visit(NodeAdd add) {
26   add.getFirst().accept(this);
27   add.getSecond().accept(this);
28   Vector right = stack.pop();
29   Vector left = stack.pop();
30   stack.push(left.add(right));
31  }
32
33  @Override
34  public void visit(NodeSub sub) {
35   sub.getFirst().accept(this);
36   sub.getSecond().accept(this);
37   Vector right = stack.pop();
38   Vector left = stack.pop();
39   stack.push(left.sub(right));
40  }
41
42  @Override
43  public void visit(NodeVariable var) {
44   String varName = var.getVarName();
45   Vector v = variables.get(varName);
46   if(v==null) {
47    if(!variables.containsKey(varName)) {
48     throw new VLangExecutionException(
49      "Undeclared variable "+varName+" in expression.");
50    } else {
51     throw new VLangExecutionException(
52      "Uninitialized variable "+varName+" in expression.");
```

```
53     }
54    }
55    stack.push(v);
56   }
57
58   @Override
59   public void visit(NodeVector vector) {
60    stack.push(vector.getVector());
61   }
62
63   public Vector getResult() {
64    if(stack.size() != 1) {
65     throw new VLangExecutionException(
66       "Evaluation of expression failed!");
67    }
68    return stack.peek();
69   }
70
71 }
72
```

Primjer 18.17. Posjetitelj koji izraz pretvara u tekst

```
 1 package hr.fer.zemris.vlang.syntax.nodes.visitors;
 2
 3 import hr.fer.zemris.vlang.syntax.ExpressionNodeVisitor;
 4 import hr.fer.zemris.vlang.syntax.nodes.NodeAdd;
 5 import hr.fer.zemris.vlang.syntax.nodes.NodeSub;
 6 import hr.fer.zemris.vlang.syntax.nodes.NodeVariable;
 7 import hr.fer.zemris.vlang.syntax.nodes.NodeVector;
 8
 9 public class ExpressionPrintVisitor
10  implements ExpressionNodeVisitor {
11
12  /**
13   * Izlaz u koji se zapisuje izvorni kod izraza.
14   */
15  private StringBuilder sb;
16
17  /**
18   * Konstruktor.
19   * @param sb izlaz u koji treba zapisati izvorni kod
20   */
21  public ExpressionPrintVisitor(StringBuilder sb) {
22   this.sb = sb;
23  }
24
25  @Override
26  public void visit(NodeAdd add) {
27   sb.append('(');
28   add.getFirst().accept(this);
29   sb.append('+');
30   add.getSecond().accept(this);
31   sb.append(')');
32  }
33
34  @Override
35  public void visit(NodeSub sub) {
```

```
36    sb.append('(');
37    sub.getFirst().accept(this);
38    sb.append('-');
39    sub.getSecond().accept(this);
40    sb.append(')');
41   }
42
43   @Override
44   public void visit(NodeVariable var) {
45    String varName = var.getVarName();
46    sb.append(varName);
47   }
48
49   @Override
50   public void visit(NodeVector vector) {
51    sb.append(vector.getVector().toString());
52   }
53  }
54
```

Pogledajmo sada kako će izgledati kod koji je zadužen za izvođenje programa uz ovako modificirani kod. Rješenje prikazuje izvorni kod 18.18.

Primjer 18.18. Poboljšani kod za izvođenje programa

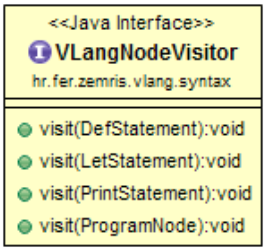
```
 1 package hr.fer.zemris.vlang.executors.visitors;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5
 6 import hr.fer.zemris.vlang.Vector;
 7 import hr.fer.zemris.vlang.executors.VLangExecutionException;
 8 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
 9 import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
10 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
11 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
12 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
13 import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
14 import hr.fer.zemris.vlang.syntax.nodes.visitors.ExpressionEvalVisitor;
15
16 /**
17  * Inačica stroja za izvođenje programa napisanog
18  * jezikom <i>vlang</i> koju ne treba mijenjati ako dođe do
19  * bilo kakve promjene u broju i vrsti operatora i operanada
20  * koji se mogu pojaviti u izrazima.
21  *
22  * @author marcupic
23  */
24 public class ExecutorVisitor1 {
25
26  /**
27   * Vršni čvor programa.
28   */
29  private ProgramNode programNode;
30  /**
31   * Mapa definiranih varijabli.
32   */
33  private Map<String, Vector> variables;
```

```
34
35  /**
36   * Konstruktor.
37   * @param programNode vršni čvor programa koji treba izvesti
38   */
39  public ExecutorVisitor1(ProgramNode programNode) {
40   this.programNode = programNode;
41  }
42
43  /**
44   * Metoda izvodi program predan u konstruktoru.
45   *
46   * @throws VLangExecutionException ako dođe do pogreške
47   *         pri izvođenju programa
48   */
49  public void execute() {
50   // Obriši mapu definiranih varijabli
51   variables = new HashMap<>();
52   // Izvedi svaku naredbu:
53   for(VLangNode node : programNode.getStatements()) {
54    // Ako je trenutna naredba naredba "def":
55    if(node instanceof DefStatement) {
56     DefStatement def = (DefStatement)node;
57     for(String varName : def.getVariables()) {
58      if(variables.containsKey(varName)) {
59       throw new VLangExecutionException(
60        "Variable "+varName+" alredy defined.");
61      } else {
62       variables.put(varName, null);
63      }
64     }
65     continue;
66    }
67    // Ako je trenutna naredba naredba "let":
68    if(node instanceof LetStatement) {
69     LetStatement let = (LetStatement)node;
70     if(!variables.containsKey(let.getVarName())) {
71      throw new VLangExecutionException(
72       "Undeclared variable " + let.getVarName() +
73       " in left side of let statement.");
74     }
75     Vector v = calculateExpression(let.getExpression());
76     variables.put(let.getVarName(), v);
77     continue;
78    }
79    // Ako je trenutna naredba naredba "print":
80    if(node instanceof PrintStatement) {
81     PrintStatement print = (PrintStatement)node;
82     for(ExpressionNode exp : print.getList()) {
83      System.out.println(calculateExpression(exp));
84     }
85    }
86   }
87  }
88
89  /**
90   * Pomoćna rekurzivna metoda za izračun vrijednosti izraza.
91   * @param node čvor koji predstavlja trenutni izraz
```

```
 92   * @return izračunata vrijednost izraza
 93   */
 94  private Vector calculateExpression(ExpressionNode node) {
 95   ExpressionEvalVisitor visitor =
 96    new ExpressionEvalVisitor(variables);
 97   node.accept(visitor);
 98   return visitor.getResult();
 99  }
100
101 }
102
```

U predloženom rješenju imamo još jedno mjesto za dodatno poboljšanje: razred `ExecutorVisitor1` u metodi `execute` za korektno izvođenje pojedinih vrsta naredbi koristi `instanceof`-pitalicu. To je isti problem koji smo već imali kod izračuna izraza i riješit ćemo ga isti način: kombinacijom oblikovnih obrazaca *Posjetitelj* (za definiranje apstraktnih operacija nad programom - izvođenje programa samo je jedna moguća operacija), te *Kompozit* (za klijentski transparentno obavljanje metode `accept` nad različitim vrstama naredbe). Stoga ćemo napraviti sljedeće izmjene u kodu: najprije ćemo definirati apstraktnog posjetitelja za provođenje operacija nad stablom naredbi (slika 18.9, izvorni kod 18.19) a potom u model naredbi dodati potporu za prihvaćanje posjetitelja (dijagram razreda na slici 18.10, izvorni kod 18.20).

Slika 18.9. Apstraktna operacija nad naredbama

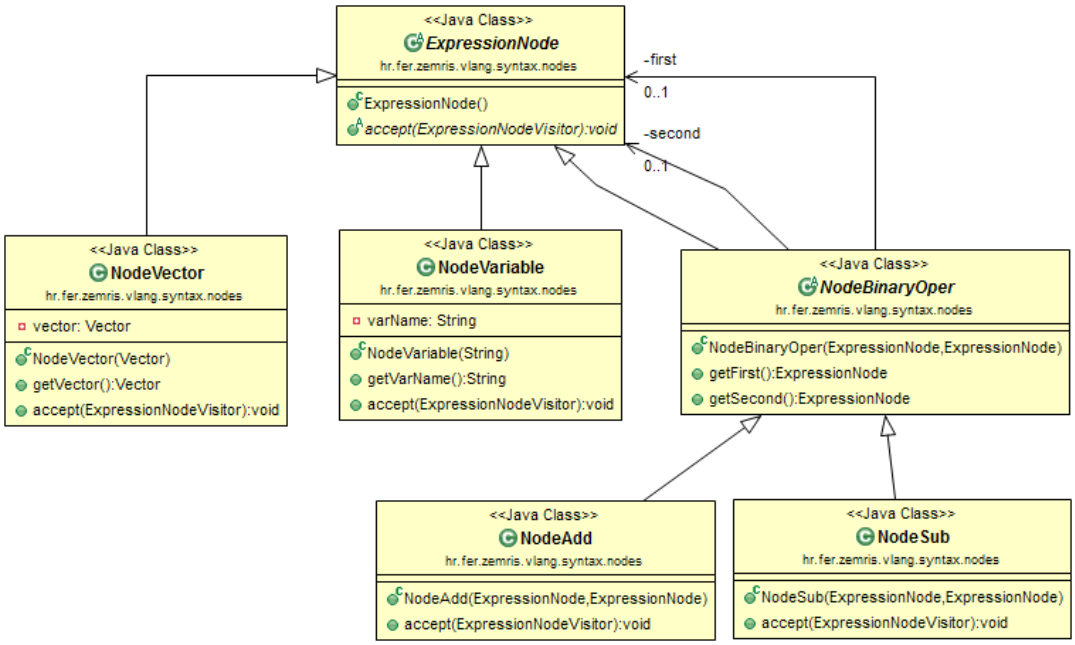


Primjer 18.19. Apstraktna definicija operacije nad naredbama

```
 1 package hr.fer.zemris.vlang.syntax;
 2
 3 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
 4 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
 5 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
 6 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
 7
 8 /**
 9  * Apstraktni posjetitelj naredbi jezika <i>vlang</i>.
10  *
11  * @author marcupic
12  */
13 public interface VLangNodeVisitor {
14
15  /**
16   * Obrada naredbe "def" ({@link DefStatement}).
17   * @param stmt naredba
18   */
19  public void visit(DefStatement stmt);
20  /**
21   * Obrada naredbe "let" ({@link LetStatement}).
22   * @param stmt naredba
23   */
24  public void visit(LetStatement stmt);
```

```
25  /**
26   * Obrada naredbe "print" ({@link PrintStatement}).
27   * @param stmt naredba
28   */
29  public void visit(PrintStatement stmt);
30  /**
31   * Obrada slijeda naredbi ({@link ProgramNode}).
32   * @param node slijed naredbi
33   */
34  public void visit(ProgramNode node);
35
36 }
37
```

Slika 18.10. Apstraktna operacija



Primjer 18.20. Modifikacije modela naredbe

```
1 public abstract class VLangNode {
2
3  // Ostatak koda ...
4
5  /**
6   * Prihvat posjetitelja.
7   * @param visitor posjetitelj
8   */
9  public abstract void accept(VLangNodeVisitor visitor);
10 }
11
```

```
1 public class ProgramNode extends VLangNode {
2
3  // Ostatak koda ...
4
5  @Override
6  public void accept(VLangNodeVisitor visitor) {
7   visitor.visit(this);
```

```
 8   }
 9  }
10
```

```
 1  public class DefStatement extends VLangNode {
 2
 3   // Ostatak koda ...
 4
 5   @Override
 6   public void accept(VLangNodeVisitor visitor) {
 7    visitor.visit(this);
 8   }
 9  }
10
```

```
 1  public class LetStatement extends VLangNode {
 2   // Ostatak koda ...
 3
 4   @Override
 5   public void accept(VLangNodeVisitor visitor) {
 6    visitor.visit(this);
 7   }
 8  }
 9
```

```
 1  public class PrintStatement extends VLangNode {
 2
 3   // Ostatak koda ...
 4
 5   @Override
 6   public void accept(VLangNodeVisitor visitor) {
 7    visitor.visit(this);
 8   }
 9  }
10
```

Sada možemo dati primjere dvaju konkretnih posjetitelja nad naredbama: prvi posjetitelj prikazan je izvornim kodom 18.21 i zadaća mu je izvođenje programa čije stablo dobiva u konstruktoru. Drugi primjer je posjetitelj prikazan izvornim kodom 18.22 čija je zadaća rekonstruirati izvorni kod programa iz sintaksnog stabla.

Primjer 18.21. Posjetitelj koji izvodi napisani program

```
 1  package hr.fer.zemris.vlang.syntax.nodes.visitors;
 2
 3  import java.util.HashMap;
 4  import java.util.Map;
 5
 6  import hr.fer.zemris.vlang.VLangException;
 7  import hr.fer.zemris.vlang.Vector;
 8  import hr.fer.zemris.vlang.syntax.VLangNodeVisitor;
 9  import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
10  import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
11  import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
12  import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
13  import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
14  import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
15
16  public class ProgramExecutorVisitor
```

```
17  implements VLangNodeVisitor {
18
19  /**
20   * Mapa definiranih varijabli.
21   */
22  private Map<String, Vector> variables = new HashMap<>();
23
24  @Override
25  public void visit(DefStatement stmt) {
26   for(String varName : stmt.getVariables()) {
27    if(variables.containsKey(varName)) {
28     throw new VLangExecutionException(
29      "Variable "+varName+" alredy defined.");
30    } else {
31     variables.put(varName, null);
32    }
33   }
34  }
35
36  @Override
37  public void visit(LetStatement stmt) {
38   if(!variables.containsKey(stmt.getVarName())) {
39    throw new VLangExecutionException(
40     "Undeclared variable " + stmt.getVarName() +
41     " in left side of let statement.");
42   }
43   Vector v = calculateExpression(stmt.getExpression());
44   variables.put(stmt.getVarName(), v);
45  }
46
47  @Override
48  public void visit(PrintStatement stmt) {
49   for(ExpressionNode exp : stmt.getList()) {
50    System.out.println(calculateExpression(exp));
51   }
52  }
53
54  @Override
55  public void visit(ProgramNode node) {
56   for(VLangNode stmt : node.getStatements()) {
57    stmt.accept(this);
58   }
59  }
60
61  /**
62   * Pomoćna rekurzivna metoda za izračun vrijednosti izraza.
63   * @param node čvor koji predstavlja trenutni izraz
64   * @return izračunata vrijednost izraza
65   */
66  private Vector calculateExpression(ExpressionNode node) {
67   ExpressionEvalVisitor visitor =
68    new ExpressionEvalVisitor(variables);
69   node.accept(visitor);
70   return visitor.getResult();
71  }
72 }
73
```

Primjer 18.22. Posjetitelj koji rekonstruira izvorni kod programa

```
package hr.fer.zemris.vlang.syntax.nodes.visitors;

import hr.fer.zemris.vlang.syntax.ExpressionNodeVisitor;
import hr.fer.zemris.vlang.syntax.VLangNodeVisitor;
import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
import hr.fer.zemris.vlang.syntax.nodes.VLangNode;

public class ProgramPrintVisitor implements VLangNodeVisitor {

 /**
  * Izlaz u koji se zapisuje izvorni kod programa.
  */
 private StringBuilder sb = new StringBuilder();

 @Override
 public void visit(DefStatement stmt) {
  sb.append("def ");
  boolean first = true;
  for(String varName : stmt.getVariables()) {
   if(first) {
    first = false;
   } else {
    sb.append(", ");
   }
   sb.append(varName);
  }
  sb.append(": vector;\r\n");
 }

 @Override
 public void visit(LetStatement stmt) {
  sb.append("let ").append(stmt.getVarName()).append(" = ");
  stmt.getExpression().accept(new ExpressionPrintVisitor(sb));
  sb.append(";\r\n");
 }

 @Override
 public void visit(PrintStatement stmt) {
  ExpressionNodeVisitor expVisitor =
   new ExpressionPrintVisitor(sb);
  sb.append("print ");
  boolean first = true;
  for(ExpressionNode node : stmt.getList()) {
   if(first) {
    first = false;
   } else {
    sb.append(", ");
   }
   node.accept(expVisitor);
  }
  sb.append(";\r\n");
 }
```

```
57
58  @Override
59  public void visit(ProgramNode node) {
60   for(VLangNode stmt : node.getStatements()) {
61    stmt.accept(this);
62   }
63  }
64
65  /**
66   * Dohvat generiranog izvornog koda.
67   *
68   * @return izvorni kod
69   */
70  public String getSourceCode() {
71   return sb.toString();
72  }
73 }
74
```

Uporabom razvijenog posjetitelja implementacija razreda koji izvodi program temeljem sintaksnog stabla dana je izvornim kodom 18.23.

Primjer 18.23. Konačna implementacija koda koji izvodi napisani program

```
 1 package hr.fer.zemris.vlang.executors.visitors;
 2
 3 import hr.fer.zemris.vlang.executors.VLangExecutionException;
 4 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
 5 import hr.fer.zemris.vlang.syntax.nodes.visitors.ProgramExecutorVisitor;
 6
 7 /**
 8  * Inačica stroja za izvođenje programa napisanog
 9  * jezikom <i>vlang</i> koju ne treba mijenjati
10  * niti ako se dodaju nove naredbe, niti ako dođe do
11  * bilo kakve promjene u broju i vrsti operatora i
12  * operanada koji se mogu pojaviti u izrazima.
13  *
14  * @author marcupic
15  */
16 public class ExecutorVisitor2 {
17
18  /**
19   * Vršni čvor programa.
20   */
21  private ProgramNode programNode;
22
23  /**
24   * Konstruktor.
25   * @param programNode vršni čvor programa koji treba izvesti
26   */
27  public ExecutorVisitor2(ProgramNode programNode) {
28   this.programNode = programNode;
29  }
30
31  /**
32   * Metoda izvodi program predan u konstruktoru.
33   *
```

```
34   * @throws VLangExecutionException ako dođe do pogreške
35   *         pri izvođenju programa
36   */
37  public void execute() {
38   ProgramExecutorVisitor exec = new ProgramExecutorVisitor();
39   programNode.accept(exec);
40  }
41
42 }
43
```

Na kraju prikazane studije slučaja prikažimo kako prevesti i izvesti konkretan program uporabom posljednje verzije koda. Primjer je prikazan izvornim kodom 18.24

Primjer 18.24. Primjer pokretanja prevođenja i izvođenja koda

```
 1 package test;
 2
 3 import hr.fer.zemris.vlang.executors.visitors.ExecutorVisitor2;
 4 import hr.fer.zemris.vlang.lexical.VLangTokenizer;
 5 import hr.fer.zemris.vlang.syntax.VLangParser;
 6 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
 7
 8 /**
 9  * Program koji predstavlja demonstraciju izvođenja programa
10  * napisanog jezikom <i>vlang</i>. Izvođenje je ostvareno
11  * implementacijom {@link ExecutorSimple}.
12  *
13  * @author marcupic
14  */
15 public class TestExecutorVisitor2 {
16
17  /**
18   * Metoda s kojom započinje izvođenje programa. Argumenti
19   * se ignoriraju.
20   * @param args argumenti naredbenog retka
21   */
22  public static void main(String[] args) {
23   String program = "def a, b: vector;\r\n" +
24     "def c: vector;\r\n" +
25     "let a = [1.5, 2.8];\r\n" +
26     "let b = [2, 5];\r\n" +
27     "let c = a - (b + [1.2,1]);\r\n" +
28     "print a, b, c;"
29     ;
30
31   VLangTokenizer tokenizer = new VLangTokenizer(program);
32   VLangParser parser = new VLangParser(tokenizer);
33
34   ProgramNode programNode = parser.getProgramNode();
35
36   new ExecutorVisitor2(programNode).execute();
37  }
38
39 }
40
```

Razmislite sada o sljedećim pitanjima i pokušajte vidjeti kako biste ih mogli uklopiti u posljednju inačicu koda (što je sve potrebno promijeniti).

- Želimo dodati mogućnosti pisanja izraza poput `~a` pri čemu se to evaluira kao vektor koji je trenutno pohranjen u varijabli i na koji se potom u sve komponente dodaje slučajni realni broj iz intervala od 0 do 1.

- Želimo dodati naredbu `input` koja bi se u programu koristila za čitanje vektora s tipkovnice; primjerice, ako u programu definiramo naredbu `input a;`, naredba bi od korisnika trebala preko tipkovnice učitati jedan vektor i pohraniti ga u varijablu `a`.

- Želimo u izraze dodati operator `*` koji djeluje nad vektorima slično kao kao i operatori zbrajanja i oduzimanja: *i*-ta komponenta rezultata jednaka je umnošku *i*-tih komponenata vektora-operanada.



Bilješka

Cjelokupni programski kodovi projekata za razvojno okruženje Eclipse svih ovdje opisanih primjera dostupni su na adresi http://java.zemris.fer.hr/nastava/opjj/

Korišteni oblikovni obrasci

U okviru ovog poglavlja koristili smo dva oblikovna obrasca: *Kompozit* te *Posjetitelj*. Stoga ćemo u nastavku dati i njihov formalni opis.

Oblikovni obrazac *Kompozit*

Osnovna namjena oblikovnog obrasca *Kompozit* jest omogućiti klijentima da na jedinstven način tretiraju jednostavne i složene objekte. Ideja je izgraditi stablo komponenata koje je izvedeno iz osnovne komponente. Iz te komponente izvodimo sve primitivne komponente te kompozitne komponente. Kompozitne komponente sadrže reference na druge osnovne komponente čime postižemo rekurzivnu kompoziciju. Također, jednostavno je dodavati nove vrste osnovnih komponenata: to možemo postići u skladu s načelom nadogradnje bez promjene.

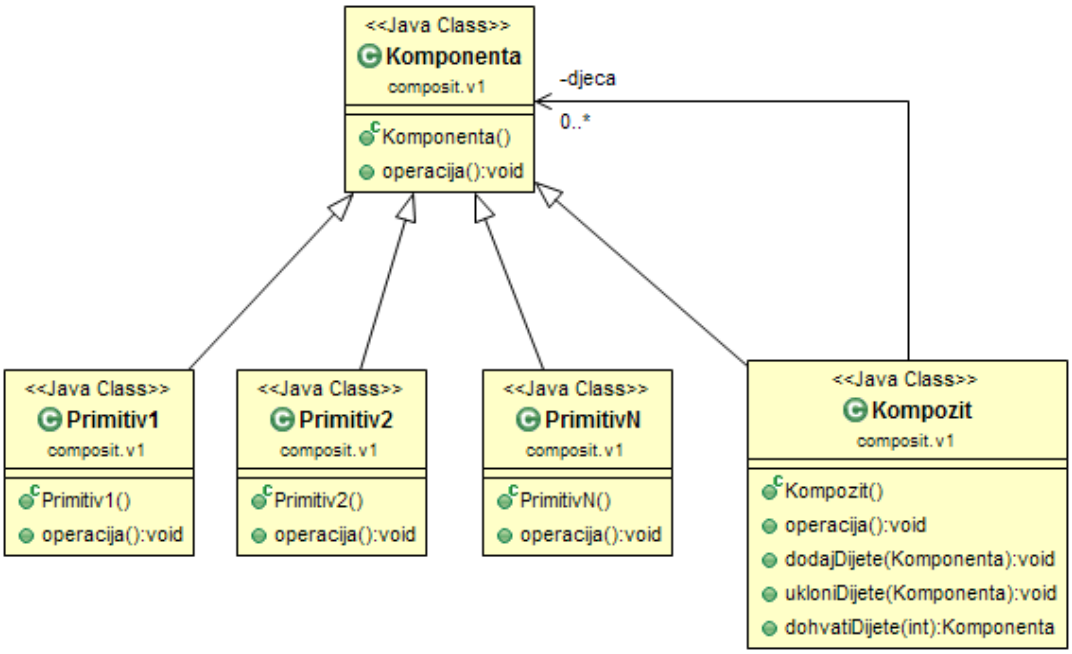
Ovom oblikovnom obrascu možemo pridružiti nekoliko sličnih strukturnih dijagrama. Jedna mogućnost prikazana je na slici 18.11. Pogledajmo koji su sudionici ovog oblikovnog obrasca.

Komponenta. Vršni razred je razred `Komponenta`. On definira sve operacije koje klijenti žele raditi s konkretnim primjercima takvih komponenata. U ovom slučaju ilustrirana je samo jedna generička metoda imena `operacija`.

Primitivi. Iz razreda `Komponenta` izvedeno je nekoliko jednostavnih komponenata - takozvanih primitiva. To su komponente koje imaju različite implementacije metode `operacija`.

Kompozit. Konačno, iz razreda `Komponenta` izveden je i razred `Kompozit`. To je razred koji koristi kolekciju u kojoj čuva reference na druge komponente; klijentima nudi metode kojima je moguće dodavanje, uklanjanje te dohvat komponenata iz kolekcije.

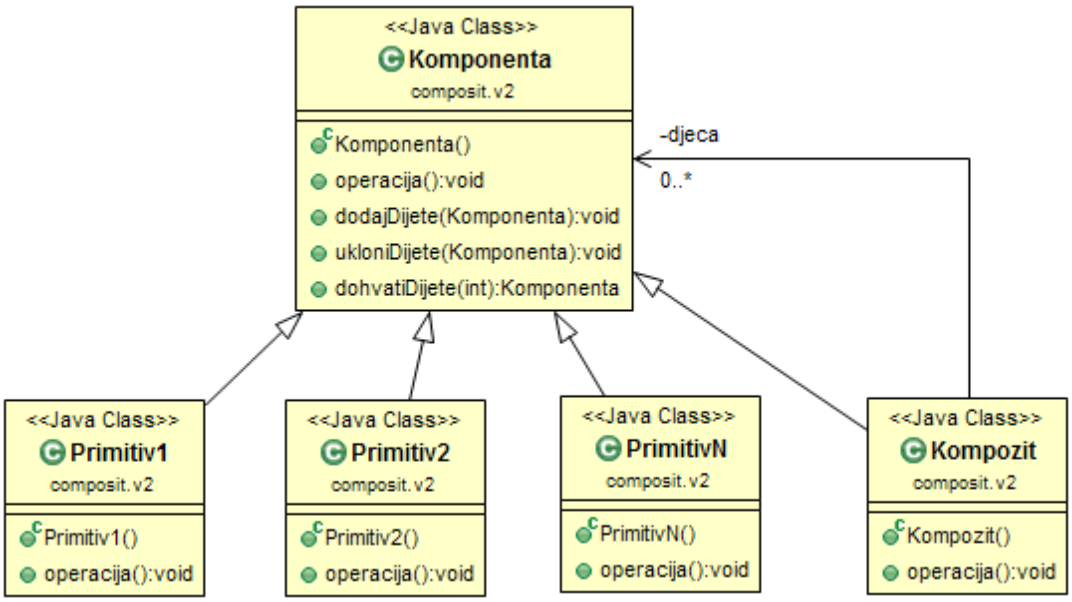
Slika 18.11. Strukturni dijagram Kompozita (sigurna verzija)



Prikazani strukturni dijagram definira dvije vrste komponenata: primitive (koji su "konačne" implementacije određenih funkcionalnosti) te kompozite koji nude mogućnost upravljanja djecom. Da bi klijent mogao koristiti ove funkcionalnosti, on mora biti svjestan postojanja obaju vrsta. S jedne strane, ovakva organizacija koda vodi ka sigurnoj implementaciji: djecu imaju samo kompoziti i nije moguće pogreškom djecu pokušati dodavati primitivnim komponentama. Mana pristupa jest postojanje dvaju (konceptualnih) sučelja: jedno primitivnih komponenata te drugo (bogatije) kompozita.

Pristup kod kojeg klijent sve komponente oblikovnog obrasca može tretirati na jednak način (transparentno) prikazan je na slici 18.12.

Slika 18.12. Strukturni dijagram Kompozita (transparentna verzija)



Kod ovog pristupa osnovna komponenta prema klijentima definira sučelje koje je unija sučelja primitivnih komponenata i kompozita: deklariraju se sve metode kojima klijenti obavljaju po-

trebne operacije kao i sve metode kojima se upravlja djecom. Prednost ovakvog pristupa jest jedinstveno sučelje prema klijentu: klijent ne mora biti svjestan postojanja različitih vrsta komponenata. Nedostatak pristupa leži u tome da sada primitivi u svojem sučelju imaju metode za upravljanje djecom pa ih klijenti mogu pozivati i nad primitivima; u tom slučaju primitivi ove metode nekako ih moraju implementirati (primjerice, tako da pri pozivu bacaju iznimku i time signaliziraju klijentu da je napravio pogrešku).

Pri uporabi ovog oblikovnog obrasca potrebno je razmotriti još nekoliko pitanja.

- *Imaju li komponente referencu na roditeljski element?* U strukturi koju smo prikazali na dijagramima 18.11 i 18.12, kompoziti imaju referencu na djecu i time je propagacija operacija moguća isključivo od korijena prema dnu stabla. Ponekad je, međutim, potrebno osigurati da se izvođenje može usmjeriti od dna stabla prema vrhu: u tom slučaju prikladno je u komponentu dodati referencu prema roditeljskoj komponenti.

- *Gdje su deklarirane metode za upravljanje djecom?* O ovome smo već diskutirali.

  - Ako ih deklariramo isključivo u kompozitu, dobivamo na sigurnosti implementacije - klijent ne može pogrešno koristiti te implementacije u primitivima pa o tome ne treba niti razmišljati. S druge strane, klijent tada mora biti svjestan postojanja dviju vrsta sučelja a to je često nepoželjno.

  - Ako ih deklariramo u komponenti, dobivamo na transparentnosti: klijent vidi jedno jedinstveno sučelje koje može koristiti. Međutim, primitivi nekako moraju definirati ove metode, a smislene implementacije nema (mogu primjerice bacati iznimku).

- *Ako su metode za upravljanje djecom deklarirane u komponenti, gdje je deklariran podatkovni član koji pamti kolekciju djece?*.

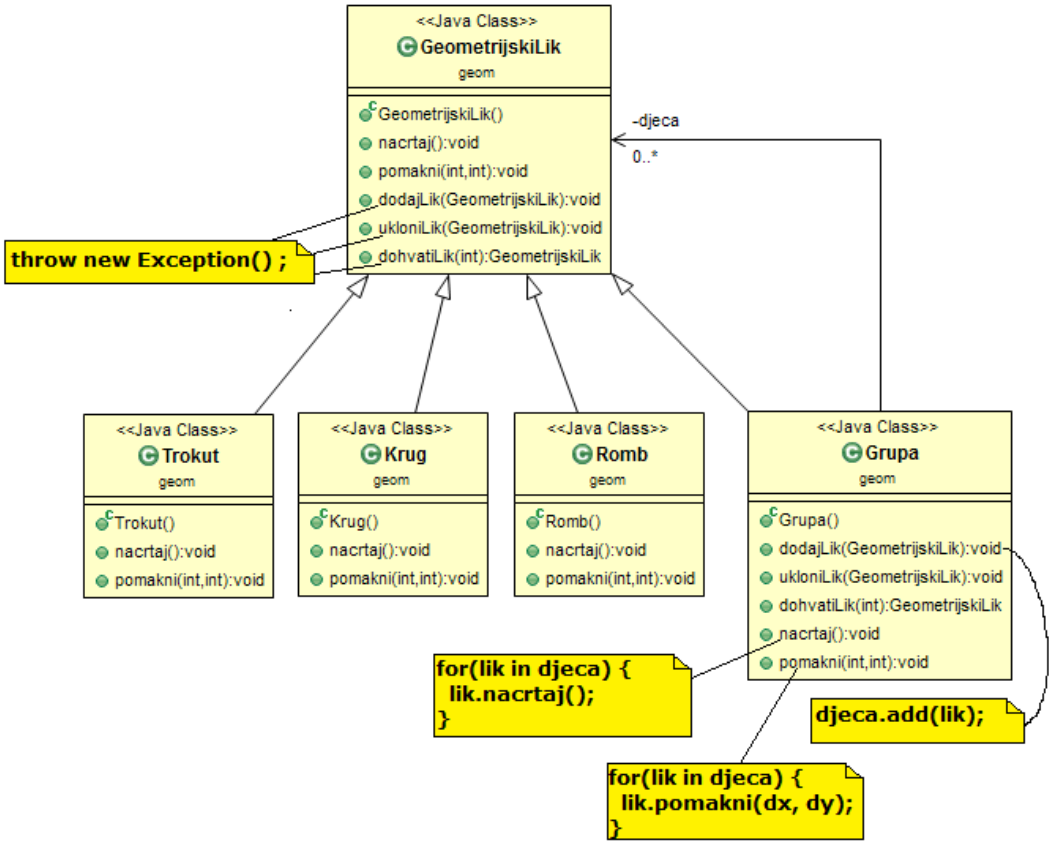
  - Najjednostavnije je i sam podatkovni član definirati u komponenti. Tada je u komponenti moguće ponuditi i potpuno funkcionalne metode za upravljenje djecom pa kompoziti ne moraju raditi ništa (jer od komponente nasljeđuju korektne implementacije). Zadaća primitiva je tada nadjačati ove metode tako da ne rade ništa jer primitivi ne mogu imati djecu. Također, negativno svojstvo pristupa je činjenica da svaki primitiv troši memoriju za pohranu reference na kolekciju - ako primitiva ima puno, ovo može biti nezanemariv trošak.

  - S druge strane, podatkovni član za pamćenje kolekcije može biti deklariran u kompozitu. Tada metode za upravljanje djecom u komponenti moraju biti prazne (ili bacati iznimku) a kompozit ih mora smisleno implementirati. Prednost pristupa je da sada primitivi troše manje memorije i mogu koristiti osnovnu izvedbu metoda za upravljanje djecom iz komponente (bacanje iznimke). Negativna strana pristupa je činjenica da kompoziti moraju definirati smislene implementacije metoda za upravljanje djecom.

- *Je li poredak djece bitan?* Ovisno o konkretnim potrebama kolekcija u kojoj se čuvaju reference na djecu može biti takva da garantira poredak djece (indeksirana/pobrojava kolekcija) ili neka koja nema takvih garancija.

Pogledajmo na kraju i jedan konkretan primjer: razmotrimo program za vektorsko crtanje koji podržava različite vrste geometrijskih likova. Nad likovima moraju biti definirane dvije operacije: crtanje na ekranu te pomicanje (translacija) za zadani (*dx*, *dy*). Pretpostavimo također da taj isti program mora podržati i mogućnost grupiranja objekata te transparentnog rada s grupama objekata: klijent na isti način treba moći crtati i pomicati osnovne geometrijske likove te grupe. Prikladan (dodatno obilježen) dijagram razreda za ovaj slučaj prikazan je na slici 18.13.

Slika 18.13. Oblikovni obrazac Kompozit primijenjen na organizaciju geometrijskih likova



Oblikovni obrazac *Posjetitelj*

Osnovna zadaća oblikovnog obrasca *Posjetitelj* je omogućavanje transparentnog dodavanja novih operacija nad postojećim porodicama objekata, bez potrebe za mijenjanjem tih porodica objekata.

Razmotrimo primjer web-aplikacije koju koristi neka internetska trgovina. Vrste artikala s kojima aplikacija zna raditi modelirane su porodicom razreda `Artikl` iz kojeg su izvedeni `Knjiga`, `CD` i `Pisač`. Razred `Artikl` definira postojanje metode `double cijena();` jer svaki artikl ima svoju cijenu. Pretpostavite za potrebe primjera da je ova struktura razreda stabilna: mala je vjerojatnost da ćete trebati uskoro mijenjati porodicu artikala.

Zahvaljujući ovakvoj strukturi ukupnu cijenu svih proizvoda u košarici korisnika možemo izračunati pišući sljedeći kod u klijentima.

```
List<Artikl> artikli = košarica.dohvatiArtikle();
double ukupnaCijena = 0;
for(Artikl a : artikli) {
 ukupnaCijena += a.cijena();
}
```

Kako međutim riješiti sljedeći problem: pretpostavite da web-trgovina za kupnju svih CD-a nudi popust od 20% na stvarnu cijenu CD-a, a za kupnju svake knjige nudi popust od 10% na stvarnu cijenu knjige? Kako ćemo sada izračunati ukupnu cijenu koju kupac mora platiti? S obzirom da naš klijentski kod od košarice dobiva isključivo listu artikala (preko zajedničkog apstraktnog sučelja), naš će klijentski kod sada morati na neki način otkrivati stvarne tipove artikala kako bi znao treba li na njih primijeniti popust. Moguće rješenje prikazano je u nastavku.

```
List<Artikl> artikli = košarica.dohvatiArtikle();
double ukupnaCijena = 0;
for(Artikl a : artikli) {
 if(a instanceof Knjiga) {
  ukupnaCijena += a.cijena()*0.9;
 } else if(a instanceof CD) {
  ukupnaCijena += a.cijena()*0.8;
 } else if(a instanceof Pisač) {
  ukupnaCijena += a.cijena();
 }
}
```

Ovakav kod, međutim, ne želimo pisati: uporaba `instanceof`-pitalica (ili sličnih `switch/case`-konstrukata) vodi ka krhkom kodu. Jedna mogućnost bila da pripremimo mapu popusta (ključevi bi bili vrste artikala, vrijednosti popusti), zatim da u razred `Artikl` dodamo još i metodu `double cijenaSPopustom(Map<String,Double> popusti);` i zatim u svakom konkretnom artiklu implementiramo tu metodu tako da dohvati osnovnu cijenu, u mapi pronađe popust koji odgovara tom tipu i vrati izračunatu vrijednost. Nešto poput sljedećeg koda za razred `Knjiga`.

```
double cijenaSPopustom(Map<String,Double> popusti) {
  Double zapisaniPopust = popusti.get("Knjiga");
  double popust = zapisaniPopust == null ? 1.0 : zapisaniPopust;
  return this.cijena() * popust;
}
```

Sličnu implementaciju trebali bismo dodati u svaki konkretan artikl.

Pretpostavimo sada da se u nekom kasnijem trenutku pojavi sljedeća potreba: htjeli bismo pripremiti katalog svih proizvoda s kojima raspolažemo, i to tako da se uz svaku vrstu proizvoda pojavi mala sličica koja opisuje tu vrstu proizvoda (dakle, uz svaki pisač htjeli bismo prikazati istu malu sličicu koja prikazuje neki generički pisač; slično i za knjige i CD-ove). Iako se problem čini drugačiji od prethodno razmatranog, konceptualno je isti: želimo riješiti zadatak koji na različite načine tretira različite vrste objekata, a dobivamo samo referencu na apstraktni tip. Rješenje bi opet moglo biti isto: kako ne želimo pisati `instanceof`-pitalicu, promijenit ćemo sučelje razreda `Artikl`, dodati odgovarajuću apstraktnu metodu i potom u svim konkretnim artiklima ponuditi prikladnu implementaciju.

Nažalost, malo po malo, ovim pristupom za svaku novu operaciju moramo mijenjati postojeću porodicu razreda: u vršni razred moramo dodati apstraktnu metodu te u izvedene razrede staviti odgovarajuću implementaciju. Ukoliko naša struktura razreda ima više različitih klijenata, mijenjenjem metoda definiranih u strukturi razreda svaki klijent postaje svjestan svih operacija koje struktura podržava, a to je loše.

Moguće rješenje opisanog problema leži u razdvajanju konkretnih operacija od domenskog modela podataka (u našem slučaju, stabla izvedenog iz razreda `Artikl`). Prvi korak u tom smjeru jest definiranje apstraktnog sučelja koje opisuje bilo koju operaciju: to sučelje ima po jednu metodu za svaki konkretni razred domenskog modela (u našem slučaju: za knjigu, za CD te za pisač). U oblikovnom obrascu *Posjetitlj* to se sučelje upravo naziva apstraktni posjetitelj (tj. engl. *Visitor*). Konkretne operacije potom izvodimo kao razrede izvedene iz ovog sučelja. Kako u tim razredima za svaki konkretni tip postoji po jedna metoda koja će biti pozvana upravo u trenutku kada treba obraditi element domene tog tipa, za svaki tip možemo napisati drugačiji kod koji na drugačiji način obrađuje trenutni element.



Bilješka

Oblikovni obrazac *Posjetitelj* posebno je pogodan u situacijama u kojima imamo stabilnu strukturu razreda (koja je rijetko mijenja) te potrebu za čestim dodavanjem novih operacija nad tom strukturom.

Sučelje posjetitelja i implementacija konkretnog posjetitelja koji računa cijenu s popustom ilustrirana je u nastavku.

```
public interface PosjetiteljArtikala {
 void obradiKnjigu(Knjiga k);
 void obradiCD(CD c);
 void obradiPisač(Pisač p);
}

public class CijeneSPopustom implements PosjetiteljArtikala {

 private Map<String,Double> popusti;
 private ukupnaCijena = 0;

 public CijeneSPopustom(Map<String,Double> popusti) {
  this.popusti = popusti;
 }

 void obradiKnjigu(Knjiga k) {
   ukupnaCijena += k.cijena() * nadiPopustZa("Knjiga");
 }

 void obradiCD(CD c) {
   ukupnaCijena += k.cijena() * nadiPopustZa("CD");
 }

 void obradiPisač(Pisač p) {
   ukupnaCijena += k.cijena() * nadiPopustZa("Pisač");
 }

 public double dohvatiUkupnuCijenu() {
  return ukupnaCijena;
 }

 private double nadiPopustZa(String tip) {
  Double zapisaniPopust = popusti.get(tip);
  return zapisaniPopust == null ? 1.0 : zapisaniPopust;
 }

}
```

Jedini preostali problem je kako osigurati da se za svaki artikl pozove prava metoda koja radi obradu baš tog tipa artikla. Klijent koji ima referencu na listu artikala i konkretnu obradu problem bi mogao riješiti uporabom `instanceof`-pitalice. Bolje je rješenje modificirati sučelje artikala (odnosno vršni razred naše porodice) tako da podrži izvođenje apstraktne operacije (pa time bilo koje operacije bez potrebe za daljnjim promjenama u kodu). Ideja je da se u porodicu razreda doda metoda zadužena za prihvat posjetitelja. U konkretnim artiklima ta će metoda nad primljenim posjetiteljem pozvati metodu koja radi obradu upravo tog konkretnog tipa. A sam klijent će, zahvaljujući dinamičkom polimorfizmu, nad apstraktnim sučeljem artikla pozvati upravo pravu metodu za prihvat posjetitelja, koja će potom u posjetitelju pozvati upravo pravu metodu za obradu tog konkretnog artikla. Ova tehnika u engleskoj terminologiji poznata je kao *double-dispatch*.

Potrebne prilagodbe razreda `Artikl` i ostatka porodice prikazane su u nastavku.

```
abstract class Artikl {
 // ostatak koda ...
 public abstract void prihvati(PosjetiteljArtikala p);
}
```

```
public class Knjiga {
 // ostatak koda ...
 public void prihvati(PosjetiteljArtikala p) {
  p.obradiKnjigu(this);
 }
}

public class CD {
 // ostatak koda ...
 public void prihvati(PosjetiteljArtikala p) {
  p.obradiCD(this);
 }
}

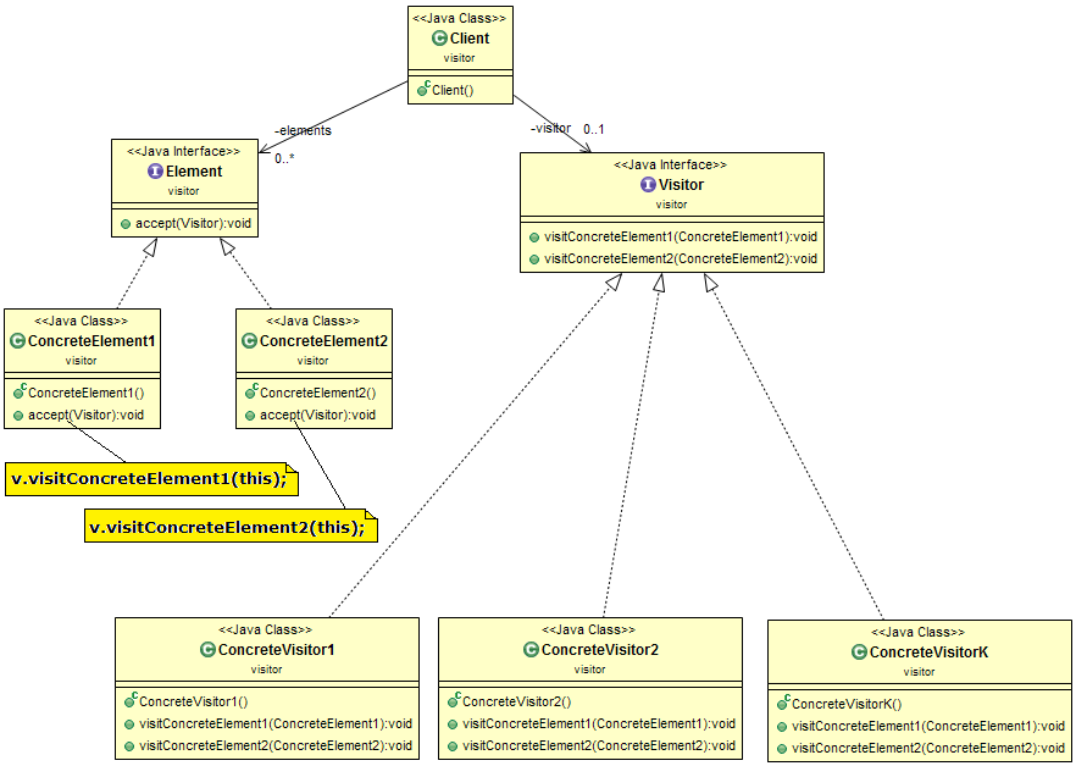
public class Pisač {
 // ostatak koda ...
 public void prihvati(PosjetiteljArtikala p) {
  p.obradiPisač(this);
 }
}
```

Uz ovakve modifikacije i sam klijentski kod postaje trivijalan.

```
Map<String,Double> popusti = pripremiMapuPopusta();
PosjetiteljArtikala p = new CijeneSPopustom(popusti);
for(Artikl a : artikli) {
 a.prihvati(p);
}
double ukupnaCijena = p.dohvatiUkupnuCijenu();
```

Pogledajmo sada kako izgleda strukturni dijagram ovog oblikovnog obrasca: slika 18.14.

Slika 18.14. Oblikovni obrazac Posjetitelj: strukturni dijagram



Sudionici ovog oblikovnog obrasca su sljedeći.

- *Posjetitelj (sučelje* `Visitor`*).* Predstavlja apstraktnu operaciju i definira metode za obradu svake konkretne vrste domenskih objekata.
- *Konkretni posjetitelji (razredi* `ConcreteVisitor1`, `ConcreteVisitor2`, ..., `ConcreteVisitorK`*).* Predstavljaju implementacije različitih vrsta operacija koje se provode nad objektima domene a čija obrada ovisi o konkretnom tipu objekta domene. Dodavanje novih operacija svodi se na implementiranje novih razreda ovog tipa.
- *Element (sučelje* `Element`*).* Predstavlja vršni razred/sučelje iz kojeg su izvedeni svi razredi domenskog modela. Definira apstraktnu metodu `accept` zaduženu za prihvat apstraktnih posjetitelja.
- *Konkretni elementi (razredi* `ConcreteElement1`, `ConcreteElement2`*).* Predstavljaju konkretne domenske razrede. Moraju implementirati/nadjačati metodu za prihvat posjetitelja tako da nad posjetiteljem pozovu metodu za obradu upravo vlastitog tipa elemenata.
- *Klijent (razred* `Client`*).* Barata s kolekcijom domenskih objekata preko apstraktnog sučelja. Stvara konkretnu operaciju (konkretnog posjetitelja) i traži od svakog elementa da prihvati operaciju. Koristi konačni rezultat obrade.

Spomenimo još i da se oblikovni obrazac *Posjetitelj* odlično nadopunjuje s oblikovnim obrascem *Kompozit*. U toj kombinaciji, kompozitni razredi mogu u metodi `accept` protrčati kroz popis svoje djece i svako dijete zatražiti da prihvati predanog posjetitelja. Na taj način sučelje posjetitelja ne mora znati za kompozitne objekte a sami kompozitni objekti pobrinut će se da traženu operaciju transparentno provedu nad sadržanom djecom. U nekim implementacijama, međutim, javlja se potreba da se operacija ne propagira svoj djeci kompozita već se želi implementirati složenija upravljačka logika. U takvom slučaju sučelje posjetitelja definirat će i metodu koja obrađuje kompozita i u njoj će moći implementirati složeniju upravljačku logiku za obilazak njegove djece; sam kompozit u tom će slučaju metodu za prihvat posjetitelja implementirati isto kao i svi ostali razredi: pozvat će nad posjetiteljem odgovarajuću metodu koja je zadužena za obradu tog kompozita (i neće samostalno prosljeđivati posjetitelja svojoj djeci). Kompromis koji se ovdje radi je kompromis između dupliciranja koda i upravljivosti: želimo li maksimalnu upravljivost, svaki će posjetitelj morati pisati svoj kod za obilazak djece kompozita što će često dovoditi do dupliciranja koda; međutim, oni posjetitelji koji trebaju složenija pravila obilaska, moći će ih ostvariti. S druge pak strane, stavljanjem tog koda u sam kompozit definirali smo zatvorenu implementaciju koja radi obilazak uvijek na isti način, ali prednost je u tome što više niti jedan posjetitelj neće morati sam pisati taj kod čime smo maksimalno eliminirali potrebu za dupliciranjem koda.

Poglavlje 19. Višedretvene aplikacije

U svim prethodnim poglavljima knjige pisali smo program uz pretpostavku da se sve u našem programu izvodi slijedno. Središnji procesor računala (engl. *Central Processing Unit, CPU*) izvodi kod koji smo napisali upravo na način na koji smo ga napisali. Naredbe se izvode slijedno, jedna za drugom.

U stvarnosti, naš se program ne mora izvoditi baš tako. Već i sami središnji procesori u današnje doba mogu mijenjati redoslijed kojim izvode instrukcije napisanog programa (takozvani *out-of-order execution* [1]), ako time mogu posao obaviti brže a garantirati da će konačni efekt izvođenja biti jednak onom koji bismo imali da su instrukcije izvedene redoslijedom kojim su navedene u memoriji. Privid da se program izvodi onako kako ga pišemo, danas je upravo to: privid. Otkad postoje računala i operacijski sustavi, postoji i razmišljanje o načinima na koje se središnji procesori mogu iskoristiti bolje: zadnje što želimo u računalu je središnji procesor koji ne radi ništa, a korisnik čeka da se obavi zadatak koji je zadao.

U davno doba operacijskog sustava Microsoft DOS "istovremeno" izvođenje više programa nije bilo moguće - operacijski sustav to nije podržavao. Postojalo je, međutim, jedno međurješenje. Korisnik je mogao napisati program koji bi se učitao u memoriju, zakvačio na određene prekidne rutine i kontrolu vratio osnovnoj ljusci operacijskog sustava čime je za korisnika program prividno završio. U stvarnosti, program bi ostao prisutan (rezidentan) u memoriji i na svako aktiviranje određene prekidne rutine pozvala bi se neka njegova metoda čime je program u pozadini mogao obavljati određene zadatke. Dakako, u to vrijeme klasična osobna računala o kojima ovdje govorimo imala su samo jedan središnji procesor koji je tada naizmjenice izvodio ili instrukcije prekidne rutine ili instrukcije programa koji je korisnik pokrenuo (ali nikada oboje istovremeno).

U modernim operacijskim sustavima i programskim jezicima ovakve su ideje bitno unaprijeđene i razrađene kako bi se programeru omogućilo pisanje programa koji sami u bilo kojem trenutku mogu zadavati nove nizove instrukcija koje procesor treba izvoditi i koji mogu napredovati nezavisno jedan od drugoga. Primjerice, razmotrite sljedeći isječak koda.

```
int[] polje1 = new int[1000];
int[] polje2 = new int[1000];

napuniPrimBrojevima(polje1);
napuniSlucajnimBrojevimaPaSortiraj(polje2);
```

Prikazani kod deklarira dva polja i potom poziva dvije metode. Prva metoda računa prvih 1000 prostih brojeva i njima popunjava `polje1`. Druga metoda na neki način generira 1000 slučajnih brojeva, njima popunjava `polje2` i potom to polje sortira. Iako su pozivi ovih metoda napisani slijedno, s obzirom da prva metoda ni na koji način ne utječe na rezultat rada druge metode (niti obrnuto), ove dvije metode mogli bismo poželjeti izvoditi istovemeno (što danas u situaciji kada računala imaju više središnjih procesora doista možemo i ostvariti).

Drugi primjer koji ćemo razmotriti prikazan je pseudokodom u nastavku.

```
void skiniDatoteku(String imePosluzitelja, String imeDatoteke,
                   Datoteka datoteka) {
  byte[] spremnik = new byte[1024];
  Socket posluzitelj = uspostaviVezu(imePosluzitelja);
  datoteka.otvoriZaPisanje();
  posluzitelj.zatraziDatoteku(imeDatoteke);
  while(true) {
    int procitano = posluzitelj.procitaj(spremnik);
    if(procitano<1) break; // gotovi smo
    datoteka.zapisi(spremnik, procitano);
```

[1] Više informacija dostupno je na adresi http://en.wikipedia.org/wiki/Out-of-order_execution

```
  }
  datoteka.zatvori();
  posluzitelj.prekiniVezu();
}

skiniDatoteku(
  "java.zemris.fer.hr",
  "/javaCourses/presentations/java_tecaj_01_prezentacija.pdf",
  new Datoteka("C:/p1.pdf")
);
skiniDatoteku(
  "java.zemris.fer.hr",
  "/javaCourses/presentations/java_tecaj_02_prezentacija.pdf",
  new Datoteka("C:/p2.pdf")
);
```

Program se sastoji od metode `skiniDatoteku` koja se spaja na poslužitelj na Internetu, šalje zahtjev za datotekom, čita malo po malo oktete (sadržaj datoteke) koje šalje poslužitelj i zapisuje ih lokalno na disk korisnika. Prilikom izvođenja ovog koda za očekivati je da će središnji procesor poslužitelja imati mnoštvo trenutaka kada ne radi ništa jer mora čekati. Primjerice, poziv `posluzitelj.procitaj(spremnik);` je takozvani blokirajući poziv. Zadaća te metode je predani spremnik napuniti podatcima koji su pristigli od poslužitelja. Kako se ovdje radi o komunikaciji putem mreže, sasvim je moguće da u trenutku poziva te metode ništa od poslužitelja još nije doputovalo mrežom do našeg računala: metoda ne može obaviti svoj posao i morat će na određen način zablokirati sve dok podatci ne pristignu. Jednom kada se to dogodi, program primljene podatke zapisuje na disk. Međutim, disk je mehanička naprava, i od trenutka kada iniciramo zapisivanje pa do trenutka kada se to doista dogodi proći će određeno vrijeme. Kako metoda `datoteka.zapisi(spremnik, procitano)` mora baciti iznimku ako pisanje nije uspjelo, metoda će opet morati zablokirati daljnje izvođenje sve dok ne dočeka kraj zapisivanja i primitak statusa na temelju kojeg će od kontrolera diska zaključiti je li došlo do pogreške.

U nastavku prethodnog pseudokoda potom slijede dva poziva ove metode kojima se skidaju dvije datoteke. Lako je zaključiti da bi ta dva poziva bilo korisno pokrenuti tako da se izvode istovremeno. Čak i na računalu na kojem to nije moguće jer postoji samo jedan procesor, ako bi nam operacijski sustav ponudio mehanizam kojim bi, u trenutku kada izvođenje jednog niza instrukcija dođe do mjesta gdje bi taj niz zaglavio, operacijski sustav prešao na izvođenje onog drugog niza instrukcija (umjesto da besposleno čeka), ukupno izvođenje posla (skidanje dviju datoteka) prije bi završilo.

Mehanizam koji smo maloprije spomenuli postoji; dapače, postoji nekoliko načina kako možemo pisati programe koji ne čekaju besposleno na blokirajućim pozivima. Za programera, najjednostavniji je koncept dretve i višedretvene aplikacije[2]. Stoga ćemo se u ovom poglavlju pozabaviti upravo tom tematikom.

Prije no što krenemo dalje, osvrnut ćemo se na još tri temeljna pitanja: što je proces, od čega se sastoji te kako se ostvaruje višezadaćnost.

Proces je apstrakcija operacijskog sustava koja predstavlja program u izvođenju. Proces se sastoji od strojnog koda programa, od podataka nad kojima program radi, od jedne ili više dretvi koje izvode nizove instrukcija programa, od opisnika datoteka preko kojih program komunicira s okolinom (primjerice, standardni ulaz, standardni izlaz te standardni izlaz za pogreške) te još nekih drugih podataka.

Svaki operacijski sustav nudi neki mehanizam kojim se stvaraju novi procesi. Ovisno od operacijskog sustava do operacijskog sustava, metode se zovu različito, ali temeljna ideja je ista. Korisniku treba nekako omogućiti da stvori novi proces, da definira koje instrukcije i podatke treba učitati u memoriju procesa te da pokrene izvođenje tako učitanog programa. Da bi

[2]Drugo rješenje koje odmah pada na pamet je uporaba neblokirajućih poziva odnosno asinkron API (http://en.wikipedia.org/wiki/Asynchronous_I/O)

operacijski sustav mogao učitati program u memoriju, program u datoteku mora biti zapisan u očekivanom formatu. Svaki operacijski sustav definira taj format. Primjerice, na operacijskom sustavu Microsoft Windows točno je definirano kakve strukture mora biti EXE-datoteka. Operacijski sustav Linux definira kakve strukture mora biti datoteka koja sadrži program namijenjen pokretanju pod tim operacijskim sustavom (ELF-format[3]).

Današnji moderni operacijski sustavi procesima nude mnoštvo naprednih mogućnosti. Primjerice, *memorijski prostori svih procesa međusobno su izolirani*. To znači da svaki proces ima privid kao da raspolaže cjelokupnom memorijom računala. Izvši li se u dva različita procesa kod koji na "istu" memoriju lokaciju zapiše različite vrijednosti, neće doći do nikakvih problema: operacijski sustav će se pobrinuti da ta pisanja završe u fizičkoj memoriji na različitim adresama i točno će znati za koji je proces odabrao koju fizičku lokaciju.

Također, moderni operacijski sustavi podržavaju *straničenje*. Straničenje je mehanizam kojim se virtualna memorija (koju vidi proces) dijeli u manje stranice koje mogu biti ili smještene u fizičku memoriju računala ili privremeno sklonjene u datoteku na disku (ako u fizičkoj memoriji nema dovoljno mjesta). Ako je fizička memorija popunjena a proces pokuša čitati podatak sa stranice koja je trenutno na disku, operacijski će sustav proces privremeno suspendirati, neku drugu stranicu koja se u tom trenutku ne koristi iz fizičke memorije prebaciti u datoteku a na tako upražnjeno mjesto u fizičku memoriju učitati traženu stranicu s diska, i potom nastaviti izvođenje suspendiranog procesa koji će sada dobiti traženi podatak direktno iz memorije. Uočite: proces nije svjestan ovog mehanizma.

Kada govorimo o *višezadaćnosti* (sposobnosti operacijskog sustava za izvodi više nizova instrukcija "istovremeno"), treba pojasniti dva temeljna načina na koja se ona ostvaruje. Imamo li u računalu samo jedan procesor, jasno je da taj procesor ne može u isto vrijeme izvoditi više nizova instrukcija. U takvom scenariju, dvije su izvedbe podrške za višezadaćnost: *kooperativna izvedba* te *preemptivna izvedba*.

Kod kooperativne izvedbe[4], korisnik mora pisati program tako da periodički poziva određenu funkciju operacijskog sustava. Pozivom te funkcije izvođenje ulazi u jezgru operacijskog sustava gdje operacijski sustav može pogledati postoji li još koji niz instrukcija koji je pribilježen za izvođenje, pa ako postoji, može napraviti određene promjene kojima će osigurati da procesor umjesto izvođenja niza instrukcija koji je pozvao tu metodu krene s izvođenjem nekog drugog od pribilježenih nizova instrukcija (koji će kad-tad i sam pozvati ovu funkciju operacijskog sustava i omogućiti novu promjenu). Mehanizam se zove kooperativni jer da bi radio korektno, programi moraju biti disciplinirani i periodički pozivati funkciju operacijskog sustava. Funkcija o kojoj ovdje govorimo najčešće se naziva `yield()`. Ovaj oblik višezadaćnosti korišten je kod operacijskog sustava Microsoft Windows 3.11. Pisanjem programa koji nije periodički pozivao spomenuti funkciju operacijskog sustava korisnik je mogao lagano napisati kod koji bi zablokirao čitavo računalo. Dovoljno je bilo pokrenuti `while(1) {}`.

Preemptivna izvedba višezadaćnosti[5] rješava problem zlonamjernih aplikacija na način da ne zahtijeva nikakvu suradnju od samih programa. Umjesto toga, operacijski se sustav oslanja na sklopovlje računala koje podešava tako da periodički izaziva prekid (engl. *interrupt*) koji potom pokreće obradu prekidne rutine samog operacijskog sustava. Ta prekidna rutina obavlja istu funkciju kao i prethodno spomenuta: u njoj operacijski sustav može aktivirati sljedeći niz instrukcija za izvođenje na procesoru. Na ovaj se način svim pribilježenim nizovima instrukcija ciklički "iznajmljuje" središnji procesor na određeno vrijeme (engl. *timeslot*). Dakako, moderniji operacijski sustavi podržavaju i definiranje prioriteta pojedinih pribilježenih nizova instrukcija pa umjesto da ravnomjerno raspoređuju sve na procesor, češće mogu procesor davati nizovima većeg prioriteta.

Ako računalo raspolaže s više središnjih procesora (ili jezgara), operacijski sustav ima mogućnost ponuditi i istinsko istovremeno izvođenje slijedova instrukcija: istovremeno je moguće izvoditi onoliko slijedova koliko imamo jezgara na raspolaganju.

[3] Executable and Linkable Format, http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[4] http://en.wikipedia.org/wiki/Computer_multitasking#COOP
[5] http://en.wikipedia.org/wiki/Computer_multitasking#Preemptive_multitasking

Njezino visočanstvo: Dretva

U uvodnom dijelu ovog poglavlja osvrnuli smo se na pojam procesa i višezadaćnosti. Pojasnimo sada što je dretva. *Dretva* je niz instrukcija u izvođenju. Dretve u prethodnom tekstu predstavljaju upravo one "pribilježene" nizove instrukcija kojima operacijski sustav iznajmljuje sklopovske resurse računala: jezgru koja će ih izvoditi. Jedan proces sastoji je od minimalno jedne dretve: ona čiji je zadatak izvođenje instrukcija koje započinju od metode `main`: početne metode svakog programa. Proces, međutim, može sadržavati više dretvi. Naime, krenuvši od metode `main` korisnik je mogao zadati naredbe koje stvaraju nove dretve i povezuju ih s nizovima instrukcija koje one trebaju izvoditi. U programskom jeziku C stvaranje novih dretvi svodi se na poziv metode kojoj se kao argument predaje pokazivač na funkciju koju ta nova dretva treba izvesti. U programskom jeziku Java ubrzo ćemo pokazati što nam stoji na raspolaganju.

Na razini operacijskog sustava, svaka se dretva sastoji od identifikatora dretve, prioriteta dretve, konteksta dretve te stoga dretve. Kontekst dretve je dio memorijskog prostora u koji se kopira stanje registara procesora u trenutku kada operacijski sustav dretvu skida s procesora. U trenutku kada operacijski sustav dretvu ponovno vraća na procesor kako bi se nastavilo izvođenje njenih instrukcija, najprije se iz pridruženog konteksta obnavljaju zapamćeni sadržaji registara i tek se tada pokreće daljnje izvođenje dretve. Zahvaljujući uporabi konteksta, dretva neće biti svjesna prekida: prva instrukcija koja se pokreće nakon prekinutog izvođenja u registrima vidi upravo ono stanje koje je bilo u trenutku kada je dretva bila prekinuta. Ovaj mehanizam bit će nam od posebnog značaja za razumijevanje primjera koji slijede a koji ilustriraju probleme koji se mogu javiti pri višedretvenom izvođenju programa koji sadrži podatkovne strukture nad kojima istovremeno radi više dretvi.

Uz kontekst, svaka dretva mora imati i svoj vlastiti stog. Stog je dio memorije preko kojeg se prenose parametri pri pozivu funkcija, pohranjuju povratne adrese (koristi se pri povratku iz potprograma) te mjesto na kojem metode stvaraju lokalne varijable. Upravo iz tog razloga stog ne može biti dijeljen između više dretvi. I upravo iz tog razloga, stvaranje novih dretvi nije baš "jeftina" operacija. Nakon što naučimo osnovno o dretvama, pogledat ćemo kako možemo postići da višestruko iskorištavamo jednom stvorene dretve (čime ćemo smanjiti broj zahtjeva za stvaranje novih dretvi).

Spomenimo još da na razini operacijskog sustava ne postoji izolacija memorijskog prostora između više dretvi istog procesa (u općem smislu), kao što to postoji između više procesa. Dretve pripadaju procesima i dijele memorijski prostor procesa. Promjenu koju jedna dretva napravi u tom memorijskom prostoru vidljiva je svim ostalim dretvama istog procesa.

Model dretve u Javi

Svakoj dretvi programa napisanog u Javi pridružen je primjerak razreda `java.lang.Thread`. U nastavku teksta primjerke tog razreda zvat ćemo još i *opisnikom dretve*. Java dretve grupira u grupe. Grupa dretvi modelirana je razredom `java.lang.ThreadGroup`. Grupe dretvi grade stablastu strukturu: članovi grupe mogu biti dretve te druge grupe. Svaka grupa ima roditelja osim vršne grupe. Svakoj se grupi dretvi može definirati ime te maksimalni prioritet koji članovi te grupe mogu imati. Grupa nudi metode preko kojih je moguće doći do roditeljske grupe te do članova grupe (do opisnika dretvi za dretve koje pripadaju toj grupi te do opisnika grupa za grupe koje su podgrupe).

Prilikom pokretanja Javinog virtualnog stroja (kada "pokrećemo" naš program) operacijski sustav stvara novi proces za javin virtualni stroj (**java.exe** na Windowsima), koji potom stvara inicijalnu grupu dretvi te u nju dodaje nekoliko dretvi. Virtualni stroj potom u memoriju učitava razred koji smo naveli kao argument naredbe **java.exe**, provjerava postoji li u njemu javna statička metoda `main` s propisanim brojem i vrstom argumenata (jedno polje stringova) i ako je sve u redu, u inicijalnu grupu dretvi stvara i dodaje novu podgrupu u koju dodaje dretvu koja poziva metodu `main`. Prioritet te dretve postavlja se na srednji (tj. "normalan") i dretva se pokreće, čime se pokreće i naš program.

Dretve koje su stvorene u inicijalnoj grupi su dretve koje opslužuju sam virtualni stroj i tipično ne interferiraju s našim programom. Iz tog razloga smo se sve do sada mogli praviti da radimo u jednodretvenom okruženju (odnosno da ne znamo ništa o dretvama). Primjer dretve koja će biti u inicijalnoj grupi je dretva koja izvodi skupljač smeća, dretva koja izvodi kod koji obrađuje signal koji se procesu šalje na pritisak tipke CTRL+C, i slično.

Za potrebe dobivanja reference na opisnik dretve koja izvodi trenutni kod na raspolaganju je statička metoda `currentThread` razreda `Thread`. Opisnik dretve sadrži niz informacija o samoj dretvi, uključivo i njezino ime, je li dretva još uvijek živa (ili je završila s izvođenjem zadanog koda), kojeg je prioriteta, kojoj grupi pripada i slično. S ovim metodama upoznat ćemo se uskoro.

Jednostavan program koji prikazuje sve grupe dretvi i dretve u njima, koje virtualni stroj stvara prilikom pokretanja programa, prikazan je u nastavku (primjer 19.1). Metoda `.list()` koju pozivamo u primjeru nad vršnom grupom ispisuje čitavo podstablo grupa i dretvi.

Primjer 19.1. Ispis svih grupa dretvi i dretvi po pokretanju programa

```
 1 package book.java.threads;
 2
 3 public class ListAll {
 4
 5  public static void main(String[] args) {
 6
 7   Thread current = Thread.currentThread();
 8   ThreadGroup group = current.getThreadGroup();
 9
10   while(true) {
11    ThreadGroup parent = group.getParent();
12    if(parent==null) break;
13    group = parent;
14   }
15
16   group.list();
17
18   System.out.println();
19   System.out.println("Ja sam: " + current.getName());
20
21  }
22
23 }
24
```

Pokretanjem programa dobit ćete ispis poput ispisa danog u nastavku.

```
java.lang.ThreadGroup[name=system,maxpri=10]
    Thread[Reference Handler,10,system]
    Thread[Finalizer,8,system]
    Thread[Signal Dispatcher,9,system]
    Thread[Attach Listener,5,system]
    java.lang.ThreadGroup[name=main,maxpri=10]
        Thread[main,5,main]

Ja sam: main
```

Program započinje tako da traži referencu na opisnik trenutne dretve (dretve koja upravo izvodi taj kod). Potom iz opisnika dohvaća referencu na opisnik grupe dretvi kojoj trenutna dretva pripada. Slijedi dio koda čija je zadaća pronaći inicijalnu grupu: prati se lanac roditelja tako dugo dok roditeljska grupa postoji. Kada dođemo do grupe koja nema roditelja, to je inicijalna grupa. Jednom kada je pronađena inicijalna grupa, nad njom se poziva metoda `.list()` koja

na standardni izlaz programa ispisuje čitavo stablo grupa i dretvi počev od grupe nad kojom je metoda pozvana. Kako smo metodu pozvali nad inicijalnom grupom, dobit ćemo ispis cjelovitog stanja u virtualnom stroju.

Uočite (na ispisu) da inicijalna grupa ima kao dijete grupu imena *main* unutar koje se nalazi dretva imena *main*: ta dretva je upravo ona koja izvodi kod metode `main` što se možemo uvjeriti iz posljednjeg retka ispisa.

Metode razreda `Thread`

Razred `Thread` sadrži nekoliko *statičkih* metoda od kojih ćemo se osvrnuti na tri. Metoda `Thread.currentThread` omogućava dretvi dohvat njezinog vlastitog opisnika (primjer uporabe vidjeli smo u primjeru 19.1). Metoda `Thread.sleep` kao argument prima broj milisekundi i dretvu koja je pozvala tu metodu uspavljuje na zatraženo vrijeme. U slučaju da se dretva probudi ranije, metoda `Thread.sleep` bacit će iznimku `java.lang.InterruptedException`. Konačno, metoda `Thread.yield` je metoda kojom trenutna dretva javlja virtualnom stroju da je sama spremna prepustiti procesor nekoj drugoj dretvi; na virtualnom stroju je da odluči hoće li to doista i napraviti ili će dretvu pozivatelja i dalje ostaviti na procesoru. Napomenimo da korektan višedretveni rad kod virtualnog stroja ne ovisi o tome hoće li dretve pozivati metodu `Thread.yield` ili ne; ona je tu više kao povijesni artifakt koji dretva može (ali ne mora) koristiti.

Od nestatičkih metoda također ćemo navesti nekoliko njih koje nudi razred `Thread`. Obratite pažnju da se te metode pozivaju nad opisnikom konkretne dretve (to nisu statičke metode) pa informacije koje vraćaju (ili postavljaju) ovise baš o konkretnoj dretvi čiji smo opisnik dohvatili. Metode su sljedeće.

Tablica 19.1. Pregled izbora nestatičkih metoda razreda Thread

Metoda	Opis
`getId()`	Vraća identifikator dretve. Svaka dretva imat će različit identifikator.
`getName()`	Vraća naziv dretve. Nazivi dretvi ne moraju biti jedinstveni.
`getPriority()`	Vraća prioritet dretve. Prioriteti se kreću od `Thread.MIN_PRIORITY` pa do `Thread.MAX_PRIORITY`. Srednji prioritet definiran je kao `Thread.NORM_PRIORITY`. Radi se o cijelim brojevima 1 (min), 5 (norm) i 10 (max).
`setPriority(...)`	Mijenja prioritet dretve na onaj predan argumentom.
`isAlive()`	Provjerava je li dretva čiji je ovo opisnik živa ili je gotova s izvođenjem dodijeljenog koda. Dretva koja to poziva nad svojim vlastitim opisnikom uvijek će dobiti `true`.
`getThreadGroup()`	Dohvaća opisnik grupe kojoj pripada dretva čiji je ovo opisnik.
`isDaemon()`	Provjerava je li dretva takozvana *demonska* dretva.
`setDaemon(...)`	Omogućava podešavanje je li dretva demonska ili nije.
`start()`	Nakon što je nova dretva stvorena i povezana s kodom koji treba izvoditi, ona još uvijek neće biti pokrenuta kako bi programer mo-

Metoda	Opis
	gao obaviti dodatna podešavanja (primjerice, definiranje prioriteta dretve). Nakon što je sve podešeno, potrebno je pozvati ovu metodu čime dretva postaje spremna za izvođenje: operacijski sustav će joj početi dodjeljivati procesor i krenut će izvođenje koda.
`join()`	Služi kao jednostavan sinkronizacijski mehanizam. Ako jedna dretva želi pričekati da neka druga dretva završi s poslom, treba dohvatiti njezin opisnik i nad njim pozvati ovu metodu. Taj će poziv trenutnu dretvu uspavati sve dok dretva nad čijim je opisnikom metoda pozvana ne završi sa zadanim poslom.

Načini stvaranja i pokretanja dretve

Java nudi dva načina kako stvoriti novu dretvu i specificirati posao koji ona mora obaviti. Prvi način (i svakako preporučeni) je posao enkapsulirati u zaseban razred koji implementira sučelje `java.lang.Runnable`. Ovo sučelje prikazano je u nastavku.

```
1 @FunctionalInterface
2 public interface Runnable {
3   public abstract void run();
4 }
```

Sučelje propisuje da mora postojati metoda imena `run` koja ne prima nikakve argumente niti išta vraća. Posao koji želimo pokrenuti u novoj dretvi možemo enkapsulirati u razred koji implementira ovo sučelje i posao zadaje u metodi `run`. Potom je dovoljno stvoriti primjerak tog razreda i referencu na taj primjerak poslati kroz konstruktor novom primjerku razreda `Thread`. Kada na taj način stvorimo novi opisnik dretve, on je povezan s modeliranim poslom nakon čega je dovoljno nad opisnikom pozvati metodu `start()`.

Program koji ilustrira ovaj način stvaranja nove dretve, zadavanja posla i pokretanje prikazan je u primjeru 19.2.

Primjer 19.2. Modeliranje posla sučeljem `java.lang.Runnable`

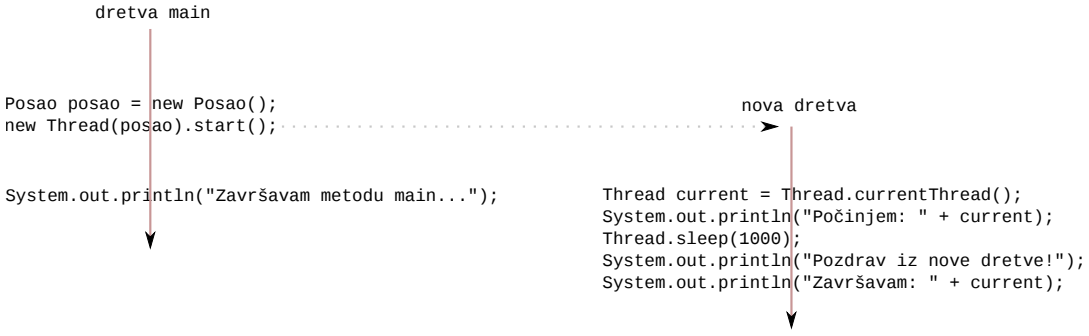
```
 1 package book.java.threads;
 2
 3 public class PokretanjeDretve1 {
 4
 5  private static class Posao implements Runnable {
 6   @Override
 7   public void run() {
 8    Thread current = Thread.currentThread();
 9    System.out.println("Počinjem: " + current);
10    try {
11     Thread.sleep(1000);
12    } catch (InterruptedException ignorable) {
13    }
14    System.out.println("Pozdrav iz nove dretve!");
15    System.out.println("Završavam: " + current);
16   }
17  }
18
19  public static void main(String[] args) {
20
```

```
21    Posao posao = new Posao();
22    new Thread(posao).start();
23
24    System.out.println("Završavam metodu main...");
25   }
26 }
27
```

U ovom primjeru posao je modeliran razredom `Posao`; sve što je potrebno napraviti zapisano je u metodi `run()`. Uočite, naravno, da ta metode može pozivati druge metode: nova dretva krenut će izvoditi taj kod, pa kud god da to ide dalje. Metoda `main` stvara primjerak posla, stvara novi opisnik dretve i predaje referencu na posao kroz konstruktor te odmah poziva metodu `start` čime novostvorena dretva postaje spremna za izvođenje. Nakon toga, dretva `main` izvodi još završni ispis i napušta metodu `main` nakon čega umire.

U sustavu je ostala još naša novostvorena dretva koja izvodi zadani posao. Na ekran ispisuje pozdravnu poruku, čeka jednu sekundu, ispisuje "Pozdrav iz nove dretve!", završnu poruku i umire. Malo poslije tog trenutka ugasit će se i sam proces čime završava izvođenje ovog programa.

Slika 19.1. Prikaz dretvi u programu



Ovisno od slučaja do slučaja (i računala, i operacijskog sustava i još svačega), ovaj program će rezultirati ili sljedećim ispisom:

```
Završavam metodu main...
Počinjem: Thread[Thread-0,5,main]
Pozdrav iz nove dretve!
Završavam: Thread[Thread-0,5,main]
```

ili sljedećim ispisom:

```
Počinjem: Thread[Thread-0,5,main]
Završavam metodu main...
Pozdrav iz nove dretve!
Završavam: Thread[Thread-0,5,main]
```

Naime, od trenutka kada smo pozvali `new Thread(posao).start();` imamo dvije dretve koje nastavljaju izvođenje programa: "glavna" dretva nastavlja prema ispisu poruke "`Završavam metodu main...`" dok novostvorena dretva kreće s izvođenjem metode `run` i s ispisom početne poruke. Koja od njih će to prva stići napraviti (i kojoj će operacijski sustav prvoj dodijeliti procesor ako imate samo jedan) nije unaprijed poznato (niti nas to zanima: ovo sve radimo upravo iz razloga što smo željeli biti u mogućnosti zadati poslove koji se obavljaju paralelno).

Drugi način koji nam stoji na raspolaganju je posao modelirati kao razred koji nasljeđuje opisnik dretve (odnosno `java.lang.Thread`). Taj novi razred mora nadjačati metodu `run()` koja je definirana u razredu `Thread`. Kod ovog pristupa, isti objekt će predstavljati i opisnik dretve i modelirani posao: oboje će stvoriti isti konstruktor. Program koji ovo ilustrira (a inače radi sve isto što i prethodni primjer) prikazan je u primjeru 19.3.

Primjer 19.3. Modeliranje posla nasljeđivanjem dretve

```
 1 package book.java.threads;
 2
 3 public class PokretanjeDretve2 {
 4
 5  private static class PosaoIDretva extends Thread {
 6   @Override
 7   public void run() {
 8    Thread current = Thread.currentThread();
 9    System.out.println("Počinjem: " + current);
10    try {
11     Thread.sleep(1000);
12    } catch (InterruptedException ignorable) {
13    }
14    System.out.println("Pozdrav iz nove dretve!");
15    System.out.println("Završavam: " + current);
16   }
17  }
18
19  public static void main(String[] args) {
20
21   new PosaoIDretva().start();
22
23   System.out.println("Završavam metodu main...");
24  }
25 }
26
```

Prikazani način, iako funkcionira, nećemo preporučiti jer krši jedno od osnovnih načela objektnog oblikovanja: načelo jedinstvene odgovornosti. Pristup koji smo prikazali spaja i model posla i dretve u jedan razred čime šalje vrlo čudnu poruku da je posao ujedno i dretva.



Bilješka

Preporučeni način modeliranja posla je uporabom sučelja `java.lang.Runnable`. Na taj način posao se razdvaja od dretve koja će ga izvoditi. Time se također omogućava da se isti posao (objekt) pošalje na izvođenje u više dretvi.

Povučene metode

Pažljivim čitanjem dokumentacije razreda `Thread` vjerojatno ćete uočiti još neke metode. Metode koje obično plijene pažnju programera su metode koje jednoj dretvi omogućavaju da upravljaju drugom dretvom u smislu da je unište (metoda `destroy`), privremeno zaustave (metoda `suspend`) pa nakon toga pokrenu (metoda `resume`) ili pak da je trajno zaustave (metoda `stop`). S obzirom da ne postoji način da se ove metode implementiraju na siguran način, iste su od ranih verzija Jave proglašene povučenim (engl. *deprecated*). Nemojte koristiti te metode.

Kada završava Java-program?

Nakon prethodna dva primjera i popratne diskusije, možda ste se zapitali kada točno umire Vaš Java-program? I što su to točno *demonske* dretve? Prema specifikaciji programskog jezika Java, svaka dretva može biti regularna dretva ili demonska dretva. Demonske dretve obično obavljaju pozadinske poslove: poslove koji ne služe izravno korisniku, ali tako dugo dok se unutar procesa obavlja nešto zanimljivo, trebaju nam i ti pozadinski poslovi (sjetite se samo skupljača smeća). I sam korisnik prilikom stvaranja novih dretvi nad opisnikom dretve može pozvati metodu `setDaemon(true)` i time dretvu proglasiti demonskom. Po ničemu drugome demonske dretve nemaju poseban tretman: one su dretve kao is sve ostale.

Postoji samo jedan sitan detalj: odgovor na pitanje do kada će virtualni stroj proces održavati na životu. Odgovor je: tako dugo dok u procesu postoji barem jedna regularna (nedemonska) dretva. U trenutku kada u procesu ostanu samo demonske dretve, virtualni stroj terminira proces.

Da je tome tako, lagano ćete se uvjeriti ako malo modificirate jedan od prethodna dva primjera. Vratite se na primjer 19.2 i modificirajte dio metode `main` tako da umjesto:

```
22 new Thread(posao).start();
```

napišete:

```
22 Thread thread = new Thread(posao);
23 thread.setDaemon(true);
24 thread.start();
```

Razlika je, dakle, u dodatnom pozivu kojim novu dretvu zaduženu za izvođenje posla proglašavate demonskom. Pokrenete li program, u najboljem ćete slučaju dobiti ispis:

```
Završavam metodu main...
Počinjem: Thread[Thread-0,5,main]
```

i ništa više. Iako posao nove dretve neće biti do kraja izvršen, u trenutku kada dretva `main` završi sa izvođenjem, u procesu neće ostati niti jedna nedemonska dretva i proces će se ugasiti.



Bilješka

U Javi, proces se gasi kada u njemu završi posljednja nedemonska dretva. Alternativno, proces neovisno o broju i vrsti postojećih dretvi možete uništiti pozivom metode `System.exit(x);` gdje je `x` povratna vrijednost (cijeli broj) koju proces vraća operacijskom sustavu.

Jednostavna sinkronizacija dretvi

Jedna od primjena višedretvenosti je paralelizacija izvođenja posla. To ćemo prikazati na jednostavnom primjeru. Pretpostavimo da našem programu treba polje od 64K slučajno izgeneriranih realnih brojeva. Isječak programa koji rješava ovaj problem prikazan je u nastavku.

```
1 double[] polje = new double[1024*64];
2
3 Random rand = new Random();
4 for(int i = 0; i < polje.length; i++) {
5   polje[i] = rand.nextDouble();
6 }
7
8 System.out.println("Imam čitavo polje slučajnih brojeva!");
```

Ako ovakvo polje u programu trebamo često, pitanje je možemo li ga izgenerirati brže, ako na raspolaganju imamo računalo s više procesora (primjerice, 4)? Odgovor je, naravno, da. Posao popunjavanja čitavog polja poopćit ćemo u posao koji prima referencu na polje i njegov dio (od kojeg indeksa, uključivo, do kojeg indeksa, isključivo) koji treba popuniti slučajnim brojevima. Potom ćemo zadatak popunjavanja čitavog polja razlomiti na više disjunktnih podzadataka. Primjerice, ako zadatak razlomimo na 4 podzadatka, ti podzadatci mogu biti:

- popunjavanje polja na indeksima od 0 do 16K,
- popunjavanje polja na indeksima od 16K do 32K,
- popunjavanje polja na indeksima od 32K do 48K te
- popunjavanje polja na indeksima od 48K do 64K.

Svaki podzadatak modelirat ćemo jednim primjerkom razreda `Posao`. Za svaki posao stvorit ćemo po jednu dretvu koja će taj posao odraditi i sve ćemo dretve pokrenuti. Raspolažete li s računalom s četiri jezgre, svaki će se posao (potencijalno) odraditi istovremeno na zasebnoj jezgri.

Imamo samo jedan nezgodan problem. Naša dretva `main` koja je alocirala polje brojeva, stvorila i pokrenula dretve-radnike koji popunjavaju dijelove polja slučajnim brojevima, treba to polje. Ali kako ona više ne popunjava to polje, mora nekako doznati je li zadatak popunjavanja polja slučajnim brojevima završen (drugim riječima, jesu li sve dretve-radnici obavile svoj posao). Pretpostavimo za potrebe ovog primjera da smo zapamtili polje opisnika dretvi koje smo stvorili i da se to polje zove `radnici`. Jedna mogućnost rješavanja opisanog problema je da glavna dretva krene redom prozivati opisnik po opisnik i ispitivati ga je li njegova dretva još uvijek živa; tako dugo dok je, nastavili bismo postupak ispitivanja. Rješenje je skicirano u nastavku.

```
1 for(int i = 0; i < radnici.length; i++) {
2  while(true) {
3   if(!radnici[i].isAlive()) break;
4  }
5 }
```

Opisani pristup radi ali je vrlo neefikasan. Naime, glavna dretva, kada dobije procesor, ne radi ništa drugo nego u petlji ispituje opisnik dretve je li dretva gotova, pa ako nije, ispituje opet. Time se troši procesorsko vrijeme koje smo mogli bolje utrošiti: umjesto da procesor bespotrebno vrti ovaj kod, mogao je izvoditi jednog od radnika i popunjavati polje slučajnim brojevima. Ovakav primjer *aktivnog čekanja* je nešto što želimo izbjegavati.

Bolje rješenje možemo postići ako aktivno ispitivanje zamijenimo blokirajućim čekanjem: možemo tražiti operacijski sustav da nas zablokira (uspava) sve dok dretva čiji opisnik imamo ne završi s radom. Time smo odmah operacijskom sustavu omogućili i efikasnije raspoređivanje procesorskog vremena jer se naša dretva sada više ne natječe s radnicima za vrijeme na procesoru (i ne troši ga beskorisno). Opisano blokirajuće čekanje moguće je ostvariti pozivom metode `join()` nad opisnikom dretve. Taj će poziv uspavati *pozivatelja* sve dok dretva nad čijim je opisnikom poziv obavljen ne bude gotova s poslom.

Ideju ovog rješenja ilustrira kod prikazan u nastavku.

```
1 for(int i = 0; i < radnici.length; i++) {
2  radnici[i].join();
3 }
```

Postoji samo jedan sitan implementacijski detalj: poziv `join()`, iako je blokirajući, može završiti i prije no što se ona druga dretva uistinu završi s poslom. Primjerice, Java nam dozvoljava da neka treća dretva (koja ima referencu na opisnik ove dretve) inicira prekid čekanja (metodom `interrupt()`). U takvom slučaju, međutim, poziv metode `join()` neće završiti regularno već s iznimkom `InterruptedException`. Ako se to dogodi (i ako stvarno želimo dočekati da se posao odradi do kraja), prethodni bi kod trebao poziv `radnici[i].join();` okružiti u blok `try { ... } catch(InterruptedException ex) { ... }` i ponavljati ga u petlji tako dugo dok taj poziv metode ne završi regularno (bez iznimke).

Cjelovito rješenje koje ovo ilustrira prikazano je u primjeru 19.4 danom u nastavku.

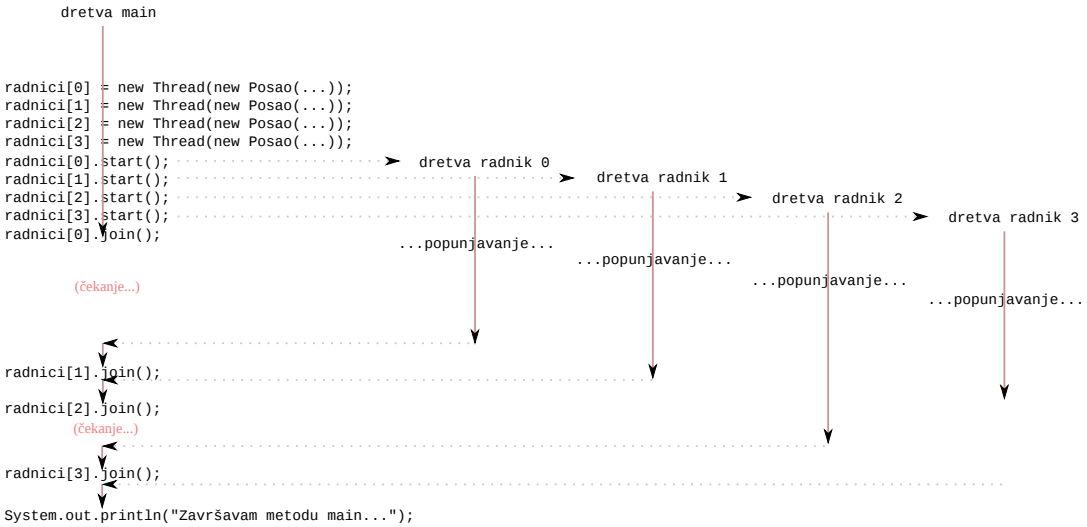
Primjer 19.4. Sinkronizacija dretvi metodom `join()`

```
1 package book.java.threads;
2
3 import java.util.Random;
4
5 public class JednostavnaSinkronizacija {
6
7  // Model posla popunjavanja dijela polja
8  private static class Posao implements Runnable {
```

```
 9   private double[] polje;
10   private int pocetniIndeks;
11   private int zavrsniIndeks;
12
13   public Posao(double[] polje, int pocetni, int zavrsni) {
14    super();
15    this.polje = polje;
16    this.pocetniIndeks = pocetni;
17    this.zavrsniIndeks = zavrsni;
18   }
19
20   @Override
21   public void run() {
22    Random rand = new Random();
23    for(int i = pocetniIndeks; i < zavrsniIndeks; i++) {
24     polje[i] = rand.nextDouble();
25    }
26   }
27  }
28
29  public static void main(String[] args) {
30
31   double[] polje = new double[1024*64];
32
33   final int brojRadnika = 4;
34   Thread[] radnici = new Thread[brojRadnika];
35
36   // Stvori poslove i opisnike dretvi
37   for(int i = 0; i < radnici.length; i++) {
38    Posao posao = new Posao(
39     polje,
40     polje.length*i/radnici.length,
41     polje.length*(i+1)/radnici.length);
42    radnici[i] = new Thread(posao);
43   }
44
45   // Pokreni dretve
46   for(int i = 0; i < radnici.length; i++) {
47    radnici[i].start();
48   }
49
50   // Čekaj jednu po jednu dretvu da završi
51   for(int i = 0; i < radnici.length; i++) {
52    while(true) {
53     try {
54      radnici[i].join();
55      break;
56     } catch (InterruptedException e) {
57      // Idemo opet zvati join()
58     }
59    }
60   }
61
62   System.out.println("Imam čitavo polje slučajnih brojeva!");
63  }
64
65 }
66
```

Jedan mogući tijek programa i utjecaj metode `join()` na izvođenje glavne dretve prikazani su na slici 19.2.

Slika 19.2. Prikaz dretvi u programu



U ovom primjeru nakon stvaranja i pokretanja svih četiriju radnika, glavna dretva poziva `radnici[0].join()` i tu biva blokirana. Glavna dretva ostaje blokirana na ovom pozivu sve dok dretva-radnik 0 ne završi s poslom. U tom trenutku glavnoj se dretvi dopušta nastavak rada, i ona poziva `radnici[1].join()`. U ovom scenariju dretva-radnik 1 baš negdje u tom trenutku završava s poslom pa glavna dretva odmah nastavlja dalje (ili biva blokirana na kratko vrijeme) i poziva `radnici[2].join()`. Kako je (iz nekog razloga) posao dretve-radnika 2 ispao malo zahtjevniji (ili joj operacijski sustav nije davao jednak broj vremenskih isječaka kao i drugim radnicima), ova dretva još nije gotova s poslom pa je glavna dretva ponovno blokirana i čeka. U trenutku kada dretva-radnik 2 konačno završi, glavna dretva se ponovno odblokira i poziva `radnici[3].join()`. Ovaj poziv glavnu dretvu ne blokira jer je dretva-radnik 3 već prije završila, tako da glavna dretva nastavlja dalje prema ispisu završne poruka. U tom trenutku glavna dretva ima garanciju da su svi radnici završili s poslom i može koristiti rezultate njihova rada.

Tipični problemi višedretvenih aplikacija

Što se događa kada više dretvi krene mijenjati *iste* podatke u memoriji? Ovo ćemo prikazati jednostavnim programom danim u nastavku. Prepišite program i isprobajte ga.

Primjer 19.5. Čitanje i pisanje više dretvi po istoj varijabli

```
 1 package book.java.threads;
 2
 3 public class DijeljenjePodataka {
 4
 5  public static long counter = 0L;
 6
 7  private static class Posao implements Runnable {
 8   private int n;
 9
10   public Posao(int n) {
11    super();
12    this.n = n;
13   }
14
```

```
15    @Override
16    public void run() {
17     for(int i = 0; i < n; i++) {
18      counter++;
19     }
20    }
21   }
22
23   public static void main(String[] args) {
24
25    final int brojRadnika = 2;
26    Thread[] radnici = new Thread[brojRadnika];
27    Posao posao = new Posao(1_000_000);
28
29    for(int i = 0; i < radnici.length; i++) {
30     radnici[i] = new Thread(posao);
31    }
32
33    for(int i = 0; i < radnici.length; i++) {
34     radnici[i].start();
35    }
36
37    for(int i = 0; i < radnici.length; i++) {
38     while(true) {
39      try {
40       radnici[i].join();
41       break;
42      } catch (InterruptedException e) {
43       // Idemo opet zvati join()
44      }
45     }
46    }
47
48    System.out.println("Konačno stanje brojača: " + counter);
49   }
50
51  }
52
```

Program deklarira jednu zajedničku varijablu (statička varijabla `counter`) te posao uvećavanja te varijable zadan broj puta modelira statičkim ugniježđenim razredom `Posao`. U metodi `run()` u petlji `for` obavlja se uvećavanje varijable `counter` zadani broj puta. Glavni program stvara primjerak ovog posla, stvara dvije dretve koje ga obavljaju, pokreće ih, čeka da dretve završe, čita vrijednost varijable `counter` i ispisuje je na zaslon. Što očekujete da će se ispisati? Modificirajte program tako da umjesto milijun puta svaki dretva obavi uvećavanje 10 (pa 100) milijun puta. Kakav je rezultat? što se događa ako umjesto dva radnika stvorite njih 5? 10?

Ako ste napravili prethodne eksperimente, ovisno o računalu na kojem radite moguće je da ste već na prvom eksperimentu dobili konačan rezultat koji je manji od broja dretvi pomnoženog brojem uvećavanja koji svaka dretva radi. Zašto je došlo do toga? Da bismo odgovorili na to pitanje, moramo zaviriti *ispod haube*. Kako izgleda strojni kod koji izvodi Javin virtualni stroj za ovaj program?

Da bismo za prethodno napisani program došli do prikaza byte-koda, u okviru JDK-a dostupna je naredba **javap**. Pozivom uz argument "-c" na zaslon će ispisati instrukcije svake od metoda razreda koji je naveden kao zadnji argument. Naredba u nastavku zadana je uz pretpostavku da se trenutni direktorij onaj koji sadrži poddirektorije `src` i `bin`, te da je program već preveden.

```
d:\Dretve>javap -cp bin -c book.java.threads.DijeljenjePodataka.Posao
```

```
Compiled from "DijeljenjePodataka.java"
class book.java.threads.DijeljenjePodataka$Posao
      implements java.lang.Runnable {

  public book.java.threads.DijeljenjePodataka$Posao(int);
    Code:
       0: aload_0
       1: invokespecial #12   // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iload_1
       6: putfield      #15   // Field n:I
       9: return

  public void run();
    Code:
       0: iconst_0
       1: istore_1
       2: goto          16
       5: getstatic     #22   // Field DijeljenjePodataka.counter:J
       8: lconst_1
       9: ladd
      10: putstatic     #22   // Field DijeljenjePodataka.counter:J
      13: iinc          1, 1
      16: iload_1
      17: aload_0
      18: getfield      #15   // Field n:I
      21: if_icmplt     5
      24: return
}
```

Od posebnog interesa ovdje je unutrašnjost petlje `for` u metodi `run()`. Naredba koju smo imali u izvornom kodu bila je `counter++;`. Java prevoditelj ovo je preveo u četiri instrukcije[6]:

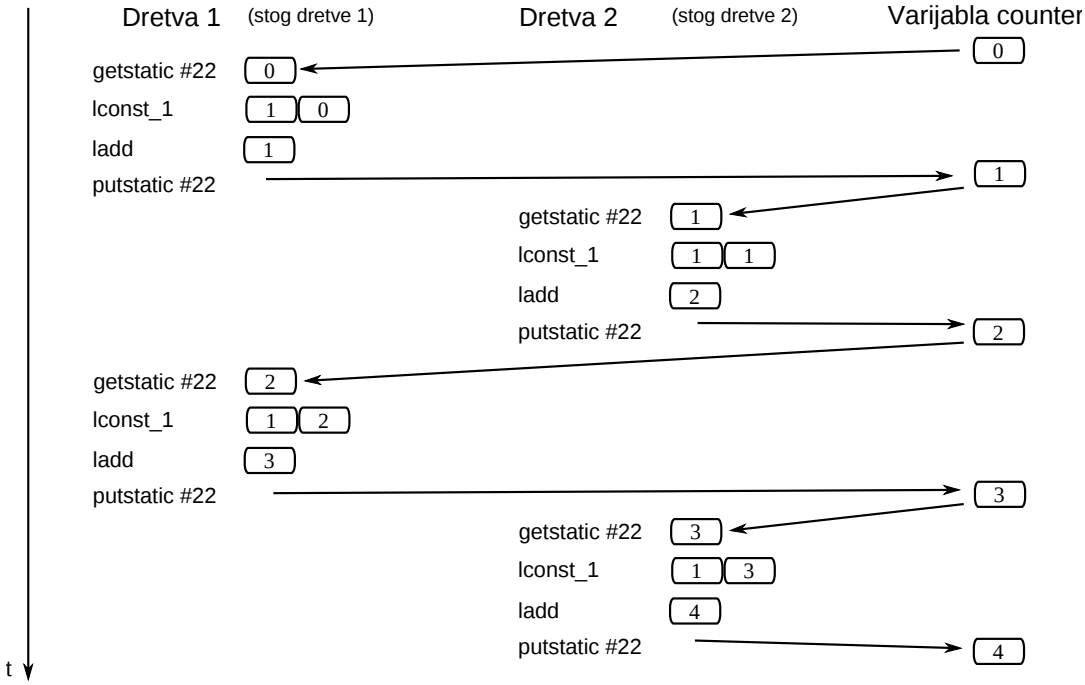
```
getstatic     #22   // Field DijeljenjePodataka.counter:J
lconst_1
ladd
putstatic     #22   // Field DijeljenjePodataka.counter:J
```

Prva instrukcija obavlja dohvat sadržaja varijable `counter` iz glavne memorije računala i pročitanu vrijednost pohranjuje na stog. Druga instrukcija na stog ubacuje vrijednost `1` (nakon toga su na stogu `1` pa pročitani sadržaj varijable `counter`). Treća instrukcija sa stoga skida dvije vrijednosti i na stog vraća njihov zbroj; time je po izvođenju te instrukcije na stog pohranjena izračunata vrijednost `counter+1`. Konačno, četvrta instrukcija skida sa stoga vrijednost i zapisuje je u memoriju: varijabla `counter` time dobiva novu vrijednost koja je uvećana za 1.

Što se događa kada ovaj kod počne izvoditi više dretvi na jednoprocesorskom sustavu? Uzmimo primjer u kojem dvije dretve ovaj isječak koda ponove 2 puta. Što možemo očekivati kao rezultat? Slika 19.3 prikazuje instrukcije koje izvode obje dretve.

[6]Instrukcijski set javinog virtualnog stroja dostupan je na adresi https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5

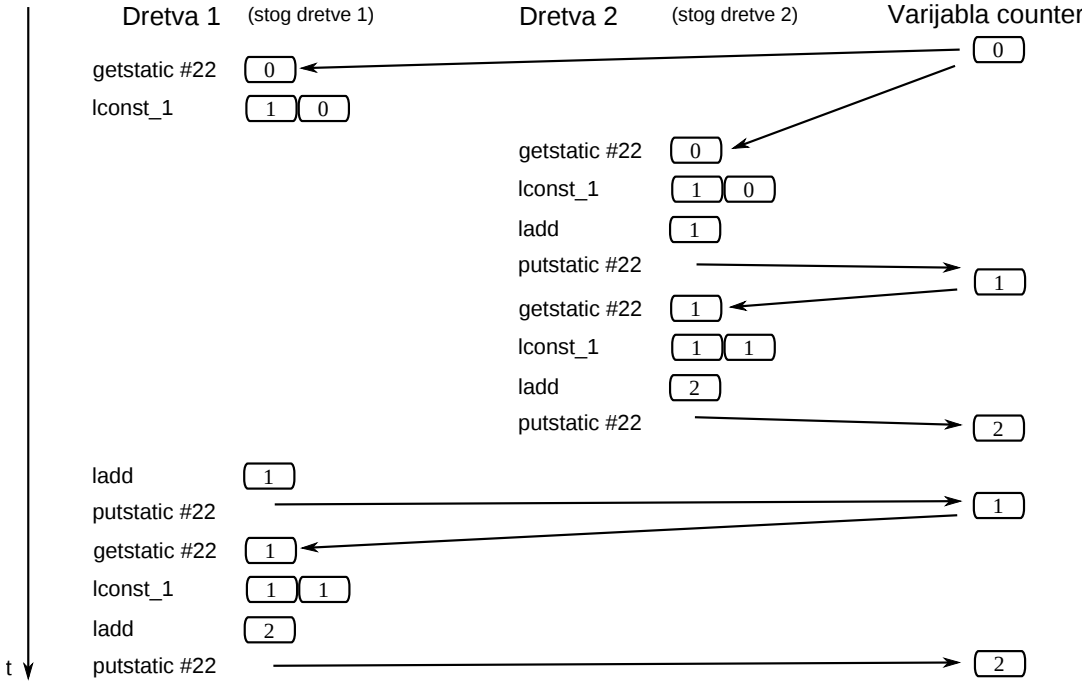
Slika 19.3. Jedan mogući slijed izvođenja instrukcija dviju dretvi



Na slici vrijeme ide prema dolje; kako je samo jedan procesor, u istom trenutku nije moguće izvođenje više instrukcija. Da bismo razumjeli što se događa, važno je prisjetiti se da svaka dretva ima svoj vlastiti stog. U ovom primjeru procesor najprije izvede 4 instrukcije prve dretve (čime se `counter` poveća na 1), potom 4 instrukcije druge dretve (čime se `counter` poveća na 2), potom 4 instrukcije prve dretve (čime se `counter` poveća na 3), i konačno posljednje 4 instrukcije druge dretve (čime se `counter` poveća na 4). Kako razmatramo primjer gdje dvije dretve brojač uvećavaju dva puta, rezultat je u skladu s očekivanjem: 4.

Ovo, međutim, nije jedini mogući način kojim su instrukcije mogle biti izvođene. Slika 19.4 prikazuje drugačiji scenarij.

Slika 19.4. Još jedan mogući slijed izvođenja instrukcija dviju dretvi



U ovom slučaju nakon izvođenja prve dvije instrukcije prve dretve na procesoru su izvedene sve instrukcije druge dretve i tek tada ostatak instrukcija prve dretve. Konačni rezultat sada je samo 2. Vidite li koji je razlog tome? Problem leži u tome što operacija uvećavanja nije atomarna. Svaka dretva taj posao obavi tako da najprije sadržaj varijable prekopira u svoju lokalnu memoriju (na stog), tamo radi operaciju nad tim podatkom i potom rezultat vraća natrag u memoriju varijable. Ako se dretvu prekine nakon što je preuzela vrijednost varijable ali nije novu vratila u memoriju, dolazimo do problema: druga dretva sada će iz memorije pročitati vrijednost varijable, obaviti određene operacije nad njom i ažurirati je, nakon čega će taj sam posao biti poništen kada prva dretva svoj rezultat iz svoje lokalne memorije zapiše natrag u varijablu.

Znate li koliko iznosi minimalna vrijednost ovakvog brojača ako u sustav dodamo više dretvi? Primjerice, ako uzmemo 10 dretvi koje svaka vrijednost povećava dva puta? Ili 100 dretvi? Odgovor je uvijek isti i ne ovisi o broju pokrenutih dretvi: minimalna vrijednost brojača bit će 2. Možete li osmisliti takav scenarij za slučaj da ste pokrenuli 10 dretvi?

U nastavku ovog podpoglavlja navest ćemo dva rješenja ovog problema. Prvo rješenje bit će uz pomoć samog procesora i sklopovske instrukcije koja atomarno može sadržaj memorije zamijeniti nekim drugim ako je trenutni sadržaj jednak specificiranom. Drugo rješenje oslonit će se na definiranje kritičnih odsječaka: to će biti dijelovi koda koje u jednom trenutku smije izvoditi samo jedna dretva i dok ona ne završi, niti jedna druga dretva neće moći započeti njegovo izvođenje.

Prije toga, osvrnut ćemo se na još jedan problem koji prethodni kod ima a koji je do sada bio namjerno prešućen. Pogledajte još jednom (ali sada malo pažljivije) kako je prevodilac preveo metodu `run()` napisanog posla. U nastavku ćemo dati prikaz tog koda mješavinom Jave i pseudokoda izgeneriranog iz bajt-koda.

```
for(int i = 0; i < n; i++) {
  tmp <-- counter
  tmp <-- tmp + 1
  counter <-- tmp
}
```

U ovom prikazu strelica označava prijenos podataka. Tako se u prvom retku u pomoćnu varijablu (stog dretve) upisuje vrijednost varijable `counter` koja se dohvaća iz memorije. Potom se pomoćna varijabla uvećava za jedan. Konačno, sadržaj pomoćne varijable zapisuje se natrag u memoriju na mjesto varijable `counter`.

Razmislite sada, ako je cilj generirati strojni kod koji će maksimalno brzo izvesti ovu metodu, jesmo li mogli nekako smanjiti broj operacija koje radimo pa time prije biti gotovi? Ono što svakako plijeni pažnju je ovo neprestano seljenje podataka između memorije i stoga. Optimizirajući jezični procesor (ili *just-in-time compiler*) bi instrukcije mogao presložiti na sljedeći način.

```
tmp <-- counter
for(int i = 0; i < n; i++) {
  tmp <-- tmp + 1
}
counter <-- tmp
```

Ovakvom organizacijom koda (i u slučaju jednodretvenog programa) rezultat izvođenja (na kraju petlje `for` bit će identičan onome koji bismo dobili da nismo radili ovu reorganizaciju. Ovaj novi program najprije iz memorije dohvati sadržaj varijable `counter` i spremi ga u lokalnu varijablu. Potom lokalnu varijablu uveća `n` puta i tek kad je unutrašnja petlja gotova, rezultat iz lokalne varijable zapiše natrag u varijablu `counter`. Takvo preslagivanje smanjilo je broj čitanja-pisanja varijable `counter` s 2*`n` na samo 2. Ovaj isječak koda sigurno će se izvesti

brže. Posebice uzmete li u obzir da se u stvarnosti bajt-kod prevodi u strojni kod stvarnog procesora na kojem će se program izvesti i da se pri tome ovakva keširanja vrijednosti ("stvorene" lokalne varijable) mogu umjesto stogom izvesti uporabom registara procesora; u tom slučaju osjetit ćemo značajnu razliku u brzini izvođenja jer je pristup podatku koji je u registru procesora višestruko brži od pristupa podatku koji je u memoriji.

Prikazan način optimizacije (koji današnji prevodioci uredno rade) ima smisla za jednodretvene programe. Ali zamislite što se sada tek može dogoditi u višedretvenom okruženju. Umjesto da svaka dretva uvećava sadržaj varijable, moguće je da svaka dretva kešira vrijednost koju pročita, lokalno radi nad keširanom vrijednosti i onda samo na kraju zapiše rezultat. To neće biti dobro. U takvom scenariju moguće je da jedna dretva uopće ne vidi da je neka druga dretva varijablu promijenila u memoriji, jer je strojni kod presložen tako da pri ulasku u metodu pročita i kešira vrijednost i potom gleda samo svoj lokalni keš.

Da bi se spriječila izvedba ovakvih optimizacija koda, varijable koje se bez ikakve dodatne sinkronizacije koriste od više dretvi potrebno je označiti ključnom riječi `volatile`. U primjeru 19.5, to bi značilo da deklaraciju varijable `counter` u retku 5 treba promijeniti iz:

```
5   public static long counter = 0L;
```

u deklaraciju koja varijablu označava `volatile`:

```
5   public static volatile long counter = 0L;
```

Tom modifikacijom i dalje ostaju prethodno prikazani izvorni problemi višedretvenog izvođenja ovog koda; razlika je što sada znamo da su to jedini problemi koji su preostali. Problema uslijed keširanja sadržaja varijable sada više neće biti, jer za sve varijable koje su označene kao `volatile` prevodilac mora generirati kod koji svaki puta odlazi u memoriju po "svježi" sadržaj.

Uporaba instrukcije CAS

Instrukcija CAS (*Compare-And-Set*; ponegdje poznata i pod imenom *Test-And-Set* ili TAS) je instrukcija koja ima tri operanda (argumenta): memorijsku lokaciju, očekivani sadržaj na toj memorijskoj lokaciji te novi sadržaj koji tamo treba zapisati ali samo pod uvjetom da je trenutni sadržaj memorije bio jednak očekivanom. Instrukcija pozivatelju ujedno vraća status je li upisivanje obavljeno (ili je netko u međuvremenu promijenio sadržaj memorije, pa on više nije bio jednak očekivanom zbog čega je i samo zapisivanje otkazano). Važno je napomenuti da je to jedna atomarna instrukcija procesora koja će, jednom kada je pokrenuta, odraditi čitav posao bez mogućnosti prekidanja.

Vratimo li se na prethodno prikazani zadatak višedretveno-sigurnog modificiranja sadržaja varijable `counter`, metodu za uvećavanje sadržaja mogli bismo napisati na način prikazan u nastavku (tretirajte kod kao pseudokod).

```
1 class Primjer {
2   public volatile long counter = 0L;
3
4   public void uvecajNaSiguranNacin(int n) {
5     // Želimo uvećati n puta
6     for(int i = 0; i < n; i++) {
7       // Pokušavaj uvećati za jedan sve dok ne uspiješ
8       while(true) {
9         long trenutnaVrijednost = counter;
10        long novaVrijednost = trenutnaVrijednost + 1;
11        boolean uspjeh =
12          CAS(counter, trenutnaVrijednost, novaVrijednost);
13        if(uspjeh) break;
14      }
15    }
```

```
16    }
17
18 }
```

Pretpostavite (za potrebe primjera) da smo stvorili dvije dretve, i obje su pozvale metodu `uvecajNaSiguranNacin` s argumentom 1000000. Imamo li na raspolaganju instrukciju CAS, uvećavanje za 1 mogli bismo ostvariti petljom `while` kako je prikazano u prethodnom primjeru. U petlji radimo 4 koraka.

1. U lokalnu varijablu `trenutnaVrijednost` dohvaćamo sadržaj varijable `counter` iz memorije (svaka dretva ima svoju vlastitu lokalnu varijablu).

2. Na temelju pročitane vrijednosti zapisane u lokalnoj varijabli `trenutnaVrijednost` računamo novu vrijednost koju želimo zapisati u varijablu `counter` i to pohranjujemo u pomoćnu varijablu `novaVrijednost`. Važno je da početno pročitani sadržaj varijable `counter` koji smo pospremili u varijablu `trenutnaVrijednost` ne modificiramo - taj nam sadržaj treba za sljedeći korak.

3. Sada kada imamo spremnu pročitanu vrijednost varijable `counter` na temelju koje smo izračunali novu vrijednost koju želimo da ta varijabla poprimi, možemo zatražiti procesor da vrijednost varijable `counter` atomarno ažurira na vrijednost `novaVrijednost` ali samo ako je u varijabli `counter` još uvijek upisana vrijednost koju smo zapamtili u `trenutnaVrijednost` te da nam javi je li upisivanje provedeno. Instrukcija će odbiti upis nove vrijednosti ako je netko u međuvremenu promijenio sadržaj varijable `counter` pa njezina trenutna vrijednost više nije jednaka onoj zapamćenoj u `trenutnaVrijednost`.

4. Posljednji korak je provjeriti je li upisivanje naredbom CAS uspjelo. Ako je naredba vratila uspjeh, prekidamo petlju `while` jer smo jedno uvećavanje napravili. Ako je CAS vratila neuspjeh, to znači da je neka druga dretva u međuvremenu uspjela izvesti barem jedan kompletan ciklus *čitaj-računaj-zapiši*. U tom slučaju znamo da moramo krenuti ispočetka da ne bismo pregazili rezultat koji je odradila druga dretva.

Pisanjem koda kako je prethodno pokazano ne može se pojaviti situacija da se dvije dretve međusobno blokiraju. Naime, ako je u jednoj dretvi naredba CAS uspjela, ona je odradila jedak korak pa je bliže kraju. Ako dretvi nije uspjela naredba CAS, to je stoga što je nekoj drugoj dretvi tada uspjela. U tom slučaju ta je druga dretva korak bliže kraju posla. U konačnom broju koraka, cjelokupni će posao biti izveden.

U programkom jeziku Java atomarne instrukcije procesora ne stoje nam na raspolaganju. Umjesto toga, imamo na raspolaganju nekoliko razreda iz paketa `java.util.concurrent.atomic` koji enkapsuliraju ovu instrukciju i nude je kroz nekoliko povezanih razreda. Primjerice, razred `AtomicLong` je razred koji deklarira jednu (`volatile`) člansku varijablu tipa `long` i u osnovnoj inačici nudi metodu za dohvat sadržaja te varijable (metoda `get()`) te metodu `compareAndSet(expectedValue, newValue)` koja atomički modificira vrijednost na prethodno opisani način. Primjerke razreda `AtomicLong` možemo koristiti kao varijable koje predstavljaju cijele brojeve koje možemo atomički mijenjati. Način na koji možemo preraditi primjer 19.5 kako bismo postigli višedretveno-sigurnu semantiku koja se oslanja na `AtomicLong` prikazan je u nastavku.

Primjer 19.6. Čitanje i pisanje više dretvi po istoj varijabli uporabom `AtomicLong`

```
1 package book.java.threads;
2
3 import java.util.concurrent.atomic.AtomicLong;
4
5 public class DijeljenjePodataka5 {
6
7  public static AtomicLong counter = new AtomicLong(0L);
```

```
 8
 9  private static class Posao implements Runnable {
10   private int n;
11
12   public Posao(int n) {
13    super();
14    this.n = n;
15   }
16
17   @Override
18   public void run() {
19    for(int i = 0; i < n; i++) {
20     while(true) {
21      long current = counter.get();
22      if(counter.compareAndSet(current, current+1)) {
23       break;
24      }
25     }
26    }
27   }
28  }
29
30  public static void main(String[] args) {
31
32   final int brojRadnika = 2;
33   Thread[] radnici = new Thread[brojRadnika];
34   Posao posao = new Posao(10_000_000);
35
36   for(int i = 0; i < radnici.length; i++) {
37    radnici[i] = new Thread(posao);
38   }
39
40   for(int i = 0; i < radnici.length; i++) {
41    radnici[i].start();
42   }
43
44   for(int i = 0; i < radnici.length; i++) {
45    while(true) {
46     try {
47      radnici[i].join();
48      break;
49     } catch (InterruptedException e) {
50      // Idemo opet zvati join()
51     }
52    }
53   }
54
55   System.out.println("Konačno stanje brojača: " + counter.get());
56  }
57
58 }
59
```

Metoda `run()` u prikazanom kodu sadrži gotovo pa doslovno prepisan prethodno pojašnjeni pseudokod. Razlika je što je ovo sada legalan Java-kod koji možete prevesti, pokrenuti i uvjeriti se da radi. Za znatiželjnije preporučamo da pogledate popis svih metoda koje nudi razred `AtomicLong` te da zavirite u izvorni kod tog razreda. Otkrit ćete da razred nudi i metode za inkrement vrijednosti, dekrement vrijednosti i slične (koje u svojoj implementaciji imaju upravo

opisanu petlju koja uporabom CAS-a pokušava provesti traženu operaciju). Izravnom uporabom ovih metoda, metodu `run()` smo mogli kraće napisati kako je prikazano u nastavku.

```
1 public void run() {
2  for(int i = 0; i < n; i++) {
3   counter.incrementAndGet();
4  }
5 }
```

Razred `AtomicLong` ne nudi direktno metodu `increment()` već metodu imena `incrementAndGet` koja povećava broj u memoriji i vraća njegovu novu vrijednost. Kako nam u ovom primjeru ta vrijednost nije bitna, povratnu vrijednost nigdje ne pohranjujemo.

Uz razred `AtomicLong` paket `java.util.concurrent.atomic` nudi još niz drugih sličnih razreda za različite jednostavne tipove podataka (primjerice, `AtomicBoolean`, `AtomicInteger` i `AtomicReference`). Zajednički nazivnik svim tim implementacijama je da ih možemo koristiti samo za rad s varijablama čiju vrijednost procesor može atomarno čitati i zapisivati. Uz to, na raspolaganju su nam i implementacije koje enkapsuliraju polja primitivnih elemenata i omogućavaju atomarno ažuriranje elemenata na određenoj poziciji (nije moguće atomarno ažurirati čitavo polje već samo jedan od elemenata); za to služe razredi `AtomicIntegerArray`, `AtomicLongArray` te `AtomicReferenceArray`. Opisani mehanizam nećemo moći direktno koristiti za provođenje složenijih operacija koje treba obaviti atomarno (poput ubacivanja novog čvora u listu). Takvi općenitiji scenariji zahtjevat će izgradnju kritičnih odsječaka i *monitora*. Više o tome slijedi u nastavku.



Bilješka

Uporaba atomičkih razreda odnosno CAS instrukcija može se koristiti kao zamjena za složenije sinkronizacijske mehanizme samo u slučaju kada se ažurira *jedna* (primitivna) varijabla.

Izgradnja kritičnog odsječka

Općenitije sinkronizacijske mehanizme (i višedretveno sigurne operacije) možemo graditi oslanjajući se na kritične odsječke. Pojam kritičnog odsječka podrazumijeva jedan ili više nizova instrukcija koje u jednom trenutku može izvoditi samo jedna dretva. Kritični odsječak možemo zamisliti kao sobu s dvoja vrata ispred kojih stoje vratari koji mogu komunicirati uporabom walkie-talkieja. Kroz jedna vrata se ulazi u sobu a kroz druga izlazi. Osoba koja želi ući u sobu i napraviti određen posao mora proći pored vratara koji upravlja ulaznim vratima. Ako se u sobi već netko nalazi, vratar će ulazna vrata držati zatvorenima sve dok mu vratar s izlaznih vrata ne dojavi da je prethodna osoba napustila sobu. Tek u tom trenutku vratar kroz ulazna vrata pušta sljedeću (jednu) osobu, i postupak se ponavlja.

Jedan vratar može upravljati i s više soba. U tom slučaju, vratar se nalazi u hodniku koji dalje vodi prema svim sobama i kroz taj hodnik propušta jednu po jednu osobu. Sljedeća osoba bit će propuštena prema sobama tek kada izlazni vratar javi da je prethodna osoba izašla iz sobe u kojoj je bila. Ulazni vratar pri tome ne propušta po jednu osobu u svaku od soba: on propušta samo jednu osobu (neovisno o sobi u koju osoba želi ići) i sve ostale zadržava dok prethodno propuštena osoba ne odabere sobu, uđe u nju, odradi željeni posao te izađe iz nje. Ovakav scenarij predstavlja vizualizaciju jednog kritičnog odsječka koji se sastoji od više nizova instrukcija. Svaka će dretva izvesti samo jedan od nizova instrukcija, ali kako svi ovi nizovi rade nad nekim dijeljenim podatkovnim strukturama, želimo osigurati da u jednom trenutku samo jedna dretva može izvoditi bilo koji od štićenih nizova instrukcija.

"Vratare" iz prethodne priče u operacijskim sustavima ostvarujemo sinkronizacijskim mehanizmima poznatim pod nazivom *varijable međusobnog isključivanja* (engl. *mutual-exclusion variables*, tj. *mutexes*). Varijabla međusobnog isključivanja je objekt koji programeru nudi dvije metode: `zaključaj()` te `otključaj()`. Ako je varijabla međusobnog isključivanja otklju-

čana, dretva koja pozove metodu `zaključaj()` prebacit će varijablu međusobnog isključivanja u zaključano stanje, postati vlasnik te varijable i nastaviti dalje s izvođenjem koda. Ako pak neka druga dretva nad varijablom međusobnog isključivanja koja je zaključana pozove metodu `zaključaj()`, ta će dretva biti blokirana u toj metodi sve dok trenutni vlasnik varijable međusobnog isključivanja istu ne otključa pozivom metode `otključaj()`; u tom trenutku on gubi vlasništvo a kroz metodu `zaključaj()` pustit će se (jedna) sljedeća dretva (u slučaju da ih je više pozvalo ovu metodu, sve osim jedne i dalje će ostati zaglavljene u toj metodi). U programskom jeziku Java, bilo koji objekt (doista, primjerak bilo kojeg razreda) možemo koristiti kao varijable međusobnog isključivanja. Razred `java.lang.Thread` sadrži metodu `boolean holdsLock(Object obj)` kojom možemo ispitati je li dretva vlasnik varijable međusobnog isključivanja.

Struktura višedretveno-sigurnog koda koji uvećava brojač prikazana je u pseudokodu u nastavku. Kao varijablu međusobnog isključivanja koristimo primjerak razreda `Object` (redak 3). Na tom mjestu mogli smo umjesto toga stvoriti primjerak razreda `String`, razreda `Long` ili bilo kojeg drugog razreda -- sve bi i dalje radilo korektno.

```
 1 class Primjer {
 2   public long counter = 0L;
 3   public Object vratar = new Object();
 4
 5   public void uvecajNaSiguranNacin(int n) {
 6     // Želimo uvećati n puta
 7     for(int i = 0; i < n; i++) {
 8       vratar.zaključaj();
 9       counter++;
10       vratar.otključaj();
11     }
12   }
13
14 }
```

U ovom primjeru, kritični odsječak je dio koda koji se nalazi između poziva `vratar.zaključaj();` i poziva `vratar.otključaj();`, što je redak 9. I u ovom kodu naredba `counter++;` bit će prilikom prevođenja prevedena u četiri instrukcije. Dretva koja krene izvoditi te instrukcije može u bilo kojem trenutku biti prekinuta od strane operacijskog sustava, i neka druga dretva može dobiti procesor i krenuti s izvođenjem koda. Ali ta druga dretva sigurno ne može proći kroz metodu `vratar.zaključaj()` i ući u kritični odsječak sve dok ona prethodna dretva koja je uspjela ući ne dovrši svoj posao i ne napusti kritični odsječak. Stoga će, prije ili kasnije, operacijski sustav dodijeliti procesor onoj prvoj dretvi koja će odraditi i ostatak instrukcija kritičnog odsječka čime je eliminirana prethodno analizirana mogućnost da jedna dretva "gazi" rezultate druge dretve.



Bilješka

Kritični odječak je dio koda koji samo jedna dretva može izvoditi u jednom trenutku. Kritični odsječak ne osigurava da operacijski sustav dretvu koja je ušla u kritični odsječak neće prekinuti pa aktivirati neku drugu dretvu. Kritični odsječak osigurava da u tom slučaju ta druga dretva i dalje neće moći ući u kritični odsječak, pa se ništa lošega ne može dogoditi.

U programskom jeziku Java postoje dva načina ugrađena u sam jezik kojima možemo definirati kritični odsječak. Prvi način je uporabom bloka `synchronized`, kako je prikazano u nastavku.

```
1 synchronized(vratar) {
2   // Ovo je kritični odsječak
3   // ...
4 }
```

Svaka dretva pri nailasku na blok `synchronized` automatski pokušava zaključati vratara. Tek kada to uspije, nastavit će s izvođenjem instrukcija koje se nalaze u tom bloku. Po izlasku iz bloka `synchronized`, dretva automatski poziva metodu za otključavanje vratara.

Višedretveno-sigurna implementacija programa kod kojeg više dretvi uvećava zajedničku varijablu `counter` a koja to uvećavanje štiti kritičnim odsječkom prikazana je u nastavku.

Primjer 19.7. Čitanje i pisanje više dretvi po istoj varijabli u kritičnom odsječku

```
1 package book.java.threads;
2
3 public class DijeljenjePodataka6 {
4
5  public static long counter = 0L;
6
7  private static class Posao implements Runnable {
8   private int n;
9   private Object vratar;
10
11   public Posao(Object vratar, int n) {
12    super();
13    this.vratar = vratar;
14    this.n = n;
15   }
16
17   @Override
18   public void run() {
19    for(int i = 0; i < n; i++) {
20     synchronized(vratar) {
21      counter++;
22     }
23    }
24   }
25  }
26
27  public static void main(String[] args) {
28
29   final int brojRadnika = 2;
30   Thread[] radnici = new Thread[brojRadnika];
31
32   final Object vratar = new Object();
33   Posao posao = new Posao(vratar, 1_000_000);
34
35   for(int i = 0; i < radnici.length; i++) {
36    radnici[i] = new Thread(posao);
37   }
38
39   for(int i = 0; i < radnici.length; i++) {
40    radnici[i].start();
41   }
42
43   for(int i = 0; i < radnici.length; i++) {
44    while(true) {
45     try {
46      radnici[i].join();
47      break;
48     } catch (InterruptedException e) {
```

```
49        // Idemo opet zvati join()
50       }
51      }
52     }
53
54     synchronized(vratar) {
55      System.out.println("Konačno stanje brojača: " + counter);
56     }
57
58   }
59
60 }
61
```

Uočite, posao smo modificirali tako da dobiva referencu na objekt koji koristi kao varijablu međusobnog isključivanja. Potom je u petlji `for` kod koji povećava brojač stavljen u kritični odsječak. Metoda `main` prije stvaranja primjerka posla stvara i objekt koji će se koristiti kao varijabla međusobnog isključivanja i potom referencu na taj objekt šalje u konstruktor posla. Prepišite ovaj primjer, pokrenite ga i uvjerite se da radi. I uvjerite se da ste ovaj puta bili svjesni da program nešto radi; vjerojatno ćete primijetiti da sada programu treba zamjetno više vremena da obavi čitav posao. Taj novi utrošak vremena odlazi na vrijeme koje dretve provode u metodama `vratar.zaključaj();` i `vratar.otključaj();` a nazivamo ga *troškovi sinkronizacije*.

Pažljivijim pregledom koda iz primjera 19.7 trebali biste uočiti da smo iz deklaracije varijable `counter` uklonili kvalifikator `volatile`: ako koristimo kritične odsječke, za prevoditelj ulazak u kritični odsječak predstavlja točku u kojoj mora napustiti prethodno keširane vrijednosti varijabli i za varijable koje se spominju u kritičnom odsječku vrijednosti mora prvi puta dohvatiti nanovo iz memorije. Tada tako korištene varijable nema potrebe označavati s `volatile` (sjetite se: ideja označavanja s `volatile` bila je natjerati prevoditelj da generira instrukciju koja će iz memorije dohvatiti svježu vrijednost a ne koristiti prethodno zapamćenu u lokalnom kešu). Međutim, sada imamo novi problem: metoda `main` na kraju želi pročitati vrijednost brojača. Kako on više nije označen s `volatile`, čitanje moramo ubaciti u kritični odsječak kako bismo bili sigurni da vidimo vrijednost koju varijabla ima nakon što su sve dretve završile s radom.

Spomenuli smo da su dva načina izgradnje kritičnog odsječka. Jedna je uporaba bloka `synchronized`. Druga mogućnost je proglašavanje čitave metode kritičnim odsječkom. To se radi tako da se u potpis (signaturu) metode doda ključna riječ `synchronized` kako je napravljeno u primjeru 19.8 prikazanom u nastavku. Pogledajte metodu `uvecajZaJedan()` u razredu `Posao`.

Primjer 19.8. Čitanje i pisanje više dretvi po istoj varijabli u kritičnom odsječku (2)

```
 1 package book.java.threads;
 2
 3 public class DijeljenjePodataka7 {
 4
 5  public static long counter = 0L;
 6
 7  private static class Posao implements Runnable {
 8   private int n;
 9
10   public Posao(int n) {
11    super();
12    this.n = n;
13   }
14
15   @Override
16   public void run() {
```

```
17     for(int i = 0; i < n; i++) {
18      uvecajZaJedan();
19     }
20    }
21
22    public synchronized void uvecajZaJedan() {
23     counter++;
24    }
25   }
26
27   public static void main(String[] args) {
28
29    final int brojRadnika = 2;
30    Thread[] radnici = new Thread[brojRadnika];
31    Posao posao = new Posao(1_000_000);
32
33    for(int i = 0; i < radnici.length; i++) {
34     radnici[i] = new Thread(posao);
35    }
36
37    for(int i = 0; i < radnici.length; i++) {
38     radnici[i].start();
39    }
40
41    for(int i = 0; i < radnici.length; i++) {
42     while(true) {
43      try {
44       radnici[i].join();
45       break;
46      } catch (InterruptedException e) {
47       // Idemo opet zvati join()
48      }
49     }
50    }
51
52    synchronized (posao) {
53     System.out.println("Konačno stanje brojača: " + counter);
54    }
55   }
56
57  }
58
```

Metoda `uvecajZaJedan()` u razredu `Posao` je članska nestatička metoda razreda. Takva metoda označena ključnom riječi `synchronized` ekvivalentna je metodi koja nije označena sa `synchronized` već ima čitavu implementaciju okruženu blokom `synchronized(this) { ... }`. Drugim riječima, ako člansku nestatičku metodu proglasimo sinkroniziranom, tada je ona kritični odsječak gdje je sam objekt svoj vlastiti "vratar". Ako više članskih nestatičkih metoda označimo kao sinkronizirane, tada one sve čine jedan razdijeljeni kritični odsječak (sjetite se početnog primjera s jednim vratarom i više soba): u bilo kojem trenutku najviše jedna dretva može biti u bilo kojoj od takvih metoda. Nemoguće je da jedna dretva uspije ući u jednu metodu a druga u drugu. Uočite, međutim: s obzirom da je "vratar" sam primjerak razreda, stvorite li dva primjerka razreda, imate dva nezavisna kritična odsječka koji se međusobno ne isključuju.

Konačno, ključnu riječ `synchronized` možemo koristiti i nad statičkim metodama razreda. U tom slučaju se kao "vratar" koristi opisnik samog razreda. Primjerice, ako bi u razredu `Posao` imali statičku metodu `m` koju smo označili sa `synchronized`, kao vratar bi se koristio objekt koji bismo dobili pozivom `Posao.class`.

Imate li u istom razredu nestatičke i statičke članske metode koje ste označili sinkroniziranima, budite svjesni da one zajedno nisu isti kritični odsječak. Ulascima u statičke sinkronizirane metode upravlja jedan "vratar" a ulascima u nestatičke sinkronizirane metode primjeraka objekata sami primjerci objekata i nitko od njih nije svjestan onih drugih.

Jednostavan program u nastavku definira razred s tri metode; "obična" metoda, sinkronizirana nestatička metoda te sinkronizirana statička metoda. Glavni program poziva svaku od njih, a svaka od njih ispisuje informacije o tome da li dretva drži ključ nad kojim od vratara.

Primjer 19.9. Tko upravlja zaključavanjem?

```
 1 package book.java.threads;
 2
 3 public class TkoJeVratar {
 4
 5  public void metoda1() {
 6   ispisi("metoda1", this);
 7  }
 8
 9  public synchronized void metoda2() {
10   ispisi("metoda2", this);
11  }
12
13  public static synchronized void metoda3() {
14   ispisi("metoda3", null);
15  }
16
17  private static void ispisi(String metoda, TkoJeVratar objekt) {
18   System.out.println("Metoda "+metoda);
19   if(objekt!=null) {
20    System.out.println("Objekt je zaključan: "
21     + Thread.holdsLock(objekt));
22   }
23   System.out.println("Opisnik razreda je zaključan: "
24     + Thread.holdsLock(TkoJeVratar.class));
25   System.out.println();
26  }
27
28  public static void main(String[] args) {
29   TkoJeVratar objekt = new TkoJeVratar();
30
31   objekt.metoda1();
32   objekt.metoda2();
33   objekt.metoda3();
34
35  }
36
37 }
38
```

Program sve metode poziva iz dretve `main`. Rezultat programa bit će sljedeći.

```
Metoda metoda1
Objekt je zaključan: false
Opisnik razreda je zaključan: false

Metoda metoda2
Objekt je zaključan: true
Opisnik razreda je zaključan: false
```

```
Metoda metoda3
Opisnik razreda je zaključan: true
```



Bilješka

Umjesto eksplicitne uporabe blokova `synchronized(vratar) { ... }` u Javi je čitave metode moguće označiti sinkroniziranima. Pazite tada tko upravlja ulaskom u takvu metodu. Ako je metoda nestatička, vratar je sam objekt. Ako je metoda statička, vratar je opisnik razreda čiji je primjerak taj objekt.

Uporaba monitora

Monitorske funkcije spadaju u najopćenitiji sinkronizacijski mehanizam. Za izgradnju su nam potrebne varijable međusobnog isključivanja ali one nisu dovoljne. Monitorskoj funkciji želimo dati mogućnost da dretvu uspava ako određeni uvjeti nisu zadovoljeni (kako bismo bili u stanju izvoditi efikasna čekanja a ne radna čekanja koja troše procesor vrteći se u petlji `while`) te da jednu ili više dretvi probudi ako su prethodno bile uspavane (jer sada postoji mogućnost da uvjeti zbog kojih smo ih uspavali više ne vrijede).

Kao radni primjer razmotrit ćemo komunikaciju više proizvođača i potrošača kroz spremnik ograničene veličine. Spremnik će biti dijeljeni objekt koji ima referencu koju možemo postaviti na objekt ("proizvod") predan metodom `postavi(...)`. Spremnik će nam ponuditi i metodu `dohvati()` koja uzima zapamćeni proizvod (resetira referencu u spremniku) i vraća proizvod pozivatelju. Takva implementacija spremnika prikazana je u nastavku. Proizvode će u spremnik ubacivati dretve proizvođači a uzimati dretve potrošači. I proizvođačima i potrošačima trebat će neko vrijeme da proizvedu odnosno potroše proizvod što ćemo u kodu modelirati čekanjem slučajnog trajanja.

Primjer 19.10. Prvi model spremnika

```
 1 package book.java.threads.conprod;
 2
 3 public class Spremnik<T> {
 4
 5  private T proizvod;
 6
 7  public synchronized void postavi(T proizvod) {
 8   if(this.proizvod!=null) {
 9    System.out.println("Gubim proizvod!");
10   }
11   this.proizvod = proizvod;
12   System.out.println(
13    "Umetnut " + proizvod
14   );
15  }
16
17  public synchronized T dohvati() {
18   System.out.println(
19    Thread.currentThread().getName() + ": dohvaćen " +
20    proizvod
21   );
22   T rezultat = proizvod;
23   proizvod = null;
24   return rezultat;
25  }
26
27 }
```

Prikazani model spremnika uzima u obzir da će u isti spremnik potencijalno stavljati nove proizvode i uzimati stavljene proizvode više dretvi. Stoga, vodeći se idejom da ne smijemo dopustiti da dva potrošača istovremeno uđu u metodu `postavi(...)`, da dva potrošača istovremeno uđu u metodu `dohvati()` ili pak da proizvođač uđe u `postavi(...)` u isto vrijeme kada je potrošač u `dohvati()`, obje metode označavamo sinkroniziranima čime su i one same postale međusobno isključive (unija njihovih instrukcija čini jedan kritični odsječak). Koliko je ovaj pokušaj dobar, ispitat ćemo na primjeru. U nastavku je dan kod koji pokreće tri proizvođača i tri potrošača gdje svatko od njih proizvodi odnosno preuzima po dva proizvoda.

Primjer 19.11. Glavni program proizvođača i potrošača

```
 1 package book.java.threads.conprod;
 2
 3 public class Program {
 4
 5  static class Proizvodac implements Runnable {
 6   int brojProizvoda;
 7   Spremnik<String> spremnik;
 8
 9   public Proizvodac(int brojProizvoda, Spremnik<String> spremnik) {
10    super();
11    this.brojProizvoda = brojProizvoda;
12    this.spremnik = spremnik;
13   }
14
15   @Override
16   public void run() {
17    String ime = Thread.currentThread().getName();
18    for(int i = 1; i <= brojProizvoda; i++) {
19     cekaj();
20     spremnik.postavi(ime+" proizvod "+i);
21    }
22   }
23  }
24
25  static class Potrosac implements Runnable {
26   int brojProizvoda;
27   Spremnik<String> spremnik;
28
29   public Potrosac(int brojProizvoda, Spremnik<String> spremnik) {
30    super();
31    this.brojProizvoda = brojProizvoda;
32    this.spremnik = spremnik;
33   }
34
35   @Override
36   public void run() {
37    for(int i = 1; i <= brojProizvoda; i++) {
38     cekaj();
39     spremnik.dohvati();
40    }
41   }
42  }
43
44  private static void cekaj() {
45   try {
46    Thread.sleep((long)(Math.random()*5000));
47   } catch (InterruptedException ignorable) {
```

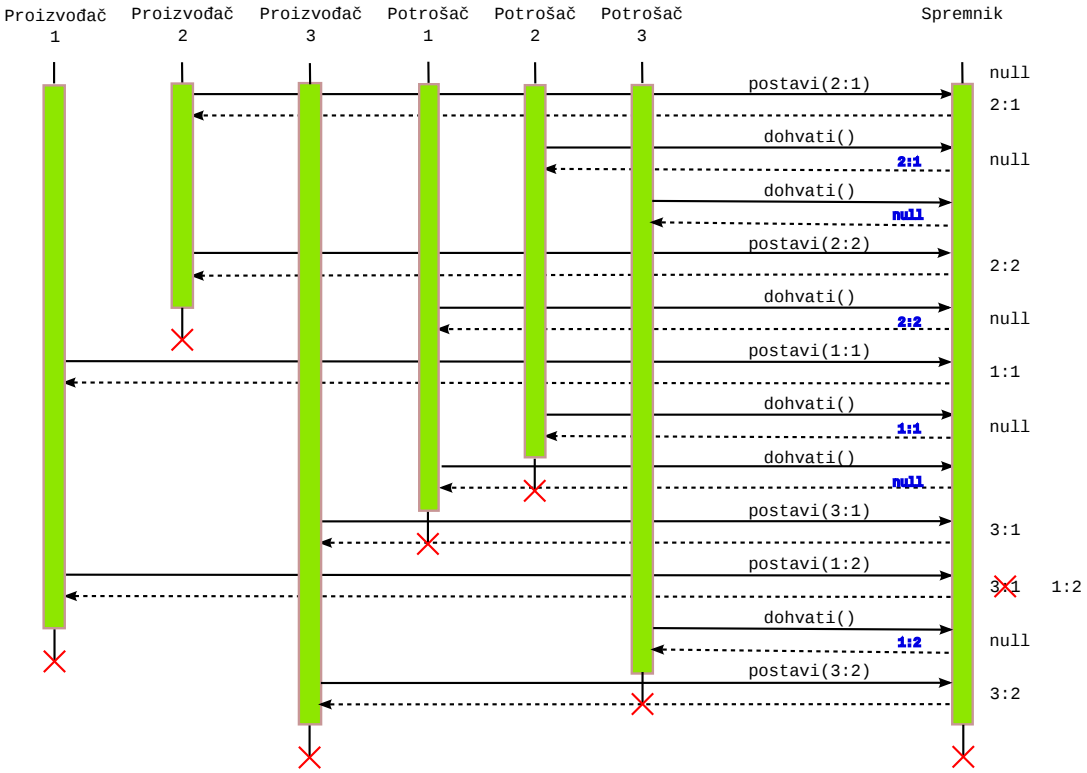
```
48    }
49   }
50
51   public static void main(String[] args) {
52    Spremnik<String> spremnik = new Spremnik<>();
53
54    for(int i = 1; i <= 3; i++) {
55     new Thread(new Proizvodac(2,spremnik), "Proizvodac"+i).start();
56     new Thread(new Potrosac(2,spremnik), "Potrosac"+i).start();
57    }
58   }
59  }
60
```

Prepišite program i pokrenite ga nekoliko puta. U nastavku je dan ispis jednog takvog izvođenja.

```
Umetnut Proizvodac2 proizvod 1
Potrosac2: dohvaćen Proizvodac2 proizvod 1
Potrosac3: dohvaćen null
Umetnut Proizvodac2 proizvod 2
Potrosac1: dohvaćen Proizvodac2 proizvod 2
Umetnut Proizvodac1 proizvod 1
Potrosac2: dohvaćen Proizvodac1 proizvod 1
Potrosac1: dohvaćen null
Umetnut Proizvodac3 proizvod 1
Gubim proizvod!
Umetnut Proizvodac1 proizvod 2
Potrosac3: dohvaćen Proizvodac1 proizvod 2
Umetnut Proizvodac3 proizvod 2
```

Sekvencijski dijagram na slici 19.5 daje nešto pregledniji prikaz dinamike izvođenja.

Slika 19.5. Sekvencijski dijagram za prikazano izvođenje programa



Iz ispisa odnosno sa slike vidimo da je proizvođač 2 u spremnik ubacio svoj proizvod 1 i taj je u sljedećem trenutku dohvatio potrošač 2 (nakon čega je spremnik ostao prazan). Potrošač 3 tada je zatražio dohvaćanje proizvoda i dobio je `null` jer je spremnik prazan. Slijede dva uspješna ciklusa umetanja proizvoda i vađenja nakon čega potrošač 1 traži dohvat proizvoda i dobiva `null` jer je u tom trenutku spremnik opet prazan. U nastavku najprije proizvođač 3 u spremnik ubacuje svoj prvi proizvod nakon čega proizvođač 1 u spremnik ubacuje svoj proizvod 2 (i time uništava proizvod proizvođača 3 koji nitko nije podigao na vrijeme). Na kraju još potrošač 3 preuzima iz spremnika proizvod 2 proizvođača 1 i time su svi potrošači napravili po dva preuzimanja pa umiru. Ostao je još samo proizvođač 3 koji umeće svoj proizvod 2 i umire.

Scenarij koji je ovdje prikazan ilustrira niz problema. Zbog neprikladne sinkronizacije proizvođača i potrošača pojavile su se sljedeće situacije.

- Proizvođač u prazan spremnik umeće svoj proizvod.
- Potrošač iz spremnika koji sadrži proizvod uzima taj proizvod.
- Proizvođač u pun spremnik umeće svoj proizvod.
- Potrošač iz praznog spremnika pokušava izvaditi proizvod.

Prva dva slučaja su slučajevi s kojima nema problema. Posljednja dva slučaja rezultiraju problemima. Što bi bilo smisleno napraviti ako proizvođač ima novi proizvod i zatraži njegovo deponiranje u spremnik koji je pun? Moguće rješenje bilo bi da malo modificiramo metodu `postavi` tako da u slučaju punog spremnika odbije preuzeti novi proizvod i to javi proizvođaču. Modificirani isječak koda prikazan je u nastavku.

```
 1 public synchronized boolean postavi(T proizvod) {
 2  if(this.proizvod!=null) {
 3   return false;
 4  }
 5  this.proizvod = proizvod;
 6  System.out.println(
 7   "Umetnut " + proizvod
 8  );
 9  return true;
10 }
```

Uz ovakvu modifikaciju proizvođač bi dobio povratnu informaciju je li njegov proizvod preuzet i ako nije, mogao bi opet zatražiti preuzimanje, pa ako opet nije, opet bi zatražio preuzimanje, i tako sve dok spremnik konačno ne preuzme proizvod. Poziv metode `postavi` u proizvođaču bi bio oblika `while(spremnik.postavi(proizvod)==false);`. Ovo je, međutim, tipičan primjer radnog čekanja. Ova dretva ne može dalje no besposleno troši procesor.

Sličnu analizu bismo mogli napraviti i na strani potrošača. Ako potrošač iz spremnika ne uspije izvaditi proizvod, mogao bi ponovno pozvati dohvat proizvoda, i ponovno, i sve dok nešto ne dobije. Opet dolazimo do radnog čekanja koje nepotrebno troši resurse računala i usporava izvođenje programa.

Rješenje opisanog problema možemo izgraditi ako nam na raspolaganju stoje dodatni sinkronizacijski primitivi: "uspavaj dretvu" te "probudi dretvu" (ili više njih). U Javi, dretvu koja je ušla u kritični odsječak možemo uspavati (uz otključavanje kritičnog odsječka kako bi druge dretve mogle ući) metodom `wait`. Ovu metodu dretva smije pozvati samo nad varijablom međusobnog isključivanja ("vratarom") čije ima vlasništvo (odnosno kojeg je zaključala). Kada se takva dretva probudi, ona će prije vraćanja iz metode `wait` nanovo zaključati varijablu međusobnog isključivanja čime će osigurati da pri nastavku njezinog izvođenja niti jedna druga dretva nije u tom kritičnom odsječku (ako je, probuđena dretva će čekati dok ne uspije zaključati varijablu). Prilikom povratka iz metode `wait` dretva mora provjeriti kakvo je sada stanje: ako i dalje ne može napraviti ono što želi, treba se ponovno uspavati i otključati varijablu međusobnog isključivanja. Ako su svi uvjeti ispunjeni pa dretva može obaviti svoj posao, ona to čini i prije napu-

štanja kritičnog odsječka sada mora signalizirati drugim uspavanim dretvama da je situacija izmijenjena i da bi se trebale probuditi i provjeriti mogu li i one dalje nastaviti s poslom.

Što to konkretno znači za naš primjer? Metodu `postavi` napisat ćemo tako da po ulasku provjerimo je li spremnik pun i ako je, dretvu proizvođača ćemo uspavati. Po buđenju, ponovit ćemo provjeru i tako sve dok ne otkrijemo da spremnik više nije pun. U tom slučaju u spremnik ćemo zapisati novi proizvod koji je proizvođač donio i prije izlaska ćemo probuditi sve uspavane dretve na spremniku.

Metodu `dohvati` napisat ćemo tako da po ulasku provjerimo je li spremnik prazan i ako je, dretvu potrošača ćemo uspavati. Po buđenju, ponovit ćemo provjeru i tako sve dok ne otkrijemo da spremnik više nije prazan. U tom slučaju iz spremnika ćemo preuzeti proizvod (i isprazniti spremnik); prije izlaska iz kritičnog odsječka još ćemo probuditi sve uspavane dretve na spremniku.

Zašto je važno da prije izlaska iz obje metode probudimo sve uspavane dretve? Ako smo proizvođač, onda smo upravo dostavili u spremnik novi proizvod. Moguće je da je prije nas došao jedan (ili više) od potrošača i pokušao (iz tada praznog spremnika) preuzeti proizvod pa se uspavao. Takve želimo probuditi jer spremnik sada više nije prazan pa jedan od njih može preuzeti proizvod i nastaviti dalje. Ako smo potrošač i ako smo upravo preuzeli proizvod, tada je nakon nas spremnik ostao prazan. Ali moguće je da je prije nas jedan ili više proizvođača pokušao u spremnik umetnuti novi proizvod i kako je spremnik već bio pun, otišao je na spavanje. Kako smo mi sada ispraznili spremnik, takve proizvođače želimo probuditi kako bi nanovo pokušali umetanje. Kada to ne bismo učinili moguće je da bi takvi proizvođači ostali trajno uspavani što svakako ne želimo.

U Javi se buđenje jedne uspavane dretve radi pozivom `notify` a buđenje svih uspavanih dretvi pozivom `notifyAll`. Ove metode smijemo pozivati samo nad varijablom međusobnog isključivanja koju smo prethodno zaključali. Razmislite zašto u ovom primjeru moramo pozivati metodu za buđenje svih uspavanih dretvi a ne samo jedne? Popravljena inačica spremnika prikazana je u nastavku.

Primjer 19.12. Višedretveno-korektno napisan spremnik

```
 1 package book.java.threads.conprod;
 2
 3 public class Spremnik<T> {
 4
 5  private T proizvod;
 6
 7  public synchronized void postavi(T proizvod)
 8   throws InterruptedException {
 9   while(this.proizvod!=null) {
10    this.wait();
11   }
12   this.proizvod = proizvod;
13   System.out.println(
14    "Umetnut " + proizvod
15   );
16   this.notifyAll();
17  }
18
19  public synchronized T dohvati() throws InterruptedException {
20   while(this.proizvod==null) {
21    this.wait();
22   }
23   System.out.println(
24    Thread.currentThread().getName() + ": dohvaćen " +
25    proizvod
```

```
26    );
27    T rezultat = proizvod;
28    proizvod = null;
29    this.notifyAll();
30    return rezultat;
31   }
32
33 }
34
```

Metoda `wait` će, u slučaju da druga dretva zatraži prekid rada uspavane dretve (metodom `interrupt`) uspavanu dretvu probuditi uz iznimku `InterruptedException` (što je provjeravana iznimka). Metode `postavi` i `dohvati` promijenili smo stoga tako da deklariraju bacanje ove iznimke a proizvođača i potrošača tako da prestanu s radom (napuste metodu `run`). Ova posljednja izmjena (zbog uštede prostora) nije nanovo prikazana; čitatelj ne bi smio u ovom trenutku imati problema s tim korekcijama.

Pokrenete li sada program nekoliko puta, svaki puta dobit ćete korektan rad. Metode `postavi` i `dohvati` pretvorili smo u *monitorske funkcije*.



Bilješka

Pojam *monitor* u terminologiji operacijskih sustava definira se kao sinkronizacijski konstrukt koji dretvama omogućava međusobno isključivanje te čekanje dok se određeni uvjeti ne ispune. Na razini ostvarenja, monitor se sastoji od varijable međusobnog isključivanja (engl. *mutex*) te uvjetne varijable (engl. *condition variable*). Varijabla međusobnog isključivanja osigurava da se u kritičnom odsječku u istom trenutku može zateći samo jedna dretva. Uvjetna varijabla nudi učinkovitu izvedbu blokiranja te naknadnog buđenja dretvi. U programskom jeziku C za ovo biste mogli koristiti primitive definirane kroz specifikaciju POSIX: `pthread_mutex_lock(...)` i `pthread_mutex_unlock(...)` za međusobno isključivanje te `pthread_cond_wait(...)`, `pthread_cond_signal(...)` i `pthread_cond_broadcast(...)` za uspavljivanje i buđenje dretvi.

Svi osnovni razredi koji su implementacije raznovrsnih kolekcija u paketu `java.util` nisu višedretveno sigurni i ne smiju se koristiti izravno u višedretvenom okruženju. Međutim, razred `java.util.Collections` nudi nekoliko metoda koje primaju referencu na kolekciju a vraćaju sinkronizacijski omotač kolekcije. Sinkronizacijski omotač je objekt koji je po sučelju jednak omotanom objektu, koji pamti referencu na omotani objekt, sve svoje metode deklarira kao sinkronizirane i jednom kada pozivatelj uspije ući u njih, dalje delegira izvođenje originalnom umotanom objektu. Pokažimo to na primjeru liste.

```
1 // Ova lista nije višedretveno sigurna
2 List<String> lista = new ArrayList<>();
3
4 // Napravimo sinkronizacijski omotač
5 List<String> sigurna = Collections.synchronizedList(lista);
6
7 // Pošaljimo ga u metodu koja stvara dretve i nešto radi s njime
8 napuniVisedretveno(sigurna);
9
```

Na sličan način možemo izgraditi i sinkronizacijske omotače za ostale kolekcije (pogledajte dokumentaciju razreda `java.util.Collections`). Posebnu pažnju treba obratiti na operaciju iteriranja u višedretvenom okruženju. Naime, sinkronizacijski omotač deklarira metodu `iterator` kao sinkroniziranu, i doista, objekt iteratora ćemo dobiti kroz kritični odsječak. Ali u trenutku kada dobijemo tu referencu, izašli smo iz kritičnog odsječka i neka druga dretva može modificirati kolekciju. Stoga se moramo pobrinuti da cjelokupni posao: dohvat iteratora i cjelo-

kupni postupak iteriranja proglasimo jednim kritičnim odsječkom. Da bismo to mogli, čitav posao moramo zaključati istom varijablom međusobnog isključivanja koju koristi i sam sinkronizacijski omotač. Pažljivim čitanjem dokumentacije (a možete i baciti oko na izvorni kod) utvrdit ćete da sinkronizacijski omotač ima metode direktno označene kao sinkronizirane, što znači da je sam omotač ujedno i "vratar". Stoga se iteriranje na višedretveno siguran način može napisati kako je prikazano u nastavku (ovaj kod nastavlja se na prethodni primjer).

```
1 // Zamotajmo čitav obilazak u kritični odsječak
2 synchronized(sigurna) {
3   Iterator<String> it = sigurna.iterator();
4   while(it.hasNext()) {
5     System.out.println("Element je: " + it.next());
6   }
7 }
```

Za neke specifične situacije uporaba sinkronizacijskih omotača oko općenitih kolekcija može rezultirati slabijom učinkovitosti koda (pad performansi). Baš za takve slučajeve, paket `java.util.concurrent` nudi implementacije nekoliko kolekcija koje jesu višedretveno sigurne ali kod kojih je sinkronizacija izvedena na drugačiji način (primjerice, `ConcurrentHashMap`, `ConcurrentLinkedDeque` i drugi).

TODO:

- implementacija razreda Mutex (možda...); uvećavanje varijable preko takvog mutexa
- SynchronousQueue<E>
- CyclicBarrier (možda...)
- Pokazati kako napraviti svoj vlastiti tread pool; reusanje dretvi; tread poisoning za gašenje
- Executors, ExecutorService, Callable, FutureTask
- Fork/Join framework

Poglavlje 20. Raspodijeljene aplikacije. Paket java.net.

Internet. Mreža svih mreža. Napraviti prolaz kroz tehnologije Jave a ne spomenuti način kako napisati jednostavnu mrežnu aplikaciju jednostavno ne bi bio pošteno odrađen posao. Internet je danas toliko sveprisutan da ga podrazumijevamo. Stolna računala spajamo na njega, prijenosna računala spajamo na njega, pametne (i manje pametne) telefone spajamo na njega. Čak i televizore spajamo na njega.

U ovom poglavlju upoznat ćemo se s osnovnim razredima i sučeljima koja daju potporu za pisanje mrežnih aplikacija: aplikacija koje su raspodijeljene i koje za komunikaciju koriste mrežu (lokalnu ili Internet). Međutim, prije no što malo dublje uronimo u ovo područje, podsjetimo se nekih osnovnih pojmova.

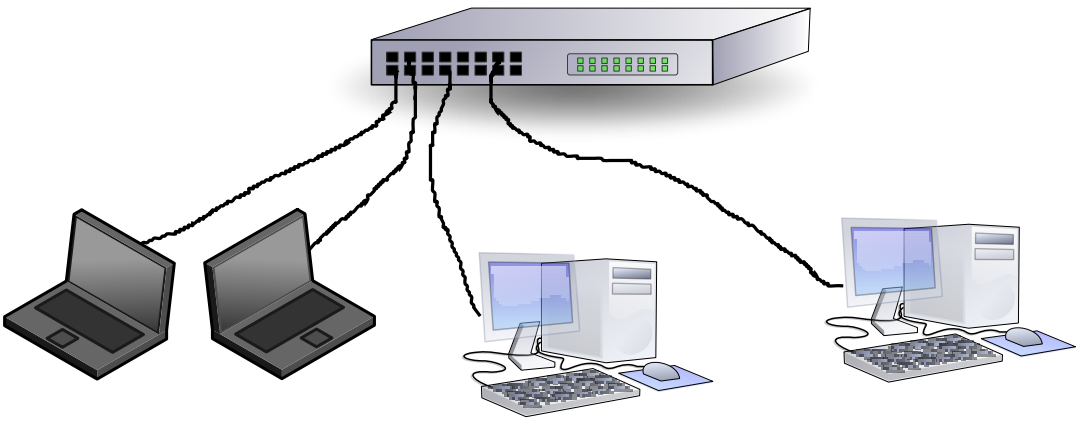
Što je Internet?

Naizgled jednostavno početničko pitanje "što je Internet" nije baš jednostavno za odgovoriti (osim ako ne želite one generičke odgovore: "mreža svih mreža" i slično). Pogledajmo stoga od čega je Internet izgrađen.

Želite li kod kuće povezati dva ili više računala u mrežu, trebat ćete jedan preklopnik (engl. *switch*) te dva ili više računala opremljena mrežnim karticama (engl. *network interface card*, NIC). Ovisno o odabranoj vrsti preklopnika (bežični ili klasični), u potonjem slučaju trebat ćete još i onoliko mrežnih kabela koliko imate računala.

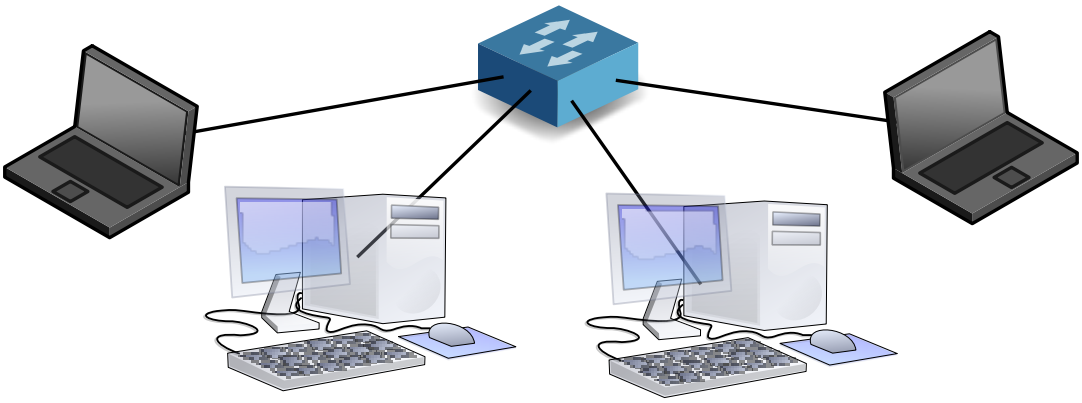
Jednom kada spojite nekoliko računala na preklopnik, izgradili ste Vašu *lokalnu mrežu*. Jedan primjer prikazan je na slici 20.1 u nastavku.

Slika 20.1. Izgradnja kućne lokalne mreže



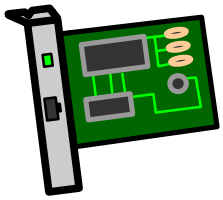
U nastavku ćemo građu mreže prikazivati jednostavnijim komponentama. Za preklopnik ćemo koristiti simbol kakav je korišten na slici 20.2.

Slika 20.2. Lokalna mreža



Da bismo povezali računalo i preklopnik, potrebna nam je mrežna kartica. U ne baš tako davnoj prošlosti, mrežne kartice kupovali smo kao diskretne komponente (slika 20.3); danas je uobičajenija situacija pronaći mrežne kartice integrirane na matičnoj ploči računala, a korisnik dodatne kartice traži tek u nekim specifičnijim situacijama. Koristite li bežičnu pristupnu točku (engl. *access point*), tada ćete dodatnu mrežnu karticu najčešće nabaviti u obliku USB-umetka.

Slika 20.3. Diskretna mrežna kartica



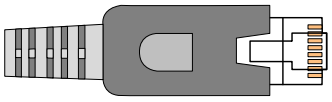
Zadaća mrežne kartice je osigurati povezivost s drugim računalom (ili mrežnim uređajem) koje se nalazi u istoj lokalnoj mreži. Način na koji se to postiže tehnološki je specifičan. Mrežne kartice mogu biti povezane bakrenim vodičem (pa razmjenjivati naponske impulse), optičkim svjetlovodom (pa razmjenjivati svjetlosne impulse), ili ničime: bežične tehnologije za ostvarivanje komunikacije koriste elektromagnetske valove (baš kao što to čine TV odašiljač i TV prijemnik). Kako bi se omogućio što transparentniji rad s mrežom (na svim razinama), osmišljen je ISO/OSI referentni model. Radi se o 7-slojnom modelu koji je prikazan na slici 20.4. Pojednostavljeno, ISO/OSI referentni model je konceptualni model koji definira interni rad i građu komunikacijskog sustava a koji se zasniva na nizu apstraktnih slojeva koji, krenuvši od najnižeg, nude sve bogatiju i bogatiju komunikacijsku uslugu.

Slika 20.4. ISO-OSI referentni model



Specifikacijama koje definiraju fizikalna svojstva medija kroz koji se obavlja komunikacija te načinom fizičkog ostvarenja uređaja bavi se *fizički sloj* ISO-OSI referentnog modela. Na tom sloju bit će definiran izgled konektora RJ45 (slika 20.5) kojim se mrežni kabel umeće u mrežnu karticu; kakva električka i mehanička svojstva treba zadovoljiti mrežni kabel; kako se točno kodiraju i dekodiraju binarni podatci u električke, svjetlosne ili elektromagnetske, i tako dalje. Razmišljamo li o fizičkom sloju kao najnižem sloju koji nudi komunikacijsku *uslugu*, onda bismo mogli reći da fizički sloj osigurava prijenos "sirovih" bitova.

Slika 20.5. RJ45 konektor



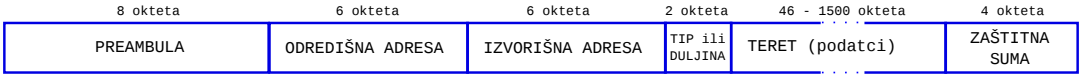
Na *podatkovnom sloju* po prvi se puta nudi usluga prijenosa *podataka* od jednog računala do drugog (ili preciznije, od jedne mrežne kartice do druge mrežne kartice koja koristi istu tehnologiju i način komuniciranja). Protokoli podatkovnog sloja mogu ponuditi mehanizme poput adresiranja i zaštite podataka od pogrešaka (ili barem detekciju pogrešaka) te kontrole toka.

Danas najrašireniji protokol za prijenos podataka na ovom sloju je protokol *Ethernet*. Ono što obično podrazumijevamo kada otiđemo u trgovinu računalne opreme i zatražimo "mrežnu karticu" je da se radi o mrežnoj kartici koja "govori" protokolom Ethernet. U današnje doba jedinu odluku koju ćete eventualno još morati donijeti jest koliko brzu karticu želite: onu koja podržava komunikaciju brzinom 100 Mbita/s ili onu koja podržava komunikaciju brzinom 1 Gbit/s. Sporije kartice danas više nema smisla kupovati (a i nije ih više niti lagano pronaći).

Ethernet za adresiranje koristi adrese duljine 48 bita. Taj adresni prostor od 2^48 adresa podijeljen je u regularne adrese, privatne adrese, adrese za difuziju u grupi (engl. *multicast*) te jednu adresu koja predstavlja difuznu adresu (engl. *broadcast*). Prilikom proizvodnje mrežnih kartica, proizvođači u svaku proizvedenu karticu tijekom proizvodnje upišu adresu koju će koristiti ta kartica. Na razini proizvođača i uz suradnju s tijelom IEEE osiguran je mehanizam kojim se garantira da će svakoj mrežnoj kartici biti upisana jedinstvena adresa. Kako se ove adrese upisuju prilikom proizvodnje (i korisnik ih obično sam ne upisuje ili mijenja), još se zovu i fizičke adrese (ili MAC-adrese). Difuzna Ethernet adresa je adresa u kojoj je svih 48 bitova postavljeno na vrijednost 1. Ispis Ethernet adresa korisnicima najčešće se radi prikazom u heksadekadskoj bazi: bitovi Ethernet adrese grupiraju se u skupine od po 8 bitova (6 je takvih grupa), i svaka se skupina prikaže kao dvoznamenkasti heksadekadski broj. Pojedine skupine odvajaju se znakom '-' (na Windowsima) odnosno znakom ':' na Linuxu. Tako će Ethernet adresa `110000000011100010010110001000101011101011100001` biti zapisana kao `c0:38:96:22:ba:e1`. Difuzna adresa bit će zapisana kao `ff:ff:ff:ff:ff:ff`.

Protokol Ethernet definira i način na koji mrežne kartice komuniciraju: mrežne kartice međusobno razmjenjuju *okvire* (engl. *frame*). Formatom Ethernet okvira definirana je interpretacija bitova koje jedna kartica šalje drugoj. Nakon fiksnog početka koji služi sinkronizaciji predajničkog i prijemničkog sklopa kartica (takozvana *preambula* duljine 64 bita) šalje se 48 bita koji predstavljaju odredišnu Ethernet-adresu odnosno identificiraju mrežnu karticu za koju je okvir namijenjen, potom novih 48 bita koji predstavljaju izvorišnu Ethernet-adresu osnosno identificiraju mrežnu karticu koja je pripremila i šalje okvir, zatim slijedi 16 bitova koji ovisno o inačici Ethernet protokola definiraju vrstu podataka koji slijede (ili njihovu duljinu; dvije su inačice Etherneta u uporabi koje razlikuju po interpretaciji ovog polja), zatim slijedi minimalno 46 a maksimalno još 1500 okteta korisničkih podataka (to je pravi teret i razlog zbog kojeg se okvir uopće šalje) nakon čega dolazi još 32 bita zaštitne sume koja je izračunata na temelju svih do tada poslanih bitova tog okvira. Ideja je jednostavna: ako se prilikom slanja bitova na fizičkom sloju dogodi pogreška pa se neki bit prenese pogrešno, zaštitna suma koju će pri prijemu okvira izračunati odredišna mrežna kartica najvjerojatnije se više neće slagati sa zaštitnom sumom koja je poslana na kraju okvira čime će prijemna kartica utvrditi da je došlo do pogreške u komunikaciji i okvir odbaciti. Grafički prikaz Ethernet okvira prikazan je na slici 20.6.

Slika 20.6. Format okvira protokola Ethernet



Sam pojam Etherneta danas podrazumijeva porodicu protokola koji svi koriste jednak format okvira koji odgovara prethodno prikazanome, a razlikuju se po brzini prijenosa bitova te načinu ostvarivanja fizičke komunikacije (žično, bežično, ...).

Kada bi ova knjiga bila o programiranju u programskom jeziku C, oboružani do sada stečenim znanjem već bismo mogli napisati jednostavan program koji razmjenjuje poruke putem lokalne mreže koristeći direktno protokol Ethernet. Taj program, nažalost, ne bismo mogli pokrenuti baš na svakom računalu. Naime, protokol Ethernet (čija se specifikacija prema do sada opisanome proteže kroz fizički i podatkovni sloj ISO-OSI referentnog modela) nije jedini postojeći protokol za povezivanje računala na podatkovnom sloju. Naš bi program radio na računalima koja koriste Ethernet i koja su fizički u istoj Ethernet-lokalnoj mreži; na ostalima ne bi. Spomenimo da različiti protokoli podatkovnog sloja definiraju različite formate okvira koji se razmjenjuju, ali i adresa koje se koriste. Na podatkovnom sloju situacija je sve samo ne unificirana.

Da bi se korisnicima omogućila razmjena poruka kroz mrežu neovisno o načinu na koji se računala fizički spajaju odnosno koji protokol koriste na podatkovnom sloju, definiran je *mrežni sloj* ISO-OSI referentnog modela. Danas su na ovom sloju kroz protokol IP ostvarene sljedeće važne zadaće.

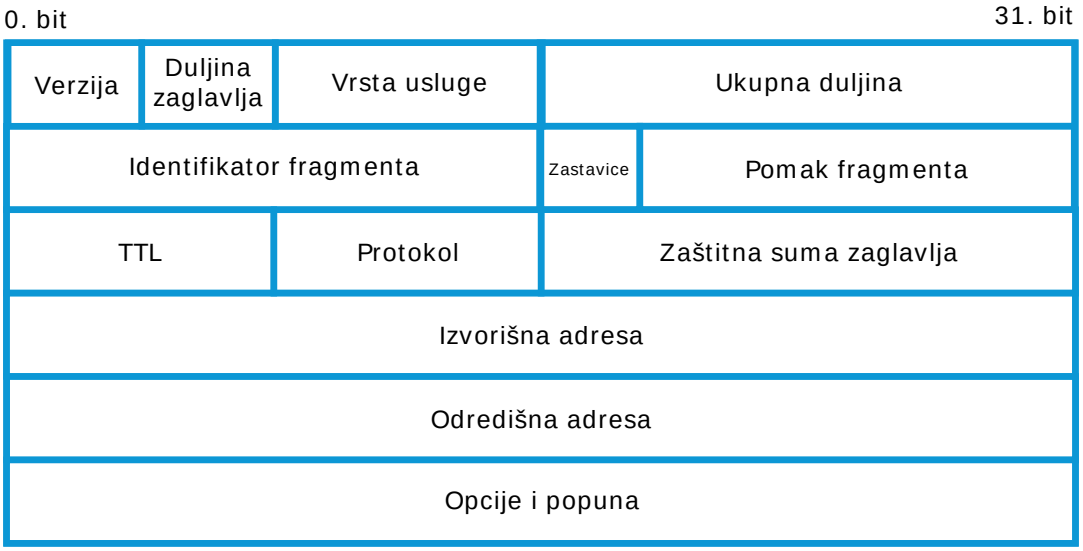
- Svim sudionicima koji "postoje" na mrežnom sloju odnosno koji mogu sudjelovati u komunikaciji dodijeljene su jedinstvene adrese unificiranog oblika (konkretno, svakoj mrežnoj kartici računala moguće je definirati jednu ili više IP adresa). Danas svaki entitet koji je vidljiv na mreži ima svoju IP adresu (ili njih više) pa na razini mrežnog sloja više nije bitno na koji način taj entitet fizički ostvaruje pristup mreži.

- Omogućena je komunikacija slanjem *paketa*. Entiteti koji komuniciraju, poruke razmjenjuju kroz IP pakete. Struktura IP paketa prikazana je na slici 20.7. U zaglavlju postoji mjesto za adresu pošiljatelja (izvorišna IP adresa), mjesto za adresu primatelja (odredišna IP adresa), mjesto koje označava vrstu tereta koju paket prenosi (polje *protocol*; preko njega se raspoznaje je li teret IP paketa UDP datagram, TCP segment, ICMP paket ili nešto četvrto) te još niz drugih informacija.

- Definiran je način kako se više računala može grupirati u mrežu. Sva računala koja se nalaze u istoj mreži imat će određen broj bitova IP adrese postavljen na istu vrijednost (taj dio IP adrese naziva se mrežni dio). Unutar iste mreže IP adrese računala razlikovat će se u bitovima koji slijede mrežni dio; taj dio IP adrese naziva se računalni dio. Definirane su difuzne IP adrese u mreži te globalna difuzna IP adresa. Definiran je i prostor privatnih IP adresa, IP adresa za difuziju u grupi te prostor IP adresa koji je rezerviran i ne smije se koristiti.

- Definiran je postupak na koji se paketi usmjeravaju kroz mrežu. Dodate li u kućnu mrežu prikazanu na slici 20.1 još i usmjernik s ugrađenim ADSL modemom kojim ćete se spojiti na Vašeg odabranog pružatelja usluge pristupa Internetu (engl. *Internet Service Provider*, ISP), preko tog usmjernika Vaša lokalna mreža povezat će se s lokalnom mrežom ISP-a. Njegova lokalna mreža povezana je pak preko nekog drugog usmjernika s lokalnom mrežom preko koje on ostvaruje pristup ostatku Interneta, i to se dalje ponavlja. Internet je u svojoj osnovi jako velik broj međusobno povezanih lokalnih mreža. Na razini protokola IP definiran je postupak kojim se određuje put (kroz koje lokalne mreže paket mora proći nakon što napusti Vašu kako bi stigao do ciljnog računala kojemu treba biti isporučen).

Slika 20.7. Format zaglavlja IP-paketa (verzije 4)



Ako ste pažljivo čitali prethodni tekst, spomenuli smo dvije vrste uređaja: preklopnike i usmjernike. Preklopnici su uređaji koji rade na podatkovnom sloju: primaju Ethernet okvire od jednog računala i isporučuju ih drugim računalima (ili svima, ako je Ethernet adresa bila difuzna). Oni su transparentni u smislu da ne postoje kao vidljivi sudionici komunikacije (preklopnik nema svoju Ethernet adresu). Usmjernici su mrežni uređaji koji rade na mrežnom sloju. Uobičajeno imaju više mrežnih kartica: na koliko su lokalnih mreža spojeni, toliko imaju mrežnih kartica. Usmjernici izoliraju promet podatkovnog sloja između lokalnih mreža. Sve što se događa u jednoj lokalnoj mreži, ostaje u toj lokalnoj mreži. Iznimka je situacija u kojoj iz jedne lokalne mreže usmjerniku podatkovnim protokolom (primjerice kao teret primljenog Ethernet okvira) stigne IP paket koji je namijenjen za drugu lokalnu mrežu. U tom slučaju, usmjernik će taj IP paket prebaciti u sljedeću lokalnu mrežu i uputiti ga dalje prema njegovom odredištu. Usmjernici su vidljivi sudionici komunikacije: oni imaju svoje vlastite IP adrese i uz to što mogu prosljeđivati dalje IP pakete, i oni sami mogu biti konačno odredište IP paketa.

Danas postoje dvije inačice protokola IP: IP inačice 4 (označavat ćemo ga s IPv4) te IP inačice 6 (koristit ćemo oznaku IPv6). IPv4 danas je još uvijek najraširenija inačica ovog protokola. Kod njega, svakom se entitetu na mrežnom sloju pridjeljuje adresa veličine 32 bita čime je teoretski moguće imati nešto manje od 4.3 milijarde entiteta. Nažalost, zbog načina kako je prostor IP adresa podijeljen (te činjenice da postoji jedan dio adresa koji je rezerviran te dio adresa koji se koristi za difuziju u grupi), za praktičnu uporabu ostaje dosta manje raspoloživih adresa. To je pak dovelo do situacije da danas imamo nestašicu IP adresa inačice 4. Upravo iz tog razloga osmišljena je novija inačica protokola: inačica 6, koja za adresiranje koristi adrese veličine 128 bita. Međutim, prelazak na IPv6 je spor i dugotrajan proces pa danas još uvijek prevladava IPv4.

Iako računala rade s binarnim zapisom IP adresa, za čovjeka je to vrlo nezgodan prikaz. Stoga se IP adrese inačice 4 prilikom prikaza čovjeku grupiraju u skupine po 8 bitova i potom se svaka skupina tumači kao cijeli broj. Zapisuje se dekadska vrijednost tog cijelog broja a grupe se odvajaju točkom. Primjerice, IP adresa inačice 4 `10100001001101010100000111100000` najprije ćemo razdijeliti u četiri grupe `10100001.00110101.01000001.11100000` i potom svaku grupu zapisati dekadski, čime dobivamo `161.53.65.224`.

IP adrese inačice 6 u ovakvom bi prikazu imale 16 dekadskih brojeva pa se koristi zapis koji ipak rezultira manjim brojem grupa i ukupno znamenaka: 128 bita adrese grupira se u 8 grupa od po 16 bitova. Svaka se grupa potom zapisuje kao 4 heksadekadske znamenke, a grupe se međusobno razdvajaju znakom dvotočke. Primjerice, adresu `00100000000000010000101101101000000000000000000000000000000000000000010011010101000101011111111111111101010001001101110010100000` bi-

smo najprije grupirali `0010000000000001.0000101101101000.0000000000000000.0000000000000000.0000001001101010.1000101011111111.1111111010100010.0110111001010000` pa zatim grupe zapisali heksadekadski `2001:0b68:0000:0000:026a:8aff:fea2:6e50` (ovo je vrlo jednostavno; čitajte četiri po četiri bita grupa i vrijednost zapišite kao jednu heksadekadsku znamenku). IP specifikacija inačice 6 dopušta da se u grupama vodeće nule ne pišu, pa bismo adresu smjeli prikazati i kao `2001:b68:0:0:26a:8aff:fea2:6e50`. Konačno, ako postoji više uzastopnih grupa u kojima su svi bitovi postavljeni na vrijednost 0, onda se *jedan takav kontinuirani slijed grupa* smije u cijelosti izostaviti, a to se mjesto označava s dvije uzastopne dvotočke. U tom slučaju, kao konačni zapis IP adrese smjeli bismo pisati `2001:b68::26a:8aff:fea2:6e50`. Ako postoji više mjesta s više uzastopnih grupa nula, ovo smijemo napraviti samo na jednom mjestu (preporuča se najljevijem).

Lokalne mreže i IPv4

Već smo spomenuli računala koja pripadaju istoj lokalnoj mreži na razini protokola IP imaju dodijeljene IP adrese koje imaju određen broj početnih bitova postavljen na jednaku kombinaciju. Taj početni dio nazivamo *mrežnim dijelom* IP adrese a njegovu veličinu određuje *mrežna maska* (što je broj). Primjerice, ako je u nekoj lokalnoj mreži mrežna maska postavljena na 24 bita, tada će sve IP adrese računala u toj lokalnoj mreži imati prvih 24 bita postavljenih na jednak način. Neki operacijski sustavi mrežne maske korisnicima ne prikazuju kao broj već u obliku IP adrese. Takav prikaz dobiva se na način da se konstruira IP adresa koja na početku ima onoliko bitova postavljenih u jedan koliko iznosi mrežna maska; ostatak se puni nulama i to se potom tumači kao IP adresa. Tako bismo mrežnu masku 24 zapisali kao `11111111111111111111111100000000` i potom pročitali: `255.255.255.0`. Takav prikaz mrežne maske korisnicima nudi operacijski sustav Microsoft Windows.

Svi bitovi nakon mrežnog dijela IP adrese čine *računalni dio* IP adrese. IP adresu u kojoj su svi bitovi računalnog dijela postavljeni na vrijednost 0 zovemo *adresom mreže*. IP adresa u kojoj su svi bitovi računalnog dijela postavljeni na vrijednost 1 također ima posebno značenje: to je lokalna difuzna adresa. Šaljemo li IP paket na tu adresu, isti će biti isporučen svim računalima koja se nalaze u toj lokalnoj mreži. IP adresa u kojoj su baš svi bitovi postavljeni na vrijednost 1 predstavlja globalnu difuznu adresu (`255.255.255.255`); teoretski, IP paket poslan na tu adresu trebao bi biti isporučen svima. U praksi, paketi poslani na tu adresu ipak neće biti propušteni prema Internetu (a ovisno o konfiguraciji usmjernika, često niti izvan lokalne mreže).

Pogledajmo sada jedan konkretan primjer. Na raspolaganju nam je lokalna mreža `161.53.65.0/24`. Koliko računala može biti u takvoj mreži? Koja je IP adresa lokalne difuzije? Kako je mrežna maska 24 bita, sve IP adrese u toj mreži moraju imati prvih 24 bita postavljenih na `101000010011010101000001` odnosno `161.53.65`. Preostaje nam samo 8 bitova za računalni dio. Kako se sve nule (IP adresa rezultira adresom mreže) i sve jedinice (IP adresa rezultira difuznom adresom) ne mogu koristiti za odresu pojedinog računala, preostaje nam još (2^8)-2 = 254 različitih računalnih dijelova, čime imamo na raspolaganju IP adrese `161.53.65.1` do `161.53.65.254`. Lokalna difuzna adresa u ovoj mreži je `161.53.65.255`.

Dohvat informacija o mrežnim karticama

Počev od Jave 1.4, u standardne biblioteke su dodana sučelja i razredi koji omogućavaju dobivanje informacija o mrežnim karticama s kojima radi operacijski sustav. Mrežna kartica modelirana je razredom `java.net.NetworkInterface`. Razred nudi nekoliko statičkih metoda čijom je uporabom moguće doći do primjeraka razreda koji predstavljaju pojedine mrežne kartice. Nudi se dohvat preko imena mrežne kartice ili pak preko IP adrese koja je postavljena na mrežnu karticu. Postoji i statička metoda `getNetworkInterfaces()` koja vraća enumeraciju (stariji rođak iteratora) svih mrežnih kartica. Jednom kada imamo objekt koji predstavlja pojedinačnu mrežnu karticu, postoje metode čijom uporabom možemo doznati informacije o samoj kartici (je li uključena ili isključena, podržava li difuziju u grupi, je li takozvano *loopback*

sučelje, koje su joj adrese pridijeljene, koliko iznosi maksimalna veličina tereta koji može prenijeti jedan okvir).

Prije no što napišemo program koji to ilustrira trebat ćemo malu pomoćnu biblioteku koju ćemo koristiti za formatirani ispis fizičkih adresa te IP adresa inačica 4 i 6. Ta je biblioteka prikazana u nastavku.

Primjer 20.1. Pomoćni razred za ispisivanje adresa

```
 1 package book.java.network;
 2
 3 import java.net.InetAddress;
 4
 5 public class Util {
 6
 7  public static String toHex(byte[] data, int group,
 8   char separator) {
 9   if(data==null) return "-";
10   if(data.length % group != 0) {
11    throw new IllegalArgumentException(
12     "Duljina adrese nije višekratnik duljine grupe.");
13   }
14   StringBuilder sb = new StringBuilder();
15   for(int i = 0; i < data.length; i++) {
16    if(i!=0 && (i%group)==0) sb.append(separator);
17    sb.append(String.format("%02X", data[i]));
18   }
19   return sb.toString();
20  }
21
22  public static String toHex(byte[] data) {
23   return toHex(data, 1, ':');
24  }
25
26  public static String toDec(byte[] data) {
27   if(data==null) return "";
28   StringBuilder sb = new StringBuilder();
29   for(int i = 0; i < data.length; i++) {
30    if(i!=0) sb.append(".");
31    sb.append(String.format("%d", (int)data[i] & 0xFF));
32   }
33   return sb.toString();
34  }
35
36  public static String addrToString(InetAddress addr) {
37   if(addr==null) return "-";
38   byte[] data = addr.getAddress();
39   if(data.length==4) return toDec(data);
40   return toHex(data, 2, ':');
41  }
42
43 }
44
```

Program u nastavku uporabom statičke metode `getNetworkInterfaces()` razreda `NetworkInterface` dohvaća popis svih mrežnih kartica te za svaku karticu ispisuje podatke.

Primjer 20.2. Ispis podataka o mrežnim karticama

```
 1 package book.java.network.nic;
 2
 3 import java.net.InetAddress;
 4 import java.net.InterfaceAddress;
 5 import java.net.NetworkInterface;
 6 import java.net.SocketException;
 7 import java.util.Enumeration;
 8
 9 import book.java.network.Util;
10
11 public class ListNetworkInterfaces {
12
13  public static void main(String[] args)
14   throws SocketException {
15   Enumeration<NetworkInterface> nics =
16    NetworkInterface.getNetworkInterfaces();
17   while(nics.hasMoreElements()) {
18    NetworkInterface nic = nics.nextElement();
19    showInfo(nic);
20    System.out.println();
21   }
22  }
23
24  private static void showInfo(NetworkInterface nic)
25   throws SocketException {
26   System.out.printf("Name: %s%n", nic.getName());
27   System.out.printf("Index: %d%n", nic.getIndex());
28   System.out.printf("Description: %s%n",
29    nic.getDisplayName());
30   System.out.printf("MTU: %d%n", nic.getMTU());
31   System.out.printf("Loopback: %b%n", nic.isLoopback());
32   System.out.printf("PointToPoint: %b%n",
33    nic.isPointToPoint());
34   System.out.printf("Up: %b%n", nic.isUp());
35   System.out.printf("Supports multicast: %b%n",
36    nic.supportsMulticast());
37   System.out.printf("Virtual: %b%n", nic.isVirtual());
38   System.out.printf("Hardware address: %s%n",
39    Util.toHex(nic.getHardwareAddress()));
40   Enumeration<InetAddress> addrs = nic.getInetAddresses();
41   while(addrs.hasMoreElements()) {
42    InetAddress addr = addrs.nextElement();
43    System.out.printf("  Internet address: %s%n",
44     Util.addrToString(addr));
45   }
46   System.out.println("  ----");
47   for(InterfaceAddress addr : nic.getInterfaceAddresses()) {
48    System.out.printf("  Internet address: %s/%d brd %s%n",
49     Util.addrToString(addr.getAddress()),
50     addr.getNetworkPrefixLength(),
51     Util.addrToString(addr.getBroadcast())
52    );
53   }
54  }
55 }
56
```

Skraćeni ispis koji program generira na mojem računalu prikazan je u nastavku, pa ga možemo prokomentirati.

```
Name: lo
Index: 1
Description: Software Loopback Interface 1
MTU: -1
Loopback: true
PointToPoint: false
Up: true
Supports multicast: true
Virtual: false
Hardware address: -
  Internet address: 127.0.0.1
  Internet address: 0000:0000:0000:0000:0000:0000:0000:0001
  ----
  Internet address: 127.0.0.1/8 brd 127.255.255.255
  Internet address: 0000:0000:0000:0000:0000:0000:0000:0001/128 brd -

Name: eth4
Index: 12
Description: Marvell Yukon 88E8072 PCI-E Gigabit Ethernet Controller
MTU: 1500
Loopback: false
PointToPoint: false
Up: true
Supports multicast: true
Virtual: false
Hardware address: 00:25:B3:64:ED:F0
  Internet address: 192.168.201.7
  Internet address: 2001:0B68:0016:0080:4926:01EB:F47B:9506
  Internet address: 2001:0B68:0016:0080:5DE0:D745:E6BC:8716
  Internet address: FE80:0000:0000:0000:4926:01EB:F47B:9506
  ----
  Internet address: 192.168.201.7/24 brd 192.168.201.255
  Internet address: 2001:0B68:0016:0080:4926:01EB:F47B:9506/64 brd -
  Internet address: 2001:0B68:0016:0080:5DE0:D745:E6BC:8716/128 brd -
  Internet address: FE80:0000:0000:0000:4926:01EB:F47B:9506/128 brd -

Name: wlan0
Index: 13
Description: Broadcom 802.11b/g WLAN
MTU: 1500
Loopback: false
PointToPoint: false
Up: false
Supports multicast: true
Virtual: false
Hardware address: 00:26:5E:3B:88:D9
  Internet address: FE80:0000:0000:0000:DDB2:CD1B:E1FD:ABC5
  ----
  Internet address: FE80:0000:0000:0000:DDB2:CD1B:E1FD:ABC5/64 brd -
```

Postoji kartica imena `lo`. Radi se o takozvanom *loopback* sučelju. To nije stvarna mrežna kartica već mrežna kartica koju operacijski sustav emulira kako bi omogućio da dva računalna programa koja su na istom računalu također mogu komunicirati preko mreže. Ideja je vrlo jednostavna: umjesto da se takav paket isporuči kroz fizičku mrežnu karticu nekome izvan raču-

nala, paket će biti poslan na *loopback* karticu koja će odmah operacijskom sustavu dojaviti da je s mreže baš primila taj paket. Ovu karticu operacijski sustav tretira kao i svaku drugu: kada dobije dojavu da je primljen IP paket, pogledat će za koga je, utvrditi da je za drugi program i njemu ga isporučiti. Na ovaj način programima više nije bitno gdje se nalazi drugi program s kojim se komunicira: da li doista na nekom drugom računalu ili na istom; način komunikacije je identičan.

Mrežnoj kartici `lo` dodijeljena je IPv4 adresa `127.0.0.1`. Radi se o privatnoj adresi koja se ne smije pojaviti nigdje na Internetu. Za svako računalo, adresa `127.0.0.1` predstavlja upravo njega samoga. Zamislite li mrežne kartice kao Vaše kolege koji razgovaraju, onda bi konkretne javne IP adrese predstavljale te osobe njihovim imenom i prezimenom dok bi adresa `127.0.0.1` odgovarala zamjenici "ja" - za svakoga tko to kaže, "ja" znači nešto drugo - znači njega samoga. U tom smislu, ova adresa nije jedinstvena na Internetu i paketi s takvom adresom ne bi smjeli otići s našeg računala u mrežu. Ista mrežna kartica ima dodijeljenu i IPv6 adresu `::1/128`; ta adresa za IPv6 ima isto značenje kao adresa `127.0.0.1` za IPv4.

Mrežna kartica `lo` nema definiran MTU (engl. *maximum transmission unit*) odnosno maksimalnu veličinu podataka koje kartica može prenijeti; kako se radi o emuliranoj kartici, kartica ne postavlja nikakva ograničenja jer se prijenos obavlja isključivo kroz radnu memoriju računala. Zastavica "Up" koja je postavljena na `true` znači da je mrežna kartica aktivirana i da omogućava slanje i primanje okvira. Ova kartica nema fizičku adresu.

Posljednje što je prikazano za karticu su dodijeljene IP adrese. Prva dva retka u ispisu rezultat su uporabe metode `getInetAddresses()` koja vraća enumeraciju svih IP adresa koje su pridijeljene promatranoj mrežnoj kartici. Ova metoda deklarira da su objekti enumeracije koja se vraća primjerci razreda `InetAddress`. Iz tog razreda za svaku su od inačica protokola IP izvedeni podrazredi koji predstavljaju vrstu adresa koju inačica koristi. Tako postoje razredi `Inet4Address` (model IP adrese inačice 4) te `Inet6Address` (model IP adrese inačice 6). Objekti koje vrati metoda `getInetAddresses()` bit će primjerci upravo tih razreda, pa ako ih je potrebno razlikovati, to je moguće ispitati uporabom operatora `instanceof`.

Sam razred `InetAddress` deklarira nekolicinu metoda kojima je moguće dobiti više informacija o samoj adresi. Mi u primjeru koristimo samo jednu: `getAddress()` koja vraća IP adresu kao polje okteta. Ako se radi o IP adresi inačice 4, dobit ćemo polje veličine 4 okteta (4 puta 8 = 32 bita), a ako se radi o IP adresi inačice 6, dobit ćemo polje veličine 16 okteta (16 puta 8 = 128 bita).

Kako metoda `NetworkInterface.getInetAddresses()` vraća samo enumeraciju adresa, postoji i metoda `NetworkInterface.getInterfaceAddresses()` koja vraća listu objekata tipa `InterfaceAddress`. Svaki objekt ovog tipa sadrži informaciju o pridjeljenoj IP adresi, mrežnoj maski te difuznoj adresi. Stoga je u programu dan prikaz i te informacije (posljednja dva retka kod mrežne kartice `lo`).

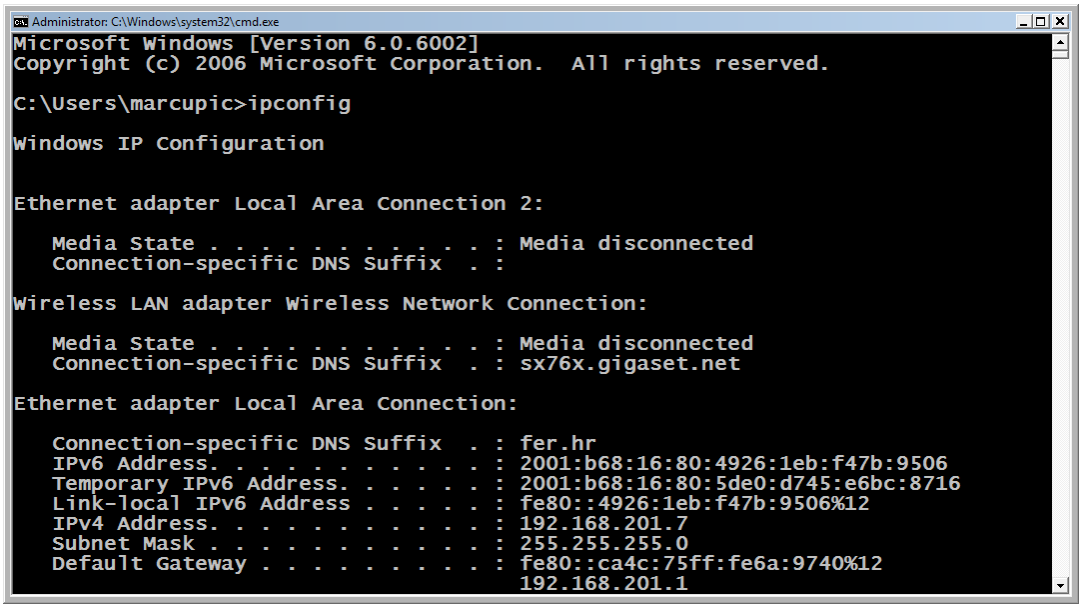
Mrežna kartica `eth4` predstavlja fizičku "žičanu" mrežnu karticu koja koristi protokol Ethernet i preko koje računalo ostvaruje pristup Internetu. Ova mrežna kartica ima ograničenu veličinu podataka koje može prenijeti jednim okvirom: radi se o 1500 okteta (što je ograničenje na veličinu polja za prijenos podataka kod okvira protokola Ethernet - pogledajte još jednom sliku 20.6). Kartica ima svoju fizičku adresu a dodijeljene su joj i četiri IP adrese (jedna verzije 4 i tri verzije 6). IP adresa verzije 4 koja je dodjeljena ovoj mrežnoj kartici odabrana pripada privatnim IP adresama. Rasponi adresa `10.0.0.0/8` te `192.168.0.0/16` dva su primjera privatnih IP adresa. Za razliku od IP adresa iz raspona `127.0.0.0/8` koje znače "ja, ja i samo ja", spomenute privatne IP adrese mogu se koristiti za komunikaciju u lokalnoj mreži. Međutim, IP paketi s takvim IP adresama ne smiju pobjeći iz lokalne mreže na javni Internet. Privatne adrese nisu javne niti su globalno jedinstvene. Svaka tvrtka može u svojoj lokalnoj mreži koristiti za računala takve IP adrese tako dugo dok je dovoljno omogućiti komunikaciju između računala u lokalnoj mreži i ništa više od toga.

Konačno, mrežna kartica `wlan0` je bežična mrežna katrica koja također koristi protokol Ethernet. Stoga kao fizičku adresu ima postavljenu Ethernet adresu koju je proizvođač kartice pos-

tavio prilikom proizvodnje. Ovoj mrežnoj kartici postavljena je i IP adresa inačice 6. Treba uočiti da je mrežna kartica isključena odnosno niti prima niti se može koristiti za slanje Ethernet okvira.

Svi operacijski sustavi koji podržavaju rad s mrežom nude i skup alata kojima je moguće dobiti informacije o trenutno važećim postavkama mrežnih kartica. Na operacijskom sustavu Microsoft Windows, u tu svrhu možete iskoristiti naredbu **ipconfig**. Primjer je prikazan na slici u nastavku.

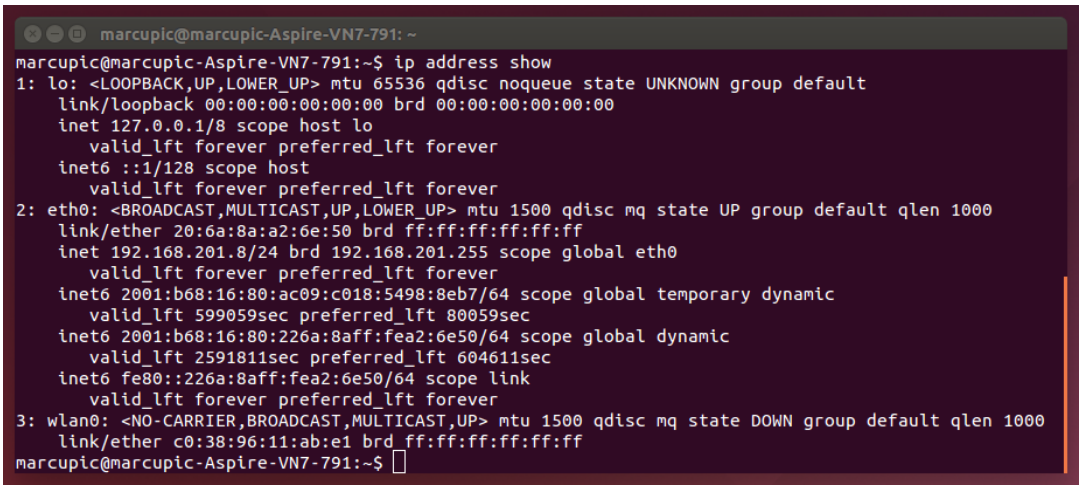
Slika 20.8. Ispis parametara mrežnih kartica na Windowsima



Usporedite ispis koji je izgeneriran za karticu pod nazivom "Ethernet adapter Local Area Connection" (posebice broj same IP adrese): to je isto računalo na kojem je bio pokrenut prethodni program napisan u Javi za koji je na prethodnim stranicama uključen ispis; tamo je ta kartica bila prikazana pod imenom `eth4` i s rednim brojem 12. Pokrenete li ovu naredbu uz dodatni prekidač: **ipconfig** /**all**, dobit ćete u ispisu detaljnije informacije, poput fizičkih adresa kartica. Isprobajte.

Na operacijskom sustavu Linux, naredba koju možete iskoristiti je **ip address show**. Primjer izvođenja te naredbe prikazan je na slici u nastavku.

Slika 20.9. Ispis parametara mrežnih kartica na Linuxu



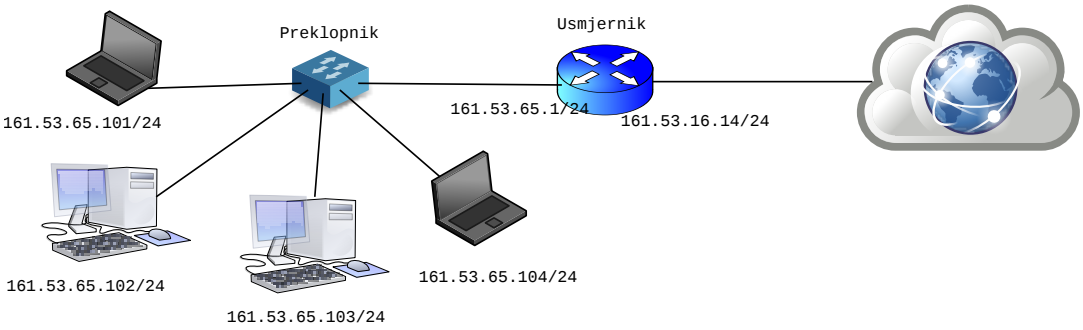
Na prikazanom ispisu možemo uočiti da računalo raspolaže s tri mrežne kartice. Jedna je emulirana (`lo`), druga je kartica koja omogućava uporabu "žičnog" Etherneta (`eth0`) a treća omo-

gućava uporabu bežičnog Etherneta (`wlan0`). Prve dvije kartice su aktivne (uočite zastavicu UP) odmah u retku uz ime kartice. Kartica `lo` ima očekivanu IP adresu `127.0.0.1` odnosno njezin IPv6 pandan `::1/128`. Kartica `eth0` je kartica preko koje računalo ostvaruje pristup na Internet i ima IPv4 adresu 192.168.201.8/24.

Dva primjera lokalne mreže

Nakon laganog teorijskog uvoda, pogledajmo dvije tipične konfiguracije lokalne mreže. Prva je prikazana na slici 20.10.

Slika 20.10. Lokalna mreža s javnim IP adresama

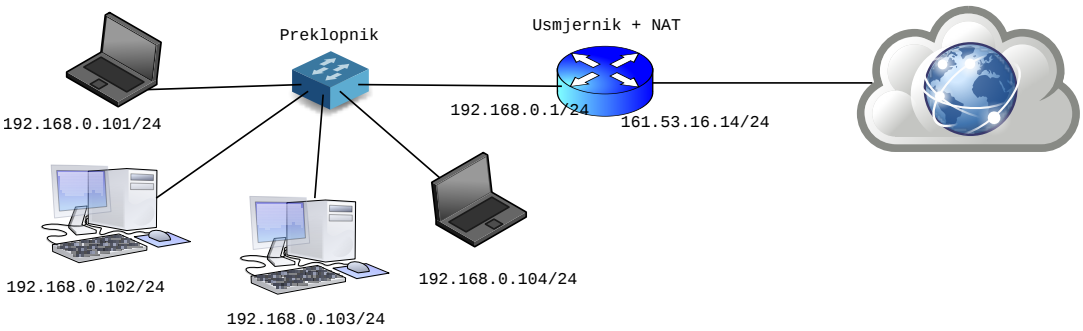


U prikazanoj lokalnoj mreži sva računala imaju javne IP adrese, odnosno sva su računala direktno vidljiva s Interneta. Mreža je temeljena na tehnologiji Ethernet; sva računala spojena su na zajednički preklopnik. Na mrežnom sloju lokalna mreža koristi adresu `161.53.65.0/24` pa se IP adrese pojedinih računala razlikuju samo u posljednjem oktetu. Kako bi mreža ostvarila izlaz na Internet, u mrežu je dodan usmjernik (u općem smislu računalo s dvije mrežne kartice) koji u našoj lokalnoj mreži ima IP adresu `161.53.65.1/24` a koji je s druge strane (preko druge mrežne kartice) spojen u lokalnu mrežu kroz koju je dostupan Internet. Prikazana računala, kada komuniciraju međusobno, IP pakete pakiraju u Ethernet okvire koji putuju od računala do preklopnika pa do ciljnog računala; usmjernik takve pakete ne vidi. IP paketi koje računala generiraju za odredište koje je na Internetu (primjerice, Google) također se pakiraju u Ethernet okvire ali ti će okviri kao adresu odredišta sadržavati fizičku adresu mrežne kartice kojom je usmjernik spojen u našu lokalnu mrežu, pa će on dobiti te IP pakete. Po njihovom primitku on će ih dalje proslijediti kroz svoju lokalnu mrežu prema Internetu.

S obzirom da prikazana računala imaju javne IP adrese, bilo tko s Interneta također može generirati i slati pakete koji su namijenjeni prikazanim računalima. Taj promet može biti koristan (primjerice, ako se na nekom od računala nalazi web-poslužitelj, tada želimo da on bude vidljiv na Internetu i da mu klijenti - web-preglednici - mogu slati upite), ali može biti i zlonamjeran (maliciozni korisnici mogu pokušavati preuzeti kontrolu nad Vašim računalom).

Slika 20.11 prikazuje nešto drugačiji scenarij koji je tipičniji za kućne korisnike koji pristup Internetu ostvaruju preko nekog ISP-a (ili za studente koji se nalaze u nekom od studentskih domova).

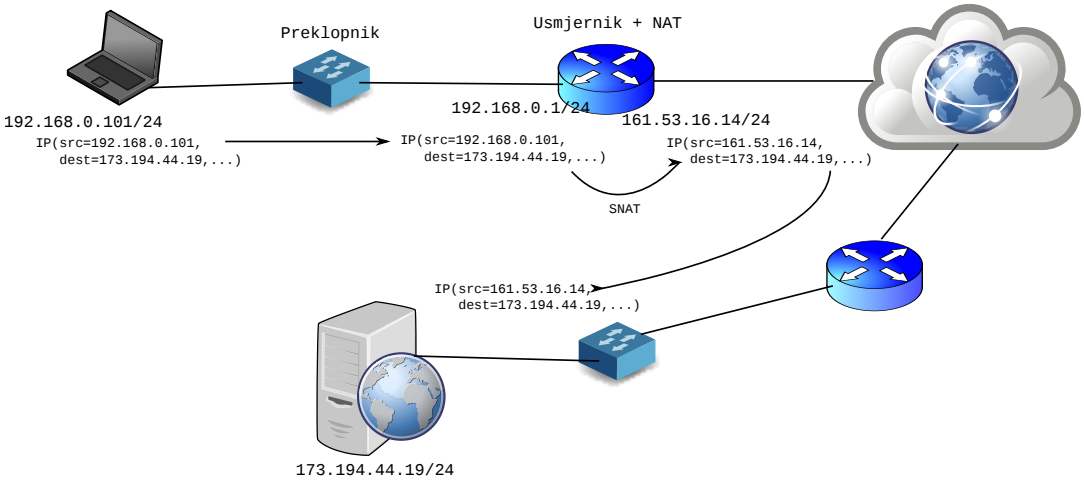
Slika 20.11. Lokalna mreža s privatnim IP adresama



U prikazanom scenariju lokalnoj je mreži dana privatna adresa: `192.168.0.0/24`. Stoga sve IP adrese pojedinih računala započinju s `192.168.0.` i razlikuju se samo u posljednjem oktetu. Sva računala u lokalnoj mreži mogu nesmetano komunicirati, baš kao što je to bio slučaj i s prethodnim scenarijem. Međutim, bez dodatne pomoći usmjernika, računala u prikazanoj lokalnoj mreži ne bi mogla koristiti Internet. Naime, kako su računalima dodijeljene privatne IP adrese, IP paketi s takvim adresama ne smiju se proslijediti u javnu mrežu (a tamo nemaju niti jednoznačnog pošiljatelja: svatko za sebe može oformiti svoju privatnu mrežu i u njoj koristiti privatne IP adrese pa se ne bi znalo kome vratiti odgovor na takav IP paket).

Da bismo sada omogućili računalima iz lokalne mreže uporabu Interneta, usmjernik prije no što propusti IP paket iz lokalne mreže prema Internetu mora izvorišnu IP adresu koja je zapisana u IP paketu pregaziti svojom *javnom IP adresom* preko koje je vidljiv na Internetu (i mora zapamtiti da je to napravio). Pogledajmo to na konkretnom primjeru prikazanom na slici 20.12, gdje računalo s IP adresom `192.168.0.101` stvara i šalje IP paket za računalo na Internetu čija je IP adresa `173.194.44.19`. Usmjernik prima taj IP paket i modificira u njemu izvorišnu adresu. Tako modificirani IP paket odlazi do krajnjeg odredišta na Internetu, gdje analizom izvorišne IP adrese to ciljno računalo zaključuje da je izvor paketa sam usmjernik; odredišno računalo ne zna da je čitava lokalna mreža "skrivena" iza tog usmjernika. Kada udaljeno računalo izgenerira odgovor, poslat će ga u IP paketu koji će kao svoje odredište imati javnu IP adresu usmjernika (u našem slučaju `161.53.65.14`). Po primitku tog IP paketa usmjernik mora raspolagati s dovoljno informacija da može zaključiti kako taj IP paket zapravo nije za njega, već da je odgovor na IP paket koji je usmjernik maloprije proslijedio prema Internetu (nakon napravljene modifikacije). Tada će usmjernik zamijeniti odredišnu IP adresu tog novopristiglog IP paketa (koja je trenutno njegova javna) s privatnom IP adresom računala koje je poslalo prvi IP paket (`192.168.0.101`), i tako izmijenjeni IP paket proslijedit će izvornom računalu. Tehnologija transparentnog manipuliranja IP adresama koju smo ovdje opisali poznata je pod nazivom NAT (engl. *Network Address Translation*). Modificiranje privatne *izvorišne* IP adrese u IP paketu s upitom te automatsko modificiranje IP adrese u paketu s odgovorom naziva se SNAT (engl. *Source Network Address Translation*). Ako se na izlazu iz lokalne mreže nalazi usmjernik koji obavlja i SNAT, tada je za ostatak Interneta čitava lokalna mreža skrivena iza tog uređaja i vidljiva kao jedno računalo s javnom IP adresom usmjernika. Ako je lokalna mreža velika, ovo ponekad može dovesti do pogrešne identifikacije Vašeg usmjernika kao zlonamjernog korisnika koji generira veliku količinu upita i time pokušava preopteretiti udaljeni poslužitelj.

Slika 20.12. Pojašnjenje rada tehnologije SNAT



Ovakav scenarij izrade lokalne mreže nezgodan je ako želite u lokalnoj mreži pokrenuti bilo kakav mrežni poslužitelj (primjerice web-poslužitelj) koji bi trebao biti dostupan korisnicima s Interneta: bez dodatne konfiguracije tako nešto neće raditi. Problem je, naime, sljedeći. Kada korisnik iz lokalne mreže inicira upit prema Internetu, usmjernik koji radi NAT to zapamti, pa kada stigne odgovor, zna kako ga modificirati i kome isporučiti. S druge pak strane, korisnik koji se nalazi na Internetu svoj upit niti ne može adresirati Vašem mrežnom poslužitelju jer taj

ima samo privatnu IP adresu; umjesto toga, najbolje što može napraviti je upit adresirati na Vaš usmjernik (na njegovu javnu IP adresu). Međutim, kada taj upit stigne na usmjernik, usmjernik će ga odbaciti jer ne zna što bi radio s njime (on nije mrežni poslužitelj). Da bi se riješio taj problem, usmjernik je potrebno dodatno podesiti tako da mu se naloži da radi DNAT (engl. *Destination Network Address Translation*). Ideja je sljedeća: na usmjerniku treba podesiti da se svi upiti koji dolaze s Interneta za Vaš lokalni web-poslužitelj modificiraju tako da se u pristiglom IP paketu upita odredišna IP adresa zamijeni privatnom IP adresom Vašeg web-poslužitelja, i potom se tako modificirani IP paket propusti u lokalnu mrežu do samog poslužitelja. Kada poslužitelj izgenerira odgovor i pošalje IP paket odgovora (koji će tada kao izvorišnu IP adresu imati privatnu IP adresu poslužitelja), usmjernik će u tom paketu prije no što ga propusti na Internet privatnu izvorišnu IP adresu zamijeniti svojom javnom. Na taj način korisnik s Interneta će imati dojam kao da se mrežni poslužitelj nalazi na samom usmjerniku: činjenica da je upit proslijeđen nekome u lokalnu mrežu te da je odgovor naknadno modificiran za takvog korisnika ostaje skrivena.

Prilikom izrade mrežnih poslužitelja koje ćemo opisati u nastavku, budete li htjeli isprobati mogu li se Vaš kolega ili kolegica s javnog Interneta spojiti na poslužitelj koji ste napravili, pripazite na koji način ostvarujete pristup Internetu. Kućni korisnici koji se spajaju preko nekog od ISP-ova uobičajeno će na korisničkoj strani dobivati privatne IP adrese za računala koja povezuju na dobiveni ADSL model/usmjernik. Te će adrese najčešće biti iz mreže `192.168.0.0/24`, `192.168.1.0/24` ili `10.0.0.0/24`. U tom slučaju usmjernik preko kojega se ostvaruje izlaz na Internet u lokalnoj će mreži biti vidljiv preko IP adrese `192.168.0.1`, `192.168.1.1` odnosno `10.0.0.1`. Morat ćete se spojiti na sam usmjernik i na njemu podesiti takozvani *port forwarding*, što se obično može napraviti preko web-sučelja koje nudi usmjernik za potrebe administracije. Ovisno kakav ste točno mrežni poslužitelj napravili (koristi li se UDP ili TCP te na kojem portu sluša poslužitelj), dodat ćete pravilo oblika: *ako je stigao UDP (ili TCP) paket za port X, proslijedi ga meni na privatnu IP adresu Y na port Z*. Vanjskom korisniku morat ćete dojaviti koju javnu IP adresu u tom trenutku ima Vaš usmjernik i na koji mu port morate slati upit (*X* u prethodnom pravilu); program koji predstavlja mrežni poslužitelj pokrenut ćete na Vašem računalu koje u lokalnoj mreži ima privatnu IP adresu *Y* i to na portu *Z*. Ovo će postati jasnije kada opišemo protokole UDP i TCP. Ostaje samo još jedno pitanje: kako u ovakvom slučaju doći do javne IP adrese koju ima Vaš usmjernik u tom trenutku? Najjednostavniji odgovor: spojite se iz web-preglednika na bilo koju web-stranicu koja ispisuje IP adresu računala s koje je stigao upit; primjer takve stranice je http://whatismyipaddress.com/, i pročitajte što je ispisano; alternativno, ukucajte u Google "what is my ip".

Protokol DNS

Kroz prethodni dio ovog poglavlja upoznali smo se s pojmom IP adrese. Računala brzo i efikasno rade s brojevima. Ljudi, nažalost, ne. Praksa je pokazala da ljudi puno jednostavnije pamte riječi i simbole u odnosu na nizove brojeva. Evo odmah ilustrativnog primjera: koliko ste puta do sada koristili tražilicu Google? Znate li njezino simboličko ime? Vjerojatno da: `www.google.com`. A znate li njezinu IP adresu? Vjerojatno ne.

Protokol DNS[1] (engl. *Domain Name System*) osmišljen je kako bi se ljudima omogućilo da barataju sa simboličkim imenima a računalima s brojevima. Iako sam protokol rješava nekoliko problema, temeljna funkcija sustava DNS je razrješavanja upita oblika: *"koja je IP adresa računala čije je simboličko ime xxx?"*.

Na većini operacijskih sustava za razrješavanje ovakvih upita imamo na raspolaganju naredbu **nslookup**. Sustav DNS možemo zamisliti kao veliku raspodijeljenu bazu podataka koja čuva mnoštvo definiranih vrsta zapisa. Zapisi tipa A definiraju u koje se sve IP adrese preslikava konkretno simboličko ime. Da bismo doznali koja je IP adresa pridružena simboličkom imenu `www.google.com`, potražit ćemo sve A-zapise za to simboličko ime. Konkretno, zadat ćemo naredbu **nslookup -type=A www.google.com**. Rezultat je prikazan na slici u nastavku.

[1] Opis sustava DNS: https://www.ietf.org/rfc/rfc1034.txt, implementacijska pitanja: https://www.ietf.org/rfc/rfc1035.txt

Slika 20.13. Naredba nslookup

```
marcupic@marcupic-Aspire-VN7-791: ~
marcupic@marcupic-Aspire-VN7-791:~$ nslookup -type=A www.google.com
Server:         127.0.1.1
Address:        127.0.1.1#53

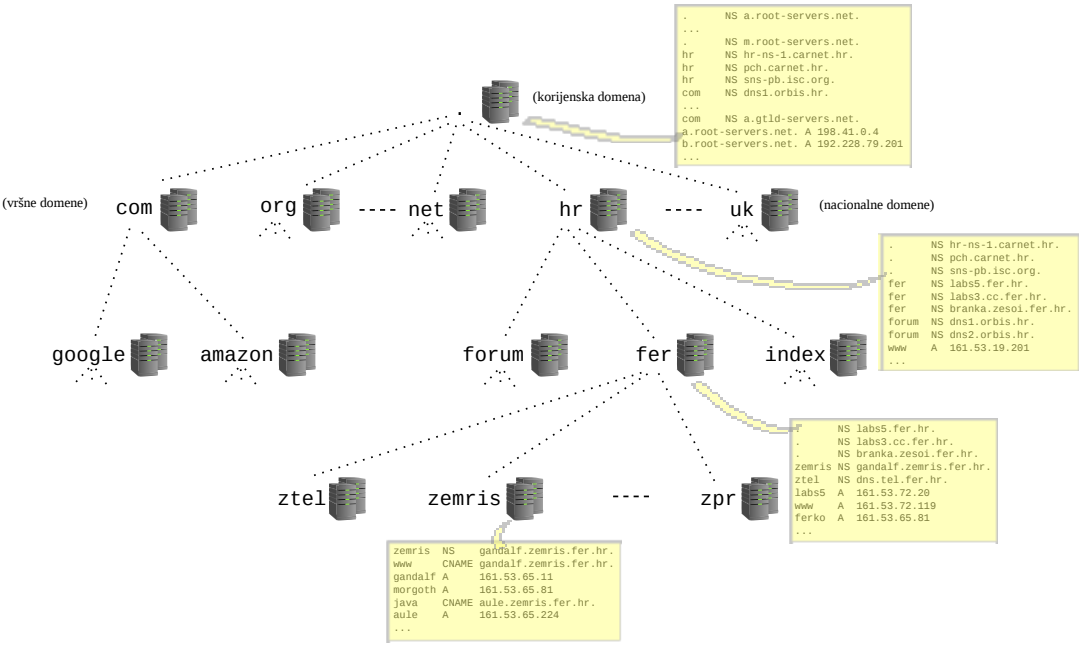
Non-authoritative answer:
Name:   www.google.com
Address: 173.194.44.49
Name:   www.google.com
Address: 173.194.44.50
Name:   www.google.com
Address: 173.194.44.48
Name:   www.google.com
Address: 173.194.44.52
Name:   www.google.com
Address: 173.194.44.51

marcupic@marcupic-Aspire-VN7-791:~$
```

Simboličkom imenu `www.google.com` pridruženo je čak 5 IP adresa. Ponavljanjem upita možete uočiti još jedan zgodan detalj: redosljed IP adresa u odgovoru se mijenja. Kako programi za potrebe komunikacije s udaljenim računalom trebaju njegovu (jednu) IP adresu, običnu uzmu prvu koju dobiju pa se na ovaj način uporabom sustava DNS na jednostavan način ostvaruje raspodjela opterećenja; iza imena `www.google.com` krije se 5 IP adresa, a iza svake jedno računalo (ili čitava nova lokalna mreža poslužitelja) i u prosjeku svaka od tih 5 adresa dobiva petinu ukupnog opterećenja.

Ponešto detaljnija organizacija sustava DNS, iz koje je vidljiva hijerarhijska organizacija, prikazana je na slici 20.14.

Slika 20.14. Organizacija DNS sustava



Imena unutar sustava DNS organizirana su hijerarhijski u domene i poddomene. *Pojednostavljeno*, svaka domena i poddomena ima svoj vlastiti DNS poslužitelj koji čuva lokalne informacije te domene. Na vrhu svega nalazi se korijenska domena ("."). Lokalne informacije o toj domeni čuva 13 DNS poslužitelja čija su imena i IP adrese javno poznata (broj fizičkih DNS poslužitelja je veći kako bi se osigurala redundantnost, što se implementacijski izvodi različitim tehnikama, između ostaloga *anycast*-om na razini mreže). Na prikazanoj slici, ti su poslužitelji prikazani na vrhu slike uz oznaku korijenske domene ("."). Isječak DNS zapisa koje čuvaju ta računala također je prikazan u žutom okviru. Zapis tipa NS definira da je za domenu navedenu s lijeve strane zapisa (ako je tamo samo točka, nju tumačimo kao "trenutna domena" što je u ovom

slučaju korijenska domena) zadužen DNS poslužitelj čije je simboličko ime navedeno s desne strane. Ako je za domenu zaduženo više poslužitelja, postojat će više zapisa. Zapise tipa A već smo upoznali; oni za simboličko ime s lijeve strane definiraju IP adresu računala koje ima to simboličko ime. Na primjeru prikazanog isječka vidimo da korijenski DNS poslužitelji imaju informaciju o sebi (tko je zadužen za tu domenu i koje su IP adrese tih poslužitelja), te o prvoj neposrednoj razini ispod, i to samo na razini pripadnih DNS poslužitelja. Tako korijenski DNS poslužitelj zna još tko je DNS poslužitelj za vršne domene poput "com", "edu", "org", "gov", "mil", za nacionalne domene poput "hr", "uk", "jp" i slično te koje su IP adrese tih DNS poslužitelja.

Na nižim razinama priča se ponavlja. Primjerice, DNS poslužitelj zadužen za domenu "hr" opet zna podatke o sebi, podatke o IP adresama simboličkih imena koja su direktno u toj domeni (primjerice, IP adresu povezanu s imenom "www.hr.") te nazive i IP adrese DNS poslužitelja za neposrednu nižu razinu poddomena (poput poddomene "fer.hr.", "index.hr.", "forum.hr." i slično). Zapisi tipa CNAME koje također možete pronaći definiraju koje je kanonsko ime nekog zamjenskog imena.

Program koji mora razriješiti simboličko ime u IP adresu uobičajeno koristi iterativni postupak razrješavanja. Pogledajmo to na primjeru razrješavanja IP adrese računala "ferko.fer.hr.". Program će najprije jednom od korijenskih DNS poslužitelja poslati upit, a korijenski DNS poslužitelj će odgovoriti "ne znam, ali pitaj hr-ns-1.carnet.hr. čija je IP adresa 161.53.160.100" (to je DNS poslužitelj zadužen za domenu "hr."); uz taj, poslat će podatke i o druga dva DNS-a zadužena za domenu "hr.". Klijent sada ponavlja upit, samo jednom od tako dobivenih DNS poslužitelja. Primjerice, poslat će upit na adresu 161.53.160.100, i dobiti odgovor od tog DNS poslužitelja "ne znam ali pitaj labs5.fer.hr. čija je IP adresa 161.53.72.20". Klijent sada i po treći puta ponavlja upit i šalje ga na 161.53.72.20; kako je taj DNS poslužitelj upravo poslužitelj koji je zadužen za domenu "fer.hr.", on će u svojoj bazi pronaći informaciju da je IP adresa računala ferko.fer.hr. 161.53.65.81 i taj će podatak vratiti.

U praksi, klijenti ne komuniciraju direktno s ovim DNS-ovima već komuniciraju s takozvanim keširajućim DNS poslužiteljima. Pri podešavanju mrežnih postavki na računalu, uz definiranje IP adrese, mrežne maske i pretpostavljenog usmjernika još se definira i IP adresa DNS poslužitelja kojemu treba slati sve upite. Taj DNS poslužitelj će razrješavanje odraditi za nas, vratiti nam rezultat i u svojo memoriji rezultat čuvati još neko vrijeme kako bi na sljedeći upit odmah vratiti zapamćenu informaciju. Na taj se način ujedno i rasterećuje globalni sustav DNS-ova.

S obzirom da se informacija o trenutno odabranom DNS-poslužitelju podešava na razini operacijskog sustava, operacijski sustav nudi zajednički API svim klijentima čime se izbjegava potreba da svaki klijent zna implementacijske detalje ovog protokola. Operacijski sustav nudi dodatni mehanizam kojim je moguće lokalno konfigurirati odgovore na neke upite (bez potrebe da se uopće ide u DNS-sustav). Radi se o datoteci `/etc/hosts` (na Linuxu) odnosno `C:\Windows\System32\drivers\etc\hosts` (na Windowsima) koju taj API prvu pregledava, i u nju je moguće upisati fiksan odgovor na pitanje koja je IP adresa vezana uz neko simboličko ime. Format datoteke je na oba operacijska sustava jednak: u svakom retku dolazi IP adresa pa simboličko ime. Pogledajte na Vašem računalu kako izgleda ta datoteka i unutra ćete pronaći redak:

```
127.0.0.1    localhost
```

što je razlog zbog kojega umjesto pisanja IP adrese `127.0.0.1` uvijek možete koristiti simboličko ime `localhost`; pokušaj razrješavanja tog imena upravo će se zbog tog retka razriješiti u IP adresu `127.0.0.1`.



Bilješka

Neki od programa za blokiranje reklama s weba koje instalirate na Vaše računalo koriste upravo ovaj mehanizam jer i web-preglednici za razrješavanje simboličkih imena u IP-adrese koriste API operacijskog sustava. Takvi programi simbolička imena poznatih web-poslužitelja s kojih se dohvaćaju reklame upišu u ovu datoteku i mapiraju na nepostojeću IP adresu (često upravo 127.0.0.1). Kako na toj adresi nema web-poslužitelja, reklama se ne učitava i korisnik dobiva "čistu" stranicu.

Ovaj mehanizam i sami možete iskoristiti ako svojem računalu lokalno (tako da to vrijedi samo na Vašem računalu) želite dati jedno ili više simboličkih imena (primjerice, ako želite isprobati jeste li dobro implementirali web-poslužitelj koji omogućava posluživanje više virtualnih domena).

Za istraživanje stabla prikazanog na slici 20.14 možete se poslužiti naredbom **nslookup** koja je dostupna na većini operacijskih sustava. Primjerice, da biste doznali koji su DNS poslužitelji zaduženi za korijensku domenu, zadajte naredbu:

```
nslookup -type=NS .
```

(uočite, tražimo zapise tipa NS). Da biste doznali tko su DNS-poslužitelji za domenu "hr.", upišite:

```
nslookup -type=NS hr.
```

Postupak možete ponoviti za bilo koju domenu. Da biste doznali IP adresu pridruženu simboličkom imenu "ferko.fer.hr.", tražite zapis tipa A:

```
nslookup -type=A ferko.fer.hr.
```

Naredba će Vam ujedno ispisati i jeste li dobili autoritativan odgovor ili ne. Neautoritativan odgovor ćete dobiti od DNS-poslužitelja koji nije "vlasnik" (odnosno koji nije zadužen za) podatka koji tražite. Primjerice, pitate li javni Googleov keširajući DNS-poslužitelj dostupan na IP adresi `8.8.8.8` da Vam dohvati IP adresu od računala "ferko.fer.hr.", dobit ćete neautoritativan odgovor:

```
nslookup -type=A ferko.fer.hr. 8.8.8.8
```

no pitate li isto DNS-poslužitelj `labs5.fer.hr` čija je IP adresa `161.53.72.20` da Vam odgovore:

```
nslookup -type=A ferko.fer.hr. 161.53.72.20
```

dobit ćete autoritativan odgovor jer je taj DNS upravo jedan od onih zaduženih za domenu "fer.hr.".

Ako ste se kojim slučajem zapitali koga pita naredba **nslookup** kada ne navedete ovaj posljednji argument (IP adresu DNS-poslužitelja kojega treba pitati), odgovor je jednostavan: DNS poslužitelj koji je podešen na razini operacijskog sustava. Na Linuxu, taj ćete podatak naći ako izlistate sadržaj datoteke `/etc/resolv.conf`; na Windowsima, podatak potražite na postavkama mrežne kartice preko koje ste spojeni na Internet.

U Javi, funkcionalnost razrješavanja simboličkih imena u IP adrese nudi se kroz razred `InetAddress`. Uporabom statičke metode `getAllByName(...)` možemo dohvatiti popis svih IP adresa koje su pridijeljene nekom simboličkom imenu. Uporabom statičke metode `getByName(...)` tražimo dohvat jedne IP adrese (bilo koje; ako ime ima više IP adresa, metoda će vratiti jednu od njih i nije specificirano koju točno). Program koji ovo ilustrira prikazan je u nastavku. Radi se o programu koji simboličko ime prima preko naredbenog retka kao jedini argument.

Primjer 20.3. Razrješavanje simboličkog imena u IP adresu

```
1 package book.java.network.dns;
2
3 import java.net.InetAddress;
4 import java.net.UnknownHostException;
5
6 import book.java.network.Util;
7
8 public class NameToIP {
```

```
 9
10  public static void main(String[] args) {
11   if(args.length != 1) {
12    System.out.println("Očekivao sam simboličko ime.");
13    return;
14   }
15
16   String name = args[0];
17
18   try {
19    InetAddress[] addresses = InetAddress.getAllByName(name);
20    System.out.printf(
21     "Sve adrese registrirane uz ime %s su:%n", name);
22    for(InetAddress addr : addresses) {
23     System.out.println(Util.addrToString(addr));
24    }
25
26    System.out.println();
27
28    InetAddress addr = InetAddress.getByName(name);
29    System.out.printf(
30     "Jedna adresa registrirana uz ime %s je:%n", name);
31    System.out.println(Util.addrToString(addr));
32   } catch(UnknownHostException ex) {
33    System.out.println("Ne mogu odrediti adresu.");
34   }
35  }
36
37 }
38
```

Obje metode u slučaju da uz simboličko ime nije vezana niti jedna IP adresa izazivaju iznimku `UnknownHostException`. Pokrenemo li program na sljedeći način:

```
java book.java.network.dns.NameToIP www.google.com
```

dobit ćemo ispis:

```
Sve adrese registrirane uz ime www.google.com su:
173.194.44.19
173.194.44.17
173.194.44.18
173.194.44.16
173.194.44.20
2A00:1450:4016:0802:0000:0000:0000:1011

Jedna adresa registrirana uz ime www.google.com je:
173.194.44.19
```

Statičku metodu `InetAddress.getByName(...)` možemo iskoristiti i ako imamo IP adresu zapisanu kao string pa je želimo pretvoriti u primjerak razreda `InetAddress` (ili u polje okteta). Naime, predamo li toj metodi string koji je oblika IP adrese, tada metoda neće kontaktirati DNS sustav već će samo odraditi posao parsiranja tog stringa i pretvorbe u polje okteta koje će zamotati u primjerak razreda `Inet4Address` ili `Inet6Address` (ovisno o tome koju smo IP adresu zapisali). Evo primjera uporabe.

```
1 // Parsiraj tekstovni zapis IP-a:
2 InetAddress address = InetAddress.getByName("161.53.65.101");
3 // Sljedeće je legalno jer je adresa bila IPv4:
4 Inet4Address ip4Address = (Inet4Address)address;
```

```
5 // Do polja okteta možemo doći ovako:
6 byte[] okteti = address.getAddress();
7 // To je polje veličine 4 elementa.
```

Detekcija poslužitelja koji šalju neželjenu e-poštu

Neželjena elektronička pošta danas u mreži stvara velike količine podataka koji troše resurse mreže a nikome nisu korisni (engleski termin za ovu vrstu pošte je *spam*). Postoji mnoštvo načina borbe protiv takve vrste pošte, a jedan od njih temelji se i na uporabi sustava DNS. Danas na Internetu postoje organizacije koje održavaju popise IP adresa računala koja šalju neželjenu poštu (te popise nazivamo crnim listama) i koje omogućavaju jednostavno pretraživanje popisa uporabom DNS-a. Pogledat ćemo konkretan primjer crne liste koja se održava na adresi http://cbl.abuseat.org. Zanima nas je li računalo čija je IP adresa `161.53.65.224` na toj listi. Da bismo to doznali, oktete IP adrese treba zapisati obrnutim redosljedom i potom nadodati `cbl.abuseat.org`. Dobit ćemo simboličko ime `224.65.53.161.cbl.abuseat.org`. To simboličko ime potrebno je uporabom DNS-a razriješiti u IP adresu (tražimo pripadni zapis tipa A). Ako takav postoji (tada će kao IP adresa obično biti vraćeno `127.0.0.1` ili nešto slično), originalna IP adresa se nalazi na listi; ako dobijemo pogrešku (zapis tipa A ne postoji), tada IP adresa nije na listi računala koja šalju neželjenu e-poštu.

Uz prethodni opis načina uporabe DNS usluge iz Java programa, sada biste trebali bez većih problema riješiti sljedeći zadatak. Napišite program koji kroz naredbeni redak prima IP adresu računala koje treba ispitati i koji na zaslon ispisuje nalazi li se to računalo na listi pošiljatelja neželjene e-pošte koja je dostupna na `cbl.abuseat.org`.

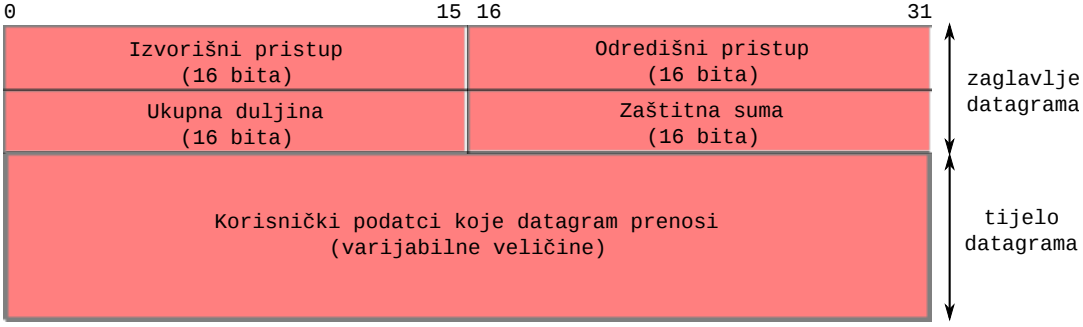
Zanima li Vas više ova problematika, predlažem da kao polaznu točku proučite tekst dostupan na http://en.wikipedia.org/wiki/DNSBL.

Protokol UDP - nepouzdan prijenos podataka

Kroz prethodni tekst podsjetili smo se da na razini mrežnog sloja protokol IP nudi mogućnost prijenosa podataka od bilo kojeg računala do bilo kojeg računala na Internetu. Protokol UDP (engl. *User Datagram Protocol*) je protokol prijenosnog sloja ISO/OSI referentnog modela koji omogućava komunikaciju između dvije udaljene aplikacije i pri tome nudi uslugu bespojnu uslugu nepouzdanog prijenosa podataka. Da bi ponudio mogućnost prijenosa podataka od aplikacije do aplikacije, uz to što mora znati na kojem se računalu aplikacija nalazi (što je određeno adresom računala na kojem je aplikacija pokrenuta), UDP mora znati i kojoj točno aplikaciji na tom računalu treba isporučiti podatke. U tu svrhu, svaka aplikacija mora za potrebe komunikacije uporabom protokola UDP stvoriti pristupnu točku i za tu pristupnu točku zatražiti jedinstvenu adresu unutar računala. Ta adresa koja (lokalno) unutar računala jednoznačno određuje pristupnu točku, naziva se *pristup* (engl. *port*). Protokol UDP za pristupe koristi 16-bitne cijele brojeve, pa je najmanji valjani pristup 1 a najveći 65535 (pristup 0 ima posebno značenje).

Implementacijski, protokol UDP razmjenjuje jedinice koje nazivamo *datagrami*. Struktura jednog datagrama prikazana je na slici 20.15.

Slika 20.15. UDP-datagram



Datagram se sastoji od zaglavlja i tijela. U zaglavlju datagrama nalazi se minimalna količina podataka: lokalna adresa pristupne točke aplikacije koja je stvorila datagram (izvorišni pristup), lokalna adresa pristupne točke aplikacije kojoj na udaljenom računalu treba isporučiti teret koji nosi datagram (odredišni pristup), duljina datagrama te zaštitna suma (koristi se pri detekciji pogrešaka u prijenosu). Ovakav datagram na mrežnom se sloju pakira u IP-paket; u zaglavlju IP-paketa zapisat će se izvorišna i odredišna adresa računala na kojima se nalaze pristupne točke.

Uslijed ovakve jednostavne strukture, komunikacija UDP-datagramima praktički se svodi na komunikaciju IP-paketima (uz bonus što negdje postoji mjesto na koje možemo upisati još i lokalne adrese pristupnih točaka kako bismo ponudili komunikaciju od aplikacije do aplikacije). Zbog toga, komunikacija UDP-om preuzima sve karakteristike IP-a: onaj tko stvara datagram mora sam odrediti koje sve podatke stavlja kao teret datagrama; ako podataka ima mnogo, sam mora odrediti kako će te podatke razlomiti na manje jedinice koje će slati u pojedinim datagramima; nema garancije da će jednom poslani datagram uopće stići do odredišta (jer se IP-paket koji ga nosi u mreži može izgubiti); nema garancije da će, ako stigne do odredišta, stići samo jednom (moguće je da se IP-paket u mreži uvišestruči pa kroz neko vrijeme više kopija stigne do odredišta); konačno, ako korisnik pošalje više UDP-datagrama, nema nikakve garancije da će redoslijed kojim će oni pristići do odredišta (ako uopće pristignu) odgovarati redoslijedu kojim su poslani.

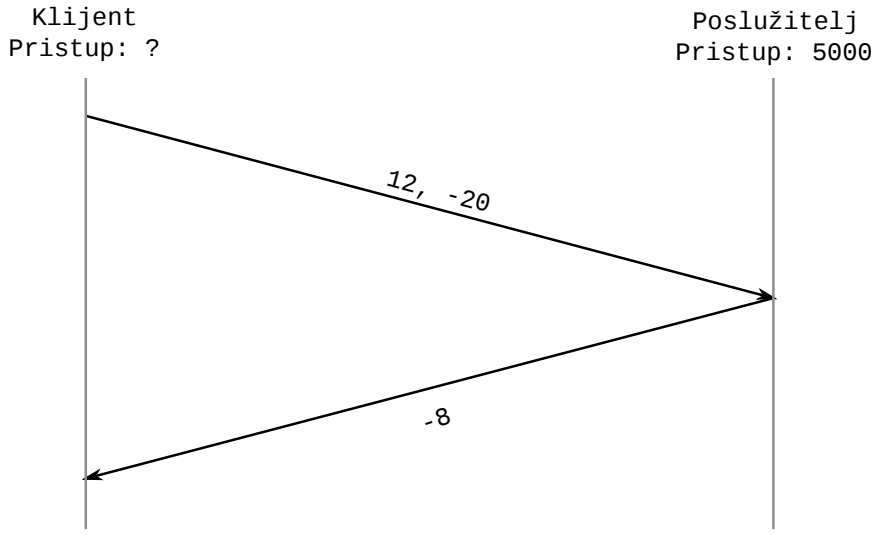
Aplikacije kojima ne trebaju takve garancije mogu koristiti UDP. Jedan takav primjer je protokol DNS koji smo prethodno obradili. Taj protokol za prijenos upita za razrješavanjem imena u IP adresu (i sličnih) koristi UDP. Klijent koji šalje zahtjev obično je izveden tako da zahtjev pošalje pa neko vrijeme čeka na odgovor. Ako odgovor ne stigne, klijent naprosto pretpostavi da upit nije niti došao do DNS poslužitelja pa ga pošalje opet. Ovaj postupak ponavlja se nekoliko puta sve dok odgovor ne stigne ili dok klijent zaključi da ne može razgovarati s poslužiteljem.

Aplikacije koje trebaju jače garancije obično koriste protokol TCP koji je nešto "skuplji" ali nudi mnoštvo garancija. Taj je protokol opisan nešto kasnije u ovom poglavlju.

Uporaba UDP-a u Javi

U nastavku ćemo pogledati primjer izrade klijenta i poslužitelja koji međusobno komuniciraju uporabom protokola UDP. Pri tome ćemo implementirati jednostavan aplikacijski protokol kojim poslužitelj nudi klijentima uslugu zbrajanja dva cijela 16-bitna broja s predznakom. Primjer jedne takve konverzacije prikazan je na slici u nastavku.

Slika 20.16. Jednostavan aplikacijski protokol "Zbrajanje"



Opišimo ukratko protokol koji ćemo implementirati. Protokol definira dvije vrste poruka: zahtjev i odgovor. Zahtjev je poruka u kojoj su navedena dva 16-bitna cijela broja u zapisu dvojnog

komplementa (što u Javi odgovara tipu `short`). Kako je poruka skup okteta, 16-bitne brojeve treba rastaviti na oktete. Naš će protokol zahtijevati da taj rastav bude u skladu s načinom *big-endian*: najprije će biti zapisan oktet koji sadrži značajnijih 8 bitova a potom oktet koji sadrži manje značajnih 8 bitova. Pribrojnici u poruci moraju biti zapisani slijedno, što znači da će zahtjev biti poruka veličine 4 okteta. Primjerice, zahtjev koji traži zbrajanje brojeva 12 i -20, sadržavat će sljedeće oktete: `00 0C FF F4` (broj 12 zapisan u 16 bitova u zapisu dvojnog komplementa je 0000000000001100 što je u heksadekadskom zapisu 000C, a broj -20 zapisan u 16 bitova u zapisu dvojnog komplementa je 1111111111110100 što je u heksadekadskom zapisu FFF4).

Odgovor će poruka koja sadrži samo jedan 16-bitni cijeli broj u zapisu dvojnog komplementa i taj broj odgovara sumi brojeva koji su pristigli u zahtjevu.

Za potrebe konverzija 16-bitnih cijelih brojeva u zapisu dvojnog komplementa u polje okteta (i obratno) pripremili smo mali pomoćni razred koji je prikazan u nastavku.



Bilješka

Svi programski kodovi koji su dani u nastavku ovog poglavlja dostupni su i na adresi http://java.zemris.fer.hr/nastava/opjj/net.zip.

Primjer 20.4. Pomoćni razred za pretvorbu `short` u/iz okteta

```
 1
 2 public class ShortUtil {
 3  /**
 4   * U predani spremnik {@code buf} od pozicije
 5   * {@code offset} zapisuje dva okteta u big-endian
 6   * poretku koja čine 16-bitova predane cjelobrojne
 7   * varijable {@code number} koja je tipa {@code short}
 8   * @param number broj koji treba rastaviti na oktete
 9   * @param buf spremnik u koji treba zapisati rezultat
10   * @param offset pozicija od koje treba zapisati rezultat
11   */
12  public static void shortToBytes(short number,
13   byte[] buf, int offset) {
14   buf[offset] = (byte)((number>>8) & 0xFF);
15   buf[offset+1] = (byte)(number & 0xFF);
16  }
17
18  /**
19   * Iz predanog spremnika {@code buf} počev od pozicije
20   * {@code offset} čita dva okteta, tumači ih kao oktete
21   * cijelog broja tipa {@code short} u
22   * formatu big-endian i rekonstruira taj cijeli broj.
23   * @param buf spremnik iz kojeg treba uzeti dva okteta
24   * @param offset pozicija na kojoj se nalazi prvi oktet
25   * @return rekonstruirani cijeli broj
26   */
27  public static short bytesToShort(byte[] buf, int offset) {
28   return (short)(
29    (buf[offset] & 0xFF) << 8 |
30    (buf[offset+1] & 0xFF));
31  }
32 }
33
```

Pri izradi ovih metoda primjetit ćete uporabu `& 0xFF`: Java prije uporabe binarnih operatora sve operande koji su kraći od tipa `int` najprije širi u `int` i tek tada obavlja operaciju. Kod negativnih brojeva to će rezultirati višim oktetima koji su popunjeni s jedinicama pa na to treba pripaziti.

Sada smo spremni za pisanje klijenta. Programski kod dan je u primjeru 20.5.

Primjer 20.5. UDP-klijent za specificirani protokol

```
 1 import java.io.IOException;
 2 import java.net.DatagramPacket;
 3 import java.net.DatagramSocket;
 4 import java.net.InetAddress;
 5 import java.net.InetSocketAddress;
 6 import java.net.SocketException;
 7 import java.net.SocketTimeoutException;
 8 import java.net.UnknownHostException;
 9
10
11 public class UDPKlijent {
12
13  public static void main(String[] args)
14   throws SocketException, UnknownHostException {
15
16   if(args.length!=4) {
17    System.out.println("Očekivao sam host port broj1 broj2");
18    return;
19   }
20
21   String hostname = args[0];
22   int port = Integer.parseInt(args[1]);
23
24   short broj1 = Short.parseShort(args[2]);
25   short broj2 = Short.parseShort(args[3]);
26
27   // Pripremi podatke upita koje treba poslati:
28   byte[] podatci = new byte[4];
29   ShortUtil.shortToBytes(broj1, podatci, 0);
30   ShortUtil.shortToBytes(broj2, podatci, 2);
31
32   // Stvori pristupnu točku klijenta:
33   DatagramSocket dSocket = new DatagramSocket();
34
35   // Umetni podatke u paket i paketu postavi adresu i
36   // port poslužitelja
37   DatagramPacket packet = new DatagramPacket(
38    podatci, podatci.length
39   );
40   packet.setSocketAddress(new InetSocketAddress(
41    InetAddress.getByName(hostname),
42    port
43   ));
44
45   // Postavi timeout od 4 sekunde na primitak odgovora:
46   dSocket.setSoTimeout(4000);
47
48   // Šalji upite sve dok ne dobiješ odgovor:
49   boolean answerReceived = false;
50   while(!answerReceived) {
```

```
51
52     // Pošalji upit poslužitelju:
53     System.out.println("Šaljem upit...");
54     try {
55      dSocket.send(packet);
56     } catch(IOException e) {
57      System.out.println("Ne mogu poslati upit.");
58      break;
59     }
60
61     // Pripremi prazan paket za primitak odgovora:
62     byte[] recvBuffer = new byte[4];
63     DatagramPacket recvPacket = new DatagramPacket(
64      recvBuffer, recvBuffer.length
65     );
66
67     // Čekaj na odgovor:
68     try {
69      dSocket.receive(recvPacket);
70     } catch(SocketTimeoutException ste) {
71      // Ako je isteko timeout, ponovno šalji upit
72      continue;
73     } catch(IOException e) {
74      // U slučaju drugih pogrešaka - dogovoriti se što dalje...
75      // (mi opet radimo retransmisiju)
76      continue;
77     }
78
79     // Analiziraj sadržaj paketa:
80     if(recvPacket.getLength() != 2) {
81      System.out.println(
82       "Primljen je odgovor neočekivane duljine."
83      );
84      break;
85     }
86
87     // Ispiši rezultat, i prekini slanje retransmisija:
88     System.out.println(
89      "Rezultat je: " +
90      ShortUtil.bytesToShort(
91       recvPacket.getData(), recvPacket.getOffset()
92      )
93     );
94     break;
95    }
96
97    // Obavezno zatvori pristupnu točku:
98    dSocket.close();
99   }
100
101 }
102
```

Pristupna točka koja omogućava komunikaciju uporabom protokola UDP modelirana je razredom `java.net.DatagramSocket`. UDP-datagram (odnosno njegovi dijelovi i dodatne informacije) modelirani su razredom `java.net.DatagramPacket`. Pri stvaranju UDP-datagrama korisnik je sam zadužen za alokaciju memorijskog spremnika koji sadrži podatke koje treba poslati partnerskoj pristupnoj točki, odnosno spremnika u koji će biti upisani podatci koji nam

pošalje partnerska pristupna točka (operacijski sustav neće za nas raditi zauzimanje memorije). Ukoliko zatražimo primitak UDP-datagrama a predamo spremnik koji sadrži manje mjesta no što je podataka u datagramu, dobit ćemo samo onoliko podataka koliko stane u predani spremnik, a ostatak primljenih podataka bit će odbačen.

Klijentski program koji pišemo očekuje četiri argumenta: ime poslužitelja (ili adresu), pristup na kojem poslužitelj prima UDP-datagrame te dva broja koje je potrebno poslati na zbrajanje. Taj se dio čita u retcima 16 do 25. U retku 28 zauzima se spremnik veličine 4 okteta u koji će biti upisani okteti brojeva koje šaljemo poslužitelju; samo upisivanje obavlja se u retcima 29 i 30. Potom stvaramo pristupnu točku (redak 33) te datagram koji sadrži pripremljene podatke (retci 37-39). Za datagram je još potrebno odrediti odredište, što radimo pozivom metode `setSocketAddress(...)`. Ova metoda kao argument očekuje primjerak razreda `SocketAddress`. Primjerci razreda `SocketAddress` predstavljaju uređeni par adrese računala na mreži s kojim želimo komunicirati te pristup na kojem poslužitelj sluša. Međutim, kako vrsta adrese na ovoj razini nije zadana, razred je apstraktan i nije moguće izravno stvarati njegove primjerke. Uređen par koji predstavlja IP adresu računala i pristup modeliran je razredom `InetSocketAddress` koji je izveden iz razreda `SocketAddress`. Taj razred nije apstraktan i njega ćemo koristiti. Konstruktor razreda `InetSocketAddress` očekuje dva argumenta: IP adresu te pristup. Do IP adrese dolazimo uporabom statičke metode `getByName` razreda `InetAddress` s kojom smo se već upoznali.

Uočite da nema potrebe definirati IP adresu i pristup koji koristi klijent. Te će podatke u UDP-datagram (i IP-paket koji će ga odnijeti) upisati sama pristupna točka kroz koju šaljemo datagram. Želimo li našu pristupnu točku fiksirati na neku IP adresu i pristup, to možemo napraviti uporabom metode `bind(...)` koju nudi pristupna točka. Ako to ne napravimo, pristupna točka će od operacijskog sustava zatražiti uporabu nekog od pristupa koji je u tom trenutku slobodan. Ako pristupnu točku stvorimo uporabom pretpostavljenog konstruktora, taj će konstruktor zatražiti od operacijskog sustava dodjelu jednog (bilo kojeg) pristupa koji je u tom trenutku slobodan i povezat će pristupnu točku na taj pristup (i "zamjensku" IP adresu koja predstavlja sve IP adrese računala; konkretnu adresu koja će se koristiti pri slanju paketa dodijelit će jezgra operacijskog sustava).

U retku 46 nad pristupnom točkom se definira vremensko ograničenje (engl. *timeout*) na čekanje na primitak odgovora kada se pozove metoda `receive` nad pristupnom točkom; bez definiranog vremenskog ograničenja metoda `receive` ponaša se kao blokirajuća metoda koja čeka sve dok ne ne primi neki datagram (čime može trajno zablokirati program u situaciji u kojoj se odgovor poslan ali je negdje putem bio izgubljen).

Sada program ulazi u petlju koju ponavlja sve dok ne dobije odgovor. Petlja se sastoji od dva koraka. U prvom koraku, šalje se zahtjev za izračun poslužitelju (ako to nije prva iteracija petlje, onda ovo predstavlja ponovno slanje - retransmisiju - zahtjeva). Drugi korak je čekanje na odgovor. Ako istekne vremensko ograničenje, kreće se u novu iteraciju petlje. Ako je odgovor pristigao, analizira se i ispisuje na zaslon.

Kada je sve gotovo, pristupna točka se zatvara i program završava.

Poslužiteljska aplikacija konceptualno je dosta slična klijentskoj, samo što akcije obavlja obrnutim redosljedom: poslužitelj najprije čeka na zahtjev, potom ga analizira, stvara odgovor i odgovor šalje. Čekanje pri tome nije vremenski ograničeno: poslužitelj je aplikacija koja čeka tako dugo dok ne dobije zahtjev; isti obrađuje i opet se vraća na čekanje.

Programski kod dan je u primjeru 20.6.

Primjer 20.6. UDP-posluzitelj za specificirani protokol

```
1 import java.io.IOException;
2 import java.net.DatagramPacket;
3 import java.net.DatagramSocket;
4 import java.net.InetAddress;
5 import java.net.InetSocketAddress;
```

```
 6 import java.net.SocketException;
 7
 8
 9 public class UDPPosluzitelj {
10
11  public static void main(String[] args) throws SocketException {
12   if(args.length!=1) {
13    System.out.println("Očekivao sam port");
14    return;
15   }
16
17   int port = Integer.parseInt(args[0]);
18
19   // Stvori pristupnu točku poslužitelja:
20   @SuppressWarnings("resource")
21   DatagramSocket dSocket = new DatagramSocket(null);
22   dSocket.bind(new InetSocketAddress((InetAddress)null, port));
23
24   while(true) {
25    // Pripremi prijemni spremnik:
26    byte[] recvBuffer = new byte[40];
27    DatagramPacket recvPacket = new DatagramPacket(
28     recvBuffer, recvBuffer.length
29    );
30
31    // Čekaj na primitak upita:
32    try {
33     dSocket.receive(recvPacket);
34    } catch (IOException e) {
35     continue;
36    }
37
38    // Je li upit korektne duljine?
39    if(recvPacket.getLength()!=4) {
40     System.out.println(
41      "Primljen neispravan upit od " +
42      recvPacket.getSocketAddress() + "."
43     );
44     continue;
45    }
46
47    // Otkomentirati ako treba odglumiti gubitak u komunikaciji:
48    // if(Math.random() < 0.5) continue;
49
50    // Izračunaj rezultat:
51    short broj1 = ShortUtil.bytesToShort(
52     recvPacket.getData(), recvPacket.getOffset());
53    short broj2 = ShortUtil.bytesToShort(
54     recvPacket.getData(), recvPacket.getOffset() + 2);
55    short result = (short)(broj1 + broj2);
56
57    // Pripremi podatke koje treba poslati natrag:
58    byte[] sendBuffer = new byte[2];
59    ShortUtil.shortToBytes(result, sendBuffer, 0);
60
61    // Pripremi odlazni paket:
62    DatagramPacket sendPacket = new DatagramPacket(
63     sendBuffer, sendBuffer.length
```

```
64      );
65      sendPacket.setSocketAddress(recvPacket.getSocketAddress());
66
67      // Pošalji ga:
68      try {
69       dSocket.send(sendPacket);
70      } catch (IOException e) {
71       System.out.println("Greška pri slanju odgovora.");
72      }
73     }
74    }
75  }
76
```

Kako bi sve prikazano u poslužiteljskoj aplikaciji sada već trebalo biti jasno, prokomentirajmo samo jedan mali detalj. Poslužitelj, nakon što stvori pristupnu točku, istu eksplicitno želi povezati na određeni pristup. Razlog leži u prethodno opisanom ponašanju pristupne točke: pri prvoj uporabi, ako pristupnoj točki nije dodijeljena neka lokalna adresa (pristup), operacijski će joj sustav dodijeliti neku slučajnu (koja je u tom trenutku slobodna). To je, međutim, nepovoljan scenarij za klijenta: klijent mora znati na kojem pristupu poslužitelj sluša kako bi mu poslao zahtjev. Stoga poslužitelji uobičajeno traže od operacijskog sustava uporabu nekog unaprijed dogovorenog pristupa koji je poznat klijentima. Ovo povezivanje obavlja se metodom `bind(...)` u retku 22. Kao adresu predajemo referencu `null`, čime tražimo isporuku datagrama koji su pristigli na bilo koju od naših IP adresa (ali na zadani pristup).

Sljedeći interesantan detalj je slanje odgovora: nakon što je stvoren datagram koji će prenijeti odgovor klijentu (retci 62-64), datagramu je još potrebno definirati odredište (kamo se šalje). No taj podatak znamo: datagram koji smo primili naprosto pitamo od kuda je došao i taj podatak predamo kao odredište odgovora (redak 65).

Da bismo provjerili rad napisanih programa, trebat ćemo dvije konzole. U prvoj konzoli najprije treba pokrenuti poslužitelj:

```
java -cp bin UDPPosluzitelj 5000
```

a zatim u drugoj konzoli pokrenite klijenta (jednom ili više puta):

```
java -cp bin UDPKlijent localhost 5000 12 -20
```

Ako iz bilo kojeg razloga komunikacija ne uspije, provjerite imate li na računalu pokrenut sigurnosni zaštitni uređaj koji blokira komunikaciju. Ako želite isprobati između različitih računala na Internetu, prisjetite se upozorenja koja smo dali za računala koja su u privatnim lokalnim mrežama koje se na Internet spajaju preko NAT-a: morat ćete podesiti proslijeđivanje portova da bi se netko "izvana" mogao spojiti na Vaš poslužitelj koji je u privatnoj mreži. Na klijentskoj strani nema takvih problema.

Protokol TCP - pouzdan prijenos podataka

Protokol TCP (engl. *Transmission Control Protocol*) pripada prijenosnom sloju ISO/OSI referentnog modela. To je protokol koji omogućava komunikaciju između dvije udaljene aplikacije i pri tome nudi uslugu pouzdanog prijenosa. Prema aplikaciji koja koristi takvu uslugu to znači da će poruke aplikacije biti prenesene na drugu stranu (ili će aplikaciji biti dojavljeno da to nije moguće), poruke će biti isporučene udaljenoj aplikaciji redosljedom kojim ih je izvorna aplikacija predala, i konačno, u slučaju da se pri prolazu kroz mrežu neke poruke uvišestruče, usluga ima mehanizam otkrivanja takvih situacija pa će udaljenoj aplikaciji i dalje biti isporučen samo po jedan primjerak (duplikati nastali u mreži bit će odbačeni i udaljena ih aplikacija neće vidjeti). Sam protokol prema aplikaciji nudi apstrakciju toka okteta (kažemo da je oktetno-orijentirani protokol), što znači da aplikacija predaje niz okteta koje treba prebaciti na drugu stranu,

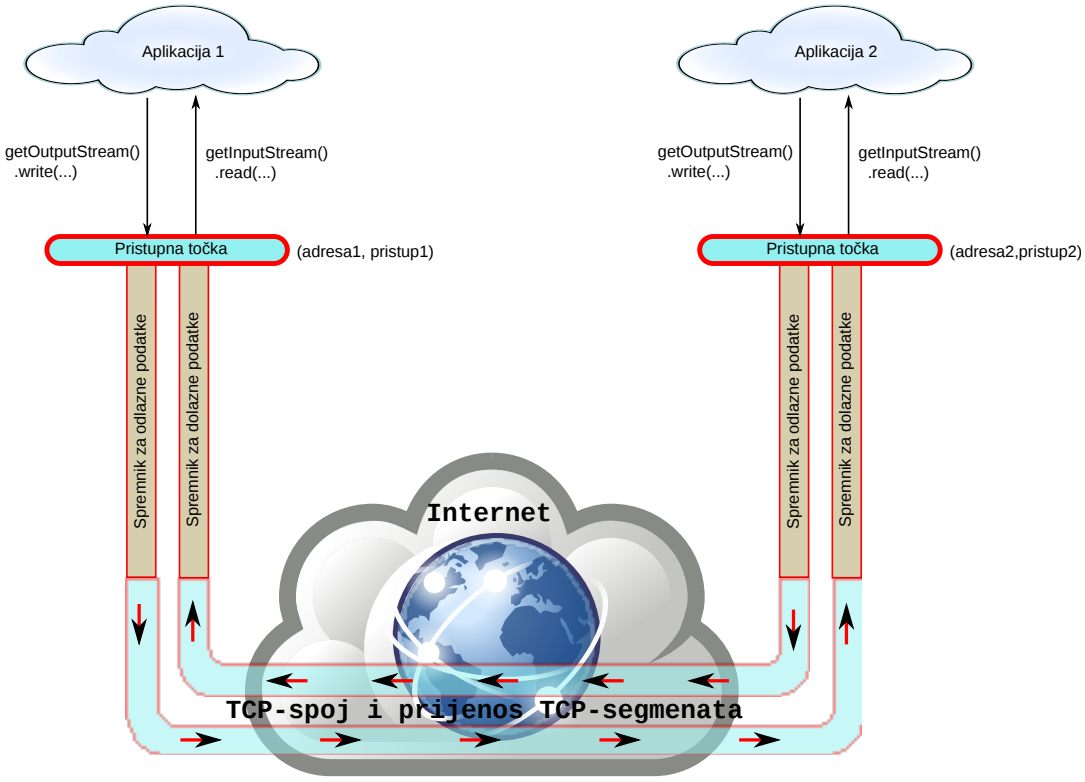
a TCP implementacija sama odlučuje kako će to napraviti (koliko će okteta u kojem trenutku prebacivati). U određenom smislu, programer na uporabu TCP-a može gledati slično kao na uporabu datoteke.

Prema programeru TCP nudi model komunikacije koji podrazumijeva dvije udaljene pristupne točke gdje se iza svake pristupne točke kriju dva zasebna spremnika: jedan je spremnik za odlazne podatke u koji aplikacija malo po malo zapisuje oktete koje treba prenijeti a TCP ih povremeno prenosi na drugu stranu (i tako prazni taj spremnik). Drugi spremnik je spremnik za dolazne podatke u koji TCP malo po malo nakrcava podatke koji stižu od udaljene pristupne točke i koje aplikacija povremeno čita (i time prazni taj spremnik).

Uočite također da su odgovarajući spremnici na obje strane međusobno povezani upravo protokolom TCP: podatci koje aplikacija na jednoj strani upisuje u odlazni spremnik odnose se TCP-om preko mreže, upisuju u spremnik za dolazne podatke na drugoj strani i iz tog spremnika udaljena ih aplikacija čita.

Kod ovog protokola spojna točka jednoznačno je određena adresom računala te pristupom (engl. *port*) koji u određenom smislu predstavlja "lokalnu" adresu pristupne točke u računalu.

Slika 20.17. Konceptualan prikaz komunikacije uporabom protokola TCP



Kako bi ponudile uslugu pouzdanog prijenosa, dvije strane koje implementiraju protokol TCP razmjenjuju poruke koje su formatirane kao TCP-segmenti (na slici prikazano strelicama). TCP-segment ima poprilično bogato zaglavlje koje se koristi kako bi se uz podatke koje je stvorila aplikacija i koje je potrebno prenijeti na drugu stranu moglo razmjenjivati i niz upravljačkih informacija. Primjerice, TCP numerira sve podatke preuzete od aplikacije i koje šalje kao teret segmenta. Upravo zahvaljujući tome, druga strana po primitku TCP-segmenta koji donosi podatke može zaključiti jesu li to podatci koji su stigli pogrešnim redoslijedom, jesu li to podatci koji su već bili isporučeni aplikaciji i slično. TCP također koristi potvrde: da bi znao da su podatci uspješno isporučeni na drugu stranu, TCP očekuje da druga strana vrati TCP-segment kojim će potvrditi prijem podataka; ako se to ne dogodi, TCP će krenuti u ponovno slanje istih podataka (jer pretpostavlja da, budući da nije stigla potvrda, podatci nisu bili isporučeni). Ako

je slučajno došlo do gubitka potvrde a ne originalnih podataka, nema nikakvih problema; kada TCP drugoj strani pošalje još jednom segment s istim podatcima, druga strana će, zahvaljujući činjenici da su podatci numerirani, zaključiti da je te podatke već isporučila prema svojoj aplikaciji pa ih neće ponovno isporučivati, ali će pošiljatelju ponovno generirati potvrdu čime će i pošiljatelj znati da su njegovi podatci doista isporučeni.

TCP prije no što dopusti razmjenu bilo kakvih podataka između dvije pristupne točke zahtjeva da se provede postupak poznat pod nazivom *3-way handshake*. Tim postupkom uspostavlja se spoj između pristupnih točaka. Ovaj postupak zahtjeva razmjenu od minimalno 3 TCP-segmenta (ako nema nikakvih grešaka, gubitaka i slično). Tijekom tog postupka implementacije na obje strane se dogovaraju oko parametara koji će biti korišteni tijekom daljnje komunikacije (poput početnih rednih brojeva s kojima će krenuti numeracija, maksimalne veličine TCP-segmenta koji se može prenijeti između dvije pristupne točke te drugih). Dok se ovaj postupak ne završi, pristupne točke ne mogu prenositi poruke svojih aplikacija; stoga obično kažemo da je postupak uspostave spoja *skup* (u vremenskom smislu). Kako se pri uspostavi spoja na oba računala ujedno zauzimaju i memorijski resursi (spremnici za podatke) te drugi resursi, važno je osigurati da se po završetku komunikacije obavi postupak poznat kao *raskid spoja*. Spoj je moguće raskinuti na više načina, a u najgorem slučaju može se provesti kroz razmjenu četiri dodatna TCP-segmenta, čime i taj postupak može biti vremenski zahtjevan.

Konačno, kako ne bi odmah zagušio mrežnu infrastrukturu koja prenosi TCP-segmente (koji se na mrežnom sloju pakiraju u IP-pakete koje IP prenosi do odredišnog računala), implementacije TCP-a tradicionalno koriste algoritam poznat pod nazivom *algoritam polaganog starta* (engl. *slow start algorithm*) koji regulira količinu podataka koju će TCP prenositi po jedinici vremena (drugim riječima, brzinu prijenosa). Naime, TCP će krenuti s vrlo malom brzinom (poslat će samo jedan TCP-segment i čekat će na potvrdu da je isti primljen) i potom će u svakom sljedećem koraku povećavati brzinu slanja (odnosno broj segmenata). Iako se koristi eksponencijalan porast (koji se kasnije prikladno korigira), kod komunikacija koje traju kratko odnosno koje razmjenjuju malo podataka, TCP je nepraktičan. Unatoč potencijalno vrlo brzoj mreži koju imate, prijenos ono malo podataka što se razmjenjuje bit će spor. Stoga je pri pisanju aplikacija koje koriste TCP nužno razumjeti ove mehanizme i biti svjestan koje su posljedice. Tako, primjerice, protokol HTTP koji ćemo opisati u nastavku u svojoj inačici 1.1 opisuje scenarij u kojem se jednom uspostavljeni TCP-spoj može koristiti za slanje više uzastopnih zahtjeva i odgovora između klijenta i poslužitelja, upravo u želji da se eliminira čekanje na uspostavu i raskid spoja te efekti algoritma polaganog starta.

Napomenimo još i da je protokol TCP asimetričan, u smislu da se jasno razlikuju klijent i poslužitelj. Poslužitelj je aplikacija koja mora biti prva pokrenuta i koja kod operacijskog sustava mora zatražiti određeni pristup na kojem će primati poruke. Poslužitelj može dopustiti operacijskom sustavu da mu sam odabere neki pristup koji u tom trenutku nitko ne koristi ili može inzistirati da želi baš određeni pristup. Ovaj posljednji scenarij je češći. Klijent je pak aplikacija koja inicira uspostavu spoja; klijent šalje prvi segment u postupku *3-way handshake* i da bi to mogao, on mora znati na kojem se računalu nalazi poslužitelj i mora znati na kojem pristupu poslužitelj prihvaća zahtjeve za uspostavom spoja.

Pogledajmo sada java-pseudokod programa koji ilustrira strukturu klijenta.

```
 1 String hostname = ...
 2 int port = ...
 3
 4 Socket socket = new Socket();
 5 socket.connect(
 6  new InetSocketAddress(
 7   InetAddress.getByName(hostname),
 8   port
 9  )
10 );
11
12 byte[] reqData = ...
```

```
13
14 socket.getOutputStream().write(reqData);
15 socket.getOutputStream().flush();
16
17 byte[] buf = new byte[1024];
18 int r = socket.getInputStream().read(buf);
19
20 ...
21
22 socket.close();
23
24
```

Pristupna točka protokola TCP modelirana je razredom `java.net.Socket`. Uporabom pretpostavljenog konstruktora (redak 4) stvara se nepovezana pristupna točka. Nad stvorenom pristupnom točkom možemo zatražiti pokretanje uspostave spoja (postupkom *3-way handshake*). Za tu namjenu na raspolaganju je metode `connect` (retci 5 do 10) koja kao argument očekuje primjerak razreda `SocketAddress`. Primjerci razreda `SocketAddress` predstavljaju uređeni par adrese računala na mreži s kojim želimo uspostaviti TCP-spoj te pristup na kojem poslužitelj sluša. Međutim, kako vrsta adrese na ovoj razini nije zadana, razred je apstraktan i nije moguće izravno stvarati njegove primjerke. Uređen par koji predstavlja IP adresu računala i pristup modeliran je razredom `InetSocketAddress` koji je izveden iz razreda `SocketAddress`. Taj razred nije apstraktan i njega ćemo koristiti. Konstruktor razreda `InetSocketAddress` očekuje dva argumenta: IP adresu te pristup. Do IP adrese dolazimo uporabom statičke metode `getByName` razreda `InetAddress` s kojom smo se već upoznali.

Nakon što metoda `connect` odradi uspostavu spoja, pristupna točka je povezana s udaljenom pristupnom točkom i kroz te dvije pristupne točke moguće je razmjenjivati podatke. U ovom primjeru pretpostavimo da poslužitelju želimo poslati oktete koji su pripremljeni u polju `reqData` (redak 12).

Povezana pristupna točka nudi nam metodu `getOutputStream()` kojom dobivamo `OutputStream` koji podatke zapisuje u spremnik za odlazne podatke koje će TCP prebaciti kroz mrežu u spremnik za dolazne podatke partnerske pristupne točke, te metodu `getInputStream()` kojom dobivamo `InputStream` koji podatke čita iz spremnika za dolazne podatke (koji su prebačeni preko mreže iz spremnika za odlazne podatke partnerske pristupne točke).

Uporabom metode `getOutputStream()` nad pristupnom točkom dohvaćamo izlazni tok okteta i u njega zapisujemo oktete pripremljenog zahtjeva (redak 14). Kako smo time u spremnik ubacili sve što smo imali, pozivamo još i metodu `flush()` nad izlaznim tokom time poručujemo implementaciji protokola TCP da ovaj tren više nemamo podataka i da bi bilo zgodno da eventualno zaostale podatke u spremniku za odlazne podatke što prije prenese partnerskoj pristupnoj točki (redak 15).

Potom dohvaćamo ulazni tok podataka i čitamo odgovor. Očekujemo li odgovor neke određene duljine, na ovom bismo mjestu možda htjeli implementirati čitanje u petlji, jer metoda `read()` ne garantira da će popuniti čitav predani spremnik (sjetite se rada s tokovima).

Jednom kada smo dobili sve oktete, zatvaramo pristupnu točku. Naime, TCP nakon odrađene uspostave spoja za spoj poznaje tri stanja: uspostavljeni spoj, poluzatvoreni spoj (kada je jedna strana inicirala zatvaranje u smislu da ona više neće slati podatke) i zatvoreni spoj (kada se i ona druga strana složila da više neće slati podatke). Tek kada se spoj raskine (zatvori) oslobodit će se svi resursi koji su bili zauzeti od strane pristupnih točaka na obje strane. Stoga se treba držati pravila da jednom otvorenu pristupnu točku nakon uporabe obavezno zatvorimo.

Pogledajmo još i java-pseudokod programa koji ilustrira strukturu poslužitelja.

```
1 public static void main(String[] args) throws IOException {
2
```

```
 3  int port = ...
 4
 5  ServerSocket serverSocket = new ServerSocket();
 6  serverSocket.bind(
 7   new InetSocketAddress((InetAddress)null, port)
 8  );
 9  while(true) {
10   Socket toClient = serverSocket.accept();
11   serve(toClient);
12  }
13 }
14
15 private static void serve(Socket toClient) throws IOException {
16  toClient.getInputStream().read();
17  toClient.getOutputStream().write(b);
18  toClient.close();
19 }
20
```

Pristupna točka koja se na poslužiteljima koristi za čekanje na uspostavu spoja uporabom protokola TCP u Javi je modelirana razredom `java.net.ServerSocket`. U retku 5 stvaramo primjerak takve pristupne točke. Potom je potrebno definirati na kojem će pristupu pristupna točka prihvaćati zahtjeve za uspostavom spoja te hoće li se prihvaćati zahtjevi koji su stigli na neku specifičnu adresu našeg računala (ako ih imamo više) ili na bilo koju. Ovaj dio definira se uporabom metode `bind` koja kao argument prima primjerak razreda `SocketAddress`. Predali smo primjerak koji na mjestu IP adrese ima vrijednost `null` čime smo definirali da želimo prihvaćati zahtjeve koji su do nas došli na bilo koju od naših adresa (ali na pristup koji se predaje kao drugi argument).

Jednom kada smo definirali te parametre, ulazimo u petlju `while` u kojoj program blokira u metodi `accept()` (redak 10). Naime, ta je metoda zadužena za odrađivanje svih zahtjeva za uspostavama spojeva (ona će odraditi ostatak postupka *3-way handshake* koji je inicirao klijent). Jednom kada se javi neki klijent i kada se s njime završi postupak uspostave spoja, metoda vraća *novu* pristupnu točku kroz koju se može komunicirati s klijentom s kojim je odrađena uspostava spoja. U prikazanom primjeru taj smo posao delegirali pomoćnoj metodi koja od klijenta nešto pročita, nešto mu zapiše iu potom raskine spoj (zatvori tu pristupnu točku). U tom trenutku vraćamo se u petlju `while` i ponovno zapinjemo u metodi `accept`. Poslužitelji koji trebaju biti u stanju obrađivati veliki broj klijenata paralelno neće direktno iz iste dretve koja je zadužena za čekanje i uspostavljanje spojeva pozivati i samu obradu već će to delegirati nekoj novoj dretvi (pa neka ona paralelno to odrađuje koliko god da obrada trajala), a glavna dretva će se vratiti na čekanje i uspostavu sljedećeg spoja (ako takav zahtjev već postoji).

Protokol HTTP

Protokol HTTP (odnosno njegova kriptografski zaštićena inačica HTTPS) koristi se za razmjenu web-sadržaja, odnosno objekata koji su dostupni preko weba. Protokol HTTP je "jezik" kojim razgovaraju web-klijent (Mozilla Firefox, Google Chrome, Opera, Safari i drugi) i web-poslužitelj (primjerice Apache HTTP Server) prilikom korisnikovog "surfanja" webom, odnosno kada korisnik zatraži dohvat jednog konkretnog objekta. Objekti o kojima ovdje govorimo mogu biti statički objekti različitog tipa (HTML dokument napisan u nekom uređivaču teksta, slika formata PNG, GIF, JPG i sličnih koja je pohranjena na disku, dokument tipa PDF, DOC i slični, komprimirane arhive, audio i video datoteke te još niz drugih) te dinamički generirani objekti različitih tipova (gdje ti objekti ne postoje unaprijed generirani i pohranjeni na disku već ih se dinamički generira i isporučuje "u letu" na svaki zahtjev korisnika). S obzirom da ćemo kao primjer uporabe protokola TCP razraditi jednostavan HTTP-poslužitelj, upoznat ćemo se s pojednostavljenom inačicom tog protokola. No prije toga moramo se upoznati s načinom adresiranja objekata koji su dostupni preko weba.

Ukratko o URL

URL (engl. *Uniform Resource Locator*) predstavlja adresu dohvatljivih (odnosno postojećih) objekata različitim protokolima poput HTTP, FTP i drugim. URL je podskup pojma URI (engl. *Uniform Resource Indetifier*). Pojam URI definiran je kroz specifikacije RFC 3986 i RFC 3987 (specifikacije su javno dostupne na Internetu), a URL kroz specifikaciju RFC 1738.

Opći oblik URL-a je:

```
<scheme>:<scheme-specific-part>
```

Shema predstavlja naziv protokola kojim se resurs dohvaća. Primjeri su `http`, `https`, `ftp` i drugi. Nakon znaka dvotočke dolazi ostatak čija građa ovisi o samoj shemi. Velika i mala slova u nazivu sheme trebalo bi tretirati na jednak način (`HTTP` predstavlja isto što i `http`).

Za mnoštvo različitih shema zajednički dio je sličnog oblika i taj je prikazan u nastavku.

```
//<user>:<password>@<host>:<port>/<url-path>
```

U slučaju protokola HTTP i HTTPS, ovaj oblik je proširen:

```
//<user>:<password>@<host>:<port>/<path>?<searchpart>
```

Neki od prikazanih dijelova su opcionalni i mogu se izostaviti. Evo nekoliko primjera valjanih URL-ova.

```
http://morgoth.zemris.fer.hr/nastava/opjj/
http://morgoth.zemris.fer.hr:80/nastava/opjj/book-2015-04-28.pdf
http://acme.org/images/search?theme=mountains&period=summer
http://ivo:tajna@acme.org/images/search?theme=mountains
http://ivo:tajna@acme.org/images/search?theme=mountain%20cabin#f1
```

Prvi URL definira naziv poslužitelja te stazu do objekta. Drugi URL uz naziv poslužitelja dodaje i informaciju o pristupu na kojem poslužitelj prihvaća zahtjeve. Treći URL ilustrira umetanje dodatnih parametara (dva su parametra, definicije su im razvojene znakom `&`, prvi ima ime `theme` i vrijednost `mountains` a drugi ime `period` i vrijednost `summer`). Četvrti URL prikazuje ugradnju korisničkog imena i zaporke a peti URL prikazuje kako se prenose nesigurni znakovi: ASCII vrijednost konvertira se u dvoznamenkasti heksadekadski broj i ispred toga dolazi znak postotka. Konkretno, u URL-u je kao vrijednost parametra `theme` zapisano `mountain%20cabin`, što se nakon dekodiranja pretvara u "mountain cabin"; ASCII vrijednost praznine je 32 (dekadski) odnosno 20 (heksadekadski). Peti URL ujedno pokazuje i kako se specificira fragment (iza znaka ljestve) na koji bi preglednik automatski trebao "odskrolati" prilikom otvaranja dugačke stranice.

Spomenuto kodiranje potrebno je (prema specifikaciji) primijeniti na sve znakove koji nemaju reprezentaciju u US-ASCII, na sve znakove čija je uporaba nesigurna te na sve znakove koji mijenjaju interpretaciju strukture bilo kojeg dijela URL-a. U znakove bez grafičke reprezentacije u US-ASCII spadaju svi znakovi iz raspona 00-1F, 7F te 80-FF. U nesigurne i ostale znakove spadaju " ", "<", ">", """, "#", "%", "{", "}", "|", "\", "^", "~", "[", "]", "`", ";", "/", "?", ":", "@", "=", "&".

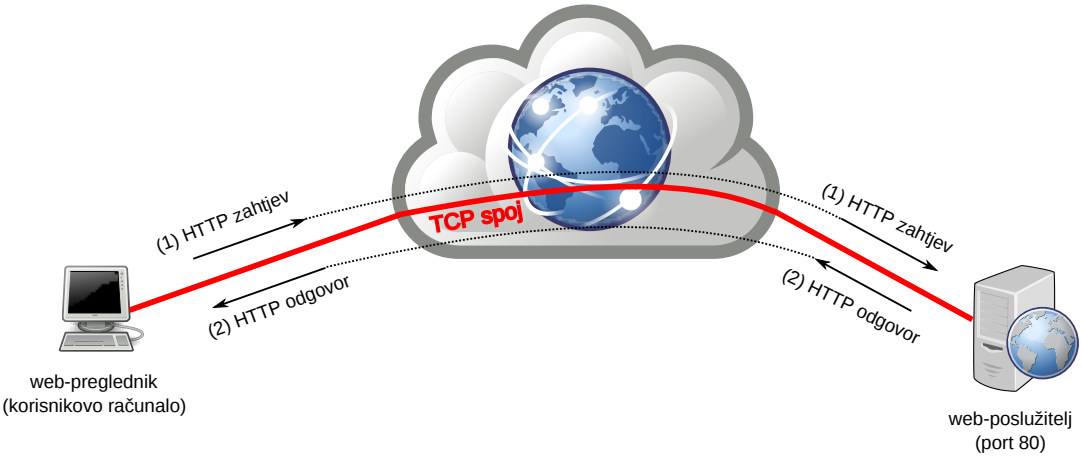
Ako poslužitelj zadužen za isporuku resursa čiji URL pišemo ne prihvaća zahtjeve na pretpostavljenom pristupu, tada se u okviru URL-a mora zadati i pristup. Pretpostavljeni pristup za protokol HTTP je 80, za protokol HTTPS 443 a za protokol FTP 21.

Ukratko o protokolu HTTP

Protokol HTTP je protokol aplikacijskog sloja ISO-OSI referentnog modela. Taj protokol za slanje svojih poruka između klijenta i poslužitelja na prijenosnom sloju ISO-OSI referentnog modela koristi protokol TCP. Povijesno je postojalo nekoliko inačica protokola HTTP, od kojih se danas još mogu susresti HTTP inačice 1.0 (polako izumire) te HTTP inačice 1.1 (danas

dominantna inačica). Protokol HTTP je protokol tipa *zahtjev-odgovor*, kako je ilustrirano na slici 20.18.

Slika 20.18. Komunikacija uporabom protokola HTTP



Kako je prikazano na slici, na poslužiteljskom računalu se nalazi pokrenut program koji koristi protokol TCP i čeka na zahtjev za uspostavom spoja na nekom pristupu (engl. *port*). Taj pristup unaprijed mora biti poznat klijentu koji će inicirati uspostavu TCP-spoja. Ovaj program zvat ćemo web-poslužiteljem. Uobičajeno, program je čitavo vrijeme pokrenut kako bi mogao opsluživati zahtjeve klijenata čim isti stignu.

Na klijentskom računalu pokrenut ćemo program koji ćemo zvati web-klijentom. Taj će program najprije zatražiti uspostavu TCP-spoja s poslužiteljskim programom. Da bi to bilo moguće, klijent unaprijed mora znati simboličko ime poslužitelja koje će u IP adresu razriješiti uporabom DNS-a (ili može odmah znati njegovu IP adresu) te pristup na kojem poslužiteljski program prihvaća zahtjeve za uspostavom TCP-spoja. Ako je sve u redu, nakon razmjene tri TCP-segmenta poslužitelj i klijent uspostavit će TCP-spoj. Postupak uspostave spoja poznat je pod nazivom *3-way handshake*. Na programskoj razini, spoj će biti uspostavljen između pristupne točke klijenta i pristupne točke poslužitelja (engleski termin za pristupnu točku je *socket*). Zbog potrebe za početnom razmjenom TCP-segmenata, postupak uspostave TCP-spoja je vremenski skupa operacija (a tijekom tog postupka klijent i poslužitelj ne mogu razmjenjivati nikakve podatke svojih programa); kada su performanse bitne, zgodno je jednom uspostavljen TCP-spoj pokušati koristiti za više zahtjeva i odgovora. Imajući to u vidu, protokol HTTP u inačici 1.1 opisuje upravo takvo ponašanje: klijentu je kroz uspostavljeni spoj dopušteno poslati jedan ili više zahtjeva koje poslužitelj prihvaća, obrađuje i kroz taj spoj vraća odgovore. Radi se o takozvanom *HTTP request and response pipelining*.

U nastavku ćemo pobliže opisati komunikaciju između klijenta i poslužitelja uporabom *pojednostavljenja* protokola HTTP inačice 1.1; naime, na nekim mjestima dat ćemo opis koji je možda jednostavniji no što to protokol omogućava, ali će svakako biti dovoljan da bismo isprobali okvirnu implementaciju. Čitatelja svakako upućujemo na dokument RFC2616 koji je službena specifikacija ovog protokola i koju treba koristiti za razrješavanje svih eventualnih nedoumica.

Protokol HTTP je tekstovni protokol. HTTP-zahtjev i HTTP-odgovor sastoje se od dva dijela: zaglavlja te tijela (koje je opcionalno). Zaglavlje se sastoji od jednog ili više redaka. Svaki redak završava slijedom znakova CR i LF (odnosno ASCII 13 te ASCII 10); kako bi se povećala robusnost, klijenti i poslužitelji obično razumiju i zahtjeve kod kojih su retci terminirani samo znakom LF (ASCII 10).

Pri opisu zaglavlja koristit ćemo se jednostavnom gramatikom koja je u cijelosti opisana u dokumentu RFC2616, o ovdje ćemo prenijeti samo dijelove koji će nam trebati. Najprije prenosimo definicije osnovnih simbola.

```
OCTET          = <any 8-bit sequence of data>
CHAR           = <any US-ASCII character (octets 0 - 127)>
```

```
UPALPHA         = <any US-ASCII uppercase letter "A".."Z">
LOALPHA         = <any US-ASCII lowercase letter "a".."z">
ALPHA           = UPALPHA | LOALPHA
DIGIT           = <any US-ASCII digit "0".."9">
CTL             = <any US-ASCII control character
                  (octets 0 - 31) and DEL (127)>
CR              = <US-ASCII CR, carriage return (13)>
LF              = <US-ASCII LF, linefeed (10)>
SP              = <US-ASCII SP, space (32)>
HT              = <US-ASCII HT, horizontal-tab (9)>
<">             = <US-ASCII double-quote mark (34)>
CRLF            = CR LF
LWS             = [CRLF] 1*( SP | HT )
TEXT            = <any OCTET except CTLs,
                  but including LWS>
```

U korištenoj gramatici, uglate zagrade označavaju opcionalni sadržaj, oble grupiranje, znak `|` izbor a znak `*` ponavljanje od nula ili više puta. Pojavljivat će se i općeniti oblik ovog operatora `n*m` gdje `n` ako nije naveden ima podrazumijevanu vrijednost 0 a `m` ako nije naveden ima podrazumijevanu vrijednost beskonačno. Primjerice, `1*` bi označavala jedno ili više pojavljivanja.

Prvi redak zaglavlja specifičnog je oblika (i u zahtjevu i u odgovoru). Nakon njega slijede retci u kojima se definiraju vrijednosti dodatnih varijabli koje se razmjenjuju. Pojednostavljeno, definicija varijable je oblika `ime: vrijednost`. Formalno, definicija jedne varijable u gramatici dana je nazavršnim simbolom `message-header`.

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
```

Prema ovome, nakon imena varijable dolazi znak dvotočke i potom opcionalan sadržaj. Taj se sadržaj sastoji od nula ili više pojava elemenata `field-content` te `LWS`.

Simbol `token` (naziv varijable) definiran je kroz sljedeća pravila.

```
token          = 1*<any CHAR except CTLs or separators>
separators     = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | SP | HT
```

Za vrijednosti varijabli ponekad će se koristiti stringovi. String je definiran na sljedeći način.

```
quoted-string  = ( <"> *(qdtext | quoted-pair ) <"> )
qdtext         = <any TEXT except <">>
quoted-pair    = "\" CHAR
```

Konačno, vrijednost koja se dodjeljuje varijabli sastoji se od elemenata `field-content` koji su definirani kao:

```
field-content  = <the OCTETs making up the field-value
                  and consisting of either *TEXT or combinations
                  of token, separators, and quoted-string>
```

HTTP-zahtjev

Pogledajmo sada jedan konkretan primjer. Pretpostavimo da smo u web-preglednik unijeli adresu (tj. URL) http://java.zemris.fer.hr/nastava/opjj/ i zatražili dohvat te stranice. Zaglavlje koje Mozilla Firefox šalje prikazano je u nastavku.

```
GET /nastava/opjj/ HTTP/1.1
Host: java.zemris.fer.hr
```

```
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:37.0)
 Gecko/20100101 Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cache-Control: max-age=0
```

Zatražimo li istu stranicu iz preglednika Google Chrome, zaglavlje koje taj preglednik šalje prikazano je u nastavku.

```
GET /nastava/opjj/ HTTP/1.1
Host: java.zemris.fer.hr
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
 image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/42.0.2311.152 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,hr;q=0.6
```

U oba primjera *za potrebe prikaza u knjizi* neki su retci slomljeni u dva retka (nastavak prethodnog retka prepoznat ćete po tome što je uvučen). U oba slučaja, zaglavlje završava slanjem jednog praznog retka. To znači da će se na razini toka okteta kraj zaglavlja prepoznati po prvoj pojavi slijeda 0D, 0A, 0D, 0A; prvi 0D, 0A terminira terminira neki prethodni redak u kojem je bilo znakova, a drugi 0D, 0A terminira sljedeći redak (koji je tada očito prazan). Prilikom čitanja zahtjeva, poslužitelj kroz spojnu točku čita znakove tako dugo dok ne pročita ovaj slijed; u tom trenutku zna da je pročitao čitavo zaglavlje pa može krenuti u "rezanje" zahtjeva u retke i analizu redaka. Konverzija pročitanih okteta u znakove radi se u skladu s kodnom stranicom US-ASCII.

Prokomentirajmo varijable koje se šalju. Prema specifikaciji HTTP inačice 1.1, klijent je poslužitelju dužan poslati varijablu `Host` čiji sadržaj mora biti ono što je u URL-u pisalo na mjestu poslužitelja (te port ako je isti bio naveden). Varijabla `Connection`, ako ima vrijednost "keep-alive" sugerira poslužitelju da po isporuci HTTP odgovora korišteni TCP-spoj ostavi otvorenim kako bi klijent kroz njega mogao poslati sljedeći zahtjev (radi se o podršci za prethodno spomenuti *pipelining*). Ako klijent želi eksplicitno onemogućiti ovaj mehanizam, u zaglavlje zahtjeva bi trebao uključiti `Connection: close`. Ako poslužitelj pri isporuci odgovora želi klijentu javiti da će nakon toga zatvoriti uspostavljeni spoj, poslužitelj bi u zaglavlje odgovoraa trebao uključiti `Connection: close`.

Varijabla `Accept` poslužitelju daje do znanja kakav oblik odgovora (tip dokumenta) je prihvatljiv. Opći oblik sadržaja ove varijable je niz zarezom razdvojenih elemenata oblika `tip;q=broj`, gdje je tip *mime-tip* formata (specifikacija koja svim formatima pridjeljuje jedinstvene oznake koje se koriste na webu, pri slanju e-mail poruka te na još mnogim drugim mjestima) te kvalifikator poželjnosti (decimalni broj iz intervala 0 do 1 gdje 0 znači apsolutno nepoželjno a 1 vrlo poželjno). Ako kvalifikator nije zadan, smatra se da je 1. Pogledajte to polje kod prethodno prikazanog zahtjeva koji je stvorio Mozilla Firefox. Traženo je da se resurs vrati kao HTML dokument (`text/html`; kako nije naveden kvalifikator, uzimamo da je 1, odnosno vrlo poželjan oblik), kao XHTML dokument (`application/xhtml+xml`, opet vrlo poželjno), kao XML dokument (`application/xml`, nešto manje poželjno, q=0.9), ili ako nije moguće u niti jednom od tih formata, onda "daj-što-daš" (`*/*`, najmanje poželjno). Ovu informaciju web-poslužitelji mogu iskoristiti kako bi odabrali u kojem formatu isporučiti traženi resurs ako ga dinamički generiraju i mogu isti sadržaj poslati u različitim oblicima).

Varijabla `Accept-Language` po istom formatu navodi popis jezika u kojima je prihvatljiv odgovor. Što se ovdje šalje, korisnik može podesiti u postavkama svojeg preglednika, a koristi se

kod web-stranica velikih tvrtki koje imaju klijente iz čitavog svijeta i koji imaju web-stranice prevedene na niz jezika. Web-poslužitelj temeljem sadržaja ove varijable može automatski odabrati koju lokaliziranu inačicu web-stranice treba poslati klijentu. Na primjeru zahtjeva Mozille Firefox kao poželjan jezik se navodi američki engleski (`en-US`) a kao manje poželjnu inačicu zajednički engleski (`en`). U primjeru zaglavlja koje šalje Google Chrome pronaći ćete i hrvatski kao prihvatljivi jezik (u skladu s postavkama kako sam ga podesio).

Varijabla `Accept-Encoding` definira transofrmacije koje poslužitelj smije primijeniti na sadržaj koji će vratiti klijentu a prije no što ga krene slati kroz TCP-spoj. Tako, primjerice, Mozilla Firefox dozvoljava da se sadržaj komprimira navedenim algoritmima kako bi u slanju njegov prijenos generirao manje podataka; Mozilla će, naravno, po primitku taj sadržaj transparentno odkomprimirati u njegov izvorni oblik prije no što ga dalje obradi. Hoće li poslužitelj to doista i napraviti ili neće, ovisi o poslužitelju; ova varijabla poslužitelju samo poručuje što smije napraviti (odnosno što klijent zna rekonstruirati).

Varijabla `User-Agent` pobliže opisuje samog klijenta. Obično se koristi od strane poslužitelja u statističke svrhe (kako bi se prikupljale informacije o udjelu različitih preglednika, operacijskih sustava na kojima se vrte ti preglednici i slično).

U zaglavlju zahtjeva Mozille Firefox još ćete pronaći varijablu `Cache-Control` postavljenu na vrijednost `max-age: 0`. Time Mozilla Firefox zahtjeva da vraćeni resurs ne smije biti stariji od 0 sekundi; na taj način eksplicitno se zahtjeva najsvježija inačica (i onemogućava vraćanje starije kopije iz keša ako takva postoji).

Prvi redak zaglavlja koje se šalje nešto je drugačijeg formata od ostatka zaglavlja koje prenosi varijable. Taj se redak sastoji od tri dijela razdvojena razmakom. Prvi dio čini zovemo *metodom*. Metodom klijent određuje što od poslužitelja očekuje da napravi s resursom. Ako je metoda `GET`, klijent zahtjeva dohvat okteta koji čine tijelo resursa. Ako je metoda `HEAD`, klijent očekuje informacije o resursu (ne očekuje se da poslužitelj pošalje i oktete koji čine tijelo resursa već samo informacije koje bi poslao da je metoda bila `GET`). Metoda `POST` koristi se kada klijent poslužitelju uz zaglavlje zahtjeva želi poslati i tijelo zahtjeva (primjerice, kada uploadate datoteku na poslužitelj). Uz ove, specifikacija definira još nekoliko metoda (`OPTIONS`, `PUT`, `DELETE`, `TRACE`, `CONNECT`) ali njih nećemo dalje obrađivati (detaljniji opis dostupan je u samoj specifikaciji).

Drugi element prvog retka je staza (i parametri) koji identificiraju traženi resurs na tom računalu. Ovaj dio obuhvaća sve znakove URL-a koji dolaze nakon naziva poslužitelja (i eventualne informacije o pristupu), a započinje prvim znakom `/`.

Konačno, treći element prvog retka je oznaka inačice protokola kojom se želi komunicirati. U prikazanim primjerima inačica je bila `HTTP/1.1`.

HTTP-odgovor

Prethodno prikazani zahhtjev klijent će (kao niz okteta) poslati kroz uspostavljeni TCP-spoj do poslužitelja. Poslužitelj će, nakon čitanja cjelovitog zahtjeva, isti pokušati izvršiti. Rezultat ovoga može biti uspjeh, pogreška ili pak dojava da treba poduzeti neku dodatnu akciju. Kada ovo utvrdi, poslužitelj generira odgovor koji se sastoji od zaglavlja odgovora te opcionalnog tijela odgovora. Primjer odgovora na prethodno opisani upit prikazan je u nastavku.

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=UTF-8
Content-Length: 5045
Date: Wed, 13 May 2015 06:13:09 GMT
Server: Apache-Coyote/1.1

<html>
<head>
<title>Directory Listing For /nastava/opjj/</title>
...
```

```
<HR size="1" noshade="noshade"><h3>Apache Tomcat/5.0.28</h3></body>

 </html>
```

I opet, zbog prostornog ograničenja, tijelo odgovora je prikazano skraćeno (što je označeno s "...").

Zaglavlje odgovora sastoji se od "posebnog" prvog retka te ostatka koji definira sadržaj iza varijabli koje se razmjenjuju. Samo zaglavlje terminirano je jednim praznim retkom (opet niz 0D, 0A, 0D, 0A) nakon čega slijede okteti koji čine tijelo traženog resursa.

Varijabla `Content-Type` spada u najvažnije varijable odgovora. Tom varijablom poslužitelj klijentu prenosi informaciju u vrsti resursa koji se vraća; zahvaljujući toj informaciji, klijent će znati treba li oktete koji čine tijelo odgovora tumačiti kao PNG-sliku, ZIP-arhivu, PDF-dokument, HTML-stranicu ili nešto drugo, i upravo temeljem te informacije klijent će znati kako postupiti s tim resursom. Primjerice, u našem slučaju klijent je bio web-preglednik, pa će on resurse tipa `text/html` tretirati kao HTML dokument koji treba prikazati, za resurse tipa `application/pdf` pokrenut će umetak (engl. *plug-in*) koji će znati prikazati taj dokument (primjerice, Acrobat Reader), za resurse tipa `application/zip` ponudit će dijalog za odabir mjesta na disku gdje treba pohraniti ZIP arhivu i slično.

Uz *mime-type* kojim je definiran format odgovora moguće je definirati i dodatne parametre. Tako je u ovom primjeru poslužitelj kao vrijednost poslao `text/html;charset=UTF-8` čime je definirao da konverziju okteta koji čine tijelo odgovora u znakove treba obavljati u skladu s UTF-8 kodnom stranicom. Uočite važan detalj: to je kodna stranica koja se koristi za rekonstrukciju *okteta odgovora*; ne samog zaglavlja odgovora. Prilikom konverzije okteta koji čine zaglavlje odgovora u znakove koristi se US-ASCII.

Varijabla `Content-Length` određuje broj okteta koji čine tijelo resursa i koji će biti poslani nakon što završi zaglavlje. Ako je resurs velik i poslužitelj pošalje ovu varijablu, klijent će u svakom trenutku tijekom čitanja okteta tijela odgovora znati koliko je posto pročitao i tu će informaciju moći prikazati i korisniku. Ako se ova varijabla ne pošalje klijentu, tada svi okteti koji slijede zaglavlje odgovora (a primljeni su do raskida TCP-spoja) čine tijelo odgovora. Uočite da slanje takvog odgovora automatski onemogućava *pipelining* novih zahtjeva.

Varijabla `Date` prenosi informaciju o vremenu na poslužitelju a varijabla `Server` klijentu prenosi informaciju o poslužitelju.

Prvi redak zaglavlja odgovora sastoji se od tri dijela (kao i kod zahtjeva); interpretacija je, međutim, drugačija. Prvi element sadrži informaciju u inačici protokola HTTP u skladu s kojom je generiran odgovor (u prikazanom primjeru to je `HTTP/1.1`). Drugi element sadrži *statusni kod* a treći dio opis statusnog koda. Statusni kod najvažniji je dio odgovora jer označava status izvođenja poslanog zahtjeva. Trenutno se za statusne kodove koriste troznamenkasti dekadski brojevi koji su podijeljeni u 5 razreda. Ukratko ćemo prokomentirati samo neke od ovih kodovi.

Kodovi koji počinju s `2` (oblika 2xx) označavaju uspješno izveden zahtjev; primjerice kod `200` će u slučaju da je metoda zahtjeva bila `GET` označavati da ćemo u tijelu odgovora dobiti oktete traženog resursa.

Kodovi koji počinju s `3` (oblik 3xx) označavaju preusmjeravanja. Tako, primjerice, ako je neki resurs na poslužitelju premješten na neku drugu lokaciju, a mi ga zatražimo na starom mjestu (i ako je poslužitelj tako konfiguriran), vraćeni odgovor imat će status 301 (*Moved Permanently*) a u zaglavlju će postojati varijabla `Location` čiji je sadržaj biti URL na kojem je resurs sada dohvatljiv. Današnji preglednici znaju automatski poslati novi zahtjev na tu adresu i dohvatiti nam traženi resurs. Status 303 namijenjen je za dojavljivanje klijentu da treba zatražiti odgovor metodom GET na drugom URL-u koji će biti poslan u varijabli `Location` a koristi se u situacijama kada je originalni zahtjev na poslužitelju pokrenuo neku skriptu koja je napravila određene promjene i koja želi da korisnik zatraži novu stranicu na kojoj će doći do traženog rezultata (koristi se u okviru *redirect-after-post* obrasca). Statusni kod 302 ima sličnu namjenu ali prema specifikaciji ne dopušta da se resurs čija je adresa vraćena preko varijable `Location` traži

drugačijom metodom od one kojom je tražen originalni resurs (što je ponašanje koje, prema napomeni u samoj specifikaciji, mnogi klijenti ne poštuju).

Kodovi koji počinju s `4` (oblika 4xx) predstavljaju greške koje su rezultat korisnika (primjerice, status 404 označava da traženi resurs na poslužitelju ne postoji; status 401 označava da korisnik mora poslati autorizacijske podatke prije no što mu se dopusti pristup resursu, 403 označava da poslužitelj odbija izvesti traženi zahtjev i slično).

Konačno, kodovi koji počinju s `5` (oblika 5xx) označavaju pogreške na poslužitelju. Primjerice, status 500 označava da je došlo do interne pogreške na poslužitelju, status 503 da traženi resurs privremeno nije dostupan (primjerice zbog nadogradnje poslužitelja) a status 505 da poslužitelj ne podržava (ili odbija podržati) verziju protokola kojom klijent želi komunicirati.

Sada možemo prikazati cjeloviti odgovor usporedno na binarnoj razini te uz interpretaciju okteta kao da su znakovi, što je prikazano u nastavku. Ideja je još jednom pokazati da je čitav odgovor naprosto niz okteta koji je stigao TCP-spojem i koji treba po prethodno opisanim pravilima rastaviti na zaglavlje i tijelo te pojedine dijelove korektno interpretirati.

```
0000  48 54 54 50 2f 31 2e 31  20 32 30 30 20 4f 4b 0d  |HTTP/1.1 200 OK.|
0010  0a 43 6f 6e 74 65 6e 74  2d 54 79 70 65 3a 20 74  |.Content-Type: t|
0020  65 78 74 2f 68 74 6d 6c  3b 63 68 61 72 73 65 74  |ext/html;charset|
0030  3d 55 54 46 2d 38 0d 0a  43 6f 6e 74 65 6e 74 2d  |=UTF-8..Content-|
0040  4c 65 6e 67 74 68 3a 20  35 30 34 35 0d 0a 44 61  |Length: 5045..Da|
0050  74 65 3a 20 57 65 64 2c  20 31 33 20 4d 61 79 20  |te: Wed, 13 May |
0060  32 30 31 35 20 30 36 3a  34 33 3a 31 36 20 47 4d  |2015 06:43:16 GM|
0070  54 0d 0a 53 65 72 76 65  72 3a 20 41 70 61 63 68  |T..Server: Apach|
0080  65 2d 43 6f 79 6f 74 65  2f 31 2e 31 0d 0a 43 6f  |e-Coyote/1.1..Co|
0090  6e 6e 65 63 74 69 6f 6e  3a 20 63 6c 6f 73 65 0d  |nnection: close.|
00a0  0a 0d 0a 3c 68 74 6d 6c  3e 0d 0a 3c 68 65 61 64  |...<html>..<head|
00b0  3e 0d 0a 3c 74 69 74 6c  65 3e 44 69 72 65 63 74  |>..<title>Direct|
00c0  6f 72 79 20 4c 69 73 74  69 6e 67 20 46 6f 72 20  |ory Listing For |
00d0  2f 6e 61 73 74 61 76 61  2f 6f 70 6a 6a 2f 3c 2f  |/nastava/opjj/</|
00e0  74 69 74 6c 65 3e 0d 0a  3c 53 54 59 4c 45 3e 3c  |title>..<STYLE><|
...
1420  6e 6f 73 68 61 64 65 22  3e 3c 68 33 3e 41 70 61  |noshade"><h3>Apa|
1430  63 68 65 20 54 6f 6d 63  61 74 2f 35 2e 30 2e 32  |che Tomcat/5.0.2|
1440  38 3c 2f 68 33 3e 3c 2f  62 6f 64 79 3e 0d 0a 3c  |8</h3></body>..<|
1450  2f 68 74 6d 6c 3e 0d 0a                           |/html>..|
```

Na ispisu je prikazano 16 okteta po retku. Skroz lijevo ispisan je pomak prvog prikazog okteta od prvog primljenog okteta odgovora, i to u heksadekadskom brojevnom sustavu. Potom slijede dvije grupe od po 8 okteta čije su vrijednosti prikazane s dvije heksadekadske znamenke, i potom skroz desno slijedi prikaz tih okteta kada ih se interpretira kao znakovi prema US-ASCII, pri čemu su znakovi koji nemaju vizualni prikaz prikazani znakom točke.

Uočite da se kao separator redaka koristi slijedi 0D, 0A (odnosno dekadski ASCII 13, ASCII 10). Zaglavlju pripadaju okteti od pozicije 0000 pa do pozicije 00A0; na pozicijama 00A1 i 00A2 nalaze se znakovi koji čine onaj prazan redak kojim se zaglavlje odvaja od tijela. Prvi oktet tijela odgovora je na poziciji 00A3 a tijelu pripadaju okteti sve do pozicije 1457 (heksadekadski). Pretvorimo li to u dekadski sustav, prvi oktet tijela je na poziciji 163 a posljenji na poziciji 5207, čime je ukupan broj okteta koji čine tijelo odgovora 5207-163+1=5045, što je upravo u skladu s vrijednošću koja je navedena u varijabli `Content-Length`.

Primjer HTTP-klijenta

Napišimo sada u Javi program koji će odglumiti jednostavnog HTTP-klijenta. Program će kao argument naredbenog retka primiti četiri argumenta: naziv poslužitelja, pristup na kojem poslužitelj sluša, stazu na poslužitelju do traženog resursa te stazu do datoteke na disku u koju će pospremiti cjelovit odgovor (i zaglavlje i tijelo odgovora). Primjerice, želimo li dohvatiti URL http://java.zemris.fer.hr/nastava/opjj/, program bismo pozvali s argumentima:

`java.zemris.fer.hr 80 /nastava/opjj/ C:\tmp\odgovor.txt`. Rješenje prikazuje primjer 20.7.

Primjer 20.7. Jednostavan program za dohvat resursa preko HTTP-a

```
 1 import java.io.BufferedOutputStream;
 2 import java.io.IOException;
 3 import java.io.OutputStream;
 4 import java.net.InetAddress;
 5 import java.net.InetSocketAddress;
 6 import java.net.Socket;
 7 import java.nio.charset.StandardCharsets;
 8 import java.nio.file.Files;
 9 import java.nio.file.Paths;
10
11
12 public class HTTPDownload {
13
14  public static void main(String[] args) throws IOException {
15   if(args.length!=4) {
16    System.out.println(
17     "Očekivao sam host port urlpath localfilepath"
18    );
19    return;
20   }
21
22   String hostname = args[0];
23   int port = Integer.parseInt(args[1]);
24   String urlpath = args[2];
25   String filepath = args[3];
26
27   OutputStream os = new BufferedOutputStream(
28    Files.newOutputStream(Paths.get(filepath))
29   );
30
31   Socket socket = new Socket();
32   socket.connect(
33    new InetSocketAddress(
34     InetAddress.getByName(hostname),
35     port
36    )
37   );
38
39   byte[] reqData = (
40    "GET "+urlpath+" HTTP/1.1\r\n"+
41    "Host: "+hostname+"\r\n"+
42    "User-Agent: Simple Java Client\r\n"+
43    "Accept: text/html,*/*;q=0.9\r\n"+
44    "Accept-Language: en-US,en;q=0.5\r\n"+
45    "Connection: close\r\n"+
46    "Cache-Control: max-age=0\r\n"+
47    "\r\n"
48   ).getBytes(StandardCharsets.US_ASCII);
49
50   socket.getOutputStream().write(reqData);
51   socket.getOutputStream().flush();
52
53   byte[] buf = new byte[1024];
```

```
54    while(true) {
55     int r = socket.getInputStream().read(buf);
56     if(r<0) break;
57     os.write(buf, 0, r);
58    }
59    os.flush();
60    os.close();
61    socket.close();
62   }
63
64 }
65
```

Program je vrlo jednostavan. Krećemo sa stvaranjem pristupne točke uporabom pretpostavljenog konstruktora (redak 31), čime se stvara nepovezana pristupna točka. Nad tom pristupnom točkom točkom tražimo pokretanje uspostave spoja metodom `connect` (retci 32 do 37). Nakon što metoda `connect` odradi uspostavu spoja, moramo pripremiti podatke koje ćemo poslati HTTP-poslužitelju; naime, HTTP-klijent prvi mora reći što želi pa u retcima 39 do 48 gradimo zaglavlje zahtjeva i zadnji prazan redak te taj tekst konvertiramo u niz okteta uporabom kodne stranice US-ASCII.

Uporabom metode `getOutputStream()` nad pristupnom točkom dohvaćamo izlazni tok okteta i u njega zapisujemo oktete pripremljenog zahtjeva (redak 50). Kako smo time riješili zahtjev, pozivamo još i metodu `flush()` (redak 51). Bez ovoga, može se dogoditi da će podatci koje smo upisali u odlazni spremnik tamo čekati još neko vrijeme prije no što ih TCP prenese poslužitelju, a time ćemo dulje čekati na odgovor.

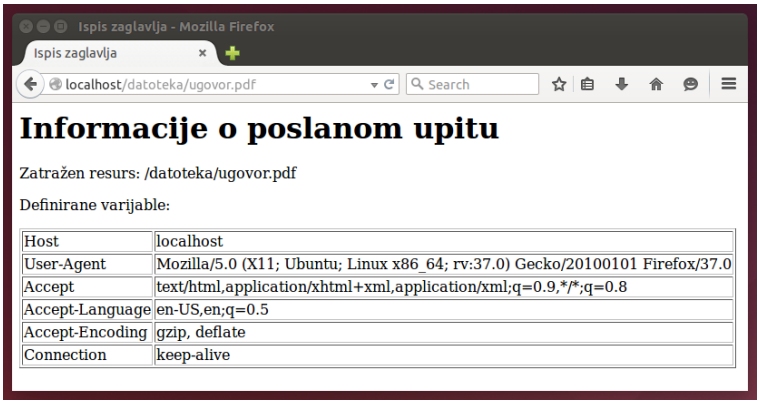
Potom u retcima 54 do 58 iz spremnika za dolazne podatke malo po malo čitamo oktete odgovora i zapisujemo ih u datoteku koju smo prethodno otvorili. U ovom slučaju ne brinemo o veličini odgovora (u smislu da smo u zaglavlju tražili varijablu `Content-Length` pa onda čitali samo još toliko okteta tijela odgovora. Razlog tomu je što smo u zahtjevu eksplicitno tražili za poslužitelj nakon slanja tijela odgovora raskine TCP-spoj (pronađite u zaglavlju zahtjeva redak `Connection: close`). Stoga čitamo oktete tako dugo dok poslužitelj ne raskine spoj i to zapisujemo u datoteku.

Jednom kada smo dobili sve oktete, zatvaramo datoteku te *obavezno* zatvaramo i pristupnu točku.

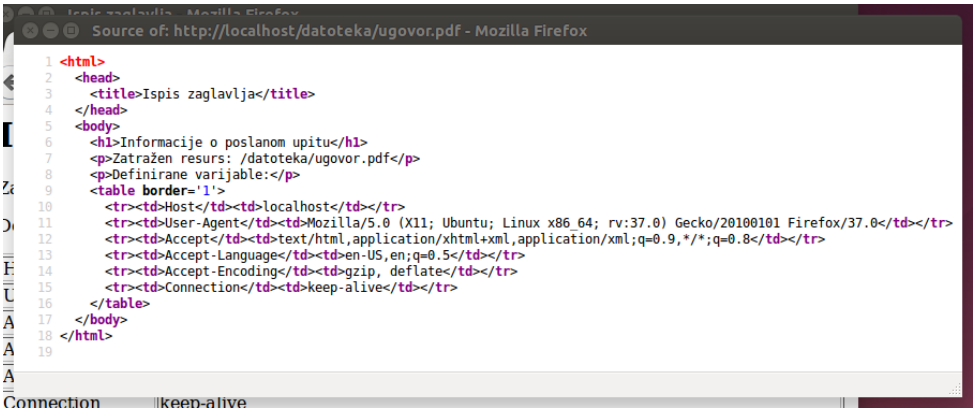
Primjer HTTP-poslužitelja

Kako bismo zaokružili ovo poglavlje, dat ćemo i prikaz jednostavnog poslužitelja za protokol HTTP. Prikazani programski kod ilustrira osnovnu građu jednodretvenog poslužitelja koji u jednom trenutku može obrađivati samo jedan zahtjev. Prikazani poslužitelj također neće dohvaćati i vraćati datoteke s diska. Umjesto toga, na svaki zahtjev on će korisniku generirati HTML dokument koji prikazuje koji je resurs bio zatražen te koja su zaglavlja bila poslana. Primjerice, spojimo li se na takav poslužitelj web-preglednikom, očekujemo rezultat kakav je prikazan na slici 20.19 (izvorni kod vraćenog HTML dokumenta prikazan je na slici 20.20).

Slika 20.19. Stranica generirana od strane našeg web-poslužitelja



Slika 20.20. Stranica generirana od strane našeg web-poslužitelja: izvorni kod



Programski kod HTTP-poslužitelja dan je u primjeru 20.8.

Primjer 20.8. Jednostavan HTTP-poslužitelj

```
 1 import java.io.BufferedInputStream;
 2 import java.io.BufferedOutputStream;
 3 import java.io.ByteArrayOutputStream;
 4 import java.io.IOException;
 5 import java.io.InputStream;
 6 import java.io.OutputStream;
 7 import java.net.InetAddress;
 8 import java.net.InetSocketAddress;
 9 import java.net.ServerSocket;
10 import java.net.Socket;
11 import java.nio.charset.StandardCharsets;
12 import java.util.ArrayList;
13 import java.util.List;
14
15
16 public class HTTPServer {
17
18  public static void main(String[] args) throws IOException {
19   if(args.length!=1) {
20    System.out.println("Očekivao sam port");
21    return;
22   }
23
```

```
24   int port = Integer.parseInt(args[0]);
25
26   ServerSocket serverSocket = new ServerSocket();
27   serverSocket.bind(
28    new InetSocketAddress((InetAddress)null, port)
29   );
30   while(true) {
31    Socket toClient = serverSocket.accept();
32    serve(toClient);
33   }
34  }
35
36  private static void serve(Socket toClient) throws IOException {
37   try {
38    InputStream cis = new BufferedInputStream(
39     toClient.getInputStream()
40    );
41    OutputStream cos = new BufferedOutputStream(
42     toClient.getOutputStream()
43    );
44
45    byte[] request = readRequest(cis);
46    if(request==null) {
47     sendError(cos, 400, "Bad request");
48     return;
49    }
50    String requestStr = new String(
51     request,
52     StandardCharsets.US_ASCII
53    );
54
55    List<String> headers = extractHeaders(requestStr);
56    String[] firstLine = headers.isEmpty() ?
57     null : headers.get(0).split(" ");
58    if(firstLine==null || firstLine.length != 3) {
59     sendError(cos, 400, "Bad request");
60     return;
61    }
62
63    String method = firstLine[0].toUpperCase();
64    if(!method.equals("GET")) {
65     sendError(cos, 405, "Method Not Allowed");
66     return;
67    }
68
69    String version = firstLine[2].toUpperCase();
70    if(!version.equals("HTTP/1.1")) {
71     sendError(cos, 505, "HTTP Version Not Supported");
72     return;
73    }
74
75    String path = firstLine[1];
76
77    composeResponse(
78     cos, path, version,
79     headers.subList(1, headers.size())
80    );
81   } finally {
```

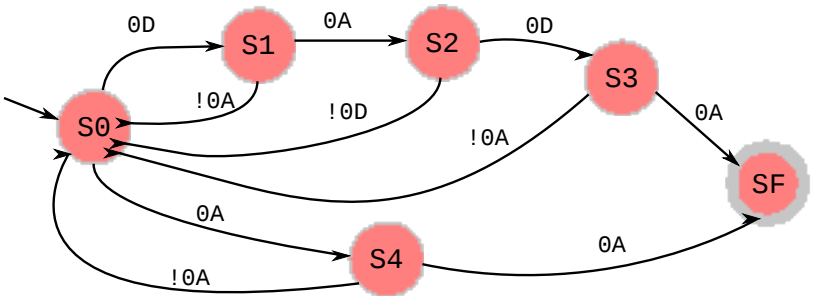
```
 82    toClient.close();
 83   }
 84  }
 85
 86  private static void composeResponse(OutputStream cos,
 87   String path, String version, List<String> headers)
 88   throws IOException {
 89
 90   StringBuilder sb = new StringBuilder(
 91    "<html>\r\n" +
 92    "  <head>\r\n" +
 93    "    <title>Ispis zaglavlja</title>\r\n"+
 94    "  </head>\r\n" +
 95    "  <body>\r\n" +
 96    "    <h1>Informacije o poslanom upitu</h1>\r\n" +
 97    "    <p>Zatražen resurs: " + path + "</p>\r\n" +
 98    "    <p>Definirane varijable:</p>\r\n" +
 99    "    <table border='1'>\r\n");
100   for(String redak : headers) {
101    int pos = redak.indexOf(':');
102    sb.append("      <tr><td>")
103     .append(redak.substring(0, pos).trim())
104     .append("</td><td>")
105     .append(redak.substring(pos+1).trim())
106     .append("</td></tr>\r\n");
107   }
108   sb.append(
109    "    </table>\r\n" +
110    "  </body>\r\n" +
111    "</html>\r\n");
112
113   byte[] tijeloOdgovora = sb.toString().getBytes(
114    StandardCharsets.UTF_8
115   );
116   byte[] zaglavljeOdgovora = (version+" 200 OK\r\n" +
117    "Server: simple java server\r\n"+
118    "Content-Type: text/html;charset=UTF-8\r\n"+
119    "Content-Length: "+tijeloOdgovora.length+"\r\n"+
120    "Connection: close\r\n"+
121    "\r\n").getBytes(StandardCharsets.US_ASCII);
122
123   cos.write(zaglavljeOdgovora);
124   cos.write(tijeloOdgovora);
125   cos.flush();
126  }
127
128  private static void sendError(OutputStream cos,
129   int statusCode, String statusText) throws IOException {
130
131   cos.write(
132    ("HTTP/1.1 "+statusCode+" "+statusText+"\r\n"+
133    "Server: simple java server\r\n"+
134    "Content-Type: text/plain;charset=UTF-8\r\n"+
135    "Content-Length: 0\r\n"+
136    "Connection: close\r\n"+
137    "\r\n").getBytes(StandardCharsets.US_ASCII)
138   );
139   cos.flush();
```

```
140
141  }
142
143  private static List<String> extractHeaders(
144   String requestHeader) {
145
146   List<String> headers = new ArrayList<String>();
147   String currentLine = null;
148   for(String s : requestHeader.split("\n")) {
149    if(s.isEmpty()) break;
150    char c = s.charAt(0);
151    if(c==9 || c==32) {
152     currentLine += s;
153    } else {
154     if(currentLine != null) {
155      headers.add(currentLine);
156     }
157     currentLine = s;
158    }
159   }
160   if(!currentLine.isEmpty()) {
161    headers.add(currentLine);
162   }
163   return headers;
164  }
165
166  private static byte[] readRequest(InputStream is)
167   throws IOException {
168
169   ByteArrayOutputStream bos = new ByteArrayOutputStream();
170   int state = 0;
171 l:  while(true) {
172    int b = is.read();
173    if(b==-1) return null;
174    if(b!=13) {
175     bos.write(b);
176    }
177    switch(state) {
178    case 0:
179     if(b==13) { state=1; } else if(b==10) state=4;
180     break;
181    case 1:
182     if(b==10) { state=2; } else state=0;
183     break;
184    case 2:
185     if(b==13) { state=3; } else state=0;
186     break;
187    case 3:
188     if(b==10) { break l; } else state=0;
189     break;
190    case 4:
191     if(b==10) { break l; } else state=0;
192     break;
193    }
194   }
195   return bos.toByteArray();
196  }
197 }
```

```
198
```

Jedini dio koji bi možda trebalo prokomentirati malo detaljnije je metoda `readRequest`. Metoda je implementirana tako da iz ulaznog toka čita oktete sve dok ne pročita ili 0D 0A 0D 0A, ili 0A 0A (kako bismo robusno obradili i klijente koji se ne drže specifikacije pa retke terminiraju samo s 0A). Metoda u pomoćni tok upisuje sve pročitane oktete (osim 0D) i to radi sve dok ne pročita jedan od navedena dva slijeda (za što koristi jednostavan automat; taj je automat prikazan na slici 20.21).

Slika 20.21. Automat za detekciju kraja zaglavlja



Metoda `extractHeaders` dobiva oktete zaglavlja i transformira ih u listu stringova. Kako specifikacija dopušta da se definicija sadržaja varijable proteže kroz više redaka (ali takvi sljedeći retci moraju započeti ili s razmaknicom ili znakom tabulatora - ASCII 32 ili ASCII 9), prikazana metoda lijepi redak sve dok ne pročita novi redak koji nije nastavak prethodnoga i tako stvoreni redak dodaje u listu.

Jednom kada imamo ovaj kostur poslužitelja, daljnje nadogradnje su jednostavne. Evo nekoliko prijedloga.

- Kod možemo nadograditi tako na tražene resurse traži kao datoteke na disku, da po ekstenziji određuje prikladan mime-tip i isporučuje takve statičke sadržaje. To po definiciji rade klasični web-poslužitelji.
- Kod možemo modificirati tako da za neke "specijalne" URL-ove (sami se dogovorimo što nam to znači, primjerice, ako počinju s `/dynamic/`) dinamički generiramo različite sadržaje (tekstovne dokumente, PDF izvještaje, ...).
- Kod možemo modificirati tako da poslužitelj obogatimo modulom za dinamičko interpretiranje skriptnih jezika, pa ako se zatraži datoteka na disku koja je skripta, klijentu se vrati rezultat izvođenja te skripte a ne njezin izvorni kod.
- Kod možemo obogatiti tako da poslužitelj podrži praćenje sjednica uporabom tehnologije *Cookies* te pohranu i dohvat podataka vezanih uz tu sjednicu.
- Kod možemo nadograditi tako da se uz metodu `GET` podrži i metoda `POST` i tako (uz još malo dorada) omogućimo slanje datoteka od klijenta na poslužitelja.

Poglavlje 21. Grafičko korisničko sučelje: uvod u Swing

Osim za izradu aplikacija koje se pokreću iz naredbenog retka i koje s korisnikom komuniciraju kroz konzolu, Java omogućava i izradu aplikacija koje s korisnikom komuniciraju kroz grafičko korisničko sučelje. U tom području, Java već godinama ima veliku prednost nad mnoštvom drugih programskih jezika: ona nudi izradu aplikacija s grafičkim korisničkim sučeljem koje radi na svim platformama na kojima postoji podrška za Javu. S Javom ne trebamo pisati posebno korisničko sučelje za program koji će se pokretati na Windowsima te drugu implementaciju za program koji će se pokretati na Linuxu.

Povijesno, Java je od samih početaka programerima jamčila još jednu odliku: na svim podržanim platformama grafičke komponente će se prikazivati na jednak način. Primjerice, određen gumb uvijek će biti prikazan upravo onako kako ga korisnik vidi na platformi na kojoj programira. Kasnije, Java je zbog ovoga došla na loš glas - često se moglo čuti: "gle, to je pisano u Javi" jer aplikacija niti na jednom operacijskom sustavu nije izgledala onako kako na tom operacijskom sustavu izgledaju aplikacije koje su pisane izravno za njega. Kako bi se doskočilo tome, u Javu je dodana tehnologija *Look and Feel* koja je razdvojila prikaz grafičkih komponenata od njihove funkcionalnosti i omogućila da se dinamički definira način na koji će se komponente prikazivati. Uz dodavanje programskog koda koji je znao obavljati crtanje na način kako to rade grafička sučelja pojedinih operacijskih sustava (i koji je distribuiran u okviru Java-platforme), grafičke su aplikacije prilikom pokretanja mogle detektirati operacijski sustav na kojem se pokreću te odabrati način prikazivanja koji je u skladu s tim operacijskim sustavom.

Od svojih začetaka pa do danas, Java-platforma donijela je tri implementacije grafičkog korisničkog sučelja. Povijesno prva implementacija došla je pod nazivom AWT (engl. *Abstract Windows Toolkit*). Ta je implementacija nudila enkapsulaciju najčešćih elemenata korisničkog sučelja u Java objekte koji su se ponašali kao zamjenski objekti (engl. *Proxy*). Programer ih je koristio za održavanje informacija o stanju komponente (položaj, veličinu, sadržaj) a sami su objekti od operacijskog sustava alocirali nativne implementacije partnerskih komponenata (takozvane *peer*-komponente) s kojima su sinkronizirale stanje. Budući da za stvaranje svake od takvih komponenata trebalo komunicirati i s operacijskim sustavom te budući da je iscrtavanje komponenata radio grafički podsustav operacijskog sustava, te su komponente bile poznate pod nazivom *teške komponente* (engl. *heavyweight components*). Takva implementacija podrške za izradu aplikacija s grafičkim korisničkim sučeljem imala je niz problema a jedan od istaknutijih je i brzina. Potpora za rad s ovom tehnologijom smještena je u paket java.awt i njegove podpakete a primjeri komponenata su razredi `Frame`, `Dialog`, `Applet`, `Label`, `Button`, `Panel` itd.

Sljedeća implementacija koja se pojavila relativno brzo nakon AWT-a izgrađena je nad AWT-om kao njegova nadogradnja i djelomična reimplementacija. Radi se o tehnologiji Swing koja je i danas dominantna tehnologija za izgradnju grafičkog korisničkog sučelja u Javi. Swing koristi takozvane lagane komponente (engl. *lightweight components*) koje ne alociraju partnerske komponente od nativnog operacijskog sustava. Zahvaljujući tome, tehnologija je sa sobom donijela i podršku za *Look and Feel* te niz drugih pogodnosti. To je tehnologija koju ćemo obraditi u ovom dijelu knjige. Potpora za rad s ovom tehnologijom smještena je u paket javax.swing i njegove podpakete a primjeri komponenata su razredi `JFrame`, `JDialog`, `JApplet`, `JLabel`, `JButton`, `JPanel` itd. Usporedite li ih s komponentama iz AWT-a, uočit ćete da Swing komponente u nazivu imaju dodano početno veliko slovo J. Analizom stabla nasljeđivanja, možemo utvrditi da je velik dio komponente u Swingu izveden iz razreda `javax.swing.JComponent` koji je pak izveden iz AWT-ovog razreda `java.awt.Container`. Neke od iznimaka su vršne komponente poput prozora i dijaloga koje su izvedene iz drugih AWT-ovih komponenata.

Konačno, u novije vrijeme (posljednjih desetak godina) Java sa sobom donosi još jednu, potpuno novu implementaciju komponenata za izgradnju grafičkog korisničkog sučelja: tehnologiju JavaFX. Prva implementacija pojavila se 2008. godine i predviđala je opisivanje grafičkog

korisničkog sučelja uporabom za to pripremljenog skriptnog jezika. Prema svim meni dostupnim spoznajama, ideja je bila vrlo loše prihvaćena. Novija inačica ove tehnologije (počev od 2.0) programerima je ponudila izravan rad s novopripremljenim komponentama dostupnim u odgovarajućim paketima. Od kasnijih inačica Jave 7 te u Javi 8 tehnologija je dostupna u okviru Java platforme (bez potrebe za dodatnim instalacijama), što će vjerojatno ostati i kroz sljedeće inačice Jave.

U nastavku ovog teksta pričat ćemo o korisničkom sučelju ostvarenom tehnologijom Swing.

Vršne komponente

Vršne komponente (engl. *top-level containers*) su osnovne komponente čija je zadaća omogućavanje prikaza korisničkog sučelja i one omogućavaju razmještanje i prikaz drugih (ne-vršnih) komponenata. Tehnologija Swing poznaje tri vršne komponente: općeniti prozor koji je modeliran razredom `javax.swing.JFrame`, dijalog koji može biti modalan a modeliran je razredom `javax.swing.JDialog` te aplet koji omogućava ugradnju Java-aplikacije u dio web-stranice i koji je modeliran razredom `javax.swing.JApplet`. Te tri komponente su jedine komponente unutar Swinga koje za aplikaciju mogu zauzeti dio površine ekrana te ponuditi mogućnost prikaza drugih komponenata (od tuda naziv "vršne komponente").

Prilikom rada s grafičkim korisničkim sučeljem tehnologije Swing, programer je dužan osigurati da sav kod izvodi posebna dretva koja je dio implementacije Swinga i koja jedina smije raditi sa Swing-komponentama. Postoje iznimke od ovog pravila i one su odgovarajuće dokumentirane. Ako iz neke druge dretve želimo pokrenuti dio koda koji je namijenjen radu sa Swingom, uobičajeno je taj kod enkapsulirati u razred koji implementira sučelje `java.lang.Runnable` i sadrži taj kod u metodi `run`, potom stvoriti primjerak tog razreda i referencu na taj primjerak poslati statičkoj metodi `javax.swing.SwingUtilities.invokeLater(runnable)` koja će zatražiti od Swingove dretve da pozove metodu `run` nad predanim objektom i time izvrši naš kod. Ova će metoda zakazati Swingovoj dretvi izvođenje te metode ali neće zablokirati tekuću dretvu koja je pozvala metodu `invokeLater`. Ako je važno osigurati da će se dretva-pozivatelj zablokirati sve dok Swingova dretva ne izvede predani kod (jer primjerice treba rezultat te obrade), tada je umjesto metode `invokeLater` potrebno pozvati metodu `invokeAndWait` (i obraditi iznimke koje ta metoda može baciti).

U istraživanje grafičkog korisničkog sučelja krenut ćemo s kodom koji iz metode `main` stvara i prikazuje novi prozor. Kako metodu `main` ne izvodi Swingova dretva, kod koji radi sa Swingom na opisani ćemo način enkapsulirati u novi razred i poslati na izvršavanje Swingovoj dretvi uporabom metode `SwingUtilities.invokeLater`. Prva implementacija takvog koda prikazana je u primjeru 21.1.

Primjer 21.1. Stvaranje prozora i prikaz na ekranu.

```
 1 package book.java.swing.p01;
 2
 3 import javax.swing.JFrame;
 4 import javax.swing.SwingUtilities;
 5 import javax.swing.WindowConstants;
 6
 7 public class OtvaranjeProzora {
 8
 9  private static class Pokretanje implements Runnable {
10   @Override
11   public void run() {
12    JFrame frame = new JFrame();
13    frame.setLocation(20, 50);
14    frame.setSize(300, 200);
15    frame.setTitle("Moj prvi prozor!");
16    frame.setDefaultCloseOperation(
17     WindowConstants.DISPOSE_ON_CLOSE
```

```
18     );
19
20     frame.setVisible(true);
21    }
22   }
23
24   public static void main(String[] args) {
25    SwingUtilities.invokeLater(new Pokretanje());
26   }
27 }
28
```

Uočite kako smo organizirali kod. Sav kod koji radi sa Swingom, enkapsuliran je u metodu `run` pomoćnog razreda `Pokretanje`. U metodi `main` stvaramo primjerak tog razreda i predajemo ga metodi `SwingUtilities.invokeLater`. Pokretanjem ćemo dobiti novi prozor, kako je to prikazano na slici 21.1.

Slika 21.1. Primjer otvaranja prozora



Umjesto smještanja u novi (imenovani) razred, čitav smo kod mogli smjestiti u metodu `main` i anonimni razred kako je prikazano u nastavku:

```
1 public class OtvaranjeProzora {
2
3  public static void main(String[] args) {
4   SwingUtilities.invokeLater(new Runnable() {
5    @Override
6    public void run() {
7     JFrame frame = new JFrame();
8     frame.setLocation(20, 50);
9     frame.setSize(300, 200);
10    frame.setTitle("Moj prvi prozor!");
11    frame.setDefaultCloseOperation(
12     WindowConstants.DISPOSE_ON_CLOSE
13    );
14
15    frame.setVisible(true);
16   }
17  });
```

```
18   }
19 }
```

ili još kraće u lambda izraz:

```
 1 public class OtvaranjeProzora {
 2
 3  public static void main(String[] args) {
 4   SwingUtilities.invokeLater(() -> {
 5    JFrame frame = new JFrame();
 6    frame.setLocation(20, 50);
 7    frame.setSize(300, 200);
 8    frame.setTitle("Moj prvi prozor!");
 9    frame.setDefaultCloseOperation(
10     WindowConstants.DISPOSE_ON_CLOSE
11    );
12
13    frame.setVisible(true);
14   });
15  }
16 }
```

Proanalizirajmo sada sam kod koji se bavi Swingom a koji je prikazan na ovom posljednjem ispisu. U retku 5 stvaramo primjerak razreda `JFrame` koji predstavlja novi prozor. Da bi se prozor mogao prikazati na ekranu, isti od operacijskog sustava treba zatražiti stvaranje nativne partnerske komponente, što u ovom trenutku još nije napravljeno. Po završetku izvođenja retka 12 jedino što smo napravili jest zazeli u memoriji mjesto za objekt tipa `JFrame` koji nam omogućava podešavanje svojstava prozora. Prozor još uvijek nije vidljiv na zaslonu.

U retcima 6 i 7 podešavamo položaj gornjeg lijevog ugla prozora (u ekranskom koordinatnom sustavu; metoda `setLocation(left,top)`) te željenu širinu i visinu prozora (metoda `setSize(width,height)`). Ova širina i visina odnosi se na cjelokupno područje koje na zaslonu zauzima prikazani prozor. Sam će prozor za prikaz korisničkog sučelja nuditi manje mjesta jer se dio troši, primjerice, na traku s naslovom prozora a na nekim operacijskim sustavima i na border koji se prikazuje uz rub prozora.

U retku 8 metodom `setTitle` podešavamo naslov koji treba prikazati u naslovnoj traci prozora.

U retcima 9-11 metodom `setDefaultCloseOperation(...)` podešavamo što će prozor učiniti kada korisnik u naslovnoj traci prozora pritisne gumb za zatvaranje prozora. Definirane su četiri konstante koje na ovom mjestu možemo koristiti.

- `WindowConstant.DO_NOTHING_ON_CLOSE`: uporabom ove konstante prozoru se nalaže da ignorira korisnikove pritiske na gumb za zatvaranje prozora. U tehnologiji AWT ovo je bila pretpostavljena postavka što je bilo izuzetno frustrirajuće za programere.

- `WindowConstant.HIDE_ON_CLOSE`: uporabom ove konstante prozoru se nalaže da se sakrije (njegova se površina više ne prikazuje na ekranu, ali prozor i dalje postoji u memoriji kao i njegova nativna partnerska komponenta). Ovo je pretpostavljena postavka za tehnologiju Swing a sa stanovišta programera pretpostavlja još gore rješenje. Naime, ako je to bio jedini prozor pokrenutog programa, korisnik njegovim skrivanjem gubi svaku mogućnost interakcije s aplikacijom no ista je i dalje pokrenuta i zauzima resurse operacijskog sustava. Jedini način da korisnik terminira takvu aplikaciju jest kroz pomoćni alat poput *Task Manager* ili naredbe **kill** operacijskog sustava Linux.

- `WindowConstant.DISPOSE_ON_CLOSE`: uporabom ove konstante prozoru se nalaže da otpusti nativnu partnersku komponentu čime otpušta nativne resurse operacijskog sustava. Posljedica je i nestanak slike prozora (prozor se više ne prikazuje). Dakako, ako program još uvijek negdje ima referencu na primjerak razreda `JFrame`, isti ostaje u memoriji i program, ako je to potrebno, pozivom metode `setVisible(true);` može zatražiti alokaciju nove

nativne partnerske komponente koja će se koristiti za prikaz prozora na ekranu. Ono što je važno naglasiti jest da će Swingova dretva završiti s izvođenjem (i nestati iz sustava) u trenutku kada detektira da više ne postoji niti jedna vršna komponenta koja ima alociranu nativnu partnersku komponentu. To će pak za posljedicu imati i gašenje samog procesa (ako u njemu više nema drugih živih nedemonskih dretvi). Za uobičajene aplikacije, ponašanje će se svesti na skrivanje prozora i nakon vrlo kratkog vremena gašenje procesa.

- `WindowConstant.EXIT_ON_CLOSE`: uporabom ove konstante prozoru se nalaže da pozove metodu `System.exit(0);` kada korisnik pritisne gumb za zatvaranje prozora. Metoda `System.exit` terminira trenutni proces neovisno o tome postoji li još koji prikazani prozor ili živa nedemonska dretva pa se uporaba ove konstante preporuča samo uz odgovarajući oprez i ako je to doista željeno ponašanje.

Spomenimo na ovom mjestu da je prozor moguće i programski sakriti. U tu svrhu prozor nudi metodu `setVisible(boolean)` koja će, ako je pozovemo uz argument `false` prozor sakriti. Želimo li prozor prisiliti da otpusti nativnu partnersku komponentu, nad prozorom možemo pozvati za to predviđenu metodu `dispose()` (otpuštanjem partnerske komponente prozor će ujedno automatski biti i skriven, ako je u trenutku poziva bio prikazan).

Kod koji smo prethodno prikazali treba tretirati samo kao ogledni kod. Prozore ne želimo stvarati na takav način jer on grubo krši enkapsulaciju. Naime, pozivatelj izvana postavlja interne parametre prozora poput položaja na kojem ga treba prikazati, širinu, visinu, naslov i slično. Bolje rješenje prikazuje primjer 21.2 i ono će dalje biti korišteno kao kostur koji ćemo nadograđivati.

Primjer 21.2. Stvaranje prozora: bolja organizacija koda

```
 1 package book.java.swing.p02;
 2
 3 import javax.swing.JFrame;
 4 import javax.swing.SwingUtilities;
 5 import javax.swing.WindowConstants;
 6
 7 public class GlavniProzor extends JFrame {
 8
 9  private static final long serialVersionUID = 1L;
10
11  public GlavniProzor() {
12   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
13   setLocation(20, 50);
14   setSize(300, 200);
15   setTitle("Moj prvi prozor!");
16   initGUI();
17  }
18
19  private void initGUI() {
20   // kod za inicijalizaciju komponenti koje prozor prikazuje
21   // ...
22  }
23
24
25  public static void main(String[] args) {
26   SwingUtilities.invokeLater(() -> {
27    new GlavniProzor().setVisible(true);
28   });
29  }
30 }
```

U ovom primjeru prozor smo modelirali razredom `GlavniProzor` koji nasljeđuje razred `JFrame`. Potom smo u njegov konstruktor stavili kod koji obavlja interna podešavanja te izvo-

đenje završava pozivom nove metode koju smo nazvali `initGUI` i za koju smo pripremili praznu implementaciju. U tu ćemo metodu dalje stavljati kod koji će u prozor dodavati korisničke komponente (gumbe, labele i sve ostalo što će prozor prikazivati).

Uočite - u ovaj kod nigdje nismo stavili poziv metode kojim će prozor biti prikazan na ekranu. Fizičko stvaranje prozora i odluku da se prozor doista želi prikazati (bez daljnjih podešavanja) prepustili smo klijentu. Taj kod prikazan je u metodi `main` koju smo prikladno smjestili u isti razred. Alternativno, metoda `main` mogla je biti i u bilo kojem drugom razredu.

Životni ciklus prozora

Nakon što je prozor stvoren, isti može prolaziti kroz niz promjena stanja: možemo ga prikazati pa ga možemo sakriti; možemo ga opet prikazati pa ga možemo minimizirati; zatim ga možemo vratiti na originalnu veličinu; možemo ga raširiti preko čitave površine zaslona (maksimizirati); možemo ga pokušati zatvoriti; možemo ga zatvoriti. Sam prozor programeru se predstavlja kao izvor takvih informacija i nudi mehanizam kojim programer može zatražiti od prozora da pri različitim promjenama stanja poziva metode koje je pripremio programer. Da bi postigao takvo ponašanje, programer kod koji želi da se takav kod poziva mora isti oblikovati u komponentu koja zadovoljava unaprijed propisano sučelje i potom mora nad prozorom registrirati tu komponentu. U ovom trenutku od interesa su nam dva sučelja: `java.awt.event.WindowListener` te `java.awt.event.WindowStateListener`.

Sučelje `java.awt.event.WindowListener` definira 7 metoda, svaka od kojih će biti pozvana kada nastupi odgovarajuća promjena nad prozorom. Promjene čije praćenje omogućava ovo sučelje su (1) otvaranje prozora - `windowOpened`, (2) klik na gumb za zatvaranje prozora - `windowClosing`, (3) zatvaranje prozora - `windowClosed`, (4) ikonifikacija prozora - `windowIconified`, (5) deikonifikacija prozora - `windowDeiconified`, (6) aktiviranje prozora - `windowActivated` te (7) deaktiviranje prozora - `windowDeactivated`. Svaka od ovih metoda prima argument tipa `java.awt.event.WindowEvent` koji sadrži dodatne informacije o događaju. Ako Vas zbunjuje razlika između (2) i (3), sjetite se da prozor može biti podešen tako da ignorira klik na gumb za zatvaranje pa nakon tog klika ne mora doista i uslijediti zatvaranje prozora.

Sučelje `java.awt.event.WindowStateListener` definira samo jednu metodu: `windowStateChanged`. Ovo sučelje po pojmom "stanje" prozora podrazumjeva: normalan prikaz, ikonificiran prikaz te maksimiziran prikaz po jednoj ili obje osi. Konstante koje definiraju ta stanja definirane su u razredu `java.awt.Frame` pa tako imamo `Frame.NORMAL = 0`, `Frame.ICONIFIED = 1`, `Frame.MAXIMIZED_HORIZ = 2`, `Frame.MAXIMIZED_VERT = 4` te `Frame.MAXIMIZED_BOTH = 2+4`. Koje je novo stanje prozora može se ispitati tako da se nad objektom tipa `WindowEvent` koji se dobije kao argument metode `windowStateChanged` pozove metoda `int getNewState()` i potom rezultat provjeri s obzirom na prethodno definirane konstante. Uočite, konstante su definirane kao bitovne zastavice. Primjerice, ako je `e` objekt tipa `WindowEvent`, da bismo provjerili je li prozor maksimiziran po horizontali, napisat ćemo `if(e.getNewState() & Frame.MAXIMIZED_HORIZ == Frame.MAXIMIZED_HORIZ) { ... }`.

Ako smo zainteresirani za dojavu ovih informacija, moramo napraviti dva koraka. Najprije moramo napisati razred koji implementira odgovarajuće sučelje i u definirane metode staviti kod koji želimo da pozove. Potom moramo stvoriti primjerak tog razreda i referencu poslati metodi za registraciju te vrste događaja koju definira sam razred. Za prethodna dva sučelja metode za registraciju su `addWindowListener` te `addWindowStateListener`.

Program prikazan u nastavku (primjer 21.3) ovo lijepo ilustrira.

Primjer 21.3. Dojava događaja vezanih uz prozor

```
1 package book.java.swing.p03;
2
3 import java.awt.event.WindowEvent;
4 import java.awt.event.WindowListener;
```

```
 5 import java.awt.event.WindowStateListener;
 6
 7 import javax.swing.JFrame;
 8 import javax.swing.SwingUtilities;
 9 import javax.swing.WindowConstants;
10
11 public class GlavniProzor extends JFrame {
12
13  private static final long serialVersionUID = 1L;
14
15  private static class ObradaOpcihDojava
16   implements WindowListener {
17
18   @Override
19   public void windowOpened(WindowEvent e) {
20    System.out.println("otvoren");
21   }
22
23   @Override
24   public void windowClosing(WindowEvent e) {
25    System.out.println("zatvara se");
26   }
27
28   @Override
29   public void windowClosed(WindowEvent e) {
30    System.out.println("zatvoren");
31   }
32
33   @Override
34   public void windowIconified(WindowEvent e) {
35    System.out.println("pretvoren u ikonu");
36   }
37
38   @Override
39   public void windowDeiconified(WindowEvent e) {
40    System.out.println("vraćen iz ikone");
41   }
42
43   @Override
44   public void windowActivated(WindowEvent e) {
45    System.out.println("aktiviran");
46   }
47
48   @Override
49   public void windowDeactivated(WindowEvent e) {
50    System.out.println("deaktiviran");
51   }
52  }
53
54  private static class ObradaPromjeneStanja
55   implements WindowStateListener {
56
57   @Override
58   public void windowStateChanged(WindowEvent e) {
59    System.out.println(e);
60   }
61
62  }
```

```
63
64  public GlavniProzor() {
65   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
66   setLocation(20, 50);
67   setSize(300, 200);
68   setTitle("Moj prvi prozor!");
69
70   addWindowListener(new ObradaOpcihDojava());
71   addWindowStateListener(new ObradaPromjeneStanja());
72
73   initGUI();
74  }
75
76  private void initGUI() {
77   // kod za inicijalizaciju komponenti koje prozor prikazuje
78   // ...
79  }
80
81  public static void main(String[] args) {
82   SwingUtilities.invokeLater(() -> {
83    new GlavniProzor().setVisible(true);
84   });
85  }
86 }
87
```

Uočite: pripremili smo dva razreda - `ObradaOpcihDojava` koji implementira `WindowListener` te `ObradaPromjeneStanja` koji implementira `WindowStateListener`. U sve propisane metode samo smo stavili jedan redak koji obavlja ispis na zaslon tako da dobijemo trag pozivanja. Konačno, u konstruktoru prozora dodali smo kod:

```
1 addWindowListener(new ObradaOpcihDojava());
2 addWindowStateListener(new ObradaPromjeneStanja());
```

čime smo objekte registrirali nad prozorom. Ako sada pokrenemo program i napravimo sljedeće: klik na neki drugi prozor, klik na naš prozor, klik na gumb za maksimizaciju, klik na gumb za povratak u normalno stanje i konačno na kraju klik na gumb za zatvaranje, dobt ćemo ispis prikazan u nastavku (dva predugačka retka ručno su prelomljena uz dodatno uvlačenje drugog retka).

```
otvoren
aktiviran
deaktiviran
aktiviran
java.awt.event.WindowEvent[WINDOW_STATE_CHANGED,opposite=null,
    oldState=0,newState=6] on frame0
java.awt.event.WindowEvent[WINDOW_STATE_CHANGED,opposite=null,
    oldState=6,newState=0] on frame0
zatvara se
deaktiviran
zatvoren
```

Ponekad će nas zanimati dojava i obrada samo jednog ili manjeg broja događaja koje propisuje sučelje. Da ne bismo u kodu morali pisati implementacije razreda koji moraju deklarirati niz metoda koje im ne trebaju (i dati im prazne implementacije), u Javinim su bibliotekama već napisani i prikladni adapteri: razredi koji implementiraju zadano sučelje tako da svim metodama daju praznu implementaciju. Tako primjerice imamo na raspolaganju razred `java.awt.event.WindowAdapter` koji, između ostaloga, implementira i oba prethodno spomenuta sučelja: `WindowListener` i `WindowStateListener` te im daje prazne imple-

mentacije. Ako nas sada samo zanima dojava da se prozor pokušava zatvoriti, sve što trebamo napraviti jest definirati novi razred koji nasljeđuje `windowClosing` i koji nadjačava metodu `windowClosing`. Niti jednu drugu metodu propisanu sučeljem ne trebamo spominjati čime dobivamo bitno pregledniji i koncizniji kod. Dodatno, takvi se razredi u kodu često deklariraju kao anonimni čime dobivamo još kraći kod.

Biste li znali sada napisati program koji će prozor zatvoriti tek kada korisnik tri puta klikne na gumb za zatvaranje? Evo ideje rješenja: podesit ćemo da prozor (sam po sebi) ništa ne radi na klik za zatvaranje. Nad istim prozorom registrirat ćemo objekt koji će pratiti klikove na gumb za zatvaranje i brojati ih. Na treći klik nad prozorom će pozvati metodu `dispose()` čime će prozor zatvoriti. Objekt ćemo modelirati kao primjerak anonimnog razreda koji nasljeđuje `WindowAdapter` i nadjačava metodu `windowClosing`. Primjer 21.4 prikazuje rješenje. Pokrenite ga i isprobajte.

Primjer 21.4. Zatvaranje prozora na treći klik gumba za zatvaranje

```
 1 package book.java.swing.p03;
 2
 3 import java.awt.event.WindowAdapter;
 4 import java.awt.event.WindowEvent;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.SwingUtilities;
 8 import javax.swing.WindowConstants;
 9
10 public class GlavniProzor extends JFrame {
11
12  private static final long serialVersionUID = 1L;
13
14  public GlavniProzor() {
15   setDefaultCloseOperation(
16    WindowConstants.DO_NOTHING_ON_CLOSE
17   );
18   setLocation(20, 50);
19   setSize(300, 200);
20   setTitle("Moj prvi prozor!");
21
22   addWindowListener(new WindowAdapter() {
23    private int brojac = 0;
24
25    @Override
26    public void windowClosing(WindowEvent e) {
27     brojac++;
28     if(brojac==3) dispose();
29    }
30   });
31
32   initGUI();
33  }
34
35  private void initGUI() {
36   // kod za inicijalizaciju komponenti koje prozor prikazuje
37   // ...
38  }
39
40
41  public static void main(String[] args) {
42   SwingUtilities.invokeLater(() -> {
```

```
43     new GlavniProzor().setVisible(true);
44    });
45   }
46 }
47
```

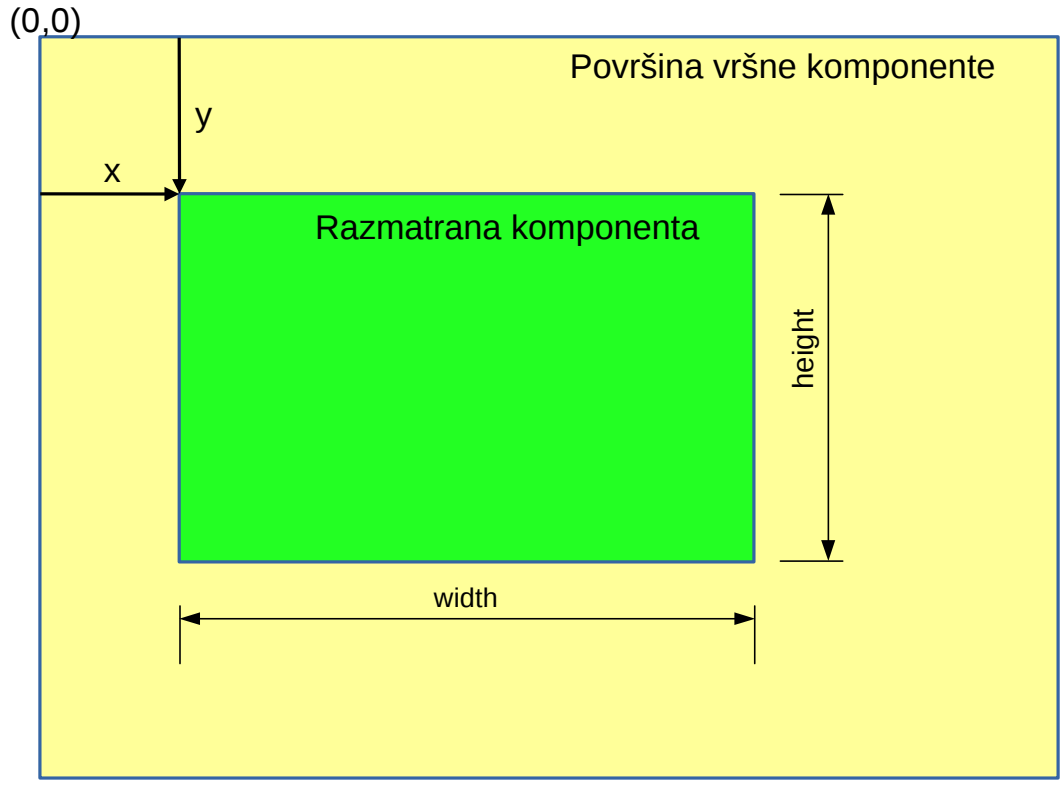
Da smo registrirani objekt modelirali statičkim ugniježđenim razredom, tada ne bismo imali direktno na raspolaganju referencu na "trenutni" prozor (nad kojim se u konačnici razrješava poziv `dispose()` u retku 28. Međutim, informaciju o prozoru kojeg se tiče prijavljeni događaj mogli bismo izvući iz primjerka razreda `WindowEvent` koji smo dobili kao argument: dovoljno bi bilo umjesto `dispose();` kod promijeniti u `e.getWindow().dispose();`.

Anatomija Swing komponente

Ne-vršne Swing-komponente izvedene su izravno ili neizravno iz razreda `javax.swing.JComponent`. Razlikujemo neinteraktivne i interaktivne komponente. Neinteraktivne komponente služe prikazu informacija ali korisnik s njima ne obavlja nikakvu interakciju. Primjer takve komponente je labela koja prikazuje statički tekst. Interaktivne komponente su komponente koje korisniku omogućavaju uređivanje informacija koje prikazuju; primjerice: kutija za unos retka teksta ili pak višeretčani uređivač teksta. Prije no što se upoznamo s postojećim Swing komponentama, razmotrit ćemo kako jedna uobičajena, za početak neinteraktivna, Swing-komponenta funkcionira.

Komponente koje ćemo prikazati u okviru ove analize izvest ćemo izravno iz razreda `javax.swing.JComponent`. Da bismo komponentu mogli učiniti vidljivom, moramo je dodati u hijerarhiju komponenata (stablo) koje na vrhu ima vršnu komponentu - vidi sliku 21.2.

Slika 21.2. Pozicioniranje komponente u roditeljskoj komponenti



Sjetite se, rekli smo da jedino vršne komponente mogu zauzeti dio ekrana na kojem će se prikazati. Smještanjem u hijerarhiju, komponenta ne mora biti izravno dodana kao dijete vršne

komponente: istu je moguće dodati u pomoćnu komponentu čiji je roditelj vršna komponenta (ili neka druga pomoćna komponenta čiji je roditelj ...). Kako bilo da bilo, jednom kada smo za komponentu definirali roditeljsku komponentu, za istu je potrebno odrediti gdje će se ona nalaziti (s obzirom na relativni koordinatni sustav roditeljske komponente koji je ekranski, odnosno gornji lijevi ugao ima koordinate 0,0; x-koordinata raste u desno a y-koordinata prema dolje).

Na slici 21.2 žutom je bojom prikazana površina roditeljske komponente. Da bismo napravili apsolutno pozicioniranje naše komponente (prikazane zelenom bojom), moramo zadati (x,y)-koordinate gornjeg lijevog ugla te širinu i visinu razmatrane komponente. Razred `JComponent` za ovo nam nudi dva načina: prvi je da nad našom komponentom pozivom metode `setLocation(x,y)` odredimo položaj gornjeg lijevog ugla te pozivom metode `setSize(width,height)` definiramo željenu širinu i visinu; drugi način je da to isto obavimo samo jednim pozivom metode `setBounds(x,y,width,height)`.

Prije no što nastavimo dalje, upoznat ćemo se s nekoliko pomoćnih razreda koji se koriste pri radu sa Swing-komponentama.

Razred `java.awt.Point` predstavlja model položaja slikovnog elementa na zaslonu. Radi se o razredu koji definira dvije javne cjelobrojne varijable tipa `int`: `x` i `y` te prikladne konstruktore. Evo primjera.

```
1 Point p = new Point(50, 100);
2 System.out.format("(%d,%d)%n", p.x, p.y);
```

Razred `java.awt.Dimension` predstavlja model širine i visine i komponente. Razred definira dvije javne cjelobrojne varijable tipa `int`: `width` i `height` te prikladne konstruktore. Evo primjera.

```
1 Dimension d = new Dimension(50, 100);
2 System.out.format("(%d,%d)%n", d.width, d.height);
```

Razred `java.awt.Rectangle` predstavlja model pravokutnog područja. Razred definira četiri javne cjelobrojne varijable tipa `int`: `x`, `y`, `width` i `height` te prikladne konstruktore i pomoćne metode. Evo primjera.

```
1 Rectangle r = new Rectangle(10, 10, 50, 100);
2 System.out.format(
3   "(%d,%d,%d,%d)%n",
4   r.x, r.y, r.width, r.height
5 );
```

U određenom smislu, `Rectangle` je ujedinjenje lokacije gornjeg lijevog ugla i širine i visine pa tako nudi i inačice konstruktora koji umjesto serije cijelih brojeva primaju reference na objekte tipa `Point` i `Dimension`.

Konačno, razred `java.awt.Insets` predstavlja model margina za pravokutno područje. Razred definira četiri javne cjelobrojne varijable tipa `int`: `left`, `right`, `top` i `bottom` te prikladne konstruktore i pomoćne metode. U konstruktoru, argumenti su redom `top`, `left`, `bottom`, `right`. Evo primjera koji definira da su vodoravne margine po 5 slikovnih elemenata a okomite po 10 slikovnih elemenata.

```
1 Insets ins = new Insets(10, 5, 10, 5);
2 System.out.format(
3   "(%d,%d,%d,%d)%n",
4   ins.top, ins.left, ins.bottom, ins.right
5 );
```

Metode kojima se obavlja apsolutno pozicioniranje komponenata dolaze i u inačicama koje primaju argumente modelirane prethodno navedenim pomoćnim razredima. Malo opširniji popis

ovih metoda koje postavljaju ili pak dohvaćaju položaj gornjeg lijevog ugla te širinu i visinu, naveden je u nastavku.

```
// Gornji lijevi ugao komponente:
public void setLocation(int x, int y);
public void setLocation(Point p);
public Point getLocation();
public Point getLocation(Point rv);
public int getX();
public int getY();

// Širina i visina komponente:
public void setSize(int width, int height);
public void setSize(Dimension d);
public Dimension getSize();
public Dimension getSize(Dimension rv);
public int getWidth();
public int getHeight();

// Čitavo područje odjednom:
public void setBounds(int x, int y, int width, int height);
public void setBounds(Rectangle r);
public Rectangle getBounds();
public Rectangle getBounds(Rectangle rv);

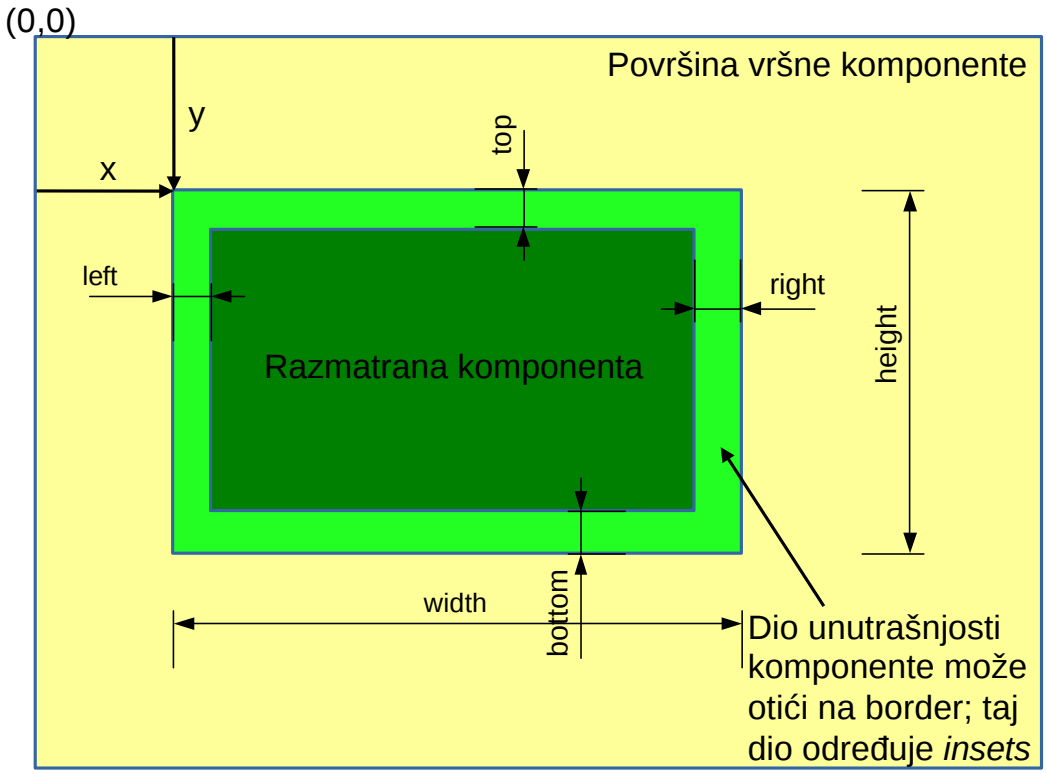
// Margine (nema izravnih setera):
public Insets getInsets();
public Insets getInsets(Insets insets);
```

Pojam margina zaslužuje dodatno pojašnjenje. Margine su dio područja komponente (pazite, ne dio izvan komponente) koji komponenta rezervira za posebne namjene. Ako margine postoje, one efektivno umanjuju raspoloživu površinu komponente na kojoj ista može prikazivati informacije korisniku. Razlog uvođenja margina jest osiguravanje potpore za rad s borderima (obrubima) koje ćemo obraditi malo kasnije. Za sada, odnos između širine i visine komponente te definiranih margina (ako postoje) prikazan je na slici 21.3. Uočite, naša komponenta i dalje je prikazana svijetlo-zelenom bojom. Aktivne margine taj prostor smanjuju pa komponenta za prikaz informacija ima na raspolaganju manji prostor koji u relativnom koordinatnom sustavu same komponente (koji ima ishodište u gornjem lijevom uglu te komponente) čini pravokutno područje (prikazano tamno zeleno) čiji je gornji lijevi ugao na položaju `(insets.left,insets.top)` i koji ima širinu `width-insets.left-insets.right` te visinu `height-insets.top-insets.bottom`.

Svim Swing-komponentama moguće je definirati i niz svojstava. Neka od njih ukratko ćemo opisati u nastavku.

Komponenta može imati definiranu boju za prikaz informacija (engl. *foreground color*) te boju za prikaz pozadine (engl. *background color*). Komponenta pri tome treba pozadinu iscrtavati samo ako je neprozirna (engl. *opaque*). Komponenta ima definiran font kojim treba ispisivati jednostavne tekstovne informacije. Komponenta može biti omogućena ili onemogućena (onemogućena komponenta ne obrađuje korisničke događaje). Također, komponenta može biti fokusabilna (primjerice, na klik miša na komponentu je moguće prebaciti fokus; ovo je važno kada komponenta želi dobivati informacije o korisnikovim pritiscima tipaka na tipkovnici - te informacije dobiva samo fokusirana komponenta i njezini roditelji).

Slika 21.3. Margine u komponenti



Izbor metoda koje su namijenjene za rad s ovim svojstvima navedene su u nastavku. Fontovi su pri tome modelirani razredom `java.awt.Font` a boje razredom `java.awt.Color` koji uz nekoliko konstruktora (primjerice onaj koji prima iznose crvene, zelene i plave boje) deklarira i čitav niz statičkih članskih varijabli koje omogućavaju izravan dohvat često korištenih boja (primjerice, `Color.RED`, `Color.YELLOW` i slično.

```
 1 // Boje:
 2 public void setForeground(Color c);
 3 public Color getForeground();
 4
 5 public void setBackground(Color c);
 6 public Color getBackground();
 7
 8 // Fontovi:
 9 public void setFont(Font f);
10 public Font getFont();
11
12 // Može li komponenta biti u fokusu:
13 public void setFocusable(boolean focusable);
14 public boolean isFocusable();
15
16 // Omogućenost komponente:
17 public void setEnabled(boolean b);
18 public boolean isEnabled();
19
20 // Neprozirnost komponente:
21 public void setOpaque(boolean isOpaque);
22 public boolean isOpaque();
```

Nakon svih ovih informacija, pojasnimo još i na koji se način obavlja prikaz (crtanje "slike") komponente. Da bismo razumjeli ovaj dio, moramo se vratiti korak unatrag (prema AWT-u)

i sagledati čitavu sliku. Razred koji u tehnologiji AWT predstavlja osnovnu komponentu je `java.awt.Component`. Iz tog razreda izveden je razred `java.awt.Container` koji predstavlja komponentu koja može sadržavati i prikazivati druge komponente. Konačno, iz razreda `java.awt.Container` izveden je razred `javax.swing.JComponent`.

Razred `java.awt.Component` za potrebe crtanja komponente definira dvije metode: `update` te `paint`. U trenutku kada je komponentu potrebno iscrtati, grafički podsustav pozvat će metodu `update` i kao argument joj poslati referencu na objekt koji je tipa `java.awt.Graphics`. Taj objekt nudi relativno bogat niz metoda za crtanje i popunjavanje linija, ovala, pravokutnika i poligona, za crtanje slika modeliranih razredom `java.awt.Image`, za ispis teksta te još niz drugih mogućnosti. U jednom trenutku razvoja Java-platforme, došlo je do potrebe da se ponudi još bogatiji niz metoda i to je učinjeno uvođenjem razreda `java.awt.Graphics2D` koji nasljeđuje razred `java.awt.Graphics`. Međutim, kako bi se očuvala kompatibilnost platforme s prethodnom inačicom, tip argumenta metoda `update` i `paint` nije mijenjan već je specifikacijom ponuđeno jamstvo da će grafički podsustav stvarati i slati objekte tipa `java.awt.Graphics2D` pa ako korisniku doista treba taj bogatiji skup mogućnosti, sve što treba je referencu na dobiveni objekt ukalupiti u ovaj tip.

U razredu `java.awt.Component` metoda `update` izravno je pozivala metodu `paint`, a ova posljednja je imala praznu implementaciju. Izvedene komponente mogle su nadjačati metodu `paint` i tamo smjestiti kod za crtanje. Okvirne implementacije tih metoda prikazane su u nastavku.

```
1 public class Component {
2
3   public void update(Graphics g) {
4     paint(g);
5   }
6
7   public void paint(Graphics g) {
8   }
9
10 }
```

Uzevši u obzir da iz njega izvedeni razred `java.awt.Container` može sadržavati djecu (druge komponente), u njemu su ove metode reimplementirane što ilustrira pseudokod u nastavku (tretirajte to kao načelnu implementaciju, ne kao stvarni kod).

```
1 public class Container extends Component {
2
3   public void update(Graphics g) {
4     if(this nije laka komponenta) {
5       //obrisi pozadinu od this:
6       g.clearRect(0,0,width,height);
7     }
8     paint(g);
9   }
10
11  public void paint(Graphics g) {
12    for(Component dijete : djeca) {
13      prilagodi g djetetovom koordinatnom sustavu
14      dijete.paint(g);
15    }
16  }
17
18 }
```

Kod ove vrste komponenata, metoda `update` ima zadaću brisanja pozadine dok metoda `paint` delegira zadatak crtanja djeci komponente.

Ovime smo došli do i Swinga koji razred `javax.swing.JComponent` izvodi iz AWT-ovog razreda `java.awt.Container`. Taj razred nanovo redefinira metode `update` i `paint`, kako je prikazano u nastavku.

```
 1 public class JComponent extends Container {
 2
 3   public void update(Graphics g) {
 4     paint(g);
 5   }
 6
 7   public void paint(Graphics g) {
 8     // dupliciraj objekt za crtanje:
 9     Graphics g2 = g.create();
10     try {
11       g2.setColor(this.getForeground());
12       g2.setFont(this.getFont());
13       g2.setClip(...);
14       paintComponent(g2);
15       paintBorder(g2);
16       paintChildren(g2);
17     } finally {
18       g2.dispose();
19     }
20   }
21
22   public void paintComponent(Graphics g) {
23     // implementacija...
24   }
25
26   public void paintBorder(Graphics g) {
27     Border b = getBorder();
28     if(b!=null) {
29       b.paintBorder(this, g, 0, 0, getWidth(), getHeight());
30     }
31   }
32
33   public void paintChildren(Graphics g) {
34     for(Component dijete : djeca) {
35       prilagodi g djetetovom koordinatnom sustavu
36       dijete.paint(g);
37     }
38   }
39
40 }
```

Najznačajnija novost jest da je postupak crtanja dalje razdijeljen u tri koraka modelirana trima novim metodama (pogledajte metodu `paint`, retke 14-16): najprije se poziva `paintComponent` čija je zadaća nacrtati površinu komponente; potom se poziva metoda `paintBorder` čija je zadaća uz obrub komponente nacrtati border, ako je isti definiran nad komponentom. Konačno, kao posljednji korak poziva se metoda `paintChildren` čija je zadaća delegirati crtanje i djeci trenutne komponente, ako ona postoje (sjetite se, razred `JComponent` je izveden iz razreda `Container` pa može imati djecu).



Važno

Kod Swing-komponenata za potrebe definiranja koda koji će obaviti crtanje površine komponente potrebno je nadjačati metodu `paintComponent`. Nemojte nadjačavati metodu `paint`!

Borderi koje smo već nekoliko puta spomenuli su objekti koji implementiraju sučelje `javax.swing.border.Border` koje je prikazano u nastavku.

```
1 public interface Border {
2     Insets getBorderInsets(Component c);
3     boolean isBorderOpaque();
4     void paintBorder(Component c, Graphics g, int x, int y,
5              int width, int height);
6 }
```

Sučelje definira samo tri metode. Metoda `getBorderInsets` mora vratiti objekt tipa `Insets` napunjen s iznosima vodoravnih i okomitih margina koje će border potrošiti pri crtanju. Metoda `isBorderOpaque` treba vratiti `true` ako border sam želi biti zadužen za čišćenje dijela pozadine komponente na kojem se prikazuje. Konačno, metoda `paintBorder` ima zadaću fizičkog crtanja bordera (iz tog razloga i dobiva kao argument objekt tipa `Graphics` kojim može crtati po površini komponente.

Border se na komponentu može instalirati pozivom metode `void setBorder(Border border);` a koji je trenutni border instaliran (ako ikoji) možemo doznati pozivom metode `Border getBorder();`.

Primjer bordera

Nakon što smo se upoznali s osnovnom arhitekturom Swing-komponente, došlo je vrijeme za konkretan kod. Evo našeg prvog zadatka. Napisat ćemo implementaciju bordera koji kroz konstruktor prima informaciju o boji bordera i debljini; border će se na komponenti prikazivati kao obrub pune linije zadane boje i debljine. Potom ćemo napisati novu komponentu (bez nadjačavanja metoda `update` i `paint`), stvoriti primjerak, postaviti mu border i na kraju samu komponentu prikazati u prozoru.

Nastavljajući se na primjer 21.2, čitava implementacija prikazana je u primjeru 21.5.

Primjer 21.5. Swing-komponenta i border

```
1 package book.java.swing.p04;
2
3 import java.awt.Color;
4 import java.awt.Component;
5 import java.awt.Container;
6 import java.awt.Graphics;
7 import java.awt.Insets;
8
9 import javax.swing.JComponent;
10 import javax.swing.JFrame;
11 import javax.swing.SwingUtilities;
12 import javax.swing.WindowConstants;
13 import javax.swing.border.Border;
14
15 public class GlavniProzor extends JFrame {
16
17  private static final long serialVersionUID = 1L;
18
19  public GlavniProzor() {
20   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
21   setLocation(20, 50);
22   setSize(300, 200);
23   setTitle("Moj prvi prozor!");
24
25   initGUI();
```

```
26  }
27
28  private void initGUI() {
29   // kod za inicijalizaciju komponenti koje prozor prikazuje
30   // ...
31   Container cp = getContentPane();
32
33   cp.setLayout(null);
34
35   Poruka poruka = new Poruka();
36   poruka.setBounds(0, 0, 100, 100);
37   poruka.setBorder(new PuniObrub(1, Color.RED));
38
39   cp.add(poruka);
40  }
41
42  private static class Poruka extends JComponent {
43   private static final long serialVersionUID = 1L;
44  }
45
46  private static class PuniObrub implements Border {
47   private int debljina;
48   private Color boja;
49
50   public PuniObrub(int debljina, Color boja) {
51    super();
52    this.debljina = debljina;
53    this.boja = boja;
54   }
55
56   @Override
57   public boolean isBorderOpaque() {
58    return false;
59   }
60
61   @Override
62   public Insets getBorderInsets(Component c) {
63    return new Insets(debljina, debljina, debljina, debljina);
64   }
65
66   @Override
67   public void paintBorder(Component c, Graphics g,
68     int x, int y, int width, int height) {
69    g.setColor(boja);
70    g.fillRect(0, 0, width, debljina);
71    g.fillRect(0, height-debljina, width, debljina);
72    g.fillRect(0, debljina, debljina, height-2*debljina);
73    g.fillRect(width-debljina, debljina,
74      debljina, height-2*debljina);
75   }
76  }
77
78  public static void main(String[] args) {
79   SwingUtilities.invokeLater(() -> {
80    new GlavniProzor().setVisible(true);
81   });
82  }
83 }
```

```
84
```

Što smo napravili? Pogledajte najprije retke 42-44: definirali smo novi privatni statički razred `Poruka` koji nasljeđuje razred `JComponent` - to je naša nova komponenta. Nismo još dodavali nikakav kod za crtanje.

Pogledajte potom retke 46-76: definirali smo novi privatni statički razred `PuniObrub` koji implementira sučelje `Border`. Definira dvije privatne članske varijable `debljina` i `boja` i njih inicijalizira kroz konstruktor. U tom razredu metoda `isBorderOpaque` vraća `false`, metoda `getBorderInsets` stvara novi objekt tipa `Insets` i sve četiri margine podešava na vrijednost određenu članskom varijablom `debljina`. Konačno, metoda `paintBorder` obavlja posao crtanja bordera. Objekt kojim se crta je objekt koji pamti stanje - jednom kada se nešto podesi, to je na snazi tako dugo dok se drugim pozivom postavka ne promijeni. Konkretno, u retku 69 podešavamo boju kojom će se crtati (i to ostaje zapamćeno). Potom popunjavamo četiri tanka pravokutnika. Prvi je uz gornji rub komponente pa mu je gornji lijevi ugao na koordinatama (0,0), širok je koliko i sama komponenta a visok onoliko kolika je željena debljina obruba. Drugi kvadrat popunjavamo uz donji rub komponente; stoga mu je položaj gornjeg lijevog ugla `(0, height-debljina)`; širina je jednaka širini komponente a visina željenoj debljini obruba. Sljedeća dva poziva popunjavaju uski pravokutnik uz lijevi rub komponente i potom uski pravokutnik uz desni rub komponente čime je čitav obrub nacrtan.

Stvaranje nove komponente i postavljanje bordera riješili smo u metodi `initGUI` kako smo prethodno nagovijestili. Pogledajte taj kod. U retku 31 od prozora metodom `getContentPane()` tražimo dohvat komponente koja predstavlja raspoloživu površinu prozora (kada od čitavog prozora odbijete dio koji se troši na prikaz naslovne trake, u nekim slučajevima glavnog izbornika te obruba prozora, ako isti postoje na grafičkom sučelju operacijskog sustava. Referencu pohranjujemo u pomoćnu varijablu `cp`. Da bismo omogućili uporabu apsolutnog pozicioniranja komponenata, sljedeći korak je nad dohvaćenom komponentom `cp` isključiti uporabu automatskog pozicioniranja pozivom metode `cp.setLayout(null);` (redak 33); više o automatskom pozicioniranju pričat ćemo nešto kasnije. I time smo spremni.

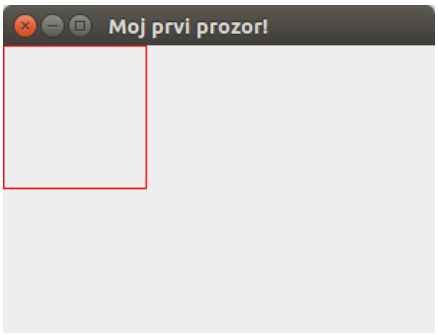
U retku 35 stvaramo jedan primjerak naše nove komponente. Time što smo stvorili komponentu, ista nije automatski prikazana. Dapače, jedini način da komponentu prikažemo je da je dodamo u hijerarhiju djece neke od vršnih komponenata. U retku 36 obavljamo apsolutno pozicioniranje komponente: njezin gornji lijevi ugao smještamo na koordinate (0,0) što je gornji lijevi ugao roditeljske komponente (u koju ćemo komponentu naknadno dodati); postavljamo širinu komponente na 100 te visinu komponente isto na 100.

U retku 37 komponenti postavljamo border: stvaramo primjerak našeg razreda `PuniObrub`, inicijaliziramo ga debljinom 1 i crvenom bojom te referencu predajemo metodi `setBorder` koja dobivenu referencu na border zapisuje u komponentu. Sjetite se: tu referencu provjerava metoda `paintBorder` i ako ista nije `null`, nad njom poziva metodu `paintBorder` koja će pokrenuti iscrtavanje bordera.

Konačno, u retku 39 komponenti `cp` u popis djece dodajemo upravo stvorenu i konfiguriranu komponentu.

Pokrenete li sada program, dobit ćete prikaz kao na slici 21.4.

Slika 21.4. Prozor s novom komponentom i borderom



Promjenom retka:

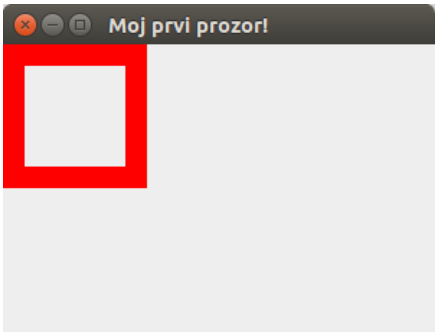
```
37 poruka.setBorder(new PuniObrub(1, Color.RED));
```

u redak:

```
37 poruka.setBorder(new PuniObrub(15, Color.RED));
```

dobit ćemo obrub debljine 15 slikovnih elemenata što je prikazano na slici 21.5. Uočite kako se border "širi" u unutrašnjost komponente jer su dimenzije komponente fiksne.

Slika 21.5. Prozor s novom komponentom i borderom (2)



Da ne biste svaki puta kada Vam zatreba prikladan border išli pisati svoju vlastitu implementaciju, razred `javax.swing.BorderFactory` nudi čitav niz metoda-tvornica za stvaranje bordera. Primjerice, border kakav smo upravo napisali naziva se linijski border i za stvaranje takvih bordera `BorderFactory` nudi čak tri statičke metode, ovisno o tome što sve nad njime želite podesiti:

```
public static Border createLineBorder(Color color);
public static Border createLineBorder(Color color, int thickness);
public static Border createLineBorder(Color color, int thickness,
    boolean rounded);
```

Sada biste redak 37 mogli zamijeniti sljedećim kodom uz isti vizualni rezultat.

```
37 poruka.setBorder(
38   BorderFactory.createLineBorder(Color.RED, 15)
39 );
```

Druga često korištena vrsta bordera je border s naslovom kojeg je moguće stvarati inačicama statičke metode `createTitledBorder`. Uz te, razred nudi još niz drugih vrsta bordera kao i kompozitne bordere koji mogu kombinirati više bordera. Za detaljnije informacije čitatelj se upućuje na službenu dokumentaciju razreda `javax.swing.BorderFactory`.

Crtanje po površini komponente

Sljedeći korak koji ćemo napraviti jest nadogradnja razreda `Poruka` tako da poštuje podešeno svojstvo neprozirnosti i podešenu pozadinsku boju. Dodatno, htjet ćemo da komponenta na svojoj površini iscrta elipsa maksimalne veličine bojom koja se može konfigurirati. Modifikacija programa prikazana je u primjeru 21.6. Dijelovi koda koji su isti su izostavljeni (uočite "...").

Primjer 21.6. Crtanje površine Swing-komponente

```
1 public class GlavniProzor extends JFrame {
2
3  ...
4
5  private void initGUI() {
6   // kod za inicijalizaciju komponenti koje prozor prikazuje
7   // ...
8   Container cp = getContentPane();
9
10  cp.setLayout(null);
11
12  Poruka poruka1 = new Poruka(Color.ORANGE);
13  poruka1.setBounds(0, 0, 130, 100);
14  poruka1.setBorder(new PuniObrub(5, Color.RED));
15  poruka1.setOpaque(true);
16  poruka1.setForeground(Color.BLACK);
17  poruka1.setBackground(Color.BLUE);
18
19  Poruka poruka2 = new Poruka(Color.ORANGE);
20  poruka2.setBounds(140, 0, 130, 100);
21  poruka2.setBorder(new PuniObrub(5, Color.RED));
22  poruka2.setOpaque(false);
23  poruka2.setForeground(Color.BLACK);
24  poruka2.setBackground(Color.BLUE);
25
26  cp.add(poruka1);
27  cp.add(poruka2);
28 }
29
30 private static class Poruka extends JComponent {
31  private static final long serialVersionUID = 1L;
32  private Color bojaKruga;
33
34  public Poruka(Color bojaKruga) {
35   super();
36   this.bojaKruga = bojaKruga;
37  }
38
39  @Override
40  protected void paintComponent(Graphics g) {
41   Dimension size = getSize();
42
43   if(isOpaque()) {
44    g.setColor(getBackground());
45    g.fillRect(0, 0, size.width, size.height);
46   }
47
48   g.setColor(bojaKruga);
49   g.fillOval(0, 0, size.width, size.height);
```

```
50    }
51   }
52
53   ...
54 }
55
```

U metodi `initGUI` kod smo izmijenili tako da stvara dva primjerka nove komponente. Za oba primjerka podešava boju za prikaz informacija (crnu), pozadinsku boju (plavu) te boju elipse (narančastu). Komponente imaju crveni obrub debljine 5. Drugu komponentu smjestili smo desno od prve komponente. Prva komponenta (lijeva) ima neprozirnost postavljenu na `true` dok je kod druge (desne) ona postavljena na `false`.

Kod za crtanje komponente prikazan je od retka 40. U kodu najprije dohvaćamo dimenzije komponente. Potom, ako je komponenta neprozirna, objekt za crtanje konfiguriramo pozadinskom bojom i pozivamo popunjavanje pravokutnika, čime površinu komponente punimo pozadniskom bojom. Nakon toga mijenjamo aktivnu boju na onu kojom je potrebno nacrtati elipsu pa pozivamo za to predviđenu metodu. Ova metoda ne prima centar elipse te željene poluosi već se zadaje pravokutnik unutar kojeg će metoda upisati maksimalnu elipsu i popuniti je.

Slika 21.6 prikazuje izgled komponenata nakon pokretanja programa.

Slika 21.6. Crtanje površine komponente



Dobivena slika očito nije dobra. Vidite li u čemu je problem? Naravno, zaboravili smo na margine. Korigirani kod prikazan je u nastavku.

Primjer 21.7. Crtanje površine Swing-komponente (2)

```
1 public class GlavniProzor extends JFrame {
2
3  ...
4
5  private static class Poruka extends JComponent {
6
7   ...
8
9   @Override
10  protected void paintComponent(Graphics g) {
11   Dimension size = getSize();
12
13   if(isOpaque()) {
14    g.setColor(getBackground());
15    g.fillRect(0, 0, size.width, size.height);
16   }
17
18   Insets ins = getInsets();
19
```

```
20     // Izračunaj iskoristivi dio komponente:
21     int x0 = ins.left;
22     int y0 = ins.top;
23     int width = size.width - ins.left - ins.right;
24     int height = size.height - ins.top - ins.bottom;
25
26     g.setColor(bojaKruga);
27     g.fillOval(x0, y0, width, height);
28    }
29   }
30
31   ...
32  }
33
```

Slika 21.7 prikazuje izgled komponenata nakon pokretanja programa.

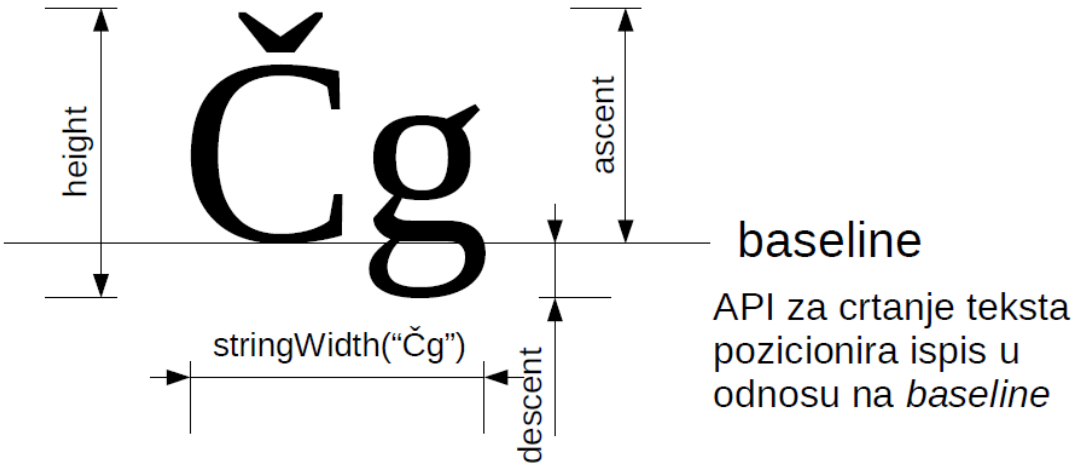
Slika 21.7. Crtanje površine komponente (2)



Da bismo dovršili primjer, modificirajmo još komponentu tako da kroz konstruktor prima i tekst koji ispisuje centrirano na svojoj površini.

Da bismo to znali napraviti, najprije moramo vidjeti što nam stoji na raspolaganju za rad s tekstom. Objekt za crtanje koji dobivamo u metodi `paintComponent` sadrži dvije korisne metode vezane uz tekst: `getFontMetrics()` vraća objekt tipa `FontMetrics` koji nudi informacije vezane uz font koji je trenutno postavljen objektu za crtanje (pozivom `setFont(font);`). Da bismo razumjeli značenje pojedinih informacija, treba razmotriti sliku 21.8.

Slika 21.8. Osnovne informacije o fontu



Prilikom prikazivanja teksta, slova se pozicioniraju tako da leže na zamišljenoj vodoravnoj crti koja je u englesnoj terminologiji poznata pod nazivom *baseline*. Neka slova (poput 'p', 'j', 'g' i 'q') imaju dio koji propada ispod te crte. Maksimalni broj slikovnih elemenata mjereno od zamišljene

crte prema gore koji može biti potrošen pri prikazu teksta razmatranim fontom dostupan je kao mjera `ascent`. Uočite u primjeru na slici da nije nužno da sva slova idu toliko u visinu: mala slova sigurno ne idu, velika slova bez dijakritika također ne idu; veliko slovo s dijakritikom (za najgori slučaj) je ono što mjeri `ascent`. Mjera `descent` mjeri slično samo od zamišljene crte prema dolje. Uobičajeno je `ascent > descent`. Suma ove dvije mjere je visina fonta (mjera `height`). Uz ove informacije koje ne ovise o konkretnom tekstu, objekt `FontMetrics` nudi metodu `stringWidth(text)` koja prima tekst i vraća informaciju o širini predanog teksta (koliko će na ekranu taj tekst zauzimati slikovnih elemenata po širini, odnosno uzduž zamišljene vodoravne crte).

Metoda `Graphics.drawString(text,x,y)` koja omogućava ispis teksta prima (x,y)-koordinate početka zamišljene vodoravne crte i od te pozicije ispisuje slova predanog teksta. Kako ćemo tom metodom postići centrirani ispis teksta? Vodoravno, treba vidjeti koliko imamo ukupno mjesta po širini a koliko ćemo potrošiti na ispis teksta; razliku želimo jednako raspodijeliti lijevo i desno od teksta pa ispis započinjeno s odmakom od pola te razlike. Razmišljanje u vodoravnom smjeru je slično. Primjer 21.8 prikazuje potrebne modifikacije.

Primjer 21.8. Crtanje površine Swing-komponente (3)

```
 1 public class GlavniProzor extends JFrame {
 2
 3  ...
 4
 5  private void initGUI() {
 6   ...
 7   Poruka poruka1 = new Poruka(Color.ORANGE, "Pozdrav!");
 8   ...
 9   Poruka poruka2 = new Poruka(Color.ORANGE, "Hello!");
10   ...
11  }
12
13  private static class Poruka extends JComponent {
14   private static final long serialVersionUID = 1L;
15   private Color bojaKruga;
16   private String tekst;
17
18   public Poruka(Color bojaKruga, String tekst) {
19    super();
20    this.bojaKruga = bojaKruga;
21    this.tekst = tekst;
22   }
23
24   @Override
25   protected void paintComponent(Graphics g) {
26
27    ...
28
29    FontMetrics fm = g.getFontMetrics();
30
31    g.setColor(getForeground());
32    g.drawString(
33     tekst,
34     x0+(width-fm.stringWidth(tekst))/2,
35     y0+height-(height-fm.getAscent())/2
36    );
37   }
38  }
39
```

```
40   ...
41 }
42
```

Slika 21.9 prikazuje izgled komponenata nakon pokretanja programa.

Slika 21.9. Crtanje površine komponente (3)



Promjena stanja komponente nakon što je prikazana

Ako komponenta nudi mehanizam promjene podataka na temelju kojih je generiran njezin prikaz (skup tih podataka nazivat ćemo *stanjem komponente*), tada je potrebno osigurati da je prikaz komponente na zaslonu u skladu s njezinim stanjem. Konkretno, to znači da grafička komponenta pri promjeni svojih podataka treba zatražiti od grafičkog podsustava da nanovo pokrene postupak crtanja čime će se osvježiti prikaz komponente. U tu svrhu komponenti na raspolaganju stoji metoda `repaint()`. Pozivanjem te metode komponenta javlja grafičkom podsustavu da bi istu u što kraćem vremenu trebalo ponovno nacrtati (i ništa više od toga). Stvarno ponovno crtanje uslijedit će u trenutku kada to odabere grafički podsustav, a razlog za takvo ponašanje je želja da se omogući da mnoštvo komponenata prilikom ažuriranja podataka u kratkom vremenskom intervalu iskaže potrebu za ponovnim iscrtavanjem te da grafički podsustav to sve riješi odjednom.

Ovo ćemo ilustrirati upravo na primjeru komponente `Poruka` kojoj ćemo dodati još i *settere* za boju i tekst. Ova modifikacija prikazana je u primjeru 21.9.

Primjer 21.9. Crtanje površine Swing-komponente (4)

```
 1 public class GlavniProzor extends JFrame {
 2
 3  ...
 4
 5  private static class Poruka extends JComponent {
 6   ...
 7
 8   public void setBojaKruga(Color bojaKruga) {
 9    this.bojaKruga = bojaKruga;
10    repaint();
11   }
12
13   public void setTekst(String tekst) {
14    this.tekst = tekst;
15    repaint();
16   }
17   ...
18  }
19
20  ...
21 }
```

22



Važno

Nemojte zaboraviti na metodu `repaint()`. Ako dobijete aplikaciju kod koje komponente misteriozno ažuriraju svoj prikaz tek kada pokušate promijeniti veličinu prozora - to je prvi simptom da imate komponente koje dozvoljavaju promjenu stanja ali ne traže od grafičkog podsustava osvježavanje slike.

Automatsko razmještanje komponenata

Nakon upoznavanja s osnovama prikaza grafičkih komponenata došlo je vrijeme da se pozabavimo načinom na koji je moguće ostvariti automatsko razmještanje komponenata koje se prilagođava trenutnoj širini i visini osnovne komponente u koju su druge komponente smještene. U tu svrhu naprije ćemo napraviti malu modifikaciju u prethodno razvijenom kodu, a potom ćemo se upoznati malo detaljnije s razredom `Container`.

Kroz prethodni tekst razvili smo grafičku komponentu `Poruka` kao privatni ugniježđeni razred unutar prozora čija je zadaća bila prikazati tu komponentu. Sada ćemo tu komponentu izvući kao javnu komponentu, smjestiti je u vlastiti paket i dati joj još jedan (jednostavniji) konstruktor koji prima samo poruku koju treba prikazati. Potreban kod prikazan je u primjeru 21.10.

Primjer 21.10. Modifikacija grafičke komponente `Poruka`

```
 1 package book.java.swing.components;
 2
 3 import java.awt.Color;
 4 import java.awt.Dimension;
 5 import java.awt.Font;
 6 import java.awt.FontMetrics;
 7 import java.awt.Graphics;
 8 import java.awt.Insets;
 9
10 import javax.swing.JComponent;
11
12 public class Poruka extends JComponent {
13  private static final long serialVersionUID = 1L;
14  private Color bojaKruga;
15  private String tekst;
16
17  public Poruka(String tekst) {
18   this(Color.ORANGE, tekst);
19  }
20
21  public Poruka(Color bojaKruga, String tekst) {
22   this(bojaKruga, tekst, null);
23  }
24
25  public Poruka(Color bojaKruga, String tekst, Color bojaPozadine) {
26   super();
27   if(bojaKruga!=null) {
28    this.bojaKruga = bojaKruga;
29   } else {
30    this.bojaKruga = Color.ORANGE;
31   }
32   this.tekst = tekst;
33   if(bojaPozadine!=null) {
34    setOpaque(true);
```

```
35     setBackground(bojaPozadine);
36    }
37   }
38
39   public void setBojaKruga(Color bojaKruga) {
40    this.bojaKruga = bojaKruga;
41    repaint();
42   }
43
44   public void setTekst(String tekst) {
45    this.tekst = tekst;
46    revalidate();
47    repaint();
48   }
49
50   @Override
51   public Dimension getPreferredSize() {
52    if(isPreferredSizeSet()) {
53     return super.getPreferredSize();
54    }
55    Font font = getFont();
56    FontMetrics fm = getFontMetrics(font);
57    Dimension dim = new Dimension();
58    dim.height = fm.getHeight() * 2;
59    if(tekst!=null) {
60     dim.width = fm.stringWidth(tekst) + 10;
61    }
62    return dim;
63   }
64
65
66   @Override
67   protected void paintComponent(Graphics g) {
68    Dimension size = getSize();
69
70    if(isOpaque()) {
71     g.setColor(getBackground());
72     g.fillRect(0, 0, size.width, size.height);
73    }
74
75    Insets ins = getInsets();
76
77    // Izračunaj iskoristivi dio komponente:
78    int x0 = ins.left;
79    int y0 = ins.top;
80    int width = size.width - ins.left - ins.right;
81    int height = size.height - ins.top - ins.bottom;
82
83    g.setColor(bojaKruga);
84    g.fillOval(x0, y0, width, height);
85
86    FontMetrics fm = g.getFontMetrics();
87
88    g.setColor(getForeground());
89    g.drawString(
90     tekst,
91     x0+(width-fm.stringWidth(tekst))/2,
92     y0+height-(height-fm.getAscent())/2
```

```
93    );
94   }
95 }
96
```

Komponentu smo promijenili na još jedan način: implementirali smo metodu `getPreferredSize` koja ako je korisnik eksplicitnim pozivom metode `setPreferredSize` komponenti nametnuo dimenzije vraća njih, a u suprotnom temeljem zadane poruke koju komponenta treba prikazati i fonta koji se za to koristi računa i vraća optimalne dimenzije uz koje će čitava poruka biti vidljiva. Dodatno, kako je sada jasno da zadana poruka ima izravni utjecaj na optimalne dimenzije komponente, metodu `setTekst` koja mijenja tekst modificirali smo tako da nakon promjene još pozove i metodu `revalidate()` koja će biti objašnjena u nastavku.

Upoznavanje s osnovnim konceptima

Razmotrimo razreda `java.awt.Container`: radi se o razredu koji je izveden iz osnovnog razreda grafičkih komponenti `java.awt.Component`. Njegova je namjena modelirati grafičke komponente koje u sebi (ili da kažemo "na sebi") mogu sadržavati druge grafičke komponente. Stoga ovaj razred definira niz metoda koje omogućavaju rad s komponentama-djecom, od kojih su neke prikazane u nastavku.

Prvi skup takvih metoda su metode koje po prvi puta definira razred `Container` a služe za dodavanje, uklanjanje i dohvat djece. Prilikom dodavanja djece (koja moraju opet biti grafičke komponente, a mogu biti i grafičke komponente koje i same mogu sadržavati druge grafičke komponente) istima možemo pridružiti i dodatni objekt (koji nazivamo *ograničenje*, engl. *constraint*) a koji će sadržavati naputak kako takvo dijete treba razmjestiti prilikom automatskog razmještanja. Primjer jednog takvog naputka bi mogao biti: neka to dijete bude smješteno uz gornji rub i razvučeno po čitavoj širini (kao što je to obično slučaj s alatnim trakama). Popis ovih metoda dan je u nastavku a iz njihova imena namjena bi im morala biti jasna. Uz svaku metodu u komentaru je napisano i u kojem je razredu ista po prvi puta deklarirana.

```
public Component add(Component comp);  //Container
public void add(Component comp, Object constraints);  //Container
public Container getParent(); // Component
public Component getComponent(int n);  //Container
public int getComponentCount();  //Container
public Component[] getComponents();  //Container
public void remove(int index);  //Container
public void remove(Component comp);   //Container
public void removeAll();   //Container
```

Prilikom dodavanja komponente djeteta u roditeljsku komponentu, komponenti-djetetu svojstvo `parent` bit će automatski postavljeno tako da pokazuje na roditeljsku komponentu (uočite da je povratni tip metode za dohvat tog svojstva `Container` jer samo komponente tog tipa mogu imati djecu).

Uz pretpostavku da su djeca razmještena po površini komponente, sljedeći blok metoda omogućava dohvat istih na temelju zadanih koordinata. Primjerice, možemo se pitati: koja se komponenta iscrtava na području koordinata (x,y). Razlika između *get* i *find*-inačice je u tome što prva gleda samo izravnu djecu a druga, ako je komponenta na zadanom području tipa `Container`, rekurzivno ulazi u nju i traži osnovnu komponentu.

```
public Component findComponentAt(Point p);  //Container
public Component findComponentAt(int x, int y);  //Container
public Component getComponentAt(Point p);  //Component
public Component getComponentAt(int x, int y);  //Component
```

Kako bi se omogućili različiti načini automatskog razmještanja djece po površini osnovne komponente, ista izravno nema nikakvu podršku za razmještanje. Umjesto toga, a u skladu s obli-

kovnim obrascem Strategija, objekti čija je zadaća razmještanje komponenata modelirani su sučeljem `java.awt.LayoutManager` a razred `Container` nudi metode za postavljanje konkretnog objekta koji će obaviti taj posao (konkretne strategije) te za njegov dohvat, kako je prikazano u nastavku. Samo sučelje obradit ćemo uskoro.

```
public LayoutManager getLayout();  //Container
public void setLayout(LayoutManager mgr);  //Container
```

Osnovni razred `Component` već nudi nekoliko informacija koje bilo koja komponenta treba ponuditi svojim klijentima. U ovom kontekstu zanimljive su nam metode `getPreferredSize`, `getMinimumSize` i `getMaximumSize` kojima komponenta može dojaviti kojih bi dimenzija ona htjela biti u idealnom slučaju (a da bi mogla prikazati sve informacije koje treba prikazati), koje su dimenzije ispod kojih ne bi trebalo ići te koje su maksimalne dimenzije iznad kojih ne bi trebalo ići. Razred `Container` na ovaj je popis još dodao metodu `getInsets` kojima komponenta može dojaviti kolike trebaju biti margine prema unutrašnjosti površine komponente (i u tom prostoru ne bi trebalo prikazivati nikakve informacije, ili u slučaju da je komponenta tipa `Container`, tu ne bi trebalo raspoređivati djecu). Konačno, izravno u razredu `Component` definirana je i metoda `doLayout` čijim se pozivom pokreće postupak razmještanja djece po površini osnovne komponente (razmislite li malo, "na prvu loptu" mogli biste zaključiti da je vrlo čudno što je ova metoda deklarirana već na razini razreda `Component` koji ne poznaje pojam djece niti objekte tipa `LayoutManager` a ne tek na razini razreda `Container`; međutim, ovakvo rješenje uvelike pojednostavljuje neke od metoda jer ih ne sili da razlikuju komponente koje nisu `Container` od onih koje jesu, i ono je u skladu s oblikovnim obrascem *Kompozit*). Iako je metoda `doLayout` javna, korisnici je izravno ne bi trebali pozivati.

```
public Insets getInsets(); // Container
public Dimension getPreferredSize(); // Component
public Dimension getMinimumSize(); // Component
public Dimension getMaximumSize(); // Component
public void doLayout(); // Component
```

Uočite da su metode `getPreferredSize`, `getMinimumSize` i `getMaximumSize` u `Container` preuzete iz razreda `Component` što znači da i svaka komponenta koja sadrži i razmješta djecu po svojoj površini (dakle, `Container`) mora ponuditi njihovu smislenu implementaciju. Nekako je za očekivati da će `getPreferredSize` nad takvom komponentom na neki način pitati svoju djecu koje su im optimalne dimenzije i da će potom tu informaciju iskombinirati s algoritmom razmještanja djece i na temelju toga zaključiti i vratiti koje su optimalne dimenzije takve komponente a uz koje će i djeca koju komponenta prikazuje imati zadovoljene optimalne dimenzije prikaza (ako je to u skladu s algoritmom razmještanja). Kako je ova informacija u cijelosti ovisna o konkretnom algoritmu razmještanja čija je provedba delegirana objektima tipa `LayoutManager`, istima je prepuštena i implementacija ove funkcionalnosti. Stoga su u razredu `Container` ove metode implementirane tako da vrate dimenzije koje je korisnik izravno zadao pozivom metoda `setPreferredSize`, `setMinimumSize` te `setMaksimumSize`, odnosno ako korisnik to nije učinio, tada se upit prosljeđuje postavljenom objektu tipa `LayoutManager` koji će to izračunati.

S obzirom da razmještanje komponenata može biti računski dosta zahtjevno, razredi `Component` i `Container` napisani su tako da taj posao odrađuju što je moguće rjeđe. Da bi to bilo moguće, razred `Component` definira svojstvo `valid` koje je za komponentu postavljeno na vrijednost `true` ako je ista korektno pozicionirana, ima korektnu veličinu te ako ima djece, sva su joj djeca također takva da im je `valid` postavljen na `true`. Za dohvat vrijednosti ovog svojstva predviđena je metoda `isValid`. Izravne metode za postavljanje ovog svojstva nema. Naime, ako se nešto promijenilo u komponenti zbog čega njezin položaj i/ili veličina više nisu adekvatne pa ih treba promijeniti - to ima implikacije i na roditelja te komponente (a to potencijalno na njegovog roditelja, itd). Stoga komponenta nudi metodu `invalidate` kojom se svojstvo `valid` postavlja na `false` u toj komponenti i svim njezinim roditeljima. Iznimku u ovom postupku postavljanja čine komponente koje su tipa `Container` i koje imaju svojstvo `validateRoot` postavljeno na `true`: one prekidaju ovaj postupak. Primjer takve komponente je dekorator `JScrollPane` koji može sadržavati djecu a koji je i sam razmješten u neku vršnu

komponentu. U slučaju da njegovo dijete promijeni dimenzije, komponenta `JScrollPane` neće mijenjati svoje dimenzije već će oko prikazanog djeteta dodati trake za skrolanje.

Metoda `validate` je javna metoda koju treba pozvati kada se želi nanovo pokrenuti postupak razmještanja. Ako je u komponenti svojstvo `valid` postavljeno na `true`, ovaj poziv ne radi ništa. U suprotnom, pokreće se postupak razmještanja u komponenti (interno se poziva metoda `doLayout` koju smo prethodno spomenuli a koja dalje taj posao delegira postavljenom objektu tipa `LayoutManager`) i ako je komponenta tipa `Container`, rekurzivno u djeci; svojstvo `valid` postavlja se na `true`.

Prilikom pokretanja postupka razmještanja koji je uzrokovan promjenom dimenzija neke od komponenata u hijerarhiji djece postupak (odnosno metodu `validate`) treba pozvati ne nad djetetom-uzročnikom već nad komponentom koja se u hijerarhiji nalazi negdje iznad njega i koja je ili vršna (nema roditelja) ili je označena kao `validateRoot`. Da bi se izbjegla potreba da samostalno pišemo kod koji radi ovu pretragu, definirana je i pomoćna metoda `revalidate`. Ona najprije "konceptualno" nad komponentom (i roditeljima prema gore do prve komponente koja je `validateRoot`) postavlja `valid` na `true`) i potom nad tom vršnom komponentom poziva `validate`.

```
public boolean isValid();  // Component
public void invalidate(); // Component
public void validate(); // Component
public boolean isValidateRoot();  // Container
public void revalidate(); // Component
```

Konačno, moramo se još upoznati i sa sučeljem `LayoutManager` koje opisuje objekte zadužene za implementaciju različitih algoritama automatskog razmještanja. Ove objekte zvat ćemo *upravljačima razmještaja*. Sučelje je prikazano u nastavku.

```
package java.awt;

public interface LayoutManager {
  void addLayoutComponent(String name, Component comp);
  void removeLayoutComponent(Component comp);
  Dimension preferredLayoutSize(Container parent);
  Dimension minimumLayoutSize(Container parent);
  void layoutContainer(Container parent);
}
```

Sučelje definira 5 metoda. Sjetite se da smo prethodno pokazali da se djeca u roditeljsku komponentu mogu dodavati bez ograničenja te s ograničenjima. Sama grafička komponenta nema mehanizma kojim bi mogla pamtiti predana ograničenja (i ona ih ignorira; samo ažurira popis djece). Međutim, ako roditelj ima instaliran upravljač razmještaja (pozivom metode `setLayout(x)`, tada će svaki poziv dodavanja djece koji kao argument dobije ograničenje koje je tipa `String` rezultirati i pozivom metode `addLayoutComponent` nad instaliranim upravljačem razmještaja gdje će to ograničenje biti predano kao prvi argument a dijete kao drugi argument. Na tom mjestu upravljač razmještaja može u svojim internim podatkovnim strukturama zapamtiti i dijete i ograničenje. Kada se nad roditeljskom komponentom pozove uklanjanje djeteta, taj će poziv također biti delegiran instaliranom upravljaču razmještaja pozivom metode `removeLayoutComponent` (gdje se onda interne podatkovne strukture mogu očistiti od viška zapamćenih podataka).

Metode `preferredLayoutSize` i `minimumLayoutSize` moraju biti implementirane tako da vraćaju optimalne dimenzije odnosno minimalne dimenzije roditeljske komponente uz koje će upravljač razmještaja moći napraviti razmještanje komponenata u skladu s algoritmom razmještanja i ograničenjima uz koja su djeca dodana u roditeljsku komponentu.

Konačno, metoda `layoutContainer` je metoda u kojoj se provodi fizičko razmještanje komponenata. U ovoj metodi nad roditeljskom komponentom koja je predana kao jedini argument

dohvaća se popis djece, za njih se računa položaj i veličina te se nad svakim djetetom na kraju poziva ili kombinacija metoda setLocation i setSize, ili setBounds. Ova se metoda poziva iz grafičke komponente (roditelja) iz njegove metode doLayout kao odgovor na poziv metode validate odnosno revalidate.

U teoriji, upravljači razmještaja mogu se dijeliti između više roditeljskih komponenata jer svaka od relevantnih metoda kao argument dobiva referencu na roditelja za kojeg je potrebno izračunati dimenzije ili napraviti razmještaj. U praksi, to se uobičajeno ne radi već se za svaku roditeljsku komponentu stvara i instalira novi primjerak upravljača razmještaja. U slučaju da se djeca u roditeljsku komponentu dodaju uz ograničenja, da bi upravljač razmještaja bio o tome informiran (i da ograničenja ne bi propala) nužno je upravljača ograničenjima stvoriti i instalirati u roditeljsku komponentu prije dodavanja prvog djeteta. Stoga se to radi ili kroz konstruktor roditeljske komponente ili kao prvu sljedeću naredbu izravnim pozivom metode setLayout.

Funkcionalnost koju nudi sučelje LayoutManager u jednom je trenutku proširena izvedenim sučeljem LayoutManager2 a koje je prikazano u nastavku.

```
public interface LayoutManager2 extends LayoutManager {
    void addLayoutComponent(Component comp, Object constraints);
    public Dimension maximumLayoutSize(Container target);
    public float getLayoutAlignmentX(Container target);
    public float getLayoutAlignmentY(Container target);
    public void invalidateLayout(Container target);
}
```

Kroz ovo sučelje upravljač razmještaja može dobivati informacije o dodavanju djece s proizvoljnim ograničenjima (ograničenja ne moraju biti tipa String). Također, skup informacija koje upravljač ograničenjima može vratiti klijentu proširen je s tri metode: dohvat maksimalnih dimenzija roditeljske komponente te dohvat željenog vodoravnog i okomitog poravnanja. Konačno, dodana je i metoda invalidateLayout kroz koju upravljač ograničenjima dobiva dojavu da je nad roditeljem pozvana metoda invalidate jer je došlo do promjene u djeci. Ovo je vrlo praktično jer sada upravljač razmještajem može pamtiti sve izračune i raditi ih samo pri prvom pozivu i nakon toga pozivatelju vraćati prethodno izračunate informacije - tako dugo dok se ne pozove metoda invalidateLayout. U tom trenutku upravljač se resetira i prvi puta kada bude potrebno nanovo kreće u izračun traženih informacija.

Ilustrativni primjeri: StretchLayout i HorizontalSplitLayout

Kako radi upravljač razmještajem najbolje ćemo ilustrirati tako da napišemo dva konkretna primjera. Jedan koji ne prima nikakva ograničenja te drugi koji za djecu prima i ograničenja.

Upravljač razmještajem StretchLayout

Prvi upravljač razmještajem koji ćemo razraditi omogućavat će automatsko razmještanje jednog djeteta. Zadaća ovog upravljača bit će osigurati da se pri svakoj promjeni veličine roditeljske komponente dijetetu korigira pozicija, širina i visina tako da se dijete uvijek proteže po čitavoj površini roditeljske komponente. U konstruktoru, upravljač će primiti objekt tipa Insets kojim je upravljaču moguće dodatno zadati kolike margine je potrebno ostaviti u roditeljskoj komponenti sa svake strane (dodatno na margine koje već postoje u samom roditelju i koje se koriste u roditelju za prikaz bordera).

Što točno želimo postići, prikazuju slike 21.10 i 21.11. Slika 21.10 prikazuje izgled komponente koju smo dodali u roditelja pri čemu je upravljač stvoren uz margine veličine 0:

```
cp.setLayout(new StretchLayout(new Insets(0, 0, 0, 0)));
Poruka poruka = ...
cp.add(poruka);
```

a slika 21.11 izgled komponente koju smo dodali u roditelja pri čemu je upravljač stvoren uz margine gore=10, lijevo=20, dolje=40, desno=80:

```
cp.setLayout(new StretchLayout(new Insets(10, 20, 40, 80)));
Poruka poruka = ...
cp.add(poruka);
```

Uočite kako su pri promjeni veličine prozora margine i dalje očuvane.

Slika 21.10. StretchLayout na djelu (1)



a) originalna veličina prozora

b) nakon promjene veličine prozora i dimenzije komponente su ažurirane

Slika 21.11. StretchLayout na djelu (2)



a) originalna veličina prozora

b) nakon promjene veličine prozora i dimenzije komponente su ažurirane

Izvorni kod upravljača razmještajem StretchLayout prikazan je u nastavku (primjer 21.11).

Primjer 21.11. Upravljač razmještajem StretchLayout

```
 1 package book.java.swing.layouts;
 2
 3 import java.awt.Component;
 4 import java.awt.Container;
 5 import java.awt.Dimension;
 6 import java.awt.Insets;
 7 import java.awt.LayoutManager;
 8
 9 public class StretchLayout implements LayoutManager {
10
11  private Insets margins;
12
13  public StretchLayout() {
14   this(new Insets(0,0,0,0));
```

```
15  }
16
17  public StretchLayout(Insets margins) {
18   super();
19   this.margins = margins;
20  }
21
22  @Override
23  public void addLayoutComponent(String name, Component comp) {
24   System.out.println("x"+name);
25  }
26
27  @Override
28  public void removeLayoutComponent(Component comp) {
29  }
30
31  @Override
32  public Dimension preferredLayoutSize(Container parent) {
33   Component c = findComponent(parent);
34   return determineSize(parent, c,
35     c==null ? null : c.getPreferredSize());
36  }
37
38  @Override
39  public Dimension minimumLayoutSize(Container parent) {
40   Component c = findComponent(parent);
41   return determineSize(parent, c,
42     c==null ? null : c.getMinimumSize());
43  }
44
45  @Override
46  public void layoutContainer(Container parent) {
47   Component c = findComponent(parent);
48   if(c==null) {
49    return;
50   }
51   Insets parentInsets = parent.getInsets();
52   int w = parent.getWidth()
53    - margins.left - margins.right
54    - parentInsets.left - parentInsets.right;
55   if(w<0) w=0;
56   int h = parent.getHeight()
57    - margins.top - margins.bottom
58    - parentInsets.top - parentInsets.bottom;
59   if(h<0) h=0;
60   c.setBounds(
61    margins.left + parentInsets.left,
62    margins.top + parentInsets.top,
63    w,
64    h
65   );
66  }
67
68  private Component findComponent(Container parent) {
69   Component c = null;
70   if(parent.getComponentCount()==1) {
71    c = parent.getComponent(0);
72   }
```

```
73   return c;
74  }
75
76  private Dimension determineSize(Container parent, Component c,
77    Dimension dim) {
78
79   int w = margins.left + margins.right;
80   int h = margins.top + margins.bottom;
81   if(dim!=null) {
82    w += dim.width;
83    h += dim.height;
84   }
85   Insets parentInsets = parent.getInsets();
86   w += parentInsets.left + parentInsets.right;
87   h += parentInsets.top + parentInsets.bottom;
88   return new Dimension(w,h);
89  }
90 }
91
```

Popratni demonstracijski program prikazan je u primjeru 21.11.

Primjer 21.12. Demonstracijski program za `StretchLayout`

```
 1 package book.java.swing.layouts;
 2
 3 import java.awt.Color;
 4 import java.awt.Container;
 5 import java.awt.Insets;
 6
 7 import javax.swing.JFrame;
 8 import javax.swing.SwingUtilities;
 9 import javax.swing.WindowConstants;
10
11 import book.java.swing.components.Poruka;
12
13 public class LayoutDemo1 extends JFrame {
14
15  private static final long serialVersionUID = 1L;
16
17  public LayoutDemo1() {
18   setTitle("Automatsko razmještanje komponenata");
19   setSize(300, 300);
20   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
21   initGUI();
22  }
23
24  private void initGUI() {
25   Container cp = getContentPane();
26   cp.setLayout(new StretchLayout(new Insets(10, 20, 40, 80)));
27   Poruka poruka = new Poruka("Tekst");
28   poruka.setOpaque(true);
29   poruka.setBackground(Color.YELLOW);
30   cp.add(poruka);
31  }
32
33  public static void main(String[] args) {
34   SwingUtilities.invokeLater(()->{
```

```
35     new LayoutDemo1().setVisible(true);
36    });
37   }
38 }
39
```

Proanalizirajmo primjer 21.11. Razred definira dva konstruktor: jedan koji margine automatski postavlja na 0 te drugi koji prima željene dodatne margine. Iste se pamte u privatnoj članskoj varijabli `margins` (redak 11).

Kako ovaj upravljač ne podržava nikakva ograničenja, metode kojima mu se dojavljuje dodavanje i uklanjanje komponenata iz roditeljskog razreda ostavili smo praznima. Zahvaljujući tome, jedan primjerak ovog upravljača mogli bismo koristiti za više roditeljskih komponenti.

Zadaća metode `prefferedLayoutSize(c)` (retci 31-36) je izračunati kolike bi trebale biti dimenzije roditeljske komponente `c` a da bi u napravljenom razmještaju djeca bila optimalnih dimenzija. Metodu smo implementirali tako da uporabom pomoćne metode `findComponent` (retci 68-74) pronalazimo prvo dijete i njega vraćamo. Potom izračun delegiramo pomoćnoj metodi `determineSize` (retci 76-90) kojoj šaljemo optimalnu veličinu djeteta (kojeg pitamo da izračuna i vrati taj podatak) - redak 35, poziv `c.getPreferredSize()`. Na sličan način riješili smo i metodu `minimumLayoutSize(c)` (retci 38-43), uz razliku da pomoćnoj metodi `determineSize` sada šaljemo minimalne dimenzije koje vrati dijete - redak 42, poziv `c.getMinimumSize()`.

Pogledajmo sada kako je napisana pomoćna metoda `determineSize`. Ista prima referencu na roditeljsku komponentu (`parent`), referencu na dijete (`c`) te željene dimenzije djeteta (`dim`); jesu li to minimalne dimenzije ili optimalne dimenzije, za ovu metodu to nije bitno i to je odredio pozivatelj. U metodi računamo potrebnu širinu i visinu roditelja uz koje će dijete imati željene dimenzije te uz koje će sve ostalo biti ispoštivano. Konkretno, krećemo s ocjenom da je širina roditelja jednaka zbroju lijeve i desne dodatne margine (redak 79) a visina zbroju gornje i donje dodatne margine (redak 80). Potom, ako smo primili željene dimenzije djeteta (`dim != null`), ocjenu širine povećavamo za željenu širinu djeteta (redak 82) a ocjenu visine za željenu visinu djeteta (redak 83). Konačno, od roditelja dohvaćamo i njegove interne margine koje tamo postoje zbog bordera (redak 85). Ocjenu širine povećavamo za dohvaćenu lijevu i desnu marginu (redak 86) a ocjenu visine za dohvaćenu gornju i donju marginu (redak 87). Na kraju metode kao potrebne dimenzije vraćamo tako izračunatu širinu i visinu (redak 88).

Uočite da zadaća ovih metoda nije bila fizičko razmještanje komponenata niti uzimanje u obzir trenutne veličine roditelja. Pitanje koje ove metode rješavaju jest, kada bismo mogli roditelja dimenzionirati potpuno proizvoljno, kolike bi bile prikladne dimenzije.

Potpuno druga priča je metoda `layoutContainer` (retci 45-66). Njezina zadaća je, uzimajući u obzir aktualne dimenzije roditelja, po njegovoj površini razmjestiti djecu u skladu s algoritmom razmještanja. Pri tome je sasvim moguće da roditelj nije dimenzioniran niti na minimalne dimenzije, niti na optimalne dimenzije. Ovdje moramo raditi s dimenzijama koje su takve kakve jesu i napraviti najbolje što možemo.

Raspoređivanje započinje tako da dohvatimo prvo dijete (redak 47). Ako takvog nema, postupak je gotov i napuštamo metodu (retci 48-50). U suprotnom, računamo koliko prostora po širini možemo dodijeliti djetetu: pitamo roditelja koliko je širok (redak 52) i od toga oduzmemo zadane vodoravne margine (redak 53) te vodoravne margine koje postoje u roditelju zbog bordera (redak 54). Ako time ispadne da bi potreba širina djeteta bila negativna, resetiramo širinu na 0 (redak 55). Na sličan način računamo i visinu djeteta (retci 56-59). Jednom kada smo napravili potreban izračun širine i visine djeteta, pozivom metode `setBounds` pozicioniramo dijete na površini roditeljske komponente: x-koordinata njegovog gornjeg lijevog ugla jednaka je zbroju zadane lijeve margine i interne lijeve margine roditelja, y-koordinata njegovog gornjeg lijevog ugla jednaka je zbroju zadane gornje margine i interne gornje margine roditelja, njegova širina postavlja se na izračunatu širinu a njagova visina na izračunatu visinu. Time smo osigurali da je dijete uvijek dobro pozicionirano i rastegnuto po cijeloj površini roditeljske komponente (izuzev zadane margine koje treba poštivati).

Upravljač razmještajem HorizontalSplitLayout

Sljedeći upravljač razmještajem koji ćemo ilustrirati dozvoljava razmještanje dviju komponenata od koje smješta jednu do druge; svakoj komponenti dodjeljuje pola širine roditelja te ih rasteže po cijeloj visini roditelja. Kako bi programer mogao definirati koja komponenta ide lijevo a koja desno, dodat ćemo u upravljač dvije string-konstante: `HorizontalSplitLayout.LEFT` i `HorizontalSplitLayout.RIGHT`. Upravljač razmještajem moći ćemo dodatno konfigurirati tako da zadamo koliko treba ostaviti razmaka između te dvije komponente. Upravljač razmještajem želimo napisati tako da se zna nositi i sa situacijom kada jedna od komponenata nije zadana: u tom slučaju onu preostalu i dalje treba korektno pozicionirati na njezinu polovicu roditeljske komponente.

Kakvo je željeno ponašanje ovog upravljača, opet ćemo ilustrirati na konkretnim primjerima, slike 21.12 i 21.13.

Slika 21.12 prikazuje situaciju u kojoj su upravljaču razmještaja definirana oba djeteta koja treba razmještati.

Slika 21.12. HorizontalSplitLayout na djelu (1)



a) originalna veličina prozora b) prozor nakon promjene veličine

Slika 21.13 prikazuje situaciju u kojoj je upravljaču razmještaja definirano samo desno dijete.

Slika 21.13. HorizontalSplitLayout na djelu (2)



a) originalna veličina prozora b) prozor nakon promjene veličine

Izvorni kod upravljača razmještajem `HorizontalSplitLayout` prikazan je u nastavku (primjer 21.13).

Primjer 21.13. Upravljač razmještajem HorizontalSplitLayout

```
1 package book.java.swing.layouts;
2
3 import java.awt.Component;
```

```
 4 import java.awt.Container;
 5 import java.awt.Dimension;
 6 import java.awt.Insets;
 7 import java.awt.LayoutManager;
 8 import java.util.HashMap;
 9 import java.util.Map;
10
11 public class HorizontalSplitLayout implements LayoutManager {
12
13  public static final String LEFT = "LEFT";
14  public static final String RIGHT = "RIGHT";
15
16  private Map<Component, String> position = new HashMap<>();
17  private int gapWidth;
18
19  public HorizontalSplitLayout(int gapWidth) {
20   super();
21   this.gapWidth = gapWidth;
22  }
23
24  public HorizontalSplitLayout() {
25   this(0);
26  }
27
28  @Override
29  public void addLayoutComponent(String name, Component comp) {
30   if (!LEFT.equals(name) && !RIGHT.equals(name)) {
31    throw new IllegalArgumentException(
32     "Nepodržano ograničenje: " + name
33    );
34   }
35   String existing = position.get(comp);
36   if (existing != null) {
37    throw new IllegalArgumentException(
38     "Komponenta je već dodana."
39    );
40   }
41   if(position.containsValue(name)) {
42    throw new IllegalArgumentException(
43     "Već postoji komponenta s ograničenjem "+name+"."
44    );
45   }
46   position.put(comp, name);
47  }
48
49  @Override
50  public void removeLayoutComponent(Component comp) {
51   position.remove(comp);
52  }
53
54  @Override
55  public Dimension preferredLayoutSize(Container parent) {
56   return calcDimension(parent, Component::getPreferredSize);
57  }
58
59  @Override
60  public Dimension minimumLayoutSize(Container parent) {
61   return calcDimension(parent, Component::getMinimumSize);
```

```
 62  }
 63
 64  @Override
 65  public void layoutContainer(Container parent) {
 66   Insets parentInsets = parent.getInsets();
 67
 68   Component leftComp = null;
 69   Component rightComp = null;
 70   int n = parent.getComponentCount();
 71   for (int i = 0; i < n; i++) {
 72    Component c = parent.getComponent(i);
 73    String place = position.get(c);
 74    if (LEFT.equals(place)) {
 75     leftComp = c;
 76    } else if (RIGHT.equals(place)) {
 77     rightComp = c;
 78    }
 79   }
 80
 81   int w = parent.getWidth();
 82   int h = parent.getHeight()
 83     - parentInsets.top - parentInsets.bottom;
 84   if (h < 0)
 85    h = 0;
 86   int available = Math.max(
 87     0,
 88     w - gapWidth - parentInsets.left - parentInsets.right
 89    );
 90
 91   int wLeft = available / 2;
 92   int wRight = available - wLeft;
 93
 94   if (leftComp != null) {
 95    leftComp.setBounds(
 96     parentInsets.left,
 97     parentInsets.top,
 98     wLeft,
 99     h
100    );
101   }
102   if (rightComp != null) {
103    rightComp.setBounds(
104     parentInsets.left + wLeft + gapWidth,
105     parentInsets.top,
106     wRight,
107     h
108    );
109   }
110  }
111
112  private interface SizeGetter {
113   Dimension getSize(Component comp);
114  }
115
116  private Dimension calcDimension(Container parent,
117   SizeGetter getter) {
118
119   int n = parent.getComponentCount();
```

```
120   Dimension dim = new Dimension(0, 0);
121   for (int i = 0; i < n; i++) {
122    Component c = parent.getComponent(i);
123    Dimension cdim = getter.getSize(c);
124    if (cdim != null) {
125     dim.width = Math.max(dim.width, cdim.width);
126     dim.height = Math.max(dim.height, cdim.height);
127    }
128   }
129   dim.width = 2 * dim.width + gapWidth;
130
131   Insets parentInsets = parent.getInsets();
132   dim.width += parentInsets.left + parentInsets.right;
133   dim.height += parentInsets.top + parentInsets.bottom;
134
135   return dim;
136  }
137 }
138
```

Popratni demonstracijski program prikazan je u primjeru 21.14.

Primjer 21.14. Demonstracijski program za `HorizontalSplitLayout`

```
 1 package book.java.swing.layouts;
 2
 3 import java.awt.Color;
 4 import java.awt.Container;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.SwingUtilities;
 8 import javax.swing.WindowConstants;
 9
10 import book.java.swing.components.Poruka;
11
12 public class LayoutDemo2 extends JFrame {
13
14  private static final long serialVersionUID = 1L;
15
16  public LayoutDemo2() {
17   setTitle("Demonstracija rada s listama");
18   setSize(300, 300);
19   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
20   initGUI();
21  }
22
23  private void initGUI() {
24   Container cp = getContentPane();
25   cp.setLayout(new HorizontalSplitLayout(10));
26   Poruka poruka1 = new Poruka("Lijevi tekst");
27   poruka1.setOpaque(true);
28   poruka1.setBackground(Color.YELLOW);
29   Poruka poruka2 = new Poruka("Desni tekst");
30   poruka2.setOpaque(true);
31   poruka2.setBackground(Color.CYAN);
32   cp.add(poruka1, HorizontalSplitLayout.LEFT);
33   cp.add(poruka2, HorizontalSplitLayout.RIGHT);
34  }
```

```
35
36  public static void main(String[] args) {
37   SwingUtilities.invokeLater(()->{
38    new LayoutDemo2().setVisible(true);
39   });
40  }
41 }
42
```

Uočite kako smo u ovom primjeru u roditeljsku komponentu dodali djecu uz navođenje ograničenja:

```
32 cp.add(poruka1, HorizontalSplitLayout.LEFT);
33 cp.add(poruka2, HorizontalSplitLayout.RIGHT);
```

To je za posljedicu imalo da se i nad postavljenim upravljačem za svako dijete pozove metode `addLayoutComponent` što smo u upravljaču iskoristili da zapamtimo koje je dijete dodano uz koje ograničenje.

Proanalizirajmo sada malo detaljnije upravljač razmještajem prikazan u primjeru 21.13.

Upravljač definira dvije članske varijable: mapu `position` koja komponente mapira na ograničenja (koja mogu biti konstante `LEFT` i `RIGHT` - retci 13 i 14), te `gapWidth` koji pamti koliki razmak treba ostaviti između dvije komponente.

Metode `addLayoutComponent` (retci 28-47) i `removeLayoutComponent` (retci 49-52) ažuriraju mapu `position`. Pri tome se ne dozvoljava da se u mapu više puta pokuša ubaciti ista komponenta, da se doda komponenta uz ograničenje uz koje je dodana već neka druga komponenta (ne možemo imati dvije desne komponente) ili da ograničenje bude neko od nepodržanih.

U u ovom upravljaču izračun potrebne veličine roditelja (minimalne ili optimalne) delegirali smo pomoćnoj metodi. Međutim, za razliku od upravljača razještaja `StretchLayout` kod kojeg smo imali samo jednu komponentu pa smo izvana mogli dohvatiti vrstu dimenzije, u ovom upravljaču usred algoritma izračuna trebamo dohvaćati za svako dijete njegovu minimalnu ili pak njegovu optimalnu dimenziju. Problem smo riješili uporabom oblikovnog obrasca Strategija: definirali smo apstraktno sučelje `SizeGetter` (retci 112-114) koje modelira objekte *"dam ti komponentu - vrati mi neku njezinu dimenziju"*. Potom smo u metodama `preferredlayoutSize` (retci 54-57) i `minimumLayoutSize` (retci 59-62) pozvali pomoćnu metodu uz referencu na prikladnu strategiju koju smo zadali lambda-izrazom.

Sama pomoćna metoda prikazana je u retcima 116-136. Metoda prolazi kroz komponente roditelja i ignorira sve osim onih koje su dodane uz valjana ograničenja (ostalih ne bi niti smjelo biti). Potom kao željenu širinu djeteta bira maksimum od željenih širina sve djece a kao željenu visinu maksimum od željene visine sve djece. Ovo je relativno lagano za objasniti: kako naš upravljač razmještajem svakoj od djece daje iste dimenzije, ako jedno dijete ne pristaje na manju širinu od 100 a drugo na manju širinu od 200, širinu treba postaviti na veću od tih jer će tek ta širina zadovoljiti oba djeteta.

Uz ovo opažanje, ostatak izračuna je jasan. Jednom kada se utvrde željena širina i željena visina koje zadovoljavaju svako od djece, potrebna širina roditelja jednaka je dvostrukoj utvrđenoj širini jednog djeteta plus zadanom razmaku između njih plus lijevoj i desnoj margini koja je postavljena u roditelju; na sličan način određuje se i potrebna visina roditelja.

Postupak razmještanja (metoda `layoutContainer`) prikazan je u retcima 65-110. Postupak najprije prolazi kroz postojeću djecu roditelja i traži ono koje je zadano uz lijevo i desno ograničenje te pamti reference na njih (retci 68-79). Potom se računa koliko je mjesta u roditelju preostalo za razmještanje djece (po horizontali, od širine roditelja odbijaju se margine i zadani razmak između djece; po vertikali od visine roditelja odbijaju se margine) - retci 81-89. Potom se računa koliko može biti široko lijevo dijete (pola utvrđene raspoložive širine) - redak 91, te

koliko može biti široko desno dijete (preostali dio utvrđene širine; namjerno ne dijelimo širinu s dva jer ista može biti neparan broj; razmislite kako se naša implementacija razlikuje od takve) - redak 92.

Potom, ako lijeva komponenta postoji, pozicioniramo ju (retci 94-100). Također, ako desna komponenta postoji, pozicioniramo ju (retci 102-109).

Pokrenite demonstracijski program i uvjerite se da radi.

za vježbu pokušajte napisati modifikaciju ovog upravljača koja lijevom djetetu širinu postavlja na njegovu optimalnu (ili minimalnu) a desnom djetetu daje sam preostali prostor. Kako ćete tada napisati metode `preferredLayoutSize` i `minimumLayoutSize`? Što ako neka od komponenata nije definirana - kako to mijenja izračun? Kako ćete riješiti konkretno raspoređivanje ako roditelj nije dovoljno širok?

Postojeći razredi tipa `LayoutManager`

Java u svojim standardnim bibliotekama nudi niz gotovih implementacija upravljača razmještajem.

- `BorderLayout`
- `BoxLayout`
- `cardLayout`
- `FlowLayout`
- `GridBadLayout`
- `GridLayout`
- `GroupLayout`
- `SpringLayout`

Neki od njih u praksi se često koriste - primjerice, `BorderLayout`, a neki vrlo rijetko (poput `GridBagLayout`). U nastavku ćemo dati prikaz samo manjeg podskupa ovih upravljača razmještajem; za više informacija čitatelj se upućuje na službeni tutorijal https://docs.oracle.com/javase/tutorial/uiswing/layout/index.html.

Zbog uštede prostora, za obrađene upravljače u nastavku će biti prikazan isječak koda koji bi u demonstracijskom programu upao u metodu `initGUI`.

Upravljač razmještajem `BorderLayout`

Ovaj upravljač može raditi s do pet komponenata. Svaka od komponenata mora biti dodana uz jedno ograničenje. Nije dozvoljeno dodavanje više komponenata uz isto ograničenje. Podržana ograničenja navedena su u nastavku.

- `BorderLayout.PAGE_START`: uz gornji ili donji rub komponente (ovisno o lokalizacijskim postavkama - čita li se odozgo prema dolje ili odozdo prema gore); poštuje preferiranu visinu djeteta.
- `BorderLayout.PAGE_END`: uz donji ili gornji rub komponente (ovisno o lokalizacijskim postavkama - čita li se odozgo prema dolje ili odozdo prema gore); poštuje preferiranu visinu djeteta.
- `BorderLayout.LINE_START`: uz lijevi ili desni rub komponente (ovisno o lokalizacijskim postavkama - čita li se redak s lijeva na desno ili s desna na lijevo); poštuje preferiranu širinu djeteta.

- `BorderLayout.LINE_END`: uz desni ili lijevi rub komponente (ovisno o lokalizacijskim postavkama - čita li se redak s lijeva na desno ili s desna na lijevo); poštuje preferiranu širinu djeteta.
- `BorderLayout.CENTER`: dijete puni sav raspoloživ prostor koji nije zauzet djecom koja su razmještena uz rubove (ako takva uopće postoje). Ne poštuje niti preferiranu širinu niti preferiranu visinu.
- `BorderLayout.NORTH`: (uporaba se ne preporuča) uvijek uz gornji rub komponente; poštuje preferiranu visinu djeteta.
- `BorderLayout.SOUTH`: (uporaba se ne preporuča) uvijek uz donji rub komponente; poštuje preferiranu visinu djeteta.
- `BorderLayout.WEST`: (uporaba se ne preporuča) uvijek uz lijevi rub komponente; poštuje preferiranu širinu djeteta.
- `BorderLayout.EAST`: (uporaba se ne preporuča) uvijek uz desni rub komponente; poštuje preferiranu širinu djeteta.

Primjer uporabe prikazan je u nastavku.

```
 1 Container cp = getContentPane();
 2 cp.setLayout(new BorderLayout());
 3 Poruka[] poruke = new Poruka[] {
 4   new Poruka(null, "LINE_START tekst", Color.YELLOW),
 5   new Poruka(null, "LINE_END tekst", Color.YELLOW),
 6   new Poruka(null, "PAGE_START tekst", Color.YELLOW),
 7   new Poruka(null, "PAGE_END tekst", Color.YELLOW),
 8   new Poruka(null, "CENTER tekst", Color.YELLOW)
 9 };
10 for(Poruka p : poruke) {
11  p.setBorder(BorderFactory.createLoweredBevelBorder());
12 }
13 cp.add(poruke[0], BorderLayout.LINE_START);
14 cp.add(poruke[1], BorderLayout.LINE_END);
15 cp.add(poruke[2], BorderLayout.PAGE_START);
16 cp.add(poruke[3], BorderLayout.PAGE_END);
17 cp.add(poruke[4], BorderLayout.CENTER);
```

Ovaj kod smješten u metodu `initGUI` demonstracijskog programa rezultira razmještajem prikazanim na slici 21.14. Uočite kako se za rubne komponente uvijek poštuje ili preferirana širina, ili preferirana visina (ovisno gdje je komponenta smještena) dok se ona druga dimenzija postavlja tako popunjava sav raspoloživi prostor.

Slika 21.14. `BorderLayout` na djelu



Upravljač razmještajem `BoxLayout`

Ovaj upravljač može raditi s proizvoljnim brojem komponenata i radi bez specificiranja ograničenja. Konstruktor upravljača prima referencu na roditelja čiju djecu treba razmještati te naputak treba li sve komponente razmjestiti u jedan redak (`BoxLayout.LINE_AXIS`) ili u jedan stupac (`BoxLayout.PAGE_AXIS`).

Primjer uporabe prikazan je u nastavku.

```
 1 Container cp = getContentPane();
 2 cp.setLayout(new BoxLayout(cp, BoxLayout.LINE_AXIS));
 3 Poruka[] poruke = new Poruka[] {
 4   new Poruka(null, "tekst 1", Color.YELLOW),
 5   new Poruka(null, "tekst 2 malo dulji", Color.YELLOW),
 6   new Poruka(null, "tekst 3", Color.YELLOW),
 7   new Poruka(null, "tekst 4", Color.YELLOW),
 8   new Poruka(null, "tekst 5", Color.YELLOW)
 9 };
10 for(Poruka p : poruke) {
11  p.setBorder(BorderFactory.createLoweredBevelBorder());
12 }
13 for(Poruka p : poruke) {
14  cp.add(p);
15 }
```

Ovaj kod smješten u metodu `initGUI` demonstracijskog programa rezultira razmještajem prikazanim na slici 21.15. Promijenimo li način razmještanja na `BoxLayout.PAGE_AXIS`, dobit ćemo razmještaj kakav je prikazan na slici 21.16.

Slika 21.15. `BoxLayout` na djelu uz `BoxLayout.LINE_AXIS`



Slika 21.16. `BoxLayout` na djelu uz `BoxLayout.PAGE_AXIS`

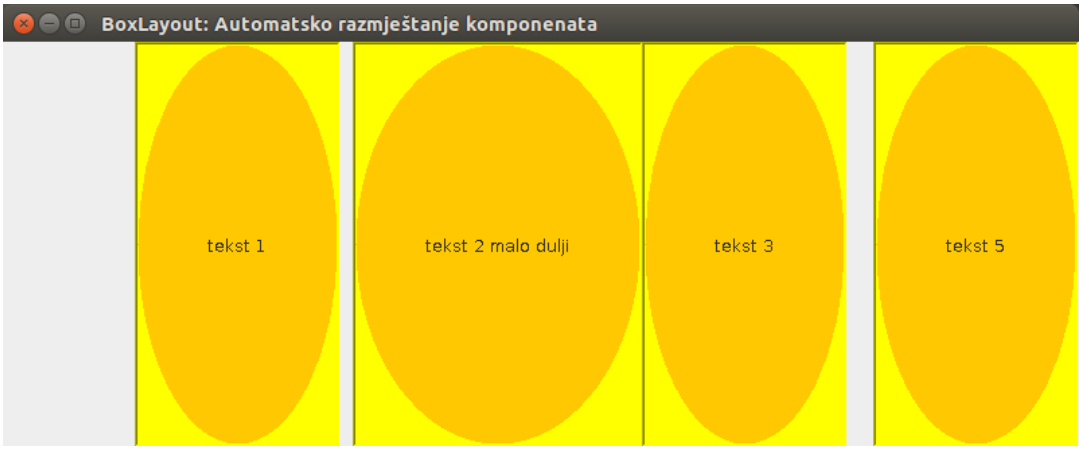


Ovaj upravljač razmještajem poštuje preferiranu širinu (ili visinu, ovisno o vrsti rasporeda) djece a ako roditelj ima više mjesta, onda proporcionalno skalira veličinu djece. Djecu smješta jedno do drugoga. Između parova djece moguće je umetati razmake fiksnih dimenzija dodavanjem prozirnih komponenata koje je moguće stvoriti pozivom `Box.createRigidArea(new Dimension(w,h))`. Ako se želi spriječiti da se komponente proporcionalno skaliraju na sav preostali raspoloživ prostor, tada se kao prvo dijete ili kao zadnje dijete može dodati nevidljiva komponenta dobivena pozivom `Box.createHorizontalGlue()` ili `Box.createVerticalGlue()` koje će "pojesti" sav raspoloživ slobodan prostor. Ovo možemo jednostavno pokazati tako da u prethodnom primjeru posljednju petlju koja dodaje svih 5 komponenata zamijenimo sljedećim isječkom koda:

```
1 cp.add(Box.createHorizontalGlue());
2 cp.add(poruke[0]);
3 cp.add(Box.createRigidArea(new Dimension(10, 0)));
4 cp.add(poruke[1]);
5 cp.add(poruke[2]);
6 cp.add(Box.createRigidArea(new Dimension(20, 0)));
7 cp.add(poruke[4]);
```

Dobit ćemo razmještaj kakav je prikazan na slici 21.17.

Slika 21.17. `BoxLayout` na djelu uz `BoxLayout.LINE_AXIS`



Kako je vidljivo sa slike, ostoji mali razmak između prvog i drugog djeteta (10 slikovnih elemenata) te duplo veći razmak između četvrtog i petog djeteta (dvadeset slikovnih elemenata). Čitav razmještaj potisnut je prema desnom kraju prozora.

Upravljač razmještajem `CardLayout`

Ovaj upravljač može raditi s proizvoljnim brojem komponenata. Prilikom dodavanja komponenata, svakoj se komponenti može dodijeliti neko proizvoljno ali jedinstveno ograničenje koje u ovom slučaju ima ulogu imena komponente. Apstrakcija koju ovaj upravljač razmještajem nudi je špil karata. Zamislimo da je jedna komponenta jedna karta špila. Gledate li na složeni špil karata odozgo, što vidite? Odgovor je: onu kartu koja je na vrhu; sve ostale karte skrivene su ispod i nisu vidljive. Lijep primjer iz prakse su moderni web-preglednici koji uz gornji rub nude niz kartica (engl. *tabs*) a ispod toga se na površini čitavog prozora nalaze prikazane sve otvorene web-stranice. Međutim, sve osim jedne (one koja odgovara aktivnoj kartici) su skrivene.

Primjer uporabe prikazan je u nastavku.

```
 1 Container cp = getContentPane();
 2 CardLayout layout = new CardLayout();
 3 cp.setLayout(layout);
 4 Poruka[] poruke = new Poruka[] {
 5   new Poruka(null, "tekst 1", Color.YELLOW),
 6   new Poruka(null, "tekst 2 malo dulji", Color.YELLOW),
 7   new Poruka(null, "tekst 3", Color.YELLOW),
 8   new Poruka(null, "tekst 4", Color.YELLOW),
 9   new Poruka(null, "tekst 5", Color.YELLOW)
10 };
11 for(Poruka p : poruke) {
12  p.setBorder(BorderFactory.createLoweredBevelBorder());
13 }
14 for(int i = 0; i < poruke.length; i++) {
15  cp.add(poruke[i], "Komponenta"+(i+1));
16 }
17
18 layout.show(cp, "Komponenta2");
```

Ovaj kod smješten u metodu `initGUI` demonstracijskog programa rezultira razmještajem prikazanim na slici 21.18. Komponentama se prilikom dodavanja u roditeljsku komponentu redom dodjeljuju imena "Komponenta1", "Komponenta2" i tako redom. Sam upravljač razmještajem `CardLayout` nudi metodu `show` koja prima ime komponente i nju izvlači na vrh špila čime ona postaje jedina prikazana komponenta. Prikazana komponenta uvijek popunjava čitavu površinu roditeljske komponente.

Slika 21.18. `CardLayout` na djelu



Uz metodu `show` koja po imenu pronalazi i prikazuje komponentu, na raspolaganju nam stoje i metode poput `first`, `last`, `next` i `previous` koje aktiviraju prvo odnosno zadnje dijete te sljedeće ili prethodno od trenutno prikazanog djeteta.

Upravljač razmještajem `FlowLayout`

Ovaj upravljač može raditi s proizvoljnim brojem komponenata. Pri tome sve komponente razmješta kao da su slova teksta koji pišete na pisaćem stroju: komponente nadodaje u jedan redak sve dok ima mjesta; potom prelazi u sljedeći redak pa u njega nadodaje komponente sve dok ima mjesta, pa potom prelazi u sljedeći redak, itd.

Primjer uporabe prikazan je u nastavku.

```
1 Container cp = getContentPane();
2 cp.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 15));
3 Poruka[] poruke = new Poruka[] {
4   new Poruka(null, "tekst 1", Color.YELLOW),
5   new Poruka(null, "tekst 2 malo je dulji", Color.YELLOW),
6   new Poruka(null, "tekst 3", Color.YELLOW),
7   new Poruka(null, "tekst 4", Color.YELLOW),
8   new Poruka(null, "tekst 5", Color.YELLOW)
9 };
10 for(Poruka p : poruke) {
11  p.setBorder(BorderFactory.createLoweredBevelBorder());
12 }
13 for(Poruka p : poruke) {
14  cp.add(p);
15 }
```

Najopćenitiji konstruktor koji smo ovdje koristili prima tri argumenta. Prvi argument određuje kako unutar jednog retka treba poravnati komponente; legalne vrijednosti su konstante `FlowLayout.LEFT` (kreni od lijevog ruba), `FlowLayout.RIGHT` (poravnaj uz desni rub), `FlowLayout.CENTER` (ostavi lijevo i desno jednaku količinu praznog mjesta), `FlowLayout.LEADING` (poravnaj uz onu stranu od koje čovjek kreće s čitanjem teksta; u Hrvatskoj čita se s lijeva na desno pa bi uz Hrvatsku lokalizaciju ovo odgovaralo lijevom rubu) te `FlowLayout.TRAILING` (poravnaj uz onu stranu s kojom čovjek završava prilikom čitanja retka teksta). Drugi i treći argument su broj slikovnih elemenata koliko je velik vodoravan razmak koji se ostavlja između dvije susjedne komponente te broj slikovnih elemenata koliko je velik razmak koji se ostavlja između dva susjedna retka komponenata.

Prethodni kod smješten u metodu `initGUI` demonstracijskog programa rezultira razmještajem prikazanim na slici 21.19.

Slika 21.19. `FlowLayout` na djelu



Upravljač razmještajem `GridLayout`

Ovaj upravljač može raditi s proizvoljnim brojem komponenata. Pri tome nad površinom roditeljske komponente stvara virtualnu tablicu čiji broj redaka i stupaca prima kroz konstruktor.

Sve retke tablice postavlja na jednaku visinu a sve stupce na jednaku širinu. Time su sve ćelije tablice potpuno jednakih dimenzija. Potom redosljedom kojim su komponente dodane u roditelja iste smješta u ćelije tablice počev od gornjeg lijevog ugla, redak po redak. Primjerice,

```
new GridLayout(5,2)
```

bi inicijalizirao upravljač razmještajem koji bi radio s tablicom koja ima pet redaka i dva stupca, odnosno mjesto za deset komponenata. Ako jedna od dimenzija tablice unaprijed nije poznata, može se ostaviti na 0; u tom slučaju upravljač će odrediti koliko mora biti taj broj s obzirom na drugu zadanu dimenziju i broj djece koja postoje u roditelju. Tako primjerice:

```
new GridLayout(0,2)
```

stvara upravljač koji je raditi s tablicom koja ima dva stupca i minimalno dovoljan broj redaka da se mogu razmjestiti sva djeca.

Primjer uporabe prikazan je u nastavku.

```
 1 Container cp = getContentPane();
 2 cp.setLayout(new GridLayout(0, 2));
 3 Poruka[] poruke = new Poruka[] {
 4   new Poruka(null, "tekst 1", Color.YELLOW),
 5   new Poruka(null, "tekst 2 malo je dulji", Color.YELLOW),
 6   new Poruka(null, "tekst 3", Color.YELLOW),
 7   new Poruka(null, "tekst 4", Color.YELLOW),
 8   new Poruka(null, "tekst 5", Color.YELLOW)
 9 };
10 for(Poruka p : poruke) {
11  p.setBorder(BorderFactory.createLoweredBevelBorder());
12 }
13 for(Poruka p : poruke) {
14  cp.add(p);
15 }
```

Ovaj kod smješten u metodu `initGUI` demonstracijskog programa rezultira razmještajem prikazanim na slici 21.20.

Slika 21.20. `GridLayout` na djelu



Ostali upravljači razmještajem

Iz ovog kratkog pregleda izostavili smo `GridBagLayout`, `GroupLayout` i `SpringLayout`. Radi se o komponentama koje nude dosta fleksibilno zadavanje razmještaja. nažalost, prva

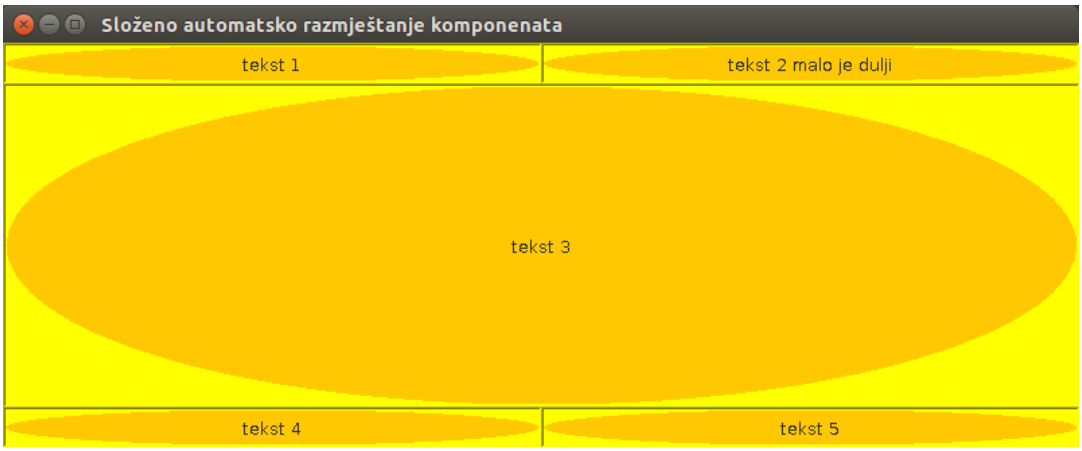
od njih je poprilično složena pa se zbog toga vrlo rijetko koristi. Za preostale dvije čitatelj se upućuje na dokumentaciju i službeni tutorijal.

Izrada kompozitnih razmještaja - komponenta `JPanel`

Uz sve prethodno opisane gotove upravljače razmještajem često ćemo se naći u situaciji da naš program treba nešto malo drugačije. Jedna mogućnost je u takvoj situaciji napisati vlastiti upravljač, što se zna pokazati kao dosta zahtjevan posao gdje se lagano unose pogreške. Druga mogućnost je pokušati kombinirati postojeće upravljače i hijerarhijsko gniježđenje komponenata. U tu svrhu možemo iskoristiti grafičku komponentu `javax.swing.JPanel` koja je upravo predviđena za nezavisno razmještanje svoje vlastite djece; bilo kroz konstruktor, bilo pozivom metode `setLayout` moguće je podesiti koji će upravljač razmještajem ona koristiti. Ako se ne zada niti jedan, konstruktor komponentu automatski podešava da koristi `FlowLayout`.

Pogledajte razmještaj prikazan na slici 21.21. Možete li se dosjetiti kako bismo ga moli ostvariti?

Slika 21.21. Složeni razmještaj



Na prvu izgleda kao da se radi o upravljaču `BorderLayout`: imamo nešto poravnato gore uz vrh, nešto dolje uz dno i imamo komponentu koja popunjava čitav preostali raspoloživi prostor, što upravo odgovara ograničenjima `BorderLayout.PAGE_START`, `BorderLayout.PAGE_END` i `BorderLayout.CENTER`.

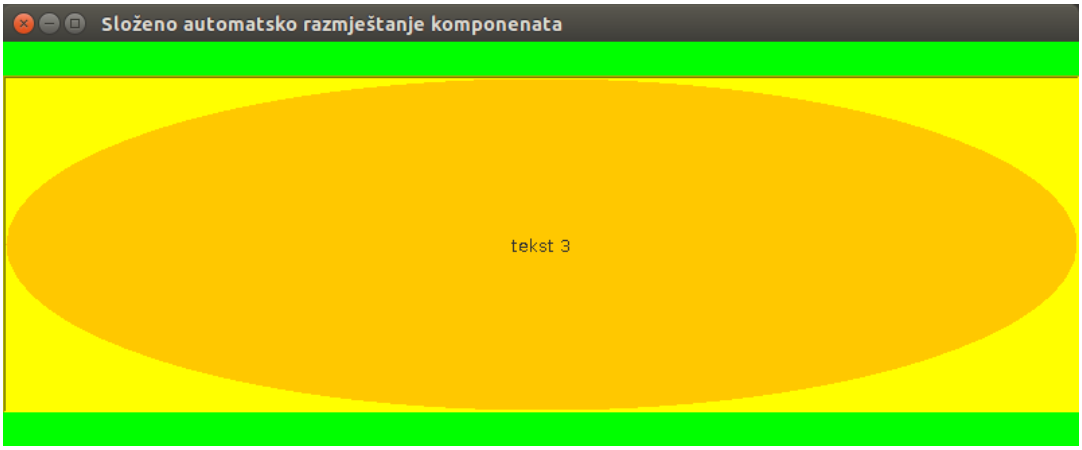
Jedini problem s ovim razmišljanjem je činjenica da gore i dolje nemamo jednu komponentu, već smo nekako uspjeli staviti dvije i poravnati ih tako da svaka popunjava po pola raspoloživog prostora. Da bismo ovo mogli ostvariti uporabom upravljača `BorderLayout`, gore i dolje moramo imati samo jednu komponentu. Stoga ćemo u prvom koraku gore i dolje dodati po jednog "općenamjenskog" roditelja: komponentu `JPanel`. Rješenje je prikazano u sljedećem isječku koda gdje su oba primjerka komponente `JPanel` privremeno učinjenja neprozirnima i obojanim u zelenu boju, tako da ih vidimo u razmještaju.

```
 1 Container cp = getContentPane();
 2 cp.setLayout(new BorderLayout());
 3 Poruka[] poruke = new Poruka[] {
 4   new Poruka(null, "tekst 1", Color.YELLOW),
 5   new Poruka(null, "tekst 2 malo je dulji", Color.YELLOW),
 6   new Poruka(null, "tekst 3", Color.YELLOW),
 7   new Poruka(null, "tekst 4", Color.YELLOW),
 8   new Poruka(null, "tekst 5", Color.YELLOW)
 9 };
10
11 for(Poruka p : poruke) {
```

```
12  p.setBorder(BorderFactory.createLoweredBevelBorder());
13 }
14
15 JPanel gornji = new JPanel();
16 gornji.setOpaque(true);
17 gornji.setBackground(Color.GREEN);
18 gornji.setPreferredSize(new Dimension(10, 25));
19
20 JPanel donji = new JPanel();
21 donji.setOpaque(true);
22 donji.setBackground(Color.GREEN);
23 donji.setPreferredSize(new Dimension(10, 25));
24
25 cp.add(gornji, BorderLayout.PAGE_START);
26 cp.add(poruke[2], BorderLayout.CENTER);
27 cp.add(donji, BorderLayout.PAGE_END);
```

Proanalizirajmo ovaj kod. U retku 2 nad vršnom komponentom postavljamo upravljač `BorderLayout`. Potom stvaramo sve poruke (retci 3-9) i dvije pomoćne roditeljske komponente (retci 15-18 i 20-23) koje bojamo u zeleno. Potom u vršnu roditeljsku komponentu dodajemo na vrh prvog pomoćnog roditelja, u centar treću poruku a na dno drugog pomoćnog roditelja. Pokretanjem ovog isječka dobit će se razmještaj kakav je prikazan na slici 21.22.

Slika 21.22. Složeni razmještaj (2)



Sada smo spremni za drugi dio posla. Svakom od pomoćnih roditelja instalirat ćemo upravljač razmještajem `GridLayout` koji koristi tablicu s jednim retkom i dva stupca. Potom ćemo preostale četiri poruke dodati kao djecu ovih roditelja (i ukloniti onaj suvišan kod koji je pomoćne roditelje bojao u zeleno). Konačno rješenje s opisanom modifikacijom prikazano je u nastavku.

```
 1 Container cp = getContentPane();
 2 cp.setLayout(new BorderLayout());
 3 Poruka[] poruke = new Poruka[] {
 4   new Poruka(null, "tekst 1", Color.YELLOW),
 5   new Poruka(null, "tekst 2 malo je dulji", Color.YELLOW),
 6   new Poruka(null, "tekst 3", Color.YELLOW),
 7   new Poruka(null, "tekst 4", Color.YELLOW),
 8   new Poruka(null, "tekst 5", Color.YELLOW)
 9 };
10
11 for(Poruka p : poruke) {
12  p.setBorder(BorderFactory.createLoweredBevelBorder());
13 }
```

```
14
15 JPanel gornji = new JPanel(new GridLayout(1,2));
16 gornji.add(poruke[0]);
17 gornji.add(poruke[1]);
18
19 JPanel donji = new JPanel(new GridLayout(1,2));
20 donji.add(poruke[3]);
21 donji.add(poruke[4]);
22
23 cp.add(gornji, BorderLayout.PAGE_START);
24 cp.add(poruke[2], BorderLayout.CENTER);
25 cp.add(donji, BorderLayout.PAGE_END);
```

Pokretanjem ovog koda dobit će se traženi razmještaj koji je bio zadan na slici 21.21.

Opisano hijerarhijsko gniježđenje nije ograničeno samo na jednu razinu kako smo ovdje prikazali. Međutim, nađete li se u situaciji da imate sedam razina komponenata `JPanel`, tada bi vjerojatno bilo isplativije sjest i napisati novi primjeren upravljač razmještajem.

Potpora izmjenjivom izgledu komponenata

Java aplikacija koja korisniku nudi grafičko korisničko sučelje na svim operacijskim sustavima izgleda jednako. Java to garantira i to jest jedna od njezinih prednosti: ako ne želite drugačije, program napisan na jednom operacijskom sustavu izgledat će jednako i na bilo kojem drugom operacijskom sustavu. U ranim danima Jave činilo se da je to velika prednost. Nešto kasnije, to je postala mana: program napisan u Javi niti na jednom operacijskom sustavu ne izgleda onako kako izgledaju ostali programi koji su izravno pisani za taj operacijski sustav (pojam "šaka u oko" pada na pamet).

Korisničko sučelje koje se oslanjalo na tehnologiju AWT od toga nije moglo pobjeći. Međutim, razvojem tehnologije Swing kao Javinog novijeg i modernijeg korisničkog sučelja u isto je ugrađena potpora za izmjenu načina na koji se prikazuju komponente (mogućnost uobičajeno poznata pod pojmom *skinnability*) kroz tehnologiju *Look & Feel* (ili kraće LAF). Ideja je vrlo jednostavna: izdvojimo iz koda grafičke komponente onaj dio koji varira (zvoni li Vam išta na oblikovni obrazac Strategija?) u zasebnu komponentu. A ono što u ovom slučaju varira jest način na koji se crta slika površine komponente te shodno tome način na koji se računaju minimalne, preferirane i maksimalne dimenzije komponente.

Ako se pitate zašto su dimenzije komponente vezane uz način na koji se ista iscrtava, uzmite samo u obzir sljedeći primjer. Zamislite dvije "teme" od kojih jedna koristi font od 12 točaka a druga font od 10 točaka. Očito je da će preferirane dimenzije iste komponente (koja prikazuje neki tekst) u ta dva slučaja biti različite.

Iz tog razloga sav kod koji se bavi ovim poslovima iz grafičkih je komponenata Swinga eksternaliziran u vanjsku strategiju koja je modelirana razredom `javax.swing.plaf.ComponentUI` a koju nazivamo *ui-delegatom*. Metode ovog razreda prikazane su u nastavku.

```
 1 package javax.swing.plaf;
 2 public abstract class ComponentUI {
 3   public void installUI(JComponent c) {}
 4   public void uninstallUI(JComponent c) {}
 5   public void paint(Graphics g, JComponent c) {}
 6   public void update(Graphics g, JComponent c) {...}
 7   public Dimension getPreferredSize(JComponent c) {...}
 8   public Dimension getMinimumSize(JComponent c) {...}
 9   public Dimension getMaximumSize(JComponent c) {...}
10   public boolean contains(JComponent c, int x, int y) {...}
11   public static ComponentUI createUI(JComponent c) {...}
```

```
12    public int getBaseline(JComponent c, int width, int height)
13        {...}
14    public Component.BaselineResizeBehavior
15       getBaselineResizeBehavior(JComponent c) {...}
16    public int getAccessibleChildrenCount(JComponent c) {...}
17    public Accessible getAccessibleChild(JComponent c, int i)
18        {...}
19 }
```

Grafička komponenta `JComponent` iz koje je dalje izvedena većina ostalih Swing komponenata sadrži zaštićenu člansku varijablu `ComponentUI ui` te metode namijenjene za potporu radu s ovim objektima:

```
1 public String getUIClassID();
2 protected void setUI(ComponentUI newUI);
3 public void updateUI();
```

Metoda `getUIClassID()` bi trebala vratiti identifikator koji je jedinstven za konkretan tip komponente. Svaka grafička komponenta koja naslijedi `JComponent` mora ponuditi i metodu `getUI()` koja ukalupljuje zaštićenu člansku varijablu `ui` u korektan tip i vraća je pozivatelju pri ćemu metoda mora biti deklarirana tako da definira ispravan povratni tip (ne općeniti razred `ComponentUI` već razred koji je specijalizacija ui-delegata za konkretan tip komponente. Metodu `setUI` izvedene komponente mogu podići na javnu razinu vidljivosti i dopustiti klijentima da je izravno pozivaju.

Unutar komponente `JComponent` dio metoda je napisan tako da izravno koristi postavljenog ui-delegata. Tako je primjerice metoda `paintComponent(g)` implementirana tako da nad instaliranim ui-delegatom poziva metodu `paint` i kao prvi argument prosljeđuje primljeni objekt `g` a kao drugi argument šalje referencu na samu sebe. Time ui-delegat može doći do komponente i ispitati njezina svojstva koja utječu na prikaz slike. Na sličan način i metode `getPreferredSize`, `getMinimumSize` i `getMaximumSize` posao delegiraju postavljenom ui-delegatu.

Statička metoda ui-delegata `createUI(c)` ima zadaću stvaranja i vraćanja objekta koji će služiti kao ui-delegat za komponentu koja je predana kao argument. Ako ui-delegat ne pamti nikakve informacije vezane uz samu komponentu, tada ova implementacija može imati jedan primjerak delegata koji vraća pri svim pozivima (imamo jednog delegata koji opslužuje sve komponente). Ako ui-delegat pamti informacije vezane uz komponentu, tada bi ova metoda u konkretnom razredu delegata trebala pri svakom pozivu stvarati i vraćati novi primjerak delegata.

Trenutna tema

Jednu temu (odnosno jedan LAF) čini:

- *popis delegata*: održava se mapa koja identifikatoru komponente pridružuje puni naziv razreda koji je njegov ui-delegat; identifikatori komponenata su stringovi koje za svaku komponentu vraća njezina metoda `getUIClassID()`,
- *niz svojstava teme*: koji će se koristiti font, koje boje i slično za neke se teme može dodatno namještati postavljanjem svojstava koja se čuvaju za temu, a koje komponente (odnosno pripadni delegati) mogu dohvaćati,
- *implementacije delegata* i ostalih razreda, uključivo vršnog razreda tipa `LookAndFeel`.

U Swingu, tema je modelirana razredom `javax.swing.LookAndFeel` koji služi kao ulazna točka prema svim ovim informacijama. Razred `javax.swing.UIManager` čuva referencu na trenutno aktivnu temu i omogućava njezin dohvat i promjenu. Ovaj razred omogućava i dohvat popisa svih instaliranih tema koje su dostupne virtualnom stroju pozivom statičke metode:

```
1 public static LookAndFeelInfo[] getInstalledLookAndFeels();
```

Za svaku temu u vraćenom polju bit će po jedan primjerak razreda `LookAndFeelInfo` nad kojim se pozivom metode

```
1 public String getClassName();
```

može doći do punog imena vršnog razreda teme koji se potom može koristiti kako bismo tu temu učinili aktivnom, uporabom statičke metode:

```
1 public static void setLookAndFeel(String className);
```

razreda `UIManager`.

Sljedeći isječak koda ilustrira aktiviranje prve teme.

```
1 LookAndFeelInfo[] lafs = UIManager.getInstalledLookAndFeels();
2 try {
3  UIManager.setLookAndFeel(lafs[0].getClassName());
4 } catch (ClassNotFoundException | InstantiationException
5   | IllegalAccessException
6   | UnsupportedLookAndFeelException e) {
7  e.printStackTrace();
8 }
```

Razred `UIManager` nudi i dvije metode kojima je moguće dohvatiti specifične teme:

```
1 String UIManager.getSystemLookAndFeelClassName();
2 String UIManager.getCrossPlatformLookAndFeelClassName();
```

Prva metoda će vratiti vršni razred teme koji nudi ui-delegate uz koje će Swingove standarne komponente izgledati onako kako na operacijskom sustavu na kojem je program pokrenut izgledaju nativne komponente. Druga metoda vraća naziv vršnog razreda teme uz koju grafičke komponente na svim operacijskim sustavima izgledaju jednako (i to je za Swing pretpostavljena tema - ako ništa ne podešavate).

Razred `UIManager` nudi i nekoliko metoda koje "ispod haube" koriste trenutnu temu za dohvat informacija. Tako će primjerice isječak:

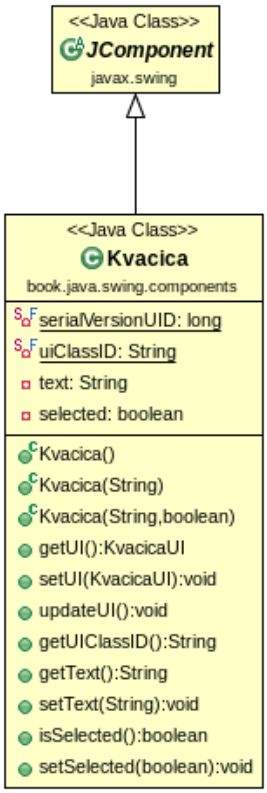
```
1 String labelDelegate = (String)UIManager.getDefaults()
2     .get("LabelUI");
3 System.out.println(labelDelegate);
```

dohvatiti i ispisati naziv razreda ui-delegata koji se koristi za prikaz Swingovih komponenata tipa `JLabel`. Za bilo koju konkretnu grafičku komponentu primjerak ui-delegata možemo zatražiti pozivom `ComponentUI ui = UIManager.getUI(c)` gdje je `c` referenca na grafičku komponentu. Ova metoda će u pozadini pozvati `String id = c.getUIClassID();`, potom će s tom informacijom pozvati `String className = (String)UIManager.get(id);` kako bi iz trenutne teme dohvatila naziv razreda koji je implementacija pridruženog ui-delegata. Nakon toga će nad tim razredom pozvati statičku metodu `.createUI(c)` koja će na kraju vratiti ili dijeljeni, ili novi primjerak ui-delegata.

Primjer komponente koja podržava LAF

Kako bismo malo detaljnije razjasnili što se tu događa, u nastavku je dan primjer jedne grafičke komponente koji lijepo ilustrira što je sve odgovornost ui-delegata i kako grafička komponenta komunicira s njime. Slika 21.23 prikazuje dijagram razreda na kojemu je vidljiva gruba struktura nove komponente. Komponentu koju razvijamo nazvali smo `Kvacica` (aludiramo na engl. *Checkbox*).

Slika 21.23. Komponenta `Kvacica`



Implementacija ove komponente prikazana je u primjeru 21.15.

Primjer 21.15. Komponenta `Kvacica`

```
 1 package book.java.swing.components;
 2
 3 import javax.swing.JComponent;
 4 import javax.swing.UIManager;
 5
 6 import book.java.swing.laf.KvacicaUI;
 7
 8 public class Kvacica extends JComponent {
 9
10  private static final long serialVersionUID = 1L;
11  private static final String uiClassID = "KvacicaUI";
12  private String text;
13  private boolean selected;
14
15  public Kvacica() {
16   this(null, false);
17  }
18
19  public Kvacica(String text) {
20   this(text, false);
21  }
22
23  public Kvacica(String text, boolean selected) {
24   super();
25   this.text = text;
26   this.selected = selected;
27   updateUI();
```

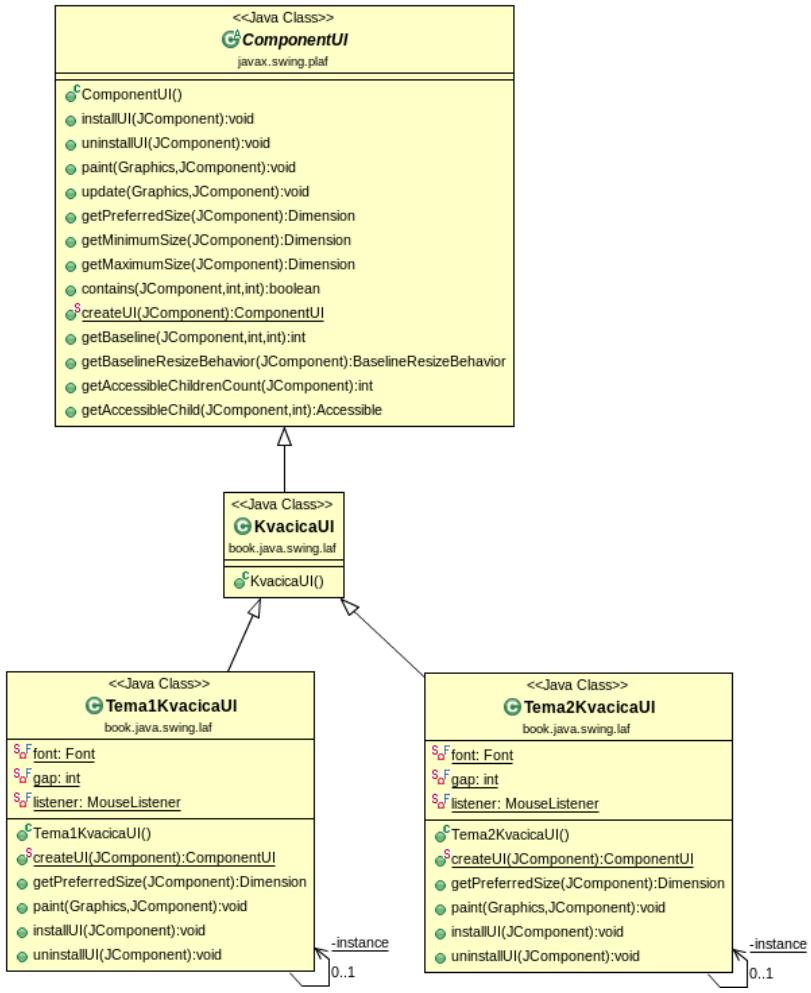
```
28  }
29
30  public KvacicaUI getUI() {
31   return (KvacicaUI)ui;
32  }
33
34  public void setUI(KvacicaUI newUI) {
35   super.setUI(newUI);
36  }
37
38  @Override
39  public void updateUI() {
40   setUI((KvacicaUI)UIManager.getUI(this));
41  }
42
43  @Override
44  public String getUIClassID() {
45   return uiClassID;
46  }
47
48  public String getText() {
49   return text;
50  }
51
52  public void setText(String text) {
53   this.text = text;
54   revalidate();
55   repaint();
56  }
57
58  public boolean isSelected() {
59   return selected;
60  }
61
62  public void setSelected(boolean selected) {
63   if(this.selected != selected) {
64    this.selected = selected;
65    repaint();
66   }
67  }
68 }
69
```

Kako razvijamo novu vrstu komponente, uobičejeno je za nju izvesti specijalizaciju od `ComponentUI` i to smo učinili razredom `KvacicaUI` (uočite da se taj razred spominje u izvornom kodu komponente). Uočite također da u konstruktoru komponente nakon namještanja teksta još pozivamo metodu `updateUI` koja će pitati `UIManager` da nam vrati ui-delegata iz trenutne teme za našu komponentu. Ako se (s pravom) pitate otkud bi `UIManager` to mogao znati za potpuno novu vrstu komponente koju do sada nikada nije vidio, odgovor je: on to ne zna. Znat će to za sve standardne Swingove komponente ali ne i za naše nove vrste komponenata; stoga ćemo u demonstracijskom programu prije prve uporabe bilo čega iz Swinga `UIManager`u izravno registrirati delegata kojeg treba koristiti za našu komponentu (sjetite se to pogledati u kodu).

Nakon što smo izveli osnovni razred za sve delegate komponente `Kvacica`, napisat ćemo još i dvije implementacije (idemo se praviti da razvijamo dvije teme). Svaka od tih implementacija bit će izvedena iz razreda `KvacicaUI`.

Dijagram razreda za ovaj dio rješenja prikazan je na slici 21.24.

Slika 21.24. Delegati za komponentu `Kvacica`



Programski kod za ova tri razreda prikazan je u primjerima 21.16, 21.17 i 21.18.

Primjer 21.16. Komponenta `KvacicaUI`

```
1 package book.java.swing.laf;
2
3 import javax.swing.plaf.ComponentUI;
4
5 public class KvacicaUI extends ComponentUI {
6 }
7
```

Primjer 21.17. Komponenta `Tema1KvacicaUI`

```
1 package book.java.swing.laf;
2
3 import java.awt.Color;
4 import java.awt.Dimension;
5 import java.awt.Font;
6 import java.awt.FontMetrics;
7 import java.awt.Graphics;
8 import java.awt.Insets;
9 import java.awt.Rectangle;
10 import java.awt.event.MouseAdapter;
11 import java.awt.event.MouseEvent;
```

```
12 import java.awt.event.MouseListener;
13
14 import javax.swing.JComponent;
15 import javax.swing.SwingUtilities;
16 import javax.swing.plaf.ComponentUI;
17
18 import book.java.swing.components.Kvacica;
19
20 public class Tema1KvacicaUI extends KvacicaUI {
21
22  private static final Tema1KvacicaUI instance =
23   new Tema1KvacicaUI();
24  private static final Font font =
25   new Font("Sans", Font.BOLD, 12);
26
27  private static final int gap = 5;
28
29  private static final MouseListener listener =
30   new MouseAdapter() {
31   public void mouseClicked(MouseEvent e) {
32    if(!SwingUtilities.isLeftMouseButton(e)) return;
33    Kvacica k = (Kvacica)e.getSource();
34    FontMetrics fm = k.getFontMetrics(k.getFont());
35    int side = fm.getHeight();
36    Insets ins = k.getInsets();
37    Rectangle rect = new Rectangle(
38     ins.left, ins.top, side, side
39    );
40    if(rect.contains(e.getPoint())) {
41     k.setSelected(!k.isSelected());
42    }
43   };
44  };
45
46  public static ComponentUI createUI(JComponent c) {
47   return instance;
48  }
49
50  @Override
51  public Dimension getPreferredSize(JComponent c) {
52   Kvacica k = (Kvacica)c;
53
54   FontMetrics fm = k.getFontMetrics(k.getFont());
55   Dimension dim = new Dimension(0, fm.getHeight());
56   String text = k.getText();
57
58   if(text!=null) {
59    dim.width += fm.stringWidth(text);
60   }
61
62   int side = fm.getHeight();
63   Insets ins = k.getInsets();
64
65   dim.width += ins.left + ins.right + side + gap;
66   dim.height += ins.top + ins.bottom;
67
68   return dim;
69  }
```

```
70
71  @Override
72  public void paint(Graphics g, JComponent c) {
73   Kvacica k = (Kvacica)c;
74
75   FontMetrics fm = k.getFontMetrics(k.getFont());
76   int side = fm.getHeight();
77   Insets ins = k.getInsets();
78
79   g.setColor(Color.WHITE);
80   g.fillOval(ins.left, ins.top, side-1, side-1);
81   g.setColor(Color.RED);
82   g.drawOval(ins.left, ins.top, side-1, side-1);
83
84   if(k.isSelected()) {
85    g.setColor(Color.BLACK);
86    g.fillOval(ins.left+2, ins.top+2, side-4, side-4);
87   }
88
89   String text = k.getText();
90   if(text!=null) {
91    g.setColor(Color.BLUE);
92    g.setFont(font);
93    g.drawString(
94     text,
95     ins.left+side+gap,
96     ins.top+fm.getAscent()
97    );
98   }
99  }
100
101 @Override
102 public void installUI(JComponent c) {
103  super.installUI(c);
104  c.addMouseListener(listener);
105  c.setFont(font);
106 }
107
108 @Override
109 public void uninstallUI(JComponent c) {
110  c.removeMouseListener(listener);
111  super.uninstallUI(c);
112 }
113}
114
```

Primjer 21.18. Komponenta `Tema2KvacicaUI`

```
1 package book.java.swing.laf;
2
3 import java.awt.Color;
4 import java.awt.Dimension;
5 import java.awt.Font;
6 import java.awt.FontMetrics;
7 import java.awt.Graphics;
8 import java.awt.Insets;
9 import java.awt.Rectangle;
10 import java.awt.event.MouseAdapter;
```

```
11 import java.awt.event.MouseEvent;
12 import java.awt.event.MouseListener;
13
14 import javax.swing.JComponent;
15 import javax.swing.SwingUtilities;
16 import javax.swing.plaf.ComponentUI;
17
18 import book.java.swing.components.Kvacica;
19
20 public class Tema2KvacicaUI extends KvacicaUI {
21
22  private static final Tema2KvacicaUI instance =
23   new Tema2KvacicaUI();
24  private static final Font font =
25   new Font("Sans", Font.BOLD, 16);
26
27  private static final int gap = 5;
28
29  private static final MouseListener listener =
30   new MouseAdapter() {
31   public void mouseClicked(MouseEvent e) {
32    if(!SwingUtilities.isLeftMouseButton(e)) return;
33    Kvacica k = (Kvacica)e.getSource();
34    FontMetrics fm = k.getFontMetrics(k.getFont());
35    int side = fm.getHeight();
36    Insets ins = k.getInsets();
37    Rectangle rect = new Rectangle(
38     k.getWidth()-side-ins.right, ins.top,
39     side, side
40    );
41    if(rect.contains(e.getPoint())) {
42     k.setSelected(!k.isSelected());
43    }
44   };
45  };
46
47  public static ComponentUI createUI(JComponent c) {
48   return instance;
49  }
50
51  @Override
52  public Dimension getPreferredSize(JComponent c) {
53   Kvacica k = (Kvacica)c;
54
55   FontMetrics fm = k.getFontMetrics(k.getFont());
56   Dimension dim = new Dimension(0, fm.getHeight());
57   String text = k.getText();
58
59   if(text!=null) {
60    dim.width += fm.stringWidth(text);
61   }
62
63   int side = fm.getHeight();
64   Insets ins = k.getInsets();
65
66   dim.width += ins.left + ins.right + side + gap;
67   dim.height += ins.top + ins.bottom;
68
```

```
 69    return dim;
 70   }
 71
 72   @Override
 73   public void paint(Graphics g, JComponent c) {
 74    Kvacica k = (Kvacica)c;
 75
 76    FontMetrics fm = k.getFontMetrics(k.getFont());
 77    int side = fm.getHeight();
 78    Insets ins = k.getInsets();
 79
 80    g.setColor(Color.WHITE);
 81    g.fillRect(
 82     k.getWidth()-side-ins.right-1, ins.top,
 83     side, side-1
 84    );
 85    g.setColor(Color.RED);
 86    g.drawRect(
 87     k.getWidth()-side-ins.right-1, ins.top,
 88     side, side-1
 89    );
 90
 91    if(k.isSelected()) {
 92     g.setColor(Color.BLACK);
 93     g.drawLine(
 94      k.getWidth()-side-ins.right-1, ins.top,
 95      k.getWidth()-side-ins.right+side, ins.top+side
 96     );
 97     g.drawLine(
 98      k.getWidth()-side-ins.right-1, ins.top+side,
 99      k.getWidth()-side-ins.right+side, ins.top
100     );
101    }
102
103    String text = k.getText();
104    if(text!=null) {
105     g.setColor(Color.BLUE);
106     g.setFont(font);
107     g.drawString(text, ins.left, ins.top+fm.getAscent());
108    }
109   }
110
111   @Override
112   public void installUI(JComponent c) {
113    super.installUI(c);
114    c.addMouseListener(listener);
115    c.setFont(font);
116   }
117
118   @Override
119   public void uninstallUI(JComponent c) {
120    c.removeMouseListener(listener);
121    super.uninstallUI(c);
122   }
123  }
124
```

Konačno, programski kod za demonstracijski program prikazan je u primjeru 21.19.

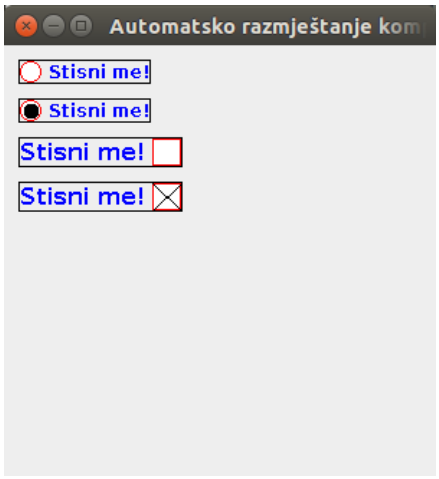
Primjer 21.19. Demonstracijski program za komponentu `Kvacica` i LAF

```
 1 package book.java.swing.laf;
 2
 3 import java.awt.Color;
 4 import java.awt.Container;
 5
 6 import javax.swing.BorderFactory;
 7 import javax.swing.JFrame;
 8 import javax.swing.SwingUtilities;
 9 import javax.swing.UIManager;
10 import javax.swing.WindowConstants;
11
12 import book.java.swing.components.Kvacica;
13
14 public class LAFDemo1 extends JFrame {
15
16  private static final long serialVersionUID = 1L;
17
18  public LAFDemo1() {
19   setTitle("Automatsko razmještanje komponenata");
20   setSize(300, 300);
21   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
22   initGUI();
23  }
24
25  private void initGUI() {
26   Container cp = getContentPane();
27   cp.setLayout(null);
28
29   Kvacica[] kvacice = new Kvacica[4];
30   for(int i = 0; i < kvacice.length; i++) {
31    kvacice[i] = new Kvacica("Stisni me!");
32    if(i>=2) {
33     kvacice[i].setUI(
34      (KvacicaUI)Tema2KvacicaUI.createUI(kvacice[i])
35     );
36    }
37    kvacice[i].setBorder(
38     BorderFactory.createLineBorder(Color.BLACK)
39    );
40    int y = 10;
41    if(i>0) {
42     y += kvacice[i-1].getY() + kvacice[i-1].getHeight();
43    }
44    kvacice[i].setLocation(10, y);
45    kvacice[i].setSize(kvacice[i].getPreferredSize());
46   }
47
48   for(Kvacica k : kvacice) {
49    cp.add(k);
50   }
51  }
52
53  public static void main(String[] args) {
54   SwingUtilities.invokeLater(()->{
55    UIManager.getDefaults().put(
56     "KvacicaUI",
```

```
57      "book.java.swing.laf.Tema1KvacicaUI"
58     );
59     new LAFDemo1().setVisible(true);
60    });
61   }
62 }
63
```

Pokretanjem ovog programa dobiva se prikaz dan na slici 21.25.

Slika 21.25. Demonstracijski program



Pogledajte na slici kako komponenta `Kvacica` izgleda u svakoj od tema. U prvoj temi, oznaka koja indicira je li komponenta selektirana ili nije je kružić i smještena je s lijeve strane. U drugoj temi radi se o kvadratiću koji se nalazi s desne strane. Promjena selektiranosti postiže se klikom na samu oznaku - klik na okolni tekst ili prazan prostor nema nikakvog efekta. Uz ovo razmatranje trebalo bi postati jasno da sama grafička komponenta `Kvacica` ne može interpretirati korisnikovu interakciju: ona naime ne zna na koji se način može mijenjati njezino stanje - to je zadaća ui-delegata.

Pogledajte sada malo detaljnije programski kod delegata (primjeri 21.17 i 21.18). U metodi `installUI` delegat komponenti postavlja atribute koji su vezani uz temu (primjerice, font koji treba koristiti) te se nad komponentom prijavljuje za dojavu o korisnikovoj interakciji koja je za delegat interesantna (u našem primjeru prijavljujemo `MouseListener`). U metodi `uninstallUI` odjavljujemo sve promatrače koje smo prethodno prijavili. Metodu `getPreferredSize` implementirali smo tako da računa potrebne dimenzije komponente a metodu `paint` tako da crta sliku komponente (uočite kako predanu komponentu ukalupljujemo u `Kvacica` jer znamo da je ona tog tipa, a tada možemo doći do svih potrebnih informacija poput teksta koji trebamo prikazati odnosno stanja selektiranosti). Oba delegata u svojim razredima definiraju po jedan statički primjerak sebe koji vraćaju u metodi `createUI`.

Sve standardne Swingove komponente imaju već unaprijed definirane delegate u svim temema koje dolaze standardno s distribucijom Jave. Na Interentu je moguće pronaći dodatne besplatne kao i komercijalne teme koje se mogu koristiti programe koje razvijate. Za to dovoljno dohvatiti JAR-datoteku koja sadrži implementaciju teme, dodati je u *classpath* i potom je u programu aktivirati pozivom `UIManager.setLookAndFeel(classname)`.



Dinamička promjena teme

Ako u programu planirate teme mijenjati dinamički (nakon što je prozor pokrenut), nakon svake promjene teme morate nad *svim* komponenatama pozvati metodu `updateUI` kako bi dohvatile i postavile nove delegate. U razredu `SwingUtilities` u tu je svrhu dodana statička metoda `updateComponentTreeUI(c)` koja

prima referencu na komponentu i nad njom i rekurzivno nad svom njenom djecom poziva metodu `updateUI()`. Obavezni pozovite tu metodu i predajte joj referencu na vršnog roditelja (prozor). Ako to zaboravite, mogu se dogoditi svakakve (lo-še) stvari.

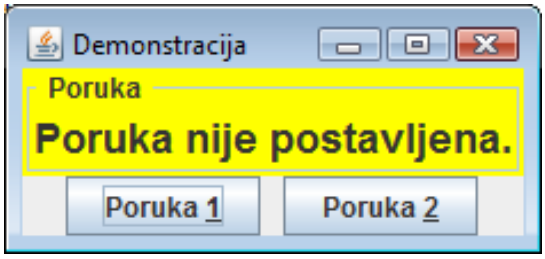
Poglavlje 22. Osnovne komponente Swinga

U ovom poglavlju osvrnut ćemo se na neke od osnovnih komponenata Swinga. Konkretno, pozabavit ćemo se labelama, gumbima i izborničkim strukturama.

Labela i dva gumba

Druženje ćemo započeti izradom programa koji prikazuje jednu labelu i dva gumba, kako je prikazano na slici 22.1.

Slika 22.1. Demonstracijski program (1)



Središnja komponenta (prikazana žutom bojom) je labela kojoj je postavljen border s imenom (*titled border*). Labela je u Swingu ostvarena komponentom `javax.swing.JLabel`.

Labela može prikazivati tekst, ikonu (sličicu; ikone su modelirane sučeljem `javax.swing.Icon`) ili pak oboje. Za postavljanje teksta na raspolaganju je metoda `setText(string)` a za postavljanje ikone metoda `setIcon(icon)`. Prilikom prikaza, tekst i ikone (uz pretpostavku da je oboje postavljeno) oboje se uvijek prikazuju jedno neposredno smješteno do drugoga, i to čini sadržaj labele. U slučaju da labela ima više prostora no što je potrebno za prikaz tog sadržaja, isti će na određen način biti pozicioniran.

Za određivanje relativnog odnosa teksta s obzirom na ikonu koriste se dvije metode: `setHorizontalTextPosition(htp)` i `setVerticalTextPosition(vtp)`. Pri tome argument `htp` može biti jedna od sljedećih statičkih konstanti iz `SwingConstants`: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING` a argument `vtp` može biti jedna od sljedećih statičkih konstanti iz `SwingConstants`: `TOP`, `CENTER`, `BOTTOM`. Primjerice, želimo li da se postavljeni tekst u labeli prikaže dolje-desno s obzirom na ikonu, koristili bismo sljedeći isječak koda:

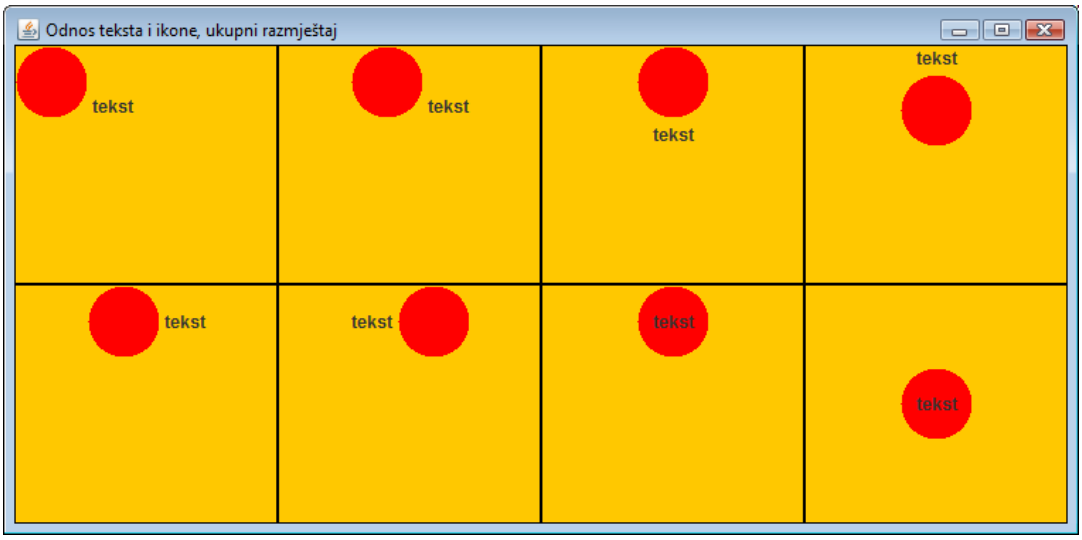
```
1 JLabel labela = new JLabel();
2 labela.setText(...);
3 labela.setIcon(...);
4 labela.setHorizontalTextPosition(SwingConstants.RIGHT);
5 labela.setVerticalTextPosition(SwingConstants.BOTTOM);
```

Takvo pozicioniranje prikazano je na slici 22.2 u gornjem retku, u drugom stupcu. Slika 22.2 prikazuje 8 različitih konfiguracija gdje je 8 labela razvučeno svaka na jednu osminu površine prozora. Ikone su veliki crveni krugovi čije su dimenzije namjerno odabrane većima od visine teksta kako bi se lijepo vidio efekt pozicioniranja teksta relativno prema ikoni. Uočite u tom istom primjeru (gornji redak, drugi stupac) da je čitava kompozicija tekst-ikona po vertikali poravnata uz gornji rub labele a po horizontali je centrirana. Za podešavanje položaja cjelokupnog sadržaja zadužene su metode `setHorizontalAlignment(ha)` te `setVerticalAlignment(va)`. Spomenuti primjer (gornji redak, drugi stupac) dobiven je uz:

```
1 labela.setHorizontalAlignment(SwingConstants.CENTER);
2 labela.setVerticalAlignment(SwingConstants.TOP);
```

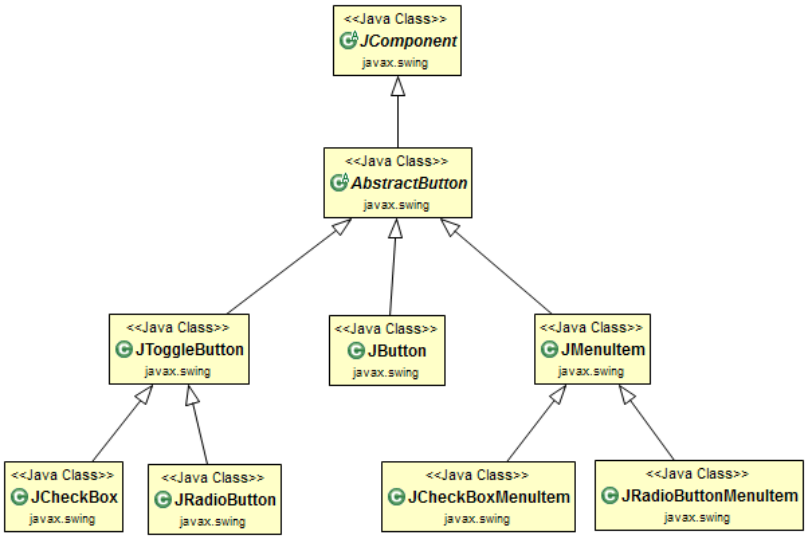
Uočite da će relativno pozicioniranje teksta po vertikali uz konstante `TOP` ili `BOTTOM` a u slučaju da je po horizontali korišten `CENTER` tekst smjestiti iznad odnosno ispod ikone. Ako se kao relativni položaj teksta s obzirom na ikonu zada `CENTER` po obje osi, tada će tekst biti ispisan preko ikone što ilustriraju posljednja dva primjera na slici (donji redak, desno).

Slika 22.2. Pozicioniranje teksta i ikone u labeli



Što se tiče gumbi, Swing biblioteka sadrži nekoliko vrsta gumbi koji su svi izvedeni iz apstraktnog razreda `javax.swing.AbstractButton` (vidi sliku 22.3).

Slika 22.3. Modeli gumbi



Razred `AbstractButton` definira niz temeljnih mogućnosti koje ima svaki gumb. Tako je gumbu moguće definirati tekst koji treba prikazati (metoda `setText(text)`), ikonu koju treba prikazati (metoda `setIcon(icon)`) te namjestiti njihov međusobni odnos (koriste se metode istog imena kao i u razredu `JLabel`). Gumb može biti selektiran (što se dalje vizualno razrađuje kroz porodicu izvedenu iz podrazreda `JToggleButton`) - potpora su metode `isSelected()` te `setSelected(value)`. Moguće je namjestiti ikonu koju će gumb prikazivati kada je selektiran (metode `setSelectedIcon(con)` i `getSelectedIcon()`) ikonu koju će gumb prikazivati kada korisnik mišem dođe iznad gumba (metode `setRolloverIcon(icon)`, `getRolloverIcon()`, `setRolloverSelectedIcon(icon)` i `getRolloverSelectedIcon()`), ikonu koju će gumb prikazivati kada je onemogućen (metode `setDisabledIcon(icon)`,

`getDisabledIcon()`, `setDisabledSelectedIcon(icon)` te `getDisabledSelectedIcon()`). Da bi automatska promjena ikone kada korisnik mišem dođe iznad gumba proradila, treba je omogućiti pozivom `setRolloverEnabled(true)`; je li taj efekt omogućen nad gumbom, moguće je ispitati metodom `isRolloverEnabled()`.

Kada korisnik "aktivira" gumb (na bilo koji način na koji to gumb podržava; primjerice, ako gumb ima fokus, dovoljno je pritisnuti tipku Space; alternativno, korisnik može kliknuti mišem na gumb ili može aktivirati na tipkovnici podešenu mnemoničku tipku), gumb o aktivaciji može obavijestiti sve zainteresirane promatrače koji su modelirani sučeljem `ActionListener`. Prijava i odjava ide ovih promatrača obavlja se metodama `addActionListener(l)` te `removeActionListener(l)`; događaj je opisan objektom tipa `ActionEvent` koji nudi metodu `getCommandName()` za dohvat imena naredbe koju treba izvršiti te metodu `getSource()` za dohvat reference na sam gumb. Sučelje promatrača prikazano je u nastavku.

Koji će naziv naredbe gumb upisati u objekt `ActionEvent` prije no što krene obavještavati promatrače, za gumb se može podesiti pozivom `setActionCommand(string)` odnosno dohvatiti pozivom `getActionCommand()`.

Gumbu je metodom `setAction(action)` moguće postaviti i "napredniju" vrstu promatrača koji će biti obaviješten o aktiviranu gumba. Takav promatrač nad gumbom može biti samo jedan a modeliran je razredom `Action` o čemu će biti riječi nešto kasnije. Postavljena akcija može se dohvatiti pozivom `getAction()`. Formalno, vanjski kontekst, razred `AbstractButton`, sučelje `Action` te konkretne implementacije ovog sučelja čine sudionike oblikovnog obrasca *Naredba* (engl. *Command design pattern*).

Želimo li dobiti gumb koji prikazuje samo ikonu i koji je inače proziran, umjesto poziva `setOpaque(false)` potrebno je pozvati `setContentAreaFilled(false)`.

Koja će biti mnemonička tipka, podešava se pozivom `setMnemonic(int code)` gdje je argument kod tipke (kodovi su definirani kao statičke konstante razreda `KeyEvent`; primjerice, tipka 'A' na tipkovnici ima kod `KeyEvent.VK_A`). Mnemonička tipka će pri prikazu teksta gumba biti prikazana kao podcrtano slovo (to možete vidjeti i u izbornicima) a pritisak na tu tipku (kada je izbornik prikazan) odnosno ponekad na `alt` i tu tipku ima za rezultat aktiviranje gumba (kao da ga je korisnik kliknuo). Mnemoničke tipke ne razlikuju velika i mala slova već doslovno predstavljaju tipku na tipkovnici. Postoji i inačica metode `setMnemonic(char)` koja kao argument prima `char` ali se njezina uporaba obeshrabruje.

Spomenimo i da se za gumb može definirati tekst koji će se na ekranu pojaviti u malom "oblačku" ako korisnik mirno zadrži miš jedno vrijeme iznad gumba. Taj se tekst postavlja metodom `setTooltipText(text)` a dohvaća metodom `getTooltipText()`.

"Klasičan" gumb modeliran je razredom `JButton`. Gumbi koji korisniku nude mogućnost vizualne indicikacije jesu li selektirani ili nisu izvedeni su iz `JToggleButton` a radi se o `JCheckBox` i `JRadioButton`. Spomenimo da se ove dvije vrste gumba razlikuju po izgledu, ali ne i po funkcionalnosti. Naime, gumbi tipa `JRadioButton` nisu automatski međusobno isključivi (na što korisnike najčešće posjeća njihov izgled). Da bismo dobili grupu međusobno isključivih gumbi, uz dodavanje u hijerarhiju komponenata koje se prikazuju iste je potrebno dodati i u pomoćni objekt (koji nije grafička komponenta) a koji je primjerak razreda `javax.swing.ButtonGroup`. Tom je objektu moguće dodati bilo proizvoljan broj bilo kakvih gumbi (i mješavina gumbi tipova `JCheckBox` i `JRadioButton` je prihvatljiva) i on će se pobrinuti da se pri selekciji jednog gumba ostali odselektiraju. Dakako, kako se ne bi zbunjivalo korisnike, preporuča se u tu svrhu koristiti isključivo primjerke razreda `JRadioButton`. Treća grupa gumbi modelirana je podrazredom `JMenuItem` i oni se koriste pri izgradnji izborničkih struktura o čemu će biti više riječi naknadno.

Programski kod koji stvara i razmješta komponente u skladu s razmještajemo prikazanim na slici 22.1 prikazan je u primjeru 22.1. Kako nam treba "običan" gumb, korišten je razred `JButton`.

Primjer 22.1. Demonstracijski program

```
 1 package book.java.swing.std;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Color;
 5 import java.awt.Container;
 6 import java.awt.Dimension;
 7 import java.awt.event.KeyEvent;
 8
 9 import javax.swing.BorderFactory;
10 import javax.swing.Box;
11 import javax.swing.BoxLayout;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JPanel;
16 import javax.swing.SwingUtilities;
17 import javax.swing.WindowConstants;
18
19 public class StdDemo1 extends JFrame {
20
21  private static final long serialVersionUID = 1L;
22  private JLabel labela;
23
24  public StdDemo1() {
25   setTitle("Demonstracija");
26   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
27   initGUI();
28   pack();
29   setLocationRelativeTo(null);
30  }
31
32  private void initGUI() {
33   Container cp  = getContentPane();
34   cp.setLayout(new BorderLayout());
35
36   labela = new JLabel("Poruka nije postavljena.");
37   labela.setBorder(
38    BorderFactory.createTitledBorder(" Poruka ")
39   );
40   labela.setOpaque(true);
41   labela.setBackground(Color.YELLOW);
42   labela.setFont(
43    labela.getFont().deriveFont(
44     labela.getFont().getSize2D()*1.5f
45    )
46   );
47
48   JButton poruka1 = new JButton();
49   poruka1.setText("Poruka 1");
50   poruka1.setToolTipText("Prikazuje pozdrav.");
51   poruka1.setMnemonic(KeyEvent.VK_1);
52   JButton poruka2 = new JButton();
53   poruka2.setText("Poruka 2");
54   poruka2.setToolTipText("Ispisuje smajliće.");
55   poruka2.setMnemonic(KeyEvent.VK_2);
56
```

```
57    JPanel donjiPanel = new JPanel();
58    donjiPanel.setLayout(
59     new BoxLayout(donjiPanel, BoxLayout.LINE_AXIS)
60    );
61    donjiPanel.add(Box.createGlue());
62    donjiPanel.add(poruka1);
63    donjiPanel.add(Box.createRigidArea(new Dimension(10, 0)));
64    donjiPanel.add(poruka2);
65    donjiPanel.add(Box.createGlue());
66
67    cp.add(labela, BorderLayout.CENTER);
68    cp.add(donjiPanel, BorderLayout.PAGE_END);
69   }
70
71   public static void main(String[] args) {
72    SwingUtilities.invokeLater(() -> {
73     new StdDemo1().setVisible(true);
74    });
75   }
76 }
77
```

U ovom primjeru koristimo jednu labelu i dva gumba. Za potrebe izrade razmještaja, nad vršnom komponentom u prozoru postavili smo `BorderLayout` i njemu kao središnju komponentu labelu a na dno novi pomoćni `JPanel` u kojem se za razmještaj brine `BoxLayout`. U taj panel dodali smo transparentnu komponentu tipa *glue*, prvi gumb, transparentnu komponentu čija je širina fiksna i čija je namjena dodavanje razmaka između dva gumba, drugi gumb te ponovno transparentnu komponentu tipa *glue*. Prva i posljednja komponenta na sebe će preuzeti (u jednakoj mjeri) sav raspoloživ neiskorišteni prostor pa će se time gumbi pojaviti u sredini.

Programski kod koji treba izvršiti kada se pritisne neki od gumba možemo dodati na nekoliko načina.

Pokušaj 1

Prvi i najjednostavniji način je svakom gumbu definirati simboličko ime naredbe koju pritisak na taj gumb treba izvršiti. Potom ćemo definirati jedan objekt tipa `ActionListener` čija je namjena obraditi aktiviranje gumba. Taj ćemo objekt postaviti nad sve gumbe a u njegovoj metodi `actionPerformed(e)` imat ćemo `switch`-pitalicu koja pokreće određen dio koda ovisno o simboličkom imenu naredbe koje je upisano u objekt `e` koji je stigao kao argument (u prikazanom rješenju, zbog preglednosti, taj je kod izbačen u zasebnu metodu koju poziva ovaj objekt a njega smo definirali lambda-izrazom). Modifikacija koda koja ovo ilustrira prikazana je u nastavku (primjer 22.2).

Primjer 22.2. Uporaba zajedničkog promatrača

```
1 public class StdDemo1 extends JFrame {
2
3  ...
4
5  private void initGUI() {
6   ...
7   poruka1.setActionCommand("prva");
8   ...
9   poruka2.setActionCommand("druga");
10  ...
11  ActionListener l = e -> {
12   obradiKlik(e);
13  };
```

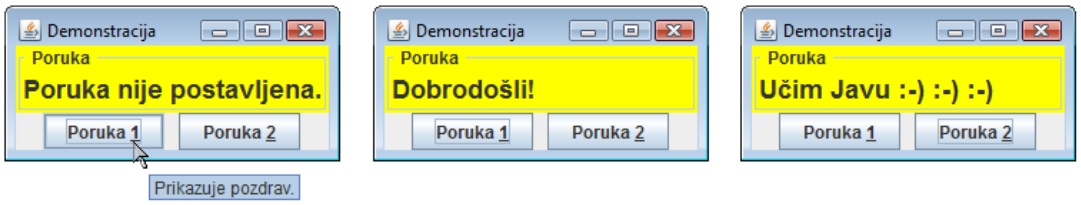
```
14
15    poruka1.addActionListener(l);
16    poruka2.addActionListener(l);
17   }
18
19   private void obradiKlik(ActionEvent e) {
20    String naredba = e.getActionCommand();
21    switch(naredba) {
22    case "prva":
23     labela.setText("Dobrodošli!");
24     break;
25    case "druga":
26     labela.setText("Učim Javu :-) :-) :-)");
27    }
28   }
29
30   ...
31  }
32
```

U prikazanom rješenju za prvi gumb definirali smo simboličko ime "prva" a za drugi gumb simboličko ime "drugi" (retci 7 i 9). Potom smo definirali objekt tipa `ActionListener` koji poziva pomoćnu metodu `obradiKlik(e)` (retci 11-13) i taj smo objekt postavili kao promatrača nad oba gumba (retci 15-16).

Pomoćna metoda `obradiKlik(e)` definirana je u retcima 19-28 i sastoji se od jedne velike naredbe `switch` koja ispituje koje je simboličko ime upisano u objekt `e` i temeljem toga izvršava određen dio koda.

Napravite ovu modifikaciju koda, pokrenite program i uvjerite se da radi. Slika 22.4 (lijevo) prikazuje *tooltip* koji se pojavi nakon što se pokazivač miša kraće vrijeme zadrži nad prvim gumbom; (sredina) prikazuje rezultat pritiska na prvi gumb; (desno) prikazuje rezultat pritiska na drugi gumb.

Slika 22.4. Demonstracijski program (2)



Prikazani način pisanja programa koji obrađuje pritiske ne gumbe radi, ali nije u skladu s Načelom nadogradnje bez promjene: želimo li dodati novi gumb, intervencije moramo raditi na više mjesta po kodu, a ujedno moramo i mijenjati već postojeći kod - konkretno metodu `obradiKlik` čija implementacija sada više nije dobra. Također, uočite da u kodu na jednom mjestu za gumb definiramo simboličko ime naredbe kao string a zatim to prepisujemo u metodi `obradiKlik` (pa možemo pogriješiti; ovo je primjer *magičnih brojeva* - samo što se ovdje ne radi o broju već stringu). Ovaj posljednji nedostatak bismo popravili tako da na vrhu razreda definiramo privatne statičke string-konstatne pa njih koristimo na svim potrebnim mjestima. Nažalost, taj pristup, iako rješava problem *magičnih brojeva*, uvodi još jedno mjesto na kojem treba modificirati kod pri dodavanju novih gumbi. Ono što bi svakako trebalo jest pronaći rješenje koje od nas ne zahtjeva centralizirane `switch`-pitalice.

Pokušaj 2

Kako bismo izbjegli pisanje `switch`-pitalice, napisat ćemo novo rješenje koje za svaki gumb koristi zasebnog promatrača. Dijelove koda iz `switch`-pitalice razmjestit ćemo po odgovara-

jućim promatračima. U ovom scenariju ne treba nam simboličko ime naredbe pa ga nećemo niti postavljati.

ovakvo rješenje prikazano je u nastavku (primjer 22.3). Tretirajte ga kao dopunu osnovnog koda (primjera 22.1).

Primjer 22.3. Uporaba zasebnih promatrača

```
 1 public class StdDemo1 extends JFrame {
 2
 3  ...
 4
 5  private void initGUI() {
 6   ...
 7   poruka1.addActionListener(e->{
 8    labela.setText("Dobrodošli!");
 9   });
10   poruka2.addActionListener(e->{
11    labela.setText("Učim Javu :-) :-) :-)");
12   });
13  }
14
15  ...
16 }
17
```

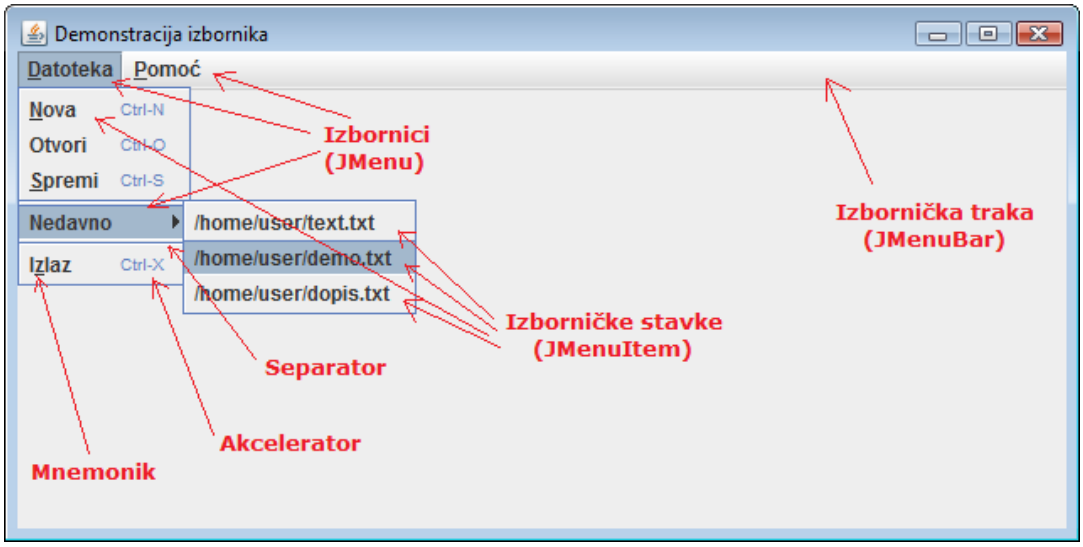
Uočite: svakom od gumba dodajemo njegovog vlastitog promatrača (gumb 1 - retci 7-9, gumb 2 - retci 10-12). U slučaju da je količina koda koju je potrebno izvršiti veća, umjesto pisanja čitavog koda preporuka je u osnovni razred dodati novu metodu (ili više njih) pa iz promatrača pozvati tu metodu.

Kakve su karakteristike ovog rješenja? Kod je svakako kvalitetniji - želimo li dodati novi gumb, ništa od postojećeg koda ne trebamo mijenjati; naprosto treba dodati novi gumb i njegovog promatrača.

Izbornici i izborničke strukture

Program koji razmatramo htjeli bismo doraditi tako da se promjene prikazanog teksta mogu pozivati i iz izbornika. U tu svrhu htjeli bismo dodati izbornik `Poruke` i u njemu izborničke stavke `Poruka 1` i `Poruka 2`. No prije toga, moramo se upoznati s načinom na koji se grade izborničke strukture (i terminologijom). Slika 22.5 prikazuje primjer izborničke strukture i komponenata kod kojih se ona gradi.

Slika 22.5. Primjer izborničke strukture



Izbornička traka modelirana je razredom `javax.swing.JMenuBar`. Ova traka ponaša se kao i bilo koja druga grafička komponenta pa je istu moguće smjestiti u bilo koju roditeljsku komponentu. Međutim, razred `JFrame` kojim je u Swingu modeliran prozor ima i zasebnu metodu `setJMenuBar(menubar)` koju se preporuča koristiti.

Pojedini izbornici (stavke čijom se aktivacijom otvara novi prozorčić sa stavkama) modelirani su razredom `javax.swing.JMenu`. Pojedine izborničke stavke (čija je predviđena namjena pokretanje zadanog koda) modelirane su razredom `javax.swing.JMenuItem`.

Pozivom metode `setMnemonic(m)` izborniku te izborničkoj stavki moguće je definirati mnemonik. Isti se korisniku prikazuje kao podcrtano slovo u ispisanom nazivu izbornika odnosno stavke. Akceleratorska kombinacija (kombinacija tipaka koja aktivira stavku neovisno o tome je li izbornik prikazan ili nije) zadaje se metodom `setAccelerator` a kao argument očekuje objekt tipa `javax.swing.KeyStroke`. Ovi objekti reprezentiraju pritisak ili otpuštanje kombinacije osnovne tipke i modifikatora (poput CTRL, SHIFT i ALT). Razred nudi nekoliko statičkih metoda tvornica a možda najjednostavnija za uporabu je ona koja prima tekstovni opis (primjerice "ctrl O", što se tumači kao pritisak tipke CTRL i tipke O).

Prilikom izgradnje izborničke strukture, izbornička traka ponaša se kao roditelj za izbornike (izbornike je izborničkoj traci potrebno dodati metodom `add(menu)`, a izbornik se ponaša kao roditelj izborničkim stavkama te ugniježđenim izbornicima (u roditelja se također dodaju metodom `add(menuitem)`). Separator se u izbornik može dodati pozivom metode `addSeparator()`, a s obzirom da su izborničke stavke samo posebna vrsta gumbi, ikone te promatrače koji se pozivaju aktiviranjem stavke moguće je postaviti na jednak način kao i kod drugih gumbi.

Programski kod koji stvara izborničku strukturu prikazanu na slici 22.5 dan u primjeru 22.4.

Primjer 22.4. Izgradnja izborničke strukture

```
 1 package book.java.swing.std;
 2
 3 import java.awt.event.KeyEvent;
 4
 5 import javax.swing.JFrame;
 6 import javax.swing.JMenu;
 7 import javax.swing.JMenuBar;
 8 import javax.swing.JMenuItem;
 9 import javax.swing.KeyStroke;
10 import javax.swing.SwingUtilities;
11 import javax.swing.WindowConstants;
```

```
12
13 public class Izbornici extends JFrame {
14
15  private static final long serialVersionUID = 1L;
16
17  public Izbornici() {
18   setTitle("Demonstracija izbornika");
19   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
20   initGUI();
21   pack();
22   setLocationRelativeTo(null);
23  }
24
25  private void initGUI() {
26   JMenuBar menuBar = createMenuBar();
27   setJMenuBar(menuBar);
28  }
29
30  private JMenuBar createMenuBar() {
31   JMenuBar menuBar = new JMenuBar();
32
33   JMenu fileMenu = new JMenu("Datoteka");
34   fileMenu.setMnemonic(KeyEvent.VK_D);
35
36   createMenuItem(
37    fileMenu, "Nova", KeyEvent.VK_N,
38    KeyStroke.getKeyStroke("ctrl N"),
39    "Stvaranje nove datoteke"
40   );
41   createMenuItem(
42    fileMenu, "Otvori", KeyEvent.VK_F,
43    KeyStroke.getKeyStroke("ctrl O"),
44    "Otvaranje datoteke"
45   );
46   createMenuItem(
47    fileMenu, "Spremi", KeyEvent.VK_S,
48    KeyStroke.getKeyStroke("ctrl S"),
49    "Spremanje datoteke"
50   );
51   fileMenu.addSeparator();
52
53   JMenu recent = new JMenu("Nedavno");
54   fileMenu.add(recent);
55
56   fileMenu.addSeparator();
57   createMenuItem(
58    fileMenu, "Izlaz", KeyEvent.VK_Z,
59    KeyStroke.getKeyStroke("ctrl X"),
60    "Izlaz iz programa"
61   );
62
63   createMenuItem(recent, "/home/user/text.txt",
64    -1, null, null);
65   createMenuItem(recent, "/home/user/demo.txt",
66    -1, null, null);
67   createMenuItem(recent, "/home/user/dopis.txt",
68    -1, null, null);
69
```

```
70    menuBar.add(fileMenu);
71
72    JMenu helpMenu = new JMenu("Pomoć");
73    helpMenu.setMnemonic(KeyEvent.VK_P);
74
75    createMenuItem(
76     helpMenu, "O programu", KeyEvent.VK_O,
77     KeyStroke.getKeyStroke("ctrl H"),
78     "Informacije o programu"
79    );
80
81    menuBar.add(helpMenu);
82
83    return menuBar;
84   }
85
86   private JMenuItem createMenuItem(JMenu parent, String text,
87    int mnemonic, KeyStroke accelerator, String tooltip) {
88
89    JMenuItem item = new JMenuItem();
90    item.setText(text);
91    if(mnemonic!=-1) {
92     item.setMnemonic(mnemonic);
93    }
94    if(accelerator!=null) {
95     item.setAccelerator(accelerator);
96    }
97    if(tooltip!=null) {
98     item.setToolTipText(tooltip);
99    }
100   parent.add(item);
101   return item;
102  }
103
104  public static void main(String[] args) {
105   SwingUtilities.invokeLater(() -> {
106    new Izbornici().setVisible(true);
107   });
108  }
109 }
110
```

Pokušaj 3

Dopunimo sada rješenje iz drugog pokušaja prikladnom izborničkom strukturom. Uz to, želimo napraviti još jednu malu modifikaciju: ne smijemo dozvoliti korisniku da dva puta za redom pokrene ispisivanje druge poruke. Rješenje je prikazano u primjeru 22.5. Kako smo radili dosta modifikacija, u primjeru je prikazan cjelovit kod.

Primjer 22.5. Dopuna izborničkom strukturom

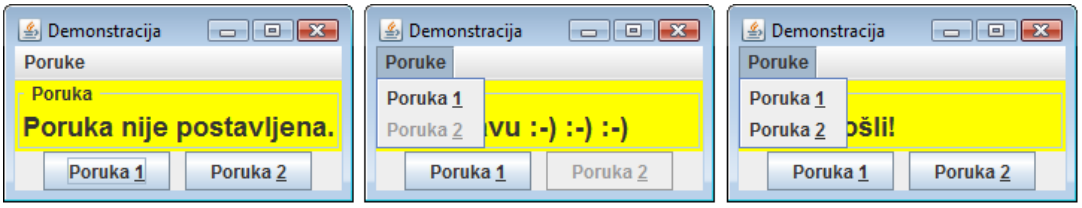
```
1 package book.java.swing.std;
2
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import java.awt.Container;
6 import java.awt.Dimension;
7 import java.awt.event.KeyEvent;
```

```
 8
 9 import javax.swing.BorderFactory;
10 import javax.swing.Box;
11 import javax.swing.BoxLayout;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JMenu;
16 import javax.swing.JMenuBar;
17 import javax.swing.JMenuItem;
18 import javax.swing.JPanel;
19 import javax.swing.SwingUtilities;
20 import javax.swing.WindowConstants;
21
22 public class StdDemo1 extends JFrame {
23
24  private static final long serialVersionUID = 1L;
25  private JLabel labela;
26
27  public StdDemo1() {
28   setTitle("Demonstracija");
29   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
30   initGUI();
31   pack();
32   setLocationRelativeTo(null);
33  }
34
35  private void initGUI() {
36   Container cp  = getContentPane();
37   cp.setLayout(new BorderLayout());
38
39   labela = new JLabel("Poruka nije postavljena.");
40   labela.setBorder(
41    BorderFactory.createTitledBorder(" Poruka ")
42   );
43   labela.setOpaque(true);
44   labela.setBackground(Color.YELLOW);
45   labela.setFont(
46    labela.getFont().deriveFont(
47     labela.getFont().getSize2D()*1.5f
48    )
49   );
50
51   JButton poruka1 = new JButton();
52   poruka1.setText("Poruka 1");
53   poruka1.setToolTipText("Prikazuje pozdrav.");
54   poruka1.setMnemonic(KeyEvent.VK_1);
55   JButton poruka2 = new JButton();
56   poruka2.setText("Poruka 2");
57   poruka2.setToolTipText("Ispisuje smajliće.");
58   poruka2.setMnemonic(KeyEvent.VK_2);
59
60   JPanel donjiPanel = new JPanel();
61   donjiPanel.setLayout(
62    new BoxLayout(donjiPanel, BoxLayout.LINE_AXIS)
63   );
64   donjiPanel.add(Box.createGlue());
65   donjiPanel.add(poruka1);
```

```
66    donjiPanel.add(Box.createRigidArea(new Dimension(10, 0)));
67    donjiPanel.add(poruka2);
68    donjiPanel.add(Box.createGlue());
69
70    cp.add(labela, BorderLayout.CENTER);
71    cp.add(donjiPanel, BorderLayout.PAGE_END);
72
73    JMenuBar menuBar = new JMenuBar();
74
75    JMenu porukeMenu = new JMenu("Poruke");
76    menuBar.add(porukeMenu);
77
78    JMenuItem poruka1MenuItem = new JMenuItem();
79    poruka1MenuItem.setText("Poruka 1");
80    poruka1MenuItem.setToolTipText("Prikazuje pozdrav.");
81    poruka1MenuItem.setMnemonic(KeyEvent.VK_1);
82    porukeMenu.add(poruka1MenuItem);
83
84    JMenuItem poruka2MenuItem = new JMenuItem();
85    poruka2MenuItem.setText("Poruka 2");
86    poruka2MenuItem.setToolTipText("Ispisuje smajliće.");
87    poruka2MenuItem.setMnemonic(KeyEvent.VK_2);
88    porukeMenu.add(poruka2MenuItem);
89
90    poruka1.addActionListener(e->{
91     labela.setText("Dobrodošli!");
92     poruka2MenuItem.setEnabled(true);
93     poruka2.setEnabled(true);
94    });
95    poruka2.addActionListener(e->{
96     labela.setText("Učim Javu :-) :-) :-)");
97     poruka2MenuItem.setEnabled(false);
98     poruka2.setEnabled(false);
99    });
100
101   poruka1MenuItem.addActionListener(e->{
102    labela.setText("Dobrodošli!");
103    poruka2MenuItem.setEnabled(true);
104    poruka2.setEnabled(true);
105   });
106   poruka2MenuItem.addActionListener(e->{
107    labela.setText("Učim Javu :-) :-) :-)");
108    poruka2MenuItem.setEnabled(false);
109    poruka2.setEnabled(false);
110   });
111
112   setJMenuBar(menuBar);
113  }
114
115  public static void main(String[] args) {
116   SwingUtilities.invokeLater(() -> {
117    new StdDemo1().setVisible(true);
118   });
119  }
120 }
121
```

Slika 22.6 prikazuje ovako modificirani program. Inicijalno, oba su gumba omogućena (lijevi dio slike). Nakon što korisnik klikne na desni gumb, ispisuje se poruka i taj gumb (kao i izborničkа stavka koja radi isto) su onemogućeni (središnji dio slike). Nakon što korisnik klikne na lijevi gumb, ispisuje se poruka a desni gumb kao i odgovarajuća izborničkа stavka ponovno postaju omogućeni.

Slika 22.6. Demonstracijski program (3)



Proanalizirajmo rješenje koje je prikazano u primjeru 22.5. Ako Vam se već površnim pregledom učinilo da tu nešto nije *baš kako bi trebalo biti* - u pravu ste. Kod je očajan. I ne, ne radi se o tome da ja nikad nisam zadovoljan kodom :-) Idemo argumentirati što tu nije dobro.

- U kodu postoji masovna redundancija. Usporedite retka 52-54 i retke 79-81; usporedite retke 56-58 i retke 85-87; usporedite potom retke 91-93 i retke 102-104 te retke 96-98 i retke 107-109. Dupliciraju se tekstovi, mnemonici, akceleratori kao i programski kod koji se pokreće aktiviranjem komponenata.
- Kod je jako povezan. Gumbi i akcije znaju jedni za druge i na više mjesta se radi omogućavanje i onemogućavanje pojedinih grafičkih komponenata.
- Prekršeno je načelo nadogradnje bez promjene. Zamislimo da želimo dodati još jedan način pokretanja ispisa poruka (primjerice, dodavanjem alatne trake - `JToolbar` i gumbima). U takvom scenariju još bismo jednom dupli cirali kod za pojedine ispise, a dodatno bismo morali proći kroz sav prethodno napisani kod i na svim mjestima na kojima radimo omogućavanje/onemogućavanje komponenata sada još dodati i komponente alatne trake.

Kako možemo preoblikovati kod da se riješimo svih istaknutih nedostataka?

Konačno rješenje: sučelje `Action`

Jedno moguće rješenje svih prethodno istaknutih nedostataka leži enkapsulaciji koda i svih opisnih podataka u jedan zaseban objekt, što omogućava sučelje `javax.swing.Action` koje je prikazano u nastavku.

```
 1 public interface Action extends ActionListener {
 2
 3   public Object getValue(String key);
 4   public void putValue(String key, Object value);
 5
 6   public void setEnabled(boolean b);
 7   public boolean isEnabled();
 8
 9   public void addPropertyChangeListener(
10       PropertyChangeListener listener);
11   public void removePropertyChangeListener(
12       PropertyChangeListener listener);
13
14   public static final String DEFAULT =
15                          "Default";
16   public static final String NAME =
17                          "Name";
18   public static final String SHORT_DESCRIPTION =
19                          "ShortDescription";
```

```
20   public static final String LONG_DESCRIPTION =
21                           "LongDescription";
22   public static final String SMALL_ICON =
23                           "SmallIcon";
24   public static final String ACTION_COMMAND_KEY =
25                           "ActionCommandKey";
26   public static final String ACCELERATOR_KEY =
27                           "AcceleratorKey";
28   public static final String MNEMONIC_KEY =
29                           "MnemonicKey";
30   public static final String SELECTED_KEY =
31                           "SwingSelectedKey";
32   public static final String DISPLAYED_MNEMONIC_INDEX_KEY =
33                           "SwingDisplayedMnemonicIndexKey";
34   public static final String LARGE_ICON_KEY =
35                           "SwingLargeIconKey";
36 }
```

Uočite: sučelje `javax.swing.Action` izvedeno je iz sučelja `java.awt.event.ActionListener` pa uz naslijeđenu apstraktnu metodu `actionPerformed` definira još tri para dodatnih metoda.

Metodom `setEnabled(e)` kompletna se akcija može omogućiti ili onemogućiti. Je li akcija omogućena ili nije, može se doznati metodom `isEnabled()`.

Sučelje `Action` također predviđa da se uz svaku akciju može definirati i niz popratnih podataka na sličan način kako se to radi s mapama: metodom `putValue(key, value)` pod zadanim ključem moguće je pohraniti predanu vrijednost a metodom `getValue(key)` iz akcije je moguće dohvatiti pohranjenu vrijednost. Pri tome se akcija s obzirom na pohranjene podatke i status omogućenosti ponaša kao subjekt koji nudi mogućnost dojavljivanja svih promjena pohranjenih podataka zainteresiranim promatračima koji su modelirani sučeljem `PropertyChangeListener`. Prijava i odjava takvih promatrača obavlja se metodama `addPropertyChangeListener(l)` i `removePropertyChangeListener(l)`.

Prilikom rada s akcijama ne preporuča se izravna uporaba ovog sučelja jer je u tom slučaju potrebno pisati dosta pomoćnog koda za pohranu spomenutih podataka, pohranu popisa promatrača te implementacija koda koji obavještava promatrače o nastalim promjenama. Umjesto toga, preporuča se koristiti apstraktni razred `javax.swing.AbstractAction` koji implementira sučelje `Action` i sve metode osim metode `actionPerformed`.

Sve grafičke komponente Swinga koje korisniku omogućavaju pokretanje neke akcije (poput klasičnih gumbi, izborničkih stavki, radioboxova i checkboxova) nude metodu `setAction(a)` kojom se akcija može postaviti toj komponenti. Komponenta će se pri tome sama inicijalizirati iz dodatnih podataka koji su pohranjeni u predanom objektu (primjerice, gumbi će pročitati što je pohranjeno uz ključ `Action.NAME` i to će prikazati kao tekst; inicijalizirat će ikone ako su definirane, mnemonike, akceleratorske tipke, tooltipove i slično). Da bi ovo funkcioniralo, podatci u akciju moraju biti zapisani prema unaprijed dogovorenim ključevima koji su u sučelju `Action` definirani kao statičke konstante.

Dodatno, takve će se grafičke komponente registrirati kao promatrači nad postavljenom akcijom i na svaku dojavu o promjeni podataka grafičke komponente će se samostalno i automatski ažurirati. Onemogućimo li akciju, sve grafičke komponente koje su povezane s tom akcijom također će se onemogućiti i korisnik ih neće moći aktivirati.

Rješenje koje se temelji na uporabi akcija prikazano je u primjeru 22.6.

Primjer 22.6. Rješenje uporabom akcija

```
1 package book.java.swing.std;
2
```

```
 3 import java.awt.BorderLayout;
 4 import java.awt.Color;
 5 import java.awt.Container;
 6 import java.awt.Dimension;
 7 import java.awt.event.ActionEvent;
 8 import java.awt.event.KeyEvent;
 9
10 import javax.swing.AbstractAction;
11 import javax.swing.Action;
12 import javax.swing.BorderFactory;
13 import javax.swing.Box;
14 import javax.swing.BoxLayout;
15 import javax.swing.JButton;
16 import javax.swing.JFrame;
17 import javax.swing.JLabel;
18 import javax.swing.JMenu;
19 import javax.swing.JMenuBar;
20 import javax.swing.JMenuItem;
21 import javax.swing.JPanel;
22 import javax.swing.KeyStroke;
23 import javax.swing.SwingUtilities;
24 import javax.swing.WindowConstants;
25
26 public class StdDemo5 extends JFrame {
27
28  private static final long serialVersionUID = 1L;
29  private JLabel labela;
30
31  private final Action poruka1Akcija = new AbstractAction() {
32   private static final long serialVersionUID = 1L;
33
34   {
35    this.putValue(Action.NAME, "Poruka 1");
36    this.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_1);
37    this.putValue(
38     Action.ACCELERATOR_KEY,
39     KeyStroke.getKeyStroke("ctrl 1"));
40    this.putValue(
41     Action.SHORT_DESCRIPTION,
42     "Prikazuje pozdrav.");
43   }
44
45   @Override
46   public void actionPerformed(ActionEvent e) {
47    labela.setText("Dobrodošli!");
48    poruka2Akcija.setEnabled(true);
49   }
50  };
51
52  private final Action poruka2Akcija = new AbstractAction() {
53   private static final long serialVersionUID = 1L;
54
55   {
56    this.putValue(Action.NAME, "Poruka 2");
57    this.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_2);
58    this.putValue(
59     Action.ACCELERATOR_KEY,
60     KeyStroke.getKeyStroke("ctrl 2"));
```

```
61     this.putValue(
62      Action.SHORT_DESCRIPTION,
63      "Ispisuje smajliće.");
64    }
65
66    @Override
67    public void actionPerformed(ActionEvent e) {
68     labela.setText("Učim Javu :-) :-) :-)");
69     setEnabled(false);
70    }
71  };
72
73  public StdDemo5() {
74   setTitle("Demonstracija");
75   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
76   initGUI();
77   pack();
78   setLocationRelativeTo(null);
79  }
80
81  private void initGUI() {
82   Container cp  = getContentPane();
83   cp.setLayout(new BorderLayout());
84
85   labela = new JLabel("Poruka nije postavljena.");
86   labela.setBorder(
87    BorderFactory.createTitledBorder(" Poruka ")
88   );
89   labela.setOpaque(true);
90   labela.setBackground(Color.YELLOW);
91   labela.setFont(
92    labela.getFont().deriveFont(
93     labela.getFont().getSize2D()*1.5f
94    )
95   );
96
97   JButton poruka1 = new JButton();
98   poruka1.setAction(poruka1Akcija);
99
100  JButton poruka2 = new JButton();
101  poruka2.setAction(poruka2Akcija);
102
103  JPanel donjiPanel = new JPanel();
104  donjiPanel.setLayout(
105   new BoxLayout(donjiPanel, BoxLayout.LINE_AXIS)
106  );
107  donjiPanel.add(Box.createGlue());
108  donjiPanel.add(poruka1);
109  donjiPanel.add(Box.createRigidArea(new Dimension(10, 0)));
110  donjiPanel.add(poruka2);
111  donjiPanel.add(Box.createGlue());
112
113  cp.add(labela, BorderLayout.CENTER);
114  cp.add(donjiPanel, BorderLayout.PAGE_END);
115
116  JMenuBar menuBar = new JMenuBar();
117
118  JMenu porukeMenu = new JMenu("Poruke");
```

```
119    menuBar.add(porukeMenu);
120
121    JMenuItem poruka1MenuItem = new JMenuItem();
122    poruka1MenuItem.setAction(poruka1Akcija);
123    porukeMenu.add(poruka1MenuItem);
124
125    JMenuItem poruka2MenuItem = new JMenuItem();
126    poruka2MenuItem.setAction(poruka2Akcija);
127    porukeMenu.add(poruka2MenuItem);
128
129    setJMenuBar(menuBar);
130   }
131
132   public static void main(String[] args) {
133    SwingUtilities.invokeLater(() -> {
134     new StdDemo5().setVisible(true);
135    });
136   }
137  }
138
```

U prikazanom rješenju najprije smo pripremili dva objekta od kojih svaki predstavlja jednu akciju. Prvi objekt (i akcija) definiran je u retcima 31-50 a drugi u retcima 52-71. Uočite da smo, s obzirom da koristimo anonimni razred izveden iz `AbstractAction`, inicijalizaciju obavili u inicijalizacijskom bloku (retci 34-43 odnosno 55-64) jer ne možemo koristiti konstruktor. Za svaku akciju uz inicijalizacijski blok definirali smo još i metodu `actionPerformed` u koju smo smjestili programski kod koji treba obaviti pokretanjem akcije.

Uz ovakvu strukturu programa, pogledajte na što se je svela inicijalizacija grafičkih komponenti: stvorimo gumb i postavimo mu akciju (npr. retci 97-98); stvorimo izborničku stavku i postavimo joj akciju (npr. retci 121-122). Kod je puno sažetiji; nema dupliciranja i kod jest nadogradiv - želimo li dodati alatnu traku, sve što treba je stvoriti je, stvoriti njezine gumbe i predati im referencu na ove akcije. "Stari kod" pri nadogradnji uopće ne trebamo modificirati.



Bilješka

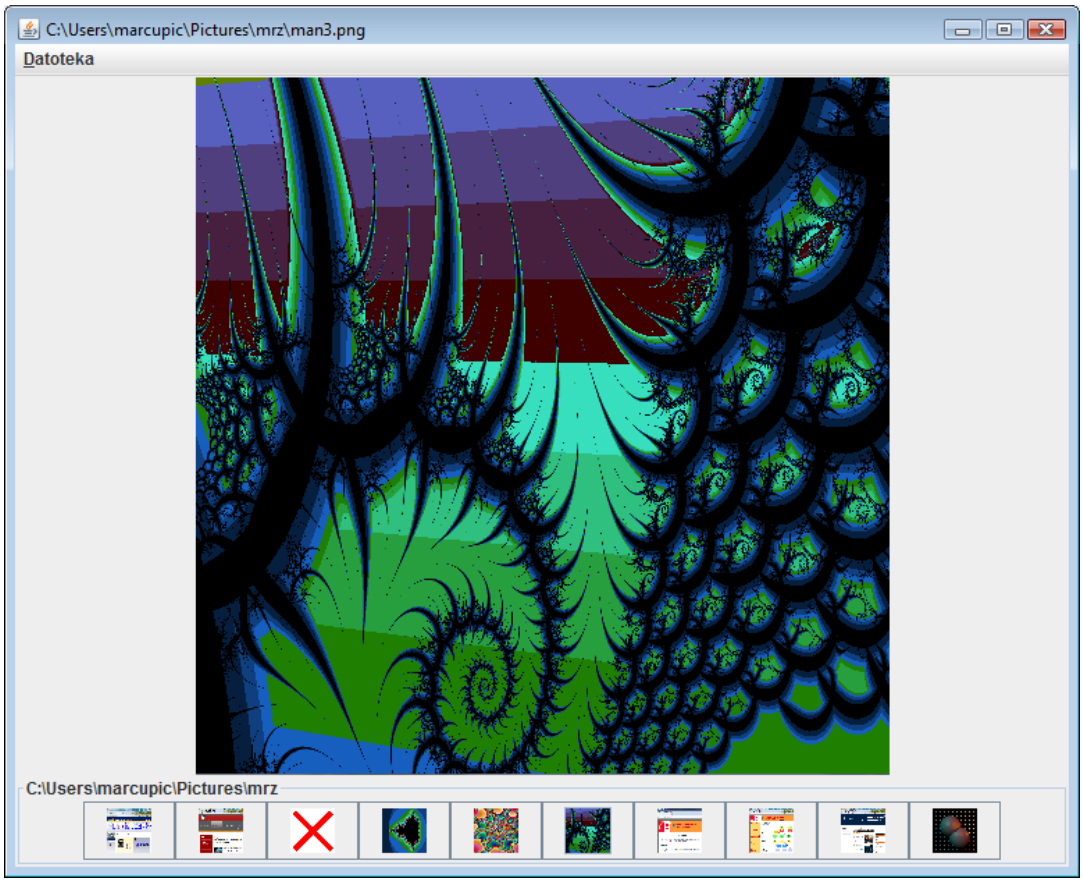
Pri izradi grafičkih programa koristite enkapsulaciju akcija u objekte tipa `Action`: to donosi niz prednosti a uklanja mnoge mane. Također, programe koji se temelje na ovako enkapsuliranim akcijama lagano je internacionalizirati: sve što treba je mijenjati tekst u akcijama - grafičke komponente automatski će ga preuzeti.

Preglednik slika. Razred `SwingWorker`

Sljedeći primjer koji ćemo razraditi je preglednik slika. Progam korisniku treba ponuditi izbornik kroz koji korisnik može odabrati direktorij u kojem se nalaze slike. Program potom treba proći kroz taj direktorij, učitati u memoriju sve slike, za svaku sliku generirati manju sličicu te na dnu prozora u traku za sličice dodati po jedan gumb za svaku sliku koji na površini prikazuje umanjenu inačicu slike. Klikom na taj gumb, u središnjem se dijelu prozora treba prikazati slika (normalne razlučivosti).

Željeni izgled programa prikazan je na slici 22.7.

Slika 22.7. Jednostavan preglednik slika



Ako se neka slika ne može učitati (pogreška u formatu datoteke ili bilo koji drugi problem), na gumbu za takvu sličicu treba nacrtati crveni znak X na bijeloj podlozi. Na slici 22.7 to možemo vidjeti na trećem gumbu.

Potreban programski kod prikazan je u primjeru 22.7.

Primjer 22.7. Jednostavan preglednik slika

```
 1 package book.java.swing.std;
 2
 3 import java.awt.BasicStroke;
 4 import java.awt.BorderLayout;
 5 import java.awt.Color;
 6 import java.awt.Component;
 7 import java.awt.Container;
 8 import java.awt.Graphics;
 9 import java.awt.Graphics2D;
10 import java.awt.Image;
11 import java.awt.MediaTracker;
12 import java.awt.RenderingHints;
13 import java.awt.event.ActionEvent;
14 import java.awt.event.KeyEvent;
15 import java.awt.image.BufferedImage;
16 import java.nio.file.DirectoryStream;
17 import java.nio.file.Files;
18 import java.nio.file.Path;
19 import java.util.List;
20
21 import javax.swing.AbstractAction;
```

```
22 import javax.swing.Action;
23 import javax.swing.BorderFactory;
24 import javax.swing.Box;
25 import javax.swing.BoxLayout;
26 import javax.swing.Icon;
27 import javax.swing.ImageIcon;
28 import javax.swing.JButton;
29 import javax.swing.JFileChooser;
30 import javax.swing.JFrame;
31 import javax.swing.JLabel;
32 import javax.swing.JMenu;
33 import javax.swing.JMenuBar;
34 import javax.swing.JMenuItem;
35 import javax.swing.JPanel;
36 import javax.swing.KeyStroke;
37 import javax.swing.SwingConstants;
38 import javax.swing.SwingUtilities;
39 import javax.swing.SwingWorker;
40 import javax.swing.WindowConstants;
41
42 public class PregledniкSlika extends JFrame {
43
44  private static final long serialVersionUID = 1L;
45  private static final int ICON_WIDTH = 32;
46  private static final int ICON_HEIGHT = 32;
47
48  private JLabel label;
49  private JPanel panel;
50
51  public PregledniкSlika() {
52   setTitle("Preglednik slika");
53   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
54   initGUI();
55   setSize(500, 500);
56   setLocationRelativeTo(null);
57  }
58
59  private void initGUI() {
60   JMenuBar menuBar = createMenuBar();
61   setJMenuBar(menuBar);
62
63   Container cp = getContentPane();
64   cp.setLayout(new BorderLayout());
65
66   label = new JLabel();
67   label.setHorizontalAlignment(SwingConstants.CENTER);
68   label.setVerticalAlignment(SwingConstants.CENTER);
69
70   panel = new JPanel();
71   BoxLayout layout = new BoxLayout(panel, BoxLayout.LINE_AXIS);
72   panel.setLayout(layout);
73   panel.add(Box.createGlue());
74   panel.add(Box.createGlue());
75
76   cp.add(label, BorderLayout.CENTER);
77   cp.add(panel, BorderLayout.PAGE_END);
78  }
79
```

```
 80  private JMenuBar createMenuBar() {
 81   JMenuBar menuBar = new JMenuBar();
 82
 83   JMenu fileMenu = new JMenu("Datoteka");
 84   fileMenu.setMnemonic(KeyEvent.VK_D);
 85
 86   fileMenu.add(new JMenuItem(odaberiAkcija));
 87   fileMenu.add(new JMenuItem(izlazAkcija));
 88
 89   menuBar.add(fileMenu);
 90
 91   return menuBar;
 92  }
 93
 94  private void clearImages() {
 95   int n = panel.getComponentCount();
 96   for(int i = n-2; i >= 1; i--) {
 97    panel.remove(i);
 98   }
 99   setTitle("");
100   label.setIcon(null);
101  }
102
103  protected void izlaz() {
104   System.exit(0);
105  }
106
107  protected void otvori() {
108   JFileChooser jfc = new JFileChooser();
109   jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
110   jfc.setMultiSelectionEnabled(false);
111   if(JFileChooser.APPROVE_OPTION != jfc.showOpenDialog(this)) {
112    return;
113   }
114   Path directory = jfc.getSelectedFile().toPath();
115   clearImages();
116   panel.setBorder(
117    BorderFactory.createTitledBorder(directory.toString())
118   );
119   new Posao(directory).execute();
120  }
121
122  private final Action odaberiAkcija = new AbstractAction() {
123   private static final long serialVersionUID = 1L;
124
125   {
126    this.putValue(Action.NAME, "Odaberi direktorij");
127    this.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_O);
128    this.putValue(
129     Action.ACCELERATOR_KEY, KeyStroke.getKeyStroke("ctrl O"));
130    this.putValue(
131     Action.SHORT_DESCRIPTION, "Odabir direktorija");
132   }
133
134   @Override
135   public void actionPerformed(ActionEvent e) {
136    otvori();
137   }
```

```
138  };
139
140  private final Action izlazAkcija = new AbstractAction() {
141   private static final long serialVersionUID = 1L;
142
143   {
144    this.putValue(Action.NAME, "Izlaz");
145    this.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_I);
146    this.putValue(
147     Action.ACCELERATOR_KEY, KeyStroke.getKeyStroke("ctrl X"));
148    this.putValue(
149     Action.SHORT_DESCRIPTION, "Izlaz iz programa");
150   }
151
152   @Override
153   public void actionPerformed(ActionEvent e) {
154    izlaz();
155   }
156  };
157
158  private class PrikaziSliku extends AbstractAction {
159   private static final long serialVersionUID = 1L;
160   private Path path;
161   private Icon bigImage;
162
163   public PrikaziSliku(String name, Path path,
164    Icon bigImage, Icon smallIcon) {
165
166    super(name);
167    this.path = path;
168    this.bigImage = bigImage;
169    putValue(Action.LARGE_ICON_KEY, smallIcon);
170   }
171
172   @Override
173   public void actionPerformed(ActionEvent e) {
174    label.setIcon(bigImage);
175    setTitle(path.toString());
176   }
177  }
178
179  private class Posao extends SwingWorker<Void, Action> {
180
181   private Path directory;
182
183   public Posao(Path directory) {
184    super();
185    this.directory = directory;
186   }
187
188   @Override
189   protected Void doInBackground() throws Exception {
190    try(DirectoryStream<Path> stream =
191     Files.newDirectoryStream(directory)) {
192
193     for(Path file : stream) {
194      if(!isSupported(file)) continue;
195      ImageIcon bigImage = new ImageIcon(file.toUri().toURL());
```

```
196       Icon smallIcon;
197       if(bigImage.getImageLoadStatus()==MediaTracker.ERRORED
198        || bigImage.getIconWidth()<0) {
199        smallIcon = new ErrorIcon();
200       } else {
201        smallIcon = createSmallIcon(bigImage.getImage());
202       }
203       publish(new PrikaziSliku(
204        file.getFileName().toString(), file, bigImage, smallIcon
205       ));
206      }
207     }
208     return null;
209    }
210
211    private boolean isSupported(Path file) {
212     if(!Files.isRegularFile(file)) return false;
213     String name = file.getFileName().toString();
214     int pos = name.lastIndexOf('.');
215     if(pos<1) return false;
216     String ext = name.substring(pos+1).toUpperCase();
217     return ext.equals("JPG")
218      || ext.equals("PNG")
219      || ext.equals("GIF");
220    }
221
222    public ImageIcon createSmallIcon(Image image) {
223     BufferedImage bim = new BufferedImage(ICON_WIDTH, ICON_HEIGHT,
224      BufferedImage.TYPE_3BYTE_BGR);
225     Graphics2D g = (Graphics2D)bim.createGraphics();
226     g.setRenderingHint(
227      RenderingHints.KEY_INTERPOLATION,
228      RenderingHints.VALUE_INTERPOLATION_BICUBIC);
229     g.drawImage(image, 0, 0, ICON_WIDTH, ICON_HEIGHT, null);
230     g.dispose();
231     return new ImageIcon(bim);
232    }
233
234    @Override
235    protected void process(List<Action> chunks) {
236     for(Action a : chunks) {
237      int index = panel.getComponentCount()-1;
238      JButton btn = new JButton();
239      btn.setHideActionText(true);
240      btn.setRolloverEnabled(true);
241      btn.setContentAreaFilled(false);
242      btn.setAction(a);
243      panel.add(btn, index);
244     }
245     panel.revalidate();
246    }
247   }
248
249   public class ErrorIcon implements Icon {
250    @Override
251    public int getIconWidth() {
252     return ICON_WIDTH;
253    }
```

```
254   @Override
255   public int getIconHeight() {
256    return ICON_HEIGHT;
257   }
258   @Override
259   public void paintIcon(Component c, Graphics g, int x, int y) {
260    Graphics2D g2 = (Graphics2D)g.create();
261    try {
262     g2.setColor(Color.WHITE);
263     g2.fillRect(x, y, getIconWidth(), getIconHeight());
264
265     g2.setStroke(new BasicStroke(4f));
266     g2.setColor(Color.RED);
267     g2.drawLine(
268      x+4, y+4,
269      x+getIconWidth()-4, y+getIconHeight()-4
270     );
271     g2.drawLine(
272      x+4, y+getIconHeight()-4,
273      x+getIconWidth()-4, y+4
274     );
275    } finally {
276     g2.dispose();
277    }
278   }
279  }
280
281  public static void main(String[] args) {
282   SwingUtilities.invokeLater(() -> {
283    new PreglednikSlika().setVisible(true);
284   });
285  }
286
287 }
288
```

Za prikaz središnje slike koristimo komponentu `JLabel` kojoj sliku postavljamo kao ikonu i labelu koristimo bez teksta. Sadržaj labele je centriran. Na dno prozora smo dodali jedan `JPanel` kojim upravlja `BoxLayout` te smo u njega inicijalno dodali samo dvije *glue*-komponente. Gumbe koje ćemo dodavati, ubacivat ćemo između te dvije komponente čime će isti završiti centrirani.

U programu imamo tri akcije. Dvije su povezane izravno s izborničkom strukturom i omogućavaju izlazak iz programa (retci 140-156) te odabiranje direktorija u kojem će se tražiti slike (retci 122-138); ova posljednja akcija samo poziva metodu `otvori` koja je prikazana u retcima 107-120. U toj metodi za odabir direktorija koristimo ugrađenu dijaloško kutiju `JFileChooser` koju smo prikladno podesili. Ovaj kod, međutim, ne obavlja i fizičko skeniranje direktorija te učitavanje slika jer je to vremenski vrlo skupa operacija i nju ne želimo pokrenuti na EDT-dretvi jer bismo tako na određeno vrijeme zablokirali čitavo grafičko korisničko sučelje. Umjesto toga, taj posao odrađujemo u novoj dretvi za što smo iskoristili pomoćni razred `SwingWorker` (odnosno razred koji smo iz njega izveli: `Posao`).

Uporaba razreda `SwingWorker` (odnosno njegovih izvedenica) zamišljena je na sljedeći način.

Iz bilo dretve (može biti EDT ali i ne mora) stvara se primjerak ovog razreda, podešava i potom se nad njim poziva metoda `execute()`. Ovaj poziv nije blokirajući i ne izvodi posao. Umjesto toga, on samo pokreće pozivanje metode `doInBackground()` u *novoj pomoćnoj* dretvi (odnosno pozadinskoj dretvi); ta dretva *nije* EDT i ona *ne smije* ništa raditi s korisničkim sučeljem.

Ako u toj metodi želimo izravno modificirati korisničko sučelje, te modifikacije moramo oblikovati kao Runnable objekte i pozvati uporabom metode SwingUtilities.invokeLater(r) ili pak metode SwingUtilities.invokeAndWait(r).

Kako je to dosta nepraktično (i ružno u kodu), SwingWorker nudi alternativni pristup: pravimo se da naš posao "proizvodi" određen broj objekata. Svaki puta kada je neki objekt "proizveden", možemo ga proslijediti SwingWorkeru pozivom metode publish izravno iz dretve koja obavlja metodu doInBackground(). Tim će pozivom objekt biti pohranjen u pomoćni red objekata. SwingWorker će periodički vaditi tako pospremljene objekte i predavati ih zapakirane u listu kao argument metode process koju će on sam *pozvati iz EDT-a*. Posljedica je da u metodi process smijemo modificirati korisničko sučelje jer se taj kod izvodi u EDT-u.

Naša implementacija razreda koji obavlja sav posao a koji je izveden iz razreda SwingWorker prikazana je u retcima 179-247. Pogledajte kako smo riješili metode doInBackground() i process(...).

U metodi doInBackground() prolazimo kroz sve datoteke u zadanom direktoriju. Ako je datoteka slika, pokušavamo je učitati uporabom razreda ImageIcon (koji implementira sučelje Icon), te zatim dohvaćamo referencu na samu sliku te stvaramo skaliranu inačicu (stvaramo novu sliku BufferedImage i objektom za crtanje po toj slici nacrtamo originalnu sliku uz umanjivanje, pa sve zamotamo natrag u novi ImageIcon). Potom stvaramo novi primjerak akcije PrikaziSliku (definirana u retcima 158-177) koji inicijaliziramo ovim slikama i tako dobiveni objekt (akciju) šaljemo u metodu publish.

Nakon određenog vremena, ti će se objekti pojaviti u metodi process koju izvodi EDT. Stoga na tom mjestu iteriramo kroz svaku akciju, stvaramo novi primjerak gumba, na taj primjerak postavljamo primljenu akciju i gumb umećemo kao predzadnju komponentu donje ploče (sjetite se: zadnja je desni *glue*). Nakon što smo završili ove modifikacije, obavezno još moramo pozvati revalidate() (sjećate li se zašto?).

Ako se prilikom učitavanja slike dogodi pogreška, onda tu sliku ne možemo skalirati već moramo generirati manju ikonicu koja prikazuje crveni znak X na bijeloj pozadini. To bismo mogli nacrtati u nekom grafičkom programu, pospremiti pa pokušati učitati na jednak način na koji učitavamo i ostale slike. No što ako se tu dogodi pogreška? Pristup koji je ovdje prikazan ide drugim putem. U retcima 249-279 prikazana je naša vlastita implementacija sučelja Icon koja nadjačava metodu paintIcon i u njoj samostalno obavlja crtanje. Primjerke ovog razreda koristimo kao ikonu svaki puta kada nam ne uspije učitavanje slike s diska.

Više informacija o razredu SwingWorker moguće je pronaći u njegovoj dokumentaciji. Metoda doInBackground na kraju može vratiti "konačan proizvod". Ako se to koristi, istoga pozivatelj može dohvatiti pozivom metode get() koja će ga zablokirati sve dok metoda doInBackground ne bude gotova. Uz to, postoji i jednostavan mehanizam za dojavljivanje postotka obavljenog posla (metoda setProgress(i)) koji je moguće pratiti prijavom odgovarajućeg promatrača.



Za vježbu

1. Pokušajte modificirati program tako da u memoriji ne čuva sve slike nativne rezolucije, već da se slika za potrebe prikaza u središnjoj komponenti učitava na zahtjev. Za izradu malih sličica morat ćete jednom proći kroz sve slike.

2. Umjesto središnje komponente JLabel napišite vlastitu komponentu koja će sliku, ako je veća od dimenzije komponente, skalirati (poštujući omjer njezine širine i visine) tako da se vidi čitava slika. Ako je slika manja, onda se ne razvlači već se prikazuje takva kakva je ali centrirano.

3. Ako odustanemo od zahtjeva da gumbići za prikaz slika moraju biti centrirani u prevelikoj komponenti, napravite modifikaciju koda uz koju će se, u slučaju da ima previše slika, pojaviti traka za skrolanje. Naša trenutna implementacija se u tom scenariju neće provesti baš bajno (isprobajte).

Poglavlje 23. Grafičko korisničko sučelje: Swing - modeli i pogledi

Uporaba višestruko iskoristivih komponenata jedna je od središnjih tema grafičke biblioteke Swing. A ideja je vrlo jednostavna: ako trebamo tablicu koja podatke čuva u memoriji i tablicu koja podatke SQL-om dohvaća iz relacijske baze podataka i tablicu koja podatke čita iz datoteke te tablicu koja podatke dohvaća preko web-usluge, programeru treba ponuditi takvo rješenje koje u cijelosti izbjegava (ili barem minimizira) ponavljanje koda.

Postoji nekoliko prikladnih rješenje opisanog problema a danas su u praksi najčešća ona temeljena na oblikovnom obrascu Model-Pogled-Upravljač (engl. *Model-View-Controller design pattern (MVC)*) te njegovim inačicama.

Ideja svih ovih pristupa jest razdvojiti grafičku komponentu koja omogućava prikaz i uređivanje podataka od samih podataka. U takvom scenariju, komponentu za prikaz podataka nazivat ćemo *Pogled* dok ćemo podatke modelirati prikladnim sučeljem koje će definirati *Model podataka*. Pri tome ćemo *Pogled* pri inicijalizaciji konfigurirati referencom na konkretan *Model* iz kojeg će *Pogled* dohvaćati podatke koje treba prikazivati.

U slučaju da *Model* programeru omogućava dinamičku izmjenu podataka (ili se isti na neki drugi način mogu mijenjati - primjerice, model sadrži kod koji nešto mjeri sa senzora svake tri sekunde ili pak preko Interneta periodički provjerava i dohvaća određene podatke), potrebno je ugraditi mehanizam kojim ćemo moći osigurati da je prikaz koji postoji u *Pogledu* usklađen s podatcima u *Modelu*. Tipično rješenje koje se za to koristi jest oblikovni obrazac *Promatrač*: *Model* je *Subjekt* koji nudi metode za registraciju *Promatrača* te brine da pri svakoj promjeni podataka o tome obavijesti sve prijavljene promatrače; *Pogled* je *Promatrač* koji će se prijaviti nad *Subjektom* za dobivanje informacija o promjenama te će svaki puta kada ga *Subjekt* obavijesti o promjeni ažurirati svoj prikaz (primjerice, pozivom metode `repaint();` zatražit će od grafičkog podsustava da što prije pokrene novo iscrtavanje).

Zapamtite ovu ideju: ona je središnja tema oko koje je izveden niz komponenata Swinga.



Važno

Velik dio komponenata Swinga utemeljen je na ideji pogleda i modela. Model enkapsulira podatke i nudi sučelje za prijavu zainteresiranih promatrača. Pogled je grafička komponenta koja dobiva referencu na model i sebe prijavljuje kao promatrača nad tim modelom. Kad god se u modelu promijene podatci, model prijavljenim promatračima (u ovom slučaju pogledu) dojavljuje promjene; time pogled može automatski ažurirati svoj prikaz i ostati usklađen s podatcima.

Razumijete li na koji način ovakva organizacija uklanja dupliciranje koda? Sada umjesto da pišemo više inačica grafičkih komponenata (za svaki od konkretnih izvora podataka po jednu), uspjeli smo izvor podataka modelirati kao zasebnu komponentu. Zahvaljujući tome, programski kod grafičke komponente napisan je samo jednom i on može raditi s različitim implementacijama izvora podataka. Za svaki (različit) konkretan izvor podataka potrebno je napisati odgovarajući razred. No kako se ionako radi o novoj vrsti izvora podataka, taj programski kod nemamo nigdje drugdje - i ne radimo dupliciranje koda.

Neke od često korištenih komponenata koje su organizirane na ovaj način prikazane su u nastavku.

Tablica 23.1. Sample Table

Grafička komponenta	Sučelje modela	Sučelje promatrača	Informacije o promjeni
JList	ListModel, AbstractListModel, DefaultListModel	ListDataListener	ListDataEvent
JTable	TableModel, AbstractTableModel, DefaultTableModel	TableModelListener	TableModelEvent
JTree	TreeModel, DefaultTreeModel	TreeModelListener	TreeModelEvent
JSpinner	SpinnerModel, AbstractSpinnerModel	ChangeListener	ChangeEvent
JTextComponent, JTextArea, JTextField, JEditorPane, JTextPane	Document, PlainDocument, StyledDocument, DefaultStyledDocument, HTMLDocument	DocumentListener	DocumentEvent

Prvi stupac tablice prikazuje grafičke komponente za različite vrste modela. U drugom stupcu navedeno je osnovno sučelje koje definira model podataka koje očekuje grafička komponenta te sučelja i razredi koji proširuju odnosno implementiraju to sučelje. Na nekoliko mjesta može se uočiti uzorak imenovanja XModel, AbstractXModel, DefaultXModel. U tom slučaju prvo predstavlja osnovno sučelje modela, drugo je apstraktni razred koji implementira to sučelje ali od metoda implementira samo metode za prijavu i odjavu promatrača (tako da to ne mora raditi programer); treće predstavlja konkretan (neapstraktni) razred koji nasljeđuje prethodni apstraktni razred i nudi pretpostavljenu implementaciju te dodaje metode koje programeru omogućavaju da radi s podatcima modela. Opisana implementacija podatke čuva izravno u memoriji i ima je smisla koristiti tamo gdje je to prihvatljivo. U svim ostalim slučajevima uobičajeno je napisati novi razred koji naslijedi apstraktni i nudi implementacije metoda za rad s podatcima koje zahtjeva osnovno sučelje.

U trećem stupcu navedeno je sučelje promatrača koje propisuje model i koje moraju implementirati komponente koje žele biti obaviještavane o promjenama u modelu. Konceptualno, to sučelje implementira grafička komponenta. U praksi, to ne mora biti baš grafička komponenta već neki njezin anonimni ugniježđeni razred (koji ima pristup komponenti) a moguća su i druga rješenja. Međutim, kakva god da je izvedba, efekt je da se korisniku komponente čini da je prikaz komponenta uvijek u skladu s podatcima modela.

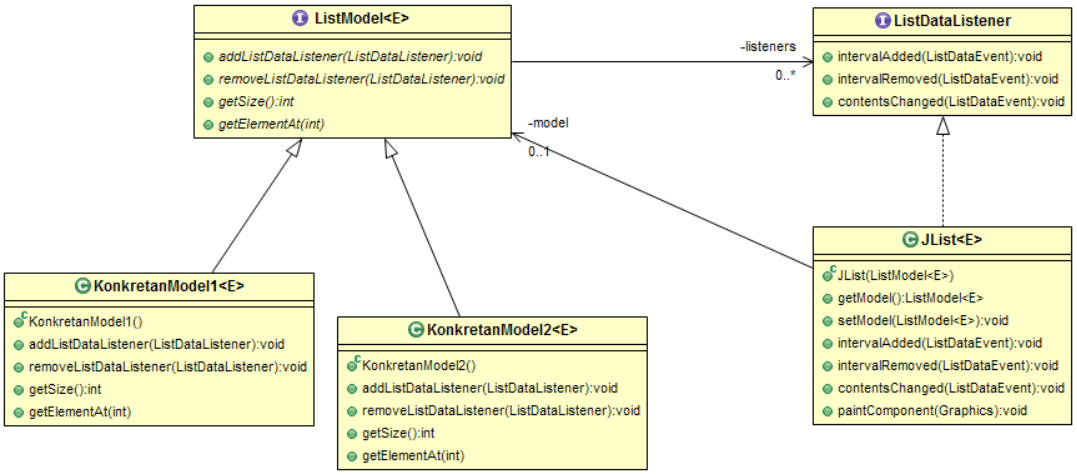
Konačno, u četvrtom stupcu naveden je tip objekta koji model stvara prilikom nastanka bilo koje promjene i koji puni informacijama o toj promjeni. Referencu na taj objekt model šalje promatračima tijekom obaviještavanja o nastalim promjenama.

U nastavku ovog poglavlja osvrnut ćemo se na nekoliko osnovnih komponenata i njihove modele. Ideja ovog poglavlja jest dobiti osjećaj za način na koji su komponente međusobno povezane i kako to sve radi u praksi. Potom je na čitatelju da se po potrebi upozna s detaljima uporabe konkretnih komponenata u trenutku kada mu to doista i zatreba. Razumijete li osnovnu ideju, dokumentacija komponenata i službeni tutorijal na stranicama Oraclea sasvim su dovoljni da se vrlo brzo snađete.

Prikaz liste

U ovom odjeljku upoznat ćemo se s grafičkom komponentom za prikaz jednostavne liste (razred `javax.swing.JList`) te pripadnog modela podataka definiranog parametriziranim sučeljem `javax.swing.ListModel<E>`. Konceptualni odnos ovih dviju komponenata prikazan je na slici 23.1.

Slika 23.1. JList i pripadni model podataka (konceptualni prikaz)



Konceptualno, model podataka opisan sučeljem `javax.swing.ListModel<E>` sastoji se od niza elemenata tipa `E` koji su poredani i imaju pridružen redni broj. Numeracija elemenata uvijek započinje od 0.

Uzevši to u obzir, sučelje modela definira dvije metode - `size` i `getElementAt` kojima pogled može dobiti informacije o podatcima. Metodom `int size()` klijent modela može dobiti informaciju koliko je elemenata prisutno a metodom `E getElementAt(int index)` može zatražiti dohvat elementa za poziciji zadanoj argumentom `index`.

Model također nudi mogućnost prijave zainteresiranih promatrača koji žele biti obaviješavani o promjenama u podatcima, te njihove odjave. S obzirom na apstrakciju podataka koje nudi, model predviđa mogućnost obavljanja tri vrste operacija nad postojećim nizom: umetanje novih elemenata, brisanje kontinuiranog podniza elemenata te općenita promjena elemenata unutar nekog podniza. Sukladno tim operacijama, promatrači modela opisani su sučeljem `ListDataListener` koje definira tri metode: `intervalAdded(e)`, `intervalRemoved(e)` te `contentsChanged(e)`. Sve tri metode detaljniju informaciju o nastaloj izmjeni prenose kroz argument koji je objekt tipa `ListDataEvent` i koji definira četiri podatka: (1) `source` (referenca na sam model), `type` (vrsta izmjene; može biti jedna od konstanti `ListDataEvent.CONTENTS_CHANGED`, `ListDataEvent.INTERVAL_ADDED` ili `ListDataEvent.INTERVAL_REMOVED`), `index0` (indeks koji predstavlja poziciju jednog kraja intervala gdje se dogodila promjena) te `index1` (indeks koji predstavlja poziciju drugog kraja intervala gdje se dogodila promjena).

Evo jednostavnog primjera. Elemente ćemo označavati velikim slovima. Neka je početni sadržaj modela niz elemenata [A, B, C, D] (na poziciji 0 je A, na poziciji 1 je B, itd). Pozivom nad modelom `getElementAt(0)` dobili bismo element A. Pretpostavimo sada da na poziciju 3 želimo umetnuti podniz [E, F, G]: time će novo stanje modela biti [A, B, C, E, F, G, D]. Pozicije na kojima se nalaze umetnuti elementi su 3, 4 i 5 pa se radi o intervalu pozicija 3 do 5. Objekt tipa `ListDataEvent` koji bi trebao popratiti ovu izmjenu imao bi `type = ListDataEvent.INTERVAL_ADDED`, `index0 = 3`, `index1 = 5`.

Pretpostavimo da smo kao sljedeću operaciju nad modelom odlučili obrisati elemente na pozicijama 1 do 3. Novo stanje modela bit će [A, F, G, D] a tu će promjenu popratiti objekt tipa `ListDataEvent` inicijaliziran s `type = ListDataEvent.INTERVAL_REMOVED`, `index0 = 1`, `index1 = 3`.

Konačno, pretpostavimo da nam model nudi metodu kojom možemo zatražiti zamijenu dvaju susjednih elemenata, pa da smo takvom metodom zamijenili elemente na pozicijama 2 i 3. Novo stanje modela bit će [A, F, D, G], i takvu promjenu možemo opisati objektom tipa `ListDataEvent` koji bi bio inicijaliziran s `type = ListDataEvent.CONTENTS_CHANGED`, `index0`

`= 2, index1 = 3`. Alternativno, ovu izmjenu mogli bismo opisati kroz slijed od dvije promjene: mogli bismo reći da podinterval 2 do 2 obrisan (element G) pa da je umetnut podinterval 3-3 (G smo umetnuli na kraj). Druga mogućnost je da javimo da smo obrisali podinterval 3 do 3 (element D), pa da smo zatim umetnuli podinterval 2 do 2 (D je umetnut ispred G).

Općenito, ako je promjena složenija, umjesto razlaganja na niz umetanja i brisanja jednostavnije je javiti da je "došlo do neke promjene" (CONTENTS_CHANGED).

Primjer 23.1 prikazuje elementarnu implementaciju modela čiji su elementi objekti tipa `java.lang.Integer` a model predstavlja kontinuirani raspon cijelih brojeva koji je zadan prvim i posljednjim elementom.

Primjer 23.1. Implementacija jednostavnog nepromijenjivog modela.

```
 1 package book.java.swing.models;
 2
 3 import javax.swing.ListModel;
 4 import javax.swing.event.ListDataListener;
 5
 6 public class IntRangeListModel implements ListModel<Integer> {
 7
 8  private int firstNumber;
 9  private int lastNumber;
10
11  public IntRangeListModel(int firstNumber, int lastNumber) {
12   super();
13   if(lastNumber < firstNumber) {
14    throw new IllegalArgumentException(
15     "Granice nisu ispravne."
16    );
17   }
18   this.firstNumber = firstNumber;
19   this.lastNumber = lastNumber;
20  }
21
22  @Override
23  public int getSize() {
24   return lastNumber-firstNumber+1;
25  }
26
27  @Override
28  public Integer getElementAt(int index) {
29   return firstNumber+index;
30  }
31
32  @Override
33  public void addListDataListener(ListDataListener l) {
34  }
35
36  @Override
37  public void removeListDataListener(ListDataListener l) {
38  }
39
40 }
41
```

Uočite da ovaj model, iako (potencijalno) predstavlja mnoštvo elemenata, iste ni u kojem trenutku ne čuva u memoriji. Jedino što čuva su informacije potrebne za rekonstrukciju elemenata

na zahtjev te za određivanje njihovog broja: `firstNumber` predstavlja prvi broj niza a `lastNumber` posljednji.

Metoda `getSize()` pri pozivu računa i vraća koliko je ukupno elemenata (brojeva) u nizu koji model predstavlja. Metoda `getElementAt(index)` na temelju predane pozicije i prvog broja niza računa i vraća koji se broj nalazi na traženoj poziciji.

Metode za prijavu i odjavu promatrača namjerno smo za sada ostavili praznima. Proučite ovu implementaciju jer ćemo je nastaviti nadograđivati. Primjer 23.2 prikazuje glavni program koji koristi komponentu `JList` za prikaz elemenata ovog modela.

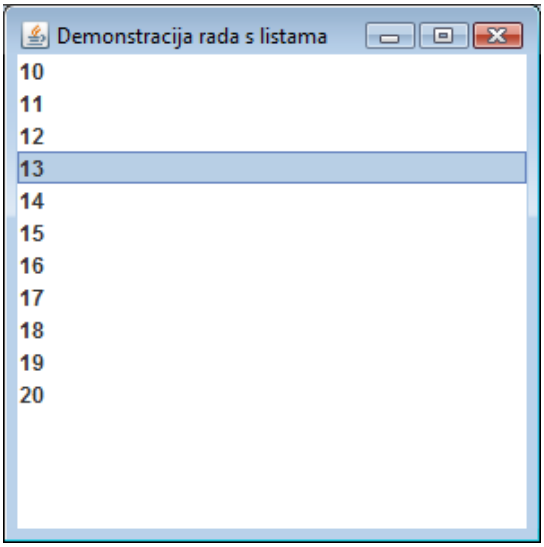
Primjer 23.2. Prikaz liste u prozoru.

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.JList;
 8 import javax.swing.SwingUtilities;
 9 import javax.swing.WindowConstants;
10
11 public class ListDemo1 extends JFrame {
12
13  private static final long serialVersionUID = 1L;
14
15  public ListDemo1() {
16   setTitle("Demonstracija rada s listama");
17   setSize(300, 300);
18   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
19   initGUI();
20  }
21
22  private void initGUI() {
23   Container cp = getContentPane();
24   cp.setLayout(new BorderLayout());
25
26   IntRangeListModel model = new IntRangeListModel(10, 20);
27   JList<Integer> lista = new JList<>(model);
28
29   cp.add(lista, BorderLayout.CENTER);
30  }
31
32  public static void main(String[] args) {
33   SwingUtilities.invokeLater(()->{
34    new ListDemo1().setVisible(true);
35   });
36  }
37 }
38
```

Najinteresantniji su retci 26 i 27. U retku 26 stvaramo primjerak modela koji predstavlja cijele brojeve od 10 do 20 (njih 11 ukupno). U retku 27 stvaramo primjerak grafičke komponente (`JList`) i kroz konstruktor joj predajemo referencu na model.

Pokretanjem programa dobit ćemo prozor kakav je prikazan na slici 23.2.

Slika 23.2. JList i pripadni model podataka: jednostavan primjer

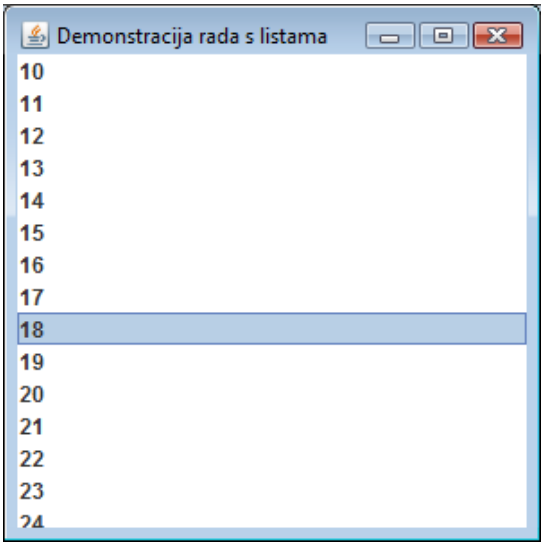


Pokrenite program i isprobajte ga. Kliknite mišem na pojedine elemente i provjerite kako radi selektiranje elemenata. Napravimo li malu izmjenu pa redak 26 promijenimo u:

```
26 IntRangeListModel model = new IntRangeListModel(10, 50);
```

i pokrenemo, dobit ćemo prikaz koji je vidljiv na slici 23.3.

Slika 23.3. JList i pripadni model podataka: jednostavan primjer (2)



Uočite što se dogodilo: prozor više nije dovoljno velik da bi prikazao listu pa se vidi samo gornji dio komponente. Kliknete li na neki od elemenata i zatim kursorskim tipkama krenete pomicati selekciju prema dolje, vidjet ćete da će selekcija nestati kad izađe iz prozora ali će komponenta i dalje korektno pratiti što je selektirano. Da bismo to popravili, sve komponente koje mogu postati "prevelike" s obzirom na raspoloživi prostor uobičajeno smještamo u novu komponentu `JScrollPane` a nju onda dodajemo u prozor (ili neku drugu roditeljsku komponentu). `JScrollPane` će pratiti veličinu komponente koju smo dodali u njega i po potrebi će ponuditi trake za namiještanje pogleda (engl. *scrollbars*).

Razmislite malo o ovom rješenju: koji je oblikovni obrazac ovdje bio primijenjen? Problem koji smo imali je da puno različitih konkretnih komponenata izvedenih iz razreda `JComponent` može biti preveliko za prikaz na raspoloživom prostoru (ovdje smo to vidjeli na primjeru liste, ali istu pojavu možemo primijetiti i sa slikama, komponentama za uređivanje teksta, komponen-

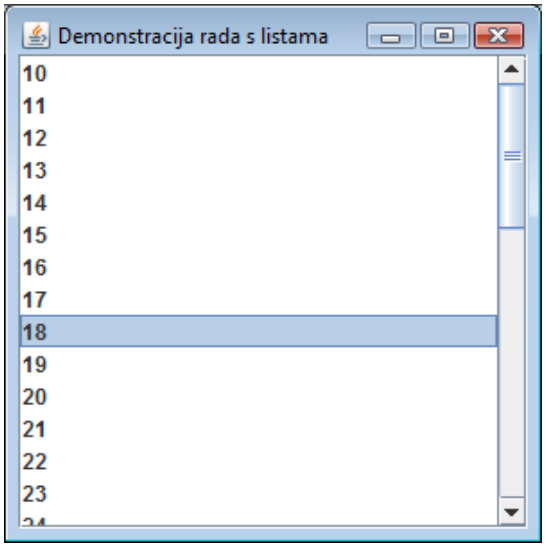
tama za prikaz tablica, komponentama za prikaz stabala, itd.). U svaku od tih komponenata bi trebalo dodavati funkcionalnost vodoravnog i okomitog skrolanja što bi predstavljalo masovno dupliciranje koda. Umjesto toga, u skup standardnih komponenata dodana je nova komponenta koja je također izvedena iz `JComponent`, koja sadrži referencu na neku drugu komponentu tipa `JComponent` ("rekurzivna kompozicija"? jeste li došli do rješenja?), koja na svojoj površini prikazuje tu komponentu i dodaje vodoravne i okomite trake za skrolanje i koju možemo poslati klijentu umjesto originalne komponente. Ta komponenta na sadržanu komponentu gleda kroz osnovni razred `JComponent` pa joj se može postaviti bilo koja konkretna komponenta izvedena (izravno ili neizravno) iz tog razreda. Radi se, dakako, o oblikovnom obrascu *Dekorator*. Možete li identificirati njegove sudionike u ovom konkretnom primjeru?

Spomenuta modifikacija koda koja će koristiti dekorator `JScrollPane` prikazana je u nastavku:

```
29 cp.add(new JScrollPane(lista), BorderLayout.CENTER);
```

Pokrenemo li sada program, dobit ćemo prozor prikazan na slici 23.4.

Slika 23.4. JList i pripadni model podataka: jednostavan primjer (3)



Razmotrimo još jedan primjer.

Primjer 23.3 prikazuje elementarnu implementaciju modela čiji su elementi objekti tipa `java.time.Instant` (služe za predstavljanje trenutka u vremenu; slično kao stariji razred `java.util.Date`). Za razliku od modela prikazanog u primjeru 23.1 koji je na zahtjev stvarao tražene elemente i vraćao ih klijentu, ovaj model svoje podatke čuva u memoriji. U tu svrhu morali smo odabrati prikladnu podatkovnu strukturu (neki od mogućih kandidata su polje, neka od javinih implementacija liste, neka od mapa gdje bi ključevi bile pozicije a vrijednosti elementi na tim pozicijama). Kako je kolekciju tipa `java.util.List` u ovom kontekstu najjednostavnije koristiti te kako je ista prirodno usklađena s modelom podataka kakav klijentima nudi `ListModel`, nju smo odabrali za konkretnu pohranu podataka (točnije, implementaciju `ArrayList` jer nudi dohvat proizvoljnog elementa te dohvat informacije o veličini u O(1)).

Primjer 23.3. Implementacija modela liste koja čuva podatke u memoriji.

```
1 package book.java.swing.models;
2
3 import java.time.Instant;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 import javax.swing.ListModel;
```

```
 8 import javax.swing.event.ListDataListener;
 9
10 public class TimeListModel implements ListModel<Instant> {
11
12  private List<Instant> data = new ArrayList<>();
13
14  public TimeListModel(List<Instant> data) {
15   if(data!=null) {
16    this.data.addAll(data);
17   }
18  }
19
20  @Override
21  public int getSize() {
22   return data.size();
23  }
24
25  @Override
26  public Instant getElementAt(int index) {
27   return data.get(index);
28  }
29
30  public void add(Instant instant) {
31   data.add(instant);
32  }
33
34  public void remove(int index) {
35   data.remove(index);
36  }
37
38  @Override
39  public void addListDataListener(ListDataListener l) {
40  }
41
42  @Override
43  public void removeListDataListener(ListDataListener l) {
44  }
45
46 }
```

S obzirom na taj odabir, metodu `size` implementirali smo tako da vrati veličinu same liste a metodu `getElementAt` tako da od liste zatražimo element i vratimo ga pozivatelju. Kako nam je namjera klijentu omogućiti i modificiranje podataka modela, dodali smo javne metode `add` i `remove` koje smo implementirali delegiranjem podatkovnoj strukturi koja pohranjuje elemente.

Primjer 23.4 prikazuje korisničko sučelje u kojem koristimo ovaj model.

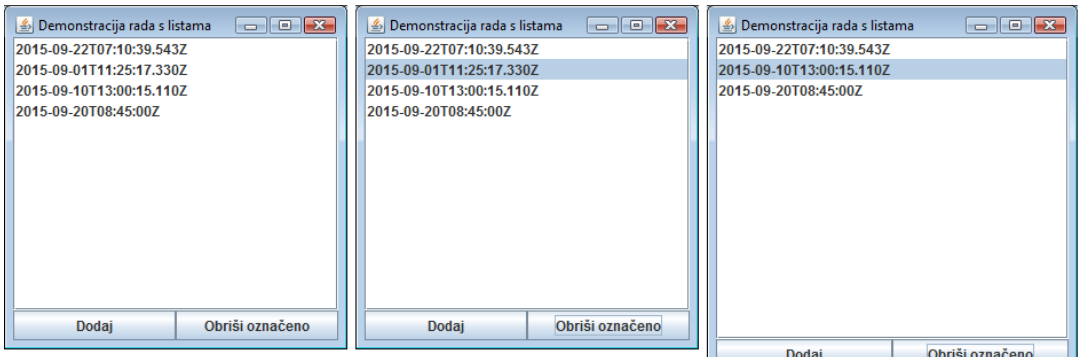
Primjer 23.4. Demonstracijski program uz predstavljeni model liste.

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5 import java.awt.GridLayout;
 6 import java.time.Instant;
 7 import java.util.Arrays;
 8
 9 import javax.swing.JButton;
10 import javax.swing.JFrame;
```

```
11 import javax.swing.JList;
12 import javax.swing.JPanel;
13 import javax.swing.JScrollPane;
14 import javax.swing.SwingUtilities;
15 import javax.swing.WindowConstants;
16
17 public class ListDemo2 extends JFrame {
18
19  private static final long serialVersionUID = 1L;
20
21  public ListDemo2() {
22   setTitle("Demonstracija rada s listama");
23   setSize(300, 300);
24   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
25   initGUI();
26  }
27
28  private void initGUI() {
29   Container cp = getContentPane();
30   cp.setLayout(new BorderLayout());
31
32   TimeListModel model = new TimeListModel(Arrays.asList(
33    Instant.now(),
34    Instant.parse("2015-09-01T11:25:17.33Z"),
35    Instant.parse("2015-09-10T13:00:15.110Z"),
36    Instant.parse("2015-09-20T08:45:00.00Z")
37   ));
38   JList<Instant> lista = new JList<>(model);
39
40   JButton dodaj = new JButton("Dodaj");
41   dodaj.addActionListener(e -> {
42    model.add(Instant.now());
43   });
44
45   JButton obrisi = new JButton("Obriši označeno");
46   obrisi.addActionListener(e -> {
47    int indeks = lista.getSelectedIndex();
48    if(indeks!=-1) {
49     model.remove(indeks);
50    }
51   });
52
53   JPanel bottomPanel = new JPanel(new GridLayout(1, 0));
54   bottomPanel.add(dodaj);
55   bottomPanel.add(obrisi);
56
57   cp.add(new JScrollPane(lista), BorderLayout.CENTER);
58   cp.add(bottomPanel, BorderLayout.PAGE_END);
59  }
60
61  public static void main(String[] args) {
62   SwingUtilities.invokeLater(()->{
63    new ListDemo2().setVisible(true);
64   });
65  }
66 }
67
```

Slika 23.5.(a) prikazuje izgled prozora koji se dobije pokretanjem ovog programa. Uočite kako smo dodali dva gumba: jedan koji u model dodaje novi element koji predstavlja trenutak u vremenu u kojem je pritisnut taj gumb te drugi koji iz modela briše element koji je u grafičkoj komponenti selektiran (ako takav postoji).

Slika 23.5. JList i pripadni model podataka: jednostavan primjer (4)



Slika 23.5.(b) prikazuje što se događa nakon što u grafičkoj komponenti odaberemo jedan element i pritisnemo gumb "Obriši označeno". Isprobajte to! Što se dogodi? Ništa! Barem ne u vizualnom smislu: prikaz liste se ne mijenja tako dugo dok, recimo, ne pokušate malo promijeniti veličinu prozora (slika 23.5.(c)). U tom trenutku odjednom će se promijeniti prikaz liste - nestat će obrisani element a oznaka selekcije će se premjestiti na pogrešan element (pogrešan u smislu da taj element korisnik niti u jednom trenutku nije svojevoljno selektirao). Slično ponašanje dobili bismo i uporabom gumba za dodavanje novih elemenata.

Možete li pogoditi što se dogodilo? Objašnjenje je vrlo jednostavno: grafička komponenta nije dobila informaciju da je sadržaj modela promijenjen pa nije niti tražila od grafičkog podsustava da pokrene ponovno iscrtavanje komponenata. U trenutku kada smo promijenili veličinu prozora, grafički podsustav je automatski pokrenuo novo iscrtavanje svih komponenta čime se slika liste uskladila sa sadržajem modela.

Da bismo riješili ovaj problem, došlo je vrijeme da u modelu implementiramo onaj dio metoda koje su u model stigle iz oblikovnog obrasca Promatrač. Implementacija je prikazana u primjeru 23.5.

Primjer 23.5. Implementacija modela liste koja čuva podatke u memoriji (2).

```
 1 package book.java.swing.models;
 2
 3 import java.time.Instant;
 4 import java.util.ArrayList;
 5 import java.util.List;
 6
 7 import javax.swing.ListModel;
 8 import javax.swing.event.ListDataEvent;
 9 import javax.swing.event.ListDataListener;
10
11 public class TimeListModel implements ListModel<Instant> {
12
13  private List<Instant> data = new ArrayList<>();
14  private List<ListDataListener> listeners = new ArrayList<>();
15
16  public TimeListModel(List<Instant> data) {
17   if(data!=null) {
```

```
18     this.data.addAll(data);
19    }
20   }
21
22   @Override
23   public int getSize() {
24    return data.size();
25   }
26
27   @Override
28   public Instant getElementAt(int index) {
29    return data.get(index);
30   }
31
32   public void add(Instant instant) {
33    int position = data.size();
34    data.add(instant);
35
36    ListDataEvent event = new ListDataEvent(
37     this, ListDataEvent.INTERVAL_ADDED, position, position
38    );
39    for(ListDataListener l : listeners) {
40     l.intervalAdded(event);
41    }
42   }
43
44   public void remove(int index) {
45    data.remove(index);
46
47    ListDataEvent event = new ListDataEvent(
48     this, ListDataEvent.INTERVAL_REMOVED, index, index
49    );
50    for(ListDataListener l : listeners) {
51     l.intervalRemoved(event);
52    }
53   }
54
55   @Override
56   public void addListDataListener(ListDataListener l) {
57    listeners = new ArrayList<>(listeners);
58    listeners.add(l);
59   }
60
61   @Override
62   public void removeListDataListener(ListDataListener l) {
63    listeners = new ArrayList<>(listeners);
64    listeners.remove(l);
65   }
66  }
```

U retku 14 dodali smo još jednu privatnu listu u koju ćemo pohranjivati reference na registrirane promatrače. Prijava (registracija) i odjava (deregistracija) promatrača implementirana je u metodama `addListDataListener` (retci 55-59) i `removeListDataListener` (retci 61-66). Pogledate li malo detaljnije tu implementaciju, mogla bi Vam se učiti čudnom: umjesto da izravno modificiramo listu promatrača, najprije stvaramo kopiju te liste pa nju modificiramo (i postavljamo kao aktualnu listu promatrača). Ovaj kod mora biti napisan na takav (ili sličan način) iz sljedećeg razloga: pretpostavite da u modelu imate kod koji iterira po listi promatrača i jednog po jednog ih obaviještava o nekoj promjeni; pretpostavite također da u trenutku kada jednog

od promatrača obavijestite o nastaloj promjeni, da on nad modelom pozove metodu kojom se odjavljuje (više ga ne zanimaju obavijesti) i da se zbog toga izravno promijeni trenutna lista prijavljenih promatrača. Što će biti posljedica? U trenutku kada se izvođenje vrati u model i nad iteratorom pozove metoda `next()` (ili `hasNext()`) iterator će baciti `ConcurrentModificationException`. Da bismo to izbjegli, imamo dvije (uobičajene) mogućnosti: ili ćemo prije no što krenemo slati obavijesti napraviti kopiju liste pa po njoj iterirati, ili ćemo pri promjenama napraviti kopiju pa nju ažurirati. Uzmete li sada u obzir da se slanje obavijesti obavlja puno češće no prijava i odjava, jasno je da ćemo u prijavi i odjavi implementirati kopiranje. Opisani pristup poznat je kao *Copy-On-Write*.

Jednom kada smo modelu omogućili da prati prijave i odjave, moramo proći i kroz sve metode koje omogućavaju klijentima da rade izmjene nad podatcima i u njih moramo dodati kod koji će slati obavijesti o učinjenim izmjenama. U našem slučaju to moramo napraviti u dvije metode: `add` i `remove`. Pogledajmo kako je riješena metoda `add` (retci 31-42). U retku 33 dohvaćamo trenutnu veličinu liste; taj broj ujedno predstavlja i poziciju na koju će biti umetnut novi element. Potom u retku 34 u korištenu podatkovnu strukturu obavljamo umetanje. U retcima 36-38 stvaramo primjerak razreda `ListDataEvent` koji opisuje nastalu promjenu: izvor je sam model (`this`), tip promjene je `ListDataEvent.INTERVAL_ADDED` a interval pozicija na kojima je nastala promjena ima kao početak i kraj istu vrijednost: `position`.

Nakon što smo stvoriti taj objekt, iteriramo po listi prijavljenih promatrača, pozivamo nad njima metodu `intervalAdded` i kao argument šaljemo referencu na stvoreni objekt s informacijama. U slučaju da je promatrač sama grafička komponenta koja prikazuje podatke modela, točne informacije o promjeni presudne su kako bi komponenta znala na korektan način ažurirati svoje stanje (primjerice, ako postoji selekcija, i ako je novi element umetnut ispred selekcije, indekse selekcije treba povećati jer su selektirani elementi u modelu potonuli prema većim pozicijama; ako je novi element umetnut iza selekcije, tada se indeksi koji označavaju početak i kraj selekcije ne mijenjaju).

Pokrenite sada program i uvjerite se da sve radi korektno. Dopunite metode `addListDataListener` i `removeListDataListener` pomoćnim ispisom i uvjerite se da će se "netko" (u našem slučaju grafička komponenta `JList`) registrirati za dobivanje informacija o promjenama.

Idemo sada napraviti i modifikaciju demonstracijskog programa (vidi primjer 23.6).

Primjer 23.6. Demonstracijski program uz predstavljeni model liste (2).

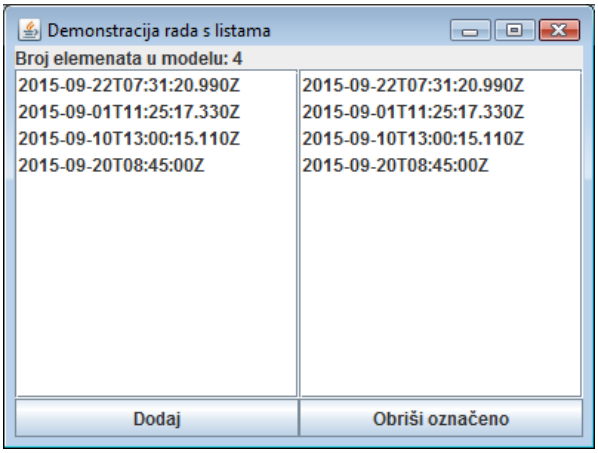
```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5 import java.awt.GridLayout;
 6 import java.time.Instant;
 7 import java.util.Arrays;
 8
 9 import javax.swing.JButton;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JList;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.ListModel;
16 import javax.swing.SwingUtilities;
17 import javax.swing.WindowConstants;
18 import javax.swing.event.ListDataEvent;
19 import javax.swing.event.ListDataListener;
20
21 public class ListDemo2 extends JFrame {
```

```
22
23  private static final long serialVersionUID = 1L;
24
25  public ListDemo2() {
26   setTitle("Demonstracija rada s listama");
27   setSize(400, 300);
28   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
29   initGUI();
30  }
31
32  private void initGUI() {
33   Container cp = getContentPane();
34   cp.setLayout(new BorderLayout());
35
36   TimeListModel model = new TimeListModel(Arrays.asList(
37    Instant.now(),
38    Instant.parse("2015-09-01T11:25:17.33Z"),
39    Instant.parse("2015-09-10T13:00:15.110Z"),
40    Instant.parse("2015-09-20T08:45:00.00Z")
41   ));
42   JList<Instant> lista = new JList<>(model);
43   JList<Instant> lista2 = new JList<>(model);
44
45   JButton dodaj = new JButton("Dodaj");
46   dodaj.addActionListener(e -> {
47    model.add(Instant.now());
48   });
49
50   JButton obrisi = new JButton("Obriši označeno");
51   obrisi.addActionListener(e -> {
52    int indeks = lista.getSelectedIndex();
53    if(indeks!=-1) {
54     model.remove(indeks);
55    }
56   });
57
58   JPanel bottomPanel = new JPanel(new GridLayout(1, 0));
59   bottomPanel.add(dodaj);
60   bottomPanel.add(obrisi);
61
62   JPanel centralPanel = new JPanel(new GridLayout(1, 0));
63   centralPanel.add(new JScrollPane(lista));
64   centralPanel.add(new JScrollPane(lista2));
65
66   StatusnaLabela<Instant> status = new StatusnaLabela<>(model);
67
68   cp.add(centralPanel, BorderLayout.CENTER);
69   cp.add(bottomPanel, BorderLayout.PAGE_END);
70   cp.add(status, BorderLayout.PAGE_START);
71  }
72
73  private static class StatusnaLabela<E> extends JLabel {
74   private static final long serialVersionUID = 1L;
75   private final ListDataListener l = new ListDataListener() {
76    @Override
77    public void intervalRemoved(ListDataEvent e) {
78     updateLabelText();
79    }
```

```
80
81     @Override
82     public void intervalAdded(ListDataEvent e) {
83      updateLabelText();
84     }
85
86     @Override
87     public void contentsChanged(ListDataEvent e) {
88      updateLabelText();
89     }
90    };
91    private final ListModel<E> model;
92
93    public StatusnaLabela(ListModel<E> model) {
94     super();
95     this.model = model;
96     model.addListDataListener(l);
97     updateLabelText();
98    }
99
100   private void updateLabelText() {
101    setText("Broj elemenata u modelu: "+model.getSize());
102   }
103
104  }
105
106  public static void main(String[] args) {
107   SwingUtilities.invokeLater(()->{
108    new ListDemo2().setVisible(true);
109   });
110  }
111 }
112
```

Umjesto jedne grafičke komponente stvorili smo dvije (retci 42 i 43) i obje smo povezali s *istim* modelom podataka. Komponente smo u prozoru postavili jednu do druge, kako je prikazano na slici 23.6. Gumb za brisanje sada prati selekciju lijeve komponente.

Slika 23.6. JList i pripadni model podataka: jednostavan primjer (5)



Pokrenite program. Ako ste u model dodali ispis prijava i odjava, sada biste trebali uočiti da se nad našim modelom registriralo više promatrača: primjerice, obje grafičke komponente. Selekcije u svakoj od grafički komponenata rade nezavisno (nisu dio modela podataka koji kompo-

nente sada dijele). Probajte dodati nekoliko elemenata i uočite da se oba pogleda automatski ažuriraju. Isprobajte brisanje i također se uvjerite da sve radi.

Pogledate li pažljivije program, uočit ćete na vrhu i statusnu traku koja u svakom trenutku ispisuje koliko je elemenata prisutno u modelu. Implementacija te statusne trake dana je u retcima 73-104. Napisali smo novi razred `StatusnaLabela<E>` koji smo izveli iz razreda `JLabel` (dakle radimo novu grafičku komponentu). Ovu komponentu oblikovat ćemo kao još jednog od promatrača modela liste. Počev od retka 75 definirali smo privatnu konstantnu člansku varijablu koja čuva referencu na primjerak promatrača modela liste i koju inicijaliziramo na primjerak anonimnog razreda koji u svim metodama poziva metodu našu metodu `updateLabelText` (definirana u retcima 100-102). Također imamo i privatnu člansku varijablu koja čuva referencu na model liste.

U konstruktoru (retci 93-98) primamo referencu na model te je pamtimo, nad modelom obavljamo registriranje našeg privatnog pomoćnog objekta preko kojeg promatramo sam model te pozivamo metodu `updateLabelText` kako bismo inicijalizirali sadržaj koji prikazujemo. S obzirom da je konstruktor nad modelom obavio registraciju promatrača, svaki puta kada se u modelu dogodi bilo kakva promjena, model će to dojaviti prijavljenom objektu a on će pozvati metodu `updateLabelText`. Ona pak iz modela dohvaća novi trenutni broj elemenata, stvara poruku koju treba prikazati i postavlja je pozivom metode `setText` labele iz koje smo izveli ovu komponentu. Ta će pak metoda uz pohranu predanog teksta automatski pozvati i `repaint()` čime će doći do automatskog osvježavanja slike komponente.

Da bi sve ovo proradilo, još smo samo u metodi `initGUI` trebali stvoriti primjerak ove komponente i dodati ga u korisničko sučelje koje prikazujemo.

U ovom primjeru na kraju smo završili s tri promatrača istog modela: dvije komponente tipa `JList` i jedna komponenta tipa `StatusnaLabela` koju smo sami napisali.



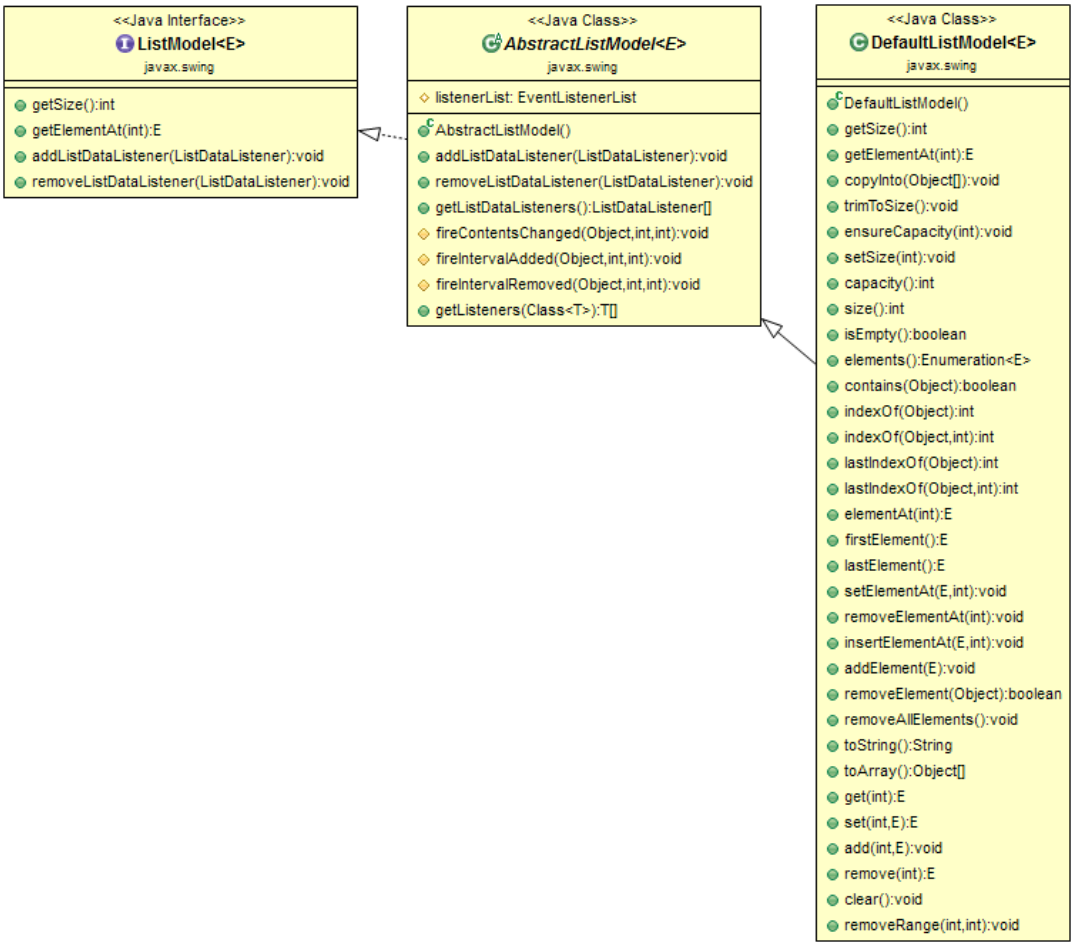
Pomoćni razredi

Sada kada razumijemo kako funkcionira model liste i kako se povezuje s pogledom, treba napomenuti da se implementacije modela najčešće ne pišu direktno od sučelja. Apstraktni razred `AbstractListModel` predstavlja implementaciju sučelja `ListModel` i već donosi implementiranu funkcionalnost za pohranu promatrača, njihovu prijavu i objavu te pomoćne metode kojima se može zatražiti slanje obavijesti prijavljenim promatračima. Pri razvoju vlastitih implementacija modela stoga se preporuča nasljeđivanje ovog razreda čime je dovoljno još samo napisati implementacije metoda `getSize` te `getElementAt`.

Dostupan je i razred `DefaultListModel` koji predstavlja gotovu implementaciju modela koji podatke čuva u memoriji u prikladnoj podatkovnoj strukturi i nudi javne metode za dodavanje/brisanje i druge operacije nad podatcima (uz dojavljivanje potrebnih informacija svim prijavljenim promatračima); razred je izveden iz razreda `AbstractListModel`. Kada je takva funkcionalnost dovoljna, preporuka je koristiti ovaj razred a ne pisati svoju kopiju istoga.

Dijagram razreda koji prikazuje osnovno sučelje i ove dvije implementacija prikazan je na slici 23.7.

Slika 23.7. Model liste i pomoćne implementacije



Uz apstrahiranje modela podataka, grafička komponenta `JList` koristi i dodatne modele. Primjerice, ažuriranje i praćenje selekcije riješeno je kroz sučelje `ListSelectionModel`. Pristup do trenutnog modela koji koristi grafička komponenta može se dobiti pozivom `.getSelectionModel()` a isti se može promijeniti pozivom `.setSelectionModel(model)`. Grafička komponenta i ovaj model također su spojeni oblikovnim obrascem Promatrač: apstraktni promatrač modeliran je sučeljem `ListSelectionListener`, apstraktni subjekt je `ListSelectionModel`, konkretni subjekt je neka od konkretnih implementacija modela a konkretni promatrač je opet grafička komponenta.

Svaka od implementacija modela selekcija trebala bi podržavati tri načina rada (svojstvo `selectionMode`) koji su definirani konstantama `ListSelectionModel.SINGLE_SELECTION`, `ListSelectionModel.SINGLE_INTERVAL_SELECTION` te `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`. Prvi način podrazumijeva da samo jedan element može biti selektiran, drugi da je moguće selektirati jedan kontinuirani podniz elemenata a treći dozvoljava selekciju proizvoljne kombinacije elemenata.

Prikaz elementa liste

Jeste li se možda zapitali kako to da komponenta `JList` "zna" kako vizualno prikazati svaki od elemenata modela liste? Idemo to istražiti na novom primjeru. Definirat ćemo novu vrstu objekata: razred `Karta`. Svaka karta ima dvije članske varijable: `oznaka` (broj koji jedinstveno određuje kartu) te `ime` (string koji predstavlja naziv karte). Kao model ćemo izravno iskoristiti razred `DefaultListModel`. Čitav kod prikazan je u primjeru 23.7.

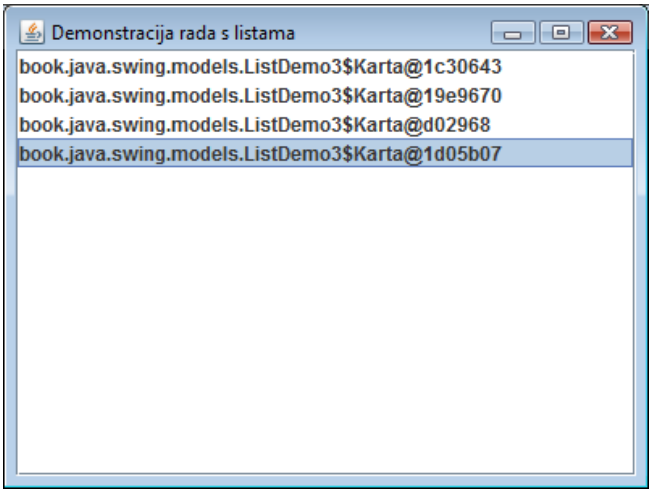
Primjer 23.7. Demonstracijski program uz prikaz elementa liste.

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5
 6 import javax.swing.DefaultListModel;
 7 import javax.swing.JFrame;
 8 import javax.swing.JList;
 9 import javax.swing.JScrollPane;
10 import javax.swing.SwingUtilities;
11 import javax.swing.WindowConstants;
12
13 public class ListDemo3 extends JFrame {
14
15  private static final long serialVersionUID = 1L;
16
17  public class Karta {
18   private int oznaka;
19   private String ime;
20   public Karta(int oznaka, String ime) {
21    super();
22    this.oznaka = oznaka;
23    this.ime = ime;
24   }
25   public int getOznaka() {
26    return oznaka;
27   }
28   public String getIme() {
29    return ime;
30   }
31  }
32
33  public ListDemo3() {
34   setTitle("Demonstracija rada s listama");
35   setSize(400, 300);
36   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
37   initGUI();
38  }
39
40  private void initGUI() {
41   Container cp = getContentPane();
42   cp.setLayout(new BorderLayout());
43
44   DefaultListModel<Karta> model = new DefaultListModel<>();
45   model.addElement(new Karta(1, "As"));
46   model.addElement(new Karta(11, "Dečko"));
47   model.addElement(new Karta(12, "Dama"));
48   model.addElement(new Karta(13, "Kralj"));
49   JList<Karta> lista = new JList<>(model);
50
51   cp.add(new JScrollPane(lista), BorderLayout.CENTER);
52  }
53
54  public static void main(String[] args) {
55   SwingUtilities.invokeLater(()->{
56    new ListDemo3().setVisible(true);
```

```
57    });
58   }
59 }
60
```

Pokretanjem programa dobit će prikaz vidljiv na slici 23.8.

Slika 23.8. Prikaz elemenata liste

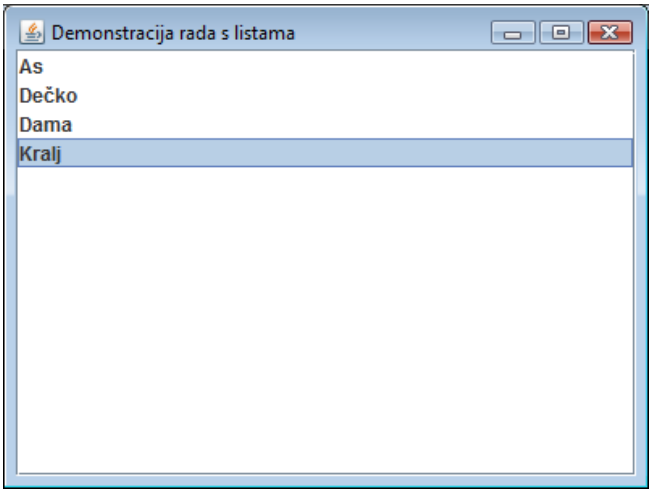


Zašto je prikaz takav, odnosno vidite li što je prikazano? Pretpostavljena implementacija koda koji razred `JList` koristi za prikaz elementa liste nad elementom poziva metodu `toString()` i ono što ta metoda vrati, to se prikazuje. Napravimo stoga malu intervenciju u kod: počev od retka 31 (na kraj razreda `Karta`) ubacite:

```
1 @Override
2 public String toString() {
3  return ime;
4 }
```

čime je za razred `Karta` definirano da u metodi `toString` vraća ime karte. Pokrenite sada program. Dobit ćete prikaz koji odgovara slici 23.9.

Slika 23.9. Prikaz elemenata liste (2)



Zavirimo li malo detaljnije u dokumentaciju razreda `JList`, brzo ćemo pronaći da se zadaća vizualnog prikaza elemenata delegira zasebnoj komponenti modeliranoj parametriziranim sučeljem `ListCellRenderer<E>`. Ovo sučelje navodi samo jednu metodu i prikazano je u nastavku.

```
 1 public interface ListCellRenderer<E> {
 2
 3     Component getListCellRendererComponent(
 4         JList<? extends E> list,
 5         E value,
 6         int index,
 7         boolean isSelected,
 8         boolean cellHasFocus);
 9
10 }
```

Ideja je sljedeća: svaki puta kada `JList` treba prikazati pojedini element (jednu stavku liste), dohvatit će instalirani `ListCellRenderer` i pitati ga (pozivom njegove metode `getListCellRendererComponent`) da joj vrati primjerak grafičke komponente (bilo što što je izvedeno iz razreda `java.awt.Component`) koji je konfiguriran tako da prikazuje zadani element. Uočite da se metodi dostavlja mnoštvo informacija temeljem kojih može podesiti tu komponentu: o kojoj se listi radi, koji se element želi prikazati, na kojoj je lokaciji u listi taj element, je li u prikazu liste taj element selektiran (pa to nekako vizualno treba naznačiti) i na kraju je li element baš onaj koji je trenutno fokusiran u listi. Implementacija grafičke komponente liste potom će izmjeriti vraćeni objekt (da dozna koliko je prikaz elementa "visok") i zatim će njegovu "sliku" preuzeti na odgovarajuće mjesto svoje površine. Ove objekte lista koristi i da bi izračunala primjerice svoje preferirane dimenzije. Treba napomenti da metoda ne mora pri svakom pozivu stvarati novu komponentu već može koristiti jednu, unaprijed stvorenu; naime, `JList` će svaki puta kada zatreba informacije o prikazu bilo kojeg od elemenata nanovo pozvati ovu metodu koja može prekonfigurirati postojeći objekt.

Nadogradnja prethodnog osnovnog primjera koja definira našu vlastitu implementaciju za prikazivanje elemenata liste prikazana je u primjeru 23.8. Pri tome je zbog uštede u prostoru s tri točkice zamijenjen kod koji nije mijenjan - iskorisite ovo samo kao nadopunu prethodnog koda.

Primjer 23.8. Demonstracijski program uz prikaz elementa liste (nadogradnja).

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Component;
 5 import java.awt.Container;
 6
 7 import javax.swing.BorderFactory;
 8 import javax.swing.DefaultListModel;
 9 import javax.swing.JFrame;
10 import javax.swing.JLabel;
11 import javax.swing.JList;
12 import javax.swing.JScrollPane;
13 import javax.swing.ListCellRenderer;
14 import javax.swing.SwingUtilities;
15 import javax.swing.WindowConstants;
16
17 public class ListDemo3 extends JFrame {
18
19  ...
20
21  private void initGUI() {
22   ...
23   JList<Karta> lista = new JList<>(model);
24   lista.setCellRenderer(new PrikazKarte());
25   ...
```

```
26  }
27
28  private static class PrikazKarte
29   implements ListCellRenderer<Karta> {
30
31   private JLabel labela;
32
33   public PrikazKarte() {
34    super();
35    labela = new JLabel();
36    labela.setOpaque(true);
37   }
38
39   @Override
40   public Component getListCellRendererComponent(
41     JList<? extends Karta> list, Karta value, int index,
42     boolean isSelected, boolean cellHasFocus) {
43    labela.setText(value.toString());
44    if(isSelected) {
45     labela.setBackground(list.getSelectionBackground());
46     labela.setForeground(list.getSelectionForeground());
47    } else {
48     labela.setBackground(list.getBackground());
49     labela.setForeground(list.getForeground());
50    }
51    if(cellHasFocus) {
52     labela.setBorder(
53      BorderFactory.createDashedBorder(null, 3f, 2f)
54     );
55    } else {
56     labela.setBorder(null);
57    }
58    return labela;
59   }
60  }
61
62  ...
63 }
64
```

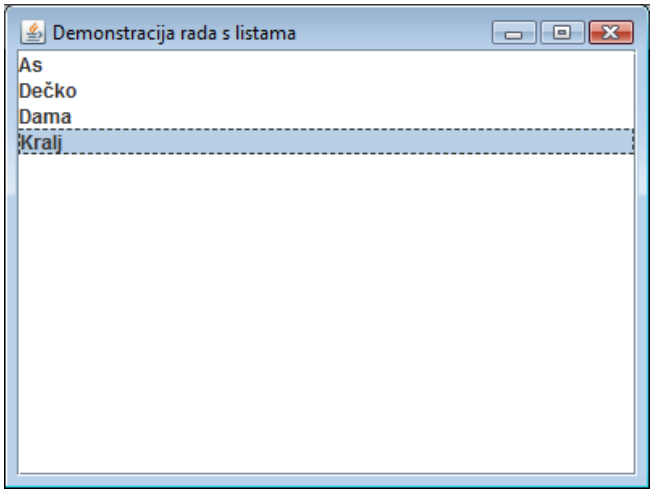
Najveći dio novog koda otpada na definiranje novog razreda `PrikazKarte` koji implementira sučelje `ListCellRenderer` a koji je prikazan u retcima 28-60. Umjesto razvoja vlastite komponente za prikaz jedne karte odlučili smo se za to koristiti postojeću komponentu `JLabel`. Referencu na primjerak labele koju ćemo koristiti definiramo kao člansku varijablu (redak 31) te stvaramo u konstruktoru (redak 35) i podešavamo da je neprozirna (redak 36).

Najveći dio koda odlazi na metodu `getListCellRendererComponent` (retci 39-59). Najprije labelu konfiguriramo tako da kao tekst prikaže ime predane karte (redak 43; sjetite se, u karti smo nadjačali `toString` tako da vraća ime). Potom provjeravamo treba li stavku prikazati kao selektiranu ili ne, i shodno tome nad labelom podešavamo boju pozadine i boju teksta (retci 44-50). Grafička komponenta `JList` ima metode kojima se mogu dohvatiti informacije o tim bojama kako bi prikaz bio u skladu s ostalim prikazima koji su generirani za druge liste. Konačno, provjeravamo i je li element (stavka) za koju generiramo prikaz fokusirana ili nije pa shodno tome labeli postavljamo i crtkani border ili ga uklanjamo (retci 51-57). Jednom kad smo tako podesili sve relevantno, vraćamo referencu na labelu.

Uz definiranje ovog razreda potreba je još samo jedna izmjena: listi trebamo reći da za iscrtavanje elemenata koristi njegov primjerak. To smo riješili u retku 24 pozivom metode `setCellRenderer`.

Pokrenite ovako modificirani program. Dobit ćete prikaz koji odgovara slici 23.10. Poigrajte se malo sa selekcijom i odabirom stavki; uvjerite se da sve radi.

Slika 23.10. Prikaz elemenata liste (3)



Za kraj ćemo napraviti još jednu modifikaciju: kako su elementi modela karte, modificirat ćemo prikaz tako da lijevo od naziva karte prikaže i malu sličicu karte. U tu svrhu pripremljena je ZIP arhiva sa sličicama karata dostupna na adresi http://java.zemris.fer.hr/nastava/opjj/karte.zip. Skinite tu arhivu i raspakirajte je u `src` direktorij projekta (ako imate uobičajenu strukturu) odnosno u `src/resources` ako koristite strukturu koja odgovara projektu alata Maven. U ovom prvom slučaju nakon raspakiravanja trebali biste imati `src/images/card-1.png` te ostale sličice. Ako radite u Eclipse-u, osvježite projekt tako da Eclipse postane svjestan ovoga i da ih automatski iskopira u direktorij `bin`.

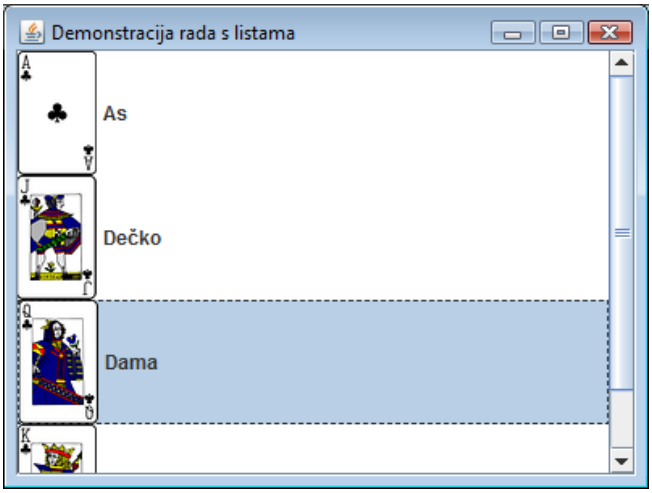
Potom modificirajte metodu `getListCellRendererComponent` našeg razreda tako da pred sam kraj (prije naredbe `return`) umetnete još sljedeći isječak koda.

```
1 try {
2  ImageIcon icon = new ImageIcon(
3   getClass().getResource(
4    "/images/cards-"+value.getOznaka()+".png"
5   )
6  );
7  labela.setIcon(icon);
8 } catch(Exception ex) {
9 }
```

Ovaj isječak će za kartu koja se prikazuje učitati sličicu i predstaviti je kao `ImageIcon` koji je izveden iz razreda `Icon`. Komponenti `JLabel` koju koristimo moguće je uz tekst zadati i ikonu koju treba prikazati i upravo tu funkcionalnost ovdje koristimo.

Pokrenite ovako modificirani program. Dobit ćete prikaz koji odgovara slici 23.11. Poigrajte se malo sa selekcijom i odabirom stavki; uvjerite se da sve radi. Ako slučajno ne vidite ikone, ispišite na zaslon iznimku koju (očito) blok `try - catch` guši: vjerojatno ste na krivo mjesto raspakirali arhivu sa slikama ili niste osvježili projekt.

Slika 23.11. Prikaz elemenata liste (4)



Komponenta JSpinner

Komponenta koja je dosta slična listi ali omogućava prikaz (potencijalno) beskonačno mnogo elemenata je `JSpinner`. Model koji bi bio prikladan za takvu komponentu očito ne može nuditi indeksirani pristup elementima - jer istih može biti beskonačno mnogo pa nemamo prikadan tip podatka za reprezentaciju indeksa (koliko nam bitova treba za beskonačno mnogo pozicija?). Stoga model koji koristi ova komponenta nudi apstrakciju koja je slična iteratoru: moguće je dohvatiti trenutni element, postaviti trenutni element te pitati koji je prethodni odnosno sljedeći element od trenutnog. Ideja je da grafička komponenta korisniku na neki način vizualizira trenutni element te ponudi strelice za odlazak na prethodni odnosno sljedeći element. Ako korisnik naklikne strelicu za sljedeći element, grafička komponenta će pitati model koji je sljedeći element i potom će ga pokušati postaviti u model kao trenutni.

Sučelje koje definira ovakav model podataka je `SpinnerModel` a prikazano je u nastavku.

```
1 public interface SpinnerModel {
2     Object getValue();
3     void setValue(Object value);
4     Object getNextValue();
5     Object getPreviousValue();
6     void addChangeListener(ChangeListener l);
7     void removeChangeListener(ChangeListener l);
8 }
```

Metoda `getValue` vraća element koji je *trenutni* (za model). Metoda `setValue(v)` kao *trenutni* element modela postavlja element predan kao argument, ako je to dozvoljeno i ako je element doista dio elemenata modela. U slučaju da predana vrijednost nije element modela, metoda treba izazvati iznimku `IllegalArgumentException`. Metode `getNextValue()` te `getPreviousValue()` vraćaju element koji je s obzirom na trenutni element njegov prethodnik odnosno njegov sljedbenik, ako takvi postoje; ako ne, metoda treba vratiti `null`.

S obzirom da model barata pojmom *trenutni element* koji je moguće promijeniti pozivom metode `setValue`, definirane su i prikladne metode za prijavu (`addChangeListener`) te odjavu (`removeChangeListener`) promatrača. Apstraktni promatrač opisan je sučeljem `ChangeListener` a objekt koji opisuje promjenu razredom `ChangeEvent`, oba iz paketa javax.swing.event. Sučelje `ChangeListener` prikazano je u nastavku.

```
1 public interface ChangeListener extends EventListener {
2     void stateChanged(ChangeEvent e);
```

```
3 }
```

Definirana je samo jedna metoda koju promatrač mora implementirati: `stateChanged`, i ona će biti pozivana pri svakoj promjeni trenutnog elementa. Razred `ChangeEvent` definira samo jedno svojstvo: `source`.



Bilješka

I komponenta `JSpinner` i model `SpinnerModel` nude sučelje za prijavu i odjavu promatrača tipa `ChangeListener`. Prijavite li promatrača nad komponentom `JSpinner`, svojstvo `source` objekta tipa `ChangeEvent` koji se šalje metodi `stateChanged` bit će postavljeno tako da čuva referencu upravo na primjerak komponente `JSpinner` nad kojim je obavljena registracija; u tom slučaju do modela se može doći tako da se referenca ukalupi u `JSpinner` i potom pozove metoda `getModel` koju komponenta nudi. Ako je promatrač izravno prijavljen nad modelom, tada će svojstvo `source` biti postavljeno tako da čuva referencu na taj model.

Pogledajmo primjer uporabe komponente `JSpinner` i modela `SpinnerModel`. Primjer koji ćemo razraditi stvara model koji omogućava prolaz kroz unaprijed pripremljeni niz različitih boja. Boje su pri tome modelirane postojećim razredom `Color`. Implementaciju jednog takvog modela prikazuje primjer 23.9.

Primjer 23.9. Jedna implementacija modela `SpinnerModel`

```
 1 package book.java.swing.models;
 2
 3 import java.awt.Color;
 4 import java.util.ArrayList;
 5 import java.util.List;
 6
 7 import javax.swing.AbstractSpinnerModel;
 8
 9 public class ColorSpinnerModel extends AbstractSpinnerModel {
10
11  private static final long serialVersionUID = 1L;
12
13  private List<Color> colors = new ArrayList<>();
14  private int currentIndex;
15
16  public ColorSpinnerModel(List<Color> colors) {
17   if(colors==null) {
18    throw new IllegalArgumentException(
19     "Lista boja ne smije biti null."
20    );
21   }
22   if(colors.isEmpty()) {
23    throw new IllegalArgumentException(
24     "Lista boja mora imati barem jedan element."
25    );
26   }
27   this.colors.addAll(colors);
28   this.currentIndex = 0;
29  }
30
31  @Override
32  public Object getValue() {
33   return colors.get(currentIndex);
```

```
34   }
35
36   @Override
37   public void setValue(Object value) {
38    int position = colors.indexOf(value);
39    if(position==-1) {
40     throw new IllegalArgumentException("Unsupported color.");
41    }
42    if(position==currentIndex) return;
43    currentIndex = position;
44    fireStateChanged();
45   }
46
47   @Override
48   public Object getNextValue() {
49    if(currentIndex+1>=colors.size()) return null;
50    return colors.get(currentIndex+1);
51   }
52
53   @Override
54   public Object getPreviousValue() {
55    if(currentIndex<1) return null;
56    return colors.get(currentIndex-1);
57   }
58 }
59
```

Da ne bismo sami morali implementirati pohranu, prijavu, odjavu i obaviještavanje promatrača, model smo ostvarili razredom nasljeđuje razred `AbstractSpinnerModel` a koji implementira sučelje modela `SpinnerModel` te spomenute metode.

Za pohranu niza boja koristimo pomoćnu privatnu listu (redak 13) koju inijaliziramo kroz konstruktor. Privatna članska varijabla `currentIndex` čuva redni broj boje (iz predane liste) koja predstavlja trenutnu boju modela. Metoda `getValue` iz te liste dohvati i vraća boju na poziciji određenoj članskom varijablom `currentIndex`. Metoda `setValue` samo je malo kompliciranija: ona prima boju koja bi trebala postati trenutna, i potom u listi traži poziciju na kojoj se nalazi ta boja. Ako tražena boja ne postoji u toj listi, tada ona ne može postati trenutna i metoda izaziva iznimku (retci 39-41). Ako je boja pronađena, članska varijabla `currentIndex` se ažurira na tu poziciju i zatim se obaviještavaju svi promatrači da je došlo do promjene (pozivom metode `fireStateChanged()` koju je ponudio razreda `AbstractSpinnerModel`). Metode `getNextValue` i `getPreviousValue` iz liste dohvaćaju i vraćaju traženu boju (ili `null` ako takva ne postoji) i ne ažuriraju trenutnu boju.

Jednostavan program koji koristi ovaj model prikazan je u primjeru 23.10.

Primjer 23.10. Demonstracijski program za `JSpinner`

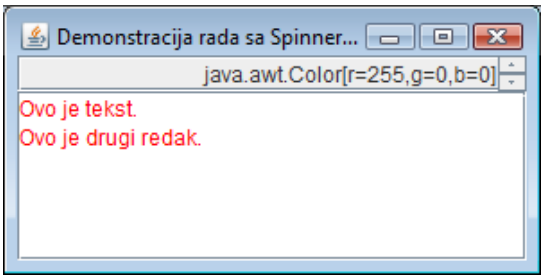
```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Color;
 5 import java.awt.Container;
 6 import java.util.Arrays;
 7
 8 import javax.swing.JFrame;
 9 import javax.swing.JScrollPane;
10 import javax.swing.JSpinner;
11 import javax.swing.JTextArea;
```

```
12 import javax.swing.SpinnerModel;
13 import javax.swing.SwingUtilities;
14 import javax.swing.WindowConstants;
15
16 public class SpinnerDemo1 extends JFrame {
17
18  private static final long serialVersionUID = 1L;
19
20  public SpinnerDemo1() {
21   setTitle("Demonstracija rada sa Spinnerom");
22   setSize(300, 150);
23   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
24   initGUI();
25  }
26
27  private void initGUI() {
28   Container cp = getContentPane();
29   cp.setLayout(new BorderLayout());
30
31   SpinnerModel model = new ColorSpinnerModel(Arrays.asList(
32    Color.RED, Color.GREEN, Color.BLUE,
33    Color.YELLOW, Color.MAGENTA, Color.CYAN
34   ));
35   JSpinner spinner = new JSpinner(model);
36
37   JTextArea textArea = new JTextArea(
38    "Ovo je tekst.\nOvo je drugi redak."
39   );
40   textArea.setForeground((Color)spinner.getValue());
41
42   spinner.addChangeListener(e -> {
43    textArea.setForeground((Color)spinner.getValue());
44   });
45
46   cp.add(spinner, BorderLayout.PAGE_START);
47   cp.add(new JScrollPane(textArea), BorderLayout.CENTER);
48  }
49
50  public static void main(String[] args) {
51   SwingUtilities.invokeLater(()->{
52    new SpinnerDemo1().setVisible(true);
53   });
54  }
55 }
56
```

U ovom primjeru, model inicijaliziramo sa šest boja (retci 31-34). Stvaramo i primjerak uređivača teksta (retci 37-39). Uređivač teksta smještamo u središte komponente a komponentu `JSpinner` koja prikazuje elemente modela na vrh (retci 46-47). Dodatno, nad modelom registriramo promatrača koji prati sve promjene trenutno odabrane boje i svaki puta kada se boja promijeni, istu postavi kao boju kojom uređivač teksta prikazuje tekst (retci 42-44).

Pokretanjem programa dobiva se prikaz kao na slici 23.12. Pokrenite ga i isprobajte.

Slika 23.12. Komponenta JSpinner



JSpinner je na prikazanoj slici komponenta koja je odmah ispod naslovne trake. Sastoji se od glavnog dijela (koji na slici zauzima najveći dio te komponente) te od dvije strelice koje su prikazane uz desni rub. Uređivač teksta (JTextArea) je središnja komponenta koja na slici prikazuje "Ovo je tekst...".

Za vizualizaciju trenutnog elementa JSpinner koristi pomoćnu komponentu koju čuva u svojstvu editor. Sama komponenta zna odabrati prikladne prikaze za tri unaprijed definirana modela koji su dio osnovne biblioteke (SpinnerDateModel, SpinnerListModel te SpinnerNumberModel). Za sve ostale modele koristi komponentu koja element prikazuje tako da nad njim pozove metodu .toString() i to što vrati ta metoda prikaže. To je lijepo vidljivo na slici 23.12.

Primjer 23.11 predstavlja nadogradnju primjera 23.10 u kojoj smo napisali vlastitu komponentu koja se koristi za vizualizaciju odabrane boje.

Primjer 23.11. Demonstracijski program za JSpinner (2)

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Color;
 5 import java.awt.Container;
 6 import java.util.Arrays;
 7
 8 import javax.swing.JFrame;
 9 import javax.swing.JLabel;
10 import javax.swing.JScrollPane;
11 import javax.swing.JSpinner;
12 import javax.swing.JTextArea;
13 import javax.swing.SpinnerModel;
14 import javax.swing.SwingUtilities;
15 import javax.swing.WindowConstants;
16
17 public class SpinnerDemo2 extends JFrame {
18
19  private static final long serialVersionUID = 1L;
20
21  public SpinnerDemo2() {
22   setTitle("Demonstracija rada sa Spinnerom");
23   setSize(300, 150);
24   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
25   initGUI();
26  }
27
28  private void initGUI() {
29   Container cp = getContentPane();
30   cp.setLayout(new BorderLayout());
```

```
31
32   SpinnerModel model = new ColorSpinnerModel(Arrays.asList(
33    Color.RED, Color.GREEN, Color.BLUE,
34    Color.YELLOW, Color.MAGENTA, Color.CYAN
35   ));
36   JSpinner spinner = new JSpinner(model);
37   spinner.setEditor(new ColorDisplay(model));
38
39   JTextArea textArea = new JTextArea(
40    "Ovo je tekst.\nOvo je drugi redak."
41   );
42   textArea.setForeground((Color)spinner.getValue());
43
44   spinner.addChangeListener(e -> {
45    textArea.setForeground((Color)spinner.getValue());
46   });
47
48   cp.add(spinner, BorderLayout.PAGE_START);
49   cp.add(new JScrollPane(textArea), BorderLayout.CENTER);
50  }
51
52  private class ColorDisplay extends JLabel {
53   private static final long serialVersionUID = 1L;
54   private SpinnerModel model;
55   public ColorDisplay(SpinnerModel model) {
56    this.model = model;
57    model.addChangeListener(e->{
58     updateDisplayColor();
59    });
60    updateDisplayColor();
61    setForeground(Color.BLACK);
62    setOpaque(true);
63   }
64   private void updateDisplayColor() {
65    Color newColor = (Color)model.getValue();
66    setBackground(newColor);
67    setText(newColor.toString());
68   }
69  }
70
71  public static void main(String[] args) {
72   SwingUtilities.invokeLater(()->{
73    new SpinnerDemo2().setVisible(true);
74   });
75  }
76 }
77
```

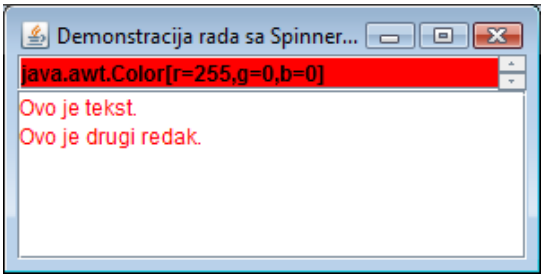
Nova komponenta prikazana je u retcima 52-69. Umjesto da smo radili potpuno novu komponentu (pa prikaz rješavali u metodi `paintComponent`, odlučili smo se za nasljeđivanje iz postojeće komponente `JLabel` koja već nudi mogućnost prikaza teksta te upravljanje bojom teksta i pozadinskom bojom. Naša komponenta definira privatnu člansku varijablu `model` koja pamti referencu na model i koja se inicijaliza u konstruktoru. U konstruktoru komponentu prijavljujemo i kao promatrača nad modelom (zapravo prijavljujemo pomoćni privatni objekt) koji na svaku dojavu poziva našu pomoćnu metodu `updateDisplayColor()` koja je definirana u retcima 64-68. Konstruktor nadalje podešava da je komponenta neprozirna, postavlja kao boju teksta crnu i na kraju još izravno poziva metodu `updateDisplayColor()`.

Metoda updateDisplayColor() iz modela dohvaća trenutnu boju, mijenja pozadinsku boju na tu dohvaćenu boju te kao tekst koji treba prikazati postavlja ono što toString() vrati nad tom bojom.

U retku 37 dodali smo poziv kojim komponenti JSpinner postavljamo našu komponentu kao vrijednost svojstva editor.

Pokrenite program i isprobajte ga. Uočite kako će se sada promjenom trenutne boje mijenjati i pozadinska boja a ne samo prikazani tekst. Slika 23.13 prikazuje rezultat pokretanja.

Slika 23.13. Komponenta JSpinner (2)



Konačno, komponenta koja prikazuje trenutni element ne mora biti korištena samo za prikaz elementa. Često će ove komponente ponuditi korisniku mogućnost da ručno unese vrijednost koja treba postati trenutna i koja će se pohraniti u model pritiskom na tipku ENTER. Stoga smo u nastavku (primjer 23.12) nadogradili komponentu koju postavljamo kao editor komponenti JSpinner tako da podrži i ručno uređivanje.

Primjer 23.12. Demonstracijski program za JSpinner (3)

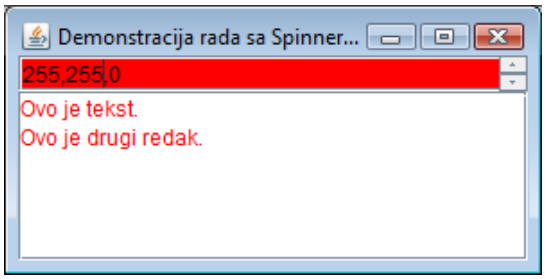
```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Color;
 5 import java.awt.Container;
 6 import java.awt.Toolkit;
 7 import java.awt.event.FocusAdapter;
 8 import java.awt.event.FocusEvent;
 9 import java.awt.event.KeyAdapter;
10 import java.awt.event.KeyEvent;
11 import java.util.Arrays;
12
13 import javax.swing.JFrame;
14 import javax.swing.JScrollPane;
15 import javax.swing.JSpinner;
16 import javax.swing.JTextArea;
17 import javax.swing.JTextField;
18 import javax.swing.SpinnerModel;
19 import javax.swing.SwingUtilities;
20 import javax.swing.WindowConstants;
21
22 public class SpinnerDemo3 extends JFrame {
23
24  private static final long serialVersionUID = 1L;
25
26  public SpinnerDemo3() {
27   setTitle("Demonstracija rada sa Spinnerom");
28   setSize(300, 150);
29   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
30   initGUI();
```

```
31  }
32
33  private void initGUI() {
34   Container cp = getContentPane();
35   cp.setLayout(new BorderLayout());
36
37   SpinnerModel model = new ColorSpinnerModel(Arrays.asList(
38    Color.RED, Color.GREEN, Color.BLUE,
39    Color.YELLOW, Color.MAGENTA, Color.CYAN
40   ));
41   JSpinner spinner = new JSpinner(model);
42   spinner.setEditor(new ColorDisplayEdit(model));
43
44   JTextArea textArea = new JTextArea(
45    "Ovo je tekst.\nOvo je drugi redak."
46   );
47   textArea.setForeground((Color)spinner.getValue());
48
49   spinner.addChangeListener(e -> {
50    textArea.setForeground((Color)spinner.getValue());
51   });
52
53   cp.add(spinner, BorderLayout.PAGE_START);
54   cp.add(new JScrollPane(textArea), BorderLayout.CENTER);
55  }
56
57  private class ColorDisplayEdit extends JTextField {
58   private static final long serialVersionUID = 1L;
59   private SpinnerModel model;
60
61   public ColorDisplayEdit(SpinnerModel model) {
62    this.model = model;
63    model.addChangeListener(e->{
64     updateDisplayColor();
65    });
66    updateDisplayColor();
67    setForeground(Color.BLACK);
68    setOpaque(true);
69    addKeyListener(new KeyAdapter() {
70     @Override
71     public void keyPressed(KeyEvent e) {
72      if(e.getKeyCode()==KeyEvent.VK_ENTER) {
73       e.consume();
74       commitEdit(getText());
75      }
76     }
77    });
78    addFocusListener(new FocusAdapter() {
79     @Override
80     public void focusLost(FocusEvent e) {
81      updateDisplayColor();
82     }
83    });
84   }
85
86   protected void commitEdit(String text) {
87    int[] components = null;
88    try {
```

```
 89      String[] elems = text.split(",");
 90      if(elems.length==3) {
 91       int[] values = new int[3];
 92       for(int i = 0; i < 3; i++) {
 93        values[i] = Integer.parseInt(elems[i].trim());
 94        if(values[i]<0 || values[i]>255) {
 95         throw new Exception();
 96        }
 97       }
 98       components = values;
 99      }
100     } catch(Exception ex) {
101     }
102     if(components==null) {
103      Toolkit.getDefaultToolkit().beep();
104      return;
105     }
106     Color newColor = new Color(
107      components[0],components[1],components[2]
108     );
109     try {
110      model.setValue(newColor);
111     } catch(IllegalArgumentException ex) {
112      Toolkit.getDefaultToolkit().beep();
113      return;
114     }
115    }
116
117    private void updateDisplayColor() {
118     Color newColor = (Color)model.getValue();
119     setBackground(newColor);
120     setText(
121      newColor.getRed() + "," +
122      newColor.getGreen() + "," +
123      newColor.getBlue()
124     );
125    }
126   }
127
128   public static void main(String[] args) {
129    SwingUtilities.invokeLater(()->{
130     new SpinnerDemo3().setVisible(true);
131    });
132   }
133  }
134
```

Pokrenite program i isprobajte ga. Slika 23.14 prikazuje rezultat pokretanja.

Slika 23.14. Komponenta `JSpinner` (3)



Novorazvijena komponenta koja omogućava i prikaz i ručno uređivanje trenutne boje prikazana je u retcima 57-126. Ovu komponentu izveli smo iz komponente `JTextField` koja predstavlja općeniti uređivač jednog retka teksta. Konstruktor, slično kao i kod prethodne komponente, registrira promatrača nad modelom koji će pri svakoj izmjeni u modelu pozvati metodu `updateDisplayColor()`. Komponenta registrira promatrača za pritiske tipki (retci 69-77) koji u slučaju da je korisnik pritisnuo tipku ENTER poziva pomoćnu metodu `commitEdit` i kao argument joj predajte trenutno upisani tekst. Komponenta registrira i promatrača koji prati kada je dobiven odnosno izgubljen fokus pri čemu je od interesa samo gubitak fokusa - u tom slučaju, pretpostavit ćemo da je korisnik odustao od bilo kakvog ručnog namještanja elementa pa se poziva `updateDisplayColor()` kako bi se prikazano stanje ponovno uskladilo sa stanjem modela.

Metoda `updateDisplayColor()` na temelju trenutne boje modela mijenja pozadinsku boju te kao novi tekst koji prikazuje uređivač postavlja tekst formata "r,g,b" gdje su pojedine komponente intenziteti crvene, zelene i plave u rasponu od 0 do 255.

Metoda `commitEdit` (retci 86-115) tekst koji je korisnik unio u komponentu dijeli po znaku zarez i pokušava rekonstruirati trokomponentno polje cijelih brojeva iz intervala 0-255. U slučaju bilo kakve greške u tom dijelu, program izaziva zvučni signal (redak 103) i napušta metodu. U suprotnom, stvara se novi objekt tipa `Color` (retci 106-108) koji ima upravo iščitane rgb-komponente. Potom se ta boja pokušava postaviti kao trenutna boja modela (redak 110). Ako model izazove iznimku jer te boje nema u modelu, proces se prekida uz generiranje zvučnog signala. U suprotnom, ručno zadana boja je pronađena i postavljena kao trenutna.

Rad s tablicama

Tablice služe prikazu podataka koji su organizirani u retke i stupce (pravokutnu matricu). Model podataka tablice definiran je sučeljem `javax.swing.table.TableModel`. Apstraktni razred koji implementira potporu za prijavu, odjavu i obaviještavanje promatrača je `javax.swing.table.AbstractTableModel`. Konačno, cjelovita implementacija modela koja podatke čuva i memoriji i koja se može koristiti gdje je to potrebno dostupna je kroz razred `javax.swing.table.DefaultTableModel`.

Promatrači modela tablice opisani su sučeljem `TableModelListener`. Objekt koji enkapsulira informacije o nastaloj promjeni u modelu modeliran je razredom `TableModelEvent` (oboje u paketu javax.swing.event).

Pogledajmo najprije kako je definirano sučelje `TableModel`.

```
1 public interface TableModel {
2   public int getRowCount();
3   public int getColumnCount();
4   public String getColumnName(int columnIndex);
5   public Class<?> getColumnClass(int columnIndex);
6   public boolean isCellEditable(
7     int rowIndex, int columnIndex);
```

```
 8   public Object getValueAt(int rowIndex, int columnIndex);
 9   public void setValueAt(Object aValue,
10     int rowIndex, int columnIndex);
11   public void addTableModelListener(TableModelListener l);
12   public void removeTableModelListener(TableModelListener l);
13 }
```

Kroz metode `getRowCount` i `getColumnCount` klijent može doznati koliko tablica koju je potrebno prikazati ima redaka i stupaca. Metodom `getColumnName` klijent može doznati koji je naslov pridijeljen kojem stupcu. Metoda `getColumnClass` za prozvani stupac treba vratiti razred kojem pripadaju podatci koje taj stupac prikazuje. Ovu informaciju klijent može iskoristiti kako bi znao na koji način vizualno predstaviti podatke tog stupca (npr. ako su brojevi može ih u prikazu poravnati u desni rub ćelije, ako su Booleove zastavice može ih prikazati na odgovarajući način grafički - kvačica ili izostanak kvačice, ako je String može ga prikazati poravnatog uz lijevi rub ćelije i slično). Uočite da je implicitna pretpostavka da su svi podatci unutar istog stupca jednakog tipa.

Metodom `isCellEditable` model može klijentu dojaviti je li dozvoljeno uređivanje prozvane ćelije ili je njezin sadržaj nepromijenjiv. Metoda `getValueAt` služi za dohvat podatka koji treba prikazati u prozvanoj ćeliji a metoda `setValueAt` za postavljanje novog podatka u prozvanu ćeliju; kako ova metoda obavlja modificiranje modela, ako je promjenu doista i provela (što ne mora ako nešto s podatkom nije u redu), mora o tome obavijestiti sve zainteresirane promatrače.

Konačno, posljednje dvije metode (`addTableModelListener` i `removeTableModelListener`) služe za prijavu i odjavu promatrača.

Sučelje promatrača prikazano je u nastavku.

```
1 public interface TableModelListener
2         extends java.util.EventListener {
3
4     public void tableChanged(TableModelEvent e);
5
6 }
```

ovaj puta sučelje je čak i jednostavnije od onog za promatrača liste. Međutim, razmislite li malo, sučelje promatrača liste je bilo malo čudno. Razlikovalo je tri različite metode za tri različite vrste izmjene, a onda je u objektu s informacijama još jednom dostavljalo tu informaciju kroz zasebnu člansku varijablu što je redundantno i vrlo je učinkovito onemogućilo da imamo na jednom mjestu napisan kod koji radi slanje obavijesti, s obzirom na to da su se različite obavijesti slale kroz različite metode. Ovo sučelje predviđa samo jednu općenitu metodu `tableChanged(e)` gdje će promatraču informacija o vrsti promjene biti poslana kroz objekt koji se šalje kao jedini argument.

Objekt koji opisuje nastalu promjenu modeliran je razredom `TableModelEvent`. Taj razred definira 5 članskih varijabli: `source` (referenca na model), `type` (vrsta promjene, može biti jedna od sljedećih konstanti: `TableModelEvent.INSERT`, `TableModelEvent.UPDATE`, `TableModelEvent.DELETE`), `firstRow` (indeks prvog od niza redaka u kojima je nastala promjena), `lastRow` (indeks zadnjeg od niza redaka u kojima je nastala promjena) te `column` (indeks stupca u kojem je nastala promjena; ako je promjena nastala u svim stupcima redaka određenih s `firstRow` i `lastRow`, tada treba postaviti na konstantu `ListModelEvent.ALL_COLUMNS`). Indeksi redaka idu od 0 na više. Ako se želi dojaviti promjena u naslovnom retku (koji sadrži nazive stupaca), kao indeks retka koristi se konstanta `TableModelEvent.HEADER_ROW`. Razred nudi i nekoliko prikladnih konstruktora.

Pogledajmo najprije kao primjer model tablice koja prikazuje zadani raspon cijelih brojeva te za svaki cijeli broj u retku redom još i njegov kvadrat, kub, recipročnu vrijednost te informaciju je li paran. Implementacija je prikazana u primjeru 23.13.

Primjer 23.13. Model tablice s različitim informacijama o cijelim brojevima.

```
 1 package book.java.swing.models;
 2
 3 import javax.swing.event.TableModelListener;
 4 import javax.swing.table.TableModel;
 5
 6 public class NumbersTableModel implements TableModel {
 7
 8  private int firstNumber;
 9  private int lastNumber;
10
11  public NumbersTableModel(int firstNumber, int lastNumber) {
12   if(lastNumber<firstNumber) {
13    throw new IllegalArgumentException(
14     "Pogrešno zadan raspon."
15    );
16   }
17   if(firstNumber<=0 && lastNumber>=0) {
18    throw new IllegalArgumentException(
19     "Raspon ne smije sadržavati broj 0."
20    );
21   }
22   this.firstNumber = firstNumber;
23   this.lastNumber = lastNumber;
24  }
25
26  @Override
27  public int getColumnCount() {
28   return 5;
29  }
30
31  @Override
32  public int getRowCount() {
33   return lastNumber-firstNumber+1;
34  }
35
36  @Override
37  public String getColumnName(int columnIndex) {
38   switch(columnIndex) {
39   case 0: return "Cijeli broj";
40   case 1: return "Kvadrat";
41   case 2: return "Kub";
42   case 3: return "Recipročno";
43   case 4: return "Broj je paran";
44   }
45   return null;
46  }
47
48  @Override
49  public Class<?> getColumnClass(int columnIndex) {
50   switch (columnIndex) {
51   case 0:
52   case 1:
53   case 2:
54    return Integer.class;
55   case 3:
```

```
56     return Double.class;
57    case 4:
58     return Boolean.class;
59    default:
60     return null;
61    }
62   }
63
64   @Override
65   public boolean isCellEditable(int rowIndex, int columnIndex) {
66    // Niti jednu ćeliju nije moguće uređivati
67    return false;
68   }
69
70   @Override
71   public Object getValueAt(int rowIndex, int columnIndex) {
72    int number = firstNumber + rowIndex;
73    switch(columnIndex) {
74    case 0: return Integer.valueOf(number);
75    case 1: return Integer.valueOf(number*number);
76    case 2: return Integer.valueOf(number*number*number);
77    case 3: return Double.valueOf(1.0/number);
78    case 4: return Boolean.valueOf(number%2==0);
79    }
80    return null;
81   }
82
83   @Override
84   public void setValueAt(Object aValue,
85    int rowIndex, int columnIndex) {
86    // Prazna implementacija - podatke nije moguće ažurirati
87   }
88
89   @Override
90   public void addTableModelListener(TableModelListener l) {
91   }
92
93   @Override
94   public void removeTableModelListener(TableModelListener l) {
95   }
96  }
97
```

Pogledajte kako smo implementirali pojedine metode. Uočite kako su usklađene metode `getColumnClass` i `getValueAt`: prva deklarira tip podatka koji se čuva u određenom stupcu a druga računa i vraća podatak upravo tog tipa.

Metoda `isCellEditable` ostvarili smo tako da uvijek vraća `false`: time poručujemo pogledu da ne smije korisniku ponuditi mogućnost uređivanja podataka niti za jedan redak niti za jedan stupac. Shodno tome, implementirali smo i metodu `setValueAt` koja je prazna - ignoriraju se svi pozivi kojima klijent (ili pogled) pokušavaju promijeniti model.

Kako je model nepromijenjiv, metode za prijavu i odjavu promatrača također smo implementirali kao prazne metode.

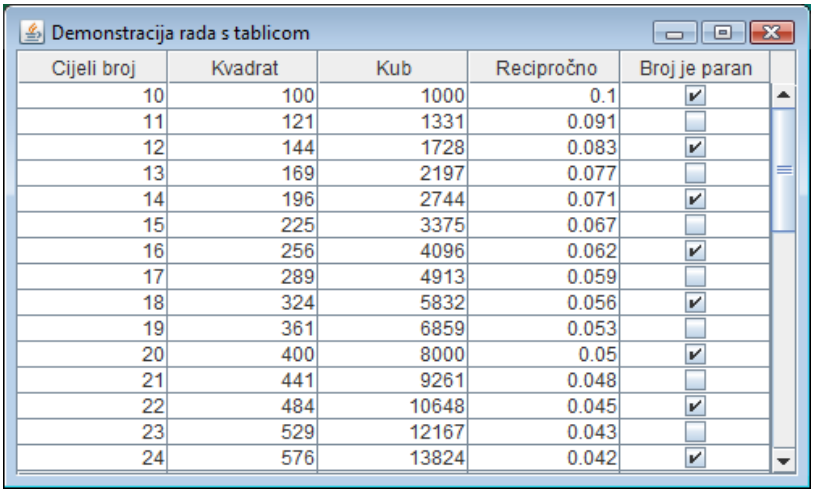
Program koji prikazuje uporabu grafičke komponente `JTable` za prikaz modela tablice prikazan je u primjeru 23.14.

Primjer 23.14. Demonstracijski program uz predstavljeni model tablice.

```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.JScrollPane;
 8 import javax.swing.JTable;
 9 import javax.swing.SwingUtilities;
10 import javax.swing.WindowConstants;
11
12 public class TableDemo1 extends JFrame {
13
14  private static final long serialVersionUID = 1L;
15
16  public TableDemo1() {
17   setTitle("Demonstracija rada s tablicom");
18   setSize(500, 300);
19   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
20   initGUI();
21  }
22
23  private void initGUI() {
24   Container cp = getContentPane();
25   cp.setLayout(new BorderLayout());
26
27   NumbersTableModel model = new NumbersTableModel(10, 50);
28   JTable tablica = new JTable(model);
29
30   cp.add(new JScrollPane(tablica), BorderLayout.CENTER);
31  }
32
33  public static void main(String[] args) {
34   SwingUtilities.invokeLater(()->{
35    new TableDemo1().setVisible(true);
36   });
37  }
38 }
39
```

Pokretanjem ovog programa dobit će se prikaz kao na slici 23.15.

Slika 23.15. Demonstracijski program za jednostavnu tablicu



Cijeli broj	Kvadrat	Kub	Recipročno	Broj je paran
10	100	1000	0.1	☑
11	121	1331	0.091	☐
12	144	1728	0.083	☑
13	169	2197	0.077	☐
14	196	2744	0.071	☑
15	225	3375	0.067	☐
16	256	4096	0.062	☑
17	289	4913	0.059	☐
18	324	5832	0.056	☑
19	361	6859	0.053	☐
20	400	8000	0.05	☑
21	441	9261	0.048	☐
22	484	10648	0.045	☑
23	529	12167	0.043	☐
24	576	13824	0.042	☑

Grafička komponenta `JTable` poprilično je bogata različitim naprednim mogućnostima. Primjerice, želimo li korisniku dopustiti da retke sortira klikom na zaglavlje pojedinog stupca, dovoljno je tablici pridijeliti objekt koji zna raditi sortiranje i koji je modeliran sučeljem `RowSorter`. Dodajte u primjeru 23.14 od retka 29 (u metodu `initGUI`) sljedeća dva retka (prvi stvara objekt koji zna raditi sortiranje, drugi ga povezuje s tablicom):

```
1 RowSorter<TableModel> sorter = new TableRowSorter<>(model);
2 tablica.setRowSorter(sorter);
```

i potom pokrenite program. Isprobajte da klikom na zaglavlje stupca doista možete sortirati retke tablice.

Ako koristite objekt za sortiranje, treba upozoriti da metode koje vraćaju informacije o prikazu (poput metode `getSelectedRow()` vraćaju redni broj retka kako je prikazan u tablici što sada više ne mora odgovarati rednom broju retka u modelu. Naime, objekt za sortiranje nakon pridruživanja grafičkoj komponenti ne radi stvarno sortiranje podataka u modelu (fizički ne premješta podatke u modelu): on samo "interno" izračuna za svaki od redaka modela, na kojem bi se on mjestu nalazio u prikazu kada bi podatci bili sortirani po zadanom stupcu u zadanom smjeru (rastuće/padajuće). Ako pišete kod koji očitava indeks selektiranog retka i s tom informacijom želite otići u model, taj indeks potrebno je pretvoriti u lokaciju na kojoj se taj redak nalazi u modelu. Za tu konverziju objekt za sortiranje nudi metodu `convertRowIndexToModel(index)` koja prima indeks retka kako je prikazan u tablici a vraća indeks retka istog zapisa u modelu. Metoda koja indeks retka modela pretvara u indeks retka prikazane tablice je `convertRowIndexToView(index)`. Da je to tako, lako ćete se uvjeriti ako u metodu `initGUI` dodate još i sljedeći isječak koda:

```
1 tablica.getSelectionModel().addListSelectionListener(
2  new ListSelectionListener() {
3   @Override
4   public void valueChanged(ListSelectionEvent e) {
5    int viewIndex = tablica.getSelectedRow();
6    if(viewIndex == -1) return;
7    int modelIndex = tablica.getRowSorter()
8     .convertRowIndexToModel(viewIndex);
9    System.out.println(viewIndex + " -> " + modelIndex);
10   }
11  }
12 );
```

Time smo nad tablicom registrirali promatrača koji prati promjene u selekciji redaka. Pri svakoj obavijesti, tablicu pita koji je redak selektiran pa ako takav postoji, njegov indeks još pretvori u

položaj u modelu i ta dva podatka ispiše na zaslon. Pokrenite program i pokušajte malo klikati i pratite ispis. Potom sortirajte retke padajuće (tako da u prvom retku imate broj 50) i opet mijenjajte selekciju. Razumijete li što se događa?

Ako je model tablice promijenjiv, potrebno je implementirati i metode za prijavu i odjavu promatrača te prikladan kod za obaviještavanje promatrača. Da to ne bismo morali raditi ručno, preporuča se takve modele izvesti iz apstraktnog razreda `AbstractTableModel` koji već implementira taj dio koda.

Želimo li model tablice koji podatke čuva u memoriji, možemo koristiti gotovu implementaciju `DefaultTableModel` koja je daljnja nadogradnja razreda `AbstractTableModel`. Prikažimo to na jednostavnom primjeru (primjer 23.15).

Primjer 23.15. Demonstracijski program i izvedeni model.

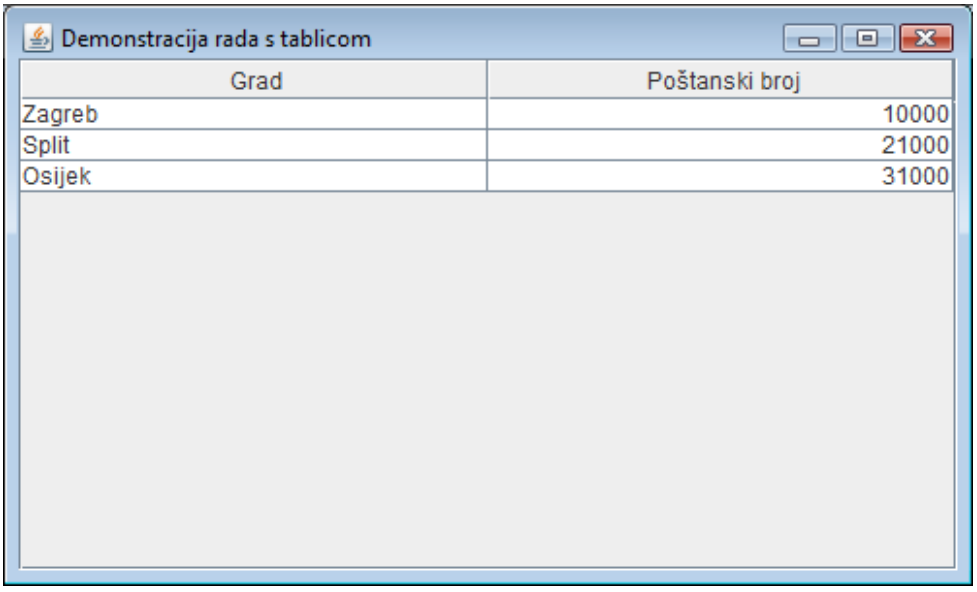
```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.JScrollPane;
 8 import javax.swing.JTable;
 9 import javax.swing.SwingUtilities;
10 import javax.swing.WindowConstants;
11 import javax.swing.table.DefaultTableModel;
12
13 public class TableDemo3 extends JFrame {
14
15  private static final long serialVersionUID = 1L;
16
17  public TableDemo3() {
18   setTitle("Demonstracija rada s tablicom");
19   setSize(500, 300);
20   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
21   initGUI();
22  }
23
24  private void initGUI() {
25   Container cp = getContentPane();
26   cp.setLayout(new BorderLayout());
27
28   Object[][] podatci = {
29    {"Zagreb", 10000},
30    {"Split", 21000},
31    {"Osijek", 31000},
32   };
33   Object[] zaglavlje = {"Grad", "Poštanski broj"};
34   DefaultTableModel model =
35    new DefaultTableModel(podatci, zaglavlje) {
36     private static final long serialVersionUID = 1L;
37     @Override
38     public Class<?> getColumnClass(int columnIndex) {
39      return columnIndex==0 ? String.class : Integer.class;
40     }
41     @Override
42     public boolean isCellEditable(int row, int column) {
43      return column==1;
```

```
44      }
45     };
46
47    JTable tablica = new JTable(model);
48
49    cp.add(new JScrollPane(tablica), BorderLayout.CENTER);
50   }
51
52   public static void main(String[] args) {
53    SwingUtilities.invokeLater(()->{
54     new TableDemo3().setVisible(true);
55    });
56   }
57 }
58
```

Uočite što smo napravili. U retcima 28-32 pripremili smo dvodimenzijsko polje podataka. U retku 33 pripremili smo polje koje čuva nazive stupaca. Potom u retcima 34-45 stvaramo primjerak anonimnog razreda izvedenog iz `DefaultTableModel` kojemu u konstruktoru predajemo podatke i nazive zaglavlja te nadjačavamo metodu koja vraća tip podatka za svaki od stupaca te metodu koja vraća informaciju što se može uređivati (dopustili smo da to bude samo drugi stupac: poštanske brojeve).

Pokretanjem programa dobit ćemo prozor kako je prikazano na slici 23.16.

Slika 23.16. Demonstracijski program

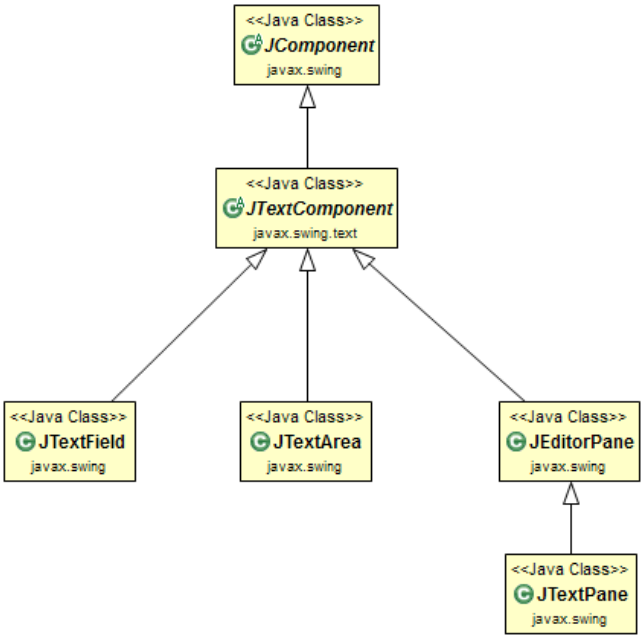


Napravite u tablici dvoklik na 10000 - ćelija će se pretvoriti u uređivač. Pokušajte upisati "x": uređivač Vam neće dopustiti da to pohranite jer je svjestan da je podatak tipa `Integer` te da se "x" ne može pretvoriti u podatak tog tipa. Ako unesete neki drugi broj, isti će uredno biti pohranjen (što će odraditi metoda `setElementAt` koju smo preuzeli iz razreda `DefaultTableModel` koji smo naslijedili.

Komponente za prikaz i uređivanje teksta

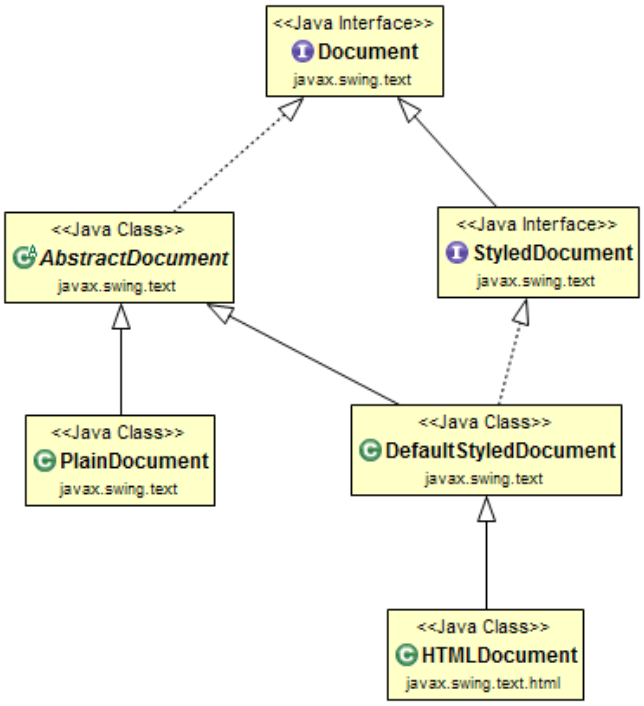
Biblioteka Swing programerima nudi bogat skup komponenata za prikaz i uređivanje teksta a koje su izvedene iz `JComponent`. Slika 23.17 prikazuje odgovarajući dijagram razreda.

Slika 23.17. Komponente za prikaz i uređivanje teksta



Osnovni model koji koriste ove komponente definiran je sučeljem `javax.swing.text.Document`. Izvedena sučelja i implementacije prikazane su dijagramom na slici 23.18.

Slika 23.18. Modeli koje koriste komponente za prikaz i uređivanje teksta



Osnovni model (`Document`) već je sam po sebi relativno složen pa ga nećemo obrađivati detaljno. On sadrži metode za dohvat, modificiranje i brisanje dijelova teksta, metode za prijavu i odjavu promatrača koji žele pratiti modlifikacije teksta te još niz drugih metoda. Promatrači su pri tome modelirani sučeljem `javax.swing.event.DocumentListener` koje definira tri metode: `insertUpdate(e)`, `removeUpdate(e)` te `changeUpdate(e)` dok je objekt koji opisuje nastalu izmjenu i koji se šalje tim metodama definiran razredom `DocumentEvent`.

Razred `PlainDocument` implementacija je takvog modela. Njega kao svoj model koriste komponente `JTextField` i `JTextArea` te `JEditorPane` i `JTextPane` kada je vrsta dokumenta koji im je zadan "text/plain".

Sučelje `StyledDocument` čije je izravna implementacija razred `DefaultStyledDocument` predstavlja proširenje sučelja `Document` i omogućava da se za kontinuirane slijedove znakova (dijelove teksta) definira stil kojim se taj tekst treba prikazati (primjerice, font i boju). Razred `DefaultStyledDocument` kao model koriste `JEditorPane` i `JTextPane` kada je vrsta dokumenta koji im je zadan "text/rtf".

Konačno, iz razreda `DefaultStyledDocument` izveden je razred `HTMLDocument` koji se koristi kao model od komponenata `JEditorPane` i `JTextPane` kada je vrsta dokumenta koji im je zadan "text/html".

Što se tiče samih komponenata, razred `JTextField` nudi uređivanje jednog retka teksta. Razred `JTextArea` nudi mogućenost uređivanja višeretčanih dokumenata ali bez potpore za stilizaciju. Razred `JTextPane` omogućava prikaz i uređivanje različitih vrsta dokumenata (text/plain, text/rtf, text/html) što znači da podržava i uporabu stilova i ugradnju slikovnih elemenata.

Sve ove komponente uz metode za dohvat modela nude i izravno metode `String getText()` i `void setText(String text)` koje dalje komuniciraju s modelom tako da korisnik pri najčešćim zadatcima uporabe ovih komponenata uopće ne mora biti svjestan postojanja modela. Za napredniju uporabu potrebno je detaljnije proučiti dokumentaciju modela te komponenata.

Za potrebe ilustracije napravit ćemo samo jedan primjer: želimo uređivač teksta koji, bilo da korisnik piše velika ili mala slova, u tekst uvijek ulaze velika slova. Rješenje je prikazano u primjeru 23.16.

Primjer 23.16. Demonstracijski program i izvedeni model.

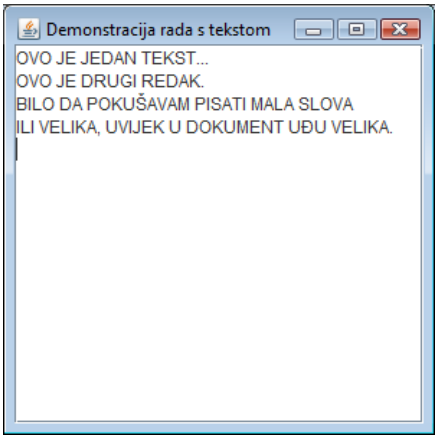
```
 1 package book.java.swing.models;
 2
 3 import java.awt.BorderLayout;
 4 import java.awt.Container;
 5
 6 import javax.swing.JFrame;
 7 import javax.swing.JScrollPane;
 8 import javax.swing.JTextArea;
 9 import javax.swing.SwingUtilities;
10 import javax.swing.WindowConstants;
11 import javax.swing.text.AttributeSet;
12 import javax.swing.text.BadLocationException;
13 import javax.swing.text.Document;
14 import javax.swing.text.PlainDocument;
15
16 public class TextDemo1 extends JFrame {
17
18  private static final long serialVersionUID = 1L;
19
20  public TextDemo1() {
21   setTitle("Demonstracija rada s tekstom");
22   setSize(300, 300);
23   setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
24   initGUI();
25  }
26
27  private void initGUI() {
28   Container cp = getContentPane();
29   cp.setLayout(new BorderLayout());
```

```
30
31   Document model = new PlainDocument() {
32    private static final long serialVersionUID = 1L;
33
34    @Override
35    public void insertString(int offs, String str,
36      AttributeSet a)
37      throws BadLocationException {
38     super.insertString(offs, str.toUpperCase(), a);
39    }
40   };
41   JTextArea textArea = new JTextArea(model);
42
43   cp.add(new JScrollPane(textArea), BorderLayout.CENTER);
44  }
45
46  public static void main(String[] args) {
47   SwingUtilities.invokeLater(()->{
48    new TextDemo1().setVisible(true);
49   });
50  }
51 }
52
```

Uočite što smo napravili: stvorili smo kao model primjerak anonimnog razreda izvedenog iz `PlainDocument` kojem smo nadjačali metodu `insertString` na način da primljeni tekst pretvori u tekst s velikim slovima i potom smo takav tekst proslijedili originalnoj metodi `insertString` koja obavlja fizičko umetanje.

Pokretanjem programa i unosom teksta dobit ćemo prozor kako je prikazano na slici 23.19.

Slika 23.19. Demonstracijski program



Poglavlje 24. Uporaba relacijskih baza podataka iz Jave: SQL

- Direktna uporaba SQL-a (PreparedStatement, ResultSet-ovi i slično)

Poglavlje 25. Uporaba relacijskih baza podataka iz Jave: JPA

- Što je ORM
- Uporaba specifikacije JPA
- Primjer izrade aplikacije koja podatke čuva uporabom specifikacije JPA
- Kao provider koristiti Hibernate

Poglavlje 26. Razvoj web aplikacija

- HTML dokumenti
- DocumentRoot, Web poslužitelj, URL adresa
- Jednostavan primjer: index.html koji sadrži reference na dvije slike (banner.png te images/fruits.png) te voce/detaljno.html koji sadrži opis jabuke i referencu na ../images/apples.png
  - Kad korisnik upiše adresu http://127.0.0.1/index.html, objasniti što će preglednik napraviti
    - uspostavi spoj, zatraži index.html, raskine spoj
    - isparsira dokument i nađe dvije reference na slike
    - uspostavi spoj, zatraži banner.png, raskine spoj
    - uspostavi spoj, zatraži images/fruits.png, raskine spoj
    - time je renderiranje stranice gotovo
  - sada korisnik klikne na link "detalji", a preglednik:
    - uspostavi spoj, zatraži voce/detaljno.html, raskine spoj
    - isparsira dokument i nađe referencu na sliku ../images/apples.png
    - uspostavi spoj, zatraži images/apples.png, raskine spoj
    - time je renderiranje stranice gotovo
- Protokol HTTP
  - protokol tipa zahtjev + odgovor
  - struktura zahtjeva
  - struktura odgovora
  - mime tipovi i značaj za preglednik; daj screenshot ako se za html pregledniku kaže da je "text/plain; charset: utf-8" ili application/octet-stream
  - praćenje korisnika (tj. održavanje sjednice): cookies i privatna mapa podataka o korisnicima
  - metode GET i POST
  - obrada parametara iz URL-a
  - primjer jednostavne autorizacije
    - stranica /prijava?username=pero&password=tajna
    - stranica /povjerljivo koja se prikazuje samo ako je korisnik prijavljen

Poglavlje 27. Web obrasci

- tagovi <form>, <input>, <submit>, <reset>
- izrada login stranice za prijavu na sustav
- ActionForm objekti, ActionForm.fillFromObject(o), ActionForm.validate(), ActionForm.updateObject(o)

Poglavlje 28. Tehnologija Java Servlet i JSP

- dinamičke web aplikacije
  - razvoj: CGI, skriptni jezici, punokrvna programska rješenja
- uporaba skriptnih jezika
  - PHP
  - JSP
- anatomija web aplikacije prema specifikaciji Servleta
- izrada servleta
- mapiranje servleta na URL koji ga pokreće
- tomcat poslužitelj i struktura
- Servleti vs. JSP: prednosti i mane
- Kombiniranje servleta i JSP-a: MVC oblikovni obrazac
- autentifikacija korisnika
- uporaba application listenera
- uporaba Filtera

Poglavlje 29. Višeslojne web-aplikacije

- Što, kako, i zašto
- Presentation layer, Application Layer, DAO Layer
- Upogoniti Filter tako da otvori i zatvori JPA vezu automatski
- Implementirati DAO preko Hibernate-a
- Pokazati kako se konfigurira koja će se implementacija koristiti kao implementacija kojeg sučelja

Poglavlje 30. IOC kontejneri

- konfiguracija kroz tekst ili već nekako drugačije te uporaba FactoryMethod oblikovnog obrasca
- type 1: interface injection
- type 2: setter injection
- type 3: constructor injection
- primjer: Pico/Nano container, Spring

Poglavlje 31. Test Driven Development

- Prema prethodnim predavanjima...

Poglavlje 32. Razvoj aplikacija za Android

- Prema prethodnim predavanjima...

Dodatak A. Instalacija JDK

Da biste na vlastitom računalu mogli razvijati programe pisane u programskom jeziku Java, nužno je da na računalu imate instaliran JDK. U ovom dodatku osvrnut ćemo se na instalaciju ovog produkta.

Operacijski sustav Windows

JDK za operacijski sustav Windows skinite direktno sa stranice proizvođača: http://www.oracle.com/technetwork/java/javase/downloads/index.html. Tamo ćete pronaći izvršni program koji je nakon preuzimanja potrebno pokrenuti kako bi se obavio postupak instalacije. Instalacijska procedura uobičajeno će sve potrebno pospremiti u direktorij `Program Files (x86)`. Primjerice, instalacijom JDK-a verzije 7 podverzije 15 nastat će direktorij `C:\Program Files (x86)\Java\jdk1.7.0_15`. Unutar tog direktorija dalje se nalazi razgranata struktura poddirektorija od kojih su dva najvažnija: `bin` u kojem se nalazi niz programa koji omogućavaju razvoj programa pisanih u Javi (poput Java prevodioca) te direktorij `jre` koji sadrži Javin virtualni stroj i potrebne biblioteke.

Nakon instalacije potrebno je provjeriti sadržaj varijable okruženja `PATH`, pa ako je instalacijski program nije prilagodio, potrebno je obaviti modifikaciju kojom će u nju još biti dodan poddirektorij `bin` te poddirektorij `jre\bin`. Dodatno, potrebno je postaviti i varijablu okruženja `JAVA_HOME` tako da pokazuje na direktorij u koji je instaliran JDK (što je u našem primjeru direktorij `C:\Program Files (x86)\Java\jdk1.7.0_15`.

Ako se to želi napraviti privremeno na razini konzole (tj. programa **Command Prompt**), to je moguće postići tako da se nakon pokretanja programa **Command Prompt** zadaju sljedeće dvije naredbe.

```
SET "JAVA_HOME=c:\Program Files (x86)\java\jdk1.7.0_15"
SET "PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%PATH%"
```

Ovako izvedeno podešavanje vrijedi samo za pokrenuti **Command Prompt** i poništava se njegovim gašenjem. Za trajno podešavanje ove varijable je potrebno podesiti direktno na razini operacijskog sustava kroz upravljačku ploču (engl. *Control Panel*). Važno je napomenuti da se podešavanje na razini operacijskog sustava neće vidjeti kroz prethodno pokrenute programe. Odnosno, ako ćete podešavanje raditi na razini operacijskog sustava (što je svakako preporuka), tada najprije napravite podešavanje a potom pokrenite program koji bi to morao vidjeti (primjerice, **Command Prompt**).

Je li sve podešeno kako treba, možete provjeriti zadavanjem dvaju naredbi. Pokrenite **Command Prompt**. Zadajte najprije naredbu: **java -version**.

```
C:\>java -version
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b05)
Java HotSpot(TM) Client VM (build 22.1-b02, mixed mode, sharing)
```

Ako ste dobili ispis poput navedenog, sve je spremno za pokretanje gotovih Java programa. Ako ste dobili poruku da program **java** ne postoji, varijabla okruženja `PATH` nije dobro podešena pa to popravite.

Sada još treba provjeriti može li se pokretati Java prevodilac. Zadajte naredbu **javac -version**.

```
C:\>javac -version
javac 1.7.0_15
```

Ako ste dobili ispis poput navedenog, sve je spremno i za prevođenje izvornih Java programa u izvršne Java programe. Ako ste dobili poruku da program **javac** ne postoji, varijabla okruženja `PATH` nije dobro podešena pa to popravite.

Operacijski sustav Linux

Za operacijski sustav Linux sve je potrebno moguće dohvatiti direktno sa stranice proizvođača: http://www.oracle.com/technetwork/java/javase/downloads/index.html (bilo kao arhivu formata `rpm` ili kao arhivu formata `tar.gz`. Međutim, danas je možda najjednostavnije instalaciju obaviti ili direktno prilikom instalacije samog operacijskog sustava Linux (odabirom odgovarajućeg paketa) ili pak uporabom odgovarajućih paketnih menadžera bilo iz komandne linije (ovisno o distribuciji koju koristite, to može biti primjerice paketni menadžer **apt** ili **yum**) bilo kroz grafičko korisničko sučelje (program **Synaptic** ili pak neki od specijaliziranih koji dolaze s Vašom distribucijom Linuxa). Prednost ovakve instalacije je što ćete od samog operacijskog sustava redovito dobivati obavijesti o novijim verzijama i zakrpama te ćete čitav postupak nadogradnje moći obaviti automatski.

U svakom slučaju, jednom kada ste napravili instalaciju, obavezno provjerite je li varijabla okruženja `JAVA_HOME` postavljena, pa ako nije, postavite je. Provjeru je najlakše obaviti direktno iz ljuske zadavanjem naredbe **echo $JAVA_HOME**. Ako naredba ne prikaže stazu do mjesta na kojem je instaliran JDK, ručno postavite tu varijablu okruženja. Dodatno, pokrenite naredbe **java -version** te **javac -version**. Obje naredbe bi trebale na zaslon ispisati verziju programa (prva naredba verziju virtualnog stroja a druga naredba verziju Java prevodioca). U slučaju da umjesto toga dobijete poruku da se program ne može pronaći/pokrenuti, prilagodite jarijablu okruženja `PATH`.

Dodatak B. Instalacija alata ant, maven i gradle

Alati **ant**, **maven** i **gradle** služe za automatizaciju pojedinih koraka u razvojnom procesu aplikacije. U nastavku slijede kratke upute za instalaciju svakog od njih.

Radite li na operacijskom sustavu Windows, predlažem da napravite zaseban direktorij u koji ćete pohraniti sve ove programe. Primjerice, neka to bude vršni direktorij `D:\usr`.

Alat Apache Ant

Alat Apache Ant skinite direktno sa stranice proizvođača: http://ant.apache.org/bindownload.cgi. Dokumentacija za rad s ovim alatom dostupna je preko stranice http://ant.apache.org/manual/index.html.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je dostupna kao `apache-ant-1.9.4-bin.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `apache-ant-1.9.4` koji sadrži sve što je potrebno za rad s ovim alatom. Da biste dovršili postupak instalacije, potrebna su još dva koraka. Definirajte varijablu okruženja `ANT_HOME` tako da pokazuje na vršni direktorij instalacije alata **ant**. U ovo slučaju to bi bio direktorij `D:\usr\apache-ant-1.9.4`. Potom u varijablu okruženja `PATH` dodajte putanju do `bin` direktorija koji se nalazi u prethodno definiranom direktoriju; u našem slučaju, u `PATH` bismo dodali direktorij `D:\usr\apache-ant-1.9.4\bin`.

Sada možete provjeriti je li sve instalirano kako spada. Otvorite ljusku (ili na Windowsima **Command Prompt** i zadajte sljedeću naredbu.

```
ant -version
```

Rezultat mora biti ispis trenutno instalirane verzije alata **ant**.

Alat Apache Maven

Alat Apache Maven skinite direktno sa stranice proizvođača: http://maven.apache.org/download.cgi. Dokumentacija za rad s ovim alatom dostupna je preko stranice http://maven.apache.org/users/index.html.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je dostupna kao `apache-maven-3.3.1-bin.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `apache-maven-3.3.1` koji sadrži sve što je potrebno za rad s ovim alatom. Da biste dovršili postupak instalacije, potrebna su još dva koraka. Definirajte varijablu okruženja `M2_HOME` tako da pokazuje na vršni direktorij instalacije alata **maven**. U ovo slučaju to bi bio direktorij `D:\usr\apache-maven-3.3.1`. Potom u varijablu okruženja `PATH` dodajte putanju do `bin` direktorija koji se nalazi u prethodno definiranom direktoriju; u našem slučaju, u `PATH` bismo dodali direktorij `D:\usr\apache-maven-3.3.1\bin`.

Sada možete provjeriti je li sve instalirano kako spada. Otvorite ljusku (ili na Windowsima **Command Prompt** i zadajte sljedeću naredbu.

```
mvn -version
```

Rezultat mora biti ispis trenutno instalirane verzije alata **maven**.

Alat Gradle

Alat Gradle skinite direktno sa stranice proizvođača: http://www.gradle.org/downloads. Dokumentacija za rad s ovim alatom dostupna je preko stranice http://www.gradle.org/documentation.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je dostupna kao `gradle-1.4-all.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `gradle-1.4` koji sadrži sve što je potrebno za rad s ovim alatom. Da biste dovršili postupak instalacije, potrebna su još dva koraka. Definirajte varijablu okruženja `GRADLE_HOME` tako da pokazuje na vršni direktorij instalacije alata **gradle**. U ovo slučaju to bi bio direktorij `D:\usr\gradle-1.4`. Potom u varijablu okruženja `PATH` dodajte putanju do `bin` direktorija koji se nalazi u prethodno definiranom direktoriju; u našem slučaju, u `PATH` bismo dodali direktorij `D:\usr\gradle-1.4\bin`.

Sada možete provjeriti je li sve instalirano kako spada. Otvorite ljusku (ili na Windowsima **Command Prompt** i zadajte sljedeću naredbu.

```
gradle -v
```

Rezultat mora biti ispis trenutno instalirane verzije alata **gradle**.

Dodatak C. Instalacija alata za kontrolu kvalitete i ispravnosti koda

Alati **CheckStyle**, **PMD**, **FindBugs** i **JUnit** služe za provođenje koraka u kojima se obavlja kontrola kvalitete i ispravnosti napisanoga koda. U nastavku slijede kratke upute za instalaciju svakog od njih.

Radite li na operacijskom sustavu Windows, predlažem da napravite zaseban direktorij u koji ćete pohraniti sve ove programe. Primjerice, neka to bude vršni direktorij `D:\usr`. Na operacijskom sustavu Linux možete napraviti isto.

Alat CheckStyle

Alat CheckStyle skinite direktno sa stranice proizvođača: http://checkstyle.sourceforge.net/. Dokumentacija za rad s ovim alatom dostupna je preko iste stranice.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je verzija 6.5 pa u tom slučaju skidate ZIP arhivu `checkstyle-6.5-bin.zip`, te također i ZIP arhivu s izvornim kodovima `checkstyle-6.5-src.zip`.

Nakon što ste skinuli arhivu `checkstyle-6.5-bin.zip`, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `checkstyle-6.5` koji sadrži sve što je potrebno za rad s ovim alatom. Ovaj alat nudi mogućnost direktnog pokretanja na način da se iz komandne linije pokrene Java program kojem se predaju svi potrebni parametri (što se provjerava, nad kojim se datotekama radi provjera i slično). Želite li ga pokretati na taj način, proučite odgovarajuću dokumentaciju (konkretno http://checkstyle.sourceforge.net/cmdline.html). Međutim, alat obično nećemo pokretati na taj način već ćemo ga integrirati s alatom **ant** i pokretati u okviru nekog cilja, što je pojašnjeno u poglavlju o uporabi alata za analizu kvalitete i ispravnosti koda.

U vršnom direktoriju alata CheckStyle sada napravite poddirektorij `contrib`. U taj direktorij skinite i spremite datoteku http://java.zemris.fer.hr/nastava/opjj/checkstyle/checkstyle-frames.xsl. To je datoteka koja opisuje kako će se XML izvještaj koji će generirati ovaj alat pretvoriti u HTML. Alat **ant** već dolazi s takvom datotekom, ali verzija 1.9.4 iz nekog razloga ne prenosi dobro imena datoteka - stoga koristite ovu datoteku s navedene adrese.

Pronađite u arhivi s izvornim kodovima (`checkstyle-6.5-src.zip`) u poddirektoriju `checkstyle-6.5/src/main/resources` datoteku `sun_checks.xml` i nju prekopirajte u vršni direktorij alata CheckStyle u poddirektorij `config`.

Modificirajte tu datoteku kako je opisano u nastavku. S obzirom da su danas veličine zaslona s kojima raspolažemo poprilično narasle u odnosu na stanje prije desetak i više godine, jednu od preporuka, a to je da duljina niti jednog retka ne bi smjela preći 80 znakova, potrebno je ponešto modificirati. U današnje doba složit ćemo se da je i redak koji ima 100 ili čak 120 znakova sasvim prihvatljiv. Stoga u konfiguraciji ovog alata potražite redak u kojem se konfigurira navedena provjera i izmjenite ga tako da se kao prihvatljiva duljina retka prihvaća sve što je do 120 znakova. Konkretno, potražite redak 111 u kojem se nalazi:

```
 <module name="LineLength"/>
```

Taj zapis korigirajte tako da postane:

```
<module name="LineLength">
     <property name="max" value="120"/>
</module>
```

Uz ovaj alat, da bi sve funkcioniralo ispravno, još je potrebno skinuti i XSLT procesor koji služi za pretvorbu XML dokumenata u druge formate. Stoga otiđite na stranicu http://xml.apache.org/xalan-j/ gdje ćete dobiti mogućnost preuzimanja biblioteke Xalan, odnosno njezine inačice napisane u Javi. U trenutku pisanja ovog teksta, zadnja verzija koju biste trebali preuzeti dolazi u ZIP arhivi imena `xalan-j_2_7_1-bin.zip`. S obzirom da nije baš jednostavno iz prve otkriti kako točno do ove arhive, evo i direktne adrese: http://archive.apache.org/dist/xml/xalan-j/.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `xalan-j_2_7_1` koji sadrži sve što je potrebno za rad s ovim procesorom.

Alat PMD

Alat PMD skinite direktno sa stranice proizvođača: http://pmd.sourceforge.net/. Dokumentacija za rad s ovim alatom dostupna je preko iste stranice.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je verzija 5.2.3 pa u tom slučaju skidate ZIP arhivu `pmd-bin-5.2.3.zip`. Skinite i arhivu s izvornim kodovima: `pmd-src-5.2.3.zip`.

Nakon što ste skinuli arhivu `pmd-bin-5.2.3.zip`, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `pmd-bin-5.2.3` koji sadrži sve što je potrebno za rad s ovim alatom. Ovaj alat nudi mogućnost direktnog pokretanja na način da se iz komandne linije pokrene Java program kojem se predaju svi potrebni parametri (što se provjerava, nad kojim se datotekama radi provjera i slično). Želite li ga pokretati na taj način, proučite odgovarajuću dokumentaciju. Međutim, alat obično nećemo pokretati na taj način već ćemo ga integrirati s alatom **ant** i pokretati u okviru nekog cilja, što je pojašnjeno u poglavlju o uporabi alata za analizu kvalitete i ispravnosti koda.

Sada još iz arhive `pmd-src-5.2.3.zip` u vršni direktorij `pmd-bin-5.2.3` koji ste maloprije napravili samo iskopirajte direktoij `etc` i njegov sadržaj, tako da dobijete `pmd-bin-5.2.3\etc`.

Alat FindBugs

Alat FindBugs skinite direktno sa stranice proizvođača: http://findbugs.sourceforge.net/. Dokumentacija za rad s ovim alatom dostupna je preko iste stranice (odnosno na adresi http://findbugs.sourceforge.net/manual/.

Jednom kada ste na stranici za skidanje programa, potražite zadnju verziju alata i skinite odgovarajuću ZIP arhivu. Primjerice, u trenutku pisanja ovog teksta posljednja verzija alata bila je verzija 3.0.1 pa u tom slučaju skidate ZIP arhivu `findbugs-3.0.1.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `findbugs-3.0.1` koji sadrži sve što je potrebno za rad s ovim alatom. Nakon raspakiravanja možete postaviti varijablu okruženja `FINDBUGS_HOME` tako da pokazuje na ovaj direktorij. S obzirom da alat nudi mogućnost pokretanja i preko komandne linije, možete modificirati i varijablu okruženja `PATH` tako da u nju dodate i direktorij `${FINDBUGS_HOME}/bin`. U okviru ove knjige alat nećemo pokretati na taj način već ćemo ga integrirati s alatom **ant** i pokretati kao jedan od ciljeva, što je pojašnjeno u poglavlju o uporabi alata za analizu kvalitete i ispravnosti koda.

Alat JUnit

Alat JUnit skinite direktno sa stranice proizvođača: http://junit.org/. Važno: postoji nekoliko verzija verzija ovog alata i trenutno aktualna je verzija 4 koja nije kompatibilna s prethodnom ver-

zijom. Mi ćemo koristiti tu najnoviju verziju (dakle, JUnit 4). Dokumentacija za rad s ovim alatom dostupna je preko iste stranice. Prilikom skidanja alata trebat ćete preuzeti dvije biblioteke: `junit-4.12.jar` i `hamcrest-core-1.3.jar`. U prethodno stvorenom direktoriju `D:\usr` napravite poddirektorij `junit-4.12` i u taj poddirektorij spremite prethodno skinute JAR arhive.

Alat JaCoCo

Alat JaCoco skinite direktno sa stranice proizvođača: http://www.eclemma.org/jacoco/. U trenutku pisanja ovog dokumenta posljednja verzija alata bila je 0.7.4. Na stranici je dostupna poveznica preko koje je moguće dohvatiti alat u obliku ZIP arhive; u tom slučaju ime će biti `jacoco-0.7.4-201502262128.zip` ili slično. Skinite tu arhivu. U prethodno stvorenom direktoriju `D:\usr` napravite poddirektorij `jacoco-0.7.4` i u taj poddirektorij raspakirajte prethodno skinute arhivu. Direktorij `lib` koji je dio te arhive trebao bi imati punu stazu: `D:\usr\jacoco-0.7.4\lib`. Osim ovoga, nikakva daljnja podešavanja nisu potrebna.

Biblioteka Mockito

Biblioteku Mockito skinite direktno sa stranice proizvođača: http://code.google.com/p/mockito/. U trenutku pisanja ovog dokumenta posljednja verzija alata bila je 1.9.5. Na stranici je dostupna poveznica preko koje je moguće dohvatiti alat u obliku JAR arhive; u tom slučaju ime će biti `mockito-all-1.9.5.jar` ili slično. Skinite tu arhivu. U prethodno stvorenom direktoriju `D:\usr` napravite poddirektorij `mockito-1.9.5` i u taj poddirektorij pohranite skinute arhivu (nemojte je raspakiravati).

Dodatak D. Instalacija poslužitelja servleta

Da biste na vlastitom računalu mogli pokretati web-aplikacije pisane uporabom tehnologija Java Servlet i Java Servlet Pages, potrebno je instalirati prikladan web-poslužitelj. Za razliku od općenitih web-poslužitelja (poput poslužitelja Apache HTTP Server), ovdje je potreban poslužitelj koji radi u skladu s tehnologijama Java Servlet i Java Servlet Pages te koji razumije format u kojem se pakiraju web-aplikacije napisane prema tim normama. Primjeri takvih poslužitelja su Apache Tomcat te Jetty. U nastavku je opisana instalacija poslužitelja Apache Tomcat te Jetty.

Instalacija poslužitelja Apache Tomcat

Otiđite na stranicu proizvođača i skinite posljednju inačicu poslužitelja. U trenutku pisanja ovog teksta to je inačica 7.0.40. Adresa je: http://tomcat.apache.org/download-70.cgi. Odaberite verziju 7.0.40, sekciju "Binary Distributions", sekciju "Core" pa skinite ZIP arhivu.

Ovisno o operacijskom sustavu, Tomcat je moguće instalirati i kao servis (Windowsi / Linux); dapače, Tomcat je moguće na Linuxu instalirati i preko paketnih managera; sve to prepuštam iskusnijim korisnicima na vlastitu odgovornost; primjerice, paketni manageri dijelove Tomcata znaju porazbacati po različitim direktorijima distribucije -- ako napravite takvu instalaciju, utrošite malo vremena i pokušajte pronaći gdje je što. "Kanonska" instalacija Tomcata sve potrebno ima u unificiranoj strukturi direktorija -- dovoljno je otpakirati ZIP arhivu i sve je na jednom mjestu. U okviru ove upute pretpostavit ćemo da ste to napravili na taj način. Tekst napisan u ovoj knjizi također polazi od pretpostavke da je to napravljeno upravo tako.

ZIP arhivu negdje raspakirajte; pretpostavit ćemo da će to biti naš uobičajeni `usr` direktorij (npr. Windowsi: `D:\usr`, Linux: `/opt/usr`); time će nastati `D:\usr\apache-tomcat-7.0.40` odnosno `/opt/usr/apache-tomcat-7.0.40`.

Sada napravite sljedeće.

- Definirajte varijablu okruženja `CATALINA_HOME` tako da pokazuje na taj instalacijski direktorij. Na Windowsima (u konzoli, svaki puta kada pokrenete konzolu) zadajte:

  ```
  set "CATALINA_HOME=D:\usr\apache-tomcat-7.0.40"
  ```

  Na Linuxu zadajte:

  ```
  export CATALINA_HOME=/opt/usr/apache-tomcat-7.0.40
  ```

  Alternativno, spremite to kao permanentnu postavku (Windowsi: Control panel, System, Advanced, Environment, ...; Linux: `/etc/profile`).

- Ako želite Tomcat pokretati iz konzole upisivanjem naredbe (pa ne trebate otvarati Windows explorer i raditi dvoklik; slično i za Linux), podesite varijablu okruženja `PATH`.

  Na operacijskom sustavu Window naredba je:

  ```
  set "PATH=%PATH%;%CATALINA_HOME%\bin"
  ```

  a na operacijskom sustavu Linux:

  ```
  export PATH=$PATH:$CATALINA_HOME/bin
  ```

  Uz ovako definirane varijable okruženja, pokretanje/gašenje Tomcata na Windowsima ostvarit ćete naredbama: `startup.bat` odnosno `shutdown.bat` a na Linuxu naredbama `startup.sh` odnosno `shutdown.sh`.

Poslužitelj možete prekinuti i pritiskom na tipke **Ctrl**+**C**. Direktorij u koji se postavljaju web-aplikacije je `webapps`.

Nakon ovih postavki potrebno je još podesiti dva detalja koja su opisana u nastavku.

Podešavanje kodne stranice i omogućavanje praćenja izvođenja

Početne postavke poslužitelja Apache Tomcat ne traže da se koristi kodna stranica UTF-8. Međutim, danas je to uobičajeno. Stoga ćemo podesiti Tomcat da kao pretpostavljenu kodnu stranicu za čitanje i pisanje datoteka koristi UTF-8. Usput ćemo pokazati i kako odrediti količinu memorije koju poslužitelj ima na raspolaganju te kako omogućiti praćenje rada programa (kada u razvojnom okruženju postavimo točku prekida u kodu i kada želimo dalje naredbu po naredbu pratiti kako se program izvodi na poslužitelju).

Apache Tomcat je poslužitelj napisan u Javi. To znači da ga se konfigurira kao i svaki drugi Java-program, barem što se tiče prethodno spomenutih postavki. Prilikom pokretanja virtualnog stroja potrebno je dodati sljedeće parametre.

- `-Xmx512M`

  Ovim parametrom podešavamo virtualni stroj tako da može koristiti do 512 MB memorije.

- `-Dfile.encoding=UTF-8`

  Ovim parametrom podešavamo virtualni stroj da za čitanje i pisanje tekstovnih datoteka koristi kodnu stranicu UTF-8.

- `-Xdebug -Xrunjdwp:transport=dt_socket,address=5555,server=y,suspend=n`

  Ovim parametrima tražimo od virtualnog stroja da sluša na TCP portu 5555 i da prihvaća zahtjeve udaljenih *debugera*; ovo će nam omogućiti da iz razvojne okoline pratimo rad razvijenih programa.

Na operacijskom sustavu Windows otvorite datoteku `%CATALINA_HOME%\bin\startup.bat` u nekom uređivaču teksta. Otiđite pred sam kraj datoteke. Vidjet ćete dio:

```
call "%EXECUTABLE%" start %CMD_LINE_ARGS%
```

Zamijenite ga (odnosno dodajte `SET` prikazan u nastavku ispred te naredbe).

```
set "JAVA_OPTS=-Xmx512M -Dfile.encoding=UTF-8 -Xdebug
   -Xrunjdwp:transport=dt_socket,address=5555,server=y,suspend=n"
call "%EXECUTABLE%" start %CMD_LINE_ARGS%
```

Zbog ograničenja širine stranice naredba `SET` razlomljena je u dva retka -- u datoteku je treba unijeti u jednom retku.

Na operacijskom sustavu Linux otvorite `$CATALINA_HOME/bin/startup.sh` u nekom uređivaču teksta. Otiđite pred sam kraj datoteke. Vidjet ćete dio:

```
exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

Zamijenite ga (odnosno dodajte definiranje varijable `JAVA_OPTS` prikazano u nastavku ispred te naredbe).

```
JAVA_OPTS="-Xmx512M -Dfile.encoding=UTF-8 -Xdebug
   -Xrunjdwp:transport=dt_socket,address=5555,server=y,suspend=n"
exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

Zbog ograničenja širine stranice deklaracija varijable okruženja `JAVA_OPTS` razlomljena je u dva retka -- u datoteku je treba unijeti u jednom retku.

Dodatno podešavanje kodne stranice

Još na jednom mjestu trebamo podesiti uporabu kodne stranice UTF-8. Pronađite datoteku `server.xml` koja se nalazi u Tomcatovom `config` direktoriju i također je otvorite u nekom uređivaču teksta. Pronađite deklaraciju konektora koji prihvaća zahtjeve na portu 8080 (počinje u retku 70).

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Korigirajte tu deklaraciju tako da odgovara deklaraciji prikazanoj u nastavku (drugim riječima, dodajte: `URIEncoding="UTF-8"`):

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" URIEncoding="UTF-8" />
```

Time je završeno osnovno podešavanje.

Instalacija poslužitelja Jetty

Otiđite na web-stranice proizvođača i skinite ZIP arhivu s distribucijom ovog poslužitelja. Adresa je: http://www.eclipse.org/jetty/. Odaberite *Jetty Downloads* te na toj novoj stranici skinite posljednju stabilnu inačicu. U trenutku pisanja ovog teksta to je bila inačica 9 (tj. Stable 9.0.3.v20130506). Preuzmite arhivu i spremite je negdje na disk.

Sadržaj arhive raspakirajte u naš `usr` direktorij. Uz prethodno navedenu inačicu, time će nastati direktorij `D:\usr\jetty-distribution-9.0.3.v20130506`. U tom direktoriju postojat će poddirektoriji `bin`, `etc`, `webapps` i drugi. U korijenskom direktoriju pronači ćete još i datoteke `start.jar` te `start.ini`.

U uređivaču teksta otvorite datoteku `start.ini` i odmah u prvu sekciju ubacite sljedeće.

```
--exec
-Xmx512M
-Dfile.encoding=UTF-8
-Xdebug
-Xrunjdwp:transport=dt_socket,address=5555,server=y,suspend=n
```

Navedene opcije imaju isto značenje kao i kod poslužitelja Apache Tomcat. Snimite postavke i zatvorite datoteku.

Poslužitelj ćete pokrenuti zadavanjem naredbe:

```
java -jar start
```

iz vršnog direktorija instalacije. Poslužitelj prekidate pritiskom na **Ctrl**+**C**.

Direktorij u koji se postavljaju web-aplikacije je `webapps`. Postavljanje podataka o korisnicima i zaporkama za aplikacije koje koriste autentifikaciju i autorizaciju opisano je na adresi: http://www.eclipse.org/jetty/documentation/current/configuring-security.html#configuring-security-authentication.

Dodatak E. Instalacija upravitelja bazama podataka

Baze podataka danas su postale sastavni dio mnoštva različitih programskih rješenja: tehnologija je sazrijela do te mjere da je uporaba relacijskih baza podataka postala vrlo jednostavna i pristupačna iz svih modernih programskih jezika.

Danas je na tržištu dostupno više različitih vrsta baza podataka: klasične relacijske baze, objektne baze, XML baze te u posljednje vrijeme i takozvane *NoSql* baze. Relacijske baze podataka najduže su s nama. Sustavi za upravljanje relacijskim bazama nude se u komercijalnim izvedbama te kao rješenja otvorenog koda. Nekoliko poznatijih rješenja tog tipa su MySql, MariaDB (GNU GPL zamjena za MySql), Postgres te druge.

U svijetu Jave posebno su zanimljiva rješenja napisana direktno u Javi jer takvi sustavi često nude nekoliko načina rada: mogu funkcionirati kao nezavisni upravitelji baze podataka na koje se klijenti spajaju uporabom mrežnih protokola; s druge pak strane, takve je upravitelje često moguće koristiti i direktno u klijentskoj aplikaciji pri čemu klijent i upravitelj komuniciraju direktno kroz pozive metoda a upravitelj klijentu nudi cjelokupnu uslugu uz uporabu privatne baze podataka. I takvih rješenja postoji više: spomenut ćemo samo H2 Database[1] te Apache Derby[2]. Napomenimo da se upravitelj Apache Derby pokazao toliko stabilnim i kvalitetnim da je njegova inačica 10.8 postala standardni dio distribucije JDK 7 i nalazi se u poddirektoriju `db` instalacijskog direktorija JDK-a. Kako to nije najnovija inačica, u nastavku je dan opis kako postaviti najnoviju inačicu (10.10).

Instalacija upravitelja Apache Derby

Otiđite na stranicu proizvođača i skinite posljednju verziju upravitelja Apache Derby. Na adresi: http://db.apache.org/derby odaberite karticu *Download* pa na stranici koja se otvori birajte posljednje službeno izdanje (*Latest Official Releases*). U trenutku pisanja ove upute to je bila inačica 10.10.1.1 (April 15, 2013 / SVN 1458268). Odabirom inačice doći ćete do stranice na kojoj se nudi više oblika preuzimanja. Odaberite cjelokupnu binarnu distribuciju (`db-derby-10.10.1.1-bin.zip`). Skinite tu arhivu i smjestite je negdje na disk.

Raspakirajte sadržaj arhive u naš uobičajeni `usr` direktorij. Trebali biste dobiti sljedeću strukturu direktorija.

```
D:\
 +- usr
     +- db-derby-10.10.1.1-bin
         +- bin
         |   +- ...
         +- lib
         |   +- ...
         ...
```

Slijedite li dalje instalacijske upute koje se nalaze na stranicama proizvođača, dobro pripazite koji dio pratite: ono što želimo dobiti jest nezavisnog upravitelja koji se ponaša kao mrežni poslužitelj i na kojeg se klijenti spajaju putem mreže. Najjednostavniji način prikazan je u nastavku.

Podešavanje upravitelja baze

Definirajte varijablu okruženja `DERBY_INSTALL` tako da pokazuje na instalacijski direktorij upravitelja. Sljedeće dvije naredbe to će napraviti na Windowsima (prva naredba) odnosno na Linuxu (druga naredba).

[1] http://www.h2database.com/html/main.html
[2] http://db.apache.org/derby

```
D:\> set "DERBY_INSTALL=d:\usr\db-derby-10.10.1.1-bin"

/> export DERBY_INSTALL=/opt/usr/db-derby-10.10.1.1-bin
```

Odaberite direktorij u kojem će upravitelj pohranjivati sve baze podataka koje stvorite. Neka to bude primjerice direktorij `D:\tmp\derby-baze`. Napravite taj direktorij. Potom u njega smjestite tekstovnu datoteku `derby.properties` koja je sljedećeg sadržaja.

```
derby.database.sqlAuthorization=true
derby.connection.requireAuthentication=true
derby.database.defaultConnectionMode=noAccess
derby.database.fullAccessUsers=sa
derby.database.readOnlyAccessUsers=sb
derby.authentication.provider=NATIVE:credentialsDB:LOCAL
derby.authentication.native.passwordLifetimeMillis=157680000000
```

Ova konfiguracijska datoteka nalaže upravitelju baze da koristi mehanizme autentifikacije i autorizacije korisnika. Podešavamo da neautorizirani korisnici nemaju nikakva prava te da je korisnik koji ima sva prava (u određenom smislu administrator) korisnik čije je korisničko ime `sa` (bez brige, još ga nismo napravili). Konačno, podešeno je da se za globalne postavke koristi baza podataka `credentialsDB` (koju također još nismo napravili).

Sada iz konzole pokrenite poslužitelj sljedećom naredbom (prvi redak: Windowsi, drugi redak: Linux). Obje naredbe su razlomljene u dra retka kako bi stale na stranicu -- prilikom unosa sve je argumente potrebno zadati slijedno.

```
java -Dderby.system.home=D:\tmp\derby-baze
     -jar %DERBY_INSTALL%\lib\derbyrun.jar server start

java -Dderby.system.home=/opt/usr/derby-baze
     -jar $DERBY_INSTALL/lib/derbyrun.jar server start
```

Uočite, kao parametar `derby.system.home` predaje se direktorij u koji će biti smještene sve baze, i još važnije, direktorij u kojem se nalazi konfiguracijska datoteka za poslužitelj. Ako je sve u redu, dobit ćete poruku:

```
Thu May 30 15:03:43 CEST 2013 : Security manager installed using
  the Basic server security policy.
Thu May 30 15:03:45 CEST 2013 : Apache Derby Network Server -
  10.10.1.1 - (1458268) started and ready to accept connections on
  port 1527
```

i program će ostati pokrenut. Sve eventualne poruke program će nastaviti ispisivati u tu konzolu.

Otvorite novu konzolu. Ako varijablu okruženja `DERBY_INSTALL` niste podesili globalno, onda je opet definirajte i u ovoj konzoli. Najprije pokušajte dobiti informacije o pokrenutom poslužitelju. Naredba je:

```
java -Dderby.system.home=D:\tmp\derby-baze
  -jar %DERBY_INSTALL%\lib\derbyrun.jar server sysinfo
```

Pokretanjem naredbe, ako je poslužitelj pokrenut i spreman, dobit ćete ispis s nizom informacija razvrstanih u nekoliko sekcija, poput općih informacija, informacija o podešenim postavkama, informacija o verziji Jave koja se koristi od poslužitelja, informacija o JAR arhivama koje koristi Apach Derby te informacije o podržanim lokalizacijskim područjima.

S obzirom da smo prethodno podesili uporabu autentifikacije i autorizacije i to korištenjem baze koja još ne postoji, dok se to ne razriješi, upravitelj Apache Derby dozvolit će nam samo jednu akciju: moramo stvoriti tu bazu i barem jednog korisnika; odabrat ćemo da taj korisnik ima korisničko ime `sa`. Kako bismo proveli te korake, potreban nam je program koji se zna spojiti na upravitelja i poslati mu odgovarajuće naredbe. Apache Derby dolazi s implementacijom takve konzole koju je najjednostavnije pokrenuti direktno iz konzole na sljedeći način:

```
java -Dfile.encoding=IBM852
  -jar %DERBY_INSTALL%\lib\derbyrun.jar ij
```

Postavka `-Dfile.encoding=IBM852` rješava problem prikaza dijakritičkih znakova na konzoli operacijskog sustava Windows i samo je u tom slučaju treba pisati (sa **chcp** provjerite koju kodnu stranicu koristi konzola i po potrebi prilagodite). Ako radite na Linuxu, dovoljno je napisati:

```
java -jar $DERBY_INSTALL/lib/derbyrun.jar ij
```

Pokretanjem programa **ij** dobit ćemo novi prompt:

```
ij>
```

Sada prođite kroz sljedeće korake.

- *Izrada baze* `credentialsDB`

  Zadajte sljedeću naredbu. Naredba je zbog ograničene širine stranice razlomljena u dva retka ali je treba zadati kao jednu naredbu.

  ```
  connect 'jdbc:derby://localhost:1527/credentialsDB;user=sa;
  password=sapwd22;create=true';
  ```

  Ovom naredbom spajamo se na upravitelj baze podataka koji je na lokalnom računalu i koji sluša na portu 1527 te tražimo pristup bazi podataka čije je ime `credentialsDB`. Pri tome šaljemo korisničke podatke (ime: `sa`, zaporuka: `sapwd22` -- slobodno odaberite neku drugu ako želite). I konačno, uporabom atributa `create=true` tražimo stvaranje baze.

  Kako ova baza još ne postoji i kako se njezino ime podudara s imenom baze za koju smo upravitelj podesili da je koristi za autentifikaciju, upravitelj će je stvoriti te će u nju zapisati podatke o korisniku (predano korisničko ime i zaporku). Od ovog trenutka na dalje, korisnik `sa` je jedini korisnik koji postoji u sustavu i koji se može spojiti na bilo koju bazu (ili stvarati nove baze).

  Zadajte naredbu:

  ```
  ij> disconnect;
  ```

  kako biste se otpojili od baze `credentialsDB`.

- *Stvaranje nove baze podataka*

  Prilikom stvaranja nove baze podataka potrebno je napraviti nekoliko koraka. Najprije je potrebno stvoriti bazu podataka -- to radimo kao korisnik `sa`. Potom, nakon što je baza stvorena i nakon što smo se na nju spojili, definirat ćemo novog korisnika za tu bazu podataka i dodijelit mu sve dozvole za rad s tom bazom. Na taj način ovlasti novog korisnika bit će ograničene samo na bazu koju smo za njega stvorili.

  Pretpostavimo da želimo napraviti novu bazu `baza1DB` te da želimo pune ovlasti za rad s tom bazom dati lokalnom korisniku čije će korisničko ime biti `perica` a zaporka `pero`. Koraci su sljedeći.

  - *Stvaranje baze*

    Kao korisnik `sa` moramo se spojiti za novu bazu i atributom `create` zatražiti njezino stvaranje. Apache Derby nema naredbe `CREATE DATABASE` koja se obično koristi za ovu namjenu već zahtjeva navođenje atributa `create` prilikom spajanja na (do tada nepostojeću) bazu. Naredba je:

    ```
    ij> connect 'jdbc:derby://localhost:1527/baza1DB;user=sa;
    password=sapwd22;create=true';
    ```

- *Dodavanje novog korisnika*

  Nakon što smo kao korisnik `sa` spojeni na novu bazu, potrebno je zadati naredbu kojom ćemo stvoriti novog korisnika.

  ```
  ij> CALL SYSCS_UTIL.SYSCS_CREATE_USER('perica', 'pero');
  ```

  Prvi argument koji dajemo ugrađenoj proceduri je korisničko ime; drugi argument je zaporka.

- *Definiranje dozvola*

  Apache Derby podržava fino definiranje dozvola uporabom SQL naredbe `GRANT`. Međutim, kako novom korisniku želimo dati apsolutne dozvole nad trenutnom bazom, dovoljno je pozvati ugrađenu proceduru koja će korisnika zapamtiti kao *super-korisnika* trenutne baze.

  ```
  ij> CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
      'derby.database.fullAccessUsers', 'perica');
  ```

  Da smo htjeli definirati više korisnika koji imaju u bazi sve ovlasti, naveli bismo ih sve kao drugi argument uz uporabu zareza za razdvajanje. Primjerice: `'perica,ivica,ante'`.

- *Otpajanje od baze*

  Nakon dodavanja novog korisnika i podešavanja svih potrebnih dozvola, potrebno je otpojiti se od baze.

  ```
  ij> disconnect;
  ```

- *Spajanje na bazu kao novi korisnik*

  Sada se, konačno, na bazu možemo spojiti kao novi korisnik kojem smo prethodno podesili.

  ```
  ij> connect 'jdbc:derby://localhost:1527/baza1DB;user=perica;
  password=pero';
  ```

Želite li ugasiti poslužitelj, u konzoli zadajte naredbu:

```
java -Dderby.system.home=D:\tmp\derby-baze
  -jar %DERBY_INSTALL%\lib\derbyrun.jar server shutdown
  -user username -password password
```

Pri tome u prethodnom pozivu vrijednosti argumenata koji se šalju kao korisničko ime i zaporka treba podesiti tako da odgovaraju nekom od globalnih korisnika. Kako u našem slučaju postoji samo jedan takav korisnik, naredba kojom bismo ugasili poslužitelj bila bi sljedeća.

```
java -Dderby.system.home=D:\tmp\derby-baze
  -jar %DERBY_INSTALL%\lib\derbyrun.jar server shutdown
  -user sa -password sapwd22
```

Time bi se poslužitelj pokrenut u prethodnoj konzoli trebao ugasiti i konzola bi trebala postati upotrebljiva za zadavanje novih naredbi. Ako ste to isprobali i ako dalje želite koristiti poslužitelj, ponovno ga pokrenite i ostavite da radi.

Osnovni razredi

Prilikom pisanja klijenta koji će se spajati na upravitelja baze podataka bit će potrebno uporabom razreda `DriverManager` registrirati JDBC upravljački program ko-

jim je moguće komunicirati s upraviteljem, ili koristiti razred koji implementira sučelje `javax.sql.DataSource`. U prvom slučaju, razred koji predstavlja upravljački program i koji treba registrirati je: `org.apache.derby.jdbc.ClientDriver`. Ako se pak klijentska aplikacija oslanja na implementacije sučelja `DataSource`, potrebno je koristiti razred: `org.apache.derby.jdbc.ClientDataSource`. U oba slučaja, uz klijenta potrebno je distribuirati i arhivu `derbyclient.jar` koja se nalazi u direktoriju `$DERBY_INSTALL/lib` i koja sadrži implementacije spomenutih razreda.

Napomena: recentnije verzije platforme Java aplikacijama koje koriste stvaranje veza prema upravitelju baze uporabom razreda `DriverManager` omogućavaju preskakanje koraka u kojem se radi registracija upravljačkog programa, ako je upravljački program napisan na odgovarajući način. Konkretno, ako JAR u kojem se distribuira upravljački program ima direktoriju `META-INF/services` te u njemu datoteku `java.sql.Driver` (dakle, puno ime: `META-INF\services\java.sql.Driver`) te ako je u toj datoteci navedeno ime upravljačkog programa, virtualni stroj će ga automatski registrirati prilikom pokretanja programa. U tom slučaju eksplicitna se registracija može preskočiti. Posljednje inačice upravljačkog programa za Apache Derby to imaju korektno napravljeno (otvorite JAR `derbyclient.jar` i uvjerite se).

Naprednije mogućnosti

Osim nezavisnog načina rada u kojem se Apache Derby ponaša kao mrežno-dostupni poslužitelj, ovaj upravitelj podržava i druge načine rada. Primjerice, klijentska aplikacija upravitelj baze podataka može koristiti kao običnu biblioteku pri čemu i upravitelj baze i klijent žive u istom procesu operacijskog sustava. Prilikom takvog rada (*embedded mode*), klijent može koristiti vlastite privatne baze podataka koje upravitelj lokalno sprema na disk i koje drugi klijenti ne dijele. Također, upravitelj podržava izradu memorijskih baza podataka -- baza koje žive samo u memoriji procesa i koje, u određenom smislu, za klijenta predstavljaju napredne implementacije kolekcija koje je moguće pretraživati i modificirati uobičajenim SQL naredbama. Zainteresirane se čitatelje upućuje na odgovarajuću dokumentaciju[3].

[3] http://db.apache.org/derby/manuals/#docs_10.10

Kazalo

A
automatsko upravljanje resursima, 413

D
dekorator, oblikovni obrazac, 406

I
iterator, oblikovni obrazac, 181

K
kompozit, oblikovni obrazac, 501

O
oblikovni obrasci
  Dekorator, 406
  Iterator, 181
  Kompozit, 501
  Okvirna metoda, 454
  Posjetitelj, 504
  Strategija, 188, 266
okvirna metoda, oblikovni obrazac, 454

P
posjetitelj, oblikovni obrazac, 504

S
strategija, oblikovni obrazac, 188, 266

Bibliography

[1] *Unicode*. http://www.unicode.org/.

[2] *Java™ Platform, Standard Edition 7*. API Specification. http://docs.oracle.com/javase/7/docs/api/index.html.

[3] *Java Platform, Enterprise Edition 6 (Java EE) Technical Documentation*. http://docs.oracle.com/javaee/index.html.

[4] *Java ME & Java Card Technology Documentation*. http://docs.oracle.com/javame/.

[5] *Android SDK*. http://developer.android.com/sdk/index.html.

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification*. Java SE 7 Edition. http://docs.oracle.com/javase/specs/jls/se7/html/. 2012-07-27.

[7] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification*. Java SE 7 Edition. http://docs.oracle.com/javase/specs/jvms/se7/html/index.html. 2012-07-27.

[8] Josh Juneau, Jim Baker, Victor Ng, Leo Soto, and Frank Wierzbicki. *The Definitive Guide to Jython*. Python for the Java Platform. http://www.jython.org/jythonbook/en/1.0/. 2010-03-25.

[9] *Code Conventions For The Java Programming Language*. http://www.oracle.com/technetwork/java/codeconv-138413.html. 1999-04-20.