

Programiranje u Javi

Marko Čupić

Programiranje u Javi

Marko Čupić

Ovo je verzija 0.3.11 knjige o programiranju u programskom jeziku Java. Napisani tekst odnosi se na verziju 7 programskog jezika Java.

Autorska prava © 2012. - 2014. 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	xv
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	12
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	33
Uporaba alata maven	38
Uporaba alata gradle	40
3. Osiguravanje kvalitete i ispravnosti koda	43
Alat CheckStyle	48
Alat PMD	50
Alat FindBugs	53
Alat JUnit	55
Alat JaCoCo	61
4. Jednostavniji primjeri	69
Primjer 1: ispis argumenata naredbenog retka	69
Primjer 2: izračun e^x	70
Primjer 3: formatirani ispis decimalnih brojeva	72
Primjer 4: izračun sume brojeva s tipkovnice	74
Naprednije čitanje ulaznih podataka	75
Formatirana izrada teksta	76
Stringovi u Javi	76
Primjer 5: metode s varijabilnim brojem argumenata	82
5. Razredi	87
Motivacijski primjer	87
Java je objektno-orientirani programski jezik	92
Konstruktori	94
Destruktori	96
Delegiranje zadaće konstrukcije objekta	97
Prvi razredi	99
Zaštita privatnosti elemenata razreda	116
Na vrhu stabla: razred Object	117
String toString();	118
int hashCode();	119
boolean equals(Object obj);	119
Metode wait, notify i notifyAll	120
protected void finalize()	120
Class<?> getClass();	121

Inicijalizacijski blokovi	122
6. Upravljanje pogreškama. Iznimke.	125
Iznimke	128
Obrada više iznimaka	135
Bezuvjetno izvođenje koda	137
Automatsko upravljanje resursima	139
Definiranje novih vrsta iznimaka	140
7. Apstraktni razredi. Sučelja.	143
Apstraktni razredi	143
Sučelja	147
Primjer uporabe sučelja	151
Modifikator <code>static</code>	155
Vrste razreda	160
Modifikator <code>final</code>	169
Programiranje prema sučeljima	170
Motivacijski primjer	172
Oblikovni obrazac <code>Iterator</code>	175
8. Načela objektno-orijentiranog oblikovanja	179
Na čvrstim temeljima	181
Načelo jedinstvene odgovornosti	183
Načelo otvorenosti za nadogradnju a zatvorenosti za promjene	188
Liskovino načelo supstitucije	194
Načelo izdvanja sučelja	205
Načelo inverzije ovisnosti	210
9. Kolekcije	223
10. Kolekcija: skup	225
11. Kolekcija: lista	227
12. Kolekcija: mapa	229
13. Kolekcije i novi razredi	231
14. Tehnologija Java Generics	233
15. Rad s datotečnim sustavom	235
Razred <code>java.io.File</code>	235
Paket <code>java.nio.file</code>	242
Razred <code>java.nio.file.Files</code>	247
Preostali važniji razredi	261
16. Tokovi okteta i znakova	263
Tokovi okteta	264
Nadogradnja funkcionalnosti	268
Oblikovni obrazac <code>Dekorator</code>	268
Znakovi i okteti	268
Tokovi znakova	268
Primjer uporabe	268
Oblikovni obrazac <code>Strategija</code>	268
Oblikovni obrazac <code>Okvirna metoda</code>	269
Dodatni zahtjevi	269
Datoteke sa slučajnim pristupom	269
17. Studija slučaja: programski jezik <i>vlang</i>	271
Osnovni razredi	272
Provedba leksičke analize	274
Izrada parsera i izgradnja sintaksnog stabla	283
Izvođenje programa: pokušaj prvi	290
Izvođenje programa: pokušaj drugi	294
Izvođenje programa: pokušaj treći	299
18. Višedretvene aplikacije	317
19. Raspodijeljene aplikacije. Paket <code>java.net</code>	319
20. Swing 1: općenito	321
21. Swing 2: modeli i pogledi	323
22. Swing 3: vlastite komponente i višedretvenost	325

23. Uporaba relacijskih baza podataka iz Jave: SQL	327
24. Uporaba relacijskih baza podataka iz Jave: JPA	329
25. Razvoj web aplikacija	331
26. Web obrasci	333
27. Tehnologija Java Servlet i JSP	335
28. Višeslojne web-aplikacije	337
29. IOC kontejneri	339
30. Test Driven Development	341
31. Razvoj aplikacija za Android	343
A. Instalacija JDK	345
Operacijski sustav Windows	345
Operacijski sustav Linux	346
B. Instalacija alata ant , maven i gradle	347
Alat Apache Ant	347
Alat Apache Maven	347
Alat Gradle	348
C. Instalacija alata za kontrolu kvalitete i ispravnosti koda	349
Alat CheckStyle	349
Alat PMD	350
Alat FindBugs	350
Alat JUnit	351
Alat JaCoCo	351
Biblioteka Mockito	351
D. Instalacija poslužitelja servleta	353
Instalacija poslužitelja Apache Tomcat	353
Podešavanje kodne stranice i omogućavanje praćenja izvođenja	354
Dodatno podešavanje kodne stranice	355
Instalacija poslužitelja Jetty	355
E. Instalacija upravitelja bazama podataka	357
Instalacija upravitelja Apache Derby	357
Podešavanje upravitelja baze	358
Bibliography	363

Popis slika

1.1. Od izvornog koda do programa u izvođenju	3
2.1. Pojednostavljen popis zadataka u razvojnom procesu	34
3.1. Sažetak izvještaja o kontroli stila pisanja koda	50
3.2. Izvještaja o kontroli stila pisanja koda datoteke <code>Formule.java</code>	50
3.3. Sažetak izvještaja o kontroli stila pisanja koda	53
3.4. Sažetak izvještaja o dupliciranju koda	53
3.5. Sažetak izvještaja o kontroli kvalitete koda	55
3.6. Izvještaj o pokrenutim testovima	60
3.7. Detaljni pregled jednog izvještaja o pokrenutim testovima	61
3.8. Izvještaj o pokrivenosti koda	66
3.9. Detaljniji izvještaj o pokrivenosti koda	66
3.10. Izvještaj o pokrivenosti koda u razredi <code>PrirodniTest</code>	66
3.11. Izvještaj o pokrivenosti koda u razredi <code>Prirodni</code>	66
3.12. Detaljni grafički prikaz o pokrivenosti koda u razredu <code>Prirodni</code>	67
4.1. Stanje u memoriji (1)	77
4.2. Stanje u memoriji (2)	78
4.3. Stanje u memoriji (3)	78
4.4. Stanje u memoriji (4)	79
5.1. Stanje u memoriji	101
5.2. Stanje u memoriji	104
5.3. Detaljno stanje u memoriji	107
5.4. Dijagram razreda	109
6.1. Struktura iznimki u Javi	132
7.1. Stanje u memoriji	156
7.2. Stanje u memoriji	158
7.3. Stanje u memoriji	163
7.4. Razredi i sučelja motivacijskog primjera	175
7.5. Dijagram razreda za oblikovni obrazac <code>Iterator</code>	176
8.1. Organizacija dnevnčkog podsustava	193
8.2. Organizacija lošeg rješenja	207
8.3. Organizacija boljeg rješenja	210
8.4. Organizacija prvog predloženog rješenja	212
8.5. Organizacija drugog predloženog rješenja	216
8.6. Organizacija rješenja uz inverziju i injekciju ovisnosti	221
16.1. Primjer ulaznih tokova okteta.	268
17.1. Dijagram toka obrade programa	272
17.2. Razred <code>Vector</code>	274
17.3. Podstablo iznimaka	274
17.4. Podstablo za pamćenje naredbi	284
17.5. Podstablo za pamćenje izraza	285
17.6. Modificirano podstablo za pamćenje izraza	295
17.7. Apstraktna operacija	300
17.8. Apstraktna operacija	302
17.9. Apstraktna operacija nad naredbama	308
17.10. Apstraktna operacija	309

Popis tablica

1.1. Primitivni tipovi u programskom jeziku Java	7
5.1. Razine pristupa članskim elementima razreda	117

Popis primjera

1.1. Ilustracija računanja s ograničenom preciznošću te usporedba rezultata	18
1.2. Usporedba jednakosti decimalnih brojeva	19
2.1. Primjer programa napisanog u programskom jeziku Java	22
2.2. Složeniji primjer -- ispis slučajno odabrane formule	26
2.3. Primjer dokumentacije paketa	30
2.4. Primjer dokumentacije razreda	31
2.5. Primjer dokumentacije metode	32
2.6. Primjer konfiguracije za alat ant	34
2.7. Definicija cilja runjar za alat ant	37
2.8. Pregledna stranica za dokumentaciju programa	38
2.9. Definicija cilja javadoc za alat ant	38
2.10. Primjer osnovne verzije datoteke <code>pom.xml</code>	39
2.11. Zadavanje verzije prevodioca koji je potrebno koristiti u datoteci <code>pom.xml</code>	40
2.12. Konfiguracijska datoteka <code>build.gradle</code> za alat gradle	41
3.1. Implementacija biblioteke za rad s prirodnim brojevima	55
3.2. Testovi za razred <code>Prirodni</code>	56
3.3. Datoteka <code>build.xml</code> za pokretanje testova	57
3.4. Cjelokupna datoteka <code>build.xml</code> koja koristi alat JaCoCo	63
4.1. Ispis argumenata naredbenog retka	69
4.2. Izračun e^x	70
4.3. Formatirani ispis brojeva	72
4.4. Sumiranje brojeva unesenih preko tipkovnice	74
4.5. Konkatenacija stringova	79
4.6. Metode s varijabilnim brojem argumenata	82
5.1. Modeliranje pravokutnika u programskom jeziku C	87
5.2. Funkcije za rad s pravokutnicima u programskom jeziku C	87
5.3. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 1	90
5.4. Modeliranje više vrsta pravokutnika u programskom jeziku C, pokušaj 2	91
5.5. Definiranje razreda <code>Pravokutnik</code> : Java	92
5.6. Definiranje razreda <code>Pravokutnik</code> : C++	93
5.7. Definiranje razreda <code>Pravokutnik</code> : Python	93
5.8. Definiranje više konstruktora	97
5.9. Definiranje više konstruktora -- bolje	98
5.10. Razred <code>GeometrijskiLik</code>	99
5.11. Primjer uporabe razreda <code>GeometrijskiLik</code>	101
5.12. Definiranje razreda <code>Pravokutnik</code>	102
5.13. Primjer uporabe razreda <code>Pravokutnik</code>	103
5.14. Nadjačavanje metoda u razredu <code>Pravokutnik</code>	104
5.15. Definiranje razreda <code>Krug</code>	108
5.16. Definiranje razreda <code>Slika</code> i primjer uporabe	109
5.17. Dodavanje mogućnosti crtanja likova	111
6.1. Primjer stvaranja iznimke	128
6.2. Primjer obrade iznimke	130
6.3. Iznimke porodice <code>RuntimeException</code>	132
6.4. Primjer rada s resursima	137
6.5. Definiranje nove vrste iznimke	140
7.1. Razred <code>GeometrijskiLik</code> u apstraktnom izdanju	144
7.2. Razred <code>Krug</code>	145
7.3. Razred <code>ComplexNumber</code> i primjer uporabe	155
7.4. Nova inačica razreda <code>ComplexNumber</code> i primjer uporabe	157
7.5. Nadopunjena inačica razreda <code>ComplexNumber</code> i primjer uporabe	160
7.6. Primjer definicije i uporabe lokalnog razreda	164
7.7. Primjer definicije i uporabe anonimnih razreda	166
7.8. Sučelje <code>java.util.Iterator<E></code>	170
7.9. Sučelje <code>java.lang.Iterable<E></code>	171

7.10. Obilazak skupnog objekta koji implementira sučelje	
<code>java.lang.Iterable<E></code>	171
7.11. Implementacija skupnog objekta koji predstavlja zadani podniz parnih brojeva	172
7.12. Implementacija klijenta za skupni objekt	174
8.1. Generiranje izvještaja, prvi pokušaj	183
8.2. Generiranje izvještaja, drugi pokušaj	184
8.3. Generiranje izvještaja, treći pokušaj	185
8.4. Razred <code>Student</code>	186
8.5. Razred <code>Student</code> i primjena načela jedinstvene odgovornosti	187
8.6. Primjer problematične metode <code>racunajPovrsinu</code>	188
8.7. Bolji primjer metode <code>racunajPovrsinu</code>	190
8.8. Primjer izrade dnevničkog podsustava	191
8.9. Bolji primjer izrade dnevničkog podsustava	192
8.10. Višestruko zapisivanje dnevničkih poruka	194
8.11. Geometrijski podsustav	195
8.12. Primjer uporabe	196
8.13. Implementacija kvadrata	197
8.14. Popravljen implementacija kvadrata	197
8.15. Nadogradnja glavnog programa	198
8.16. Moguće rješenje	199
8.17. Lokalizirane poruke	201
8.18. Poruke bez mogućnosti ažuriranja	202
8.19. Poruke bez mogućnosti ažuriranja	202
8.20. Problematična organizacija koda	205
8.21. Bolja organizacija koda	208
8.22. Podsustav za provjeru valjanosti transakcija, prvi pokušaj	211
8.23. Podsustav za provjeru valjanosti transakcija, drugi pokušaj	213
8.24. Podsustav za provjeru valjanosti transakcija, rješenje	217
15.1. Ispis separatora	236
15.2. Ispis vršnih datotečnih objekata	236
15.3. Rekurzivni ispis sadržaja direktorija	239
15.4. Ispis stabla direktorija	240
15.5. Filtrirani ispis sadržaja direktorija	241
15.6. Modeliranje staze sučeljem <code>Path</code>	243
15.7. Nadzor direktorija uporabom <code>WatchService</code>	245
15.8. Ispis sadržaja direktorija	250
15.9. Čitanje iz i zapisivanje u datoteku	251
15.10. Izračun statističkih podataka, 1. pokušaj	253
15.11. Sučelje <code>FileVisitor</code>	256
15.12. Izračun statističkih podataka, bolje	257
16.1. Primjer ulaznih tokova okteta.	265
17.1. Primjer programa napisanog jezikom <i>vlang</i>	271
17.2. Model nepromjenjivog vektora realnih brojeva.	272
17.3. Enumeracija različitih vrsta tokena.	275
17.4. Model jednog tokena.	276
17.5. Implementacija tokenizatora.	277
17.6. Demonstracija rada tokenizatora.	282
17.7. Gramatika jezika <i>vlang</i>	283
17.8. Implementacija parsera.	285
17.9. Demonstracija uporabe parsera.	290
17.10. Izvođenje programa (1)	290
17.11. Izvođenje programa (1)	293
17.12. Izmjene razreda koji predstavljaju izraze	295
17.13. Poboljšana izvedba izvođenja programa	297
17.14. Definicija apstraktne operacije nad izrazom	300
17.15. Uvođenje potpore za primjenu oblikovnog obrasca <i>Posjetitelj</i> u model izraza	302
17.16. Posjetitelj koji računa vrijednost izraza	304
17.17. Posjetitelj koji izraz pretvara u tekst	305

17.18. Poboljšani kod za izvođenje programa	306
17.19. Apstraktna definicija operacije nad naredbama	308
17.20. Modifikacije modela naredbe	309
17.21. Posjetitelj koji izvodi napisani program	310
17.22. Posjetitelj koji rekonstruira izvorni kod programa	312
17.23. Konačna implementacija koda koji izvodi napisani program	313
17.24. Primjer pokretanja prevođenja i izvođenja koda	314

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 višepלטformnost, 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 biblioteke 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. Umjesto 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.

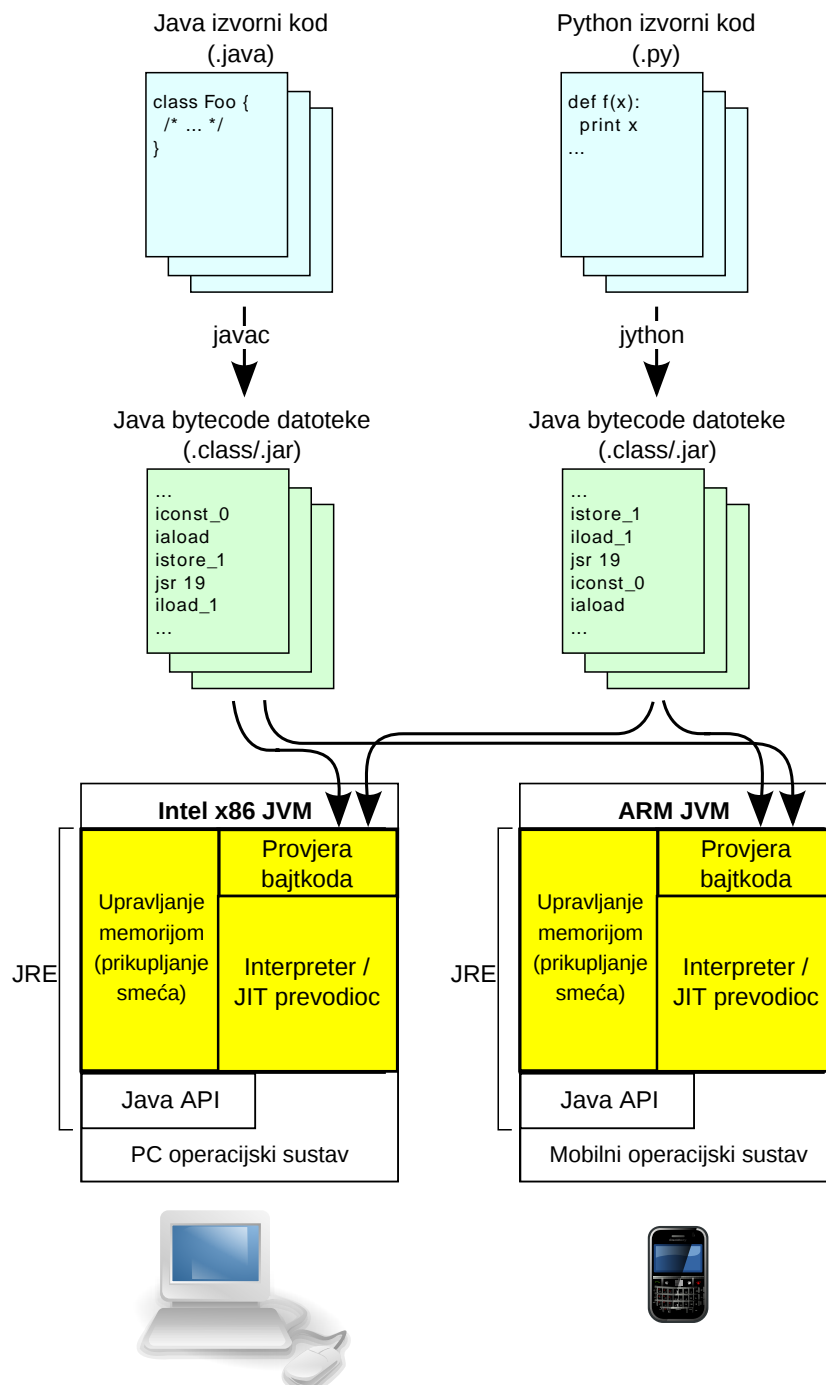
Spomenimo na kraju pregleda još i Javin ekosustav. Osim standardnih biblioteka koje su dostupne u okviru Java platforme, oko Java platforme stvoren je bogat i živ ekosustav koji je iznjedrio čitav niz drugih biblioteka, razvojnih okvira pa čak i novih programskih jezika koji su danas u uporabi. Primjerice, 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, Haskell, JavaScripta i sličnima te da te programe izvede 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, višepatformne 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 rješenja preporučljivo 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 NetBeans.
- 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 razvojni 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 raspoložemo 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 konkatencija 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 implementacije 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, virtualni 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 programski 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 razriješi oве 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.

Tip	Opis
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.
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 brojeve 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 brojeve 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: <code>true</code> te <code>false</code> .
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 <code>'\u0000'</code> do <code>'\uffff'</code> . 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 postfix 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 `&` (`&`), `|` (`|`) te isključivo `|` (`^`). 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 cjelobrojnim 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 `||` 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 razrešiti 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 prepozna 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 zahtijeva 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 $x \geq 0$. 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 Java 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 ugniježdene `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 prepisite 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 okviru 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 razmještanju 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 dojavljuje 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 proglašavaju 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 datoteka `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 stroja 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 obaviti 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 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 razložiti 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 pozdravnu 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 generirane 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 poprilično 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 raspoložemo 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 višelinijjskih 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 poprilično 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

```
1 /**
2  * Razred <code>ParniBrojevi</code> korisnicima omogućava rad
3  * i obilazak po cijelim parnim brojevima.
4  *
5  * <p>Najčešći način uporabe jest stvaranje ograničenog
6  * podskupa cijelih brojeva i njihov obilazak petljom, kako
7  * je ilustrirano u sljedećem primjeru.</p>
8  *
9  * <pre>
10 * ParniBrojevi pb = new ParniBrojevi(100);
11 * for(Integer broj : pb) {
12 *     // radi nešto s brojem
13 * }
14 * </pre>
15 *
16 * @author Marko Čupić
17 * @version 1.1
18 */
19 public class ParniBrojevi {
20
21     ...
22
23 }
24
```

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

```

1 /**
2  * Metoda provjerava nalazi li se broj predan kao argument
3  * u skupu parnih brojeva koje trenutni primjerak razreda
4  * predstavlja. Argument može biti bilo koji cijeli broj,
5  * međutim, ne smije biti null.
6  *
7  * @param broj broj koji treba ispitati
8  * @return true ako je predani argument u
9  *         skupu brojeva koje predstavlja primjerak razreda;
10 *        false inače.
11 * @throws NullPointerException ako se kao argument preda
12 *        vrijednost null.
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 razvojnem 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` iz prethodnog projekta `projekt2`. Struktura koja će time nastati prikazana je u nastavku.

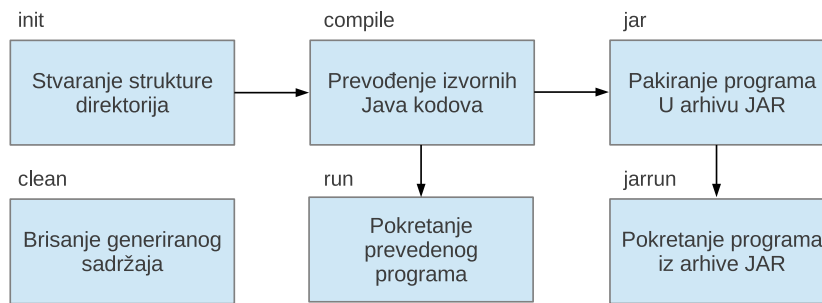
```
D:\
  +-- javaproj
    +-- projekt3
      +-- src
        + main
          +-- java
            +-- hr
              +-- fer
                +-- zemris
                  +-- java
                    +-- tecaj_1
                      +-- Formule.java
                      +-- Glavni.java
```

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 razvojnog 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*) služiti 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 razrješavati 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 preduvijeta 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**

```

1 <target name="jarrun" depends="jar"
2   description="Pokretanje programa iz JAR-a">
3   <!-- Poziv virtualnog stroja koji će pokrenuti
4     napisani program. -->
5   <java classname="${program}"
6     classpath="${dist.lib}/${ant.project.name}-${DSTAMP}.jar"
7     fork="true">
8     <jvmarg value="-Dfile.encoding=IBM852" />
9   </java>
10 </target>
11
12
```

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

```

1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2 <HTML>
3   <HEAD>
4     <TITLE>Prvi program</TITLE>
5   </HEAD>
6   <BODY>
7     Ovaj program služi demonstraciji osnovnih koraka pri izradi
8     Java programa, njegovog prevođenja, pakiranja, pokretanja
9     te generiranja dokumentacije. Više informacija o samim
10    razredima dostupno je u okviru dokumentacije istih.
11  </BODY>
12 </HTML>
13
14

```

Primjer 2.9. Definicija cilja javadoc za alat ant

```

1 <target name="javadoc" depends="compile"
2   description="Generiranje dokumentacije" >
3   <!-- Stvaranje direktorija za distribuciju -->
4   <mkdir dir="${dist}/doc"/>
5   <javadoc packagenames="hr.*"
6     sourcepath="${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. Obratite pažnju da bi prilikom rada s ovim alatom morali biti spojeni na Internet jer alat često želi dohvaćati potrebne verzije biblioteka direktno s Interneta (barem prvi puta, a kasnije pokušava provjeravati ima li još uvijek zadnju verziju biblioteka ili je u međuvremenu izašlo nešto novije). Pozicionirajte se u vršni direktorij u kojem smo radili sve projekte (D:\javaproj). Potom zadajte sljedeću naredbu.

```
D:\javaproj>mvn archetype:generate \
    -DgroupId=hr.fer.zemris.java.tecaj_1 \
```



```
-DartifactId=projekt3m \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DinteractiveMode=false
```

Pokrenut će se alat **maven** koji će za nas izgenerirati prazan projekt i cjelokupnu strukturu direktorija. Vršni direktorij projekta odgovarat će zadanom nazivu artifakta 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 **maven**. 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 redoslijedom 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 1.6. 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>2.3.2</version>
7     <configuration>
8       <source>1.6</source>
9       <target>1.6</target>
10    </configuration>
11  </plugin>
12 </plugins>
13 </build>
14
```

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`.

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 2.12.

Primjer 2.12. 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
```

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 smisljeno.*

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 višedimenzijских 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 assembleru; 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, uspjeh 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 zahtijevati 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.pupercrawl.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.puppcrawl.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 učitava 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 temeljem 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

CheckStyle Audit	
Designed for use with CheckStyle and Ant .	
Summary	
Files	Errors
2	12
Files	
Name	Errors
d:\javaproj\projekt4\src\main\java\hr\fer\zemris\java\tečaj_1\Formule.java	7
d:\javaproj\projekt4\src\main\java\hr\fer\zemris\java\tečaj_1\Glavni.java	5

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

CheckStyle Audit	
Designed for use with CheckStyle and Ant .	
File d:\javaproj\projekt4\src\main\java\hr\fer\zemris\java\tečaj_1\Formule.java	
Error Description	Line
File does not end with a newline.	0
Missing package-info.java file.	0
Utility classes should not have a public or default constructor.	13
File contains tab characters (this is the first instance).	15
Line has trailing spaces.	18
'0.5' is a magic number.	22
'if' is not followed by whitespace.	25

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 pomoćna 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

PMD 5.0.2 Report 2013-03-07 - 19:30:10

Summary

Files	Total	Priority 1	Priority 2	Priority 3	Priority 4	Priority 5
2	4	0	0	4	0	0

Prio	File	Line	Description
3	hr\fer\zemris\java\tecaj_1\Formule	13	All methods are static. Consider using Singleton instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
3	hr\fer\zemris\java\tecaj_1\Formule	22	Local variable 'prva' could be declared final
3	hr\fer\zemris\java\tecaj_1\Glavni	9	All methods are static. Consider using Singleton instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
3	hr\fer\zemris\java\tecaj_1\Glavni	15	Parameter 'args' is not assigned and could be declared final

Generated by [PMD 5.0.2](#) on 2013-03-07 - 19:30:10.

Slika 3.4. Sažetak izvještaja o dupliciranju koda

Summary of duplicated code

This page summarizes the code fragments that have been found to be replicated in the code. Only those fragments longer than 30 lines of code are shown.

# duplications	Total lines	Total tokens	Approx # bytes
0	0	0	0

You expand and collapse the code fragments using the + buttons. You can also navigate to the source code by clicking on the file names.

ID **Files** **Lines**

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 temeljem 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

Package	Code Size	Bugs	High Prio Bugs	Medium Prio Bugs	Low Prio Bugs	Exp. Bugs	Ratio
Overall (1 packages), (2 classes)	20	0					

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  * između 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
```

```

13  * pozitivni cijeli brojevi (dakle, nula se neće tretirati kao
14  * pozitivan cijeli broj).
15  *
16  * <p>Evo i jednostavnog primjera uporabe. Da biste postavili
17  * vrijednost zastavice {@code prirodni} na {@code true} ako
18  * je predani broj {@code n} prirodni, možete se poslužiti
19  * sljedećim isječkom koda.</p>
20  * <pre>
21  *     int n = 153;
22  *     boolean prirodni = Prirodni.jePrirodni(n);
23  *     System.out.println("Broj " + n + " je prirodni? "
24  *                         + prirodni);
25  * </pre>
26  *
27  * @since 1.0
28  * @author Marko Čupić
29  */
30  public class Prirodni {
31
32      /**
33       * Funkcija ispituje je li predani broj prirodni broj.
34       * @param n cijeli broj koji se ispituje
35       * @return <code>true</code> ako je predani cijeli broj
36       *         ujedno i prirodni broj; inače vraća
37       *         <code>false</code>.
38       */
39      public static boolean jePrirodni(int n) {
40          return n > 0;
41      }
42
43      /**
44       * Funkcija vraća sljedbenika od predanog broja.
45       * @param n broj čijeg sljedbenika treba vratiti
46       * @return <code>true</code> ako je predani cijeli broj
47       *         ujedno i prirodni broj; inače vraća
48       *         <code>false</code>.
49       */
50      public static int sljedbenik(int n) {
51          return n+1;
52      }
53
54  }

```

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` napraviti 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 spremati 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 odrediti 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 pronade 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.

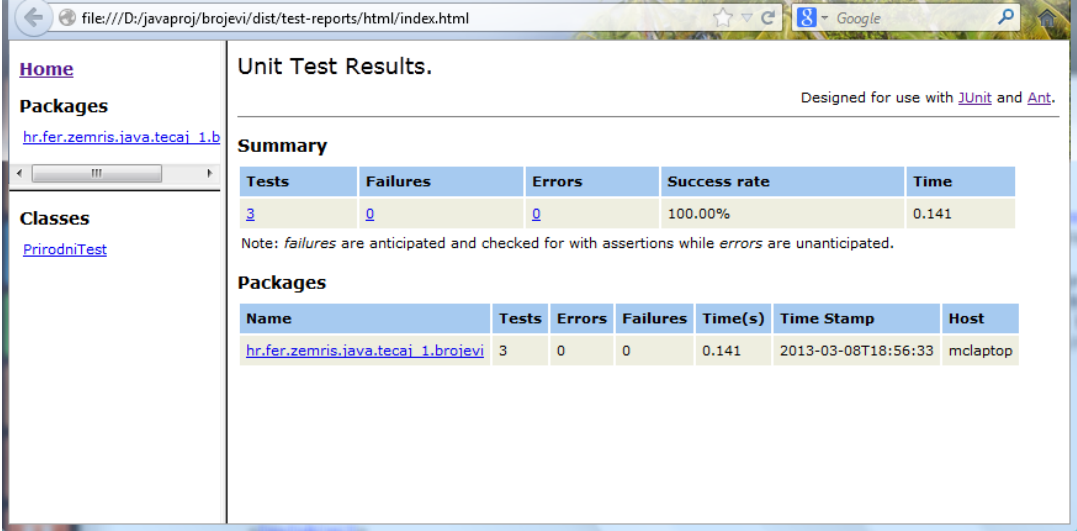
Prikazana datoteka `build.xml` je nešto jednostavnija od prethodnih jer su iz nje uklonjeni svi dijelovi koji nam trenutno nisu bili u fokusu (primjerice, izrada JAR arhive, izrada dokumentacije i slično).

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 temeljem 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



Unit Test Results. Designed for use with [JUnit](#) and [Ant](#).

Summary

Tests	Failures	Errors	Success rate	Time
3	0	0	100.00%	0.141

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
hr.fer.zemris.java.tecaj 1.brojevi	3	0	0	0.141	2013-03-08T18:56:33	mcclaptop

Slika 3.7. Detaljni pregled jednog izvještaja o pokrenutim testovima

Class	Name	Status	Type	Time(s)
PrirodniTest	nulaNijePrirodniBroj	Success		0.015
PrirodniTest	pozitivniBrojJePrirodan	Success		0.000
PrirodniTest	negativanBrojNijePrirodan	Success		0.000

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 ispitati 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 pokazatelj 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="Prevodenje izvornog koda">
50   <mkdir dir="${build.classes}"/>
51   <!-- Prevodenje Java koda iz ${src} u ${build} -->
52   <javac srcdir="${src.java}" destdir="${build.classes}"
53     classpathref="compile.path"
54     encoding="UTF-8" debug="on"
55     debuglevel="lines,vars,source"
56     includeAntRuntime="false" />
57 </target>
58
59 <target name="compile-tests" depends="compile"
60   description="Prevodenje izvornog koda testova">
61   <mkdir dir="${build.test}"/>
62   <!-- Prevodenje 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

```

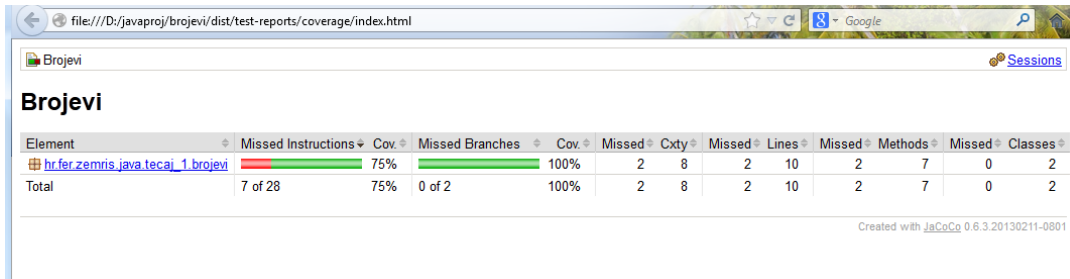
Nakon ovih modifikacija pokrenite cilj `run-tests`.

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

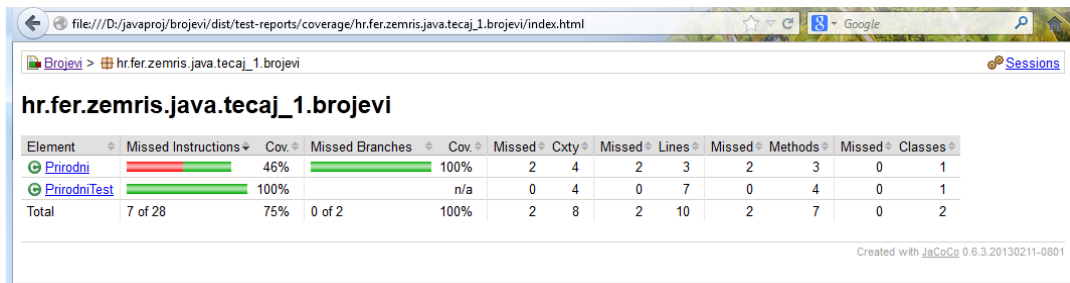
Kao rezultat ove akcije u direktoriju `dist/test-reports/html` nastat će kao i prije izvještaj za sve pokrenute testove u formatu HTML. Izvorni izvještaj u formatu XML bit će spremljen u direktoriju `dist/test-reports/xml` tako da je temeljem njega lagano generirati i izvještaje u drugim formatima. Uz 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 važno -- 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.

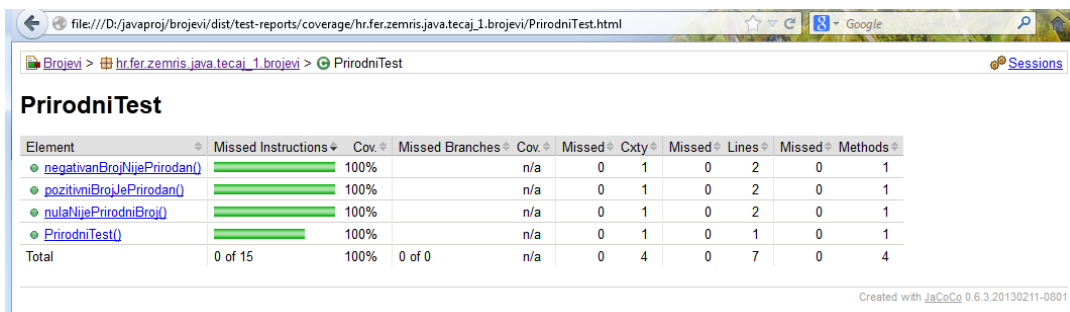
Slika 3.8. Izvještaj o pokrivenosti koda



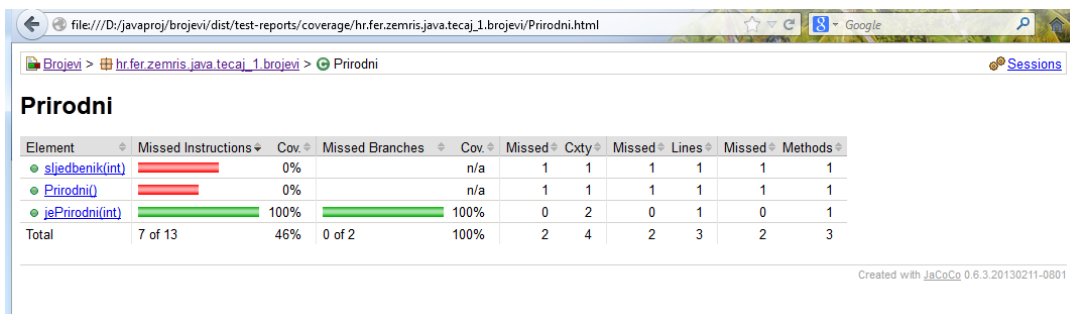
Slika 3.9. Detaljniji izvještaj o pokrivenosti koda



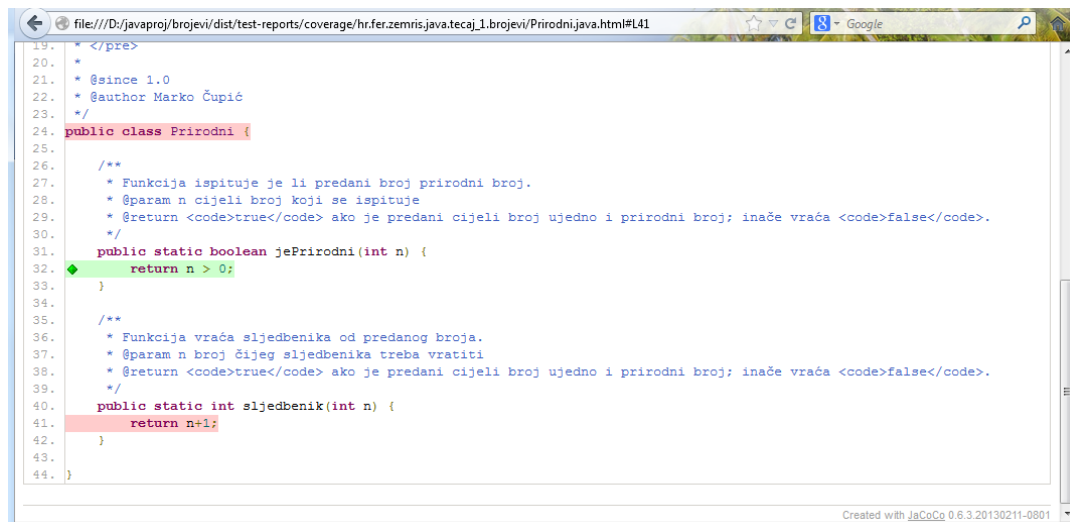
Slika 3.10. Izvještaj o pokrivenosti koda u razredi `PrirodniTest`



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 ljske).

```
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. Redoslijedom 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 zaslone 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 smjestiti 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  *  $e^x = 1 + x + (x^2/(2!)) + (x^3/(3!)) + (x^4/(4!)) + \dots$ 
36  *
37  * @param broj argument funkcije  $e^x$ 
38  *
39  * @return iznos funkcije u točki  $x = \text{broj}$  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 faktoriijela = 1.0;
46
47     suma += 1.0;
48
49     for(int i = 1; i < 10; i++) {
50         potencija = potencija * broj;
51         faktoriijela = faktoriijela * i;
52         suma += potencija/faktoriijela;
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 smjestiti 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 specifikaciji 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 točkazareza, 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 cjelobrojni 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 smjestiti 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 će 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`. Konkretni 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 potrebno 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 praznina. 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$d` možemo direktno referencirati na `x`-ti predani argument. Tako posljednji navedeni specifikator `%1$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

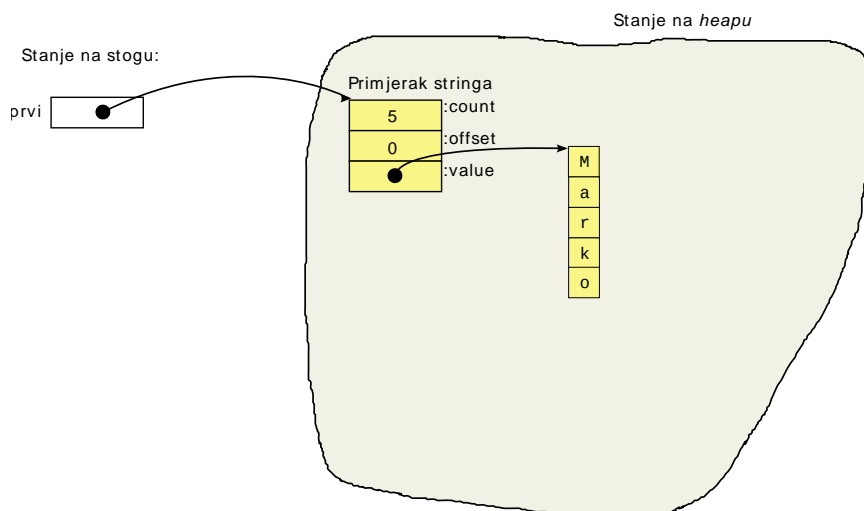
kompleksnost koja je povezana s provođenjem operacija nad njima. To pak, nažalost, često dovodi do situacije u kojoj korisnik nesvjesno piše loš kod jer nije svjestan što je zapravo napravio. Zadaća ovog podpoglavlja jest pojasniti kako se u Javi radi sa stringovima i koje su opasnosti.

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.

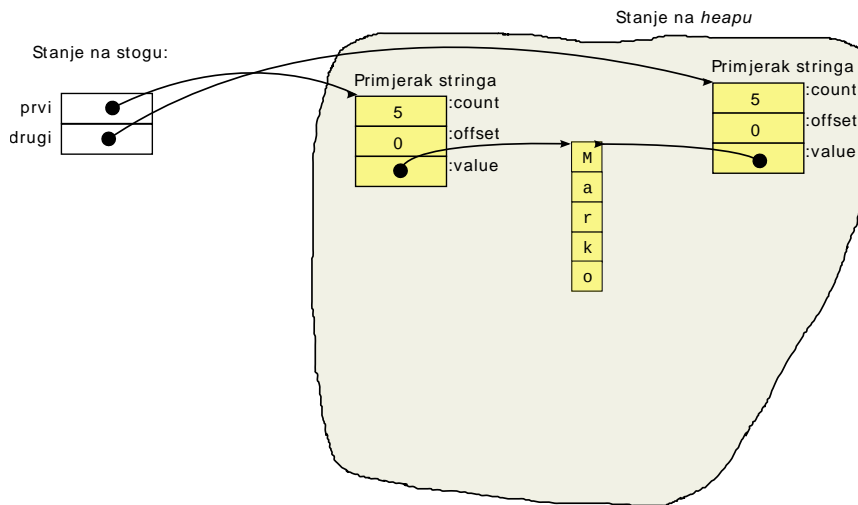
Slika 4.1. Stanje u memoriji (1)



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.

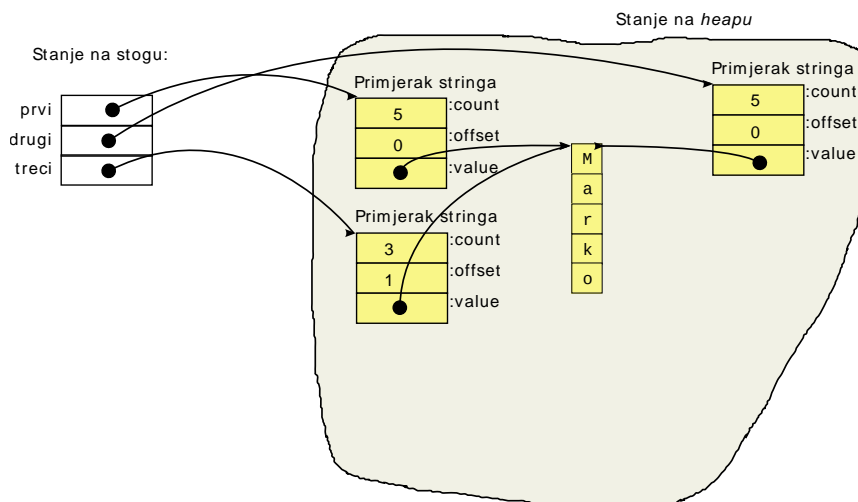
Slika 4.2. Stanje u memoriji (2)



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.

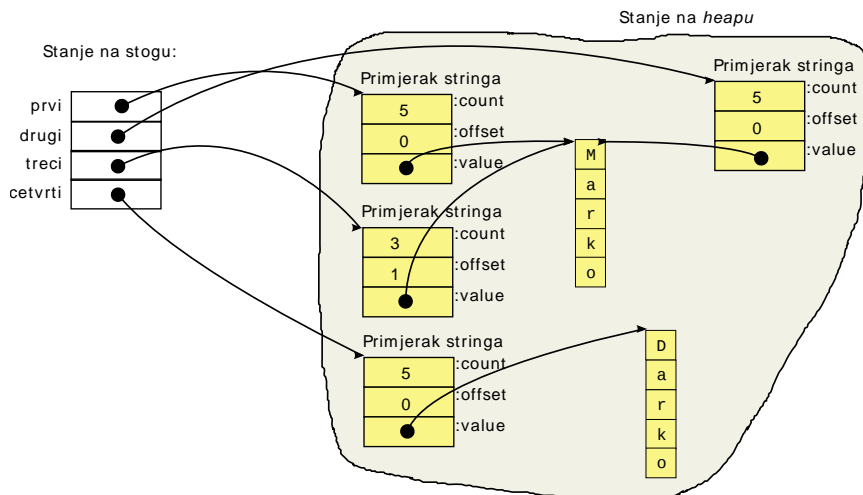
Slika 4.3. Stanje u memoriji (3)



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 provođenja ove operacije stvoren je novi objekt koji opet koristi isto znakovno polje. Razlika je samo u sadržaju varijable `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 konkatencija.



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. Konkatencija 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 konkatencija 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 konkatencije.

Redak 30 trenutni sadržaj konkatencira 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 konkatencija, 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 nadodavanja. 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 konkatencije 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 konkatencije 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 konkatencije 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 konkatenciju stringova: kada korisnik napiše sljedeći kod:

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

jezični prevodilac to prepíše 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 navedenom.

Dobro, i sada se vjerojatno pitate, znači li to da nikada ne koristite operator `+` za konkatenciju 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 lijevom simbolom, slijede predani brojevi
28  * koji su razdvojeni zarezima i završava desnom 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
65 1 package hr.fer.zemris.java.primjeri;
66 2
67 3 public class Primjer1 {
68 4
69 5     public static void main(String[] args) {
70 6         System.out.println(
71 7             Matematika.kaoTekst('[', ']', 1, 4, 8.2, 13.1)
72 8         );
73 9         System.out.println(Matematika.kaoTekst('[', ']', 3.3));
74 10        System.out.println(Matematika.kaoTekst('[', ']'));
75 11
76 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 varijabilnim 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*. Pogledajte ponovno razred `Primjer1`. Svaki poziv metoda `suma` te kao `Tekst` 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 prosljeđivati 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 nedostacima 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 riješeno 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 potihom 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, opisan 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 moguće 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 neriješen 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;
```

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 nudi 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 zahtijeva 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 Java 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 Java, 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 destrukturu 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, uočiti 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 potvrđan. 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 `getXxx` a *setter* metoda čiji je naziv `setXxx`; 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 *brojac* 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`

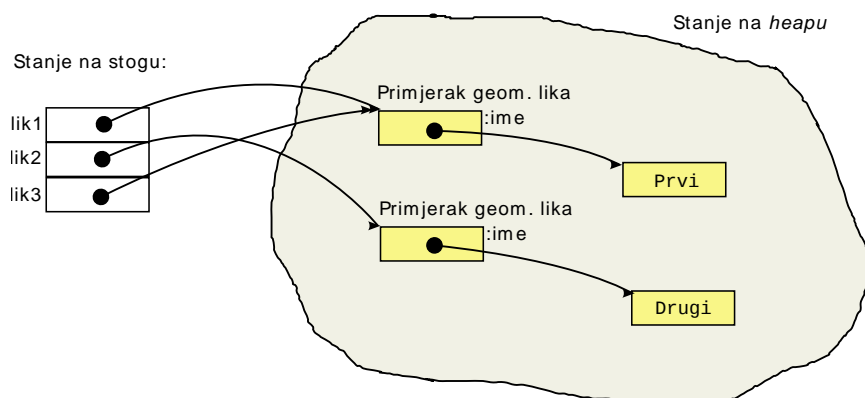
```

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 `String`ovi 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 `string` 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 imaće 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`.

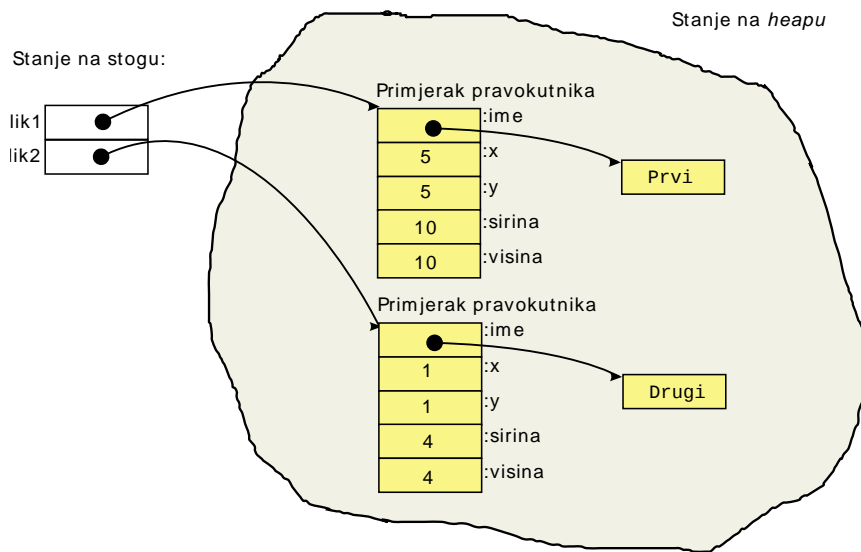
Primjer 5.13. Primjer uporabe razreda `Pravokutnik`

```

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. Popravljen 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, prijaviti će pogrešku. Ovo je zgodan kontrolni mehanizam koji će se pobrinuti da se odmah otkriju pogreške poput definiranja imena metode `getPovrsina` 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 nasljeđivanja 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.

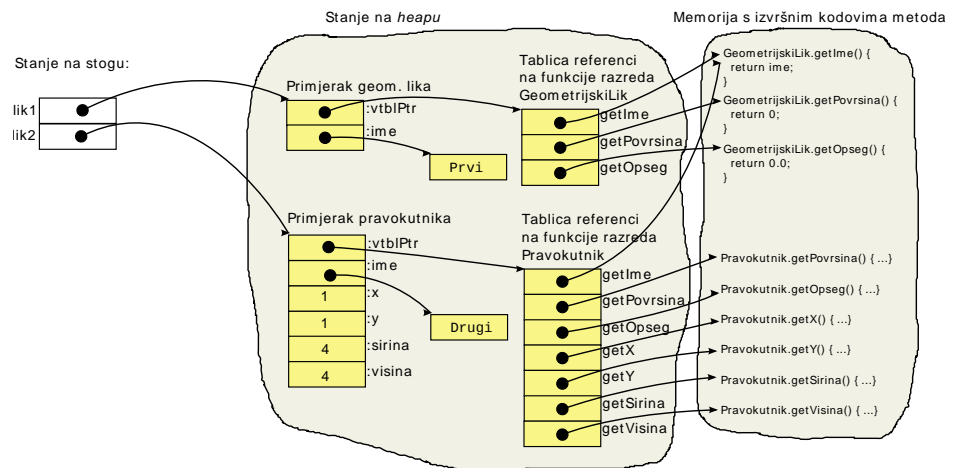
```
Površina lika Prvi je 0.0  
Površina lika Drugi je 16.0
```

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 taj razred 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`

```

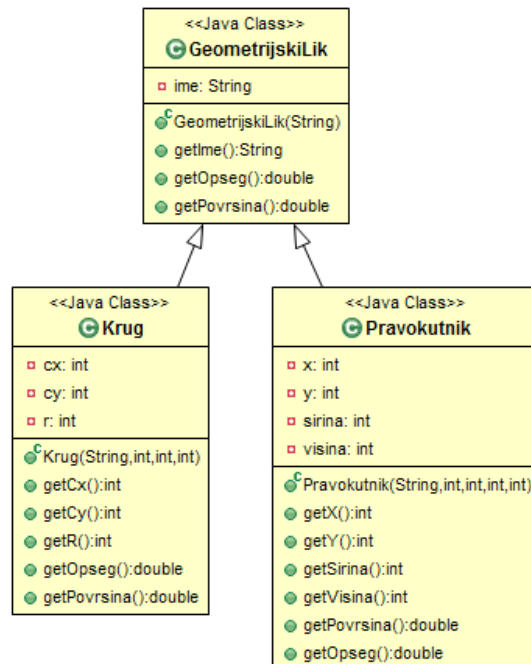
1 package hr.fer.zemris.java.tecaj_1;
2
3 public class Krug extends GeometrijskiLik {
4
5     private int cx;
6     private int cy;
7     private int r;
8
9     public Krug(String ime, int cx, int cy, int r) {
10         super(ime);
11         this.cx = cx;
12         this.cy = cy;
13         this.r = 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 poprilično 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 pamtit i kao dvodimenzijско 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     }
13 }
  
```

```

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
43
44  1 package hr.fer.zemris.java.tecaj_1;
45  2
46  3 public class PrimjerSlike {
47  4
48  5     public static void main(String[] args) {
49  6         Slika slika = new Slika(5, 3);
50  7         slika.upaliElement(0, 0);
51  8         slika.upaliElement(0, 1);
52  9         slika.upaliElement(0, 2);
53  10        slika.upaliElement(1, 1);
54  11        slika.upaliElement(2, 2);
55  12        slika.upaliElement(3, 2);
56  13        slika.upaliElement(4, 2);
57  14        System.out.print(slika.toString());
58  15    }
59  16
60  17 }
61  18

```

Razred `Slika` u konstruktoru prima informacije o širini i visini slike odnosno o dimenzijama rastera. Podatci o svakom slikovnom elementu pamte se u dvodimenzijском 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 }
```

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 rezultat će sljedećim ispisom.

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

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.



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.

```
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
<code>public</code>	DA	DA	DA	DA
<code>protected</code>	DA	DA	DA	NE
bez modifikatora	DA	DA	NE	NE
<code>private</code>	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 hashCode();`

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 uvjerali 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 destruktorkama 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 zadataka na koje se u ovom trenutku nećemo osvrnati.

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

saturn 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 redoslijedom 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 ilustrira 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
19
20 package hr.fer.zemris.java.primjeri;
21
22 public class RazredB extends RazredA {
23
24     {
25         System.out.println("B: prvi blok");
26     }
27
28     public RazredB() {
29         System.out.println("B: konstruktor.");
30     }
31
32     {
33         System.out.println("B: drugi blok");
34     }
35
36 }
37
38
39 package hr.fer.zemris.java.primjeri;
40
41 public class PrimjerIB {
42
43     public static void main(String[] args) {
44         new RazredB();
45     }
46
47 }
48
49 }
```

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 redoslijedom 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 daju 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 dojavljuje 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 rješenje, uz pretpostavku da realno rješenje uopće postoji. Kako rezultat izračuna funkcija vraća preko povratne vrijednosti, 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 dojava 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 `switch`ima, 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 prosljedila 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 dojavu što se dogodilo. Pri tome statusi pogrešaka 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` dojavu pogrešku: "Tag nije zatvoren znakom >" svom pozivatelju. Funkcija `parsirajXML` tada bi trebala svom pozivatelju javiti "XML dokument nije valjan". Funkcija `izgradiXMLStablo` tada bi svom pozivatelju 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 pronade 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 može 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).

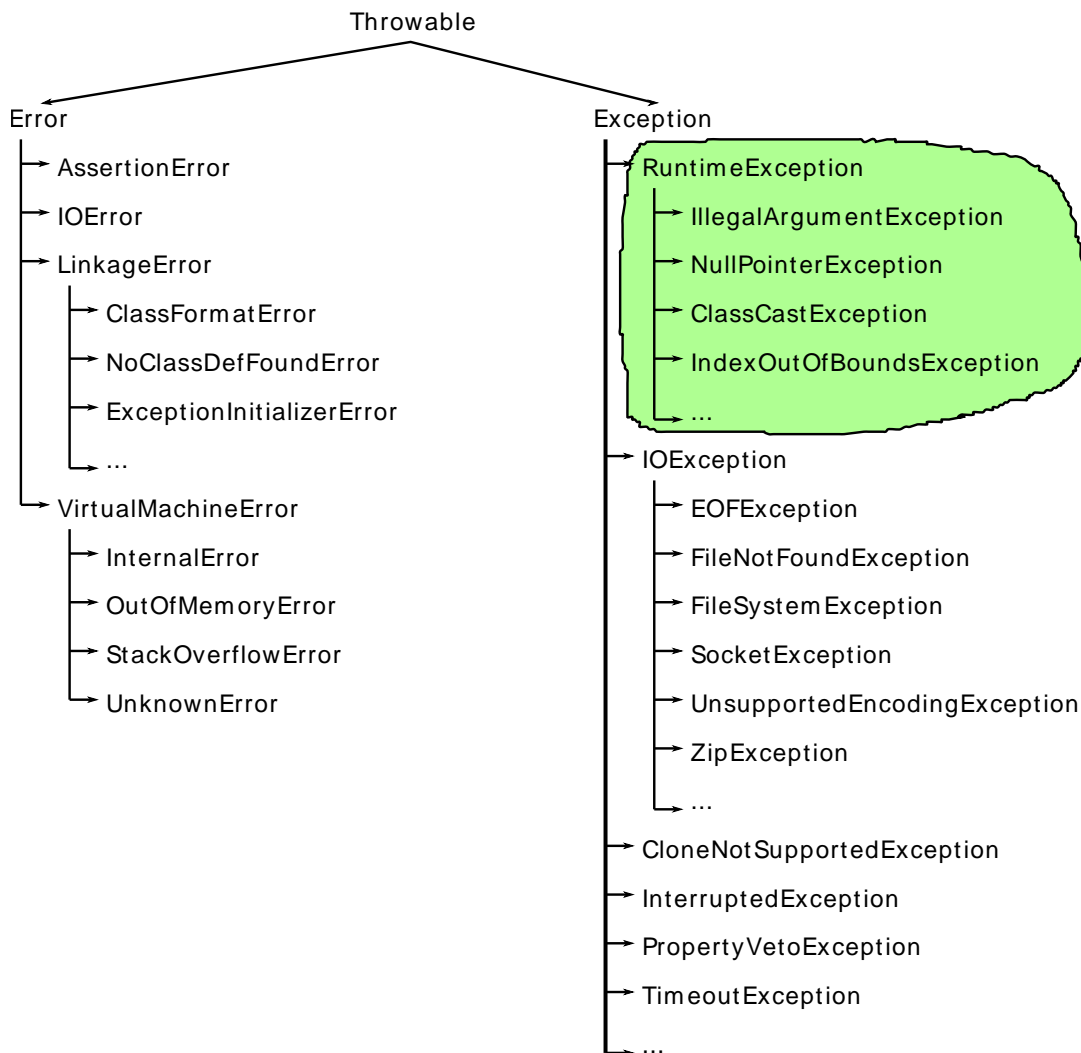
Primjer izvođenja ovako korigiranog koda dan je u nastavku.

```

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 proslijeđuje 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-catch`. Međutim, u našem primjeru, to nije opcija jer u toj metodi ne znamo što bismo smisljeno 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 }
```


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 uvjerali 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 posljednicu 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 međuspremnici 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 iznimka 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 `finally`. 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 datotekama, bazama podataka, mrežnim

aplikacijama i slično), počev od Java 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 razredima 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 obliha 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 Javae 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 neproveravane 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 nastaviti ćemo upoznavanje s razredima i njima srodnim konceptima. Razmatranje ćemo započeti pojmom apstraktnog razreda nakon čega 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 pitano 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 apstraktnih metoda 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 višestrukome nasljeđivanju mogu javiti vrlo nezgodni problemi i da jezik mora ponuditi odgovarajuće mehanizme za njihovo razrješavanje. 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 }
```

```
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` zahtijevaju 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 zahtijevane 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 implementacije 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     }
40 }
```

```

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
26 package hr.fer.zemris.java.tecaj_1;
27
28 public class Pravokutnik extends GeometrijskiLik {
29
30     private int x;
31     private int y;
32     private int sirina;
33     private int visina;
34
35     public Pravokutnik(String ime, int x, int y,
36         int sirina, int visina) {
37         super(ime);
38         this.x = x;
39         this.y = y;
40     }
41
42 }

```

```
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
51 package hr.fer.zemris.java.tecaj_1;
52
53 public class Krug extends GeometrijskiLik {
54     private int cx;
55     private int cy;
56     private int r;
57
58     public Krug(String ime, int cx, int cy, int r) {
59         super(ime);
60         this.cx = cx;
61         this.cy = cy;
62         this.r = r;
63     }
64
65     public int getCx() {
66         return cx;
67     }
68
69     public int getCy() {
70         return cy;
71     }
72 }
```

```
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.

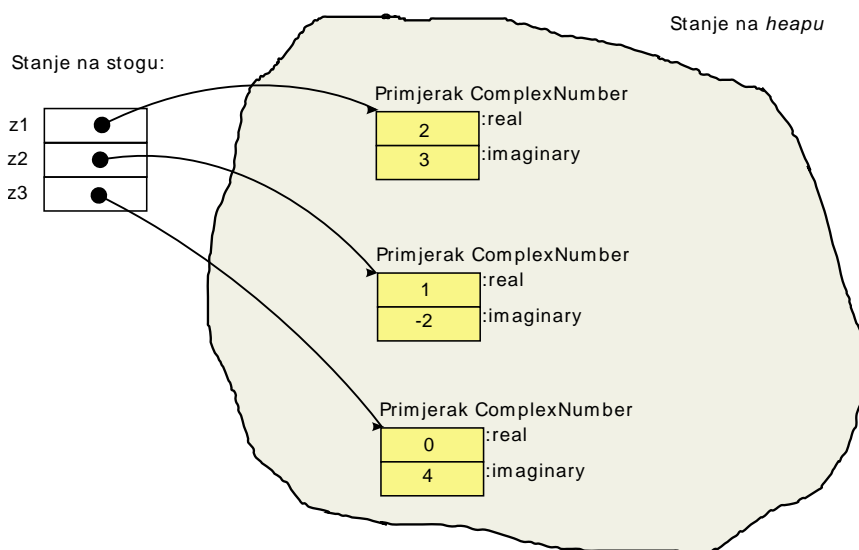
```

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 varijable¹: stoga je za svaki primjerak zauzeto dovoljno mjesta za pohranu varijable `real` i `imaginary`.

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
40 package hr.fer.zemris.java.primjeri;
41
42 public class Primjer3 {
43
44     public static void main(String[] args) {
45         ComplexNumber z1 = new ComplexNumber(2, 3);
46         ComplexNumber z2 = new ComplexNumber(1, -2);

```

¹ pažnja, ovo nije u cijelosti točno što ćemo naučiti vrlo brzo -- pravimo se za sada da je to tako

```

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.

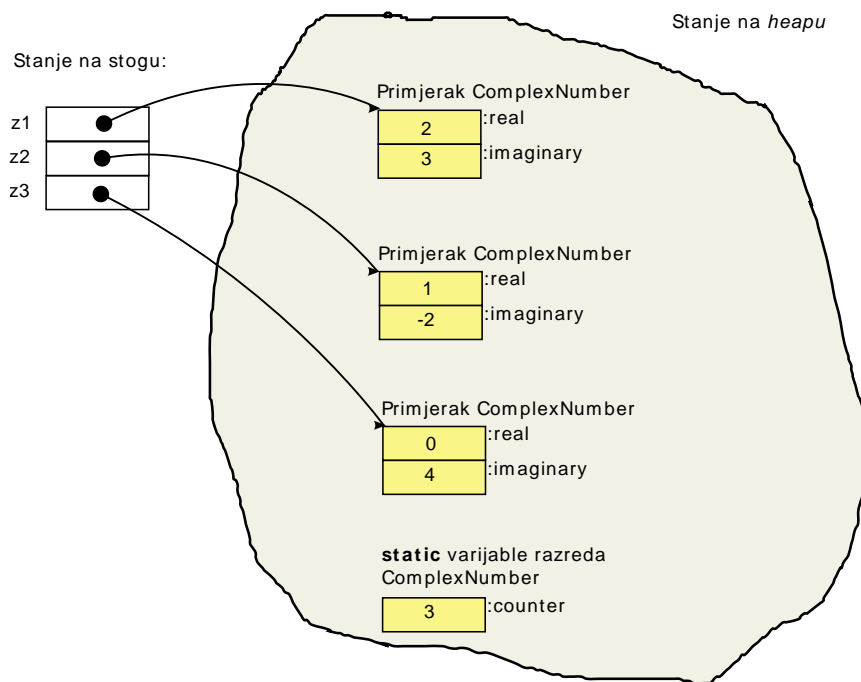
```

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. Statičke 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` promijenite 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 isključivo 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.pow(ComplexNumber.this.imaginary, 2)
58         );
59     }
60
61     public double getAngle() {
62         return Math.atan2(imaginary, real);
63     }
64 }
65 }
66
67
68 1 package hr.fer.zemris.java.primjeri;
69 2
70 3 import hr.fer.zemris.java.primjeri.ComplexNumber.
71 4     ComplexNumberInfo;
72 5
73 6 public class Primjer3 {
74 7
75 8     public static void main(String[] args) {
76 9         ComplexNumber z1 = new ComplexNumber(2, 3);
77 10        ComplexNumberInfo z1info = z1.getInfo();
78 11        ComplexNumber z2 = new ComplexNumber(1, -2);
79 12        ComplexNumberInfo z2info = z2.getInfo();
80 13        ComplexNumber z3 = new ComplexNumber(0, 4);
81 14        ComplexNumberInfo z3info = z3.new ComplexNumberInfo();
82 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
```

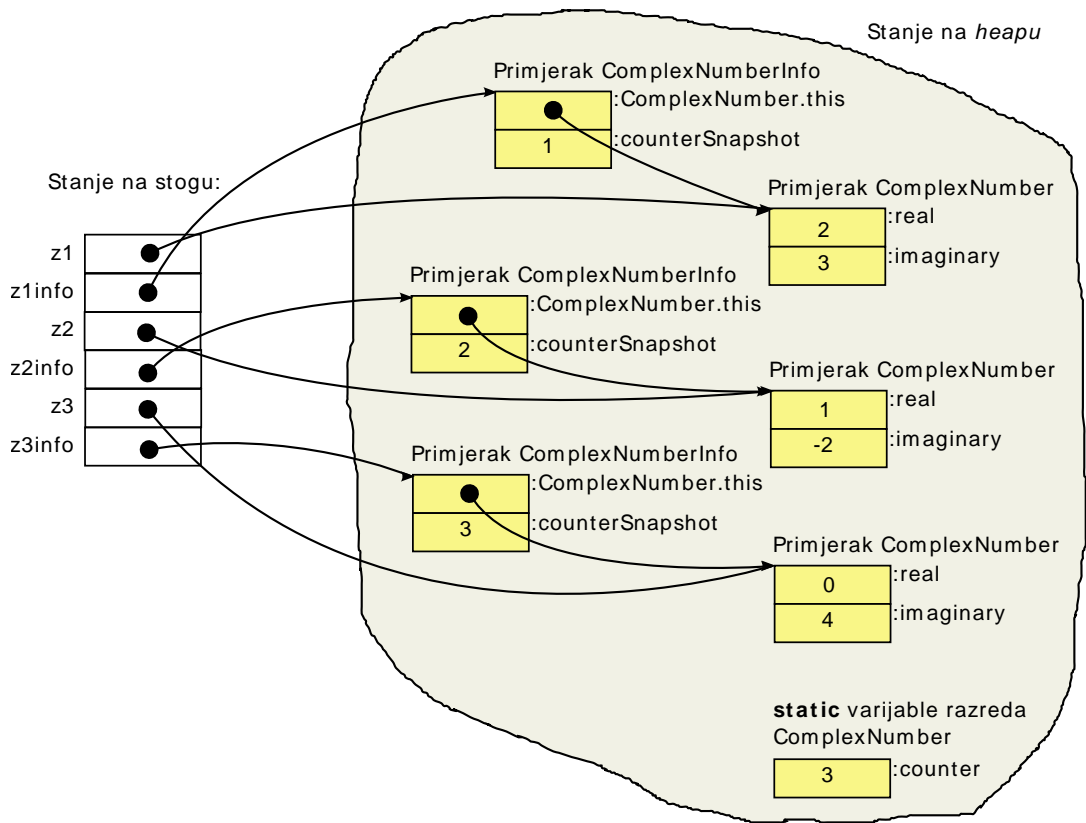
Pogledajte najprije kod razreda `ComplexNumber`. U retcima 43 do 61 dana je definicija novog razreda: razreda `ComplexNumberInfo`. Napišemo li unutar definicije jednog razreda definiciju nekog drugog razreda i to bez uporabe ključne riječi `static`, kažemo da smo napisali definiciju *unutarnjeg razreda*; 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, nužno 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 deklarirali 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
9
10 package hr.fer.zemris.java.primjeri;
11
12 public class Ispisivac {
13
14     public static void ispisi(IISPisivo ispisivo) {
15         System.out.println(ispisivo.poruka());
16     }
17 }
18
19 }
```


10

```
1 package hr.fer.zemris.java.primjeri;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 public class DatumIVrijeme implements IISPisivo {
7
8     private String format;
9
10    public DatumIVrijeme(String format) {
11        super();
12        this.format = format;
13    }
14
15    @Override
16    public String poruka() {
17        return new SimpleDateFormat(format).format(new Date());
18    }
19 }
20
21 package hr.fer.zemris.java.primjeri;
22
23 public class Primjer5 {
24
25     public static void main(String[] args) {
26
27         Ispisivac.ispisi(new Poruka("Pozdrav!"));
28
29         Ispisivac.ispisi(new IISPisivo() {
30             @Override
31             public String poruka() {
32                 return "Broj dostupnih procesora: " +
33                     Runtime.getRuntime().availableProcessors();
34             }
35         });
36
37         Ispisivac.ispisi(new DatumIVrijeme("yyyy-MM-dd HH:mm:ss"));
38
39         Ispisivac.ispisi(new DatumIVrijeme("yyyy-MM-dd HH:mm:ss") {
40             @Override
41             public String poruka() {
42                 return super.poruka() + ", dretva: " +
43                     Thread.currentThread().getName();
44             }
45         });
46
47     }
48
49     private static class Poruka implements IISPisivo {
50         private String poruka;
51
52         public Poruka(String poruka) {
53             super();
54             this.poruka = poruka;
55         }
56     }
57 }
```

```
36  @Override
37  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 najbliži 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 zahtijeva 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 RazredIliSučelje(argumenti) { definicija metoda }`. Pri tome, ako je `RazredIliSučelje` naziv sučelja, definira se anonimni razred koji implementira to sučelje a ako je `RazredIliSučelje` 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 konačnim -- 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 implementirati) 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 obišeni svi elementi. Metoda `next` zadužena je za dohvat i vraćanje sljedećeg još neobišenog 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 dohvaćao

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:

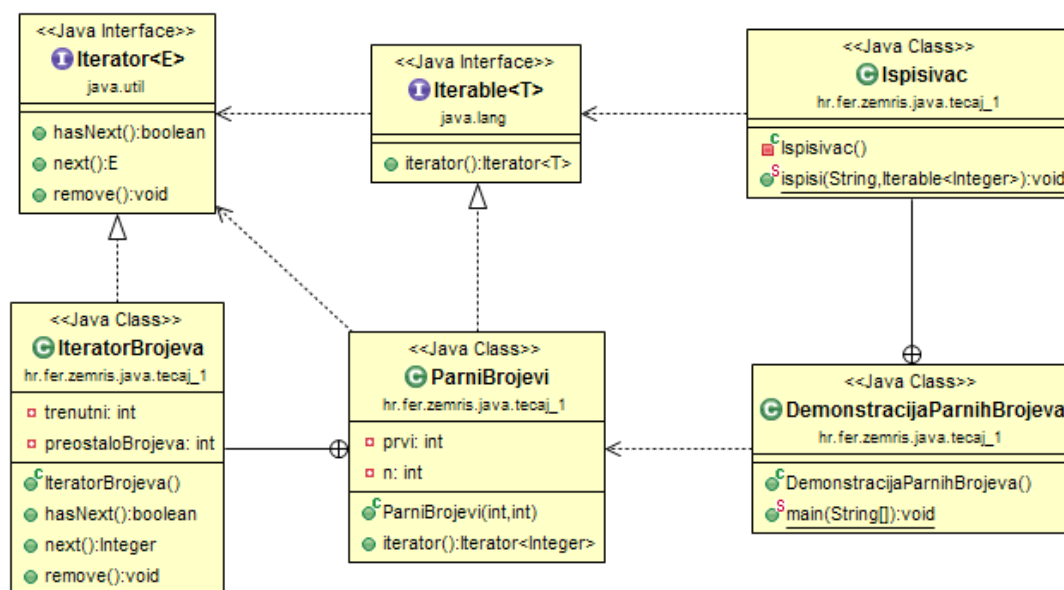
```

-----
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 pristupiti 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 datoteke, 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 nudi odgovarajućih iteratora. Taj slučaj prikazan je i u prethodnom primjeru gdje je implementacija iteratora skrivena unutar razreda skupnog objekta.

Oblikovni obrazac Iterator

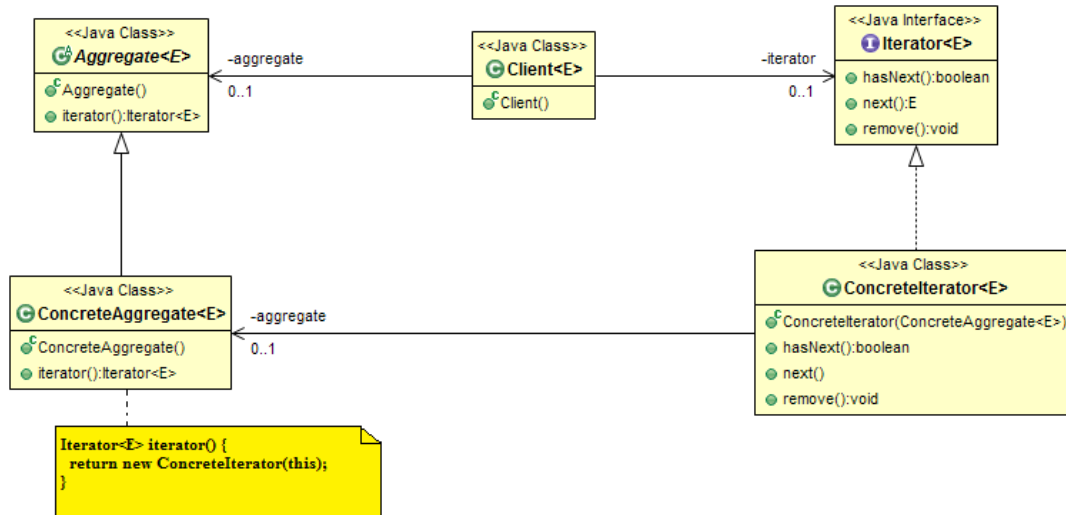
Skupni objekti izuzetno su česti u programskim sustavima a obilazak njihovih elemenata jedan je od najčešćih zadataka. Oblikovni obrazac *Iterator* naputak je koji opisuje dobar način organizacije koda koji će osigurati izradu skupnih objekata koji s kojima će moći raditi proizvoljni klijenti, a bez 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 isključivo 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).

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 modificira. 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 prekrojanju koda (engl. *code refactoring*). Zadaća prekrojanja 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 sljedeća. 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. Sljeđenjem 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*.

¹<http://www.codinghorror.com/blog/2007/03/curlys-law-do-one-thing.html>

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 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 izdvojiti 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 Substitution 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 zahtijevati 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-orientiranog 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 ovisе 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 funkcionalnost 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 nudaenje 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` ovisе 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 zadataka 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 redoslijedom 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 IspisIzvestaja {
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 }
10
11 package hr.fer.zemris.principles.srp;
12
13 public class PassedCourseReporter {
14
15     public String generateForStudent(Student student) {
16         ...
17         return ...;
18     }
19 }
20
21 package hr.fer.zemris.principles.srp;
22
23 public class StudentRepository {
24
25     public void save(Student student) {
26         ...
27     }
28 }
```

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: *Kružnica* i *Pravokutnik*. Razred *Povrsina* 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
21 package hr.fer.zemris.principles.ocp;
22
23 public class Kruznica {
24
25     private double r;
26
27     public Kruznica(double r) {
28         super();
29         this.r = r;
30     }
31
32     public double getR() {
33         return r;
34     }
35 }

```

```
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 prepoznamo. 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 Kruznicica) {
            Kruznicica k = (Kruznicica)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
9
10 package hr.fer.zemris.principles.ocp;
11
12 public class Povrsina {
13
14     public static double izracunajPovrsinu(Lik[] likovi) {
15         double suma = 0.0;
16         for(Lik lik : likovi) {
17             suma += lik.povrsina();
18         }
19         return suma;
20     }
21 }
22
23
24 package hr.fer.zemris.principles.ocp;
25
26 public class Kruznica implements Lik {
27
28     private double r;
29
30     public Kruznica(double r) {
31         super();
32         this.r = r;
33     }
34
35     public double getR() {
36         return r;
37     }
38 }
```



```
15
16 @Override
17 public double površina() {
18     return r * r * Math.PI;
19 }
20
21 }
22
23 package hr.fer.zemris.principles.ocp;
24
25 public class Pravokutnik implements Lik {
26
27     private double a;
28     private double b;
29
30     public Pravokutnik(double a, double b) {
31         super();
32         this.a = a;
33         this.b = b;
34     }
35
36     public double getA() {
37         return a;
38     }
39
40     public double getB() {
41         return b;
42     }
43
44     @Override
45     public double površina() {
46         return a * b;
47     }
48 }
```

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
9 package hr.fer.zemris.principles.ocp;
10
11 public class Logger {
```

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
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
```

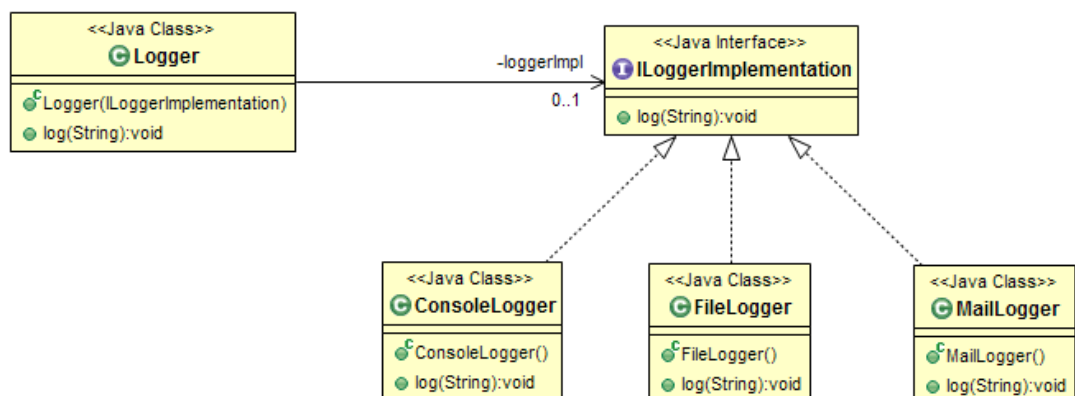
```

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
12
13 package hr.fer.zemris.principles.ocp;
14
15 public class Logger {
16
17     private ILoggerImplementation loggerImpl;
18
19     public Logger(ILoggerImplementation loggerImpl) {
20         super();
21         this.loggerImpl = loggerImpl;
22     }
23
24     public void log(String message) {
25         loggerImpl.log(message);
26     }
27 }

```

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 dnevnčkog 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 poruke istovremeno šalje i putem e-maila, i koje zapisuje u datoteku. Kako bi izgledala takva implementacija? Takvo ponašanje moguće je dobiti oslanjajući se na postojeće implementacije! Evo primjera u nastavku.

Primjer 8.10. Višestruko zapisivanje dnevnikaških 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. Konkretni 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 nekonzistentno 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. Popravljen 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 }
```

```

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         Klient.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 `Kružnica` 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 potom nudili samo metode za dohvrat 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
21
22 package hr.fer.zemris.principles.liskov;
23
24 import java.util.Map;
25
26 public class Messages {
27
28     // Mapa: ključ => poruka
29     private Map<String, Message> messages;
30
31     public Messages() {
32         // inicijaliziraj poruke
33     }
34
35     public void updateMessages() {
36         for (Message m : messages.values()) {
37             m.update();
38         }
39     }
40 }
41
42 }
```

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 `Message` 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 obećanjem 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) {
```

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 argumenta 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 preduvjet 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 skrsiti 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 skrsiti 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 izdvajanja 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 daje 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
10 package hr.fer.zemris.principles.srp;
11
12 public interface IDatabase {
13     public int executeQuery(Query query);
14     public void updateIndexes();
15     public void compactUnusedSpace();
16     public double calcFragmentationRatio();
17 }
18
19 package hr.fer.zemris.principles.srp;
20
21 public class ConcreteDatabase implements IDatabase {
22
23     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) {
```



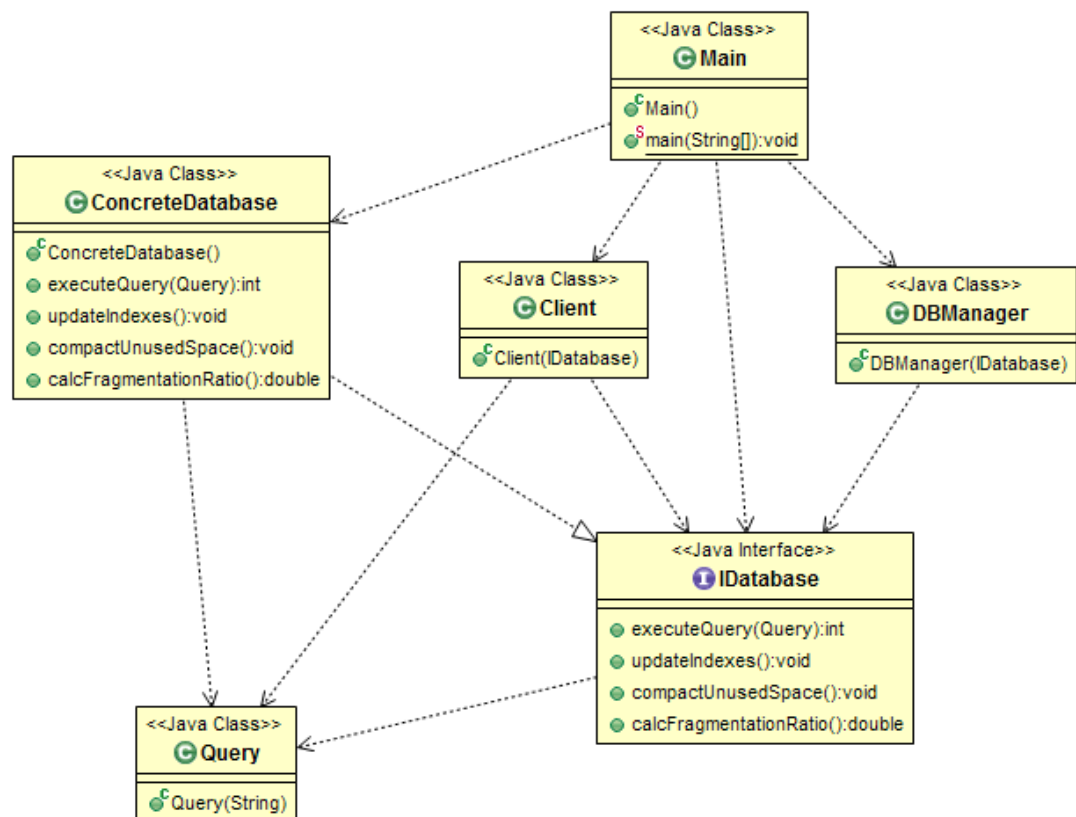
```

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 popravljavanje 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 ovisi 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
10
11 package hr.fer.zemris.principles.srp;
12
13 public interface IQueryable {
14     public int executeQuery(Query query);
15 }
16
17
18 package hr.fer.zemris.principles.srp;
19
20 public interface IManageable {
21     public void updateIndexes();
22     public void compactUnusedSpace();
23     public double calcFragmentationRatio();
24 }
25
26
27 package hr.fer.zemris.principles.srp;
28
29 public class ConcreteDatabase implements IQueryable, IManageable {
30
31     public ConcreteDatabase() {
32     }
33
34     @Override
35     public int executeQuery(Query query) {
36         // izvrši upit
37         return ... // vrati rezultat
38     }
39
40     @Override
41     public void updateIndexes() {
42         // ...
43     }
44
45     @Override
46     public void compactUnusedSpace() {
47         // ...
48     }
49
50     @Override
51     public double calcFragmentationRatio() {
52         // ...
53         return ... // vrati rezultat
54     }
55 }
56
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public class Client {
4
5     public Client(IQueryable 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
```

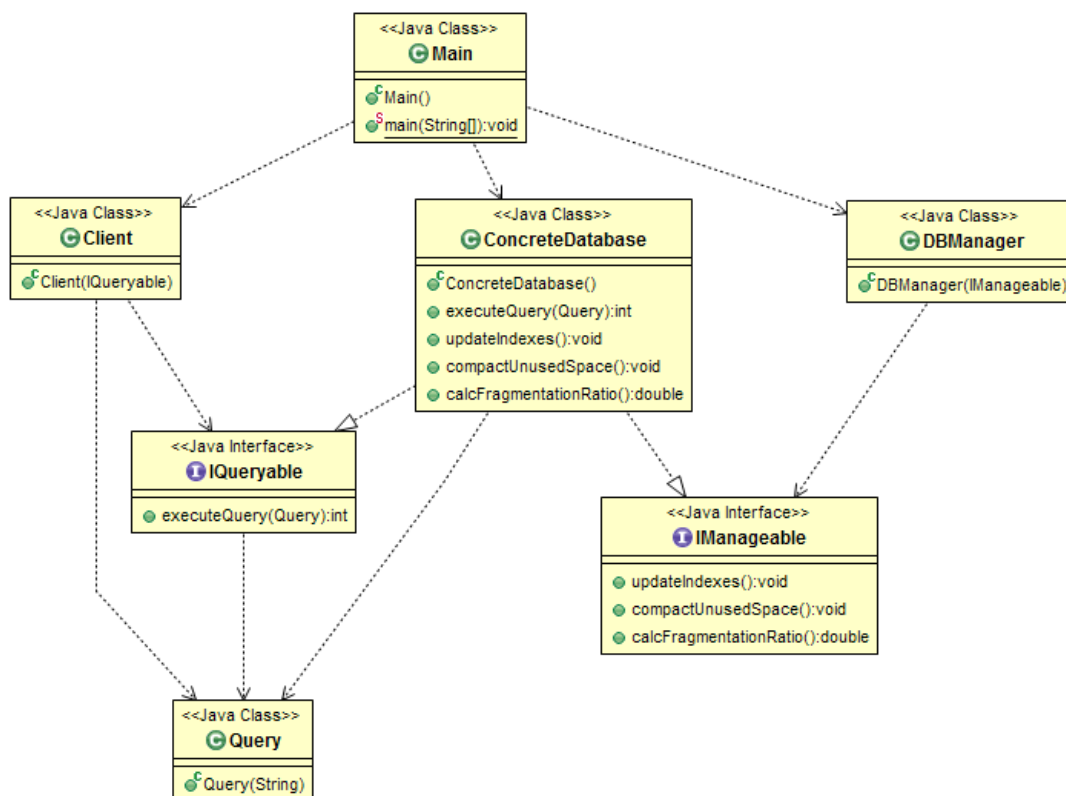
```
1 package hr.fer.zemris.principles.srp;
2
3 public class DBManager {
4
5     public DBManager(IManegeable database) {
6         if(database.calcFragmentationRatio() > 0.4) {
7             database.compactUnusedSpace();
8             database.updateIndexes();
9         }
10    }
11
12 }
13
```

```
1 package hr.fer.zemris.principles.srp;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         ConcreteDatabase db = new ConcreteDatabase();
7
8         new Client(db);
9         new DBManager(db);
10    }
11
12 }
13
```

Umjesto jednog sučelje, sada smo definirali dva sučelja. Sučelje `IQueryable` klijentima omogućava postavljanje upita nad bazom podataka. Sučelje `IManegeable` 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 proslijeđuje 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
22
23
24
25
26
27
28
29
30
31 package hr.fer.zemris.principles.dip;
32
33 /**
34  * Podsustav koji provjerava valjanost provedenih
35  * transakcija i po potrebi bilježi problematične
36  * transakcije.
37  *
38  */
39 public class RecordVerifier {
40
41     private DatabaseInformer dbinf;
42
43     public RecordVerifier() {
44         dbinf = new DatabaseInformer();
45     }
46
47     public void verifyRecords(Record[] records) {
48         for(Record record : records) {
49             boolean valid = true;
50             // Napravi niz provjera nad zapisom i ako
51             // se pronađu problemi, postavi zastavicu
52             // valid na false.
53             if(!valid) {
54                 dbinf.issueWarning(record);
55             }
56         }
57     }
58 }
59
60
61
62
63 package hr.fer.zemris.principles.dip;
64
65 /**
66  * Podsustav koji u bazu podataka zapisuje podatke
```

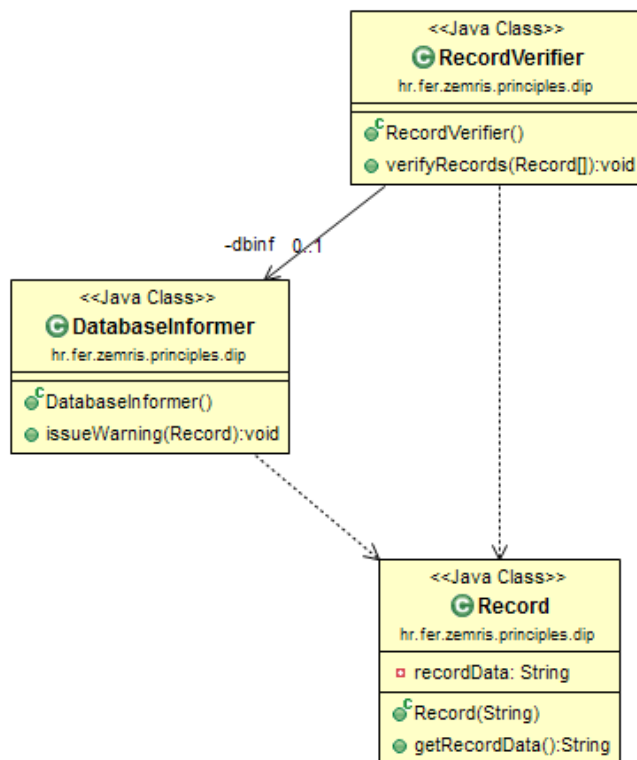
```

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 zaporka 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 *setter*a 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 *setter*a 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

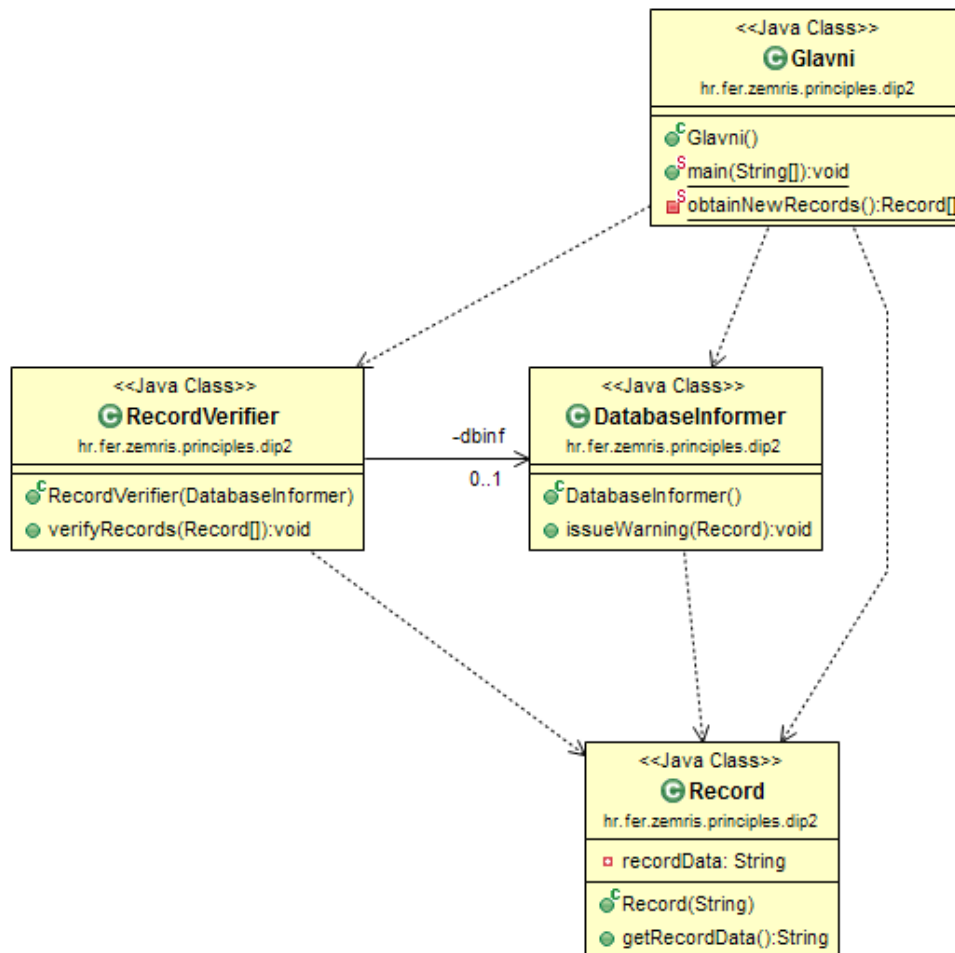
1 package hr.fer.zemris.principles.dip2;
2
3 /**
4  * Podsustav koji u bazu podataka zapisuje podatke
5  * problematičnim transakcijama.
6  *
7  */
8 public class DatabaseInformer {
9
10     public DatabaseInformer() {
11     }
12
13     public void issueWarning(Record record) {
14         // Zapiši u bazu podataka podatke o problematičnom
```



```
15  // zapisu kako bi ih kasnije odgovorna osoba
16  // provjerila.
17  }
18
19  }
20
21
22  1 package hr.fer.zemris.principles.dip2;
23  2
24  3 public class Glavni {
25  4
26  5     public static void main(String[] args) {
27  6
28  7         DatabaseInformer dbinf = new DatabaseInformer();
29  8
30  9         // Injekcija ovisnosti:
31 10        RecordVerifier recVer = new RecordVerifier(dbinf);
32 11
33 12        while(true) {
34 13            Record[] records = obtainNewRecords();
35 14            recVer.verifyRecords(records);
36 15        }
37 16
38 17    }
39 18
40 19    private static Record[] obtainNewRecords() {
41 20        // Aplikacijska logika koja dohvaća još neprovjerene
42 21        // zapise koji su u međuvremenu nastali i vraća ih
43 22        return null; // novi neprovjereni zapisi
44 23    }
45 24
46 25 }
47 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, proslijeđuje 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
22 package hr.fer.zemris.principles.dip3;
23
24 /**
25  * Sučelje koje opisuje proizvolju implementaciju
26  * razreda koji zna poslati upozorenje o problematičnom
27  * zapisu.
28  *
29  */
30 public interface IInformer {
31
32     public void issueWarning(Record record);
33 }
34
35 package hr.fer.zemris.principles.dip3;
36
37 /**
38  * Podsustav koji provjerava valjanost provedenih
39  * transakcija i po potrebi bilježi problematične
40  * transakcije.
41  *
42  */
```

```
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 pronadu problemi, postavi zastavicu
22             // valid na false.
23             if(!valid) {
24                 inf.issueWarning(record);
25             }
26         }
27     }
28 }
29
30
31 package hr.fer.zemris.principles.dip3;
32
33 /**
34  * Podsustav koji u bazu podataka zapisuje podatke o
35  * problematičnim transakcijama.
36  *
37  */
38 public class DatabaseInformer implements IInformer {
39
40     public DatabaseInformer() {
41     }
42
43     public void issueWarning(Record record) {
44         // Zapiši u bazu podataka podatke o problematičnom
45         // zapisu kako bi ih kasnije odgovorna osoba
46         // provjerila.
47     }
48 }
49
50
51 package hr.fer.zemris.principles.dip3;
52
53 /**
54  * Podsustav koji elektroničkom poštom šalje podatke
55  * problematičnim transakcijama.
56  *
57  */
58 public class EMailInformer implements IInformer {
59
60     public EMailInformer() {
61     }
62
63     public void issueWarning(Record record) {
64         // Pošalji e-mail o problematičnom zapisu
65         // kako bi ga kasnije odgovorna osoba
```

```
16  // provjerila.
17  }
18
19  }
20
21  package hr.fer.zemris.principles.dip3;
22
23  /**
24   * Implementacija podsustava koji sam po sebi ne zna pohranjivati
25   * podatke o problematičnim zapisima već taj zadatak može delegirati
26   * drugim podsustavima.
27   */
28  public class CompositeInformer implements IInformer {
29
30      private IInformer[] informers;
31
32      public CompositeInformer(IInformer[] informers) {
33          // Zapamti kopiju predanih podataka
34          this.informers = new IInformer[informers.length];
35          for(int i = 0; i < informers.length; i++) {
36              this.informers[i] = informers[i];
37          }
38      }
39
40      @Override
41      public void issueWarning(Record record) {
42          for(IInformer inf : informers) {
43              inf.issueWarning(record);
44          }
45      }
46  }
47
48  package hr.fer.zemris.principles.dip3;
49
50  public class Glavni {
51
52      public static void main(String[] args) {
53
54          // Kao implementaciju biramo kompozitni objekt
55          // koji će obavljati delegiranje na dvije konkretne
56          // implementacije.
57          IInformer inf = new CompositeInformer(
58              new IInformer[] {
59                  new DatabaseInformer(),
60                  new EMailInformer()
61              }
62          );
63
64          RecordVerifier recVer = new RecordVerifier(inf);
65
66          while(true) {
67              Record[] records = obtainNewRecords();
68              recVer.verifyRecords(records);
69          }
70      }
71  }
```

```
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.

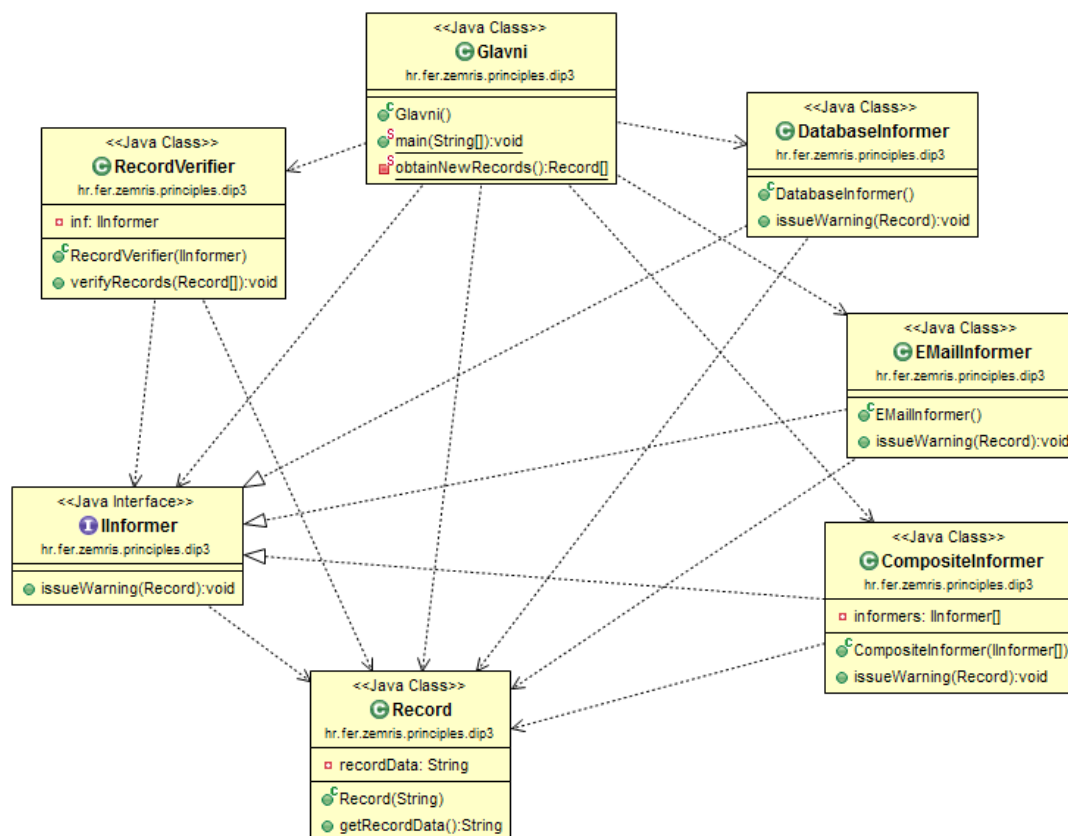
Razred `DatabaseInformer` u ovom je rješenju prilagođen tako da implementira sučelje `IInformer` kako bi se njegovi primjerci mogli slati kroz konstruktor razreda `RecordVerifier`.

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. Kolekcije

- definicija kolekcije
- primjer definicije preko razreda
- problem s tim pristupom
- primjer definicije preko sučelja
- različite implementacije tog sučelja
- Javin okvir kolekcija
 - od čega se sastoji (sučelja, implementacije, algoritmi)
 - prednosti uporabe
 - iteriranje kroz kolekcije
- definirana sučelja
- prikaz sučelja
- građa sučelja: osnovne operacije, grupne operacije, potpora poljima
- komentar svake od metoda
- nepodržane metode i UnsupportedOperationException
- zašto uopće nepodržane metode? (npr. read-only kolekcije)

Poglavlje 10. Kolekcija: skup

- kako se Set razlikuje od Collection
- koje su podržane implementacije
- kratki opis svake od implementacija
- složenosti pojedinih operacija
- primjer: ispis argumenata komandne linije ali bez duplikata
- programiranje prema sučeljima a ne prema implementacijama
- kratak osvrt na tehnologiju Java Generics

Poglavlje 11. Kolekcija: lista

- kako se razlikuje od Collection
- koje su podržane implementacije
- kratki opis svake od implementacija
- složenosti
- primjer: program koji ispisuje argumente obrnutim poretkom i bez duplikata
- za vježbu: program koji čita brojeve s tipkovnice i ispisuje one koji su barem 20% veći od prosjeka, sortirano od manjih prema većim

Poglavlje 12. Kolekcija: mapa

- asocijativna polja, riječnici (Dictionary), mape
- sučelje Map
- interne kolekcije: keySet, values, entrySet i komentar vrste tih kolekcija
- komentar pojedinih metoda
- kratki opis svake od implementacija
- Pomoćni razredi Collections i Arrays: različite metode poput sort i slično

Poglavlje 13. Kolekcije i novi razredi

- novi razredi i liste
 - važnost metode equals()
- novi razredi i TreeSet
 - važnost sučelja Comparable, prirodni poredak
 - primjer s TreeSet-om
 - važnost sučelja Comparator
 - primjer s TreeSet-om
 - anonimni razredi i implementacije komparatora
 - inverzni komparatori
 - komparatori i generičke metode (reverseOrder)
 - oblikovni obrazac Dekorator
 - kompozitni komparatori
- novi razredi i HashMap
 - važnost metode hashCode()
 - delegiranje na hashCode() jednostavnijih tipova

Poglavlje 14. Tehnologija Java Generics

- zašto uporaba tehnologije Java Generics
- primjer generičkog pointera
- primjer generičkog para koji vraća max i min
- ograničevanje s gornje i donje strane (X extends T, X super T)

Poglavlje 15. 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 15.1. 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 15.2. 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 getAbsolutePath();`

`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();`

Objekti 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);`

Objekti 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 15.3. 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 15.4. 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 rekurzivnim 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 15.5. 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 agrsm64.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 prihvata 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 15.6. 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 dohvata 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 učitava 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 Java 7 ona je dostupna i korisnicima Java.

Primjer 15.7. 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 izdvojiti 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`.

Konkretni primjer prikazan je u nastavku.

Primjer 15.8. 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 15.9. 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[] sadržaj = {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, sadržaj);
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 koda 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 15.10. 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             "Prosječna 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 običenih 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 15.11. 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 pronađen 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 15.12. 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      "Prosječna velicina datoteka je: " +
43      info.prosjeckDatoteka());
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.prosjeck());
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 pamtim
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 pamtim
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 razred 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 16. Tokovi okteta i znakova

- apstrakcija binarnih tokova: `InputStream` i `OutputStream`
- apstrakcija znakovnih tokova: `Reader` i `Writer`
- primjer binarnog toka: `FileInputStream` i `FileOutputStream`
- ideja Decorator design patterna
- dekoriranje tokova
 - `BufferedInputStream`, `BufferedOutputStream`
 - `DataInputStream`, `DataOutputStream`
 - `PushbackInputStream`
- primjer znakovnih tokova
 - `StringReader`, `StringWriter`
 - `FileReader`, `FileWriter`
- primjena oblikovnog obrasca okvirne metode (Template method)
- primjena oblikovnog obrasca strategije (Strategy)
- kodne stranice i znakovi
 - "Šeće šumom Čevapčić s džemom" zapisan kao Windows-1250, ISO8859-2, UTF-8
 - pretvorba binarnog niza u znakovni: `new String(byte[], charsetName)`
 - `InputStreamReader`, `OutputStreamWriter`

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 uočiti da 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 koja je tekstovna i koja sadrži nalog kako nacrtati sliku: konceptualno, informacija koju ona predstavlja je slika.

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. 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 uobičajene 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 redoslijedom.

Tokovi okteta

Rad sa slijednim binarnim datotekama apstrahiran je uporabom pojma *tok okteta*. Sa strane korisnika, tok može biti *ulazni* ili *izlazni*. Primjer jednostavnog modela ulaznog i izlaznog toka prikazan je u nastavku.

```
interface UlazniTokOkteta {
    public byte procitajOktet() throws IOException;
    public void close() throws IOException;
    public void preskoci(int n) throws IOException;
}

interface IzlazniTokOkteta {
    public void zapisiOktet(byte oktet) throws IOException;
    public void close() throws IOException;
}
```

Ulazni tok modeliran je sučeljem `UlazniTokOkteta`. Ulazni tok definira samo tri metode: metoda `procitajOktet` iz toka dohvaća prvi sljedeći nepročitani oktet. Kako je ovo model slijednog pristupa, sadržaj se čita oktet po oktet, od prvog do posljednjeg raspoloživog okteta. Kada su svi okteti 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; 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 16.1. Primjer ulaznih tokova okteta.

```
1 package hr.fer.zemris.java.tokovi.okteta.ulazni;
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
12  */
13 public interface UlazniTokOkteta {
14
15     /**
16      * Metoda dohvaća sljedeći nepročitani oktet i
17      * vraća ga. Ako iz toka više nije moguće pročitati
18      * niti jedan oktet jer su svi podatci iscrpljeni,
19      * metoda izaziva iznimku {@link IOException}.
20      *
21      * @return sljedeći nepročitani oktet
22      * @throws IOException ako više nema podataka
23      */
24     public byte procitajOktet() throws IOException;
25
26     /**
27      * Metoda zatvara trenutni tok i otpušta sve njegove
28      * resurse. Tok koji je zatvoren ne može se opet
29      * zatvoriti (u tom slučaju bit će izazvana iznimka
30      * {@link IOException}).
31      *
32      * @throws IOException ako je tok već zatvoren
33      */
34     public void close() throws IOException;
35
36     /**
37      * Metoda preskače zadan broj okteta. Efekt je isti
38      * kao da je n puta pozvana metoda
39      * {@link #procitajOktet()} čiji se rezultati potom
40      * zanemare.
41      *
42      * @param n željeni broj okteta koje treba preskočiti
43      * @throws IOException ako nastupi pogreška prilikom
44      *       preskakanja
45      * @throws IllegalArgumentException ako je n
46      *       negativan
47      */
48     public void preskoci(int n) throws IOException;
49 }
50
51 package hr.fer.zemris.java.tokovi.okteta.ulazni;
52
```

```
3 import java.io.EOFException;
4 import java.io.IOException;
5
6 /**
7  * Ulazni tok koji čita podatke iz polja okteta koje dobiva
8  * prilikom konstrukcije. Čitanje kreće od prvog okteta
9  * u predanom polju i napreduje prema posljednjem oktetu.
10 *
11 * @author marcupic
12 */
13 public class UlazniTokOktetaIzPolja
14     implements UlazniTokOkteta {
15
16     // 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 null.
26      *
27      * @param polje polje okteta iz kojeg će se čitati podatci
28      * @throws IllegalArgumentException ako je predan
29      *         null
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(gotovo || trenutnaPozicija >= polje.length) {
45             gotovo = true;
46             throw new EOFException("Sadržaj je pročitán.");
47         }
48         byte oktet = polje[trenutnaPozicija];
49         trenutnaPozicija++;
50         return oktet;
51     }
52
53     @Override
54     public void close() throws IOException {
55         if(polje==null) {
56             throw new IOException("Tok je već zatvoren.");
57         }
58         // Otpusti resurse...
59         polje = null;
60         gotovo = true;
```

```
61 }
62
63 @Override
64 public void preskoci(int n) throws IOException {
65     for(int i = 0; i < n; i++) {
66         procitajOktet();
67     }
68 }
69 }
70

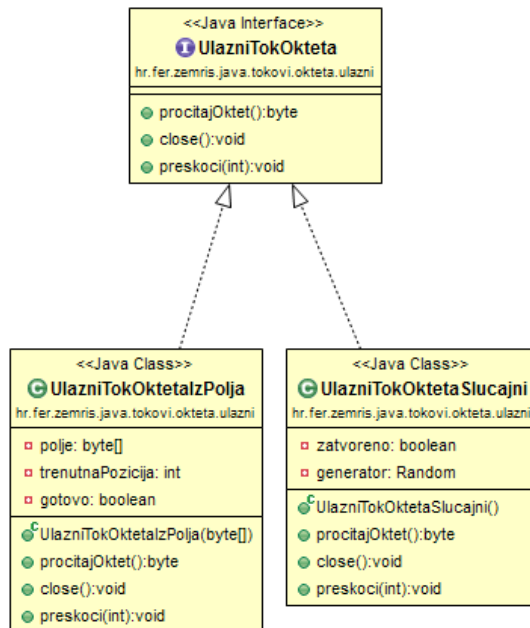
1 package hr.fer.zemris.java.tokovi.okteta.ulazni;
2
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.util.Random;
6
7 /**
8  * Implementacija ulaznog toka koji predstavlja
9  * neograničen izvor slučajno generiranih
10 * okteta.
11 *
12 * @author marcupic
13 *
14 */
15 public class UlazniTokOktetaSlucajni
16     implements UlazniTokOkteta {
17
18     // Zastavica koja govori je li tok zatvoren
19     private boolean zatvoreno;
20     // Privatni generator slučajnih brojeva
21     private Random generator;
22
23     /**
24      * Konstruktor.
25      */
26     public UlazniTokOktetaSlucajni() {
27         generator = new Random();
28     }
29
30     @Override
31     public byte procitajOktet() throws IOException {
32         if(zatvoreno) {
33             throw new EOFException(
34                 "Čitanje zatvorenog toka nije moguće.");
35         }
36         return (byte) (generator.nextInt() & 0xFF);
37     }
38
39     @Override
40     public void close() throws IOException {
41         if(zatvoreno) {
42             throw new IOException("Tok je već zatvoren.");
43         }
44         // Otpusti resurse...
45         generator = null;
46         zatvoreno = true;
47     }
48 }
```

```

48
49  @Override
50  public void preskoci(int n) throws IOException {
51      // ne treba ništa raditi; rezultate ionako
52      // generiramo slučajno.
53  }
54  }
55

```

Slika 16.1. Primjer ulaznih tokova okteta.



Nadogradnja funkcionalnosti

... dodavanje bufferiranja

Oblikovni obrazac Dekorator

...

Znakovi i okteti

* Kodne stranice

Tokovi znakova

...

Primjer uporabe

...

Oblikovni obrazac Strategija

...

Oblikovni obrazac Okvirna metoda

...

Dodatni zahtjevi

različite vrste komentara u različitim datotekama - nova strategija

Datoteke sa slučajnim pristupom

...

Poglavlje 17. Studija slučaja: programski jezik *vlang*

U okviru ovog poglavlja napraviti 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 17.1. 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;
```

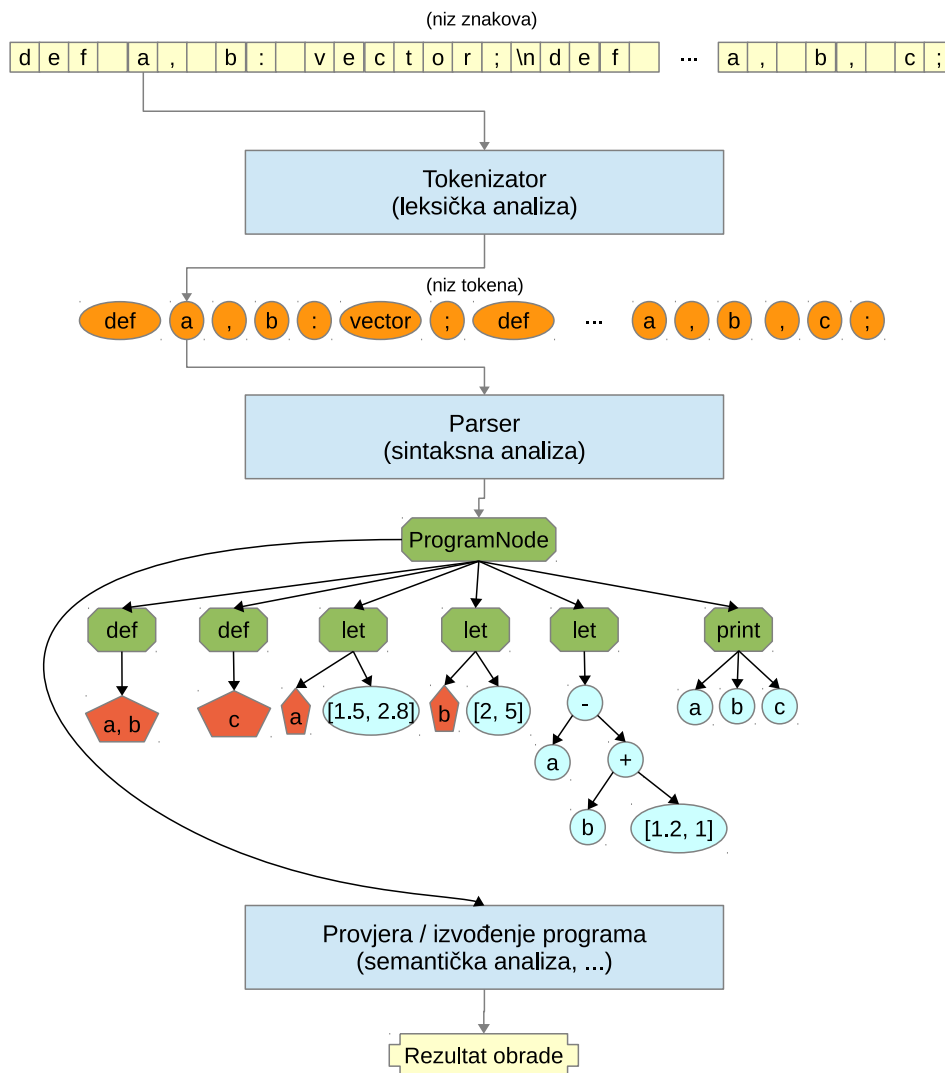
Jezik *vlang* definira tri naredbe. Naredba **def** omogućava definiranje varijabli. Naredba **let** je nadredba pridruživanja kojom se vrijednost izraza specificiranog s desne strane operatora = pridružuje varijabli čije je ime navedeno s lijeve strane operatora =. Konačno, naredba **print** omogućava 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 17.1 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 17.1. Dijagram toka obrade programa



Osnovni razredi

Da bismo započeli s izgradnjom programskog ostvarenja sustava opisanog slikom 17.1, 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 17.2 dok je sučelje tog razreda prikazano na slici 17.2.

Primjer 17.2. 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 {
  
```

```

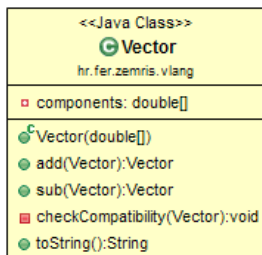
12
13  /**
14   * Komponente vektora.
15   */
16  private double[] components;
17
18  /**
19   * Konstruktor.
20   * @param components komponente vektora
21   */
22  public Vector(double...components) {
23      this.components = Arrays.copyOf(components, components.length);
24  }
25
26  /**
27   * Zbrajanje 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() {

```

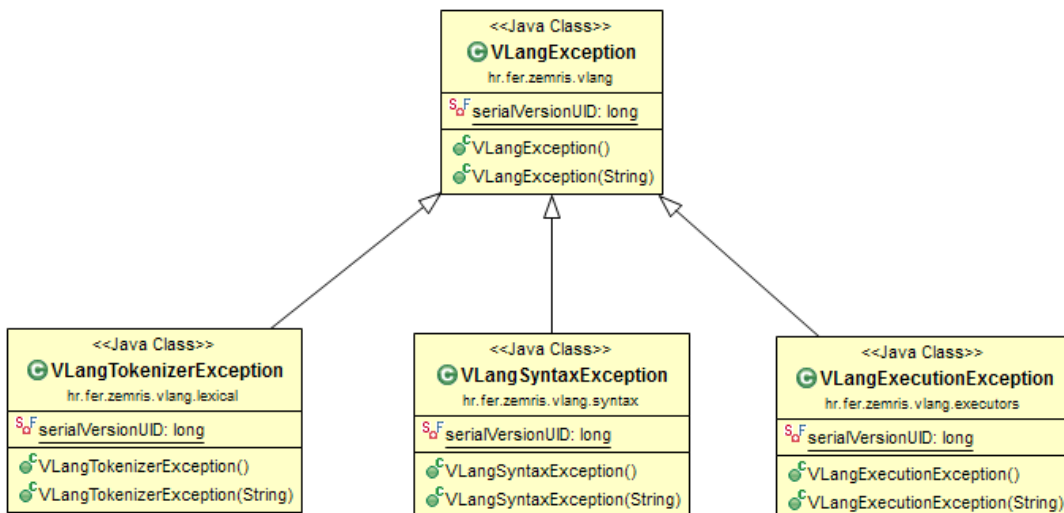
```

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 17.2. 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 17.3 prikazuje ovo podstablo.

Slika 17.3. 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 (17.3). 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 zagrada 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 17.3. 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_PARENTHESSES,
31    /** Zatvorena obla zagrada */
32    CLOSED_PARENTHESSES
33 }
34
```

Potom je potrebno definirati model jednog tokena. Tokeni će biti primjerci razreda `VLangToken`, i pri tome će pamtit i 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 17.4.

Primjer 17.4. 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 17.5. 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 17.5.

Primjer 17.5. 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_PARENTHESSES);
52 mapper.put (Character.valueOf( ')' ),
53     VLangTokenType.CLOSED_PARENTHESSES);
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]=='J') {
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 17.6.

Primjer 17.6. 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 redoslijedom, te ako jesu, izgraditi sintakšno 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, okružiti 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 prikazana gramatika koju ćemo koristiti.

Primjer 17.7. Gramatika jezika *vlang*

```

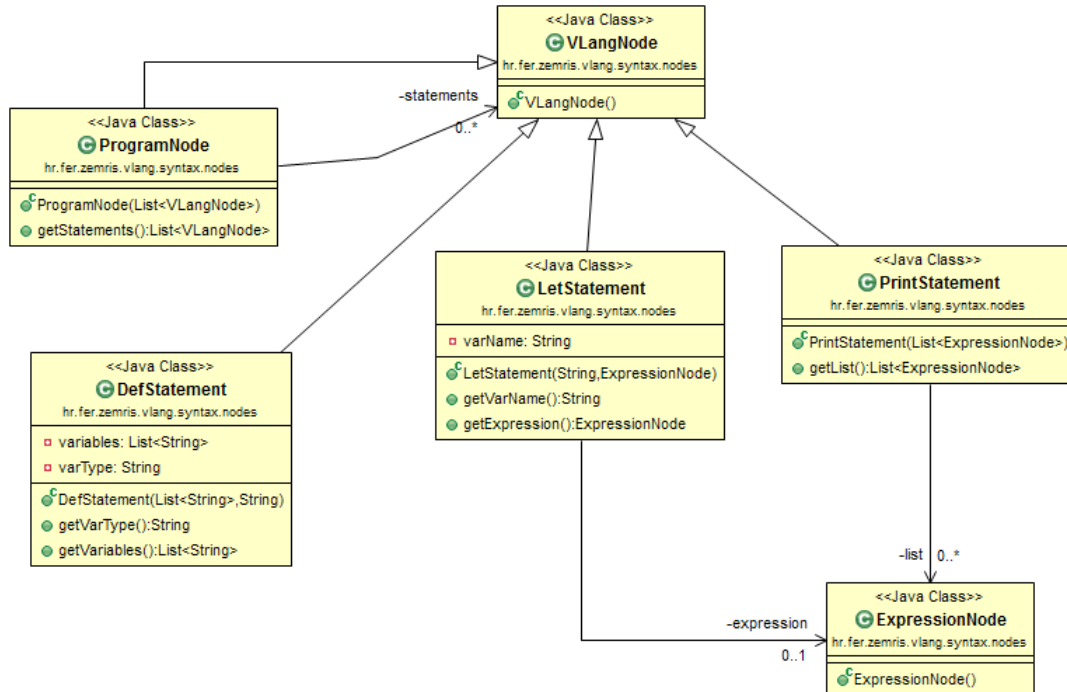
[1]   <Start> ::= { <DefStmt> | <LetStmt> | <PrintStmt> }*
[2]   <DefStmt> ::= 'def' id { ',' id }* ':' 'vector' ';'
[3]   <LetStmt> ::= 'let' id '=' <Expr> ';'
[4]   <PrintStmt> ::= 'print' <Expr> { ',' <Expr> }* ';'
[5]   <Expr> ::= <Atom> { '+' <Atom> } | '-' <Atom> }*
[6]   <Atom> ::= id | vect | '(' <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 17.4.

Slika 17.4. 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.

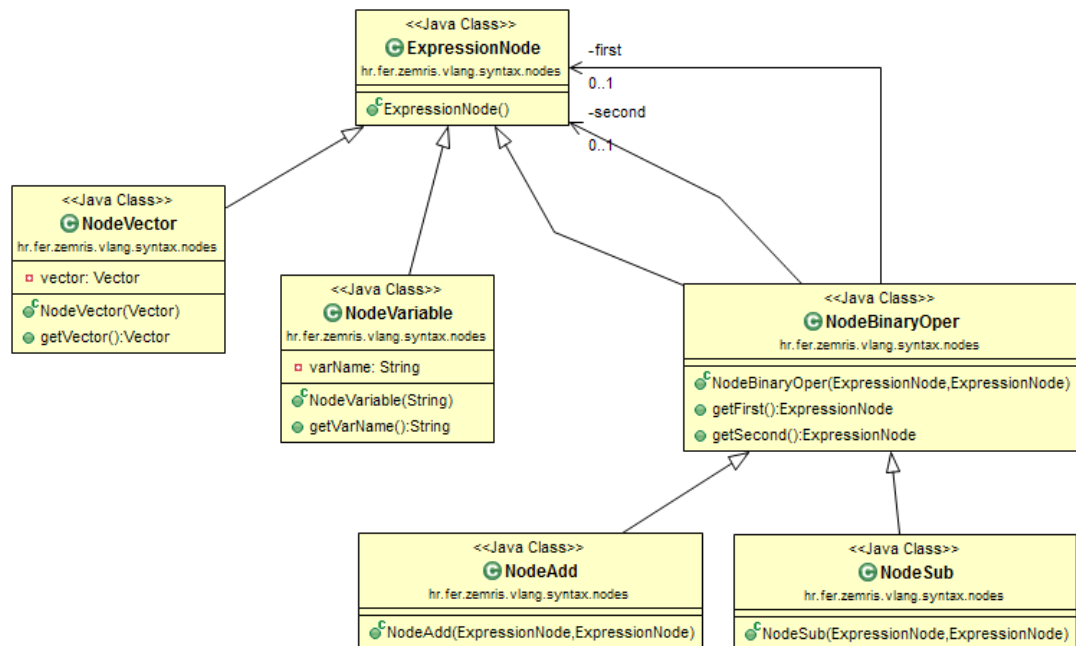
Pravilo za `<LetStmt>` opisuje strukturu naredbe **let**: naredba započinje ključnom riječi **let** nakon čega slijedi jedan identifikator, znak pridruživanja, algebarski izraz i konačno znak točka-zarez.

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 17.5.

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 17.5. Podstablo za pamćenje izraza



Razred `VLangParser` koji predstavlja parser za izgradnju sintaksnog stabla tehnikom rekursivnim spustom prikazan je izvornim kodom 17.8.

Primjer 17.8. 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> rekursivnim
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   * rekursivnim 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 obliku
213  * zagrada.
214  * @return čvor koji predstavlja ovaj izraz
215  */
216 private ExpressionNode parseAtomicValue() {
217     if (isTokenType(VLangTokenType.IDENT)) {
218         String varName =
219             (String)tokenizer.getCurrentToken().getValue();
220         tokenizer.nextToken();
221         return new NodeVariable(varName);
222     }
223     if (isTokenType(VLangTokenType.VECTOR_CONSTANT)) {
224         Vector vector =
225             (Vector)tokenizer.getCurrentToken().getValue();
226         tokenizer.nextToken();
227         return new NodeVector(vector);
228     }
229     if (isTokenType(VLangTokenType.OPEN_PARENTHESSES)) {
230         tokenizer.nextToken();
231         ExpressionNode expression = parseExpression();
232         if (!isTokenType(VLangTokenType.CLOSED_PARENTHESSES)) {
233             throw new VLangSyntaxException(
234                 "Closed parentheses was expected.");
235         }
236         tokenizer.nextToken();
237         return expression;
238     }
239     throw new VLangSyntaxException("Unexpected 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 (!isTokenType(VLangTokenType.COMMA)) {
251             break;
252         }
253         tokenizer.nextToken();
254         list.add(parseExpression());
255     }
256     if (!isTokenType(VLangTokenType.SEMICOLON)) {
257         throw new VLangSyntaxException("Semicolon was expected.");
258     }
259     tokenizer.nextToken();
260     return new PrintStatement(list);
261 }
262 }

```

Jednostavan demonstracijski program koji konstruira tokenizator i potom gradi sintakšno stablo uporabom razvijenog parsera prikazan je izvornim kodom 17.9.

Primjer 17.9. 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 sintakšno stablo, pogledat ćemo kako napisati kod koji će napisati 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 17.10.

Primjer 17.10. 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             }
66         }
67     }
68 }

```

```

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 17.11.

Primjer 17.11. 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 uvjerali da kod radi, pogledajmo malo detaljniju implementaciju prikazanu izvornim kodom 17.10. Razred `ExecutorSimple` kroz konstruktor dobiva referencu na sintakšno 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 riješene 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 prozivanju različitih vrsta izraza uporabom operatora **instanceof**.

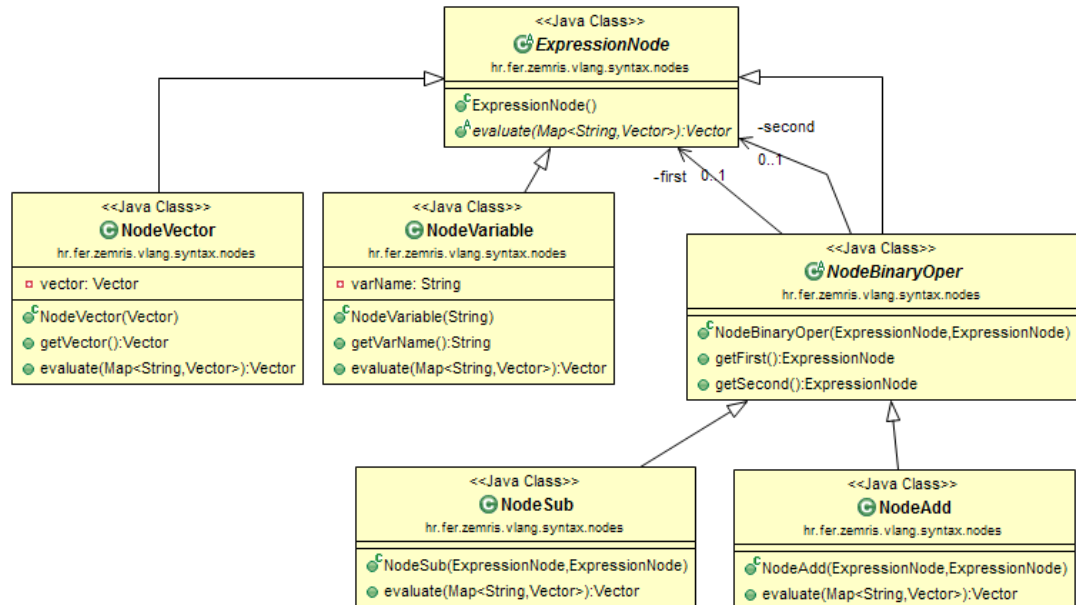
Što možemo zaključiti o napisanom rješenju? Velika prednost rješenja je da ono *radi*. Dano rješenje nažalost iskazuje svojstvo krhkosti: primjerice, želimo li dodati novi binarni operator, ne samo da moramo napraviti novi razred koji ga modelira i prilagoditi parser da ga bude svjestan (koji se simbol 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 konkretne implementacije 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 vrijednosti desnog djeteta i potom izračunati sumu tih rezultata i to vratiti kao svoj rezultat. Ovakvo rješenje prikazano je dijagramom razreda na slici 17.6. 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 17.6. Modificirano podstablo za pamćenje izraza



Implementacije razreda koji predstavljaju različite vrste izraza prikazane su u izvornom kodu 17.12. Pri tome su ostavljeni samo dijelovi koda koji su novi u odnosu na početnu implementaciju.

Primjer 17.12. 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 dođe do
11     *       pogreške pri izračunu
12     */
13     public abstract Vector evaluate(Map<String, Vector> variables);
14 }
15
16 public class NodeVector extends ExpressionNode {
17
18     // ostatak koda ...
19
20     @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 }
```

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 rekursivno 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 rekursivno delegiranje sadržanim objektima. Pojednostavljenje koje nudi ovaj oblikovni obrazac temelji se na *rekursivnoj 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` rekursivno 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 17.13.

Primjer 17.13. 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 dode 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+" already 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.

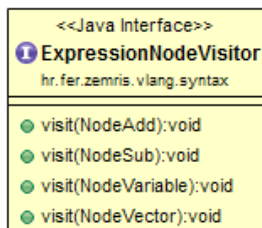
Izvođenje programa: pokušaj treći

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 potvrđan, 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 17.7 odnosno izvorni kod 17.14 prikazuje jednu moguću definiciju takvog sučelja.

Slika 17.7. Apstraktna operacija



Primjer 17.14. 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     /**
22      * Operacija posjećivanja čvora {@link NodeSub}.
23      * @param add čvor
24      */
25     public void visit(NodeSub sub);
26
27     /**
28      * Operacija posjećivanja čvora {@link NodeVariable}.
29      * @param add čvor
30      */
31     public void visit(NodeVariable var);
32
33     /**
34      * Operacija posjećivanja čvora {@link NodeVector}.
35      * @param add čvor
36      */
37     public void visit(NodeVector vector);
38 }
  
```

Pretpostavimo sada da imamo konkretnu implementaciju ovog sučelja u kojem je svaka metoda implementirana tako da zna obaviti traženu operaciju nad svakim od podržanih

konkretnih 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.

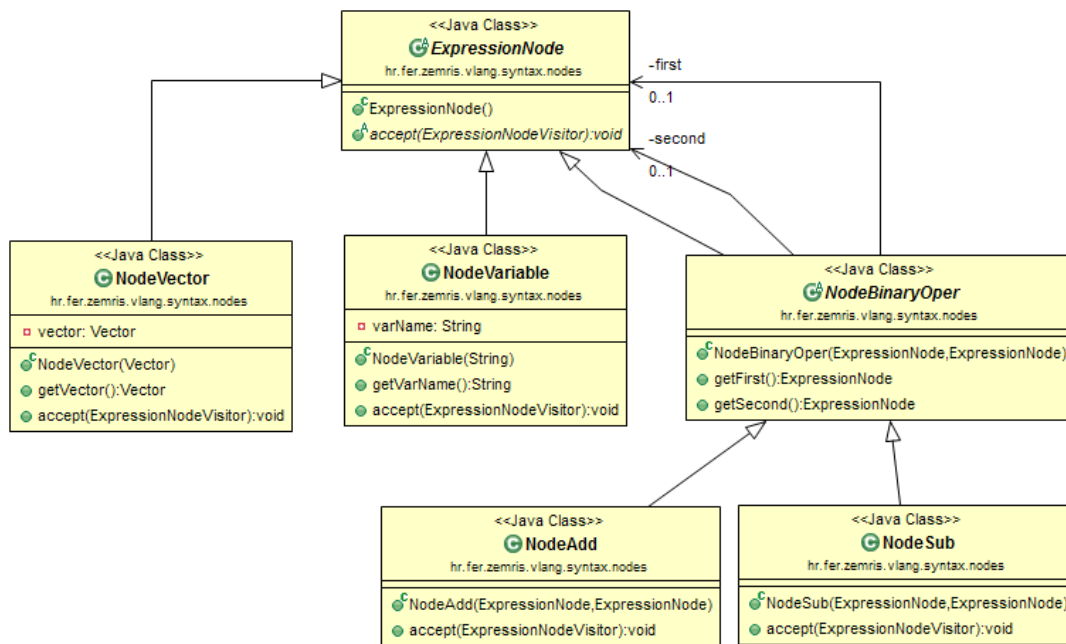
```
List<ExpressionNode> nodes = getExpressions();
ExpressionNodeVisitor visitor = getConcreteVisitor();

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 konretan 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 17.8.

Slika 17.8. Apstraktna operacija



Izvorni kod 17.15 prikazuje provedene korekcije u kodu.

Primjer 17.15. 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
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```



```
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 polimornim 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 polimornim 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 17.16 a druge u izvornom kodu 17.17.

Primjer 17.16. 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 17.17. 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 17.18.

Primjer 17.18. 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 dode do
19  * bilo kakve promjene u broju i vrsti operatora i operandada
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+" 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  }

```

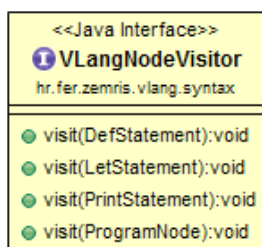
```

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      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 17.9, izvorni kod 17.19) a potom u model naredbi dodati potporu za prihvaćanje posjetitelja (dijagram razreda na slici 17.10, izvorni kod 17.20).

Slika 17.9. Apstraktna operacija nad naredbama



Primjer 17.19. 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);

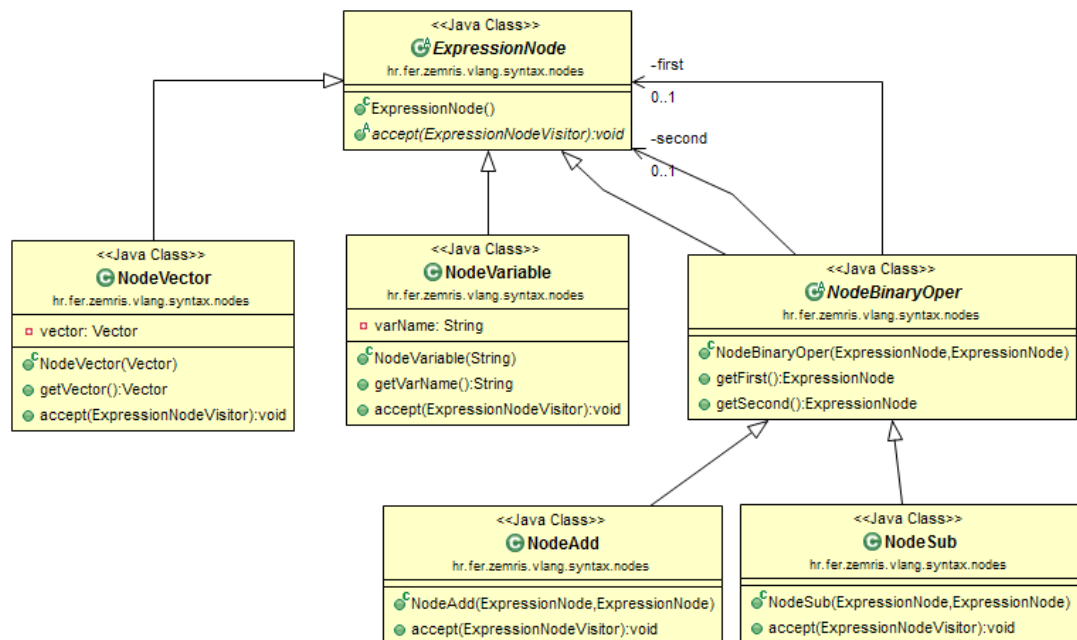
```

```

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 17.10. Apstraktna operacija



Primjer 17.20. 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
12 public class ProgramNode extends VLangNode {
13
14

```

```
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
10

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 17.21 i zadaća mu je izvođenje programa čije stablo dobiva u konstruktoru. Drugi primjer je posjetitelj prikazan izvornim kodom 17.22 čija je zadaća rekonstruirati izvorni kod programa iz sintaksnog stabla.

Primjer 17.21. 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+" already 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 17.22. Posjetitelj koji rekonstruira izvorni kod programa

```
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.VLangNodeVisitor;
5 import hr.fer.zemris.vlang.syntax.nodes.DefStatement;
6 import hr.fer.zemris.vlang.syntax.nodes.ExpressionNode;
7 import hr.fer.zemris.vlang.syntax.nodes.LetStatement;
8 import hr.fer.zemris.vlang.syntax.nodes.PrintStatement;
9 import hr.fer.zemris.vlang.syntax.nodes.ProgramNode;
10 import hr.fer.zemris.vlang.syntax.nodes.VLangNode;
11
12 public class ProgramPrintVisitor implements VLangNodeVisitor {
13
14     /**
15      * Izlaz u koji se zapisuje izvorni kod programa.
16      */
17     private StringBuilder sb = new StringBuilder();
18
19     @Override
20     public void visit(DefStatement stmt) {
21         sb.append("def ");
22         boolean first = true;
23         for(String varName : stmt.getVariables()) {
24             if(first) {
25                 first = false;
26             } else {
27                 sb.append(", ");
28             }
29             sb.append(varName);
30         }
31         sb.append(": vector;\n");
32     }
33
34     @Override
35     public void visit(LetStatement stmt) {
36         sb.append("let ").append(stmt.getVarName()).append(" = ");
37         stmt.getExpression().accept(new ExpressionPrintVisitor(sb));
38         sb.append(";\n");
39     }
40
41     @Override
42     public void visit(PrintStatement stmt) {
43         ExpressionNodeVisitor expVisitor =
44             new ExpressionPrintVisitor(sb);
45         sb.append("print ");
46         boolean first = true;
47         for(ExpressionNode node : stmt.getList()) {
48             if(first) {
49                 first = false;
50             } else {
51                 sb.append(", ");
```

```
52     }
53     node.accept(expVisitor);
54 }
55 sb.append("; \r\n");
56 }
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 17.23.

Primjer 17.23. 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 }
```

```
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 ???

Primjer 17.24. 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 }
```

```
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 `a` 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 komponenta 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/>

Poglavlje 18. Višedretvene aplikacije

- proces i dretva
- enumeracija detvi u Java programu
- dretve main, garbage collector i slično
- obične dretve, demonske dretve
- kada proces umire? (System.exit, sve obične dretve kada umru)
- pokretanje dretve nasljeđivanjem
- sučelje Runnable
- sinkronizacija dviju dretvi: metoda join
- problem višedretvenosti: neatomarnost operacija
 - pokazati na primjeru uvećavanja iste varijable
- java.util.concurrent.atomic.AtomicLong.getAndIncrement
 - to je rješenje za atomaran rad s jednom varijablom
- pojam kritičnog odsječka
 - isto rješenje preko kritičnog odsječka i eksterne sinkronizacije
 - metode wait, notify, notifyAll
- na što se sinkroniziraju synchronized metode (statičke vs. one koje pripadaju primjerku razreda)
- implementacija razreda Mutex
- uvećavanje varijable preko takvog mutexa
- implementacija oblikovnog obrasca Producer-Consumer uz ograničenu veličinu spremnika
- SynchronousQueue<E>
- CyclicBarrier
- Executors, ExecutorService, Callable, FutureTask
- Fork/Join framework

Poglavlje 19. Raspodijeljene aplikacije. Paket java.net.

- Model komunikacije prema ISO-OSI
- Uporaba DNS sustava za razrješavanje IP adresa
- Dohvat adrese lokalnog računala
- IP adresa, par IP adresa + port
- Izrada aplikacija koje koriste protokol UDP
 - UDP poslužitelj
 - UDP klijent
- Izrada aplikacija koje koriste protokol TCP
 - UDP poslužitelj
 - UDP klijent

Poglavlje 20. Swing 1: općenito

- prozori
- različite komponente poput JButton, JLabel, JTextField, JTextArea, JTextPane, JScrollPane
- Layout manageri
 - BorderLayout
 - CardLayout
 - GridLayout
 - FlowLayout
 - drugi
- Vlastita izrada layout managera
- Izbornici
- Akcije
- Gumbi, izbornici i akcije
- InputMap, ActionMap kod JTextPane-a

Poglavlje 21. Swing 2: modeli i pogledi

- Ideja Observer design patterna
- Model liste: ListModel
 - sučelje za observe
 - model koji sadrži prvih 100 parnih brojeva i ne dozvoljava promjene
 - stvaranje dvije swing List komponente koje prikazuju identični model
 - model koji sadrži polje brojeva i omogućava dodavanje i brisanje elemenata
 - stvaranje dvije swing List komponente koje prikazuju taj model, a u GUI-ju postoji mogućnost dodavanja/brisanja elemenata
 - AbstractListModel, DefaultListModel
- Model tablice: TableModel
 - sučelje za observe
 - model koji sadrži podatke o prvih 100 brojeva: vrijednost, te je li paran ili nije
 - AbstractTableModel, DefaultTableModel
- `JTextField.getDocument()` -- proučiti, `Document`, `PlainDocument`, primjer s `UpperCaseDocument`-om

Poglavlje 22. Swing 3: vlastite komponente i višedretvenost

- crtanje vlastitih komponenti: `paintComponent`
- `size`, `insets`
- ukrašavanje borderima, `BorderFactory`
- definiranje vlastitog modela i `repaint()` kad je potrebno
- komponenta koja predstavlja digitalni sat koji samostalno odbrojava uporabom druge dretve
 - `SwingUtilities.invokeLater`, `SwingUtilities.invokeLaterLater`
 - `SwingWorker`

Poglavlje 23. Uporaba relacijskih baza podataka iz Jave: SQL

- Direktna uporaba SQL-a (PreparedStatement, ResultSet-ovi i slično)

Poglavlje 24. 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 25. 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 26. 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 27. Tehnologija Java Servlet i JSP

- dinamične web aplikacije
 - razvoj: CGI, skriptni jezici, punokrvna programska rešenja
- 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 28. 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 29. 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 30. Test Driven Development

- Prema prethodnim predavanjima...

Poglavlje 31. 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 varijablu okruženja `PATH`.

Dodatak B. Instalacija alata ant, maven i gradle

Alati **ant**, **maven** i **gradle** služe za automatizaciju pojedinih koraka u razvojnem 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-1.8.4-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-1.8.4` 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-1.8.4`. 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-1.8.4\bin`.

Sada možete provjeriti je li sve instalirano kako spada. Otvorite ljsku (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-3.0.5-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-3.0.5` 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-3.0.5`. 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-3.0.5\bin`.

Sada možete provjeriti je li sve instalirano kako spada. Otvorite ljsku (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 5.6 pa u tom slučaju skidate ZIP arhivu `checkstyle-5.6-bin.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `checkstyle-5.6` 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.



Nužno korigirati u verziji 5.6

Popis stavki na koje je potrebno paziti (tj. definicija stila) kod ovog se alata definira pomoću za to namijenjene XML datoteke. Alat već dolazi s pripremljenim opisom stila koji odgovara konvenciji koja je korištena od strane tvrtke Sun, i taj je opis dostupan u vršnom direktorij u koji je instaliran alat kao datoteka `sun_checks.xml`. Međutim, kako je od verzije 7 Javine platforme došlo do određenih korekcija, situacija koja je prije bila pogrešna od te verzije više nije pogrešna pa je iz alata izbačena jedna provjera ali je u datoteci s definicijom stila ostala referenca na tu provjeru što uzrokuje lažno pucanje analize. Stoga iskopirajte datoteku `sun_checks.xml` u `sun_checks_2.xml`, otvorite datoteku `sun_checks_2.xml` u uređivaču teksta, pronađite redak u kojem se nalazi definicija:

```
<module name="DoubleCheckedLocking"/>
```

(trebalo bi biti u retku 148) i zakomentirajte poziv modula `DoubleCheckedLocking`:

```
<!-- module name="DoubleCheckedLocking" -->
```

čime ćete otkloniti navedeni problem. Potom prilikom definicije stila koristite ovu novu datoteku.

Uz prethodno navedenu modifikaciju datoteke `sun_checks_2.xml` preporuča se provesti i sljedeću modifikaciju. 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 složiti 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 izmijenite 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 5.0.2 pa u tom slučaju skidate ZIP arhivu `pmd-bin-5.0.2.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `pmd-bin-5.0.2` 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.

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 2.0.2 pa u tom slučaju skidate ZIP arhivu `findbugs-2.0.2.zip`.

Nakon što ste skinuli arhivu, raspakirajte njezin sadržaj u prethodno stvoreni vršni direktorij `D:\usr`. Unutar tog direktorija nastat će poddirektorij `findbugs-2.0.2` 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 ovog alata i trenutno aktualna je verzija 4 koja nije kompatibilna s prethodnom verzijom. 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.jar` i `hamcrest-core.jar`. U prethodno stvorenom direktoriju `D:\usr` napravite poddirektorij `junit-4.11` 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 0.6.3. Na stranici je dostupna poveznica preko koje je moguće dohvatiti alat u obliku ZIP arhive; u tom slučaju ime će biti `jacoco-0.6.3-20130211.131521-4.zip` ili slično. Skinite tu arhivu. U prethodno stvorenom direktoriju `D:\usr` napravite poddirektorij `jacoco-0.6.3` i u taj poddirektorij raspakirajte prethodno skinute arhivu. Direktorij `lib` koji je dio te arhive trebao bi imati punu stazu: `D:\usr\jacoco-0.6.3\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`
`-Xrunjdp: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  
-Xrunjdp: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  
-Xrunjdp: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 pronaći 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 Java 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 Database¹ te Apache Derby². 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.

¹<http://www.h2database.com/html/main.html>

²<http://db.apache.org/derby>

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).

```
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 dva 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 Apache 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 SYCS_UTIL.SYCS_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 SYCS_UTIL.SYCS_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 kojim 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 dokumentaciju³.

³http://db.apache.org/derby/manuals/#docs_10.10

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.
