Univerzitet Singidunum



Dejan Živković

JAVA PROGRAMIRANJE

OBJEKTNO ORIJENTISANI KONCEPT PROGRAMIRANJA

Beograd, 2021.

UNIVERZITET SINGIDUNUM

Dejan Živković

JAVA PROGRAMIRANJE

Peto izdanje

Beograd, 2021.

JAVA PROGRAMIRANJE

Autor:
dr Dejan Živković

Recenzent:
dr Dragan Cvetković
dr Slavko Pešić

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32
www.singidunum.ac.rs

Za izdavača:
dr Milovan Stanišić

Tehnička obrada:
Dejan Živković

Dizajn korica:
Aleksandar Mihajlović

Godina izdanja:
2021.

Tiraž:
1500 primeraka

Štampa:
BiroGraf, Beograd

ISBN: 978-86-7912-521-7

Copyright:
© 2021. Univerzitet Singidunum
Izdavač zadržava sva prava.
Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljeno.

Sadržaj

Spisak programa . . . . . **iii**

Predgovor . . . . . **v**

1 **Osnove Jave** **1**
1.1 Osnovni koncepti – *1* • 1.2 Naredbe – *17* • 1.3 Metodi – *36* • 1.4 Uvod u klase – *50*

2 **Klase** **61**
2.1 Klase i objekti – *62* • 2.2 Promenljive klasnog tipa – *67* • 2.3 Konstrukcija i inicijalizacija objekata – *74* • 2.4 Uklanjanje objekata – *81* • 2.5 Skrivanje podataka (enkapsulacija) – *83* • 2.6 Službena reč `this` – *87*

3 **Nizovi** **93**
3.1 Jednodimenzionalni nizovi – *93* • 3.2 Dvodimenzionalni nizovi – *112* • 3.3 Dinamički nizovi – *122*

4 **Nasleđivanje klasa** **135**
4.1 Osnovni pojmovi – *135* • 4.2 Hijerarhija klasa – *142* • 4.3 Službena reč `super` – *150* • 4.4 Klasa `Object` – *157* • 4.5 Polimorfizam – *161*

5 **Posebne klase i interfejsi** **167**
5.1 Nabrojivi tipovi – *168* • 5.2 Apstraktne klase – *173* • 5.3 Interfejsi – *178* • 5.4 Ugnježđene klase – *184*

6 **Grafičko programiranje** **195**

6.1 Grafički programi – *195* • 6.2 Grafički elementi – *199* • 6.3 Definisanje grafičkih komponenti – *216* • 6.4 Klase za crtanje – *220* • 6.5 Rukovanje događajima – *232*

7 **Programske greške** **249**
7.1 Ispravni i robustni programi – *249* • 7.2 Izuzeci – *254* • 7.3 Postupanje sa izuzecima – *264* • 7.4 Definisanje klasa izuzetaka – *270*

8 **Programski ulaz i izlaz** **275**
8.1 Tokovi podataka – *276* • 8.2 Datoteke – *289* • 8.3 Imena datoteka – *295*

9 **Programske niti** **311**
9.1 Osnovni pojmovi – *311* • 9.2 Zadaci i niti za paralelno izvršavanje – *313* • 9.3 Sinhronizacija niti – *326* • 9.4 Kooperacija niti – *343*

10 **Mrežno programiranje** **349**
10.1 Komunikacija računara u mreži – *349* • 10.2 Veb programiranje – *352* • 10.3 Klijent-server programiranje – *359* • 10.4 Višenitno mrežno programiranje – *371*

11 **Generičko programiranje** **389**
11.1 Uvod – *389* • 11.2 Sistem kolekcija u Javi – *391* • 11.3 Definisanje generičkih klasa i metoda – *423* • 11.4 Ograničenja za parametar tipa – *430*

Literatura **443**

Indeks **445**

Spisak programa

1.1 Program *Zdravo* 4
2.1 Bacanja dve kocke 79
3.1 Prebrojavanje glasova na izborima 101
3.2 Argumenti programa u komandnom redu 104
3.3 Izvlačenje loto brojeva 108
3.4 Profit firme sa više prodavnica 119
3.5 Telefonski imenik 131
6.1 Program *Zdravo* sa grafičkim dijalozima za ulaz/izlaz . . . 198
6.2 Prikaz praznog okvira 201
6.3 Postupak *FlowLayout* za razmeštanje komponenti 209
6.4 Postupak *GridLayout* za razmeštanje komponenti 211
6.5 Postupak *BorderLayout* za razmeštanje komponenti 214
6.6 Test klase `Graphics` 218
6.7 Aplet za crtanje kamiona 230
6.8 Brojanje pritisaka tasterom miša na dugme 239
6.9 Brojanje pritisaka tasterom miša na dugme, kraća verzija . . 241
6.10 Uslovno zatvaranje prozora programa 246
8.1 Sortiranje datoteke brojeva 293
8.2 Lista svih datoteka u direktorijumu 299
8.3 Kopiranje datoteke 306
9.1 Tri paralelne niti 315
9.2 Animirani tekst 324
9.3 Trka niti za brojanje 328
9.4 Nit za izvršavanje grafičkih događaja 337
9.5 Prikaz procenta završenog kopiranja datoteke 339
10.1 Jednostavni veb čitač 356
10.2 Jednostavni server 365
10.3 Jednostavni klijent 368

10.4 Jednostavni višenitni server 373
10.5 Program za „četovanje” 379

Predgovor

Programski jezik Java je stekao veliku popularnost zbog svoje elegancije i savremene koncepcije. Programi napisani u Javi mogu se zato naći u mobilnim uređajima, Internet stranama, multimedijalnim servisima, distribuiranim informacionim sistemima i mnogim drugim okruženjima ljudskog delovanja. Cilj ove knjige je da programerima pomogne da što lakše nauče Javu kako bi u svakodnevnom radu mogli da koriste moćan i moderan alat.

Java je objektno orijentisan programski jezik, stoga se u prvom delu ove knjige govori o bitnim konceptima objektno orijentisanog programiranja, kao i o načinu na koji su ti koncepti realizovani u Javi. Java je i programski jezik opšte namene koji se koristi za rešavanje zadataka iz različitih oblasti primene. Programi napisani u Javi mogu biti obični tekstualni ili grafički programi, ali i složenije aplikacije zasnovane na principima mrežnog programiranja, višenitnog programiranja, generičkog programiranja i mnogih drugih posebnih tehnika.

Naglasak u ovoj knjizi je uglavnom na naprednijim mogućnostima Jave o kojima se govori u drugom delu knjige. Ali zbog toga se od čitalaca očekuje određeno predznanje o osnovnim konceptima programiranja. Specifično, pretpostavlja se da čitaoci dobro poznaju osnovne elemente programiranja kao što su promenljive i tipovi podataka. S tim u vezi se podrazumeva poznavanje osnovnih upravljačkih programskih konstrukcija u koje spadaju naredbe dodele vrednosti promenljivim, naredbe grananja i naredbe ponavljanja. Na kraju, neophodno je i poznavanje načina za grupisanje ovih prostih upravljačkih konstrukcija u složenije programske strukture, od kojih su najznačajniji potprogrami (obični i rekurzivni). Kratko rečeno, od čitalaca se očekuje da su ranije koristili neki programski jezik i da su solidno ovladali proceduralnim načinom programiranja.

Obim i sadržaj knjige su prilagođeni nastavnom planu i programu odgovarajućeg predmeta na Fakultetu za informatiku i računarstvo Univerzi-

teta Singidunum u Beogradu. Tekst je propraćen sa velikim brojem slika i urađenih primera kojima se konkretno ilustruju novo uvedeni pojmovi. Primeri su birani tako da budu jednostavni, ali i da budu što realističniji i interesantniji. Njihova dodatna svrha je da posluže kao početna tačka za eksperimentisanje i dublje samostalno učenje. A kao što je to već uobičajeno za računarske knjige, za prikazivanje programskog teksta se koristi font sa znakovima `iste širine`.

Zahvalnost. Moje veliko zadovoljstvo je što mogu da izrazim najdublju zahvalnost svima koji su mi pomogli u pripremi ove knjige. Ovde bih posebno izdvojio kolege i studente koji su svojim sugestijama učinili da knjiga bude bolja. Veoma bih bio zahvalan i čitaocima na njihovom mišljenju o knjizi. Sve primedbe i pronađene greške (kao i pohvale) mogu se poslati na adresu `dzivkovic@singidunum.ac.rs`.

DEJAN ŽIVKOVIĆ
Beograd, Srbija
decembar 2012.

GLAVA 1

Osnove Jave

U ovoj knjizi se uglavnom govori o naprednijim mogućnostima Jave i zbog toga se od čitalaca očekuje određeno predznanje iz programiranja. Pod time se podrazumeva da čitaoci mogu bez velikih teškoća razumeti proceduralni aspekt objektno orijentisanog jezika Java. Ipak, radi kompletnosti, predmet ovog uvodnog poglavlja je kratak pregled tog aspekta i osnovnih mogućnosti jezika Java. Ovaj pregled obuhvata komentare, tipove podataka, izraze, naredbe, procedure(metode) i, na kraju, osnovne pojmove o klasama u jeziku Java. Čitaoci koji već poznaju neki programski jezik mogu verovatno samo prelistati ili čak preskočiti ovo uvodno poglavlje, ali svakako treba imati u vidu da su osnove u ovom poglavlju neophodan temelj za složenije koncepte Java programiranja.

1.1 Osnovni koncepti

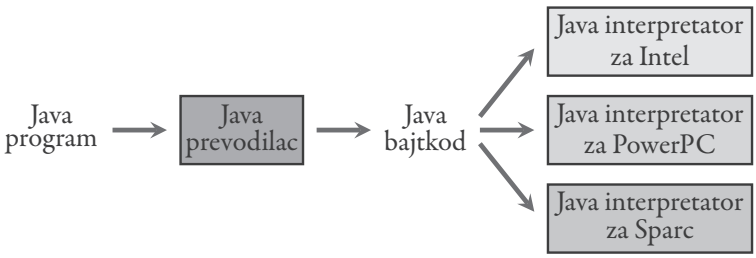
Kada se govori o Javi, treba istaći da formalna specifikacija Jave obuhvata dve relativno nezavisne celine: opis samog programskog jezika Java i opis fiktivnog računara (platforme) za izvršavanje programa napisanih u Javi. Opis programskog jezika Java ne razlikuje se mnogo od sličnih opisa za druge jezike opšte namene i tome je posvećena cela ova knjiga. Novi pristup koji Java donosi je dakle jedan fiktivni računar, tzv. *Java virtuelna mašina*, za koji se Java programi prevode radi izvršavanja. O tome se, vrlo kratko, govori u sledećem odeljku.

Java virtuelna mašina

Svaki tip procesora računara (Intel, PowerPC, Sparc, ...) ima poseban mašinski jezik i može izvršiti neki program samo ako je taj program izražen na tom jeziku. Programi napisani na jeziku visokog nivoa kao što su Java, C, C++, C#, Pascal i tako dalje, ne mogu se direktno izvršiti na računaru. Takvi programi se moraju najpre prevesti na mašinski jezik odgovarajućeg procesora računara. Ovo prevođenje programa na jeziku visokog nivoa u ekvivalentan program na mašinskom jeziku obavljaju specijalni programi koji se zovu *prevodioci* (ili *kompajleri*).

U slučaju programskog jezika Java se koristi pristup koji je drugačiji od uobičajenog prevođenja programa na mašinski jezik stvarnog procesora računara. Naime, programi napisani u Javi se i dalje prevode na mašinski jezik, ali taj mašinski jezik nije jezik nekog stvarnog procesora nego izmišljenog računara koji se naziva *Java virtuelna mašina* (skraćeno *JVM*). Mašinski jezik Java virtuelne mašine se naziva *Java bajtkod*.

Prema tome, Java program se prevodi u Java bajtkod koji se ne može direktno izvršiti na stvarnom računaru. Za izvršavanje Java programa prevedenog u Java bajtkod potreban je dodatni program koji interpretira instrukcije Java bajtkoda na stvarnom računaru. Ovaj interpretator Java bajtkoda često se u žargonu naziva istim imenom „Java virtuelna mašina", što ponekad dovodi do zabune jer treba razlikovati opis fiktivnog računara od specijalnog programa stvarnog računara koji simulira instrukcije tog fiktivnog računara na stvarnom računaru. Šema izvršavanje Java programa na različitim platformama prikazana je na slici 1.1.



Slika 1.1: Izvršavanje Java programa na različitim platformama.

Opis Java virtuelne mašine nije predmet ove knjige, jer za potrebe izučavanja programskog jezika Java nije potrebno znati mašinski jezik računara na kome se Java program izvršava. Detaljnije poznavanje Java virtuelne mašine potrebno je specijalnim stručnjacima koji pišu prevodioce ili interpretatore za Javu. Običnim programerima koji koriste programski jezik Java za pisanje svojih programa dovoljno je samo poznavanje opšte šeme na slici 1.1 koja se primenjuje za izvršavanje Java programa.

Napomenimo još da programski jezik Java u principu nema nikakve veze sa Java virtuelnom mašinom. Programi napisani u Javi bi se svakako mogli prevoditi u mašinski jezik stvarnog računara. Ali i obrnuto, programi na drugom programskom jeziku visokog nivoa bi se lako mogli prevoditi u Java bajtkod. Ipak, kombinacija Java jezika i Java virtuelne mašine obezbeđuje programiranje na modernom, objektno orijentisanom jeziku i istovremeno bezbedno izvršavanje napisanih programa na različitim tipovima računara.

Razvoj programa

Razvoj programa u svim programskim jezicima obuhvata tri glavna koraka: unošenje, prevođenje i pokretanje (testiranje) programa. Ovaj često mukotrpan proces može se u slučaju Jave obavljati u okviru dva osnovna razvojna okruženja: tekstualnog i grafičkog. Dok je tekstualno razvojno okruženje za Javu relativno standardno (konzolni prozor za Unix, DOS prozor za Windows ili slično komandno okruženje za druge operativne sisteme), programerima su na raspolaganju mnoga grafička razvojna okruženja, kako besplatna tako i komercijalna (NetBeans, Eclipse, DrJava, BlueJ, JCreator i tako dalje).

Među ovim okruženjima se ne može izdvojiti jedno koje je najbolje, jer svako ima svoje prednosti i mane, a i njihov izbor često zavisi od ličnog ukusa. Zbog toga se u nastavku ograničavamo na najjednostavnije tekstualno okruženje za Windows. Ovim se ne gubi mnogo na kompletnosti, jer se na kraju sve svodi na izvršavanje određenih komandi operativnog sistema. Pored toga, sva okruženja rade na sličan način i nije teško preći na drugo okruženje ukoliko se poznaje bilo koje.

Program u Javi se na logičkom nivou sastoji od jedne ili više klasa. Izvor-

ni tekst svake klase se piše u posebnoj datoteci čiji naziv mora početi imenom klase koju sadrži, a ekstenzija naziva te datoteke mora biti `java`.[1] Na primer, u nastavku je u listingu 1.1 prikazan jednostavan program koji se sastoji od samo jedne klase `Zdravo`.

Listing 1.1: Program *Zdravo*

```
public class Zdravo {
   public static void main(String[] args) {
      System.out.println("Zdravo narode!");
   }
}
```

U ovom primeru dakle, koristeći bilo koji editor, tekst klase (programa) `Zdravo` treba uneti u računar i taj izvorni tekst treba sačuvati u datoteku pod obaveznim nazivom `Zdravo.java`.

Pokretanjem izvršavanja svake klase (programa) u Javi započinje se izvršavanje metoda (funkcije) u toj klasi sa službenim imenom `main`. U prethodnom primeru klase `Zdravo`, taj metod se sastoji od samo jedne naredbe kojom se prikazuje određena poruka na ekranu.

Ali pre izvršavanja, bilo koji program u Javi se mora prevesti u odgovarajući bajtkod. Svaka klasa u programu se može prevoditi nezavisno od drugih klasa programa. Prevođenje jedne klase se obavlja Java prevodiocem koji se pokreće komandom `javac`. Na primer, komanda za prevođenje klase `Zdravo` u datoteci `Zdravo.java` je:

```
javac Zdravo.java
```

Rezultat prevođenja klase `Zdravo` je njen bajtkod koji se smešta u datoteku pod nazivom `Zdravo.class`. Primetimo da naziv ove datoteke počinje imenom klase čiji bajtkod sadrži, a završava se standardnom ekstenzijom `class`.

Najzad, izvršavanje (interpretiranje) dobijenog bajtkoda neke klase obavlja se Java interpretatorom koji se pokreće komandom `java`. Na primer, komanda za izvršavanje bajtkoda klase `Zdravo` u datoteci `Zdravo.class` je:

[1] Iako neki Java prevodioci dopuštaju pisanje više klasa u istoj datoteci, takva praksa se ne preporučuje.

```
java Zdravo
```

Primetimo da je ovde argument Java interpretatora samo klasa čiji bajtkod treba interpretirati, a ne datoteka `Zdravo.class` koja sadrži taj bajtkod. Naravno, interpretiranje bajtkoda klase `Zdravo` počinje od metoda `main()` u toj klasi, a efekat toga je izvršavanje jedine naredbe u tom metodu kojom se prikazuje navedena poruka na ekranu.

U prethodnom primeru se program sastojao od samo jedne klase, ali se sličan postupak primenjuje i za izvršavanje programa koji se sastoji od više klasa. Praktično jedina razlika je što se sve klase moraju pojedinačno prevesti u Java bajtkod. Nakon toga, izvršavanje celog programa izvodi se interpretiranjem bajtkoda logički početne klase programa. A time će se onda prema logici programa u odgovarajućem trenutku interpretirati i bajtkodovi ostalih klasa programa.

Istaknimo još jednu mogućnost koja se često koristi radi testiranja programa. Naime, nema nikakve prepreke da se u više klasa koje čine program nalazi metod `main()`. Ali onda se postavlja pitanje odakle se počinje sa izvršavanjem takvog programa? Odgovor je logičan — to se određuje argumentom komande za izvršavanje programa navođenjem one klase za to od čijeg metoda `main()` treba početi izvršavanje.

Komentari

Komentar je tekst na prirodnom jeziku u programu koji služi za objašnjenje delova programa drugim programerima. Takva konstrukcija se zanemaruje od strane Java prevodioca i potpuno se preskače prilikom prevođenja programa. Iako za Java prevodilac komentari kao da ne postoje, to ne znači da su oni nevažni. Bez komentara je vrlo teško razumeti kako radi neki iole složeniji program. Zbog toga se pisanje dobrih i kratkih komentara u programu nikako ne sme olako shvatiti.

U Javi se mogu koristiti tri vrste komentara. Prva vrsta komentara su kratki komentari koji se pišu u jednom redu teksta. Njihov početak u nekom redu se označava simbolom `//` i obuhvataju tekst do kraja tog reda. Na primer:

```
// Prikazivanje ocena svih studenata
int i = 0;  // inicijalizacija broja studenata
```

Druga vrsta komentara sadrži tekst koji se prostire u više redova. Oni počinju simbolom `/*`, obuhvataju tekst preko proizvoljnog broja redova i završavaju se simbolom `*/`. Ceo tekst između simbola `/*` i `*/` potpuno se zanemaruje od strane Java prevodioca. Na primer:

```
/*
 * Uspostavljanje veze sa veb serverom.
 * Ako uspostavljanje veze ne uspe nakon tri pokušaja,
 * program se prekida jer nema smisla nastaviti obradu.
 */
```

Treća vrsta komentara su specijalni slučaj druge vrste i počinju simbolom `/**`, a ne `/*`, ali se isto završavaju simbolom `*/`. Ovi komentari se nazivaju *dokumentacioni komentari* jer služe za pisanje dokumentacije o klasi direktno unutar njenog izvornog teksta. Pomoćni program `javadoc` izdvaja ove komentare u poseban hipertekst dokument koji se može lako čitati bilo kojim veb pretraživačem. Pored običnog teksta, dokumentacioni komentari mogu sadržati i HTML tagove i specijalne reči koje počinju znakom `@`. Na primer:

```
/**
 * Prenošenje datoteke na veb server.
 *
 * @param datoteka Datoteka koja se prenosi.
 * @return <tt>true</tt> ako je prenos bio uspešan,
 *         <tt>false</tt> ako je prenos bio neuspešan.
 * @author Dejan Živković
 */
```

Potpun opis sintakse dokumentacionih komentara nije svakako neophodan za učenje Jave. Zato se čitaocima prepušta da, kada im zatrebaju detaljnija objašnjenja u vezi sa dokumentacionim komentarima, o tome sami pročitaju u zvaničnoj dokumentaciji za pomoćni program `javadoc`.

Slobodan format programa

Java je programski jezik *slobodnog formata*, odnosno ne postoje sintaksna ograničenja u vezi sa načinom na koji se pojedine sastavne konstrukcije pišu unutar programa. To znači da je izgled teksta programa sasvim slobodan. Tako se, između ostalog,

- tekst programa može pisati od proizvoljnog mesta u redu, a ne od početka reda;
- naredbe se ne moraju pisati u pojedinačnim redovima, nego se više njih može nalaziti u istom redu;
- između reči teksta programa može biti proizvoljan broj razmaka, a ne samo jedan;
- između dva reda teksta programa može biti proizvoljan broj praznih redova.

Sa gledišta Java prevodioca, na primer, potpuno prihvatljiv izgled klase `Zdravo` na strani 4 je i ovakav njen oblik:

```
     public class Zdravo{public                    static
            void main(String[]
args) {       System.              out                    .println
  ("Zdravo narode!"          )         ;      }}
```

Ali, naravno, ova „ispravna" verzija klase `Zdravo` je potpuno nečitljiva za ljude. Slobodan format jezika Java ne treba zloupotrebljavati, već to treba iskoristiti radi pisanja programa čija vizuelna struktura jasno odražava njegovu logičku strukturu. Pravila dobrog stila programiranja zato nalažu da se piše jedna naredba u jednom redu, da se uvlačenjem i poravnavanjem redova označi blok naredbi koje čine jednu logičku celinu, da imena kojima se označavaju razni elementi program budu smislena i tako dalje.

Slobodan format programa u Javi treba dakle iskoristiti da bi se struktura teksta programa učinila što jasnijom za onoga ko pokušava da na osnovu tog teksta shvati kako program radi. Uz pridržavanje ove preporuke i pisanjem dobrih komentara u ključnim delovima programa, razumljivost komplikovanih programa se može znatno poboljšati. Laka razumljivost programa je važna zbor njegovog eventualnog kasnijeg modifikovanja, jer tada treba tačno razumeti kako program radi da bi se mogao ispravno izmeniti.

Tipovi podataka

Podaci sa kojima se u Java programima može raditi dele se u dve glavne kategorije:

1. *Primitivni (prosti) tipovi podataka* su neposredno raspoloživi tipovi podataka koji su unapred „ugrađeni" u jezik.
2. *Klasni tipovi podataka* su novi tipovi podataka koje programer mora sâm da definiše.

Primitivnim tipovima podataka u Javi su obuhvaćeni osnovni podaci koji nisu objekti. Vrednosti primitivnih tipova podataka su atomični u smislu da se ne mogu deliti na sastavne delove. Tu spadaju celi brojevi, realni brojevi, alfabetski znakovi i logičke vrednosti. U tabeli 1.1 se nalazi spisak svih primitivnih tipova sa njihovim osnovnim karakteristikama.

Tip podataka	Veličina podataka	Opseg vrednosti
`byte`	1 bajt	$[-128, 127]$
`short`	2 bajta	$[-32768, 32767]$
`int`	4 bajta	$[-2^{32}, 2^{32} - 1]$
`long`	8 bajtova	$[-2^{64}, 2^{64} - 1]$
`float`	4 bajta	$\approx \pm 10^{38}$
`double`	8 bajtova	$\approx \pm 10^{308}$
`char`	2 bajta	—
`boolean`	1 bit	—
`void`	—	—

Tabela 1.1: Primitivni tipovi podataka.

Tipovi podataka `byte`, `short`, `int` i `long` služe za rad sa celim brojevima, odnosno brojevima bez decimala kao što su 17, –1234 i 0. Ovi celobrojni tipovi podataka se razlikuju po veličini memorijske lokacije koja se koristi za binarni zapis celih brojeva.

Tipovi podataka `float` i `double` služe za predstavljanje realnih brojeva, odnosno brojeva sa decimalama kao što su 1.69, –23.678 i 5.0. Ova dva tipa

se razlikuju po veličini memorijske lokacije koja se koristi za binarni zapis realnih brojeva, ali i po tačnosti tog zapisa (tj. maksimalnom broju cifara realnih brojeva).

Tip podataka `char` sadrži pojedinačne alfabetske znakove kao što su slova (`A`, `a`, `B`, `b`, ...), cifre (`0`, `1`, ..., `9`), znakovi punktuacije (`&`, `%`, razmak, ...) i neki specijalni znakovi (za novi red, za tabulator, ...). Znakovi tipa `char` predstavljaju se u binarnom obliku prema standardu *Unicode*, koji je izabran zbog mogućnosti predstavljanje pisama svih jezika na svetu, odnosno zbog internacionalizacije koja je uvek bila među osnovnim ciljevima Jave.

Tip podataka `boolean` služi za predstavljanje samo dve logičke vrednosti: tačno (`true`) i netačno (`false`). Logičke vrednosti u programima se najčešće dobijaju kao rezultat izračunavanja relacijskih operacija.

Najzad, deveti primitivni tip podataka `void` je na naki način degenerativan jer ne sadrži nijednu vrednost. On je analogan praznom skupu u matematici i služi u nekim sintaksnim konstrukcijama za formalno označavanje tipa podataka bez vrednosti.

Svi primitivni tipovi podataka imaju tačno određenu veličinu memorijske lokacije u bajtovima za zapisivanje njihovih vrednosti. To znači da primitivni tipovi podataka imaju tačno definisan opseg vrednosti koje im pripadaju. Na primer, vrednosti koje pripadaju tipu `short` su svi celi brojevi od –32768 do 32767.

Osnovna osobina primitivnih tipova podataka je da se njima predstavljaju proste vrednosti (brojevi, znakovi i logičke vrednosti) koje se ne mogu dalje deliti. Pored toga, za svaki primitivni tip podatka su definisane i različite operacije koje se mogu primenjivati nad vrednostima određenog tipa. Na primer, za vrednosti numeričkih tipova (celi i realni brojevi) definisane su uobičajene aritmetičke operacije sabiranja, oduzimanja, množenja i deljenja.

Svi ostali tipovi podataka u Javi pripadaju kategoriji klasnih tipova podataka kojima se predstavljaju klase objekata. Objekti su „složeni" podaci po tome što se grade od sastavnih delova kojima se može nezavisno manipulisati. O klasama i objektima biće mnogo više reči u nastavku knjige.

Treba na kraju primetiti da stringovi (tj. nizovi znakova), koji su često korišćena vrsta podataka u programiranju, nemaju odgovarajući primitivni tip u Javi. Skup stringova se u Javi smatra klasnim tipom, odnosno stringo-

vi se podrazumevaju da su objekti koji pripadaju unapred definisanoj klasi `String`.

Promenljive

Promenljiva je simboličko ime za neku memorijsku lokaciju koja služi za smeštanje podataka u programu. Sve promenljive se moraju deklarisati pre nego što se mogu koristiti u Java programu. U najprostijem obliku, deklaracija jedne promenljive se sastoji od navođenja tipa promenljive, kojim se određuje tip podataka koje promenljiva sadrži, i u produžetku imena promenljive. Na primer:

```
int godina;
long n;
float a1;
double obim;
```

U ovom primeru su deklarisane promenljive `godina`, `n`, `a1` i `obim` odgovarajućeg tipa koji je naveden ispred imena svake promenljive. Efekat deklaracije jedne promenljive je rezervisanje potrebnog memorijskog prostora u programu prema tipu promenljive i uspostavljanje veze između simboličkog imena promenljive i njenog rezervisanog prostora.

Obratite pažnju na to da promenljiva može imati različite vrednosti tokom izvršavanja programa, ali sve te vrednosti moraju biti istog tipa — onog navedenog u deklaraciji promenljive. U prethodnom primeru, vrednosti recimo promenljive `obim` mogu biti samo realni brojevi tipa `double` — nikad celi brojevi, znakovi, logičke vrednosti ili vrednosti nekog klasnog tipa.

U Javi se promenljiva ne može koristiti u programu ukoliko ne sadrži neku vrednost. Kako početno rezervisanje memorijskog prostora za promenljivu ne uključuje nikakvo dodeljivanje vrednosti promenljivoj, prosta deklaracija promenljive obično se kombinuje sa njenom inicijalizacijom da bi se promenljiva mogla odmah koristiti. Na primer:

```
int i = 0;
double x = 3.45;
String imePrezime = "Pera Perić";
```

U ovom primeru su deklarisane promenljive `i`, `x` i `imePrezime` odgovarajućeg tipa, ali im se dodatno dodeljuju početne vrednosti navedene iza znaka = u njihovim deklaracijama.

Deklaracija promenljivih može biti, u stvari, malo složenija od gore prikazanih oblika. Naime, iza tipa promenljive se može pisati lista više promenljivih (sa ili bez početnih vrednosti) koje su odvojene zapetama. Time se prosto deklariše (i inicijalizuje) više promenljivih istog tipa. Na primer:

```
int i, j, k = 32;
long n = 0L;
boolean indikator = false;
String ime, srednjeIme, prezime;
float a = 3.4f, b, c = 0.1234f;
double visina, širina;
```

Imena promenljivih (kao i imena drugih programskih elemenata) u Javi moraju početi slovom, a iza početnog slova mogu se nalaziti slova ili cifre.[2] Dužina imena promenljivih je praktično neograničena. Znakovi kao što su + ili `@` ne mogu se koristiti unutar imena promenljive, niti se mogu koristiti razmaci. Velika i mala slova se razlikuju u imenima promenljivih tako da se `visina`, `Visina`, `VISINA` i `viSiNa` smatraju različitim imenima. Izvesne reči u Javi imaju specijalnu ulogu i zato se one ne mogu korisiti za imena promenljivih. Ove *službene* (ili *rezervisane*) reči su u primerima u ovoj knjizi posebno istaknute drugom bojom ili tamnijim slovima.

Pored ovih formalnih pravila za davanje ispravnih imena promenljivim, u Java programima se koristi i jedna neformalna konvencija radi poboljšanja razumljivosti programa. Naime, imena promenljivih se pišu samo malim slovima, na primer `visina`. Ali ako se ime promenljive sastoji od nekoliko reči, recimo `brojStudenata`, onda se svaka reč od druge piše sa početnim velikim slovom.

[2] Pojmovi „slovo" i „cifra" su mnogo širi u Javi. Tako sva slova obuhvataju ne samo velika i mala slova engleske azbuke, nego i bilo koji Unicode znak koji označava slovo u nekom jeziku. Zato se, recimo, i naša slova č, ć, š, đ, ž mogu koristiti u imenima.

Konstante

Jedna promenljiva može sadržati promenljive vrednosti istog tipa tokom izvršavanja programa. U izvesnim slučajevima, međutim, vrednost neke promenljive u programu ne treba da se menja nakon dodele početne vrednosti toj promenljivoj. U Javi se u naredbi deklaracije promenljive može dodati službena reč `final` kako bi se obezbedilo da se promenjiva ne može promeniti nakon što se inicijalizuje. Na primer, iza deklaracije

```
final int MAX_POENA = 100;
```

svaki pokušaj promene vrednosti promenljive `MAX_POENA` proizvodi sintaksnu grešku.

Ove „nepromenljive promenljive" se nazivaju *konstante* zato što njihove vrednosti ostaju konstantne za sve vreme izvršavanja programa. Obratite pažnju na konvenciju u Javi za pisanje imena konstanti: njihova imena se sastoje samo od velikih slova, a za eventualno razdvajanje reči služi donja crta. Ovaj stil se koristi i u standardnim klasama Jave u kojima su definisane mnoge konstante. Tako, na primer, u klasi `Math` se nalazi konstanta `PI` koja sadrži vrednost broja π, ili u klasi `Integer` se nalazi konstanta `MIN_VALUE` koja sadrži minimalnu vrednost tipa `int`.

Izrazi

Izrazi su, pojednostavljeno rečeno, formule na osnovu kojih se izračunava neka vrednost. Primer jednog izraza u Javi je `2*r*Math.PI`.

Bitna karakteristika izraza je to što se tokom izvršavanja programa svaki izraz izračunava i kao rezultat se dobija tačno jedna vrednost koja predstavlja *vrednost izraza*. Ova vrednost izraza se može dodeliti nekoj promenljivoj, koristiti u drugim naredbama, koristiti kao argument u pozivima metoda, ili kombinovati sa drugim vrednostima u građenju složenijeg izraza.

Izrazi se dele na proste i složene. Prost izraz može biti literal (tj. „bukvalna" vrednost nekog tipa kao što je `17`, `0.23`, `true` ili `'A'`), promenljiva ili poziv metoda. Vrednost prostog izraza je sama literalna vrednost, vrednost promenljive ili rezultat poziva metoda.

Složeni izrazi se grade kombinovanjem prostih izraza sa dozvoljenim operatorima (aritmetičkim, relacijskim i drugim). Izračunavanje vredno-

sti složenog izraza odvija se primenom operacija koje označavaju operatori na vrednosti prostijih izraza koji se nalaze u složenom izrazu. Izraz `2 * r * Math.PI`, na primer, izračunava se tako što se vrednost literala `2` (koja je broj 2) množi sa aktuelnom vrednošću promenljive `r`, a zatim se ovaj međurezultat množi sa vrednošću konstante `Math.PI` (koja je 3.14 ⋯) da bi se dobila konačna vrednost izraza.

Operatori. Operatori koji služe za gradnju složenih izraza u Javi označavaju uobičajene operacije nad odgovarajućim tipovima podataka.[3] Nepotpun spisak raspoloživih operatora može se podeliti u nekoliko grupa:

- aritmetički operatori: `+`, `-`, `*`, `/`, `%`, `++`, `--`
- logički operatori: `&&`, `||`, `!`
- relacijski operatori: `<`, `>`, `<=`, `>=`, `==`, `!=`
- operatori nad bitovima: `&`, `|`, `~`, `^`
- operatori pomeranja: `<<`, `>>`, `>>>`
- operatori dodele: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- operator izbora: `?:`

Opis svih ovih operatora nije težak, ali je preobiman i verovatno suvišan, jer se pretpostavlja da čitaoci poznaju makar aritmetičke, logičke i relacijske operatore. Ukoliko to nije slučaj, detaljnija objašnjenja čitaoci mogu potražiti u svakoj uvodnoj knjizi za Javu (videti spisak literature na kraju ove knjige).

Ukoliko se u složenom izrazu nalazi više operatora, redosled primene tih operatora radi izračunavanja izraza određuje se na osnovu *prioriteta* svakog operatora. Izraz, na primer,

```
x + y * z
```

izračunava se tako što se najpre izračunava podizraz `y * z`, a zatim se njegov rezultat sabira sa `x`. To je tako zato što množenje označeno operatorom `*` ima viši prioritet od sabiranja koje je označeno operatorom `+`. Ako ovaj unapred definisan prioritet operatora nije odgovarajući, redosled izračunavanja izraza može se promeniti upotrebom zagrada. Na primer, kod izraza

[3] Operatori i njihove oznake su skoro potpuno preuzeti iz jezika C (ili C++).

`(x+y)*z` bi se najpre izračunavao podizraz u zagradi `x+y`, a zatim bi se njegov rezultat pomnožio sa `z`.

Ako se u izrazu nađe više operatora koji imaju isti prioritet, onda se redosled njihove primene radi izračunavanja izraza određuje na osnovu *asocijativnosti* operatora. Asocijativnost većine operatora u Javi definisana je tako da se oni primenjuju sleva na desno. Operatori za sabiranje i oduzimanje, na primer, imaju isti prioritet i asocijativnost sleva na desno. Zato se izraz `x+y-z` izračunava kao da stoji `(x+y)-z`. S druge strane, asocijativnost operatora dodele i unarnih operatora je zdesna na levo. Zato se izraz `x=y=z++` izračunava kao da stoji `x=(y=(z++))`.

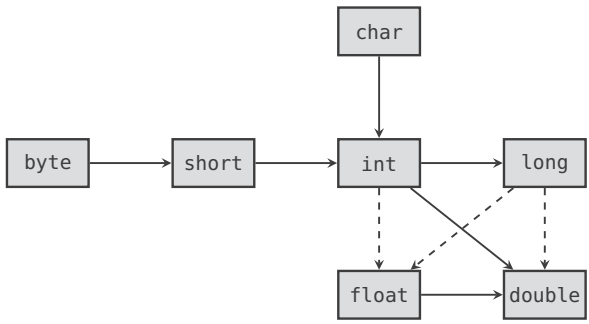
Unapred definisan prioritet i asocijativnost svih operatora u Javi može se lako saznati iz zvanične dokumentacije jezika. Međutim, pravila prioriteta i asocijativnosti operatora ne treba pamtiti napamet, jer se ona iaonako brzo zaboravljaju ili pomešaju sa sličnim, ali ne i identičnim, pravilima iz drugih programskih jezika. Umesto toga, pošto se zagrade mogu slobodno koristiti, uvek kada može doći do zabune treba navesti zagrade da bi se eksplicitno naznačio željeni redosled primene operatora kod izračunavanja izraza. Time se znatno povećava razumljivost izraza, naročito onih komplikovanijih, i smanjuje mogućnost suptilnih grešaka koje je teško otkriti.

Konverzija tipa. Prilikom izračunavanja izraza, pored prioriteta i asocijativnosti operatora, obraća se pažnja i na to to da li se neki operator primenjuje na vrednosti istog tipa. Ako to nije slučaj, vrednost jednog tipa pretvara se u ekvivalentnu vrednost drugog tipa. Na primer, prilikom izračunavanja izraza `17.4+10`, najpre se ceo broj `10` pretvara u ekvivalentni realni broj `10.0`, a zatim se izračunava `17.4+10.0` i dobija realni broj `27.4`. Ovaj postupak, koji se naziva *konverzija tipa*, obavlja se automatski ukoliko ne dolazi do gubitka podataka.

Preciznije, među primitivnim tipovima podataka dozvoljena je konverzija između celih i realnih brojeva. Pored toga, kako svakom znaku tipa `char` odgovara jedan broj prema šemi kodiranja Unicode, znakovi se mogu konvertovati u cele ili realne brojeve. U stvari, vrednosti logičkog tipa `boolean` su jedine koje ne mogu učestvovati ni u kakvoj konverziji u neki primitivni tip podataka ili iz njega. (Ovo, na primer, znači da se u Javi *ne*

može koristiti C-stil pisanja naredbi u obliku `while (1) {...}`, kao ni `if (a) {...}` gde je `a` celobrojna promenljiva.)

Primetimo da se mogu razlikovati dve osnovne vrste konverzija: *proširujuće* i *sužavajuće* konverzije. Proširujuća konverzija je ona kod koje dolazi do pretvaranja vrednosti „manjeg" tipa u ekvivalentnu vrednost „većeg" tipa. Ovde odrednice „manji" ili „veći" za tip označavaju manji ili veći skup (opseg) vrednosti koje pripadaju tipu. Na primer, ako se celobrojna vrednost tipa `int` dodeljuje promenljivoj tipa `double` ili se znak tipa `char` dodeljuje promenljivoj tipa `int`, onda se radi o proširujućoj konverziji iz tipa `int` u tip `double`, odnosno iz tipa `char` u tip `int`. U Javi se proširujuća konverzija vrši automatski kada je to potrebno i o tome obično ne treba voditi računa u programu. Na slici 1.2 su prikazane proširujuće konverzije primitivnih tipova koje su ispravne u Javi.



Slika 1.2: Proširujuće konverzije primitivnih tipova.

Pune strelice na slici 1.2 označavaju konverzije bez gubitka podataka. Isprekidane strelice označavaju konverzije bez gubitka podataka, ali uz moguć gubitak tačnosti. Na primer, ceo broj 123456789 ima više od 8 značajnih cifara koliko se najviše može predstaviti tipom `float`. Zato kada se taj veliki ceo broj konvertuje u tip `float`, rezultujući realni broj je istog reda veličine ali sa smanjenom tačnošću:

```
int i = 123456789;
float f = i; // f je 123456792.0
```

Sužavajuća konverzija je suprotna od proširujuće — vrednost većeg tipa se pretvara u manji tip. Sužavajuća konverzija nije uvek moguća: broj

17 tipa `int` ima smisla pretvoriti u istu vrednost tipa `byte`, ali broj 1700 tipa `int` ne može se pretvoriti u tip `byte`, jer brojevi tipa `byte` pripadaju intervalu samo od –128 do 127. Zbog mogućeg gubitka podataka, sužavajuća konverzija bez dodatnog zahteva proizvodi sintaksnu grešku u programu, čak i u slučaju kada vrednost koja se pretvara pripada zapravo manjem tipu:

```
int i = 17;
byte b = i; // GREŠKA!
```

Ukoliko sužavajuća konverzija svakako ne dovodi do gubitka podataka, njeno obavezno izvršavanje u izrazu može se zahtevati operatorom *eksplicitne konverzije tipa* (engl. *type cast*). Oznaka tog operatora je ime ciljnog tipa podataka u zagradi, što se piše ispred vrednosti koju treba pretvoriti u navedeni ciljni tip. Na primer:

```
int i = 17;
byte b = (byte) i; // eksplicitna konverzija 17 u tip byte
i = (int) 45.678;  // eksplicitna konverzija 45.678 u tip int
```

Eksplicitne konverzije primitivnih tipova najčešće se koriste između realnih brojeva i celih brojeva. Jedna od takvih primena operatora eksplicitne konverzije jeste za sužavajuću konverziju realnih brojeva u cele brojeve. U tom slučaju se decimalni deo realnog broja prosto odbacuje. Tako, rezultat eksplicitne konverzije `(int) 45.678` u poslednjoj naredbi prethodnog primera predstavlja ceo broj 45 koji se dodeljuje promenljivoj `i`. Primetimo da ovo nije isto što i zaokruživanje realnog broja na najbliži ceo broj — zaokruživanje realnih brojeva u tom smislu dobija se kao rezultat metoda `Math.round()`.

Mada se proširujuća konverzija celih brojeva u realne brojeve automatski izvršava, u nekim slučajevima je potrebno eksplicitno zahtevati takvu konverziju. Jedan od tih slučajeva je posledica efekta operacije deljenja za cele brojeve: rezultat celobrojnog deljena dva cela broja je isto ceo broj i eventualne decimale rezultata se odbacuju. Rezultat izraza `7 / 2`, recimo, jeste ceo broj 3, a ne tačan broj 3.5 koji je realna vrednost. Ako je svakako potrebno dobiti tačan rezultat celobrojnog deljena, onda se deljenik (ili delilac) može ekplicitno pretvoriti u realan broj. To će izazvati automatsko pretvaranje i drugog operanda u realan broj, pa će se deliti dva realna broja i dobiti tačan rezultat koji je realan broj. Na primer:

```
int m = 7;
int n = 2;
double x = (double) m / n;  // x = 3.5
```

Primetimo u ovom primeru da se na desnoj strani poslednje naredbe izraz `(double)n/m` ispravno izračunava kao da stoji `((double)n)/m`, jer operator eksplicitne konverzije ima viši prioritet u odnosu na operator deljenja. Da to nije slučaj, odnosno da operator deljenja ima viši prioritet, navedeni izraz bi se izračunavao kao da stoji `(double)(n/m)`, što bi dalo pogrešan rezultat (doduše realan broj) 3.0.

1.2 Naredbe

Naredba je jedna komanda koju izvršava Java interpretator tokom izvršavanja Java programa. Podrazumevani redosled izvršavanja naredbi je onaj u kojem su naredbe napisane u programu, mada u Javi postoje mnoge upravljačke naredbe kojima se ovaj sekvencijalni redosled izvršavanja može promeniti.

Naredbe u Javi mogu biti proste ili složene. Proste naredbe obuhvataju naredbu dodele i naredbu deklaracije promenljive. Složene naredbe služe za komponovanje prostih i složenih naredbi radi definisanja komplikovanijih upravljačkih struktura u programu. U složene naredbe spadaju blok naredbi, naredbe grananja (naredba `if`, naredba `if-else` i naredba `switch`) i naredbe ponavljanja (naredba `while`, naredba `do-while` i naredba `for`).

Naredba dodele

Naredba dodele je osnovni način da se vrednost nekog izraza dodeli promenljivoj u programu. Glavni oblik naredbe dodele je:

```
promenljiva = izraz;
```

Efekat izvršavanja naredbe dodele može se podeliti u dva koraka: najpre se izračunava vrednost izraza na desnoj strani znaka jednakosti, a zatim se ta vrednost dodeljuje promenljivoj na levoj strani znaka jednakosti. Na primer, izvršavanje naredbe dodele

```
obim = 2 * r * Math.PI; // obim kruga
```

sastoji se od izračunavanja izraza `2*r*Math.PI` na desnoj strani i upisivanja rezultujuće vrednosti u promenljivu `obim` na levoj strani znaka jednakosti. (Naravno, izraz `2 * r * Math.PI` se izračunava tako što se broj 2 množi sa aktuelnom vrednošću promenljive `r`, a taj međurezultat se zatim množi sa 3.14⋯.)

Obratite pažnju na tačku-zapetu na kraju naredbe dodele. Bez tačke-zapete se zapravo dobija *operator* dodele čija oznaka je znak jednakosti i čiji rezultat je izračunata vrednost izraza na desnoj strani znaka jednakosti (pored dodele te vrednosti promenljivoj na levoj strani znaka jednakosti). U stvari, bilo koji operator dodele može se koristiti kao naredba dodele dodavanjem tačke-zapete na kraju. (Ovo važi i za druge operatore koji proizvode sporedno dejstvo.) Na primer:

```
brojStudenata = 30; // dodela
a += b;             // kombinovana dodela
n++;                // inkrementiranje
m--;                // dekrementiranje
```

Naredba deklaracije promenljive

Sve promenljive u Javi se moraju deklarisati pre nego što se mogu koristiti u programu. Kako je Java *striktno tipovan* programski jezik, svaka promenljiva pored imena mora imati deklarisan i tip kojim se određuje tip podataka koje promenljiva sadrži. Opšti oblik *naredbe deklaracije promenljive* je:

```
tip promenljiva;
```

ili

```
tip promenljiva = izraz;
```

Ovde *`tip`* može biti jedan od primitivnih ili klasnih tipova, *`promenljiva`* predstavlja ime promenljive koja se deklariše, a *`izraz`* se izračunava da bi se dodelila početna vrednost promenljivoj koja se deklariše.

Budući da je o primerima deklarisanja promenljivih već bilo reči na strani 10, ovde ćemo se osvrnuti samo još na neke detalje u vezi sa deklaracijama

promenljivih. Pre svega, prethodni oblik naredbe deklaracije promenljive nije najopštiji jer može početi službenom rečju `final`. U tom slučaju se deklarišu zapravo konstante o kojima se bilo reči u odeljku 1.1.

Za razliku od nekih programskih jezika kod kojih se deklaracije svih promenljivih moraju nalaziti na početku programskog koda, u Javi se deklaracije promenljivih mogu pisati na bilo kom mestu. Naravno, to važi uz poštovanje pravila da svaka promenljiva mora biti deklarisana (i inicijalizovana) pre nego što se koristi u metodu.

Promenjive se mogu deklarisati unutar metoda ili unutar klasa. Promenljive deklarisane unutar nekog metoda se nazivaju *lokalne promenljive* za taj metod. One postoje samo unutar metoda u kojem su deklarisane i potpuno su nedostupne izvan tog metoda. Za lokalne promenljive se često koristi običan termin promenljive. To ne izaziva zabunu, jer kada se deklarišu unutar neke klase radi opisa atributa objekata te klase, promenljive se nazivaju *polja*.

Blok naredbi

Blok naredbi (ili kraće samo *blok*) je niz od više naredbi napisanih između otvorene i zatvorene vitičaste zagrade. Opšti oblik zapisa bloka od n naredbi je:

```
{
    naredba_1;
    naredba_2;
       ⋮
    naredba_n;
}
```

Osnovna svrha bloka naredbi je samo da se niz bilo kakvih naredbi grupiše u jednu (složenu) naredbu. Blok se obično nalazi unutar drugih složenih naredbi kada je potrebno da se više naredbi sintaksno objedine u jednu naredbu. U opštem slučaju, međutim, blok se može nalaziti na bilo kom mestu u programu gde je dozvoljeno pisati neku naredbu.

Efekat izvršavanja bloka naredbi je sekvencijalno izvršavanje niza naredbi unutar bloka. Naime, od otvorene vitičaste zagrade bloka, sve nared-

be u nizu koji sledi izvršavaju se redom jedna za drugom, sve do zatvorene vitičaste zagrade bloka.

U sledećem primeru bloka naredbi, međusobno se zamenjuju vrednosti celobrojnih promenljivih `x` i `y`, pod pretpostavkom da su ove promenljive u programu deklarisane i inicijalizovane ispred bloka:

```
{
   int z;  // pomoćna promenljiva
   z = x;  // vrednost z je stara vrednost x
   x = y;  // nova vrednost x je stara vrednost y
   y = z;  // nova vrednost y je stara vrednost x
}
```

Ovaj blok čini niz od 4 naredbe koje se redom izvršavaju. Primetimo da je pomoćna promenljiva `z` deklarisana unutar samog bloka. To je potpuno dozvoljeno i, u stvari, dobar stil programiranja nalaže da treba deklarisati promenjivu unutar bloka ako se ona nigde ne koristi van tog bloka.

Promenljiva koja je deklarisana unutar nekog bloka, potpuno je nedostupna van tog bloka, jer se takva promenljiva „uništava" nakon izvršavanja njenog bloka, odnosno memorija koju zauzima promenljiva se oslobađa za druge svrhe. Za promenljivu deklarisana unutar bloka se kaže da je *lokalna* za svoj blok ili da je taj blok njena *oblast važenja*. (Tačnije, oblast važenja lokalne promenljive je deo bloka od mesta deklaracije promenljive do kraja bloka.)

Naredbe `if` i `if-else`

Naredba `if` je najprostija upravljačka naredba koja omogućava uslovno izvršavanje niza od jedne ili više naredbi. Svaka naredba `if` se sastoji od jednog logičkog izraza i jedne naredbe (ili bloka naredbi) koji se u opštem obliku pišu na sledeći način:

```
if (logički-izraz)
   naredba
```

Izvršavanje naredbe `if` se izvodi tako što se najpre izračunava vrednost logičkog izraza u zagradi. Zatim, zavisno od toga da li je dobijena vrednost logičkog izraza tačno ili netačno, naredba koja se nalazi u produžetku logičkog izraza izvršava se ili preskače — ako je dobijena vrednost tačno (`true`),

ta naredba se izvršava; u suprotnom slučaju, ako je dobijena vrednost netačno (`false`), ta naredba se *ne* izvršava. Time se u oba slučaja izvršavanje naredbe `if` završava, a izvršavanje programa se dalje nastavlja od naredbe koja sledi iza naredbe `if`.

Konkretan primer jedne naredbe `if` je:

```
if (r != 0)
   obim = 2 * r * Math.PI;
```

U ovom primeru se naredbom `if` proverava da li je vrednost promenljive `r` (poluprečnik kruga) različita od nule. Ako je to slučaj, izračunava se obim kruga; u suprotnom slučaju, preskače se izračunavanje obima kruga.

Kako niz naredbi unutar vitičastih zagrada predstavlja jednu (blok) naredbu, naredba unutar naredbe `if` može biti i blok naredbi:[4]

```
if (x > y) {
   int z;
   z = x;
   x = y;
   y = z;
}
```

U ovom primeru se izvršava međusobna zamena vrednosti promenljivih `x` i `y` samo ukoliko je vrednost promenljive `x` početno veća od vrednosti promenljive `y`. U suprotnom slučaju se ceo blok od četiri naredbe preskače. Primetimo da je posle izvršavanja ove `if` naredbe sigurno to da je vrednost promenljive `x` manja ili jednaka vrednosti promenljive `y`.

U slučaju kada je vrednost logičkog izraza naredbe `if` jednaka netačno, programska logika često nalaže da treba uraditi nešto drugo, a ne samo preskočiti naredbu unutar naredbe `if`. Za takve slučajeve služi naredba `if-else` čiji je opšti oblik:

```
if (logički-izraz)
   naredba1
else
   naredba2
```

[4] Pridržavamo se preporuke dobrog stila Java programairanja da se znak `{`, koji označava početak bloka naredbi, piše na kraju reda iza kojeg sledi, a odgovarajući znak `}` da se piše propisno poravnat u novom redu.

Izvršavanje naredbe `if-else` je slično izvršavanju naredbe `if`, osim što u slučaju kada je vrednost logičkog izraza u zagradi jednaka netačno, izvršava se *naredba*$_2$ i preskače *naredba*$_1$. U drugom slučaju, kada je vrednost logičkog izraza jednaka tačno, izvršava se *naredba*$_1$ i preskače *naredba*$_2$. Time se u oba slučaja izvršavanje naredbe `if-else` završava, a izvršavanje programa se dalje nastavlja od naredbe koja sledi iza naredbe `if-else`.

Obratite pažnju na to da se kod naredbe `if-else` izvršava tačno jedna od dve naredbe unutar naredbe `if-else`. Te naredbe predstavljaju alternativne tokove izvršavanja programa koji se biraju zavisno od vrednosti logičkog izraza.

Naravno, *naredba*$_1$ i *naredba*$_2$ u opštem obliku naredbe `if-else` mogu biti blokovi naredbi. Na primer:

```
if (r < 0)
   System.out.println("Greška: poluprečnik je negativan!");
else {
   obim = 2 * r * Math.PI;
   System.out.println("Obim kruga je: " + obim);
}
```

U ovom primeru naredbe `if-else` se u `if` delu nalazi samo jedna naredba i zato ta naredba ne mora (ali može) da bude unutar para vitičastih zagrada.[5] S druge strane, `else` deo se sastoji od dve naredbe i zato se mora nalaziti unutar para vitičastih zagrada u formi bloka.

Ne samo da sastavni delovi obične naredbe `if` ili naredbe `if-else` (ili bilo koje složene naredbe) mogu biti blokovi naredbi, nego ti delovi mogu biti i bilo koja naredbe jezika Java. Specifično, ti delovi mogu biti druge naredbe `if` ili `if-else`. Jedan primer ove mogućnosti koja se naziva *ugnježđavanje* jeste:

[5] Neki programeri uvek koriste blokove za sastavne delove složenih naredbi, čak i kada se ti delovi sastoje od samo jedne naredbe.

```
if (logički-izraz1)
   naredba1
else
   if (logički-izraz2)
      naredba2
   else
      naredba3
```

U ovom primeru se u `else` delu prve naredbe `if-else` nalazi druga naredba `if-else`. Ali zbog slobodnog formata, ovo se skoro uvek piše u drugom obliku:

```
if (logički-izraz1)
   naredba1
else if (logički-izraz2)
   naredba2
else
   naredba3
```

Ovaj drugi oblik sa klauzulom „`else if`" je bolji jer je razumljiviji, naročito u slučaju potrebe za većim brojem nivoa ugnježđavanja radi formiranja naredbe višestrukog grananja:

```
if (logički-izraz1)
   naredba1
else if (logički-izraz2)
   naredba2
else if (logički-izraz3)
   naredba3
⋮
else if (logički-izrazn)
   naredban
else
   naredban+1
```

Izvršavanje ove naredbe višestrukog grananja se izvodi tako što se najpre logički izrazi u zagradama izračunavaju redom odozgo na dole dok se ne dobije prvi koji daje vrednost tačno. Zatim se izvršava njegova pridružena naredba i preskače se sve ostalo. Ukoliko vrednost nijednog logičkog izraza nije tačno, izvršava se naredba u poslednjem `else` delu. U stvari, ovaj

`else` deo na kraju nije obavezan, pa ako nije naveden i nijedan logički izraz nema vrednost tačno, nijedna od *n* naredbi se ni ne izvršava.

Naglasimo još da svaka od naredbi unutar ove naredbe višestrukog grananja može biti blok koji se sastoji od niza naredbi između vitičastih zagrada. To naravno ne menja ništa konceptualno što se tiče izvršavanja naredbe višestrukog grananja, osim što se izvršava niz naredbi pridružen prvom logičkom izrazu koji ima vrednost tačno.

Jedan primer upotrebe naredbe višestrukog grananja je sledeći programski fragment u kojem se određuje ocena studenta na ispitu na osnovu broja osvojenih poena i „standardne" skale:

```
// Podrazumeva se 0 <= brojPoena <= 100
if (brojPoena >= 91)
   ocena = 10;
else if (brojPoena >= 81)
   ocena = 9;
else if (brojPoena >= 71)
   ocena = 8;
else if (brojPoena >= 61)
   ocena = 7;
else if (brojPoena >= 51)
   ocena = 6;
else
   ocena = 5;
```

Na kraju, obratimo pažnju na jedan problem u vezi sa ugnježđavanjem koji se popularno naziva „viseći" `else` deo. Razmotrimo sledeći programski fragment:

```
if (x >= 0)
   if (y >= 0)
      System.out.println("Prvi slučaj");
else
   System.out.println("Drugi slučaj");
```

Ovako kako je napisan ovaj fragment, izgleda kao da se radi o jednoj naredbi `if-else` čiji se `if` deo sastoji od druge naredbe `if`. Međutim, kako uvlačenje redova nema značaja za Java prevodilac i kako u Javi važi pravilo da se `else` deo uvek vezuje za najbliži prethodni `if` deo koji već nema svoj

`else` deo, pravi efekat prethodnog fragmenta je kao da je napisano:

```
if (x >= 0)
   if (y >= 0)
      System.out.println("Prvi slučaj");
   else
      System.out.println("Drugi slučaj");
```

Efekat ovog fragmenta i sugerisana prvobitna interpretacija nisu ekvivalentni. Ukoliko promenljiva `x` ima vrednost manju od 0, efekat prvobitne interpretacije je prikazivanje teksta „`Drugi slučaj`" na ekranu. Ali pravi efekat za `x` manje od 0 jeste zapravo da se preskače ugnježđena `if-else` naredba, odnosno ništa se ne prikazuje na ekranu.

Ukoliko se zaista želi efekat koji sugeriše prvobitna interpretacija, on se može obezbediti na dva načina. Prvo, ugnježđena naredba `if` se može pisati unutar bloka:

```
if (x >= 0) {
   if (y >= 0)
      System.out.println("Prvi slučaj");
}
else
   System.out.println("Drugi slučaj");
```

Drugo rešenje je da se ugnježđena naredba `if` piše u obliku naredbe `if-else` čiji se `else` deo sastoji od takozvane *prazne naredbe* označene samo tačka-zapetom:

```
if (x >= 0)
   if (y >= 0)
      System.out.println("Prvi slučaj");
   else
      ;  // prazna naredba
else
   System.out.println("Drugi slučaj");
```

Naredba `switch`

Treća naredba grananja, naredba `switch`, na neki način je specijalni slučaj naredbe višestrukog grananja iz prethodnog odeljka. Naime, u naredbi `switch` se izbor jednog od više alternativnih blokova naredbi za izvršavanje izvodi na osnovu jednog celobrojnog ili znakovnog izraza, a ne na osnovu logičkih izraza u višestruko ugnježđenim naredbama `if`.[6] Najčešći oblik naredbe `switch` je:

```
switch (izraz) {
   case konstanta_1:  // izvrši odavde ako izraz == konstanta_1
      niz-naredbi_1
      break;          // kraj izvršavanja prvog slučaja
   case konstanta_2:  // izvrši odavde ako izraz == konstanta_2
      niz-naredbi_2
      break;          // kraj izvršavanja drugog slučaja
   ⋮                  ⋮
   case konstanta_n:  // izvrši odavde ako izraz == konstanta_n
      niz-naredbi_n
      break;          // kraj izvršavanja n-tog slučaja
   default:           // izvrši odavde u krajnjem slučaju
      niz-naredbi_n+1
}                     // kraj izvršavanja krajnjeg slučaja
```

Ovom naredbom `switch` se najpre izračunava izraz u zagradi na početku naredbe. Zatim se zavisno od dobijene vrednosti tog izraza izvršava niz naredbi koji je pridružen jednom od slučajeva označenih klauzulama `case` unutar `switch` naredbe. Pri tome se redom odozgo na dole traži prva konstanta uz klauzulu `case` koja je jednaka vrednosti izračunatog izraza i izvršava se odgovarajući niz naredbi pridružen pronađenom slučaju. Poslednji slučaj u naredbi `switch` može opciono biti označen klauzulom `default`, a taj slučaj se izvršava ukoliko izračunata vrednost izraza nije jednaka nijednoj konstanti uz klauzule `case`. Najzad, ukoliko klauzula `default` nije navedena i izračunata vrednost izraza nije jednaka nijednoj konstanti uz klauzule `case`, ništa se dodatno ne izvršava nego se izvršavanje `switch` naredbe time odmah završava.

[6] Izraz u naredbi `switch` može biti i nabrojivog tipa. O nabrojivim tipovima podataka se govori u poglavlju 5.

Na kraju svakog slučaja naredbe `switch` obično se, ali ne uvek, navodi naredba `break`. (O naredbi `break` se više govori na strani 34 u ovom poglavlju.) Rezultat naredbe `break` je prekid izvršavanja odgovarajućeg slučaja, a time i cele naredbe `switch`. Ako se na kraju nekog slučaja ne nalazi naredba `break`, prelazi se na izvršavanje narednog slučaja unutar naredbe `switch`. Ovaj prelazak na naredni slučaj se nastavlja sve dok se ne naiđe na prvu naredbu `break` ili dok se ne dođe do kraja naredbe `switch`.

Iza zapisa `case` $konstanta_i$`:` u nekom od slučajeva naredbe `switch` mogu se potpuno izostaviti niz naredbi i naredba `break`. Ako iza tog „praznog" slučaja dolazi drugi „pravi" slučaj označen sa `case` $konstanta_j$`:`, onda niz naredbi pridružen ovom drugom slučaju odgovara zapravo dvema konstantama $konstanta_i$ i $konstanta_j$. To prosto znači da će se pridruženi niz naredbi izvršiti kada je vrednost izraza naredbe `switch` jednaka bilo kojoj od te dve konstante.

U sledećem primeru su pokazane neke od prethodno pomenutih mogućnosti naredbe `switch`. Primetite da se konstante uz klauzule `case` mogu pisati u proizvoljnom redosledu, pod uslovom da su sve različite. Takođe, ako jedan slučaj odgovara za više konstanti, onda se radi bolje istaknutosti odgovarajuće klauzule `case` mogu pisati u jednom redu.

```
// n je celobrojna promenljiva
switch (2 * n) { // vrednost izraza 2*n je uvek paran broj
   case 1:
      System.out.println("Ovo se nikad neće izvršiti,");
      System.out.println("jer je 2*n paran broj!");
      break;
   case 2:
   case 4:
   case 8:
      System.out.println("n ima vrednost 1, 2 ili 4,");
      System.out.println("jer je 2*n jednako 2, 4 ili 8.");
      break;
   case 6:
      System.out.println("n ima vrednost 3.");
      break;
   case 5: case 3:
      System.out.println("Ovo se prikazuje ako 2*n daje");
```

```
            System.out.println("5 ili 3. Ali to je nemoguće!");
            break;
        }
```

Naredba `while`

Naredba `while` je osnovna naredba za ciklično izvršavanja bloka naredbi više puta. Pošto ciklično izvršavanje bloka naredbi obrazuje petlju u sledu izvršavanja naredbi programa, naredba `while` se često naziva i `while` petlja. Blok naredbi unutar `while` petlje čije se izvršavanje ciklično ponavlja naziva se *telo petlje*.

Broj ponavljanja tela petlje se kontroliše vrednošću jednog logičkog izraza. Ovaj logički izraz se naziva *uslov nastavka* (ili *uslov prekida*) petlje, jer se izvršavanje tela petlje ponavlja sve dok je vrednost tog logičkog izraza jednaka tačno, a ovo ponavljanje se prekida u trenutku kada vrednost tog logičkog izraza postane netačno. Ponovljeno izvršavanje tela petlje se može, u principu, izvoditi beskonačno, ali takva *beskonačna petlja* obično predstavlja grešku u programu.

Osnovni oblik pisanja naredbe `while` je:

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

Izvršavanje naredbe `while` se izvodi tako što se najpre izračunava vrednost logičkog izraza u zagradi. U jednom slučaju, ako je dobijena vrednost jednaka netačno, odmah se prekida izvršavanje naredbe `while`. To znači da se preskače ostatak naredbe `while` i izvršavanje programa se normalno nastavlja od naredbe koja sledi iza naredbe `while`. U drugom slučaju, ako izračunavanje logičkog izraza daje tačno, najpre se izvršava naredba unutar naredbe `while`, a zatim se ponovo izračunava logički izraz u zagradi i ponavlja isti ciklus. To jest, ako logički izraz daje netačno, prekida se izvršavanje naredbe `while`; ako logički izraz daje tačno, ponovo se izvršava naredba unutar naredbe `while`, opet se izračunava logički izraz u zagradi i zavisno od njegove vrednosti dalje se opet ili ponavlja ovaj postupak ili se prekida izvršavanje naredbe `while`.

Telo petlje koje čini naredba unutar naredbe `while` može biti, a obično i jeste, blok naredbi, pa naredba `while` ima često ovaj oblik:

```
while (logički-izraz) {
   niz-naredbi
}
```

Efekat naredbe `while` je dakle ponavljanje jedne naredbe ili niza naredbi u bloku sve dok je vrednost logičkog izraza jednaka tačno. U trenutku kada ta vrednost postane netačno, naredba `while` se završava i nastavlja se normalno izvršavanje programa od naredbe koja sledi iza naredbe `while`.

U sledećem jednostavnom primeru naredbe `while` se na ekranu prikazuje niz brojeva 0, 1, 2, ..., 9 u jednom redu:

```
int broj = 0;
while (broj < 10) {
   System.out.print(broj + " ");
   broj = broj + 1;
}
System.out.println(); // pomeranje kursora u novi red
```

U ovom primeru se na početku celobrojna promenljiva `broj` inicijalizuje vrednošću 0. Zatim se izvršava `while` naredba, kod koje se najpre određuje vrednost logičkog izraza `broj < 10`. Vrednost ovog izraza je tačno, jer je trenutna vrednost promenljive `broj` jednaka 0 i, naravno, broj 0 je manji od broja 10. Zato se izvršavaju dve naredbe bloka u telu petlje, odnosno na ekranu se prikazuje vrednost 0 promenljive `broj` i ta vrednost se uvećava za 1. Nova vrednost promenljive `broj` je dakle 1. Sledeći korak kod izvršavanja naredbe `while` je ponovno izračunavanje logičkog izraza `broj < 10`. Vrednost ovog izraza je opet tačno, jer je trenutna vrednost promenljive `broj` jednaka 1 i, naravno, broj 1 je manji od broja 10. Zato se opet izvršavaju dve naredbe u telu petlje, odnosno na ekranu se prikazuje trenutna vrednost 1 promenljive `broj` i ta vrednost se uvećava za 1. Nova vrednost promenljive `broj` je dakle 2. Pošto izvršavanje naredbe `while` još nije završeno, ponovo se izračunava logički izraz `broj < 10`. Opet se dobija da je njegova vrednost jednaka tačno, jer je trenutna vrednost promenljive `broj` jednaka 2 i, naravno, broj 2 je manji od broja 10. Zato se opet izvršava telo petlje, izračunava se uslov prekida petlje i tako dalje.

U svakoj iteraciji petlje se prikazuje aktuelna vrednost promenljive `broj` i ta vrednost se uvećava za 1, pa je jasno je da će se to ponavljati sve dok promenljiva `broj` ne dobije vrednost 10. U tom trenutku će logički izraz `broj < 10` imati vrednost netačno, jer 10 nije manje od 10, a to predstavlja uslov prekida izvršavanja naredbe `while`. Nakon toga se izvršavanje programa nastavlja od naredbe koja se nalazi iza naredbe `while`, što u ovom primeru dovodi do pomeranja kursora na ekranu u novi red.

Potrebno je na kraju razjasniti još neke detalje u vezi sa naredbom `while`. Prvo, šta se događa ako je vrednost logičkog izraza naredbe `while` jednaka netačno pri prvom njegovom izračunavanju, kada se telo petlje još nijedanput nije izvršilo? U tom slučaju, izvršavanje naredbe `while` se odmah završava i zato se telo petlje nikad ni ne izvršava. To znači da broj iteracija naredbe `while` može biti proizvoljan, uključujući nulu.

Drugo, šta se događa ako vrednost logičkog izraza naredbe `while` postane netačno negde u sredini izvršavanja tela petlje? Da li se naredba `while` odmah tada završava? Odgovor je negativan, odnosno telo petlje se izvršava do kraja. Tek kada se zatim ponovo izračuna logički izraz i dobije njegova netačna vrednost, dolazi do završetka izvršavanja naredbe `while`.

Treće, u telu naredbe `while` se mogu nalaziti proizvoljne naredbe, pa tako i druge petlje. Ako se jedna petlja nalazi unutar tela druge petlje, onda se govori o *ugnježđenim* petljama.

Naredba `do-while`

Kod naredbe `while` se uslov prekida petlje proverava na početku svake iteracije. U nekim slučajevima, međutim, prirodnije je proveravati taj uslov na kraju svakog izvršavanja tela petlje. U takvim slučajevima se može koristiti naredba `do-while` koja je vrlo slična naredbi `while`, osim što su reč `while` i uslov prekida pomereni na kraj, a na početku se nalazi službena reč `do`. Osnovni oblik naredbe `do-while` je:

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

ili, ako se blok naredbi nalazi unutar naredbe `do-while`, njen opšti oblik je:

```
do {
   niz-naredbi
} while (logički-izraz);
```

Na primer, prikazivanje brojeva 0, 1, 2, ..., 9 u jednom redu na ekranu može se postići naredbom `do-while` na sledeći način:

```
int broj = 0;
do {
   System.out.print(broj + " ");
   broj = broj + 1;
} while (broj < 10);
System.out.println();  // pomeranje kursora u novi red
```

Obratite pažnju na tačka-zapetu koja se nalazi na samom kraju opšteg oblika naredbe `do-while`. Tačka-zapeta je deo naredbe `do-while` i ne može se izostaviti. (Generalno, na kraju svake naredbe u Javi se mora nalaziti tačka-zapeta ili zatvorena vitičasta zagrada.)

Kod izvršavanja naredbe `do-while` se najpre izvršava telo petlje, odnosno izvršava se naredba ili niz naredbi u bloku između službenih reči `do` i `while`. Zatim se izračunava vrednost logičkog izraza u zagradi. U jednom slučaju, ako je ta vrednost jednaka netačno, prekida se izvršavanje naredbe `do-while` i nastavlja se normalno izvršavanje programa od naredbe koja sledi iza naredbe `do-while`. U drugom slučaju, ako izračunavanje logičkog izraza daje tačno, izvršavanje se vraća na početak tela petlje i ponavlja se ciklus od početka.

Primetimo da, pošto se logički izraz izračunava na kraju iteracije svakog ciklusa, telo petlje se kod naredbe `do-while` izvršava bar jedanput. To se razlikuje od naredbe `while` kod koje se telo petlje ne mora uopšte izvršiti. Napomenimo i to da se naredba `do-while` može uvek simulirati naredbom `while` (a i obrnuto), jer je efekat osnovne naredbe `do-while` ekvivalentan sledećem programskom fragmentu:

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

Naredba `for`

Treća vrsta petlji u Javi se obično koristi kada je potrebno izvršiti telo petlje za sve vrednosti određene promenljive u nekom intervalu. Za prikazivanje brojeva 0, 1, 2, ..., 9 u jednom redu na ekranu, na primer, koristili smo naredbe `while` i `do-while`. U tim naredbama se zapravo telo petlje sastoji od prikazivanja promenljive `broj`, a ovo telo petlje se izvršava za sve vrednosti promenljive `broj` u intervalu od 0 do 9. U ovom slučaju je mnogo prirodnije koristiti naredbu `for`, jer je odgovarajući ekvivalentni programski fragment mnogo kraći i razumljiviji:

```
for (int broj = 0; broj < 10; broj = broj + 1)
   System.out.print(broj + " ");
System.out.println(); // pomeranje kursora u novi red
```

Opšti oblik naredbe `for` je:

```
for (kontrolni-deo)
   naredba
```

U posebnom slučaju, ako se telo naredbe `for` sastoji od bloka naredbi, njen opšti oblik je:

```
for (kontrolni-deo) {
   niz-naredbi
}
```

Kod naredbe `for` je kontrolnim delom na početku naredbe obuhvaćeno na jednom mestu sve što je bitno za upravljanje petljom, a to doprinosi čitljivosti i boljem razumevanju logike petlje. Ovaj kontrolni deo je dalje podeljen u tri dela koji se međusobno razdvajaju tačka-zapetom:

```
inicijalizacija; logički-izraz; završnica
```

Prema tome, pun oblik pisanja naredbe `for` je zapravo:

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

Efekat izvršavanja ovog oblika naredbe `for` se može predstaviti sledećim programskim fragmentom u kome se koristi naredba `while`:

```
inicijalizacija
while (logički-izraz) {
   naredba
   završnica
}
```

Kod izvršavanja naredbe `for` dakle, na samom početku se izvršava deo *`inicijalizacija`* konrolnog dela, i to samo jedanput. Dalje se uslov nastavka `for` petlje predstavljen logičkim izrazom izračunava pre svakog izvršavanja tela `for` petlje, a izvršavanje petlje se prekida kada izračunata vrednost logičkog izraza jeste netačno. Deo *`završnica`* konrolnog dela se izvršava na kraju svakog izvršavanja tela `for` petlje, neposredno pre ponovne provere uslova nastavka petlje.

Deo *`inicijalizacija`* može biti bilo koji izraz, mada je to obično naredba dodele. Deo *`završnica`* može takođe biti bilo koji izraz, mada je to obično naredba inkrementiranja, dekrementiranja ili dodele. Interesantno je primetiti da svaki od tri dela u kontrolnom delu naredbe `for` može biti prazan (ali se tačka-zapete ne mogu izostaviti). Ako je logički izraz u kontrolnom delu izostavljen, podrazumeva se da umesto njega stoji `true`. Zato naredba `for` sa „praznim" logičkim izrazom obrazuje beskonačnu petlju.

U kontrolnom delu naredbe `for`, u delu *`inicijalizacija`* obično se nekoj promenljivi dodeljuje početna vrednost, a u delu *`završnica`* se vrednost te promenljive uvećava ili smanjuje za određen korak. Pri tome se vrednost te promenljive dodatno proverava u logičkom izrazu radi nastavka ili prekida petlje. Promenljiva koja se na ovaj način koristi u naredbi `for` se naziva *kontrolna promenljiva* ili kratko *brojač* petlje.

Kako *`inicijalizacija`* tako i *`završnica`* u kontrolnom delu naredbe `for` mogu se sastojati od nekoliko izraza razdvojenih zapetama. Tako se u sledećem primeru istovremeno prikazuju brojevi od 0 do 9 i od 9 do 0 u dve kolone:

```
for (int n = 0, int m = 9; n < 10; n++, m--) {
   System.out.printf("%5d", n); // kolona širine 5 mesta za n
   System.out.printf("%5d", m); // kolona širine 5 mesta za m
   System.out.println();        // prelazak kursora u novi red
}
```

Naredbe `break` i `continue`

Kod `while` petlje i `do-while` petlje se uslov prekida tih petlji proverava na početku odnosno na kraju svake iteracije. Slično, iteracije `for` petlje se završavaju kada se ispuni uslov prekida u kontrolnom delu te petlje. Ponekad je ipak prirodnije prekinuti neku petlju u sredini izvršavanja tela petlje. Za takve slučajeve se može koristiti naredba `break` koja ima jednostavan oblik:

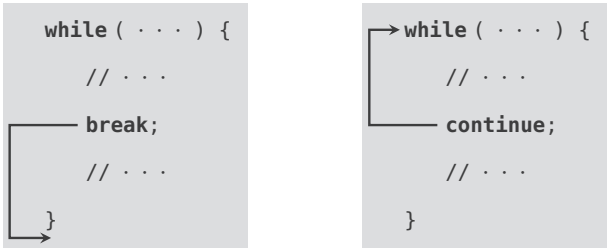
```
break;
```

Kada se izvršava naredba `break` unutar neke petlje, odmah se prekida izvršavanje te petlje i prelazi se na normalno izvršavanje ostatka programa od naredbe koja sledi iza petlje.

U petljama se slično može koristiti i naredba `continue` čiji je oblik takođe jednostavan:

```
continue;
```

Međutim, izvršavanjem naredbe `continue` se samo preskače ostatak aktuelne iteracije petlje. To znači da se, umesto prekidanja izvršavanja cele petlje kao kod naredbe `break`, naredbom `continue` prekida izvršavanje samo aktuelne iteracije petlje i nastavlja sa izvršavanjem naredne iteracije petlje (uključujući i izračunavanje uslova prekida petlje radi provere da li treba uopšte nastaviti izvršavanje petlje). Razlika u dejstvu naredbi `break` i `continue` na primeru `while` petlje ilustrovana je na slici 1.3.



Slika 1.3: Efekat naredbi `break` i `continue` u petljama.

S obzirom na to da se petlje mogu ugnježđavati jedna u drugu, ukoliko se naredba `break` nalazi u unutrašnjoj petlji, postavlja se pitanje koju petlju

prekida naredba `break` — unutrašnju ili spoljašnju? Pravilo je da naredba `break` uvek prekida izvršavanje samo prve obuhvatajuće petlje, ne i eventualnu spoljašnju petlju koja obuhvata petlju u kojoj se neposredno nalazi naredba `break`. Slično pravilo važi i za naredbu `continue`: ako se naredba `continue` nalazi u unutrašnjoj petlji, ona prekida aktuelnu iteraciju samo te petlje i nema nikakvog uticaja na eventualne obuhvatajuće petlje.

Ovakvo dejstvo naredbi `break` i `continue` u unutrašnjim petljama je prepreka da se prekine neki glavni postupak koji se sastoji od više ugnježđenih petlji, a logičko mesto prekida je duboko ugnježđena petlja. Takvi slučajevi se često javljaju pri otkrivanju logičkih grešaka u programu kada više nema smisla nastaviti dalju obradu, nego treba potpuno prekinuti započeti postupak.

Da bi se izvršilo potpuno prekidanje spoljašnje petlje ili samo njene aktuelne iteracije, u Javi se mogu koristiti *označene petlje*. Oznaka petlje se navodi ispred bilo koje vrste petlje u formi imena sa dvotačkom. Na primer, početak označene `while` petlje sa oznakom `spolj_petlja` može biti:

```
spolj_petlja: while (...) . . .
```

Unutar tela ove označene `while` petlje, duboko u nekoj unutrašnjoj petlji može se zatim koristiti naredba

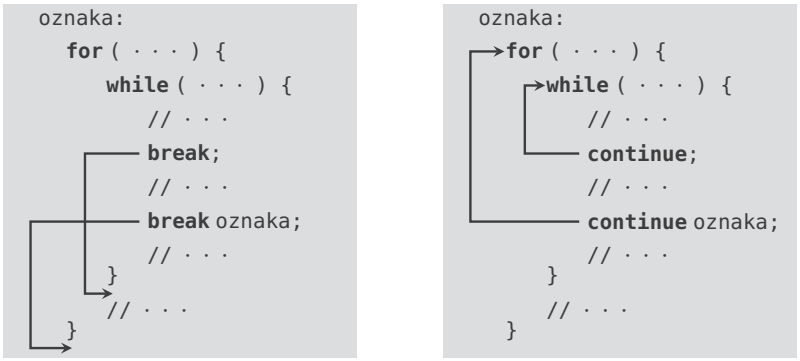
```
break spolj_petlja;
```

ili

```
continue spolj_petlja;
```

radi potpunog prekida ili prekida samo aktuelne iteracije petlje koja ima oznaku `spolj_petlja`. Efekat naredbi `break` i `continue` u ugnježđenim označenim i neoznačenim petljama ilustrovan je na slici 1.4.

Pored toga što se naredbe `break` i `continue` mogu koristiti u svakoj vrsti petlje (`while`, `do-while` i `for`), naredba `break` se može koristiti i za prekid izvršavanja naredbe `switch`. U stvari, naredba `break` se može koristili i unutar naredbe `if` (ili `if-else`), ali jedino u slučaju kada se sama naredba `if` nalazi unutar neke petlje ili naredbe `switch`. U tom slučaju, naredba `break` zapravo prekida izvršavanje obuhvatajuće petlje ili naredbe `switch`, a ne same naredbe `if`.



Slika 1.4: Efekat naredbi `break` i `continue` u ugnježđenim petljama.

1.3 Metodi

Uopšteno govoreći, metodi u Javi su potprogrami koje obavljaju neki specifični zadatak. Koncept potprograma postoji u svim programskim jezicima, ali se pojavljuje pod različitim imenima: metod, procedura, funkcija, rutina i slično. Termin koji se u Javi koristi za potprogram je „metod", u smislu postupak ili način za rešavanje nekog zadatka.

Metod je dakle samostalan, imenovan niz naredbi jezika Java koje se mogu koristiti u drugim delovima programa. Korišćenje metoda se tehnički naziva *pozivanje* metoda za izvršavanje. Svakim pozivom metoda se izvršava niz naredbi od kojih se metod sastoji. Pre početka izvršavanja ovog niza naredbi, metodu se mogu preneti neke vrednosti koje se nazivaju *argumenti* poziva metoda. Isto tako, na kraju izvršavanja ovog niza naredbi, metod može vratiti neku vrednost koja se naziva *rezultat* izvršavanja metoda.

Određena programska realizacija funkcionalnosti metoda naziva se *definicija* metoda. Definisanjem metoda se specificira niz naredbi i ostali elementi koji čine metod. Među ove elemente spadaju formalni *parametri* koji određuju koji se ulazni podaci kao argumenti mogu preneti prilikom pozivanja metoda za izvršavanje.

Definisanje metoda

Pre svega, važna osobina Jave je to što se definicija svakog metoda mora nalaziti unutar neke klase. Metod koji je član neke klase može biti statički (klasni) ili nestatički (objektni) čime se određuje da li metod pripada toj klasi kao celini ili pojedinačnim objektima te klase.

Definicija metoda se sastoji od zaglavlja i tela metoda. Telo metoda ima oblik običnog bloka naredbi, odnosno telo metoda se sastoji od niza proizvoljnih naredbi između vitičastih zagrada. Zaglavlje metoda sadrži sve informacije koje su neophodne za pozivanje metoda. To znači da se u zaglavlju metoda, između ostalog, nalazi:

- ime metoda;
- tipovi i imena parametara metoda;
- tip vrednosti koju metod vraća kao rezultat;
- modifikatori kojima se dodatno određuju karakteristike metoda.

Preciznije, opšti oblik definicije metoda u Javi je:

```
modifikatori tip-rezultata ime-metoda(lista-parametara)
{
   niz-naredbi
}
```

Na primer, definicija glavnog metoda `main()` u programu obično ima sledeći oblik:

```
public static void main (String[] args) { // Zaglavlje metoda
   .
   . // Telo metoda
   .
}
```

U zaglavlju ove definicije metoda `main()`, službene reči `public` i `static` su primeri modifikatora, `void` je tip rezultata, `main` je ime metoda i, na kraju, `String[] args` u zagradi čini listu od jednog parametra.

Deo *`modifikatori`* na početku zaglavlja nekog metoda nije obavezan, ali se može sastojati od jedne ili više službenih reči koje su međusobno razdvojene razmacima. Modifikatorima se određuju izvesne karakteristike metoda. Tako, u primeru metoda `main()`, službena reč `public` ukazuje

da se metod može slobodno koristiti u bilo kojoj drugoj klasi; službena reč `static` ukazuje da je metod statički, a ne objektni, i tako dalje. Za metode je u Javi na raspolaganju ukupno oko desetak modifikatora. O njima, kao i drugim elementima definicije metoda, govori se u kontekstu daljeg teksta knjige.

Ako metod izračunava jednu vrednost koja se vraća kao rezultat poziva metoda, onda deo *`tip-rezultata`* u njegovom zaglavlju određuje tip podataka kojem pripada rezultujuća vrednost metoda. S druge strane, ako metod ne vraća nijednu vrednost, onda se deo *`tip-rezultata`* sastoji od službene reči `void`. (Time se želi označiti da tip nepostojeće vraćene vrednosti predstavlja tip podataka koji ne sadrži nijednu vrednost.)

Formalna pravila za davanje imena metodu u delu *`ime-metoda`* jesu uobičajena pravila za imena u Javi o kojima smo govorili na strani 11. Dodajmo ovde samo to da je neformalna konvencija za imena metoda u Javi ista kao i za imena promenljivih: prva reč imena metoda počinje malim slovom, a ako se to ime sastoji od nekoliko reči, onda se svaka reč od druge piše sa početnim velikim slovom. Naravno, opšte nepisano pravilo je da imena metoda treba da budu smislena, odnosno da imena metoda dobro opisuju šta je funkcija pojedinih metoda.

Na kraju zaglavlja metoda, deo *`lista-parametara`* u zagradi određuje sve parametre metoda. Parametri su promenljive koje se inicijalizuju vrednostima argumenata prilikom pozivanja metoda za izvršavanje. Parametri predstavljaju dakle ulazne podatke metoda koji se obrađuju naredbama u telu metoda.

Lista parametara metoda može biti prazna (ali se zagrade moraju pisati), ili se ta lista može sastojati od jednog ili više parametara. Budući da parametar konceptualno predstavlja običnu promenljivu, za svaki parametar u listi navodi se njegov tip i njegovo ime u obliku:

```
tip-parametra ime-parametra
```

Ako lista sadrži više od jednog parametra, onda se parovi specifikacija tih parametara međusobno razdvajaju zapetama. Primetimo da svaki par za tip i ime parametra mora označavati samo jedan parametar. To znači da ako neki metod ima, recimo, dva parametra `x` i `y` tipa `double`, onda se u listi parametara mora pisati `double x, double y`, a ne skraćeno `double x, y`.

U nastavku su prikazani još neki primeri definicija metoda u Javi, bez naredbi u telu tih metoda kojima se realizuje njihova funkcija:

- ```
  public static void odigrajPotez() {
      .
      . // Telo metoda
      .
  }
  ```

  U ovom primeru su `public` i `static` modifikatori; `void` je tip rezultata, tj. metod ne vraća nijednu vrednost; `odigrajPotez` je ime metoda; lista parametara je prazna, tj. metod nema parametre.

- ```
  int nacrtaj2DSliku(int n, int m, String naslov) {
      .
      . // Telo metoda
      .
  }
  ```

  U ovom primeru nema modifikatora u zaglavlju metoda; tip vraćene vrednosti metoda je `int`; ime metoda je `nacrtaj2DSliku`; lista parametara sadrži tri parametra: `n` i `m` tipa `int` i `naslov` tipa `String`.

- ```
  static boolean manjeOd(float x, float y) {
      .
      . // Telo metoda
      .
  }
  ```

  U ovom primeru je `static` jedini modifikator; tip vraćene vrednosti je `boolean`; ime metoda je `manjeOd`; lista parametara sadrži dva parametra `x` i `y` tipa `float`.

Drugi metod `nacrtaj2DSliku()` u prethodnim primerima je objektni (nestatički) metod, jer njegovo zaglavlje ne sadrži službenu reč `static`. Generalno, metod se podrazumeva da je objektni ukoliko nije statički (što se određuje modifikatorom `static`). Modifikator `public` koji je korišćen u prvom primeru označava da je `odigrajPotez()` „javni" metod, odnosno da se može pozivati i izvan klase u kojoj je definisan, čak i u nekom drugom programu. Srodni modifikator je `private` kojim se označava da je metod
  ```

„privatan", odnosno da se može pozivati samo unutar klase u kojoj je definisan. Modifikatori `public` i `private` su takozvani *specifikatori pristupa* kojima se određuju ograničenja pristupa metodu, odnosno na kojim mestima se metod može koristiti. Treći, manje korišćeni specifikator pristupa je `protected` kojim se njegovo pozivanje ograničava na klasu u kojoj je definisan i sve njene izvedene klase. Ukoliko u zaglavlju metoda nije naveden nijedan specifikator pristupa (kao u drugom i trećem od prethodnih primera), onda se metod može pozivati u svakoj klasi istog paketa kome pripada klasa u kojoj je metod definisan, ali ne i izvan tog paketa.

Modifikatori se u zaglavlju metoda moraju pisati ispred tipa rezultata metoda, ali njihov međusobni redosled nije bitan. Konvencija je ipak da se najpre piše eventualni specifikator pristupa, zatim eventualno reč `static` i, na kraju, eventualno ostali modifikatori.

Pozivanje metoda

Definicijom novog metoda se uspostavlja njegovo postojanje i određuje kako on radi. Ali metod se uopšte ne izvršava sve dok se ne pozove na nekom mestu u programu — tek pozivom metoda se na tom mestu izvršava niz naredbi u telu metoda. (Ovo je tačno čak i za metod `main()` u nekoj klasi, iako se taj glavni metod ne poziva eksplicitno u programu, već ga implicitno poziva JVM na samom početku izvršavanja programa.)

Generalno, poziv nekog metoda u Javi može imati tri oblika. Najprostiji oblik naredbe poziva metoda je:

```
ime-metoda(lista-argumenata)
```

Prema tome, na mestu u programu gde je potrebno izvršavanje zadatka koji neki metod obavlja, navodi se samo njegovo ime i argumenti u zagradi koji odgovaraju parametrima metoda. Na primer, naredbom

```
nacrtaj2DSliku(100, 200, "Moja slika");
```

poziva se metod `nacrtaj2DSliku()` za izvršavanje sa argumentima koji su navedeni u zagradi.

Obratite pažnju na to da parametar u definiciji metoda mora biti neko *ime*, jer je parametar konceptualno jedna promenljiva. S druge strane, argument u pozivu metoda je konceptualno neka *vrednost* i zato argument

može biti bilo koji izraz čije izračunavanje daje vrednost tipa odgovarajućeg parametra. Zbog toga, pre izvršavanja naredbi u telu metoda, izračunavaju se izrazi koji su navedeni kao argumenti u pozivu metoda i njihove vrednosti se prenose metodu tako što se dodeljuju odgovarajućim parametrima. Pozivom metoda se dakle izvršava telo metoda, ali sa inicijalnim vrednostima parametara koje su prethodno dobijene izračunavanjem vrednosti odgovarajućih argumenata.

Na primer, poziv metoda

```
nacrtaj2DSliku(i+j, 2*k, s)
```

izvršava se tako što se izračunava vrednost prvog argumenta `i+j` i dodeljuje prvom parametru, zatim se izračunava vrednost drugog argumenta `2*j` i dodeljuje drugom parametru, zatim se izračunava vrednost trećeg argumenta `s` (čiji je rezultat aktuelna vrednost promenljive `s`) i dodeljuje se trećem parametru, pa se s tim početnim vrednostima paramatera najzad izvršava niz naredbi u telu metoda `nacrtaj2DSliku()`. Ovaj način koji se u Javi primenjuje za prenošenje vrednosti argumenata u pozivu nekog metoda odgovarajućim parametrima u definiciji metoda tehnički se naziva *prenošenje po vrednosti*.

Druga dva načina pozivanja metoda u Javi zavise od toga da li je metod definisan da bude statički (klasni) ili nestatički (objektni), odnosno da li je metod definisan s modifikatorom `static` ili bez njega. Ako je metod statički i poziva se izvan klase u kojoj je definisan, onda se mora koristiti tačka-notacija za njegov poziv:

```
ime-klase.ime-metoda(lista-argumenata)
```

Ovde je *`ime-klase`* ime one klase u kojoj je metod definisan.[7] Statički metodi pripadaju celoj klasi, pa upotreba imena klase u tačka-notaciji ukazuje na to u kojoj klasi treba naći definiciju metoda koji se poziva. Na primer, naredbom

```
Math.sqrt(x+y)
```

[7] Statički metod se može pozvati i preko nekog objekta klase u kojoj je metod definisan, odnosno na isti način kao što se nestatički (objektni) metod poziva. Međutim, takav način pozivanja statičkog metoda se ne preporučuje.

poziva se statički metod `sqrt()` koji je definisan u klasi `Math`. Argument koji se prenosi metodu `sqrt()` je vrednost izraza `x+y`, a rezultat izvršavanja tog metoda je kvadratni koren vrednosti njegovog argumenta.

Na kraju, treći način pozivanja metoda koristi se za nestatičke (objektne) metode koji pripadaju pojedinim objektima, a ne celoj klasi. Zbog toga se oni pozivaju uz odgovarajući objekat koji predstavlja implicitni argument poziva metoda. Prema tome, ako je metod nestatički (objektni) i poziva se izvan klase u kojoj je definisan, onda je opšti oblik tačka-notacije njegovog poziva:

```
ime-objekta.ime-metoda(lista-argumenata)
```

Ovde je *`ime-objekta`* zapravo jedna promenljiva koja sadrži referencu na onaj objekat za koji se poziva metod. Na primer, naredbom

```
rečenica.CharAt(i)
```

poziva se objektni metod `CharAt()` u klasi `String`, sa vrednošću promenljive `i` kao argumentom, za objekat klase `String` na koji ukazuje promenljiva `rečenica`. Rezultat ovog poziva je *i*-ti znak u stringu koji je referenciran promenljivom `rečenica`.

Vraćanje rezultata

Metod može vraćati jednu vrednost koja se dobija kao rezultat poziva metoda. Ako je to slučaj, onda vraćena vrednost metoda mora biti onog tipa koji je naveden kao tip rezultata u definiciji metoda. Druga mogućnost je da metod ne vraća nijednu vrednost. U tom slučaju se tip rezultata u definiciji metoda sastoji od službene reči `void`. Primetimo da metod u Javi može vraćati *najviše* jednu vrednost.

Pozivi metoda koji vraćaju jednu vrednost se navodi tamo gde je ta vrednost potrebna u programu. To je obično na desnoj strani znaka jednakosti kod naredbe dodele, ili kao argument poziva drugog metoda ili unutar nekog složenijeg izraza. Pored toga, poziv metoda koji vraća logičku vrednost može biti deo logičkog izraza kod naredbi grananja i ponavljanja. (Dozvoljeno je navesti poziv metoda koji vraća jednu vrednost i kao samostalnu naredbu, ali u tom slučaju se vraćena vrednost prosto zanemaruje.)

U definiciji metoda koji vraća jednu vrednost mora se navesti, pored tipa te vrednosti u zaglavlju metoda, bar jedna naredba u telu metoda čijim izvršavanjem se ta vrednost upravo vraća. Ta naredba se naziva *naredba povratka* i ima opšti oblik:

```
return izraz;
```

Ovde se tip vrednosti izraza iza službene reči `return` mora slagati sa navedenim tipom rezultata metoda koji se definiše.

U ovom obliku, naredba `return` ima dve funkcije: vraćanje vrednosti izraza i prekid izvršavanja pozvanog metoda. Preciznije, izvršavanje naredbe `return` *`izraz`* u telu pozvanog metoda se odvija u dva koraka:

1. Izračunava se navedeni izraz u produžetku na uobičajeni način.
2. Završava se izvršavanje pozvanog metoda i izračunata vrednost izraza i kontrola toka izvršavanja prenose se na mesto u programu gde je metod pozvan.

Da bismo bolje razumeli sve aspekte rada sa metodima, razmotrimo definiciju jednostavnog metoda za izračunavanje obima pravouglog trougla ukoliko su date obe katete trougla:

```
double obim(double a, double b) {
   double c = Math.sqrt(a*a + b*b); // hipotenuza trougla
   return a + b + c;                // obim trougla
}
```

Iz zaglavlja metoda `obim()` se može zaključiti da taj metod ima dva parametra `a` i `b` tipa `double` koji predstavljaju vrednosti kateta pravouglog trougla. Tip rezultata metoda `obim()` je takođe `double`, jer ako su katete realne vrednosti, onda i obim u opštem slučaju ima realnu vrednost. Primetimo i da nije naveden nijedan modifikator u zaglavlju ovog metoda, jer to ovde nije bitno i samo bi komplikovalo primer. (Podsetimo se da se u tom slučaju smatra da je metod objektni i da se može pozivati u svim klasama iz istog paketa u kome se nalazi klasa koja sadrži definiciju metoda.) U telu metoda `obim()` se najpre izračunava hipotenuza pravouglog trougla po Pitagorinoj formuli, a zatim se vraća rezultat metoda kao zbir kateta i hipotenuze pravouglog trougla.

Kao što je ranije napomenuto, definisanjem metoda se ništa ne izvršava, nego se tek pozivanjem metoda, u tački programa gde je potreban rezultat

metoda, izvršavaju naredbe u telu metoda uz prethodni prenos argumenata. Zbog toga, pretpostavimo da se na nekom mestu u jednom drugom metodu u programu izvršavaju sledeće dve naredbe:

```
double k1 = 3, k2 = 4;  // vrednosti kateta trougla
double O = obim(k1,k2); // poziv metoda obim()
```

Prvom naredbom deklaracije promenljivih `k1` i `k2` se, naravno, rezerviše memorijski prostor za promenljive `k1` i `k2` i u odgovarajućim memorijskim lokacijama se upisuju vrednosti, redom, 3 i 4. Drugom naredbom se rezerviše memorijski prostor za promenljivu `O`, a zatim joj se dodeljuje izračunata vrednost izraza koji se nalazi na desnoj strani znaka jednakosti. U okviru ovog izraza se nalazi poziv metoda `obim()` tako da je vrednost tog izraza, u stvari, vraćena vrednost poziva metoda `obim()`. Poziv tog metoda se dalje izvršava na način koji smo prethodno opisali: vrednosti argumenata u pozivu metoda se dodeljuju parametrima metoda u njegovoj definiciji, a sa tim inicijalnim vrednostima parametara se izvršava telo metoda u njegovoj definiciji.

U konkretnom slučaju dakle, argumenti poziva metoda `obim()` su promenljive `k1` i `k2`, pa se njihove trenutne vrednosti 3 i 4 redom dodeljuju parametrima ovog metoda `a` i `b`. Zatim se sa ovim inicijalnim vrednostima parametara `a` i `b` izvršava telo metoda `obim()`. Kontrola toka izvršavanja se zato prenosi u telo metoda `obim()` i redom se izvršavaju sve njegove naredbe dok se ne naiđe na naredbu `return`. To znači da se najpre izračunava hipotenuza izrazom `Math.sqrt(a*a + b*b)` čija se vrednost $\sqrt{3*3+4*4} = \sqrt{9+16} = \sqrt{25} = 5$ dodeljuje promenljivoj `c`. Zatim se izvršava naredba `return` tako što se izračunava izraz u produžetku `a + b + c` i dobija njegova vrednost 3+4+5 = 12. Nakon toga se prekida izvršavanje metoda `obim()` i izračunata vrednost 12 i kontrola izvršavanja se vraćaju tamo gde je taj metod pozvan. To znači da se nastavlja izvršavanje naredbe dodele `O = obim(k1,k2)` koje je bilo privremeno prekinuto radi izvršavanja poziva metoda `obim()` na desnoj strani znaka jednakosti. Drugim rečima, vraćena vrednost 12 tog poziva se konačno dodeljuje promenljivoj `O`.

Iako u ovom primeru to nije slučaj, obratite pažnju na to da generalno naredba `return` ne mora biti poslednja naredba u telu metoda. Ona se može pisati u bilo kojoj tački u telu metoda u kojoj se želi vratiti vrednost i završiti izvršavanje metoda. Ako se naredba `return` izvršava negde u sre-

dini tela metoda, izvršavanje metoda se odmah prekida. To znači da se sve eventualne naredbe iza `return` naredba preskaču i kontrola se odmah vraća na mesto gde je metod pozvan.

Naredba povratka se može koristiti i kod metoda koji ne vraća nijednu vrednost. Budući da tip rezultata takvog metoda mora biti „prazan" tip (tj. `void`) u zaglavlju metoda, naredba povratka u telu metoda u ovom slučaju ne sme sadržati nikakav izraz iza reči `return`, odnosno ima jednostavan oblik:

```
return;
```

Efekat ove naredbe je samo završetak izvršavanja nekog metoda koji ne vraća nijednu vrednost i vraćanje kontrole na mesto u programu gde je taj metod pozvan.

Upotreba naredbe povratka kod metoda koji ne vraća nijednu vrednost nije obavezna i koristi se kada izvršavanje takvog metoda treba prekinuti negde u sredini. Ako naredba povratka nije izvršena prilikom izvršavanja tela takvog metoda, njegovo izvršavanje se normalno prekida (i kontrola izvršavanja se vraća na mesto gde je metod pozvan) kada se dođe do kraja izvršavanja tela metoda. S druge strane, ukoliko metod vraća jednu vrednost, u njegovom telu se mora nalaziti bar jedna naredba povratka u punom obliku `return` *`izraz`*, makar to bila poslednja naredba u telu metoda. (Ako ih ima više od jedne, sve one moraju imati ovaj pun oblik, jer metod uvek mora vratiti jednu vrednost.)

Preopterećeni metodi

Da bi se neki metod mogao pozvati radi izvršavanja, mora se poznavati njegovo ime, kao i broj, redosled i tipovi njegovih parametara. Ove informacije se nazivaju *potpis* metoda. Na primer, metod

```
public void nekiMetod(int n, double x, boolean test) {
    .
    . // Telo metoda
    .
}
```

ima potpis:

```
nekiMetod(int, double, boolean)
```

Obratite pažnju na to da potpis metoda *ne* obuhvata imena parametara metoda, nego samo njihove tipove. To je zato što za pozivanje nekog metoda nije potrebno znati imena njegovih parametara, već se odgovarajući argumenti u pozivu mogu navesti samo na osnovu poznavanja tipova njegovih parametara. Primetimo i da potpis metoda ne obuhvata tip rezultata metoda, kao ni eventualne modifikatore u zaglavlju.

Definicija svakog metoda u Javi se mora nalaziti unutar neke klase, ali jedna klasa može sadržati definicije više metoda sa istim imenom, pod uslovom da svaki takav metod ima različit potpis. Metodi iz jedne klase sa istim imenom i različitim potpisom se nazivaju *preopterećeni* (engl. *overloaded*) metodi. Na primer, u klasi u kojoj je definisan prethodni metod `nekiMetod()` može se definisati drugi metod sa istim imenom, ali različitim potpisom:

```
public void nekiMetod(String s) {
   .
   . // Telo metoda
   .
}
```

Potpis ovog metoda je

```
nekiMetod(String)
```

što je različito od potpisa prvog metoda `nekiMetod()`.

Potpis dva metoda u klasi mora biti različit kako bi Java prevodilac tačno mogao da odredi koji metod se poziva. A to se lako određuje na osnovu broja, redosleda i tipova argumenata navedenih u pozivu metoda. Naredbom, na primer,

```
nekiMetod("super program");
```

očigledno se poziva druga, a ne prva, verzija metoda `nekiMetod()`, jer je u pozivu naveden samo jedan argument tipa `String`.

Rekurzivni metodi

Metodi u Javi mogu u svojoj definiciji da pozivaju sami sebe. Takav način rešavanja problema u programiranju se naziva rekurzivni način. Rekurzija predstavlja vrlo važnu i efikasnu tehniku za rešavanje složenih problema.

Razmotrimo, na primer, izračunavanje stepena x^n, gde je x neki realan broj različit od nule i n ceo broj veći ili jednak nuli. Mada se za ovaj problem može iskoristiti gotov metod `Math.pow(x,n)`, pošto stepen x^n predstavlja množenje broja x sa samim sobom n puta, nije teško napisati i sopstveni metod za izračunavanje stepena x^n. Naime, n-ti stepen broja x možemo dobiti uzastopnim množenjem broja x sa parcijalno izračunatim stepenima x^i za $i = 0, 1, \ldots, n-1$. Naredni metod `stepen()` koristi `for` petlju radi realizacije ovog iterativnog postupka:

```
double stepen(double x, int n) {
   double y = 1;  // rezultat
   for (int i = 0; i < n; i++)
      y = y * x;
   return y;
}
```

Pored ovog iterativnog načina, za izračunavanje stepena može se primeniti drugačiji (rekurzivni) pristup ukoliko se uoči da je $x^0 = 1$ i $x^n = x \cdot x^{n-1}$ za stepen n veći od nule. Drugim rečima, ako je stepen $n = 0$, rezultat je uvek 1, dok se za stepen $n > 0$ rezultat dobija ako se x pomnoži sa $(n-1)$-im stepenom broja x. Druga verzija metoda `stepen()` je definisana tako da se koristi ovaj način za izračunavanje n-tog stepena broja:

```
double stepen(double x, int n) {
   if (n == 0)
      return 1;
   else
      return x * stepen(x,n-1);
}
```

Obratite ovde posebnu pažnju na to da ovaj metod `stepen()`, radi izračunavanja n-tog stepena broja x, u naredbi `return` poziva sâm sebe radi izračunavanja $(n-1)$-og stepena broja x.

Rekurzivni metodi su oni koji pozivaju sami sebe, bilo direktno ili indirektno. Metod direktno poziva sâm sebe ako se u njegovoj definiciji nalazi poziv samog metoda koji se definiše. (Ovo je slučaj kod prethodnog metoda `stepen()`.) Metod indirektno poziva sâm sebe ako se u njegovoj definiciji nalazi poziv drugog metoda koji sa svoje strane poziva polazni metod koji se definiše (bilo direktno ili indirektno).

Rekurzivni metodi u opštem slučaju rešavaju neki zadatak svođenjem polaznog problema na sličan prostiji problem (ili više njih). Na primer, rekurzivni metod `stepen()` rešava problem izračunavanja n-tog stepena nekog broja svođenjem na problem izračunavanja $(n-1)$-og stepena istog broja. Prostiji zadatak (ili više njih) onda se rešava rekurzivnim pozivom metoda koji se definiše.

Važno je primetiti da rešavanje prostijeg zadatka rekurzivnim pozivom sa svoje strane dovodi do svođenja tog prostijeg zadatka na još prostiji zadatak, a ovaj se onda opet rešava rekurzivnim pozivim. Na primer, kod rekurzivnog metoda `stepen()` se problem izračunavanja $(n-1)$-og stepena broja svodi na problem izračunavanja $(n-2)$-og stepena tog broja. Naravno, ovaj proces uprošćavanja polaznog zadatka se dalje nastavlja i zato se mora obezbediti da se završi u određenom trenutku. U suprotnom, dobija se beskonačan lanac poziva istog metoda, što je programska graška slična beskonačnoj petlji.

Zbog toga se u telu svakog rekurzivnog metoda mora razlikovati *bazni slučaj* za najprostiji zadatak čije je rešenje unapred poznato i koje se ne dobija rekurzivnim pozivom. To je kod rekurzivnog metoda `stepen()` slučaj kada je stepen $n = 0$, jer se onda rezultat 1 neposredno dobija bez daljeg rekurzivnog rešavanja.

Rekurzivni metod generiše dakle lanac poziva za rešavanje niza prostijih zadataka sve dok se ne dođe do najprostijeg zadatka u baznom slučaju. Tada se lanac rekurzivnih poziva prekida i započeti pozivi se završavaju jedan za drugim u obrnutom redosledu njihovog pozivanja, odnosno složenosti zadataka koje rešavaju (od najprostijeg zadatka ka složenijem zadacima). Na taj način, na kraju se dobija rešenje najsloženijeg, polaznog zadatka.

Primer: Fibonačijev niz brojeva. Radi boljeg razumevanja osnovnih elemenata rekurzivne tehnike programiranja, razmotrimo poznat problem izračunavanja n-tog broja Fibonačijevog niza. Fibonačijev niz brojeva počinje sa prva dva broja koja su oba jednaka 1, a svaki sledeći broj u nizu se zatim dobija kao zbir prethodna dva broja niza. Tako, treći broj niza je zbir drugog i prvog, odnosno $1 + 1 = 2$; četvrti broj niza je zbir trećeg i drugog, odnosno $2 + 1 = 3$; peti broj niza je zbir četvrtog i trećeg, odnosno $3 + 2 = 5$; i tako dalje. Drugim rečima, početni deo beskonačnog Fibonačijevog niza brojeva je:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Ako su brojevi Fibonačijevog niza redom označeni slovima f_1, f_2, f_3, ... , onda je očigledno rekurzivna definicija za n-ti broj Fibonačijevog niza f_n određena sledećim jednakostima:

$$f_1 = 1,\ f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \quad n > 2$$

Drugim rečima, prva dva broja f_1 i f_2 su jednaki 1, a n-ti broj za $n > 2$ jednak je zbiru prethodna dva, odnosno $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza.

Ukoliko treba napisati metod za izračunavanje n-tog broja Fibonačijevog niza, prethodna rekurzivna definicija za izračunavanje tog broja može se neposredno iskoristiti za rekurzivno rešenje:

```
// Pretpostavlja se da je n ≥ 1
int fib(int n) {
   if (n <= 2)  // bazni slučaj
      return 1;
   else
      return fib(n-1) + fib(n-2);
}
```

U ovom metodu obratite pažnju na to da se zadatak izračunavanja n-tog broja Fibonačijevog niza svodi na dva slična prostija zadatka, odnosno izračunavanje $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza. Zato se pomoću dva rekurzivna poziva `fib(n-1)` i `fib(n-2)` izračunavaju vrednosti $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza i njihov zbir se kao rezultat vraća za n-ti broj Fibonačijevog niza.

1.4 Uvod u klase

Posle promenljivih, operatora, izraza, naredbi i metoda, sledeći viši nivo programske konstrukcije u Javi su klase. Klasa služi za opis objekata sa zajedničkim svojstvima koji se koriste u programu radi realizovanja funkcije programa. O ovoj glavnoj ulozi klasa na kojoj se zasniva objektno orijentisano programiranje opširno se govori u narednom poglavlju. Pre toga, međutim, potrebno je upoznati se sa nekim osnovnim pojmovima o klasama kako bi se mogli razumeti još neki detalji koji imaju veze sa prostijim programskim elementima o kojima smo govorili u prethodnom delu ovog poglavlja.

Definicije metoda nalaze se unutar klasa. Ali jedna klasa može pored definicija metoda obuhvatati i deklaracije promenljivih. Ove promenljive, da bi se bolje razlikovale od običnih promenljivih koje se deklarišu u metodima, nazivaju se i *polja*. Klasa je dakle jedna imenovana programska jedinica u kojoj se nalaze polja za čuvanje podataka i metodi za manipulisanje tim podacima. Kako klasa opisuje objekte koji joj pripadaju, definicijom neke klase se zapravo određuju polja i metodi koje poseduju svi objekti te klase.

U najprostijem slučaju, definicija klase ima sledeći oblik:

```
class ime-klase
{
   .
   . telo-klase
   .
}
```

Drugim rečima, iza službene reči `class` navodi se ime klase, a zatim se u vitičastima zagradama u telu klase navode definicije polja i metoda. Na primer, definicija klase kojom se opisuju objekti koji predstavljaju zgrade može imati ovaj izgled:

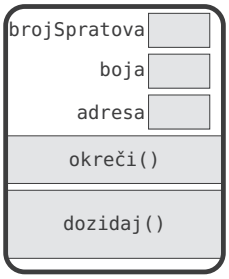
```
class Zgrada {

   int brojSpratova;
   String boja;
```

```
    String adresa;

    public void okreči(...) { ... };
    public void dozidaj(...) { ... };
}
```

Klasa `Zgrada` sadrži dakle tri polja `brojSpratova`, `boja` i `adresa`, kao i dva metoda `okreči()` i `dozidaj()`. Generalno, jedna klasa definišu šablon kako izgledaju pojedini objekti koji pripadaju toj klasi. Na primer, izgled svakog objekta klase `Zgrada` je prikazan na slici 1.5.



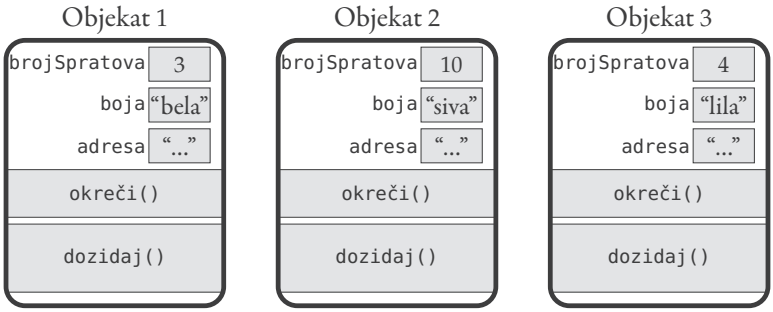
Slika 1.5: Jedan objekat klase `Zgrada`.

Polja i metodi u klasi se jednim imenom nazivaju *članovi* klase. Polja se u klasi definišu izvan svih metoda, pa posmatrano iz perspektive metoda u klasi, polja su *globalne* promenljive za sve metode klase. S druge strane, promenljive koje su deklarisane unutar nekog metoda su *lokalne* promenljive za taj metod. (Može se govoriti i o lokalnim promenljivima za blok, ukoliko su promenljive deklarisane unutar bloka između vitičastih zagrada.)

Isto kao metodi, polja mogu biti statička (klasna) ili nestatička (objektna). Statičko polje pripada celoj klasi, odnosno jedno jedino statičko polje dele svi objekti klase. Statičko polje se definiše pisanjem službene reči `static` ispred imena polja. Ukoliko se ispred imena polja ne nalazi modifikator `static`, takvo polje se podrazumeva da je objektno. Objektna polja pripadaju pojedinačnim objektima klase, odnosno svaki objekat klase ima svoju kopiju objektnog polja.

Na primer, sva polja u klasi `Zgrada` su objektna polja. Ako se u programu konstruišu, recimo, tri objekta te klase, svaki konstruisani objekat ima

svoje primerke objektnih polja i metoda koji se nalaze u klasi `Zgrada`. Ova činjenica ilustrovana je na slici 1.6.



Slika 1.6: Tri objekta klase `Zgrada`.

U opštem slučaju, deklaracija polja ima isti oblik kao deklaracija obične promenljive, uz dve razlike:

1. Deklaracija polja se mora nalaziti izvan svih metoda, ali naravno i dalje unutar neke klase.
2. Ispred tipa i imena polja se mogu nalaziti modifikatori `static`, `public` ili `private` koji bliže određuju vrstu polja.

Neki primeri deklaracija polja su:

```
static String imeKorisnika;
public int brojIgrača;
private static double brzina, vreme;
```

Modifikatorom `static` se polje određuje da bude statičko; ako se ovaj modifikator ne navede, podrazumeva se da je polje objektno. Modifikatori pristupa `public` i `private` određuju gde se polje može koristiti. Polje sa modifikatorom `private` („privatno polje") može se koristiti samo unutar klase u kojoj je deklarisano. Za polje sa modifikatorom `public` („javno polje") nema nikakvih ograničenja u pogledu njegovog pristupa, odnosno ono se može koristiti u bilo kojoj klasi.

Slično kao kod metoda, ako se javno polje koristi u drugoj klasi od one u kojoj je deklarisano, onda se mora koristiti tačka-notacija. Pri tome, ako je

polje statičko, taj zapis ima oblik *ime-klase.ime-polja*, dok se za objektna polja mora pisati *ime-objekta.ime-polja*. Ovde je *ime-klase* ona klasa u kojoj je javno statičko polje deklarisano, a *ime-objekta* je neki konstruisani objekat klase u kojoj je objektno polje deklarisano.

Dužina trajanja promenljivih

Svaka promenljiva ima svoj „životni vek" u memoriji tokom izvršavanja programa. Postojanje neke promenljive u programu počinje od trenutka izvršavanja naredbe deklaracije promenljive. Time se rezerviše memorijski prostor odgovarajuće veličine za smeštanje vrednosti promenljive. Ovaj postupak se naziva *alociranje* promenljive.

Postojanje promenljive u programu se završava kada se oslobodi memorijski prostor koji je rezervisan za promenljivu, moguće da bi se iskoristio za neke druge potrebe. Ovaj postupak se naziva *dealociranje* promenljive, a momenat njegovog dešavanja tokom izvršavanja programa zavisi od vrste promenljive. U telu nekog metoda se mogu koristiti tri vrste promenljivih: lokalne promenljive deklarisane unutar metoda, parametri metoda i globalne promenljive deklarisane izvan metoda, ali u istoj klasi kao i metod. (Ove globalne promenljive su naravno drugo ime za polja klase.)

Lokalne promenljive se dealociraju čim se izvrši poslednja naredba metoda u kojem su deklarisane i neposredno pre povratka na mesto gde je taj metod pozvan. Stoga lokalne promenljive postoje samo za vreme izvršavanja svog metoda, pa se zato ni ne mogu koristiti negde izvan svog metoda, jer tamo više ne postoje. Lokalne promenljive dakle nemaju nikakve veze sa okruženjem metoda, odnosno one su potpuno deo internog rada metoda.

Parametri metoda služe za prihvatanje ulaznih vrednosti metoda od argumenata kada se metod pozove. Kako parametri metoda imaju konceptualno istu ulogu kao lokalne promenljive, za parametre važi slično pravilo kao i za lokalne promenljive: paramatri metoda se alociraju u momentu poziva metoda za izvršavanje i dealociraju u momentu završetka izvršavanja metoda.

Globalna promenljiva metoda koja je deklarisane unutar neke klase postoji, grubo rečeno, od momenta kada se njena klasa prvi put koristi pa sve do kraja izvršavanja celog programa. Tačni trenuci alociranja i dealocira-

nja globalnih promenljivih nisu toliko bitni, nego je važno samo to što su globalne promenljive u nekoj klasi nezavisne od metoda te klase i uvek postoje dok se metodi te klase izvršavaju. Zato globalne promenljive dele svi metodi u istoj klasi. To znači da promene vrednosti globalne promenljive u jednom metodu imaju prošireni efekat na druge metode: ako se u jednom metodu promeni vrednost nekoj globalnoj promenljivoj, onda se u drugom metodu koristi ta nova vrednost globalne promenljive ukoliko ona učestvuje u, recimo, nekom izrazu. Drugim rečima, globalne promenljive su zajedničke za sve metode neke klase, za razliku od lokalnih promenljivih (i parametara) koje pripadaju samo jednom metodu.

Treba istaći još jedan detalj u vezi sa inicijalizacijom promenljivih prilikom njihovog alociranja. Lokalne promenljive metoda se u Javi ne inicijalizuju nikakvom vrednošću prilikom alociranja tih promenljivih na samom početku izvršavanja metoda. Stoga one moraju imati eksplicitno dodeljenu vrednost u metodu pre nego što se mogu koristiti. Parametri metoda se prilikom alociranja na početku izvršavanja metoda, naravno, inicijalizuju vrednostima argumenata. Na kraju, globalne promenljive (polja) se automatski inicijalizuju unapred definisanim vrednostima. Za numeričke promenljive, ta vrednost je nula; za logičke promenljive, ta vrednost je `false`; i za znakovne promenljive, ta vrednost je Unicode znak `'\u0000'`. (Za objektne promenljive, ta inicijalna vrednost je specijalna vrednost `null`.)

Oblast važenja imena

Programiranje se u svojoj suštini svodi na davanje imena. Neko ime u programu može da označava razne konstrukcije: promenljive, metode, klase i tako dalje. Ove konstrukcije se preko svojih imena koriste na raznim mestima u tekstu programu, ali u prethodnom odeljku smo pokazali da se, recimo, promenljive dinamički alociraju i dealociraju tokom izvršavanja programa. Teorijski je zato moguće da se neke programske konstrukcije koriste u tekstu programa iako one tokom izvršavanja programa fizički više ne postoje u memoriji. Da bi se izbegle ove vrste grešaka, u Javi postoje pravila o tome gde se uvedena (prosta) imena mogu koristiti u programu. Oblast teksta programa u kome se može upotrebiti neko ime, radi korišćenja programske konstrukcije koju označava, naziva se *oblast važenja imena*

(engl. *scope*).

Promenljivoj se ime daje u deklaraciji promenljive. Oblast važenja imena globalne promenljive (polja) je cela klasa u kojoj je globalna promenljiva definisana. To znači da se globalna promenljiva može koristiti u tekstu cele klase, čak i eventualno ispred naredbe njene deklaracije.[8] S druge strane, oblast važenja imena lokalne promenljive je od mesta njene deklaracije do kraja bloka u kome se njena deklaracija nalazi. Zato se lokalna promenljiva može koristiti samo u svom metodu, i to od naredbe svoje deklaracije do kraja bloka u kojem se nalazi ta naredba deklaracije.

Ova pravila za oblast važenja imena promenljivih su ipak malo složenija, jer je dozvoljeno lokalnoj promenljivoj ili parametru dati isto ime kao nekoj globalnoj promenljivoj. U tom slučaju, unutar oblasti važenja lokalne promenljive ili parametra, globalna promenljiva je *zaklonjena*. Da bismo ovo ilustrovali, razmotrimo klasu `IgraSaKartama` čija definicija ima sledeći oblik:

```
class IgraSaKartama {

   static String pobednik;  // globalna promenljiva (polje)

   static void odigrajIgru() {

      String pobednik;      // lokalna promenljiva
      .
      . // Ostale naredbe metoda ...
      .
   }
   . . .
}
```

U telu metoda `odigrajIgru()`, ime `pobednik` se odnosi na lokalnu promenljivu. U ostatku klase `IgraSaKartama`, ime `pobednik` se odnosi na globalnu promenljivu (polje), sem ako to ime nije opet zaklonjeno istim imenom lokalne promenljive ili parametra u nekom drugom metodu. Ako se ipak mora koristiti statičko polje `pobednik` unutar metoda `odigrajIgru()`, onda

[8] Zato je moguće pisati deklaracije polja na kraju definicije klase, što je po nekima bolji stil pisanja klasa.

se mora pisati njegovo puno ime `IgraSaKartama.pobednik`.

Ove komplikacije, kada lokalna promenljiva ili parametar imaju isto ime kao globalna promenljiva, uzrok su mnogih grešaka koje se najjednostavnije izbegavaju davanjem različitih imena radi lakšeg razlikovanja. To je upravo i preporuka dobrog stila programiranja koje se treba pridržavati.

Drugi specijalni slučaj oblasti važenja imena promenljivih pojavljuje se kod deklarisanja kontrolne promenljive petlje unutar `for` petlje:

```
for (int i = 0; i < n; i++) {
   .
   . // Telo petlje
   .
}
```

U ovom primeru bi se moglo reći da je promenljiva `i` deklarisana lokalno u odnosu na metod koji sadrži `for` petlju. Ipak, oblast važenja ovako deklarisane kontrolne promenljive je samo kontrolni deo i telo `for` petlje, odnosno ne proteže se do kraja tela metoda koji sadrži `for` petlju. Zbog toga se u Javi vrednost brojača `for` petlje ne može koristiti van te petlje. Na primer, ovo nije dozvoljeno:

```
for (int i = 0; i < n; i++) {
   .
   . // Telo petlje
   .
}
if (i == n)  // GREŠKA: ovde ne važi ime "i"
   System.out.println("Završene sve iteracije");
```

Oblast važenja imena parametra nekog metoda je blok od kojeg se sastoji telo tog metoda. Nije dozvoljeno redefinisati ime parametra ili lokalne promenljive u oblasti njihovog važenja, čak ni u ugnježđenom bloku. Na primer, ni ovo nije dozvoljeno:

```
void neispravanMetod(int n) {

   int x;
   while (n > 0) {
      int x;  // GREŠKA: ime "x" je već definisano
```

```
        . . .
    }
}
```

Prema tome, u Javi se globalne promenljive mogu redefinisati (ali su onda zaklonjene), dok se lokalne promenljive i parametri ne mogu redefinisati. S druge strane, izvan oblasti važenja lokalne promenljive ili parametra, njihova imena se mogu ponovo koristiti. Na primer:

```
void ispravanMetod(int n) {

   while (n > 0) {
      int x;
      . . .
   } // Kraj oblasti važenja imena "x"
   while (n > 0) {
      int x;  // OK: ime "x" se može opet davati
      . . .
   }
}
```

Pravilo za oblast važenja imena metoda je slično onom za globalne promenljive: oblast važenja nekog metoda je cela klasa u kojoj je metod definisan. To znači da je dozvoljeno pozivati metod na bilo kom mestu u klasi, uključujući tačke u programskom tekstu klase koje se nalaze ispred tačke definicije metoda. Čak je moguće pozivati neki metod i unutar definicije samog metoda. (U tom slučaju se radi, naravno, o rekurzivnim metodima.)

Ovo opšte pravilo ima dodatni uslov s obzirom na to da metodi mogu biti statički ili objektni. Naime, objektni članovi klase (metodi i polja), koji pripadaju nekom objektu klase, ne moraju postojati u memoriji dok se statički metod izvršava. Na primer, statički metod neke klase se može pozvati upotrebom njegovog punog imena u bilo kojoj tački programa, a do tada objekat kome pripadaju objektni članovi iste klase možda nije još bio ni konstruisan tokom izvršavanja programa. Zato se objektni metodi i polja klase ne mogu koristiti u statičkim metodima iste klase.

Paketi

Programski jezik Java (tačnije Java platforma) sadrži veliki broj unapred napisanih klasa koje se mogu koristiti u programima. Da bi se olakšalo nalaženje i upotreba tih gotovih klasa, one su grupisane u pakete, analogno organizaciji datoteka i direktorijuma u okviru sistema datoteka na disku.

Paket u Javi je dakle kolekcija srodnih klasa koje čine funkcionalnu celinu.[9] Glavne klase jezika Java nalaze se u paketima čija imena počinju prefiksom `java`. Na primer, najvažnije klase su deo paketa `java.lang`; razne pomoćne klase nalaze se u paketu `java.util`; klase za programski ulaz i izlaz su u paketu `java.io`; klase za mrežno programiranje su u paketu `java.net`; i tako dalje.

Poslednja verzija Jave sadrži preko 200 paketa i oko 4000 klasa! Sve pakete dakle ne vredi nabrajati, a pogotovo nije izvodljivo opisati sve klase u njima. Obično se namena paketa može prepoznati na osnovu njegovog imena, ali tačno poznavanje svake klase je praktično nemoguće. Zbog toga je neophodno da se Java programeri dobro snalaze u korišćenju zvanične dokumentacije jezika Java. Dokument u kome je detaljno opisana Java može se naći u elektronskoj formi besplatno na Internetu. Dokumentacija je napisana u formatu pogodnom za čitanje u nekom veb čitaču, tako da se jednostavnim izborom hiperveza u tekstu može relativno brzo pronaći ono što se traži.

Pored toga što olakšavaju nalaženje i upotrebu gotovih klasa, paketi imaju još dve dobre strane:

1. Paketima se sprečavaju sukobi imena klasa, jer različiti paketi mogu da sadrže klase sa istim imenom.

2. Paketima se omogućava dodatni vid kontrole nad upotrebom klasa.

Korišćenje paketa. Svaka klasa ima zapravo dva imena: prosto ime koje se daje u definiciji klase i puno ime koje dodatno sadrži ime paketa u kome se klasa nalazi. Na primer, za rad sa datumima u programu stoji na raspolaganju klasa `Date` koja se nalazi u paketu `java.util`, pa je njeno puno ime `java.util.Date`.

[9] Paketi sadrže i interfejse o kojima se govori u poglavlju 5.

Klasa se bez ograničenja može koristiti samo pod punim imenom, ali upotreba dugačkog imena klase dovodi do čestih slovnih grešaka i smanjuje čitljivost programa. Zbog toga postoji mogućnost „uvoženja" pojedinih klasa na početku programa. Iza toga se u programu može koristiti prosto ime klase bez imena paketa u kome se ona nalazi. Ova mogućnost se postiže pisanjem deklaracije `import` na početku teksta klase. Na primer, ukoliko je u klasi `NekaKlasa` potrebno koristi objekat klase `Date`, umesto dužeg pisanja

```
class NekaKlasa {
   . . .
   java.util.Date d = new java.util.Date();
   . . .
}
```

može se kraće pisati:

```
import java.util.Date;
class NekaKlasa {
   . . .
   Date d = new Date();
   . . .
}
```

Ako je potrebno koristiti više klasa iz istog ili različitih paketa, onda za svaku klasu treba navesti posebne deklaracije `import` jednu ispod druge. Bolja mogućnost je upotreba džoker-znaka `*`, čime se „uvoze" sve klase iz određenog paketa. Prethodni primer može se zato ekvivalentno napisati na sledeći način:

```
import java.util.*;
class NekaKlasa {
   . . .
   Date d = new Date();
   . . .
}
```

U stvari, to je način koji se najčešće koristi u programima, jer se time veštački ne povećava veličina programa, odnosno sve klase iz nekog paketa se fizički ne dodaju programu. Deklaracija `import` ukazuje samo na to u kojem paketu treba tražiti korišćene klase, što znači da se fizički dodaju samo

one klase koje se zaista pojavljuju u tekstu definicije nove klase.

Definisanje paketa. Svaka klasa u Javi mora pripadati nekom paketu. Kada se definiše nova klasa, ukoliko se drugačije ne navede, klasa automatski pripada jednom bezimenom paketu. Taj takozvani *anonimni* paket sadrži sve klase koje nisu eksplicitno dodate nekom imenovanom paketu.

Za dodavanje klase u imenovani paket koristi se deklaracija `package` kojom se određuje paket kome pripada klasa. Ova deklaracija se navodi na samom početku teksta klase, čak pre deklaracije `import`. Na primer, za dodavanje klase `NekaKlasa` u paket `nekipaket` piše se:

```
package nekipaket;
import java.util.*;
class NekaKlasa {
   .
   . // Telo klase
   .
}
```

Određivanje imenovanog paketa kome pripada nova klasa zahteva posebnu organizaciju datoteka sa tekstom klase. U prethodnom primeru, datoteka `NekaKlasa.java`, koja sadrži tekst klase `NekaKlasa`, mora se nalaziti u posebnom direktorijumu čije ime mora biti `nekipaket`. U opštem slučaju, ime posebnog direktorijuma mora biti isto kao ime paketa kojem klasa pripada. Pored toga, prevođenje i izvršavanje klase `NekaKlasa` se mora obaviti iz direktorijuma na prvom prethodnom nivou od direktorijuma `nekipaket`.

GLAVA 2

Klase

U prethodnom uvodnom poglavlju je samo ovlaš dotaknut centralni koncept klase u objektno orijentisanom programiranju. Naime pomenuto je samo onoliko o tome koliko je bilo potrebno radi razumevanja glavnih programskih celina od kojih se sastoje Java programi. Sve druge osnovne programske konstrukcije spomenute u tom poglavlju mogu se naći, uz malo izmenjenu terminologiju, i u klasičnim, proceduralnim programskim jezicima.

Objektno orijentisano programiranje (OOP) je noviji pristup za rešavanje problema na računaru kojim se rešenje opisuje na prirodan način koji bi se primenio i u svakodnevnom životu. U starijem, proceduralnom programiranju je programer morao da identifikuje računarski postupak za rešenje problema i da napiše niz programskih naredbi kojima se realizuje taj postupak. Naglasak u OOP nije na računarskim postupcima nego na objektima koji mogu da dejstvuju jedni na druge. Objektno orijentisano programiranje se zato svodi na projektovanje različitih klasa objekata kojima se na prirodan način modelira dati problem i njegovo rešenje. Pri tome, softverski objekti u programu mogu predstavljati ne samo realne, nego i apstraktne entitete iz odgovarajućeg problemskog domena.

Objektno orijentisane tehnike mogu se, u principu, koristiti u svakom proceduralnom programskom jeziku. To je posledica činjenice što se objekti iz standardnog ugla gledanja na stvari mogu smatrati običnom kolekcijom promenljivih i procedura koje manipulišu tim promenljivim. Međutim, za efektivnu realizaciju objektno orijentisanog načina rešavanja problema je vrlo važno imati jezik koji to omogućava na neposredan i elegantan

način.

Java je moderan objektno orijentisan programski jezik koji podržava sve koncepte objektno orijentisanog programiranja. Programiranje u Javi bez korišćenja njenih objektno orijentisanih mogućnosti nije svrsishodno, jer su drugi jezici lakši i pogodniji za proceduralno rešavanje problema na računaru. U ovom poglavlju se zato obraća posebna pažnja na detalje koncepta klase i njenog definisanja u Javi, kao i na potpuno razumevanje objekata i njihovog konstruisanja u programu.

2.1 Klase i objekti

Metodi su programske celine za obavljanje pojedinačnih specifičnih zadataka u programu. Na sledećem višem nivou složenosti su programske celine koje se nazivaju *klase*. Naime, jedna klasa može obuhvatati ne samo promenljive (polja) za čuvanje podataka, nego i više metoda za manipulisanje tim podacima radi obavljanja različitih zadataka. Pored ove organizacione uloge klasa u programu, druga njihova uloga je da opišu određene objekte čijim se manipulisanjem u programu ostvaruje potrebna funkcionalnost. Klase imaju i treću ulogu, jer se njima u programu formalno uvode novi tipovi podataka sa vrednostima koje su objekti tih klasa. U ovom odeljku se detaljnije govori o ovim višestrukim ulogama klasa.

Članovi klase

Jedna klasa obuhvata polja (globalne promenljive) i metode o kojima smo više govorili u prethodnom poglavlju. Doduše, polja klase smo uglavnom posmatrali iz perspektive metoda unutar klase i zato smo za polja koristili termin globalne promenljive. Budući da od ovog poglavlja polja smatramo sastavnim delovima klasa, potrebno je precizirati još neke osnovne činjenice o poljima. Definicija polja ima isti oblik kao definicija obične promenljive, uz dve razlike:

1. Definicija polja se mora nalaziti izvan svih metoda, ali naravno i dalje unutar neke klase.

2. Ispred tipa i imena polja se mogu nalaziti modifikatori kao što su `static`, `public` i `private`.

Neki primeri definicija polja su:

```
static String imeKorisnika;
public int brojIgrača;
private static double brzina, vreme;
```

Modifikatorom `static` se polje definiše da bude statičko; ako se ovaj modifikator ne navede, podrazumeva se da je polje objektno. Modifikatori pristupa `public` i `private` određuju gde se polje može koristiti. Polje sa modifikatorom `private` (privatno polje) može se koristiti samo unutar klase u kojoj je definisano. Za polje sa modifikatorom `public` (javno polje) nema nikakvih ograničenja u vezi sa njegovim korišćenjem, odnosno ono se može koristiti u bilo kojoj klasi.

Slično kao kod metoda, mora se pisati tačka-notacija kada se javno polje koristi u drugoj klasi od one u kojoj je definisano. Pri tome, za statička polja ovaj zapis ima oblik *`klasa.polje`*, dok se za objektna polja mora pisati *`objekat.polje`*. Ovde je *`klasa`* ime one klase u kojoj je javno statičko polje definisano, a *`objekat`* je ime klasne promenljive koja pokazuje na neki konstruisani objekat one klase u kojoj je objektno polje definisano.

* * *

U telu klase se dakle definišu polja i metodi koji se jednim imenom nazivaju *članovi* klase. Ovi članovi klase mogu biti statički (što se prepoznaje po modifikatoru `static` u definiciji člana) ili nestatički (objektni). S druge strane, govorili smo da klasa opisuje objekte sa zajedničkim osobinama i da svaki objekat koji pripada nekoj klasi ima istu strukturu koju čine polja i metodi definisani unutar njegove klase. Tačnije je reći da samo nestatički deo neke klase opisuje objekte koji pripadaju toj klasi.

Iz programerske perspektive, međutim, klasa služi za konstruisanje objekata. To znači da je jedna klasa zapravo šablon na osnovu koga se konstruišu njeni objekti. Pri tome, svaki konstruisan objekat neke klase sadrži sve nestatičke članove te klase. Za objekat konstruisan na osnovu definicije neke klase se kaže da *pripada* toj klasi ili da je *instanca* te klase. Nestatički članovi klase (polja i metodi), koji ulaze u sastav konstruisanih objekata, nazivaju se *objektni* ili *instancni* članovi.

Glavna razlika između klase i objekata je dakle to što se klasa *definiše* u tekstu programa, dok se objekti te klase *konstruišu* posebnim operatorom tokom izvršavanja programa. To znači da se klasa stvara prilikom prevođenja programa i zajedno sa svojim statičkim članovima postoji od početka do kraja izvršavanja programa. S druge strane, objekti se stvaraju dinamički tokom izvršavanja programa i zajedno sa nestatičkim članovima klase postoje od trenutka konstruisanja objekata, moguće negde u sredini programa, do trenutka kada više nisu potrebni.

Da bismo bolje razumeli ove osnovne koncepte klasa i objekata, posmatrajmo jednostavnu klasu koja može poslužiti za čuvanje osnovnih informacija o nekim korisnicima:

```
class Korisnik {
   static String ime;
   static int id;
}
```

Klasa `Korisnik` sadrži statička polja `Korisnik.ime` i `Korisnik.id`, pa u programu koji koristi ovu klasu postoji samo po jedan primerak promenljivh `Korisnik.ime` i `Korisnik.id`. Taj program može dakle modelirati situaciju u kojoj postoji samo jedan korisnik u svakom trenutku, jer postoji memorijski prostor za čuvanje podataka o samo jednom korisniku. Klasa `Korisnik` i njena dva statička polja `Korisnik.ime` i `Korisnik.id` postoje sve vreme izvršavanja programa.

Posmatrajmo sada sličnu klasu čija je definicija skoro identična:

```
class Klijent {
   String ime;
   int id;
}
```

U klasi `Klijent` su definisana nestatička (objektna) polja `ime` i `id`, pa u ovom slučaju ne postoje promenljive `Klijent.ime` i `Klijent.id`. Ova klasa dakle ne sadrži ništa konkretno, osim šablona za konstruisanje objekata te klase. Ali to je veliki potencijal, jer se taj šablon može iskoristiti za stvaranje velikog broja objekata klase `Klijent`. Svaki konstruisani objekat ove klase imaće *svoje* primerke polja `ime` i `id`. Zato program koji koristi ovu klasu može modelirati više klijenata, jer se po potrebi mogu konstruisati novi

objekti za predstavljanje novih klijenata. Ako je to, recimo, neki bankarski program, kada klijent otvori račun u banci može se konstruisati nov objekat klase `Klijent` za čuvanje podataka o tom klijentu. Kada neki klijent banke zatvori račun, objekat koji ga predstavlja u programu može se ukloniti. Stoga u tom programu kolekcija objekata klase `Klijent` na prirodan način modelira rad banke.

Primetimo da ovi primeri ne znače da klasa može imati samo statičke ili samo nestatičke članove. U nekim primenama postoji potreba za obe vrste članova klase. Na primer:

```
class ČlanPorodice {
   static String prezime;
   String ime;
   int uzrast;

   void čestitajRođendan() {
      uzrast++;
      System.out.println("Srećan " + uzrast + ". rođendan!")
   }
}
```

Ovom klasom se u programu mogu predstaviti članovi neke porodice. Pošto je prezime nepromenljivo za sve članove porodice, nema potrebe taj podatak vezivati za svakog člana porodice i zato se može čuvati u jedinstvenom statičkom polju `ČlanPorodice.prezime`. S druge strane, ime i uzrast su karakteristični za svakog člana porodice, pa se ti podaci moraju čuvati u nestatičkim poljima `ime` i `uzrast`.

Obratite pažnju na to da se u definicji klase određuju tipovi svih polja, i statičkih i nestatičkih. Međutim, dok se aktuelne vrednosti statičkih polja nalaze u samoj klasi, aktuelne vrednosti nestatičkih (objektnih) polja se nalaze unutar pojedinih objekata, a ne klase.

Slična pravila važe i za metode koji su definisani u klasi. Statički metodi pripadaju klasi i postoje za sve vreme izvršavanja programa. Zato se statički metodi mogu pozivati u programu nezavisno od konstruisanih objekata njihove klase, pa čak i ako nije konstruisan nijedan objekat.

Definicije nestatičkih (objektnih) metoda se nalaze unutar teksta klase, ali takvi metodi logički pripadaju konstruisanim objektima, a ne klasi.

To znači da se objektni metodi mogu primenjivati samo za neki objekat koji je prethodno konstruisan. Na primer, u klasi `ČlanPorodice` je definisan objektni metod `čestitajRođendan()` kojim se uzrast člana porodice uvećava za 1 i prikazuje čestitka za rođendan. Metodi `čestitajRođendan()` koji pripadaju različitim objektima klase `ČlanPorodice` obavljaju isti zadatak u smislu da čestitaju odgovarajući rođendan različitim članovima porodice. Ali njihov stvarni efekat je različit, jer uzrasti članova porodice mogu biti različiti. To je upravo odlika nestatičkih (objektnih) metoda u opštem slučaju: definisanjem tih metoda u klasi se određuje zadatak koji takvi metodi obavljaju nad konstruisanim objektima. S druge strane, njihov specifični efekat koji proizvode može varirati od objekta do objekta, zavisno od aktuelnih vrednosti nestatičkih (objektnih) polja različitih objekata.

Statički i nestatički članovi klase su dakle vrlo različiti koncepti i služe za različite svrhe. Važno je praviti razliku između teksta definicije klase i samog pojma klase. Definicija klase određuje kako klasu, tako i objekte koji se konstruišu na osnovu te definicije. Statičkim delom definicije se navode članovi koji su deo same klase, dok se nestatičkim delom navode članovi koji će biti deo svakog konstruisanog objekta.

Definicija klase

Opšti oblik definicije obične klase u Javi je vrlo jednostavan:

```
modifikatori class ime-klase {
   telo-klase
}
```

Modifikatori na početku definicije klase nisu obavezni, ali se mogu sastojati od jedne ili više službenih reči kojima se određuju izvesne karakteristike klase koja se definiše. Na primer, modifikator pristupa `public` ukazuje na to da se klasa može koristiti izvan svog paketa. Inače, bez tog modifikatora, klasa se može koristiti samo unutar svog paketa. Na raspolaganju su još tri modifikatora koje ćemo upoznati u daljem tekstu knjige. Ime klase se gradi na uobičajeni način, uz dodatnu konvenciju u Javi da sve reči imena klase počinju velikim slovom. Najzad, telo klase sadrži definicije statičkih i nestatičkih članova (polja i metoda) klase.

Konkretan primer, recimo u programu se za evidentiranje studenata i njihovih rezultata na testovima za jedan predmet, bila bi klasa `Student` koja je definisana na sledeći način:

```
public class Student {

   public String ime;              // ime studenta
   public int id;                  // matični broj studenta
   public double test1,test2,test3; // ocene na tri testa

   public double prosek() {        // izračunavanje prosečne ocene
      return (test1 + test2 + test3) / 3;
   }
}
```

Nijedan od članova klase `Student` nema modifikator `static`, što znači da ta klasa nema statičkih članova i da je zato ima smisla koristiti samo za konstruisanje objekata. Svaki objekat koji je instanca klase `Student` sadrži polja `ime`, `test1`, `test2` i `test3`, kao i metod `prosek()`. Primerci ova četiri polja u različitim objektima imaće generalno različite vrednosti. Zbog toga, metod `prosek()` primenjen za različite objekte klase `Student` davaće različite rezultate prosečnih ocena, jer će se za svaki objekat koristiti aktuelne vrednosti polja tog objekta. (Ovaj efekat je zapravo tačno ono što se misli kada se kaže da nestatički (objektni) metod pripada pojedinačnim objektima, a ne klasi.)

2.2 Promenljive klasnog tipa

Jedan aspekt definicije neke klase je to što se time opisuje struktura svih objekata koji pripadaju klasi. Drugi aspekt definicije neke klase je to što se time uvodi novi tip podataka u programu. Vrednosti tog klasnog tipa su objekti same klase. Iako ova činjenica zvuči pomalo tehnički, ona ima dalekosežne posledice. Naime, pošto se klasom definiše jedan tip podataka, to znači da se ime definisane klase može koristiti za tip promenljive u naredbi njene definicije, kao i za tip formalnog parametra i za tip rezulatata u definiciji nekog metoda. Na primer, ako se ima u vidu prethodna definicija klase

`Student`, u programu se može definisati promenljiva klasnog tipa `Student`:

```
Student s;
```

Kao što je to uobičajeno, ovom naredbom se u memoriji računara rezerviše prostor za promenljivu `s` radi čuvanja njenih vrednosti tipa `Student`. Međutim, iako vrednosti tipa `Student` jesu objekti klase `Student`, promenljiva `s` *ne* sadrži ove objekte. U stvari, u Javi važi opšte pravilo da nijedna promenljiva nikad ne sadrži neki objekat.

Da bismo ovo razjasnili, moramo bolje razumeti postupak konstruisanja objekata. Objekti se konstruišu u specijalnom delu memorije programa koji se zove *hip* (engl. *heap*). Pri tome se, zbog brzine, ne vodi mnogo računa o redu po kojem se zauzima taj deo memorije. (Reč *heap* na engleskom znači zbrkana gomila, hrpa.) To znači da se novi objekat smešta u hip-memoriju tamo gde se nađe prvo slobodno mesto, a i da se njegovo mesto prosto oslobađa kada nije više potreban u programu. Naravno, da bi se moglo pristupiti objektu u programu, mora se imati informacija o tome gde se on nalazi u hip-memoriji. Ta infomacija se naziva *referenca* ili *pokazivač* na objekat i predstavlja adresu memorijske lokacije objekta u hip-memoriji.

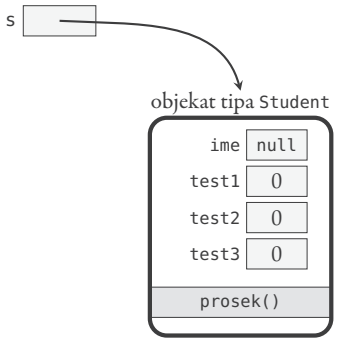
Promenljiva klasnog tipa dakle ne sadrži neki objekat kao svoju vrednost, već referencu na taj objekat. Zato kada se u programu koristi promenljiva klasnog tipa, ona služi za indirektan pristup objektu na koji ukazuje aktuelna vrednost (referenca) te promenljive.

Konstruisanje objekata svake klase u programu se izvodi posebnim operatorom koji se označava službenom rečju `new`. Pored konstruisanja jednog objekta, odnosno rezervisanja potrebnog memorijskog prostora za njega u hip-memoriji, rezultat ovog operatora je i vraćanje reference na novokonstruisani objekat. To je neophodno da bi se kasnije u programu moglo pristupiti novom objektu i s njim uradilo nešto korisno. Zbog toga se vraćena referenca na novi objekat mora sačuvati u promenljivoj klasnog tipa, jer se inače novom objektu nikako drugačije ne može pristupiti. Na primer, ako je promenljiva `s` tipa `Student` definisana kao u prethodnom primeru, onda se izvršavanjem naredbe dodele

```
s = new Student();
```

najpre konstruiše novi objekat koji je instanca klase `Student`, a zatim se referenca na njega dodeljuje promenljivoj `s`. Ovaj efekat je ilustrovan na sli-

ci 2.1 u kojoj je prikazano stanje memorije nakon izvršavanja prethodne naredbe dodele. Na toj slici, referenca na novi objekat klase `Student` koja je sadržaj promenljive `s`, prikazana je u obliku strelice koja pokazuje na taj objekat. Primetimo da je prikazan i automatski sadržaj dodeljen svim poljima novog objekta.



Slika 2.1: Efekat izvršavanja naredbe `Student s = new Student()`.

Prema tome, vrednost promenljive `s` je referenca na novi objekat, a ne sam taj objekat. Nije zato sasvim ispravno kratko reći da je taj objekat vrednost promenljive `s`, mada je ponekad teško izbeću ovu kraću terminologiju. Još manje je ispravno reći da promenljiva `s` sadrži taj objekat. Ispravan način izražavanja je da promenljiva `s` ukazuje na novi objekat. (U daljem tekstu se pridržavamo ove tačnije terminologije u vezi sa promenljivim klasnog tipa, sem ako to nije zaista rogobatno kao na primer u slučaju stringova.)

Iz definicije klase `Student` može se zaključiti da novokonstruisani objekat klase `Student`, na koji ukazuje promenljiva `s`, sadrži polja `ime`, `test1`, `test2` i `test3`. U programu se ova polja tog konkretnog objekta mogu koristiti pomoću standardne tačka-notacije, odnosno `s.ime`, `s.test1`, `s.test2` i `s.test3`. Tako, recimo, mogu se pisati sledeće naredbe:

```
System.out.println("Ocene studenta " + s.ime + " su:");
System.out.println("Prvi test: " + s.test1);
System.out.println("Drugi test: " + s.test2);
System.out.println("Treći test: " + s.test3);
```

Ovim naredbama bi se na ekranu prikazali ime i ocene studenta predstavljenog objektom na koji ukazuje promenljiva `s`. Opštije, polja `s.ime` i, recimo, `s.test1` mogu se koristiti u programu na svakom mestu gde je dozvoljena upotreba promenljivih tipa `String` odnosno `double`. Na primer, ako treba odrediti broj znakova stringa u polju `s.ime`, onda se može koristiti zapis `s.ime.length()`.

Slično, objektni metod `prosek()` se može pozvati za objekat na koji ukazuje `s` zapisom `s.prosek()`. Da bi se prikazala prosečna ocena studenta na koji ukazuje `s`, može se pisati na primer:

```
System.out.print("Prosečna ocene studenta " + s.ime);
System.out.println(" je: " + s.prosek());
```

U nekim slučajevima je potrebno naznačiti da promenljiva klasnog tipa ne ukazuje ni na jedan objekat. To se postiže dodelom specijalne reference `null` promenljivoj klasnog tipa. Na primer, može se pisati naredba dodele

```
s = null;
```

ili naredba grananja u obliku:

```
if (s == null) . . .
```

Ako je vrednost promenljive klasnog tipa jednaka `null`, onda ta promenljiva ne ukazuje ni na jedan objekat, pa bi se tokom izvršavanja programa dobila greška ukoliko se s tom promenljivom koriste objektna polja i metodi odgovarajuće klase. Tako, ako promenljiva `s` ima vrednost `null`, onda bi bila greška ukoliko se izvršava, recimo, `s.test1` ili `s.prosek()`.

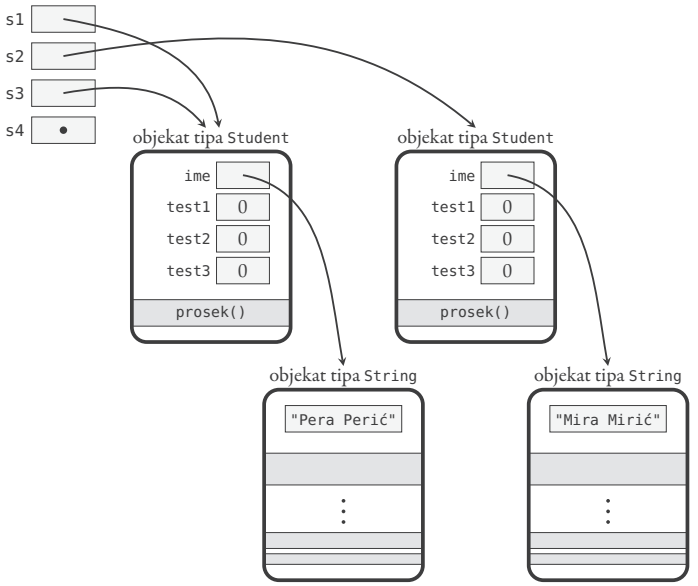
Da bismo bolje razumeli vezu između promenljivih klasnog tipa i objekata, razmotrimo detaljnije sledeći programski fragment:

```
1   Student s1, s2, s3, s4;
2   s1 = new Student();
3   s2 = new Student();
4   s1.ime = "Pera Perić";
5   s2.ime = "Mira Mirić";
6   s3 = s1;
7   s4 = null;
```

Prvom naredbom u ovom programskom fragmentu se definišu četiri promenljive klasnog tipa `Student`. Drugom i trećom naredbom se kon-

struišu dva objekta klase `Student` i reference na njih se redom dodeljuju promenljivim `s1` i `s2`. Narednim naredbama u redovima 4 i 5 se vlastitom polju `ime` ovih novih objekata dodeljuju vrednosti odgovarajućih stringova. (Podsetimo se da ostala polja konstruisanih objekata dobijaju podrazumevane vrednosti odmah nakon konstruisanja objekata.) Na kraju, pretposlednjom naredbom u redu 6 se vrednost promenljive `s1` dodeljuje promenljivoj `s3`, a poslednjom naredbom u redu 7 se referenca `null` dodeljuje promenljivoj `s4`.

Stanje memorije posle izvršavanja svih naredbi u ovom primeru prikazano je na slici 2.2. Na toj slici su reference na objekte prikazane na uobičajeni način kao strelice, dok je specijalna referenca `null` (koju sadrži promenljiva `s4`) prikazana kao podebljana tačka.



Slika 2.2: Veza između promenljivih klasnih tipova i objekata.

Sa slike 2.2 se vidi da promenljive `s1` i `s3` ukazuju na isti objekat. Ta činjenica je posledica izvršavanja pretposlednje naredbe dodele `s3 = s1` kojom se kopira referenca iz `s1` u `s3`. Obratite pažnju na to da ovo važi i u opštem slučaju: kada se jedna promenljiva klasnog tipa dodeljuje drugoj,

onda se kopira samo referenca, ali ne i objekat na koji ta referenca ukazuje. To znači da je vrednost polja `s1.ime` posle izvršavanja naredbi u prethodnom primeru jednaka stringu `"Pera Perić"`, ali i da je vrednost polja `s3.ime` jednaka `"Pera Perić"`.

U Javi se jednakost ili nejednokost promenljivih klasnog tipa može proveravati relacijskim operatorima == i `!=`, ali pri tome treba biti obazriv. Nastavljajući prethodni primer, ako se napiše, recimo,

```
if (s1 == s3) . . .
```

onda se, naravno, ispituje da li su vrednosti promenljivih `s1` u `s3` jednake. Ali te vrednosti su reference, tako da se time ispituje samo da li promenljive `s1` u `s3` ukazuju na isti objekat, ali ne i da li su jednaki objekti na koje ove promenljive ukazuju. To je u nekim slučajevima baš ono što je potrebno, ali ako treba ispitati jednakost samih objekata na koje promenljive ukazuju, onda se mora pojedinačno ispitati jednakost svih polja tih objekata. Drugim rečima, za objekte na koje ukazuju promenljive `s1` i `s3` treba ispitati da li je tačan sledeći uslov:

```
s1.ime.equals(s3.ime) && (s1.test1 == s3.test1) &&
      (s1.test2 == s3.test2) && (s1.test3 == s3.test3)
```

Budući da su stringovi zapravo objekti klase `String`, a ne primitivne vrednosti, na slici 2.2 su stringovi `"Pera Perić"` i `"Mira Mirić"` prikazani kao objekti. Neka promenljiva klasnog tipa `String` može dakle sadržati samo referencu na objekat stringa (kao i referencu `null`), a ne i sâm niz znakova koji čine string. To objašnjava zašto su na slici 2.2 vrednosti dva primerka polja `ime` tipa `String` prikazane strelicama koje pokazuju na objekte odgovarajućih stringova. Primetimo da se u vezi sa stringovima često primenjuje neprecizna objektna terminologija, iako su stringovi pravi objekti kao i svaki drugi u Javi. Tako, na primer, kažemo da je vrednost polja `s1.ime` baš string `"Pera Perić"`, a ne pravilno ali rogobatnije da je vrednost polja `s1.ime` referenca na objekat stringa `"Pera Perić"`.

Spomenimo na kraju još dve logične posledice činjenice da promenljive klasnog tipa sadrže reference na objekte, a ne same objekte. (Istaknimo još jednom to važno pravilo: objekti se fizički nalaze negde u hip-memoriji, a promenljive klasnog tipa samo ukazuju na njih.) Prvo, pretpostavimo da je promenljiva klasnog tipa deklarisana s modifikatorom `final`. To zna-

či da se njena vrednost nakon inicijalizacije više ne može promeniti. Ta početna, nepromenljiva vrednost ovakve promenljive klasnog tipa je referenca na neki objekat, što znači da će ona ukazivati na njega za sve vreme svog postojanja. Međutim, to nas ne sprečava da promenimo vrednosti polja koja se nalaze u objektu na koji ukazuje `final` promenljiva klasnog tipa. Drugim rečima, vrednost `final` promenljive je konstantna referenca, a nije konstantan objekat na koji ta promenljiva ukazuje. Ispravno je zato pisati, na primer:

```
final Student s = new Student();  // vrednost polja ime je:
                                  // s.ime = null
s.ime = "Laza Lazić";  // promena polja ime,
                       // a ne promenljive s
```

Drugo, pretpostavimo da se vrednost promenljive klasnog tipa `x` prenosi kao argument prilikom poziva nekog metoda. Onda se, kao što je poznato, vrednost promenljive `x` dodeljuje odgovarajućem parametru metoda i telo metoda se izvršava. Vrednost promenljive `x` se ne može promeniti izvršavanjem metoda, ali pošto dodeljena vrednost odgovarajućem parametru predstavlja referencu na neki objekat, u metodu se mogu promeniti polja u tom objektu. Drugačije rečeno, nakon izvršavanja metoda, promenljiva `x` će i dalje ukazivati na isti objekat, ali polja u tom objektu mogu biti promenjena.

Konkretnije, pretpostavimo da je definisan metod

```
void promeni(Student s) {
   s.ime = "Mika Mikić";
}
```

i da se izvršava sledeći programski fragment:

```
Student x = new Student();
x.ime = "Pera Perić";
promeni(x);
System.out.println(x.ime);
```

Rezultat izvršavanja ovog fragmenta je da se na ekranu prikazuje string `"Mika Mikić"` kao vrednost polja `x.ime`, a ne `"Pera Perić"`. Naime, izvršavanje naredbe poziva `promeni(x)` ekvivalentno je izvršavanju ove dve naredbe dodele:

```
s = x;
s.ime = "Mika Mikić";
```

Stoga polje `ime` objekta na koji ukazuje promenljiva `s`, a time i `x`, dobiće novu vrednost.

2.3 Konstrukcija i inicijalizacija objekata

Klasni tipovi u Javi su vrlo različiti od primitivnih tipova. Vrednosti primitivnih tipova su „ugrađene" u jezik, dok se vrednosti klasnih tipova, koje predstavljaju objekte odgovarajuće klase, moraju u programu eksplicitno konstruisati. Primetimo da, s druge strane, ne postoji suštinska razlika u postupanju sa promenljivim primitivnih i klasnih tipova, jer se razlikuju samo vrednosti koje mogu biti njihov sadržaj — kod jednih su to primitivne vrednosti, a kod drugih su to reference.

Postupak konstruisanja jednog objekta se sastoji od dva koraka: rezervisanja dovoljno mesta u hip-memoriji za smeštanje objekta i inicijalizacije njegovih polja podrazumevanim vrednostima. Programer ne može da utiče na prvi korak pronalaženja mesta u memoriji za smeštanje objekta, ali za drugi korak u složenijem programu obično nije dovoljna samo podrazumevana inicijalizacija. Podsetimo se da se ova automatska inicijalizacija sastoji od dodele vrednosti nula poljima numeričkog tipa (`int`, `double`, ...), dodele vrednosti `false` logičkim poljima, dodele Unicode znaka `'\u0000'` znakovnim poljima i dodele reference `null` poljima klasnog tipa. Ukoliko podrazumevana inicijalizacija nije dovoljna, poljima se mogu dodeliti početne vrednosti u njihovim deklaracijama, baš kao što se to može uraditi za bilo koje druge promenljive.

Radi konkretnosti, pretpostavimo da u programu za simuliranje neke društvene igre treba predstaviti kocku za bacanje. U nastavku je prikazana klasa `KockaZaIgru` čiji su objekti računarske reprezentacije realnih objekata kocki za bacanje u društvenim igrama. Ova klasa sadrži jedno objektno polje čiji sadržaj odgovara broju na gornjoj strani kocke (tj. broju koji „pao" pri bacanju kocke) i jedan objektni metod kojim se simulira slučajno bacanje kocke.

```
public class KockaZaIgru {

   public int broj = 6;  // broj na gornjoj strani kocke

   public void baci() {  // "bacanje" kocke
      broj = (int)(6 * Math.random()) + 1;
   }
}
```

U ovom primeru klase `KockaZaIgru`, polje `broj` dobija početnu vrednost 6 uvek kada se konstruiše novi objekat klase `KockaZaIgru`. Pošto se u programu može konstruisati više objekata klase `KockaZaIgru`, svi oni će imati svoj primerak polja `broj` i svi oni dobijaju vrednost 6 na početku.

Drugi primer je možda interesantniji, jer je početni broj kocke za igru slučajan:

```
public class KockaZaIgru {

   public int broj = (int)(6 * Math.random()) + 1;

   public void baci() {
      broj = (int)(6 * Math.random()) + 1;
   }
}
```

U ovom primeru se polje `broj` inicijalizuje slučajnom vrednošću. Kako se inicijalizacija obavlja za svaki novokonstruisani objekat, različiti objekti druge verzije klase `KockaZaIgru` imaće verovatno različite početne vrednosti svojih primeraka polja `broj`.

Inicijalizacija statičkih polja neke klase nije mnogo drugačija od inicijalizacije nestatičkih (objektnih) polja, osim što treba imati u vidu da statička polja nisu vezana za pojedinačne objekte nego za klasu kao celinu. Kako postoji samo jedan primerak statičkog polja klase, ono se inicijalizuje samo jednom i to kada se klasa prvi put učitava od strane JVM prilikom izvršavanja programa.

Posvetimo sada više pažnje detaljima konstrukcije objekata u Javi. Konstruisanje objekata u programu se obavlja posebnim operatorom `new`. Na primer, u programu koji koristi klasu `KockaZaIgru` može se pisati:

```
KockaZaIgru kocka;          // deklaracija promenljive
                            //    klasnog tipa KockaZaIgru
kocka = new KockaZaIgru(); // konstruisanje objekta klase
                            //    KockaZaIgru i dodela njegove
                            //    reference toj promenljivoj
```

Isti efekat se može postići kraćim zapisom:

```
KockaZaIgru kocka = new KockaZaIgru();
```

U ovim primerima, izrazom `new KockaZaIgru()` na desnoj strani znaka jednakosti konstruiše se novi objekat klase `KockaZaIgru`, inicijalizuju se njegova objektna polja i vraća se referenca na taj novi objekat kao rezultat tog izraza. Ova referenca se zatim dodeljuje promenljivoj `kocka` na levoj strani znaka jednakosti, tako da na kraju promenljiva `kocka` ukazuje na novokonstruisani objekat.

Primetimo da deo `KockaZaIgru()` iza operatora `new` podseća na poziv metoda, što zapravo to i jeste. Naime, to je poziv specijalnog metoda koji se naziva *konstruktor* klase. Ovo je možda malo iznenađujuće, jer se u definiciji klase `KockaZaIgru` ne nalazi takav metod. Međutim, svaka klasa ima bar jedan konstruktor, koji se automatski dodaje ukoliko nije eksplicitno definisan nijedan drugi konstruktor. Taj podrazumevani konstruktor ima samo formalnu funkciju i praktično ne radi ništa. Ali kako se konstruktor poziva odmah nakon konstruisanja objekta i inicijalizacije njegovih polja podrazumevanim vrednostima, u klasi se može definisati jedan ili više posebnih konstruktora radi dodatne inicijalizacije novih objekta.

Definicija konstruktora klase se piše na isti način kao definicija običnog metoda, uz tri razlike:

1. Ime konstruktora mora biti isto kao ime klase u kojoj se definiše.
2. Jedini dozvoljeni modifikatori su `public`, `private` i `protected`.
3. Konstruktor nema tip rezultata, čak ni `void`.

S druge strane, jedan konstruktor sadrži uobičajeno telo metoda u formi bloka naredbi unutar kojeg se mogu pisati bilo koje naredbe. Isto tako, konstruktor može imati parametre kojima se mogu preneti vrednosti za inicijalizaciju objekata nakon njihove konstrukcije.

U trećoj verziji klase KockaZaIgru je definisan poseban konstruktor koji ima parametar za početnu vrednost broja na gornjoj strani kocke:

```
public class KockaZaIgru {

   public int broj;             // broj na gornjoj strani kocke

   public KockaZaIgru(int n) {  // konstruktor
      broj = n;
   }

   public void baci() {         // "bacanje" kocke
      broj = (int)(6 * Math.random()) + 1;
   }
}
```

U ovom primeru klase KockaZaIgru, konstruktor

```
   public KockaZaIgru(int n) {
      broj = n;
   }
```

ima isto ime kao klasa, nema tip rezultata i ima jedan parametar. Poziv konstruktora, sa odgovarajućim argumentima, navodi se iza operatora new. Na primer:

```
KockaZaIgru kocka = new KockaZaIgru(5);
```

Na desnoj strani znaka jednakosti ove naredbe definicije promenljive kocka konstruiše se novi objekat klase KockaZaIgru, poziva se konstruktor te klase kojim se polje broj novokonstruisanog objekta inicijalizuje vrednošću argumenta 5 i, na kraju, referenca na taj objekat se dodeljuje promenljivoj kocka.

Primetimo da se podrazumevani konstruktor dodaje klasi samo ukoliko nije eksplicitno definisan nijedan drugi konstruktor u klasi. Zato, na primer, izrazom new KockaZaIgru() ne bi više mogao da se konstruiše objekat prethodne klase KockaZaIgru. Ali to i nije veliki problem, jer se konstruktori mogu preopterećivati kao i obični metodi. To znači da klasa može imati više konstruktora pod uslovom, naravno, da su njihovi potpisi različiti. Na primer, prethodnoj klasi KockaZaIgru može se dodati konstruktor bez pa-

rametara koji polju `broj` konstruisanog objekta početno dodeljuje slučajan broj:

```
public class KockaZaIgru {

   public int broj;             // broj na gornjoj strani kocke

   public KockaZaIgru() {       // konstruktor bez parametara
      baci();                   // poziv metoda baci()
   }

   public KockaZaIgru(int n) {  // konstruktor sa parametrom
      broj = n;
   }

   public void baci() {         // "bacanje" kocke
      broj = (int)(6 * Math.random()) + 1;
   }
}
```

Objekti ove klase `KockaZaIgru` se mogu konstruisati na dva načina: bilo izrazom `new KockaZaIgru()` ili izrazom `new KockaZaIgru(x)`, gde je `x` izraz tipa `int`.

Na osnovu svega do sada rečenog može se zaključiti da su konstruktori specijalna vrsta metoda. Oni nisu objektni metodi jer ne pripadaju objektima, već se pozivaju samo u trenutku konstruisanja objekata. Ali oni nisu ni statički metodi klase, jer se za njih ne može koristiti modifikator `static`. Za razliku od drugih metoda, konstruktori se mogu pozvati samo uz operator `new` u izrazu oblika:

```
new ime-klase(lista-argumenata)
```

Rezultat ovog izraza je referenca na konstruisani objekat, koja se najčešće odmah dodeljuje promenljivoj klasnog tipa u naredbi dodele. Međutim, ovaj izraz se može pisati svuda gde to ima smisla, na primer kao neki argument u pozivu metoda ili kao deo nekog složenijeg izraza. Zbog toga je važno razumeti tačan postupak izračunavanja operatora `new` koji se sastoji od izvršavanja, redom, ova četiri koraka:

1. Pronalazi se dovoljno veliki blok slobodne hip-memorije za objekat navedene klase koji se konstruiše.
2. Inicijalizuju se objektna polja tog objekta. Početna vrednost nekog polja objekta je ili ona izračunata iz deklaracije tog polja u klasi ili podrazumevana vrednost predviđena za njegov tip.
3. Poziva se konstruktor klase na uobičajen način: najpre se eventualni argumenti u pozivu konstruktora izračunavaju i dodeljuju odgovarajućim parametrima konstruktora, a zatim se izvršavaju naredbe u telu konstruktora.
4. Referenca na konstruisani objekat se vraća kao rezultat operatora `new`.

Referenca na konstruisani objekat, koja je dobijena na kraju ovog postupka, može se zatim koristiti u programu za pristup poljima i metodima novog objekta.

Primer: bacanje dve kocke dok se ne pokaže isti broj

Jedna od prednosti objektno orijentisanog programiranja je mogućnost *višekratne upotrebe* programskog koda. To znači da, recimo, klase koje su jednom napisane, mogu se iskoristiti u različitim programima u kojima je potrebna njihova funkcionalnost. Na primer, poslednja klasa `KockaZaIgru` predstavlja realne kocke za igranje i obuhvata sve njihove relevantne atribute i mogućnosti. Zato se ta klasa može koristi u svakom programu čija se logika zasniva na kockama za igranje. To je velika prednost, pogotovo za komplikovane klase, jer nije potrebno gubiti vreme na ponovno pisanje i testiranje novih klasa.

Da bismo ilustrovali ovu mogućnost, u listingu 2.1 je prikazan program koji koristi klasu `KockaZaIgru` za određivanje broja puta koliko treba baciti dve kocke pre nego što se pokaže isti broj na njima.

Listing 2.1: Bacanja dve kocke

```
public class BacanjaDveKocke {

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

```
        int brojBacanja = 0;  // brojač bacanja dve kocke
        KockaZaIgru kocka1 = new KockaZaIgru();  // prva kocka
        KockaZaIgru kocka2 = new KockaZaIgru();  // druga kocka

        do {
           kocka1.baci();
           System.out.print("Na prvoj kocki je pao broj: ");
           System.out.println(kocka1.broj);

           kocka2.baci();
           System.out.print("Na drugoj kocki je pao broj: ");
           System.out.println(kocka2.broj);

           brojBacanja++;   // uračunati bacanje

        } while (kocka1.broj != kocka2.broj);

        System.out.print("Dve kocke su bačene " + brojBacanja);
        System.out.println(" puta pre nego što je pao isti broj.");
    }
}

class KockaZaIgru {

   public int broj;             // broj na gornjoj strani kocke

   public KockaZaIgru() {       // konstruktor bez parametara
      baci();                   // poziv metoda baci()
   }

   public KockaZaIgru(int n) {  // konstruktor sa parametrom
      broj = n;
   }

   public void baci() {         // "bacanje" kocke
      broj = (int)(6 * Math.random()) + 1;
```

```
    }
}
```

2.4 Uklanjanje objekata

Novi objekat se konstruiše operatorom `new` i (dodatno) inicijalizuje konstruktorom klase. Od tog trenutka se objekat nalazi u hip-memoriji i može mu se pristupiti preko promenljivih koje sadrže referencu na njega. Ako nakon izvesnog vremena objekat više nije potreban u programu, postavlja se pitanje da li se on može ukloniti? Odnosno, da li se radi uštede memorije može osloboditi memorija koju objekat zauzima?

U nekim jezicima sâm programer mora voditi računa o uklanjanju objekata i u tim jezicima su predviđeni posebni načini kojima se eksplicitno uklanja objekat u programu. U Javi, uklanjanje objekata se događa automatski i programer je oslobođen obaveze da brine o tome. Osnovni kriterijum na osnovu kojeg se u Javi prepoznaje da objekat nije više potreban je da ne postoji više nijedna promenljiva koja ukazuje na njega. To ima smisla, jer se takvom objektu više ne može pristupiti u programu, što je isto kao da ne postoji, pa bespotrebno zauzima memoriju.

Da bismo ovo ilustrovali, posmatrajmo sledeći metod (primer je doduše veštački, jer tako nešto ne treba pisati u pravom programu):

```
void noviStudent() {
   Student s = new Student();
   s.ime = "Pera Perić";
   . . .
}
```

U metodu `noviStudent()` se konstruiše objekat klase `Student` i referenca na njega se dodeljuje lokalnoj promenljivoj `s`. Ali nakon izvršavanja poziva metoda `noviStudent()`, njegova lokalna promenljiva `s` se dealocira tako da više ne postoji referenca na objekat konstruisan u metodu `noviStudent()`. Više dakle nema načina da se tom objektu pristupi u programu, pa se objekat može ukloniti i memorija koju zauzima osloboditi za druge namene.

U programu se može i eksplicitno ukazati da neki objekat nije više potreban. Na primer:

```
Student s = new Student();
. . .
s = null;
. . .
```

U ovom primeru, nakon konstruisanja novog objekta klase `Student` i njegovog korišćenja preko promenljive `s`, toj promenljivoj se eksplicitno dodeljuje referenca `null` kada objekat na koji ukazuje nije više potreban. Time se efektivno gubi veza s tim objektom, pa se njemu više ne može pristupiti u programu.

Objekti koji se nalaze u hip-memoriji, ali se u programu više ne mogu koristiti jer nijedna promenljiva ne sadrži referencu na njih, popularno se nazivaju „otpaci". U Javi se koristi posebna procedura *sakupljanja otpadaka* (engl. *garbage collection*) kojom se automatski s vremena na vreme „čisti đubre", odnosno oslobađa memorija onih objekata za koje se nađe da je broj referenci na njih u programu jednak nuli.

U prethodna dva primera se može vrlo lako otkriti kada jedan objekat klase `Student` nije dostupan i kada se može ukloniti. U složenim programima je to obično mnogo teže. Ako je neki objekat bio korišćen u programu izvesno vreme, onda je moguće da više promenljvih sadrže referencu na njega. Taj objekat nije „otpadak" sve dok postoji bar jedna promenljiva koja ukazuje na njega. Drugi komplikovaniji slučaj je kada grupa objekata obrazuje lanac referenci između sebe, a ne postoji promenljiva izvan tog lanca sa referencom na neki objekat u lancu. Srećom, procedura sakupljanja otpadaka može sve ovo prepoznati i zato su Java programeri u velikoj meri oslobođeni odgovornosti od racionalnog upravljanja memorijom uklanjanjem nepotrebnih objekata u programu.

Iako procedura sakupljanja otpadaka usporava izvršavanje samog programa, razlog zašto se u Javi uklanjanje objekata obavlja automatski, a ne kao u nekim jezicima ručno od strane programera, jeste to što je vođenje računa o tome u složenijim programima vrlo teško i podložno greškama. Prva vrsta čestih grešaka se javlja kada se nenamerno obriše neki objekat, mada i dalje postoje reference na njega. Ove greške *visećih pokazivača* dovode do problema pristupa nedozvoljenim delovima memorije. Druga vrsta greša-

ka se javlja kada programer zanemari da ukloni nepotrebne objekte. Tada dolazi do greške *curenja memorije* koja se manifestuje time što program zauzima veliki deo memorije, iako je ona praktično neiskorišćena. To onda dovodi do problema nemogućnosti izvršavanja istog ili drugih programa zbog nedostatka memorije.

2.5 Skrivanje podataka (enkapsulacija)

Do sada je malo pažnje bilo posvećeno kontroli pristupa članovima neke klase. U dosadašnjim primerima, članovi klase su uglavnom bili deklarisani sa modifikatorom `public` kako bi bili dostupni iz bilo koje druge klase. Međutim, važno objektno orijentisano načelo *enkapsulacije* (ili *učaurivanja*) nalaže da sva objektna polja klase budu skrivena i definisana s modifikatorom `private`. Pri tome, pristup vrednostima tih polja iz drugih klasa treba omogućiti samo preko javnih metoda.

Jedna od prednosti ovog pristupa je to što su na taj način sva polja (i interni metodi) klase bezbedno zaštićeni unutar „čaure" klase i mogu se menjati samo na kontrolisan način. To umnogome olakšava testiranje i pronalaženje grešaka. Ali možda važnija korist je to što se time mogu sakriti interni implementacioni detalji klase. Ako drugi programeri ne mogu koristiti te detalje, već samo elemente dobro definisanog interfejsa klase, onda se implementacija klase može lako promeniti usled novih okolnosti. To znači da je dovoljno zadržati samo iste elemente starog interfejsa u novoj implementaciji, jer se time neće narušiti ispravnost nekog programa koji koristi prethodnu implementaciju klase.

Da bismo ove opštu diskusiju učinili konkretnijom, razmotrimo primer jedne klase koja predstavlja radnike neke firme, recimo, radi obračuna plata:

```
public class Radnik {

   // Privatna polja
   private String ime;
   private long jmbg;
   private int staž;
```

```
    private double plata;

    // Konstruktor
    public Radnik(String i, long id, int s, double p) {
       ime = i;
       jmbg = id;
       staž = s;
       plata = p;
    }

    // Javni interfejs
    public String getIme() {
       return ime;
    }

    public long getJmbg() {
       return jmbg;
    }

    public int getStaž() {
       return staž;
    }

    public void setStaž(int s) {
       staž = s;
    }

    public double getPlata() {
       return plata;
    }

    public void povećajPlatu(double procenat) {
       plata += plata * procenat / 100;
    }
}
```

Obratite pažnju na to da su sva objektna polja klase `Radnik` definisana da budu privatna. Time je obezbeđeno da se ona izvan same klase `Radnik`

ne mogu direktno koristiti. Tako, u nekoj drugoj klasi se više ne može pisati, na primer:

```
Radnik r = new Radnik("Pera Perić",111111,3,1000);
System.out.println(r.ime); // GREŠKA: ime je privatno polje
r.staž = 17;               // GREŠKA: staž je privatno polje
```

Naravno, vrednosti polja za konkretnog radnika se u drugim klasama programa moraju koristiti na neki način, pa su u tu svrhu definisani javni metodi sa imenima koja počinju rečima *get* i *set*. Ovi metodi su deo javnog interfejsa klase i zato se u drugoj klasi može pisati:

```
Radnik r = new Radnik("Pera Perić",111111,3,1000);
System.out.println(r.getIme());
r.setStaž(17);
```

Metodi čija imena počinju sa *get* samo vraćaju vrednosti objektnih polja i nazivaju se *geteri*. Metodi čija imena počinju sa *set* menjaju sadržaj objektnih polja i nazivaju se *seteri* (ili *mutatori*). Nažalost, ova terminologija nije u duhu srpskog jezika, ali je uobičajena usled nedostatka makar približno dobrog prevoda. Čak i da postoje bolji srpski izrazi, konvencija je da se ime geter-metoda za neku promenljivu pravi dodavanjem reči „get" ispred imena te promenljive s početnim velikim slovom. Tako, za promenljivu `ime` se dobija `getIme`, za promenljivu `jmbg` se dobija `getJmbg` i tako dalje. Ime geter-metoda za logičko polje se dozvoljava da počinje i sa „is". Da u klasi `Radnik` postoji, recimo, polje `vredan` tipa `boolean`, njegov geter-metod bi se mogao zvati `isVredan()` umesto `getVredan()`. Konvencija za ime seter-metoda za neku promenljivu je da se ono pravi dodavanjem reči „set" ispred imena te promenljive s početnim velikim slovom. Zato je za promenljivu `staž` u klasi `Radnik` definisan seter-metod `setStaž()`.

Ovo mešanje engleskih reči „get", „set" i „is" sa moguće srpskim imenima promenljivih dodatno doprinosi rogobatnosti tehnike skrivanja podataka. S druge strane, neki aspekti naprednog Java programiranja se potpuno zasnivaju na prethodnoj konvenciji za imena getera i setera. Na primer, tehnologija programskih komponenti JavaBeans podrazumeva da geter ili seter metodi u klasi definišu takozvano *svojstvo* klase (koje čak ni ne mora odgovarati polju). Zbog toga se preporučuje da se programeri pridržavaju konvencije za imena getera i setera, kako bi se olakšalo eventualno prilago-

đavanje klase naprednim tehnikama.

Da li se nešto dobija time što su polja `ime`, `jmbg`, `staž` i `plata` u klasi `Radnik` definisana da budu privatna na račun dodatnog pisanja nekoliko naizgled nepotrebnih metoda u klasi `Radnik`? Zar nije jednostavnije da sva ova polja budu definisana da budu javna i tako da se izbegnu dodatne komplikacije?

Prvo što se dobija za dodatno uloženi trud je lakše otkrivanje grešaka. Polja `ime` i `jmbg` se ne mogu nikako menjati nakon početne dodele vrednosti u konstruktoru, čime je osigurano to da se u nijednom delu programa izvan klase `Radnik` ne može (slučajno ili namerno) ovim poljima dodeliti neka nekonzistentna vrednost. Vrednosti polja `staž` i `plata` se mogu menjati, ali samo na konrolisan način metodima `setStaž()` i `povećajPlatu()`. Prema tome, ako su vrednosti tih polja na neki način pogrešne, jedini uzročnici mogu biti ovi metodi, pa je zato veoma olakšano traženje greške. Da su polja `staž` i `plata` bila javna, onda bi se izvor greške mogao nalaziti bilo gde u programu.

Druga korist je to što geter i seter metodi nisu ograničeni samo na čitanje i upisivanje vrednosti odgovarajućih polja. Ova mogućnost se može iskoristiti tako da se, na primer, u geter-metodu prati (broji) koliko puta se pristupa nekom polju:

```
public double getPlata() {
   plataBrojPristupa++;
   return plata;
}
```

Slično, u seter-metodu se može obezbediti da dodeljene vrednosti polju budu samo one koje su smislene:

```
public void setStaž(int s) {
   if (s < 0) {
      System.out.println("Greška: staž je negativan");
      System.exit(-1);
   }
   else
      staž = s;
}
```

Treća korist od skrivanja podataka je to što se može promeniti interna

implementacija neke klase bez posledica na programe koji koriste tu klasu. Na primer, ako predstavljanje punog imena radnika mora da se razdvoji u dva posebna dela za ime i prezime, onda je dovoljno klasi `Radnik` dodati još jedno polje za prezime, recimo,

```
private String prezime;
```

i promeniti geter-metod `getIme()` tako da se ime radnika formira od dva dela:

```
public String getIme() {
   return ime + " " + prezime;
}
```

Ove promene su potpuno nezavisne od drugih programa koji koriste klasu `Radnik` i ti programi se ne moraju uopšte menjati (čak ni ponovo prevoditi) da bi ispravno radili kao ranije. U opštem slučaju, geteri i seteri, pa i ostali metodi, verovatno moraju pretrpeti velike izmene radi prelaska sa stare na novu implementaciju klase. Ali poenta enkapsulacije je da stari programi koji koriste novu verziju klase ne moraju uopšte da se menjaju.

2.6 Službena reč `this`

Objektni metodi neke klase se primenjuju za pojedine objekte te klase i tokom svog izvršavanja ti metodi koriste konkretne vrednosti polja onih objekata za koje su pozvani. Na primer, objektni metod `povećajPlatu()` klase `Radnik` iz prethodnog odeljka

```
public void povećajPlatu(double procenat) {
   plata += plata * procenat / 100;
}
```

dodeljuje novu vrednost polju `plata` koje pripada objektu za koji se ovaj metod pozove. Efekat poziva, recimo,

```
pera.povećajPlatu(10);
```

sastoji se od povećanja vrednosti polja `plata` za 10% onog objekta na koji ukazuje promenljiva `pera` tipa `Radnik`. Drugim rečima, ovaj efekat je ekvivalentan izvršavanju naredbe dodele:

```
pera.plata += pera.plata * 10 / 100;
```

Slično, ako na neki drugi objekat klase `Radnik` ukazuje promenljiva `laza`, onda se pozivom

```
laza.povećajPlatu(10);
```

uvećava za 10% vrednost polja `plata` tog drugog objekta, odnosno taj efekat je ekvivalentan izvršavanju naredbe dodele:

```
laza.plata += laza.plata * 10 / 100;
```

Prema tome, poziv objektnog metoda `povećajPlatu()` sadrži dva argumenta. Prvi, *implicitni* argument se nalazi ispred imena metoda i ukazuje na objekat klase `Radnik` za koji se metod poziva. Drugi, *eksplicitni* argument se nalazi iza imena metoda u zagradama.

Poznato je da se parametri koji odgovaraju eksplicitnim argumentima poziva metoda navode u definiciji metoda. S druge strane, parametar koji odgovara implicitnom argumentu se ne navodi u definiciji metoda, ali se može koristiti u svakom metodu. Njegova oznaka u Javi je službena reč `this`. U ovom slučaju dakle, reč `this` označava promenljivu klasnog tipa koja u trenutku poziva metoda dobija vrednost reference na objekat za koji je metod pozvan. To znači da se, recimo, u metodu `povećajPlatu()` može pisati:

```
public void povećajPlatu(double procenat) {
   this.plata += this.plata * procenat / 100;
}
```

Primetimo da je ovde reč `this` uz polje `plata` nepotrebna i da se podrazumeva. Ipak, neki programeri uvek koriste ovaj stil pisanja, jer se tako jasno razlikuju objektna polja klase od lokalnih promenljivih metoda.

Prilikom poziva konstruktora, implicitni parametar `this` ukazuje na objekat koji se konstruiše. Zbog toga se parametrima konstruktora često daju ista imena koja imaju objektna polja klase, a u telu konstruktora se ta polja pišu sa prefiksom `this`. Na primer:

```
public Radnik(String ime, long jmbg, int staž, double plata) {
   this.ime = ime;
   this.jmbg = jmbg;
   this.staž = staž;
```

```
        this.plata = plata;
    }
```

Prednost ovog stila pisanja konstruktora je to što ne moraju smišljati dobra imena za parametre konstruktora kojima se jasno ukazuje šta svaki od njih znači. Primetimo da se u telu konstruktora moraju pisati puna imena polja sa prefiksom `this`, jer nema smisla pisati naredbu dodele, recimo, `ime = ime`. U stvari, to bi bilo pogrešno, jer se `ime` u telu konstruktoru odnosi na parametar konstruktora. Zato bi se naredbom `ime = ime` vrednost parametra `ime` opet dodelila njemu samom, a objektno polje `ime` bi ostalo nepromenjeno.

U prethodnim primerima nije neophodno koristiti promenljivu `this`. U prvom primeru se ona implicitno dodaje, pa je njeno isticanje više odlika ličnog stila. U drugom primeru se može izbeći njena upotreba davanjem imena parametrima konstruktora koja su različita od onih koja imaju objektna polja.

Ipak, u nekim slučajevima je promenljiva `this` neophodna i bez nje se ne može dobiti željena funkcionalnost. Da bismo to pokazali, pretpostavimo da je klasi `Radnik` potrebno dodati objektni metod kojim se upoređuju dva radnika na osnovu njihovih plata. Tačnije, treba napisati objektni metod `većeOd()` tako da se pozivom tog metoda u obliku, na primer,

```
pera.većeOd(laza)
```

od dva data objekta na koje ukazuju promenljive `pera` i `laza`, kao rezultat dobija referenca na onaj objekat koji ima veću platu. Ovaj metod mora koristiti promenljivu `this`, jer rezultat metoda može biti implicitni parametar metoda:

```
public Radnik većeOd(Radnik drugi) {
    if (this.getPlata() > drugi.getPlata())
        return this;
    else
        return drugi;
}
```

Obratite ovde pažnju na to da se u zapisu `this.getPlata()` može izostaviti promenljiva `this`, jer se bez nje poziv metoda `getPlata()` ionako odnosi na implicitni parametar metoda `većeOd()` označen promenljivom `this`. S dru-

ge strane, promenljiva `this` jeste obavezna u naredbi `return this` u prvoj grani `if` naredbe, jer je rezultat metoda `većeOd()` u toj grani baš implicitni parametar ovog metoda.

Službena reč `this` ima još jedno, potpuno drugačije značenje od prethodnog. Naime, konstruktor klase je običan metod koji se može preopterećivati, pa je moguće imati više konstruktora sa različitim potpisom u istoj klasi. U tom slučaju se različiti konstruktori mogu međusobno pozivati, ali se poziv jednog konstruktora unutar drugog piše u obliku:

```
this(lista-argumenata);
```

Na primer, ukoliko klasi `Radnik` treba dodati još jedan konstruktor kojim se inicijalizuje pripravnik bez radnog staža i sa fiksnom platom, to se može uraditi na standardan način:

```
public Radnik(String ime, long jmbg) {
   this.ime = ime;
   this.jmbg = jmbg;
   this.staž = 0;
   this.plata = 100;
}
```

Ali umesto toga, novi konstruktor se može kraće pisati:

```
public Radnik(String ime, long jmbg) {
   this(ime,jmbg,0,100);
}
```

U telu ovog konstruktora je zapisom

```
this(ime,jmbg,0,100);
```

označen poziv prvobitnog konstruktora klase `Radnik` sa četiri parametra. A izvršavanje tog konstruktora sa navedenim argumentima je ekvivalentno baš onome što treba uraditi za pripravnika.

Prednost primene službene reči `this` u ovom kontekstu je dakle to što se zajedničke naredbe za konstruisanje objekata mogu pisati samo na jednom mestu u najopštijem konstruktoru. Onda se pozivom tog konstruktora pomoću `this` sa odgovarajućim argumentima mogu obezbediti drugi konstruktori za inicijalizovanje specifičnijih objekata. Pri tome treba imati

u vidu i jedno ograničenje: poziv nekog konstruktora pomoću this mora biti prva naredba u drugom konstruktoru. Zato nije ispravno pisati

```
public Radnik(String ime, long jmbg) {
   System.out.println("Konstruisan pripravnik ... ");
   this(ime,jmbg,0,100);
}
```

ali jeste ispravno:

```
public Radnik(String ime, long jmbg) {
   this(ime,jmbg,0,100);
   System.out.println("Konstruisan pripravnik ... ");
}
```

GLAVA 3

Nizovi

Osnovna jedinica za čuvanje podataka u programu je promenljiva. Ali, jedna promenljiva u svakom trenutku može sadržati samo jednu vrednost. Ako je u programu potrebno istovremeno imati na raspolaganju više srodnih podataka koji čine jednu celinu, onda se u Javi oni mogu čuvati u formi objekta koji se sastoji samo od polja, bez metoda. Takva forma organizacije podataka se u drugim jezicima naziva *slog*. To međutim ima smisla samo za mali broj podataka, jer definisanje velikog broja polja nije lako izvodljivo zbog toga što sva polja moraju imati različita imena.

Istovremeno raspolaganje velikim (čak neograničenim) brojem podataka zahteva poseban način rada sa njima koji omogućava relativno lako dodavanje, uklanjanje, pretraživanje i slične operacije sa pojedinačnim podacima. Ako se ima u vidu kolekcija podataka organizovanih u ovom smislu, onda se govori o *strukturi podataka*. U ovom poglavlju se govori o najosnovnijoj strukturi podataka u Javi koja se naziva *niz*.

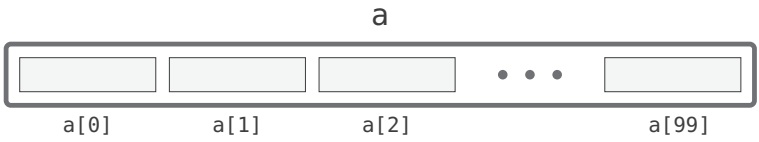
3.1 Jednodimenzionalni nizovi

Niz je struktura podataka koja predstavlja numerisan niz promenljivih istog tipa. Pojedinačne promenljive u nizu se nazivaju *elementi* niza, a njihov redni broj u nizu se naziva *indeks*. Ukupan broj elemenata niza se naziva *dužina* niza. Dužina niza je jedina, „horizontalna" dimenzija niza elemenata (promenljivih), pa se za ovu vrstu nizova kaže da su jednodimenzio-

nalni ukoliko ih je potrebno razlikovati od višedimenzionalnih koji pored dužine mogu imati i visinu, dubinu i tako dalje.

U Javi, numeracija elemenata niza počinje od nule. To znači da ako je n dužina niza, onda indeks nekog elementa niza može biti u intervalu od 0 do $n-1$. Svi elementi niza moraju biti istog tipa koji se naziva *bazni tip* niza. Za bazni tip elemenata niza nema ograničenja — to može biti bilo koji tip u Javi, primitivni ili klasni.

Niz elemenata se kao celina u programu predstavlja jednom promenljivom specijalnog, nizovnog tipa. Radi jednostavnijeg izražavanja, ova sama promenljiva se često naziva niz, mada je preciznije reći „promenljiva koja ukazuje na niz elemenata". Za označavanje bilo kog elementa niza se koristi zapis koji se sastoji od imena te promenljive i indeksa odgovarajućeg elementa u uglastim (srednjim) zagradama. Na primer, niz `a` od 100 elemenata se sastoji od 100 promenljivih istog tipa čiji je konceptualni izgled prikazan na slici 3.1.



Slika 3.1: Niz `a` od 100 elemenata.

Svaki element niza je obična promenljiva baznog tipa niza i može imati bilo koju vrednost baznog tipa. Na primer, ako je bazni tip niza `a` na slici 3.1 definisan da bude `int`, onda je svaka od 100 promenljivih `a[0]`, `a[1]`, `a[2]`, ..., `a[99]` obična celobrojna promenljiva koja se u programu može koristiti na svakom mestu gde su dozvoljene celobrojne promenljive.

Definisanje nizova

U Javi je niz elemenata realizovan na objektno orijentisan način: nizovi se u Javi smatraju specijalnom vrstom objekata. Pojedinačni elementi niza su, u suštini, polja unutar objekta niza, s tim što se ona ne označavaju svojim imenima nego indeksima.

Posebnost nizova u Javi se ogleda i u tome što, mada kao objekti moraju pripadati nekoj klasi, njihova klasa ne mora da se definiše u programu. Naime, svakom postojećem tipu `T` se automatski pridružuje klasa nizova koja se označava `T[]`. Ova klasa `T[]` je upravo ona kojoj pripada objekat niza čiji su pojedinačni elementi tipa `T`. Tako, na primer, tipu `int` odgovara klasa `int[]` čiji su objekti svi nizovi baznog tipa `int`. Ili, ako je u programu definisana klasa `Student`, onda je automatski raspoloživa i klasa `Student[]` kojoj pripadaju svi objekti nizova baznog tipa `Student`.

Za objekte klase `T[]` se koristi kraći termin „niz baznog tipa `T`" ili češće još kraći „niz tipa `T`". Primetimo da su strogo govoreći ti termini neprecizni, jer se onda nizom naziva i klasa i objekat. Nadamo se ipak da, posle početnog upoznavanja, ova dvosmislenost neće izazvati konfuziju kod čitalaca.

Postojanje klasnog tipa niza, recimo `int[]`, omogućava da se definiše neka promenljiva tog klasnog tipa. Na primer:

```
int[] a;
```

Kao i svaka promenljiva klasnog tipa, ova promenljiva `a` može sadržati referencu na neki objekat klase `int[]`. A kao što smo upravo objasnili, objekti klase `int[]` su nizovi baznog tipa `int`. Kao i svaki objekat, objekat niza tipa `int[]` se konstruiše operatorom `new`, doduše u posebnom obliku, a referenca na taj novi niz se zatim može dodeliti promenljivoj `a`. Na primer:

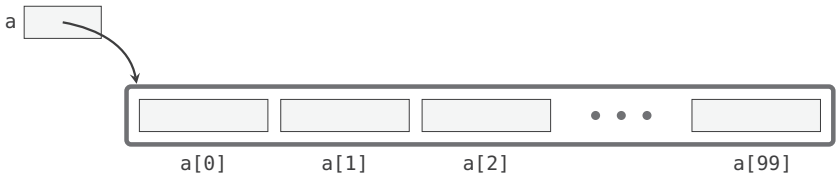
```
a = new int[100];
```

U ovoj naredbi dodele, vrednost 100 u uglastim zagradama iza reči `int` određuje broj elemenata niza koji se konstruiše. Prethodna dva koraka se, kao što je to uobičajeno, mogu kraće pisati jednom naredbom:

```
int[] a = new int[100];
```

Ovom naredbom se alocira promenljiva `a` klasnog tipa `int[]`, konstruiše se objekat niza od 100 elemenata koji su svi primitivnog tipa `int` i referenca na novokonstruisani objekat niza dodeljuje se promenljivoj `a`. Efekat prethodne naredbe je ilustrovan na slici 3.2.

Svaki od 100 elemenata konstruisanog niza `a` je obična promenljiva tipa `int` čiji sadržaj može biti bilo koja celobrojnu vrednost tipa `int`. Pomenuli smo da se često za promenljivu koja ukazuje na neki niz kraće kaže da je ona sâm taj niz. Tako se u ovom primeru kraće kaže „niz `a` od 100 elemenata".



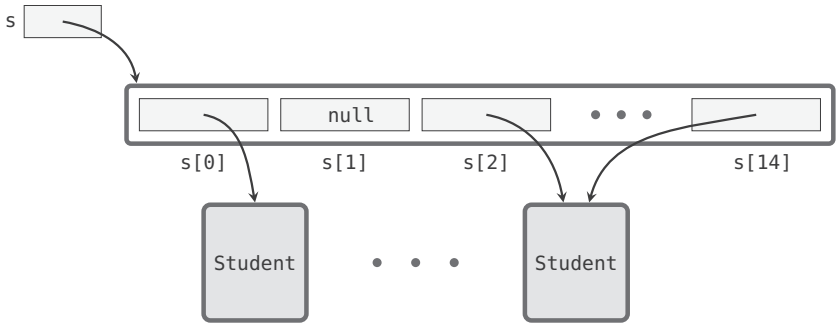
Slika 3.2: Objekat niza a od 100 elemenata celobrojnog tipa.

Međutim, promenljiva a je obična promenljiva klasnog tipa i njena vrednost može biti samo referenca na neki objekat niza, ili specijalna referenca null.

Definicija niza čiji je bazni tip drugačiji primitivni tip od int nije mnogo drugačija — reč int treba zamenti imenom drugog primitivnog tipa i u uglastim zagradama navesti potrebnu dužinu niza. Nešto slično važi i za nizove čiji je bazni tip neki klasni tip. Na primer:

```
Student[] s = new Student[15];
```

Ovom definicijom se alocira promenljiva s klasnog tipa Student[], konstruiše se niz od 15 elemenata klasnog tipa Student i referenca na novi niz se dodeljuje promenljivoj s. Svaki od 15 elemenata ovog niza predstavlja promenljivu klasnog tipa Student čiji sadržaj može biti referenca na objekte klase Student ili null. Jedan primer sadržaja niza s prikazan je na slici 3.3.



Slika 3.3: Objekat niza s sa elementima klasnog tipa Student.

U opštem slučaju, ako je N celobrojni izraz, onda se operatorom new u izrazu

```
new bazni-tip[N]
```

konstruiše niz koji kao objekat pripada klasi *bazni-tip*`[]` i vraća se referenca na taj niz. Izračunata vrednost *n* celobrojnog izraza `N` u uglastim zagradama određuje dužinu tog niza, odnosno broj njegovih elemenata.

Nakon konstruisanja nekog niza, broj elemenata tog niza se ne može promeniti. Prema tome, iako se može pisati, recimo,

```
int k = 5;
double[] x = new double[2*k+10];
```

to ne znači da je broj elemenata realnog niza `x` promenljiv, već je fiksiran u trenutku njegovog konstruisanja vrednošću celobrojnog izraza `2*k+10` u uglastim zagradama koja iznosi 20. U ovom primeru dakle, niz `x` ima tačno 20 elemenata i taj njegov broj elemenata se kasnije ne može niti smanjiti niti povećati.

Svaki niz u Javi, pored svojih elemenata, automatski sadrži i jedno javno celobrojno polje `length` u kojem se nalazi dužina niza. Zato, u prethodnom primeru, u programu se može koristiti polje `x.length` čija je vrednost 20 nakon konstruisanja niza `x`. Vrednost tog polja se naravno ne može menjati u programu. To je obezbeđeno time što je polje `length` svakog niza deklarisano s modifikatorom `final`, pa se ne može menjati nakon inicijalizacije.

Pošto elementi niza u suštini predstavljaju polja objekta niza, ti elementi se u trenutku konstruisanja niza inicijalizuju podrazumevanim vrednostima za bazni tip niza. (Podsetimo se još jednom: podrazumevane vrednosti su nula za numerički tip, `false` za logički tip, `'\u0000'` za znakovni tip i `null` za klasni tip.) Početne vrednosti elemenata niza se mogu i eksplicitno navesti u definiciji niza, unutar vitičastih zagrada i međusobno razdvojene zapetama. Tako, naredbom

```
int[] a = new int[] {1, 2, 4, 8, 16, 32, 64, 128};
```

konstruiše se niz `a` od 8 elementa čije su početne vrednosti 1, 2, 4, 8, 16, 32, 64 i 128. Drugim rečima, element `a[0]` dobija početnu vrednost 1, element `a[1]` dobija početnu vrednost 2 i tako dalje, element `a[7]` dobija početnu vrednost 128.

Opšti oblik operatora `new` u ovom kontekstu je:

```
new bazni-tip[] {lista-vrednosti}
```

U stvari, početni deo `new` *`bazni-tip`* `[]` nije obavezan i često se izostavlja u naredbi definicije niza sa početnim vrednostima. Rezultat prethodnog izraza je referenca na novokonstruisani niz sa elementima koji su inicijalizovani datim vrednostima. Zbog toga se takav izraz može koristiti u programu na svim mestima gde se očekuje niz tipa *`bazni-tip`* `[]`. Na primer, ako je `prikaži()` metod čiji je jedini parametar tipa niza stringova, onda se u programu taj metod može kratko pozvati u obliku:

```
prikaži(new String[] {"Nastavi", "Odustani"});
```

Ovaj primer pokazuje da je konstruisanje „anonimnih" nizova ponekad vrlo korisno. Bez te mogućnosti naime, za neke pomoćne nizove u programu morale bi da se pišu naredbe definicije (i smišljaju imena) za promenljive koje ukazuju na ne toliko važne nizove. Na primer, prethodna jedna naredba bi se morala zameniti ovim naredbama:

```
String[] opcije = new String[] {"Nastavi", "Odustani"};
prikaži(opcije);
```

Ukoliko se elementima niza eksplicitno dodeljuju početne vrednosti prilikom konstruisanja, dužina niza se nigde ne navodi i implicitno se određuje na osnovu broja navedenih vrednosti. Pored toga, te vrednosti ne moraju da budu obične konstante, nego mogu biti promenljive ili proizvoljni izrazi pod uslovom da su njihove vrednosti odgovarajućeg baznog tipa niza. Na primer:

```
Student pera = new Student("Pera Perić", 1111, 99, 100, 100);
Student[] odlikaši = new Student[] {
                pera,
                new Student("Laza Lazić", 2222, 99, 95, 100),
                new Student("Mira Mirić", 3333, 100, 100, 100)
            };
```

Korišćenje nizova

Elementi niza su obične promenljive baznog tipa niza i mogu se koristiti u programu na svim mestima gde je dozvoljena upotreba promenljive baznog tipa niza. Ovi elementi niza se koriste preko svojih indeksa i ime-

na promenljive koja ukazuje na objekat niza. Na primer, jedan niz a čija je definicija

```
int[] a = new int[1000];
```

u programu zapravo obezbeđuje ništa drugo do 1000 celobrojnih promenljivih sa imenima a[0], a[1], ..., a[999].

Puna snaga nizova ipak ne leži samo u mogućnosti lakog dobijanja velikog broja promenljivih za čuvanje velikog broja podataka. Druga odlika nizova koja ih izdvaja među strukturama podataka jeste lakoća rada sa njihovim proizvoljnim elementima. Ova druga mogućnost u radu sa nizovima je posledica činjenice da se za indekse elemenata nizova mogu koristiti proizvoljni celobrojni izrazi. Tako, ako je i promenljiva tipa int, onda su, recimo, a[i] i a[3*i-7] takođe ispravni zapisi elemenata prethodnog niza a. Elementi niza a na koje se odnose ovi zapisi se određuju dinamički tokom izvršavanja programa. Naime, zavisno od konkretne vrednosti promenljive i, element niza na koji se odnosi zapis, recimo, a[3*i-7] dobija se izračunavanjem vrednosti celobrojnog izraza u uglastim zagradama: ako promenljiva i ima vrednost 3, dobija se element a[2]; ako promenljiva i ima vrednost 10, dobija se element a[23] i slično.

Praktičniji primer je sledeća jednostavna petlja kojom se na ekranu prikazuju vrednosti svih elemenata niza a:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Telo ove petlje se sastoji samo od jedne naredbe kojom se prikazuje *i*-ti element niza a. Početna vrednost brojača i je 0, pa se u prvoj iteraciji prikazuje element a[0]. Zatim se izrazom i++ brojač i uvećava za 1 i dobija njegova nova vrednost 1, pa se u drugoj iteraciji prikazuje element a[1]. Ponavljajući ovaj postupak, brojač i se na kraju svake iteracije uvećava za 1 i time se redom prikazuju vrednosti elemenata niza a. Poslednja iteracija koja se izvršava je ona kada je na početku vrednost brojača i jednaka 999 i tada se prikazuje element a[999]. Nakon toga će i uvećanjem za 1 dobiti vrednost 1000 i zato uslov nastavka petlje i < a.length neće biti zadovoljen pošto polje a.length ima vrednost 1000. Time se prekida izvršavanje petlje i završava prikazivanje vrednosti tačno svih elemenata niza a.

Obratite pažnju na jednostavnost gornje petlje kojom je realizovan za-

datak prikazivanja svih elemenata niza `a` — pomoću praktično dva reda se prikazuju vrednosti 1000 promenljivih! Ne samo to, i da niz `a` ima 100000 ili više elemenata, potpuno istom petljom bi se prikazale vrednosti mnogo većeg broja tih elemenata.

U radu sa nizovima u Javi su moguće dve vrste grešaka koje izazivaju prekid izvršavanja programa. Prvo, ako promenljiva `a` tipa niza sadrži vrednost `null`, onda promenljiva `a` čak ni ne ukazuje na neki niz, pa naravno nema smisla koristiti neki element `a[i]` nepostojećeg niza. Drugo, ako promenljiva `a` zaista ukazuje na prethodno konstruisan niz, onda vrednost indeksa `i` za element `a[i]` može pogrešno biti van dozvoljenih granica indeksa niza. To će biti slučaj kada se izračunavanjem indeksa `i` dobije da je `i<0` ili `i>=a.length`.

Primer: prebrojavanje glasova na izborima

Pretpostavimo da posle završetka glasanja na nekim izborima treba prebrojati glasove sa glasačkih listića na jednom biračkom mestu. Umesto ručnog brojanja, koje je podložno greškama, potrebno je naravno napisati program u Javi kojim se glasovi unose i prebrojavaju kako se redom pregledaju glasački listići. Budući da se program piše mnogo pre raspisivanja izbora radi njegovog potpunog testiranja, unapred se ne zna broj partija koji izlazi na izbore, pa taj podatak treba da bude deo ulaza programa.

U listingu 3.1 je prikazan jednostavan program za prebrojavanje glasova na izborima. U tom programu se koristi celobrojni niz `partije` čiji elementi sadrže broj glasova pojedinih partija. Drugim rečima, ako su partije u programu numerisane od 0, onda se za sabiranje njihovih glasova koristi niz `partije` čiji *i*-ti element `partije[i]` sadrži broj glasova *i*-te partije. Primetimo i da su zbog ovoga indeksi niza `partije` u programu morali da budu prilagođeni za 1, pošto su partije na glasačkim listićima prirodno numerisane od 1, a elementi nizova u Javi su numerisani od 0.

Obratite pažnju i na još jedan detalj u listingu 3.1. Naime, prestanak unošenja glasova se programski realizuje unosom glasa za nepostojeću partiju. Ovo možda nije najelegantnije rešenje za taj zadatak, ali je najjednostavnije u ovom malom primeru. A nakon završetka unošenja glasova partija, u programu se prikazuje ukupan broj osvojenih glasova svih partija.

Listing 3.1: Prebrojavanje glasova na izborima

```
/*
 * Program za prebrojavanje glasova partija na biračkom mestu.
 */
import java.util.*;

public class Glasanje {

   public static void main(String[] args) {

      Scanner tastatura = new Scanner(System.in);

      // Učitavanje ukupnog broja partija
      System.out.print("Unesite ukupan broj partija: ");
      int brojPartija = tastatura.nextInt();

      // Konstruisanje niza za sabiranje glasova partija
      int[] partije = new int[brojPartija];

      // Učitavanje i brojanje glasova za pojedinačne partije
      for ( ; ; ) { // beskonačna petlja
         System.out.print("Redni broj partije koja dobija glas> ");
         int p = tastatura.nextInt();
         if (p < 1 || p > brojPartija)
            break;
         else
            partije[p-1] = partije[p-1] + 1;
      }

      // Prikazivanje osvojenih glasova svih partija
      for (int i = 0; i < partije.length; i++) {
         System.out.print("Partija pod rednim brojem " + (i+1));
         System.out.println(" ima " + partije[i] + " glasova.");
      }
   }
}
```

Primer: kopiranje nizova

Ako su `a` i `b` promenljive istog tipa, onda se naredbom dodele `a = b` sadržaj promenljive `b` prepisuje u promenljivu `a`. Ovaj efekat u slučaju promenljivih `a` i `b` nizovnog tipa ponekad nije odgovarajući, nego je potrebno napraviti još jednu identičnu kopije svih elemenata niza `b` na koje ukazuje promenljiva `a`. Konkretnije, ako je konstruisan niz `b` naredbom, recimo,

```
double[] b = new double[20];
```

i ako je deklarisana promenljiva `a` tipa istog tog niza, recimo,

```
double[] a;
```

onda se naredbom dodele

```
a = b;
```

u promenljivu `a` prepisuje samo referenca iz promenljive `b`. To znači da na konstruisani objekat niza na koji je prvobitno ukazivala promenljiva `b`, sada ukazuju obe promenljive `a` i `b`, ali pri tome i dalje postoji samo po jedan primerak svakog elementa niza. Zbog toga se ti elementi sada mogu koristiti na dva načina: zapisi `a[i]` i `b[i]` se odnose na jedinstveni *i*-ti element niza. Ukoliko je potrebno fizički kopirati svaki element niza, to se mora uraditi ručno. (Ili upotrebom posebnog metoda standardne klase `Arrays`.) Radi ilustracije, u nastavku je prikazan poseban metod kojim se ovaj problem rešava za nizove tipa `double`.

Bilo koji tip niza u Javi je klasni tip kao i svaki drugi, pa se može koristiti na sve načine na koje se mogu koristiti tipovi u Javi. Specifično, tip niza može biti tip nekog parametra metoda, kao i tip rezultata metoda. Upravo se ova činjenica koristi za metod kojim se fizički kopiraju svi elementi jednog niza u drugi:

```
public static double[] kopirajNiz(double[] original) {
   if (original == null)
      return null;
   double[] kopija = new double[original.length];
   for (int i = 0; i < kopija.length; i++)
      kopija[i] = original[i];
   return kopija;
}
```

Ako su `a` i `b` promenljive koje su definisane kao na početku ovog primera, onda se naredbom dodele

```
a = kopirajNiz(b);
```

dobija novi niz na koji ukazuje `a`. Elementi novog niza su fizičke kopije elementa niza na koji ukazuje `b` tako da se sada zapisi `a[i]` i `b[i]` odnose na različite *i*-te elemente odgovarajućih nizova.

Primer: argumenti programa u komandnom redu

Pokretanje nekog programa radi izvršavanja od strane Java interpretatora postiže se navođenjem njegove glavne klase kao argumenta Java interpretatora. Tačan način kako se to izvodi na računaru zavisi od razvojnog okruženja pod čijom kontrolom se piše Java program, ali na kraju se to svodi na jednu komandu operativnog sistema računara u obliku:

```
java glavna-klasa
```

Ovom komandom se pokreće Java interpretator `java` koji sa svoje strane počinje izvršavanje programa pozivom specijalnog metoda `main()` koji se nalazi u navedenoj glavnoj klasi.

Metod `main()` se dakle ne poziva direktno u nekom drugom metodu programa, nego ga indirektno poziva Java interpretator na samom početku izvršavanja celog programa. Kao što su čitaoci verovatno primetili, zaglavlje u definiciji metoda `main()` ima oblik koji u svim primerima do sada nije menjan:

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

To nije slučajno, jer metod `main()` mora pre svega imati službeno ime kako bi Java interpretator pozvao upravo taj metod radi izvršavanja programa. Taj metod mora biti definisan i sa modifikatorom `public`, jer inače metod `main()` ne bi bio uopšte dostupan Java interpretatoru koji ga poziva. Taj metod mora biti definisan i sa modifikatorom `static`, jer bi inače metod `main()` mogao da se pozove samo uz neki objekat glavne klase, a takav objekat ne može biti konstruisan pre početka izvršavanja programa.

Pored toga, iz zaglavlja se može uočiti da metod `main()` ima jedan parametar `args` tipa `String[]`.[1] To znači da se metodu `main()` prilikom poziva može kao argument preneti niz stringova. Ali budući da se metod `main()` poziva implicitno, kako se kao njegov argument može navesti neki niz stringova koje treba preneti glavnom metodu? Ovo se postiže pisanjem željenog niza stringova u komandnom redu iza glavne klase čiji se metod `main()` poziva:

```
java glavna-klasa niz-stringova
```

Metod `main()` dobija dakle vrednosti za parametar `args` iz komandnog reda tako što Java interpretator automatski inicijalizuje niz `args` stringovima koji su navedeni iza glavne klase u komandnom redu, a u polje `args.length` se upisuje broj tih stringova. Ako nije ništa navedeno, dužina niza `args` dobija vrednost 0. Kao praktičan primer, u listingu 3.2 je prikazan program kojim se samo prikazuje niz stringova naveden kao argument tog programa u komandnom redu.

Listing 3.2: Argumenti programa u komandnom redu

```
public class Poruka {

   public static void main(String[] args) {

      // Da li su navedeni argumenti u komandnom redu?
      if (args.length == 0)
         return;

      // Ispitivanje prvog argumenta u komandnom redu
      if (args[0].equals("-d"))
         System.out.print("Dobar dan");
      else if (args[0].equals("-z"))
         System.out.print("Zbogom");
      else
         return;
```

[1] Ime parametra `args` metoda `main()` je tradicionalno, ali proizvoljno, i jedino se ono može promeniti u zaglavlju metoda `main()`.

```
      // Prikazivanje ostalih argumenata u komandnom redu
      for (int i = 1; i < args.length; i++)
         System.out.print(" " + args[i]);
      System.out.println("!");
   }
}
```

Ako se program u listingu 3.2 izvrši komandom

```
java Poruka -z okrutni svete
```

onda elementi niza `args` u metodu `main()` dobijaju sledeće vrednosti:

```
args[0] = "-z";
args[1] = "okrutni";
args[2] = "svete";
```

Zbog toga, izvršavanjem prethodnog programa se na ekranu prikazuje ova poruka:

```
Zbogom okrutni svete!
```

Ako se pak program izvrši komandom

```
java Poruka -d tugo
```

onda elementi niza `args` u metodu `main()` dobijaju sledeće vrednosti:

```
args[0] = "-d";
args[1] = "tugo";
```

U ovom slučaju se izvršavanjem prethodnog programa na ekranu prikazuje druga poruka:

```
Dobar dan tugo!
```

Klasa `Arrays`

Radi lakšeg rada sa nizovima, u Javi se može koristiti standardna klasa `Arrays` iz paketa `java.util`. Ova pomoćna klasa sadrži nekoliko statičkih metoda koji obezbeđuju korisne operacije nad nizovima čiji je bazni tip jedan od primitivnih tipova. U nastavku je naveden nepotpun spisak i opis

ovih metoda, pri čemu oznaka *tip* ukazuje na jedan od primitivnih tipova. Izuzetak su jedino metodi `sort()` i `binarnySearch()` kod kojih nije dozvoljeno da to bude primitivni tip `boolean`.

- `String toString(`*tip*`[] a)` — vraća se reprezentacija niza `a` u obliku stringa. U tom obliku, vrednosti svih elemenata niza `a` se nalaze unutar uglastih zagrada po redu njihovih pozicija u nizu i međusobno su razdvojene zapetama.
- *tip*`[] copyOf(`*tip*`[] a, int n)` — vraća se nova kopija niza `a` koja se sastoji od prvih `n` njegovih elemenata. Ako je je vrednost `n` veća od vrednosti `a.length`, onda se višak elemenata kopije niza inicijalizuje nulom ili vrednošću `false`. U suprotnom slučaju, kopira se samo početnih `n` elemenata niza `a`.
- *tip*`[] copyOfRange(`*tip*`[] a, int i, int j)` — vraća se nova kopija niza `a` od indeksa `i` do indeksa `j`. Element sa indeksom `i` niza `a` se uključuje, dok se onaj sa indeksom `j` ne uključuje u novi niz. Ako je indeks `j` veći od `a.length`, višak elemenata kopije niza se inicijalizuje nulom ili vrednošću `false`.
- `void sort(`*tip*`[] a)` — sortira se niz `a` u mestu u rastućem redosledu.
- `int binarnySearch(`*tip*`[] a, `*tip*` v)` — pronalazi se data vrednost `v` u sortiranom nizu `a` korišćenjem binarne pretrage. Ako je vrednost `v` nađena u nizu, vraća se indeks odgovarajućeg elementa. U suprotnom slučaju, vraća se negativna vrednost k tako da $-k - 1$ odgovara poziciji gde bi data vrednost trebalo da se nalazi u sortiranom nizu.
- `void fill(`*tip*`[] a, `*tip*` v)` — dodeljuje se svim elementima niza `a` vrednost `v`.
- `boolean equals(`*tip*`[] a, `*tip*`[] b)` — vraća se vrednost `true` ukoliko nizovi `a` i `b` imaju istu dužinu i jednake odgovarajuće elemente. U suprotnom slučaju, vraća se vrednost `false`.

Primer: izvlačenje loto brojeva

Radi ilustracije primene klase `Arrays` u radu sa nizovima, u listingu 3.3 je prikazan program koji simulira izvlačenje brojeva u igri na sreću loto. U

igri loto se izvlači 7 slučajnih, međusobno različitih brojeva među celim brojevima od 1 do 39. U programu se generišu 7 takvih brojeva koji predstavljaju jedno kolo igre loto. (Drugi ugao gledanja na rezultat programa je da generisane brojeve čitaoci mogu igrati u stvarnoj igri loto da bi osvojili neku od nagrada.)

U programu se koristi konstanta *N* za označavanje dužine niza svih mogućih brojeva, kao i konstanta *K* za označavanje dužine niza brojeva jednog kola izvlačenja. Naravno, njihove vrednosti 39 i 7 mogle bi se i direktno koristiti u programu, ali onda bi se program teže mogao prilagoditi za neka druga pravila igre loto. Na primer, u nekim varijantama se izvlači 5 brojeva od 36.

Svi brojevi koji učestvuju u jednom kolu igre loto predstavljeni su celobrojnim nizom `bubanj`. Ovaj niz sadrži brojeve od 1 do *N* i od njih treba izvući *K* slučajnih brojeva. Za sekvencijalno izvlačenje slučajnih brojeva se koristi metod `Math.random()`, ali problem je to što se time ne moraju obavezno dobiti različiti izvučeni brojevi. Zbog toga je niz `bubanj` podeljen u dva dela: u levom delu su brojevi koji su preostali za izvlačenje, a u desnom delu su oni brojevi koji su već izvučeni. Granica između dva dela niza je određena vrednošću promenljive `m` koja sadrži indeks poslednjeg elementa levog dela. Na početku pre prvog izvlačenja, vrednost promenljive `m` je jednaka indeksu poslednjeg elementa niza `bubanj`.

Za generisanje slučajnog broja između 1 i *N* generiše se, u stvari, slučajni indeks levog dela niza `bubanj` i uzima broj u elementu niza `bubanj` koji se nalazi na toj poziciji. Zatim se taj element međusobno zamenjuje sa poslednjim elementom levog dela niza `bubanj` i pomera ulevo granica između dva dela niza smanjivanjem vrednosti `m` za jedan. Ponavljanjem ovog postupka *K* puta za svaki izvučeni broj, na kraju se u desnom delu niza `bubanj`, od indeksa *N* – *K* do kraja, nalaze svi izvučeni slučajni brojevi.

Primetimo da je redosled izvučenih brojeva u desnom delu niza `bubanj` proizvoljan. Zbog toga, da bi se izvučeni brojevi prikazali u rastućem redosledu, desni deo niza `bubanj` se kopira u novi niz `kombinacija` i ovaj niz se sortira odgovarajućim metodom klase `Arrays`.

Listing 3.3: Izvlačenje loto brojeva

```
import java.util.*;

public class Loto {

   public static void main (String[] args) {

      final int N = 39;          // ukupan broj svih loto brojeva
      final int K = 7;           // ukupan broj izvučenih brojeva
      int[] bubanj = new int[N]; // niz sa svim loto brojevima

      // Inicijalizacija niza brojevima 1, 2, ..., N
      for (int i = 0; i < N; i++)
         bubanj[i] = i + 1;

      int m;  // granica levog i desnog dela niza

      // Izvlačenje k brojeva i premeštanje u desni deo niza
      for (m = N-1; m > N-K-1; m--) {

         // Generisanje slučajnog indeksa levog dela niza
         int i = (int) (Math.random() * (m+1));

         // Međusobna zamena slučajnog elementa i poslednjeg
         // elementa levog dela niza
         int broj = bubanj[i];
         bubanj[i] = bubanj[m];
         bubanj[m] = broj;
      }

      // Kopiranje izvučenih brojeva u novi niz
      int[] kombinacija = Arrays.copyOfRange(bubanj, m+1, N);

      // Sortiranje novog niza
      Arrays.sort(kombinacija);
```

```
      // Prikazivanje izvučenih brojeva u rastućem redosledu
      System.out.println("Dobitna kombinacija je: ");
      for (int i = 0; i < kombinacija.length; i++)
         System.out.print(kombinacija[i] + "  ");
      System.out.println();
   }
}
```

Naredba `for-each`

U verziji Java 5 je dodat novi oblik `for` petlje kojom se omogućuje lakši rad sa *svim* elementima nekog niza. Ta petlja se popularno naziva `for-each` petlja, iako se reč *each* ne pojavljuje u njenom zapisu. U stvari, `for-each` petlja se može koristiti ne samo za nizove, nego i za opštije strukture podataka u kojima je definisana operacija „sledbenika" za svaki element strukture.

Ako je *niz* tipa *bazni-tip* `[]`, onda `for-each` petlja za *niz* ima opšti oblik:

```
for (bazni-tip elem : niz) {
   .
   . // Obrada aktuelnog elementa elem
   .
}
```

Obratite pažnju na to da su dve tačke u kontrolnom delu ove petlje obavezne. Pored toga, *elem* je kontrolna promenljiva baznog tipa koja se mora definisati unutar kontrolnog dela petlje. Prilikom izvršavanja `for-each` petlje, kontrolnoj promenljivoj *elem* se redom dodeljuje vrednost svakog elementa niza i izvršava se telo petlje za svaku tu vrednost. Preciznije, prethodni oblik `for-each` petlje za nizove je ekvivalentan sledećoj običnoj `for` petlji:

```
for (int i = 0; i < niz.length; i++) {
   bazni-tip elem = niz[i];
   .
   . // Obrada aktuelnog elementa elem
   .
}
```

Na primer, sabiranje svih pozitivnih elemenata niza `a` tipa `double[]` može se uraditi `for-each` petljom na sledeći način:

```
double zbir = 0;
for (double e : a) {
   if (e > 0)
      zbir = zbir + e;
}
```

Ili, prikazivanje izvučenih loto brojeva u programu na strani 107 može se uraditi na elegantniji način:

```
for (int broj : kombinacija)
   System.out.print(broj + "  ");
```

Naglasimo da je `for-each` petlja korisna u slučajevima kada treba obraditi sve elemente nekog niza, jer se ne mora voditi računa o indeksima i granici tog niza. Međutim, ona nije od velike pomoći ako treba nešto uraditi samo sa nekim elementima niza, a ne sa svim elementima.

Obratite pažnju i na to da se u telu `for-each` petlje zapravo samo čitaju vrednosti elemenata niza (i dodeljuju kontrolnoj promenljivoj), dok upisivanje vrednosti u elemente niza nije moguće. Na primer, ako treba svim elementima konstruisanog celobrojnog niza `a` dodeliti neku vrednost, recimo 17, onda bi bilo pogrešno napisati:

```
for (int e : a) {
   e = 17;
}
```

U ovom slučaju se kontrolnoj promenljivoj `e` dodeljuju redom vrednosti elemenata niza `a`, pa se odmah zatim istoj promenljivoj `e` dodeljuje vrednost 17. Međutim, to nema nikakav efekat na elemente niza i njihove vrednosti ostaju nepromenjene.

Metodi sa promenljivim brojem argumenata

Od verzije Java 5 je omogućeno pozivanje metoda sa promenljivim brojem argumenata. Metod `printf()` za formatizovano prikazivanje vrednosti na ekranu predstavlja primer metoda sa promenljivim brojem argumena-

ta. Naime, prvi argument metoda `printf()` mora biti tipa `String`, ali ovaj metod može imati proizvoljan broj dodatnih argumenata bilo kog tipa.

Poziv metoda sa promenljivim brojem argumenata se ne razlikuje od poziva drugih metoda, ali njihova definicija zahteva malo drugačiji način pisanja. Ovo se najbolje može razumeti na jednom primeru, pa pretpostavimo da treba napisati metod `prosek()` kojim se izračunava i vraća prosek bilo kog broja vrednosti tipa `double`. Prema tome, pozivi ovog metoda mogu imati promenljiv broj argumenata, na primer:

- `prosek(1, 2, 3, 4)`
- `prosek(1.41, Math.PI, 2.3)`
- `prosek(Math.sqrt(3))`
- `prosek()`

Ovde su dakle u prvom pozivu navedena četiri argumenta, u drugom pozivu tri argumenta, u trećem pozivu jedan argument, a u poslednjem pozivu nula argumenata.

Zaglavlje definicije metoda `prosek()` mora pretrpeti male izmene u listi parametara u zagradi u odnosu na uobičajeni zapis. Na primer:

```
public static double prosek(double... brojevi) {
    .
    . // Telo metoda
    .
}
```

Tri tačke iza tipa `double` parametra `brojevi` u zagradama ukazuju da se na mesto tog parametra može navesti promenljiv broj argumenata u pozivu metoda. Prilikom poziva metoda `prosek()`, pridruživanje više argumenata parametru `brojevi` razrešava se tako što se implicitno najpre konstruiše niz `brojevi` tipa `double[]` čija je dužina jednaka broju navedenih argumenata, a zatim se elementi tog niza inicijalizuju ovim argumentima. To znači da se telo metoda `prosek()` mora pisati pod pretpostavkom da je na raspolaganju konstruisan niz `brojevi` tipa `double[]`, da se aktuelni broj njegovih elemenata nalazi u polju `brojevi.length` i da se vrednosti stvarnih argumenata nalaze u elementima `brojevi[0]`, `brojevi[1]` i tako dalje. Imajući ovo u vidu, kompletna definicija metoda `prosek()` je:

```
public static double prosek (double... brojevi) {
   double zbir = 0;
   for (int i = 0; i < brojevi.length; i++)
      zbir = zbir + brojevi[i];
   return zbir / brojevi.length;
}
```

Još bolje, ukoliko se koristi `for-each` petlja, dobija se elegantnije rešenje:

```
public static double prosek (double... brojevi) {
   double zbir = 0;
   for (double broj : brojevi)
      zbir = zbir + broj;
   return zbir / brojevi.length;
}
```

Sličan postupak pridruživanja promenljivog broja argumenata nekom parametru metoda se primenjuju i u opštem slučaju. Naime, ako je taj parametar tipa `T`, u trenutku poziva se konstruiše odgovarajući niz tipa `T[]` i njegovi elementi se inicijalizuju datim argumentima.

Primetimo da parametar kojem odgovara promenljiv broj argumenata (tj. onaj iza čijeg tipa se nalaze tri tačke) mora biti poslednji parametar u zaglavlju definicije metoda. Razlog za ovo je prosto to što se u pozivu metoda podrazumeva da njemu odgovaraju svi navedeni argumenti do kraja, odnosno do zatvorene zagrade.

Obratite pažnju i na to da se u pozivu, na mesto parametra kome odgovara promenljiv broj argumenata, može navesti stvarni niz, a ne lista pojedinačnih vrednosti. Tako, u prethodnom primeru, ako je u programu prethodno konstruisan niz `ocene` tipa `double[]`, onda je ispravan i poziv `prosek(ocene)` radi dobijanja proseka vrednosti ocena u nizu.

3.2 Dvodimenzionalni nizovi

Nizovi o kojima smo govorili do sada poseduju samo jednu „dimenziju" — dužinu. Obični nizovi se zato često nazivaju jednodimenzionalni nizovi. U Javi se mogu koristiti i nizovi koji imaju dve dimenzije — dužinu i visinu, ili tri dimenzije — dužinu, visinu i dubinu, i tako dalje. Zato se o

ovim nizovima govori kao o dvodimenzionalnim, trodimenzionalnim ili, jednom rečju, višedimenzionalnim nizovima.

U Javi se svaki tip podataka može koristiti za bazni tip nekog (jednodimenzionalnog) niza. Specifično, kako je neki tip niza takođe običan tip u Javi, ukoliko je bazni tip nekog niza baš takav tip, dobija se niz nizova. Na primer, jedan celobrojni niz ima tip `int[]`, a to znači da automatski postoji i tip `int[][]` koji predstavlja niz celobrojnih nizova. Ovaj niz nizova se kraće zove dvodimenzionalni niz.

Ništa nije neobično da se dalje posmatra i tip `int[][][]` koji predstavlja niz dvodimenzionalnih nizova, odnosno trodimenzionalni niz. Naravno, ova linija razmišljanja se može nastaviti i tako se dobijaju višedimenzionalni nizovi, mada se nizovi čija je dimenzija veća od tri zaista retko primenjuju.

U daljem tekstu se ograničavamo samo na dvodimenzionalne nizove, jer se svi koncepti u vezi s njima lako proširuju na više dimenzija. Dvodimenzionalni nizovi se popularno nazivaju i *matrice* ili *tabele*. Definisanje promenljive tipa dvodimenzionalnog niza je slično definisanju običnog, jednodimenzionalnog niza — razlika je samo u dodatnom paru uglastih zagrada. Naredbom, na primer,

```
int[][] a;
```

definiše se promenljiva `a` čiji sadržaj može biti referenca na objekat dvodimenzionalnog niza tipa `int[][]`. Konstruisanje aktuelnog dvidimenzionalnog niza se vrši operatorom `new` kao i u jednodimenzionalnom slučaju. Na primer, naredbom dodele

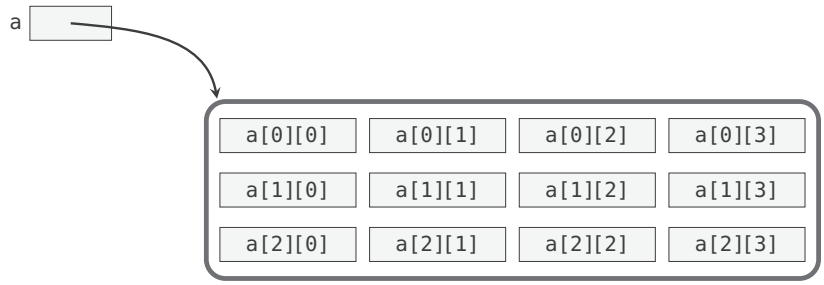
```
a = new int[3][4];
```

konstruiše se dvodimenzionalni niz veličine 3×4 i referenca na njega se dodeljuje prethodno definisanoj promenljivoj `a`. Kao i obično, ove dve posebne naredbe se mogu spojiti u jednu:

```
int[][] a = new int[3][4];
```

Na slici 3.4 je prikazan izgled relevantne memorije računara nakon izvršavanja ove naredbe dodele.

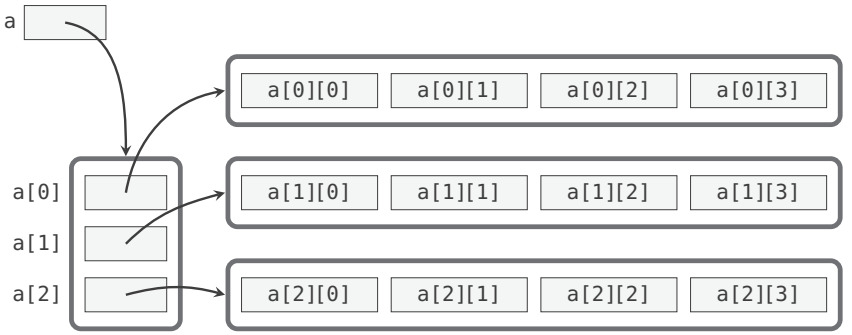
Slika 3.4 pokazuje da se jedan dvodimenzionalni niz na koji ukazuje promenljiva `a` može najbolje zamisliti kao matrica (ili tabela) elemenata koja ima tri vrste i četiri kolone. Elementi matrice su obične, u ovom slučaju



Slika 3.4: Dvodimenzionalni celobrojni niz a veličine 3×4.

celobrojne, promenljive koje imaju dvostruke indekse `a[i][j]`: prvi indeks pokazuje vrstu i drugi indeks kolonu u kojima se element nalazi u matrici. Pri tome treba imati u vidu da, kao što je to uobičajeno u Javi, numeracija vrsta i kolona matrice ide od nule, a ne od jedan.

Tačnije govoreći, dvodimenzionalni niz `a` u prethodnom primeru nije matrica, nego je zaista niz celobrojnih nizova. Tako, izrazom `new int[3][4]` zapravo se konstruiše niz od tri celobrojna niza, od kojih svaki ima četiri elementa. Pravi izgled dvodimenzionalnog niza `a` je zato onaj koji je prikazan na slici 3.5.



Slika 3.5: Prava slika dvodimenzionalnog niza `a` dimenzije 3×4.

Prava slika dvodimenzionalnog niza je komplikovanija za razumevanje i, srećom, može se u većini slučajeva zanemariti i dvodimenzionalni niz smatrati matricom elemenata. Ponekad je ipak potrebno znati da svaka vrsta

matrice predstavlja zapravo jedan niz za sebe. Ovi nizovi u prethodnom primeru su tipa `int[]` i na njih ukazuju promenljive čija su imena `a[0]`, `a[1]` i `a[2]`. Te promenljive i nizovi se u programu mogu koristiti na svim mestima gde su dozvoljeni obični celobrojni nizovi. Na primer, jedna vrsta matrice može biti argument u pozivu metoda čiji je parametar tipa `int[]`.

Zbog prave slike dvodimenzionalnog niza treba imati na umu da vrednost polja `a.length` u prethodnom primeru iznosi tri, odnosno jednaka je ukupnom broju vrsta matrice `a`. Ako je potrebno dobiti broj kolona matrice, onda je to dužina jednodimenzionalnih nizova od kojih se sastoji svaka vrsta matrice, odnosno `a[0].length`, `a[1].length` ili `a[2].length`. U stvari, sve vrste matrice ne moraju biti jednake dužine i u nekim složenijim primenama se koriste matrice sa različitim dužinama vrsti.

Elementi dvodimenzionalnog niza se inicijalizuju uobičajenim podrazumevanim vrednostima nakon konstruisanja takvog niza. To se može promeniti pisanjem svih početnih vrednosti iza operatora `new`, slično načinu na koji se to postiže kod jednodimenzionalnih nizova. Pri tome treba voditi računa da se početne vrednosti matrice navode po vrstama, odnosno redom za jednodimenzionalne nizove vrsta matrice u vitičastim zagradama. Na primer:

```
int[][] a = new int[][] {{ 1, -1,  0,  0},
                         {10, 17, -2, -3},
                         { 0,  1,  2,  3}
                        };
```

Korišćenje dvodimenzionalnih nizova u programu se, slično jednodimenzionalnom slučaju, zasniva na dvostrukim indeksima pojedinih elemenata matrice. Ovi indeksi mogu biti bilo koji celobrojni izrazi i njihovim izračunavanjem se dobijaju brojevi vrste i kolone u kojima se nalazi odgovarajući element matrice.

U radu sa dvodimenzionalnim nizovima se često koriste ugnježđene `for` petlje tako što se u spoljašnjoj petlji prate vrste matrice, a u unutrašnjoj petlji se prate kolone matrice. Na taj način se za svaku aktuelnu vrstu matrice obrađuju svi elementi te vrste, a kako se ovo ponavlja za sve vrste matrice, time se obrađuju tačno svi elementi matrice redom vrsta po vrsta. Na primer, ukoliko za matricu

```
double[][] a = new double[10][10];
```

treba elementima te matrice na glavnoj dijagonali dodeliti vrednost 1 i ostalim elementima vrednost 0, to se može uraditi na sledeći način:

```
for (int i = 0; i < a.length; i++) {       // za svaku vrstu
                                           // u matrici
   for (int j = 0; j < a[i].length; j++) { // i za svaku kolonu
                                           // u aktuelnoj vrsti
      if (i == j)  // da li je element na glavnoj dijagonali?
         a[i][j] = 1;
      else
         a[i][j] = 0;
   }
}
```

Na sličan način se mogu sabrati svi elementi matrice `a`:

```
double zbir = 0;
for (int i = 0; i < a.length; i++)
   for (int j = 0; j < a[i].length; j++)
      zbir = zbir + a[i][j];
```

Sabiranje svih elemenata matrice `a` može se postići i ugnježđenim `for-each` petljama:

```
double zbir = 0;
for (double[] vrsta : a)
   for (double e : vrsta)
      zbir = zbir + e;
```

Primer: rad sa tabelarnim podacima

Dvodimenzionalni nizovi su naročito korisni za smeštanje podataka koji se prirodno predstavljaju u obliku tabele po vrstama i kolonama. Radi konkretnosti, razmotrimo primer firme ABC sa 10 prodavnica koja vodi podatke o mesečnom profitu svake prodavnice tokom neke godine. Tako, ako su prodavnice numerisane od 0 do 9 i meseci u godini od 0 do 11, onda se ovi podaci o profitu mogu predstaviti matricom `profit` na sledeći način:

```
double[][] profit = new double[10][12];
```

Na ovaj način su vrstama matrice `profit` obuhvaćene sve prodavnice firme, dok kolone te matrice ukazuju na mesece godine. Element matrice, recimo, `profit[5][2]` označava vrednost profita koji je ostvarila prodavnica broj 5 u mesecu martu. Opštije, element matrice `profit[i][j]` sadrži vrednost profita koji je ostvarila *i*-ta prodavnica u *j*-tom mesecu (sa numeracijom prodavnica i meseca od 0). Jednodimenzionalni niz `profit[i]`, koji predstavlja *i*-tu vrstu dvodimenzionalnog niza `profit`, sadrži dakle vrednosti profita koji je ostvarila *i*-ta prodavnica tokom cele godine po mesecima.

Na osnovu podataka u matrici `profit` mogu se dobiti različiti analitički pokazatelji o poslovanju firme ABC. Ukupni godišnji profit firme, na primer, može se izračunati na sledeći način:

```
public double godišnjiProfit() {

   double godProfit = 0;
   for (int i = 0; i < 10; i++)
      for (int j = 0; j < 12; j++)
         godProfit += profit[i][j];
   return godProfit;
}
```

Često je potrebno obraditi i samo pojedine vrste ili pojedine kolone matrice podataka. Da bi se, recimo, izračunao ukupni profit svih prodavnica u datom mesecu, treba sabrati vrednosti profita u koloni koja odgovara tom mesecu:

```
private double mesečniProfit(int m) {

   double mesProfit = 0;
   for (int i = 0; i < 10; i++)
      mesProfit += profit[i][m];
   return mesProfit;
}
```

Sledeći metod se može iskoristiti za prikazivanje ukupnog profita svih prodavnica po svim mesecima:

```
public void prikažiProfitPoMesecima() {

   if (profit == null) {
      System.out.println("Greška: podaci ne postoje!");
      return;
   }

   System.out.println("Ukupni profit prodavnica po mesecima:");
   for (int m = 0; m < 12; m++)
      System.out.printf("%6.2f", mesečniProfit(m));
   System.out.println();
}
```

Korisno je imati i pokazatelj ukupnog profita pojedinih prodavnica za celu godinu. To prevedeno za matricu profita znači da treba formirati jednodimenzionalni niz čiji elementi predstavljaju zbir vrednosti pojedinih vrsta matrice. Ovaj postupak i prikazivanje dobijenih vrednosti godišnjeg profita po prodavnicama obuhvaćeni su sledećim metodom:

```
public void prikažiProfitPoProdavnicama() {

   if (profit == null) {
      System.out.println("Greška: podaci ne postoje!");
      return;
   }

   double[] profitProdavnice = new double[n];
   for (int i = 0; i < 10; i++)
      for (int j = 0; j < 12; j++)
         profitProdavnice[i] += profit[i][j];
   System.out.println("Ukupni profit firme po prodavnicama:");
   for (int i = 0; i < 10; i++) {
      System.out.print("Prodavnica " + i + ": ");
      System.out.printf("%7.2f", profitProdavnice[i]);
      System.out.println();
   }
}
```

U listingu 3.4 je prikazan kompletan program kojim se korisniku nude

razne opcije za rad sa podacima o profitu neke firme. Program se sastoji od dve klase. Prva, glavna klasa programa služi za izbor pojedinih opcija iz korisničkog menija radi prikazivanja različitih pokazatelja o profitu jedne firme na ekranu. Druga klasa `Firma` opisuje konkretnu firmu i sadrži polje `n` za broj njenih prodavnica, kao i polje `profit` koje ukazuje na njenu matricu profita. Pored ovih atributa, klasa `Firma` sadrži i prethodne primere metoda kojima se manipuliše matricom profita konkretne firme.

Listing 3.4: Profit firme sa više prodavnica

```
import java.util.*;

public class ProfitFirme {

   public static void main(String[] args) {

      System.out.print("Program za rad sa tabelom profita ");
      System.out.println("neke firme sa više prodavnica.");
      System.out.println();

      Firma abc = new Firma(10);  // firma sa 10 prodavnica
      Scanner tastatura = new Scanner(System.in);
      int brojOpcije;

      do {
         prikažiMeni();

         brojOpcije = tastatura.nextInt();

         switch (brojOpcije) {
            case 1:
               abc.unesiProfit();
               break;
            case 2:
               abc.prikažiProfit();
               break;
            case 3:
```

```
                if (abc.getProfit() == null)
                   System.out.println("Greška: podaci ne postoje!");
                else {
                   System.out.print("Ukupni godišnji profit firme:");
                   System.out.printf("%8.2f", abc.godišnjiProfit());
                   System.out.println();
                }
                break;
             case 4:
                abc.prikažiProfitPoMesecima();
                break;
             case 5:
                abc.prikažiProfitPoProdavnicama();
                break;
             case 0:
                System.out.println("Kraj programa ...");
                break;
             default:
                System.out.println("Greška: pogrešna opcija!");
          }
       } while (brojOpcije != 0);
   }

   private static void prikažiMeni() {

      System.out.println();
      System.out.println("Izaberite jednu od ovih opcija:");
      System.out.println("  1. Unos tabele profita");
      System.out.println("  2. Prikaz tabele profita");
      System.out.println("  3. Prikaz ukupnog godišnjeg profita");
      System.out.println("  4. Prikaz profita po mesecima");
      System.out.println("  5. Prikaz profita po prodavnicama");
      System.out.println("  0. Kraj rada");
      System.out.print("Unesite broj opcije: ");
   }
}
```

```
class Firma {

   private int n;              // broj prodavnica firme
   private double[][] profit;  // matrica profita firme

   // Konstruktor
   public Firma(int n) {
      this.n = n;
   }

   // Geter metod za polje profit
   public double[][] getProfit() {
      return profit;
   }

   public void unesiProfit() {

      profit = new double[n][12];
      Scanner tastatura = new Scanner(System.in);

      for (int i = 0; i < n; i++)
         for (int j = 0; j < 12; j++) {
            System.out.print("Unesite profit prodavnice " + i);
            System.out.print(" za mesec " + j + ": ");
            profit[i][j] = tastatura.nextDouble();
         }
   }

   public void prikaziProfit() {

      if (profit == null) {
         System.out.println("Greška: podaci ne postoje!");
         return;
      }

      System.out.println(
         "Tabela profita po prodavnicama i mesecima:");
```

```
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < 12; j++)
            System.out.printf("%6.2f", profit[i][j]);
         System.out.println();
      }
   }

   .
   . // Ostali metodi klase: godišnjiProfit(), mesečniProfit(),
   . // prikažiProfitPoMesecima(), prikažiProfitPoProdavnicama()
   .
}
```

3.3 Dinamički nizovi

Broj elemenata običnog niza je fiksiran u trenutku konstruisanja niza i ne može se više menjati. Dužina niza određuje dakle maksimalan broj podataka koji se pojedinačno čuvaju u elementima niza, ali svi elementi niza ne moraju biti iskorišćeni u svakom trenutku izvršavanja programa. Naime, u mnogim primenama broj podataka koji se čuvaju u elementima niza varira tokom izvršavanja programa. Jedan primer je program za pisanje dokumenata u kojem se pojedinačni redovi teksta dokumenta čuvaju u nizu tipa `String[]`. Drugi primer je program za neku kompjutersku igru preko Interneta u kojem se koristi niz čiji elementi sadrže igrače koji trenutno učestvuju u igri. U ovim i sličnim primerima je karakteristično to što postoji niz (relativno velike) fiksne dužine, ali su elementi niza iskorišćeni samo delimično.

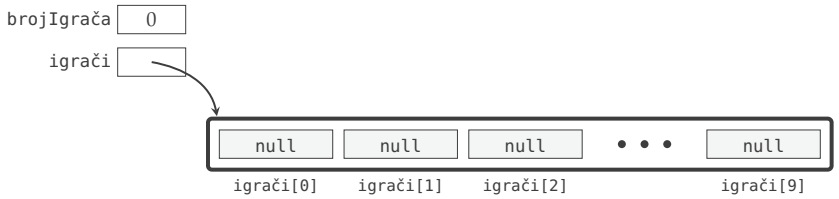
Ovaj način korišćenja nizova ima nekoliko nedostatka. Manji problem je to što se u programu mora voditi računa o tome koliko je zaista iskorišćen niz. Za tu svrhu se može uvesti dodatna promenljiva čija vrednost ukazuje na indeks prvog slobodnog elementa niza.

Ostali problemi se mogu lakše razumeti na konkretnom primeru, recimo, kompjuterske igre preko Interneta kojoj se igrači mogu priključiti ili je

mogu napustiti u svakom trenutku. Ako pretpostavimo da klasa `Igrač` opisuje pojedinačne igrače te igre, onda igrači koji se trenutno igraju mogu biti predstavljeni nizom `igrači` tipa `Igrač[]`. Kako je broj igrača promenljiv, potrebno je imati dodatnu promenljivu `brojIgrača` koja sadrži aktuelni broj igrača koji učestvuju u igri. Pod pretpostavkom da nikad neće biti više od 10 igrača istovremeno, relevantne deklaracije u programu su:

```
Igrač[] igrači = new Igrač[10];  // najviše 10 igrača
int brojIgrača = 0;  // broj igrača je 0 na početku
```

Izgled odgovarajuće memorije u programu posle izvršavanja ovih naredbi je ilustrovan na slici 3.6.



Slika 3.6: Početna slika niza `igrači` i promenljive `brojIgrača`.

Nakon što se nekoliko igrača priključi igri, njihov aktuelni broj će se nalaziti u promenljivoj `brojIgrača`. Pored toga, elementi niza `igrači` na pozicijama od 0 do `brojIgrača - 1` ukazuju na objekte konkretnih igrača koji učestvuju u igri. Promenljiva `brojIgrača` ima dakle dvostruku ulogu — pored aktuelnog broja igrača, ova promenljiva sadrži i indeks prvog „slobodnog" elementa niza igrača koji je neiskorišćen.

Registrovanje novog igrača koji se priključio igri, recimo `noviIgrač`, u programu se postiže dodavanjem tog igrača na kraj niza `igrači`:

```
igrači[brojIgrača] = noviIgrač; // novi igrač se smešta u prvi
                                // slobodni element niza
brojIgrača++;  // aktuelni broj igrača je veći za 1
```

Kada neki igrač završi igru, uklanjanje tog igrača u programu mora se izvoditi tako da ne ostane „rupa" u nizu `igrači`. Zbog toga, ukoliko treba ukloniti igrača koji se nalazi na *k*-toj poziciji u nizu `igrači`, postupak njegovog uklanjanja zavisi od toga da li je redosled aktivnih igrača bitan ili

ne. Ako redosled igrača nije bitan, jedan način za uklanjanje *k*-tog igrača je premeštanje poslednjeg igrača na *k*-to mesto u nizu `igrači` i smanjivanje vrednosti promenljive `brojIgrača` za jedan:

```
igrači[k] = igrači[brojIgrača - 1];
brojIgrača--;
```

Igrač koji se nalazio na *k*-tom mestu nije više u nizu `igrači`, jer se na to mesto premešta igrač na poslednjoj poziciji `brojIgrača - 1`. S druge strane, ova pozicija nije više logički poslednja i postaje slobodna za upisivanje novih igrača, jer se promenljiva `brojIgrača` umanjuje za jedan.

U drugom slučaju kada je bitan redosled igrača, radi uklanjanja *k*-tog igrača se svi igrači od indeksa $k + 1$ moraju pomeriti ulevo za jedno mesto. Drugim rečima, $(k + 1)$-vi igrač dolazi na *k*-to mesto igrača koji se uklanja, zatim $(k + 2)$-gi igrač dolazi na $(k + 1)$-vo mesto koje je oslobođeno prethodnim pomeranjem, i tako dalje:

```
for (int i = k+1; i < brojIgrača; i++)
   igrači[i-1] = igrači[i];
brojIgrača--;
```

Veći problem u radu sa nizovima fiksne dužine je to što se mora postaviti prilično proizvoljna granica za broj elemenata niza. U prethodnom primeru kompjuterske igre, recimo, postavlja se pitanje šta uraditi ako se igri priključi više od 10 igrača kolika je dužina niza `igrači`? Očigledno rešenje je da se taj niz konstruiše sa mnogo većom dužinom. Ali i ta veća dužina ne garantuje da (na primer, zbog popularnosti igre) broj igrača neće narasti toliko da opet ni veća dužina niza nije dovoljna za registrovanje svih igrača. Naravno, dužina niza se uvek može izabrati da bude vrlo velika i tako da bude dovoljna za sve praktične slučajeve. Ovo rešenje ipak nije zadovoljavajuće, jer se može desiti da veliki deo niza bude neiskorišćen i da zato niz bespotrebno zauzima veliki memorijski prostor, možda sprečavajući izvršavanje drugih programa na računaru.

Najbolje rešenje ovog problema je to da niz početno ima skromnu dužinu, a da se zatim može produžiti (ili smanjiti) po potrebi u programu. Ali kako ovo nije moguće, moramo se zadovoljiti približnim rešenjem koje je skoro isto tako dobro. Naime, podsetimo se da promenljiva tipa niza ne sadrži elemente niza, nego samo ukazuje na objekat niza koji zapravo sadrži

njegove elemente. Ovaj niz se ne može produžiti, ali se može konstruisati novi objekat dužeg niza i promeniti vrednost promenljive koja ukazuje na objekat starog niza tako da ukazuje na objekat novog niza. Pored toga, naravno, elementi starog niza se moraju prekopirati u novi niz. Konačan rezultat ovog postupka biće dakle to da ista promenljiva tipa niza ukazuje na novi objekat dužeg niza koji sadrži sve elemente starog niza i ima dodatne „slobodne" elemente. Uz to, objekat starog niza biće automatski uklonjen iz memorije procedurom sakupljanja otpadaka, jer nijedna promenljiva više ne ukazuje na objekat starog niza.

Ako se ovaj pristup primeni u programu za kompjutersku igru, onda se u postupak za dodavanja novog igrača moraju ugraditi prethodne napomene u slučaju kada je popunjen ceo niz `igrači` početne dužine 10:

```
// Konstruisanje dužeg niza ako je niz igrača popunjen
if (brojIgrača == igrači.length) {
   int novaDužina = 2 * igrači.length;
   Igrač[] noviIgrači = Arrays.copyOf(igrači, novaDužina);
   igrači = noviIgrači;
}

// Dodavanje novog igrača u (novi) niz igrača
igrači[brojIgrača] = noviIgrač;
brojIgrača++;
```

Primetimo ovde da promenljiva `igrači` ukazuje bilo na stari delimično popunjen niz ili na novi niz dvostruko duži od starog. Zbog toga, nakon izvršavanja `if` naredbe, element `igrači[brojIgrača]` je sigurno slobodan i može mu se dodeliti novi igrač bez bojazni da će se premašiti granica niza `igrači`.

* * *

Nizovi čija se dužina prilagođava broju podataka koje treba da sadrže nazivaju se *dinamički nizovi*. U radu sa dinamičkimm nizovima su dozvoljene iste dve osnovne operacije kao i sa običnim nizovima: upisivanje vrednosti u element na datoj poziciji i čitanje vrednosti koja se nalazi u elementu na datoj poziciji. Ali pri tome ne postoji gornja granica za broj elemenata dinamičkog niza, osim naravno one koju nameće veličina memorije raču-

nara.

Umesto ručnog manipulisanja običnim nizom da bi se proizveo efekat dinamičkog niza, kako je to pokazano u primeru za program kompjuterske igre, u Javi se može koristiti standardna klasa `ArrayList` iz paketa `java.util`. Objekti klase `ArrayList` su dinamički nizovi koji u svakom trenutku imaju određenu dužinu, ali se ona automatski povećava kada je to potrebno. U klasi `ArrayList` su definisani mnogi objektni metodi, ali oni najznačajniji su:

- `size()` vraća aktuelnu dužinu niza tipa `ArrayList`. Dozvoljeni indeksi niza su samo oni u granicama od 0 do `size() - 1`. Konstruisanjem dinamičkog niza izrazom `new ArrayList()` konstruiše se dinamički niz dužine nula.
- `add(elem)` dodaje element `elem` na kraj niza i povećava dužinu niza za jedan. Rezultat poziva ovog metoda je sličan onome koji se dobija za običan niz `a` kada se uveća vrednosti `a.length` za jedan i izvrši naredba dodele `a[a.length] = elem`.
- `get(n)` vraća vrednost elementa niza na poziciji *n*. Argument *n* mora biti ceo broj u intervalu od 0 do `size() - 1`, inače se program prekida usled greške. Rezultat poziva ovog metoda je sličan onome koji se dobija za običan niz `a` kada se napiše `a[n]`, osim što se `get(n)` ne može nalaziti na levoj strani znaka jednakosti u naredbi dodele.
- `set(n, elem)` dodeljuje vrednost `elem` onom elementu u nizu koji se nalazi na poziciji *n*, zamenjujući njegovu prethodnu vrednost. Argument *n* mora biti ceo broj u intervalu od 0 do `size() - 1`, inače se program prekida usled greške. Rezultat poziva ovog metoda je sličan onome koji se dobija za običan niz `a` kada se napiše `a[n] = elem`.
- `remove(elem)` uklanja dati element `elem` iz niza, ukoliko se vrednost datog elementa nalazi u nizu. Svi elementi iza uklonjenog elementa se pomeraju jedno mesto ulevo i dužina niza se smanjuje za jedan. Ako se u nizu nalazi više vrednosti datog elementa, uklanja se samo prvi primerak koji se nađe idući sleva na desno.
- `remove(n)` uklanja *n*-ti element iz niza. Argument *n* mora biti ceo broj u intervalu od 0 do `size() - 1`. Svi elementi iza uklonjenog ele-

menta se pomeraju jedno mesto ulevo i dužina niza se smanjuje za jedan.

- `indexOf(elem)` pretražuje niz sleva na desno radi nalaženja vrednosti elementa `elem` u nizu. Ako se takva vrednost pronađe u nizu, indeks prvog nađenog elementa se vraća kao rezultat. U suprotnom slučaju, kao rezultat se vraća vrednost –1.

Koristeći klasu `ArrayList` u primeru programa za kompjutersku igru, niz `igrači` može se deklarisati da bude dinamički niz koji je početno prazan:

```
ArrayList igrači = new ArrayList();
```

U tom slučaju se više ne mora voditi računa o dužini niza, već se za dodavanje novog igrača može jednostavno pisati:

```
igrači.add(noviIgrač);
```

Slično, za uklanjanje *k*-tog igrača koristi se odgovarajući metod:

```
igrači.remove(k);
```

Ili, ako promenljiva `isključenIgrač` tipa `Igrač` ukazuje na objekat igrača kojeg treba ukloniti, onda se može pisati:

```
igrači.remove(isključenIgrač);
```

U ovim primerima je primena objektnih metoda klase `ArrayList` bila prirodna i očigledna. Međutim, za metod `get(n)` kojim se dobija vrednost elementa dinamičkog niza na poziciji *n*, moraju se razumeti još neki detalji o objektima u Javi. Prvo, izrazom `new ArrayList()` konstruiše se dinamički niz tipa `Object[]`, odnosno elementi konstruisanog niza su tipa `Object`. Klasa `Object` sadrži metode koji opisuju osnovne mogućnosti svih objekata i o njoj će biti više reči u odeljku 4.4. Svaka klasa u Javi automatski nasleđuje klasu `Object` kojom se objekti predstavljaju u najširem smislu. Drugo, prema načelu OOP koje se naziva *princip podtipa*, o čemu se detaljnije govori u odeljku 4.2, referenca na objekat bilo kog tipa može biti dodeljena promenljivoj tipa `Object`. Zbog ovoga, kratko rečeno, u prethodnim primerima se ne mora voditi računa o tipu elemenata dinamičkog niza.

S druge strane, tip elemenata dinamičkog niza je bitan za korišćenje metoda `get()`. Naime, deklarisan tip rezultata metoda `get()` je `Object`, ali

rezultat primene tog metoda na niz `igrači` jeste zapravo objekat tipa `Igrač`. Zbog toga, da bi se išta korisno moglo uraditi sa rezultatom poziva metoda `get()`, mora se izvršiti eksplicitna konverzija tipa njegovog rezultata iz `Object` u `Igrač`:

```
Igrač pacer = (Igrač)igrači.get(n);
```

Na primer, ako klasa `Igrač` sadrži objektni metod `odigrajPotez()` koji se poziva kada igrač treba da odigra svoj potez u igri, onda deo programa u kojem svaki aktivni igrač odigrava svoj potez može biti:

```
for (int i = 0; i < igrači.size(); i++) {
   Igrač sledećiIgrač = (Igrač)igrači.get(i);
   sledećiIgrač.odigrajPotez();
}
```

Dve naredbe u telu prethodne `for` petlje se mogu spojiti u samo jednu, doduše komplikovaniju naredbu:

```
((Igrač)igrači.get(i)).odigrajPotez();
```

Ako se ova, naizgled zamršena, naredba raščlani na sastavne delove, uočava se da se najpre dobija *i*-ti element niza `igrači`, zatim se vrši konverzija u njegov pravi tip `Igrač` i, na kraju, poziva se metod `odigrajPotez()` za rezultujućeg igrača. Obratite pažnji i na to da se izraz `(Igrač)igrači.get(i)` mora navesti u zagradi zbog unapred definisanih pravila prioriteta operatora u Javi.

U stvari, problem obavezne eksplicitne konverzije tipa prilikom čitanja vrednosti nekog elementa dinamičkog niza metodom `get()` više ne postoji od verzije Java 5. Taj problem je prevaziđen uvođenjem *parametrizovanih tipova* o kojima se opširno govori u poglavlju 11. U nastavku se samo kratko uvode najosnovniji pojmovi na primeru klase `ArrayList`.

Parametrizovani tip `ArrayList<T>` se može koristiti umesto običnog tipa `ArrayList`. Parametar `T` ovde označava bilo koji klasni tip (`T` ne sme biti primitivni tip). Dok klasa `ArrayList` predstavlja dinamičke nizove čiji su elementi tipa `Object`, klasa `ArrayList<T>` predstavlja dinamičke nizove čiji su elementi tipa `T`. Na primer, naredbom

```
ArrayList<Igrač> igrači = new ArrayList<Igrač>();
```

deklariše se promenljiva niza `igrači` tipa `ArrayList<Igrač>`, konstruiše se

prazan dinamički niz čiji su elementi tipa `Igrač` i referenca na njega se dodeljuje promenljivoj `igrači`.

Obratite pažnju na to da je `ArrayList<Igrač>` ime tipa kao i svako drugo i da se koristi na uobičajen način — sufiks `<Igrač>` je običan deo imena tipa. Dodavanje elementa dinamičkom nizu parametrizovanog tipa, ili uklanjanje elementa iz njega, izvodi se na jednostavan način kao i ranije. Na primer,

```
igrači.add(noviIgrač);
```

ili

```
igrači.remove(isključenIgrač);
```

Upotreba parametrizovanih tipova dodatno olakšava pisanje programa, jer Java prevodilac u fazi prevođenja programa može lako proveriti da li su promenljive `noviIgrač` i `isključenIgrač` zaista tipa `Igrač` i otkriti grešku ukoliko to nije slučaj. Ali ne samo to, pošto elementi niza `igrači` parametrizovanog tipa `ArrayList<Igrač>` moraju biti tipa `Igrač`, eksplicitna konverzija tipa nije više neophodna prilikom čitanja vrednosti nekog elementa tog niza:

```
Igrač pacer = igrači.get(n);
```

Parametrizovani tipovi se mogu koristiti na potpuno isti način kao i obični tipovi za deklarisanje promenljivih i za tipove parametara ili rezultata nekog metoda. Jedino ograničenje parametrizovanih tipova je to što parametar jednog takvog tipa ne može biti primitivni tip. Na primer, tip `ArrayList<int>` nije dozvoljen. Međutim, to i nije veliki nedostatak zbog mogućnosti primene klasa omotača za primitivne tipove. Sve klase koji su omotači primitivnih tipova mogu se koristiti za parametrizovane tipove. Objekat, recimo, tipa `ArrayList<Double>` je dinamički niz čiji elementi sadrže objekte tipa `Double`. Kako objekat klasnog tipa `Double` sadrži vrednost primitivnog tipa `double`, to je skoro isto kao da `ArrayList<Double>` predstavlja dinamički niz čiji su elementi realni brojevi tipa `double`. Na primer, iza naredbi

```
double x;
ArrayList<Double> nizBrojeva = new ArrayList<Double>();
```

u programu se metodom `add()` može dodati vrednost promenljive `x` na kraj

niza `nizBrojeva`:

```
nizBrojeva.add(new Double(x));
```

Ili, zbog autopakovanja i raspakovanja, može se čak kraće pisati

```
nizBrojeva.add(x);
```

jer se sve neophodne konverzije obavljaju automatski.

Primer: telefonski imenik

Jedan konkretan primer parametrizovanog dinamičkog niza je struktura podataka koju čini telefonski imenik u mobilnom telefonu ili sličnom uredaju. U najprostijem slučaju, telefonski imenik je niz kontakata koji se sastoje od dva dela: imena osobe i telefonskog broja te osobe. Pojedini kontakti telefonskog imenika mogu se dakle predstaviti kao objekti sledeće klase:

```
public class Kontakt {
   String ime;      // ime osobe
   String telBroj;  // telefonski broj osobe
}
```

Ceo telefonski imenik u programu se može prosto realizovati kao niz elemenata, pri čemu je svaki element niza jedan objekat tipa `Kontakt`. Kako se broj stavki u telefonskom imeniku ne može unapred dobro predvideti, prirodno se nameće reprezentacija telefonskog imenika u obliku dinamičkog niza kontakata. Kratko rečeno, telefonski imenik kao struktura podataka u programu je dinamički niz tipa `ArrayList<Kontakt>`. Međutim, pored ovakve šeme za čuvanje podataka u telefonskom imeniku, neophodno je obezbediti i metode kojima se realizuju osnovne operacije nad podacima u telefonskom imeniku.

Imajući ova zapažanja u vidu, klasa `TelImenik` u listingu 3.5 predstavlja telefonski imenik kao strukturu podataka. Ta klasa sadrži polje `imenik` koje ukazuje na dinamički niz sa podacima u imeniku, kao i sledeće metode kojima se realizuju osnovne operacije nad podacima u imeniku:

- Metod `nađiBroj()` za dato ime osobe pronalazi njegov telefonski broj u imeniku.

- Metod `dodajKontakt()` dodaje dati par ime/broj u imenik. U ovom metodu se najpre proverava da li se dato ime nalazi u imeniku. Ako je to slučaj, stari broj se zamenjuje datim brojem. U suprotnom slučaju, nov kontakt se dodaje u imenik.
- Metod `ukloniKontakt()` uklanja kontakt sa datim imenom osobe iz imenika. Ako se takav kontakt ne pronađe u imeniku, ništa se posebno ne preduzima.

Obratite pažnju na to da je u klasi `TelImenik` definisan i pomoćni metod `nađiKontakt()` u kojem se primenjuje linearna pretraga za nalaženje pozicije kontakta sa datim imenom u imeniku. Metod `nađiKontakt()` je privatan metod, jer je potreban samo za interno korišćenje od strane ostalih javnih metoda klase `TelImenik`.

Klasa `TelImenik` predstavlja objekte telefonskih imenika kojima se bez ograničenja na veličinu mogu dodavati imena i brojevi osoba. Ova klasa dodatno obezbeđuje mogućnost uklanjanja kontakta iz imenika i nalaženje telefonskog broja osobe na osnovu datog imena osobe. Jedan primer primene klase `TelImenik` je prikazan u metodu `main()` u kojem je testiran tipični način rada sa telefonskim imenikom.

Listing 3.5: Telefonski imenik

```
import java.util.*;

public class TelImenik {

   private ArrayList<Kontakt> imenik;  // niz kontakata

   // Konstruktor klase za konstruisanje praznog imenika
   public TelImenik() {
      imenik = new ArrayList<Kontakt>();
   }

   private int nađiKontakt(String imeOsobe) {

      for (int i = 0; i < imenik.size(); i++) {
         Kontakt k = imenik.get(i);
```

```
      // Da li i-ta osoba ima dato ime?
      if (k.ime.equals(imeOsobe))
         return i; // i-ta osoba ima dato ime
   }
   return -1;      // nema osobe sa datim imenom
}

public String nađiBroj(String imeOsobe) {

   int i = nađiKontakt(imeOsobe);
   if (i >= 0)  // osoba je u imeniku?
      // Ako jeste, vratiti njen tel. broj
      return imenik.get(i).telBroj;
   else
      // Ako nije, vratiti referencu null
      return null;
}

public void dodajKontakt(String imeOsobe, String brojOsobe) {

   if (imeOsobe == null || brojOsobe == null) {
      System.out.println("Greška: prazno ime ili broj kontakta!");
      return;
   }
   int i = nađiKontakt(imeOsobe);
   if (i >= 0)  // osoba je u imeniku?
      // Ako jeste, zameniti stari broj novim brojem
      imenik.get(i).telBroj = brojOsobe;
   else {
      // Ako nije, dodati nov par ime/broj u imenik
      Kontakt k = new Kontakt();
      k.ime = imeOsobe;
      k.telBroj = brojOsobe;
      imenik.add(k);
   }
}
```

```
   public void ukloniKontakt(String imeOsobe) {

      int i = nađiKontakt(imeOsobe);
      if (i >= 0)  // osoba je u imeniku?
         // Ako jeste, ukloniti taj kontakt iz imenika
         imenik.remove(i);
   }

   public static void main(String[] args) {

      TelImenik mojImenik = new TelImenik();
      mojImenik.dodajKontakt("Pera", null);
      mojImenik.dodajKontakt("Pera", "111-1111");
      mojImenik.dodajKontakt("Žika", "222-2222");
      mojImenik.dodajKontakt("Laza", "333-3333");
      mojImenik.dodajKontakt("Mira", "444-4444");
      System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
      mojImenik.dodajKontakt("Laza", "999-9999");
      System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
      System.out.println("Pera: " + mojImenik.nađiBroj("Pera"));
      mojImenik.ukloniKontakt("Žika");
      System.out.println("Žika: " + mojImenik.nađiBroj("Žika"));
      System.out.println("Mira: " + mojImenik.nađiBroj("Mira"));
   }
}
```

GLAVA 4

Nasleđivanje klasa

Mogućnost nasleđivanja klasa je jedna od odlika objektno orijentisanog programiranja koja ga izdvaja od proceduralnog programiranja. Primenom načela nasleđivanja klasa omogućava se pisanje novih klasa koje proširuju ili modifikuju postojeće klase. U ovom poglavlju se najpre pokazuju prednosti nasleđivanja klasa uopšte, a zatim se govori o pojedinostima tog koncepta u Javi.

4.1 Osnovni pojmovi

Neka klasa predstavlja skup objekata koji imaju zajedničku strukturu i mogućnosti. Klasa određuje strukturu objekata na osnovu objektnih polja, dok se mogućnosti objekata nekom klasom određuju preko objektnih metoda u klasi. Ova ideja vodilja objektno orijentisanog programiranja (OOP) nije doduše velika novost, jer se nešto slično može postići i primenom drugih, tradicionalnijih principa programiranja. Centralna ideja objektno orijentisanog programiranja, koja ga izdvaja od ostalih paradigmi programiranja, jeste to da se klasama mogu izraziti hijerarhijski odnosi među objektima koji imaju neke, ali ne sve, zajedničke osobine.

U objektno orijentisanom programiranju, nova klasa se može formirati na osnovu postojeće klase. To znači da nova klasa proširuje postojeću klasu i nasleđuje sva njena polja i metode. Preneseno na objekte nove klase, ovo znači da oni nasleđuje sve atribute i mogućnosti postojećih objekata. Ovaj koncept u OOP se naziva *nasleđivanje klasa* ili kraće samo *nasleđiva-*

nje. Mogućnost proširivanja postojeće klase radi formiranja nove klase ima mnoge prednosti od kojih su najvažnije polimorfizam i apstrakcija, kao i višekratna upotrebljivost i olakšano menjanje programskog koda.

Treba imati u vidu da terminologija u vezi sa nasleđivanjem klasa nije standardizovana. Uglavnom iz istorijskih razloga, ali i ličnih afiniteta autora, u upotrebi su različiti termini koji su sinonimi za postojeću klasu i novu klasu naslednicu: osnovna i proširena klasa, bazna i izvedena klasa, natklasa i potklasa, klasa-roditelj i klasa-dete, pa i nadređena i podređena klasa.

U svakodnevnom radu, naročito za programere koji su tek počeli da se upoznaju sa objektno orijentisanim pristupom, nasleđivanje se koristi uglavnom za menjanje već postojeće klase koju treba prilagoditi sa nekoliko izmena ili dopuna. To je mnogo češća situacija nego pravljenje kolekcije klasa i proširenih klasa od početka.

Definicija nove klase koja proširuje postojeću klasu ne razlikuje se mnogo od uobičajene definicije obične klase:

```
modifikatori class nova-klasa extends postojeća-klasa {
   .
   . // Izmene ili dopune postojeće klase
   .
}
```

U ovom opštem obliku, deo *nova-klasa* je ime nove klase koja se definiše, a deo *postojeća-klasa* je ime stare klase koja se proširuje. Telo nove klase između vitičastih zagrada ima istu strukturu kao telo neke uobičajene klase. Prema tome, jedina novost kod nasleđivanja klase je pisanje službene reči `extends` i imena postojeće klase iza imena nove klase. Na primer, kostur definicije klase `B` koja nasleđuje klasu `A` ima ovaj izgled:

```
class B extends A {
   .
   . // Novi članovi klase B ili
   . // izmene postojećih članova klase A
   .
}
```

Suštinu i prednosti nasleđivanja klasa je najbolje objasniti na primeru. Posmatrajmo zato program za obračun plata radnika neke firme i razmotrimo klasu koja može predstavljati te radnike u kontekstu obračuna njihovih

plata. Ako zanemarimo enkapsulaciju podataka da ne bismo primer komplikovali sa geter i seter metodima, prvi pokušaj definisanja klase `Radnik` bi mogao biti:

```
public class Radnik {

   String ime;    // ime i prezime radnika
   long jmbg;     // jedinstven broj radnika
   long račun;    // bankovni račun radnika
   double plata;  // plata radnika

   public void uplatiPlatu() {
      System.out.print("Plata za radnika " + ime);
      System.out.print(", broj " + jmbg + ", ");
      System.out.println("uplaćena na račun " + račun);
   }

   public double izračunajPlatu() {
      .
      . // Obračunavanje mesečne plate radnika
      .
   }
}
```

Na prvi pogled izgleda da klasa `Radnik` dobro opisuje radnike za obračunavanje njihovih mesečnih plata. Svaki radnik ima svoje ime, jedinstven broj i račun u banci. Jedina dilema postoji oko metoda kojim se obračunava plata svakog radnika. Problem je u tome što u firmi može raditi više vrsta radnika u pogledu načina obračunavanja njihove mesečne plate.

Pretpostavimo da u firmi neki radnici imaju fiksnu mesečnu zaradu, dok su drugi plaćeni po radnim satima po određenoj ceni sata. Rešenje koje se odmah nameće, naročito ako se dolazi iz sveta proceduralnog programiranja, jeste da se klasi `Radnik` doda indikator koji ukazuje na to da li se radi o jednoj ili drugoj vrsti radnika. Ako se doda logičko polje `plaćenPoSatu`, na primer, onda se njegova vrednost može koristiti u metodu za obračun mesečne plate da bi se ispravno izračunala plata konkretnog radnika:

```
public double izračunajPlatu() {
   if (plaćenPoSatu)
      .
      . // Obračunavanje plate radnika plaćenog po satu
      .
   }
   else {
      .
      . // Obračunavanje plate radnika sa fiksnom zaradom
      .
   }
}
```

Ovo rešenje međutim nije dobro sa aspekta objektno orijentisanog programiranja, jer se faktički koristi jedna klasa za predstavljanje dva tipa objekata. Pored ovog koncepcijskog nedostatka, veći problem nastaje kada treba nešto menjati u programu. Šta ako firma počne da zapošljava i radnike koji se plaćaju na treći način — na primer, po danu bez obzira na broj radnih sati? Onda logički indikator nije više dovoljan, nego ga treba zameniti celobrojnim poljem, recimo `tipRadnika`, čija vrednost ukazuje na to o kom tipu radnika se radi. Na primer, vrednost 0 određuje radnika sa fiksnom mesečnom zaradom, vrednost 1 određuje radnika plaćenog po satu i vrednost 2 određuje radnika plaćenog po danu. Ovakvo rešenje zahteva, pored zamene indikatora, ozbiljne modifikacije prethodnog metoda za obračun mesečne plate:

```
public double izračunajPlatu() {
   switch (tipRadnika) {
      0 :
        // Obračunavanje plate radnika sa fiksnom zaradom
        . . .
        break;
      1 :
        // Obračunavanje plate radnika plaćenog po satu
        . . .
        break;
      2 :
```

```
            // Obračunavanje plate radnika plaćenog po danu
            . . .
            break;
      }
   }
```

Naravno, svaki put kada se u firmi zaposle novi radnici sa drugačijim načinom plaćanja, mora se menjati ovaj metod za obračun plate i dodati novi slučaj.

Bolje rešenje, ali još uvek ne i ono pravo, jeste da se klasa `Radnik` podeli u dve (ili više) klasa koje odgovaraju vrstama radnika prema načinu obračuna njihove plate. Na primer, za radnike koji su mesečno plaćeni fiksno ili po radnim satima, mogu se definisati dve klase na sledeći način:

```
public class RadnikPlaćenFiksno {

   String ime;    // ime i prezime radnika
   long jmbg;     // jedinstven broj radnika
   long račun;    // bankovni račun radnika
   double plata;  // mesečna plata radnika

   public void uplatiPlatu() {
      System.out.print("Plata za radnika " + ime);
      System.out.print(", broj " + jmbg + ", ");
      System.out.println("uplaćena na račun " + račun);
   }

   public double izračunajPlatu() {
      return plata;
   }
}

public class RadnikPlaćenPoSatu {

   String ime;       // ime i prezime radnika
   long jmbg;        // jedinstven broj radnika
   long račun;       // bankovni račun radnika
   double brojSati;  // broj radnih sati radnika
```

```
    double cenaSata;  // iznos za jedan radni sat

    public void uplatiPlatu() {
       System.out.print("Plata za radnika " + ime);
       System.out.print(", broj " + jmbg + ", ");
       System.out.println("uplaćena na račun " + račun);
    }

    public double izračunajPlatu() {
       return brojSati * cenaSata;
    }
}
```

Ovo rešenje je bolje od prvog, jer mada za novu vrstu radnika treba dodati novu klasu, bar se ne moraju menjati postojeće klase za stare vrste radnika. Ali, nedostatak ovog rešenja je što se moraju ponavljati zajednička polja i metodi u svim klasama. Naime, bez obzira da li su radnici plaćeni fiksno ili po satu (ili na neki drugi način), oni pripadaju kategoriji radnika i zato imaju mnoga zajednička svojstva. To je tipični slučaj kada se nasleđivanje klasa može iskoristiti kako za eliminisanje ponavljanja programskog koda, tako i za pisanje programa koji se lako mogu menjati.

Najbolje rešenje za obračun plata radnika je dakle da se izdvoje zajednička svojstva radnika u baznu klasu, a njihova posebna svojstva ostave za nasleđene klase. Na primer:

```
public class Radnik {

    String ime;    // ime i prezime radnika
    long jmbg;     // jedinstven broj radnika
    long račun;    // bankovni račun radnika

    public void uplatiPlatu() {
       System.out.print("Plata za radnika " + ime);
       System.out.print(", broj " + jmbg + ", ");
       System.out.println("uplaćena na račun " + račun);
    }
}
```

```
public class RadnikPlaćenFiksno extends Radnik {

   double plata;  // mesečna plata radnika

   public double izračunajPlatu() {
      return plata;
   }
}

public class RadnikPlaćenPoSatu extends Radnik {

   double brojSati;  // broj radnih sati radnika
   double cenaSata;  // iznos za jedan radni sat

   public double izračunajPlatu() {
      return brojSati * cenaSata;
   }
}
```

Obratite pažnju na to da radnici ne dele isti metod `izračunajPlatu()`. Svaka klasa koja nasleđuje klasu `Radnik` ima svoj poseban metod za obračun plate. To međutim nije ponavljanje programskog koda, jer svaku klasu naslednicu upravo karakteriše logički različit način izračunavanja plate odgovarajuće vrste radnika.

U opštem slučaju, relacija *jeste* predstavlja relativno jednostavan i pouzdan test da li treba primeniti nasleđivanje klasa u nekoj situaciji. Naime, kod svakog nasleđivanja mora biti slučaj da objekat klase naslednice *jeste* i objekat nasleđene klase. Ako se može reći da postoji taj odnos, onda je nasleđivanje klasa primereno za rešenje odgovarajućeg problema. U prethodnom primeru recimo, radnik plaćen fiksno na mesečnom nivou *jeste* radnik. Slično, radnik plaćen po radnim satima *jeste* radnik. Zato je primereno pisati klase `RadnikPlaćenFiksno` i `RadnikPlaćenPoSatu` tako da nasleđuju klasu `Radnik`.

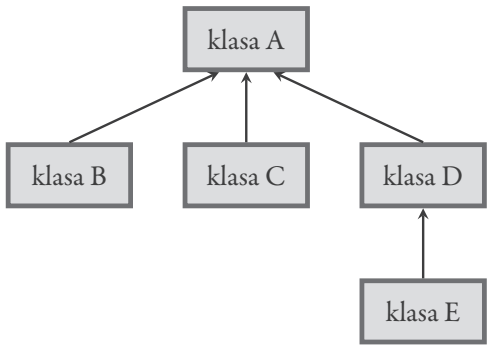
S druge strane, pretpostavimo da za svakog radnika želimo da čuvamo podatak o datumu zaposlenja. Kako u Javi postoji standardna klasa `Date` za predstavljanje datuma, da li je u ovom slučaju dobra ideja da klasu `Radnik`

napišemo kao naslednicu klase `Date`? Odgovor je negativan, jer naravno ne važi odnos da radnik *jeste* datum. Odnos koji postoji u ovom slučaju između radnika i datuma je da radnik *ima* datum zaposlenja. Ako objekat *ima* neki atribut, taj atribut treba realizovati kao polje u klasi tog objekta. Zato za predstavljanje datuma zaposlenja svakog radnika, klasa `Radnik` treba da ima polje tipa `Date`, a ne da nasleđuje klasu `Date`.

4.2 Hijerarhija klasa

Klasa naslednica može dopuniti ili modifikovati strukturu i mogućnosti klase koju nasleđuje. Treba istaći jednu važnu osobinu nasleđivanja klasa u Javi koja sledi iz opšteg oblika definicije klase naslednice na strani 136: klasa može direktno nasleđivati samo jednu klasu. Ali, s druge strane, nekoliko klasa može direktno nasleđivati istu klasu. Sve klase naslednice dele neke osobine koje nasleđuju od zajedničke klase-roditelja. Drugim rečima, nasleđivanjem se uspostavlja hijerarhijska relacija između srodnih klasa.

Relacija nasleđivanja srodnih klasa se slikovito prikazuje *klasnim dijagramom* u kome se klasa-dete nalazi ispod klase-roditelja i s njom je povezana strelicom. Primer jednog klasnog dijagrama je prikazan na slici 4.1.

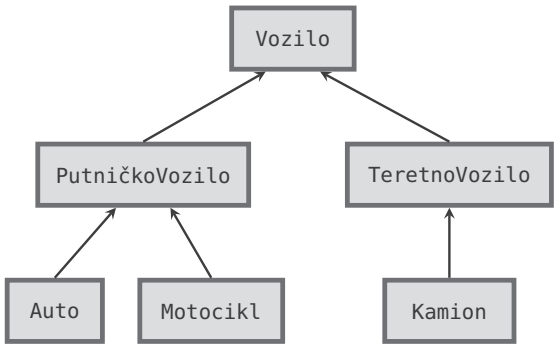


Slika 4.1: Hijerarhija klasa.

Na slici 4.1 su klase `B`, `C` i `D` direktne naslednice zajedničke klase `A`, a klasa `E` je direktna naslednica klase `D`. Dubina nasleđivanja klasa u Javi nije

ograničena i može se prostirati na nekoliko „generacija" klasa. Ovo je na slici 4.1 ilustrovano klasom `E` koja nasleđuje klasu `D`, a ova sa svoje strane nasleđuje klasu `A`. U ovom slučaju se smatra da je klasa `E` indirektna naslednica klase `A`. Cela kolekcija klasa povezanih relacijom nasleđivanja obrazuje na ovaj način *hijerarhiju klasa*.

Razmotrimo jedan konkretniji primer. Pretpostavimo da treba napisati program za evidenciju motornih vozila i da zato u programu treba definisati klasu `Vozilo` za predstavljanje svih vrsta motornih vozila. Pošto su putnička i teretna vozila posebne vrste motornih vozila, ona se mogu predstaviti klasama koje su naslednice klase `Vozilo`. Putnička i teretna vozila se dalje mogu podeliti na auta, motocikle, kamione i tako dalje, čime se obrazuje hijerarhija klasa prikazana na slici 4.2.



Slika 4.2: Hijerarhija klasa motornih vozila.

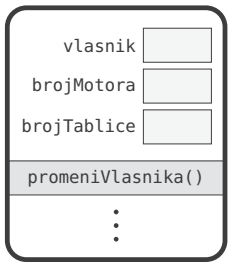
Osnovna klasa `Vozilo` treba dakle da sadrži polja i metode koji su zajednički za sva vozila. Zato ta klasa obuhvata, na primer, polja za vlasnika i brojeve motora i tablice, kao i metod za prenos vlasništva (vlasnici su predstavljeni posebnom klasom `Osoba`):

```
public class Vozilo {

   Osoba vlasnik;
   int brojMotora;
   String brojTablice;
```

```
    public void promeniVlasnika(Osoba noviVlasnik) {
       . . .
    }
    .
    . // Ostala polja i metodi
    .
}
```

Jedan objekat klase `Vozilo` na osnovu ove definicije je prikazan na slici 4.3.



Slika 4.3: Objekat klase `Vozilo`.

Ostale klase u hijerarhiji mogu zatim poslužiti za dodavanje polja i metoda koji su specifični za odgovarajuću vrstu vozila. Na primer:

```
public class PutničkoVozilo extends Vozilo {

   int brojVrata;
   String boja;
   .
   . // Ostala polja i metodi
   .
}

public class TeretnoVozilo extends Vozilo {

   int brojOsovina;
   .
   . // Ostala polja i metodi
   .
```

```
}

public class Auto extends PutničkoVozilo {

   int brojSedišta;

   public void upaliKlimu() {
      . . .
   }
   .
   . // Ostala polja i metodi
   .
}

public class Motocikl extends PutničkoVozilo {

   boolean sedišteSaStrane;
   .
   . // Ostala polja i metodi
   .
}

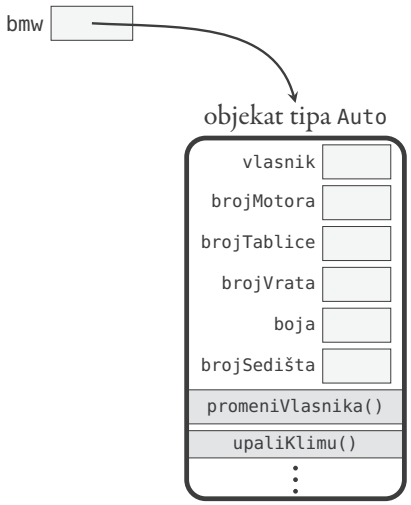
public class Kamion extends TeretnoVozilo {

   int nosivost;
   .
   . // Ostala polja i metodi
   .
}
```

Budući da, recimo, klasa `Auto` nasleđuje klasu `PutničkoVozilo`, a ova nasleđuje klasu `Vozilo`, jedan objekat klase `Auto` sadrži sva polja i sve metode ovih klasa koje nasleđuje, direktno ili indirektno. Pored toga, klasa `Auto` sadrži i polja i metode koji su neposredno definisani u telu te klase. Tako, efekat naredbe, na primer,

```
Auto bmw = new Auto();
```

koji se dobija njenim izvršavanjem u programu, ilustrovan je na slici 4.4.



Slika 4.4: Objekat tipa `Auto` na koji ukazuje `bmw`.

Pošto je `brojSedišta` objektno polje klase `Auto`, ono je naravno deo objekta na koji ukazuje promenljiva `bmw`. Zato se u programu može pisati, na primer:

```
bmw.brojSedišta = 2;
```

Ali kako `Auto` nasleđuje `PutničkoVozilo`, objekat klase `Auto` na koji ukazuje promenljiva `bmw` sadrži i sva polja i metode iz klase `PutničkoVozilo`. To znači da je uz promenljivu `bmw` ispravno koristiti i polja `brojVrata` i `boja` koja su definisana u klasi-roditelju `PutničkoVozilo`. Na primer:

```
System.out.println(bmw.brojVrata + bmw.boja);
```

Najzad, klasa `PutničkoVozilo` nasleđuje klasu `Vozilo`, što znači da objekti klase `PutničkoVozilo` (u koje spadaju objekti klase `Auto`) sadrže sva polja iz klase-roditelja `Vozilo`. Prema tome, u programu je takođe ispravno koristi polja `bmw.vlasnik`, `bmw.brojMotora` i `bmw.brojTablice`, kao i metod `bmw.promeniVlasnika()`.

Ova strana objektno orijentisanog programiranja se slaže sa načinom razmišljanja na koji smo navikli u svakodnevnom životu. Naime, auta su vrsta putničkih vozila, a putnička vozila su vrsta vozila u opštem smislu.

Drugim rečima, objekat tipa `Auto` je automatski objekat tipa `PutničkoVozilo`, kao i objekat tipa `Vozilo`. Zato neki objekat tipa `Auto` ima, pored svojih specifičnih osobina, sve osobine „roditeljskog" tipa `PutničkoVozilo`, kao i sve osobine tipa `Vozilo`. Osobine nekog tipa su predstavljene poljima i metodama odgovarajuće klase, pa je zato u programu dozvoljeno koristiti sva prethodno pomenuta polja i metode sa objektom na koji ukazuje promenljiva `bmw`.

Važna specifičnost objektno orijentisanog programiranja u ovom pogledu tiče se promenljivih klasnog tipa. Ako je definisana klasa `A`, onda je poznato da promenljiva `x` klasnog tipa `A` može imati vrednosti koje su reference na objekte klase `A`. Dodatak ove činjenice u kontekstu nasleđivanja klasa je da promenljiva `x` može sadržati i reference na objekte svih klasa koje (direktno ili indirektno) nasleđuju klasu `A`.

Ova mogućnost da promenljiva opštijeg tipa ukazuje na objekat specifičnijeg podtipa je deo opštijeg objektno orijentisanog načela koje se naziva *princip podtipa*: objekat nekog podtipa (klase-deteta) može se uvek koristiti tamo gde se očekuje objekat njegovog nadtipa (klase-roditelja). Princip podtipa je posledica činjenice što se, sužavanjem ugla gledanja time što se objekat nekog podtipa posmatra kao da je to objekat odgovarajućeg nadtipa, mogu koristiti samo umanjene osobine tog objekta, ali ne i dodatne specifičnije osobine. Naime, nasleđivanjem klase se izvedenoj klasi mogu samo dodati ili zameniti polja i metodi, a nikad se ne mogu oduzeti.

Praktična posledica primene principa podtipa za klase u prethodnoj hijerarhiji vozila jeste to što referenca na objekat tipa `Auto` može biti dodeljena promenljivoj tipa `PutničkoVozilo` ili `Vozilo`. Ispravno je pisati, na primer,

```
Vozilo mojeVozilo;
mojeVozilo = bmw;
```

ili kraće

```
Vozilo mojeVozilo = bmw;
```

ili čak:

```
Vozilo mojeVozilo = new Auto();
```

Efekat izvršavanja bilo kog od ovih programskih fragmenata je isti: pro-

menljiva `mojeVozilo` tipa `Vozilo` sadrži referencu na objekat tipa `Auto` koji je podtip tipa `Vozilo`.

U programu se čak može proveravati da li dati objekat pripada određenoj klasi. Za tu svrhu se koristi logički operator `instanceof` koji se u opštem obliku piše na sledeći način:

promenljiva **instanceof** *klasa*

Ovde promenljiva na levoj strani mora biti klasnog tipa. Ako ta promenljiva sadrži referencu na objekat koji pripada klasi na desnoj strani, rezultat operatora `instanceof` je `true`; u suprotnom slučaju, rezultat je `false`. Na primer, naredbom

```
if (mojeVozilo instanceof Auto) . . .
```

ispituje se da li objekat na koji ukazuje promenljiva `mojeVozilo` zaista pripada klasi `Auto`.

Radi dobijanja rezultata operatora `instanceof`, u svakom objektu se, između ostalog, nalazi informacija o tome kojoj klasi pripada taj objekat. Drugim rečima, prilikom konstruisanja novog objekta u hip-memoriji, pored uobičajene inicijalizacije rezervisane memorije za njega, upisuje se i podatak o definisanoj klasi kojoj novi objekat pripada.

Obrnut smer principa podtipa ne važi, pa dodela reference objekta opštijeg tipa promenljivoj specifičnijeg tipa nije dozvoljena. Naredba dodele, na primer,

```
bmw = mojeVozilo;
```

nije ispravna, jer promenljiva `mojeVozilo` tipa `Vozilo` može sadržati reference na objekte drugih tipova vozila koji nisu auta. Tako, ako promenljiva `mojeVozilo` sadrži referencu na objekat klase `Kamion`, onda bi nakon izvršavanja ove naredbe dodele i promenljiva `bmw` sadržala referencu na isti objekat klase `Kamion`. Ali onda bi upotreba polja `bmw.brojSedišta`, recimo, dovela do greške u programu zato što objekat na koji trenutno ukazuje promenljiva `bmw` ne sadrži polje `brojSedišta`, jer to polje nije definisano u klasi `Kamion`.

Princip podtipa podseća na pravila konverzije primitivnih tipova. Na primer, nije dozvoljeno dodeliti neku vrednost tipa `int` promenljivoj tipa `short`, jer nije svaka vrednost tipa `int` istovremeno i vrednost tipa `short`.

Slično, ne može se neka vrednost tipa `Vozilo` dodeliti promenljivoj tipa `Auto`, jer svako vozilo nije i auto.

Kao i kod konverzije vrednosti primitivnih tipova, rešenje je i ovde upotreba eksplicitne konverzije tipa. Ako je u programu poznato na neki način da promenljiva `mojeVozilo` zaista ukazuje na objekat tipa `Auto`, ispravno je pisati recimo

```
bmw = (Auto)mojeVozilo;
```

ili čak:

```
((Auto)mojeVozilo).brojSedišta = 4;
```

Ono što se napiše u programu ne mora naravno odgovarati ispravnom slučaju tokom izvršavanja programa, pa se za klasne tipove prilikom izvršavanja programa proverava da li je zahtevana konverzija tipa ispravna. Tako, ako promenljiva `mojeVozilo` ukazuje na objekat tipa `Kamion`, onda bi eksplicitna konverzija `(Auto)mojeVozilo` izazvala grešku i prekid programa. Da ne bi dolazilo do greške tokom izvršavanja programa kada stvarni tip objekata na koji ukazuje promenljiva klasnog tipa nije unapred poznat, eksplicitna konverzija tipa se često kombinuje sa operatorom `instanceof`. Na primer:

```
if (mojeVozilo instanceof Auto)
   ((Auto)mojeVozilo).brojSedišta = 4;
else if (mojeVozilo instanceof Motocikl)
   ((Motocikl)mojeVozilo).sedišteSaStrane = false;
else if (mojeVozilo instanceof Kamion)
   ((Kamion)mojeVozilo).nosivost = 2;
```

* * *

Obratite pažnju na to da u definiciji hijerarhije klase `Vozilo` nismo namerno koristili specifikatore pristupa za polja da ne bismo komplikovali opštu sliku nasleđivanja. Ali svi principi enkapsulacije podataka koji važe za obične klase, stoje bez izmene i za nasleđene klase. To znači generalno da polja neke klase treba da budu privatna, a pristup njima treba obezbediti preko javnih geter i seter metoda. Isto tako, samo metodi koji su deo interfejsa klase treba da budu javni, a oni koji su deo interne implementacije treba da budu privatni.

Pored specifikatora pristupa `private` i `public` za članove klase, treba imati u vidu da kod nasleđivanja dolazi u obzir i treći specifikator pristupa `protected`. Zaštićen član, odnosno član klase sa modifikatorom `protected`, može se slobodno koristiti i u svakoj klasi koja nasleđuje klasu, direktno ili indirektno, u kojoj je taj član definisan. Dodatno, zaštićen član se može koristiti u klasama koje pripadaju istom paketu kao klasa koja sadrži taj član. Podsetimo se da se član bez ikakvog specifikatora pristupa može koristiti samo u klasama iz istog paketa. To znači da je pristup `protected` striktno liberalniji od toga, jer dozvoljava korišćenje zaštićenog člana u klasama iz istog paketa i u klasama naslednicama koje nisu deo istog paketa.

Uzmimo na primer polje `boja` u klasi `PutničkoVozilo`. Zbog enkapsulacije podataka bi to polje trebalo da bude privatno da ne bi moglo da se menja izvan te klase. Naravno, dobijanje vrednosti tog polja iz druge klase bi se obezbedilo geter metodom. Ali ako je klasa `PutničkoVozilo` predviđena za proširivanje, polje `boja` bismo mogli da definišemo da bude zaštićeno kako bi dozvolili klasama naslednicama, ali ne i svim klasama, da mogu slobodno menjati to polje. Još bolje rešenje bi bio `protected` seter metod za to polje u klasi `PutničkoVozilo`. U tom metodu bi se proveravalo, na primer, da li kod dodele boje za putničko vozilo to ima smisla.

U vezi sa specifikatorima pristupa za članove klase, kod nasleđivanja klasa treba naglasiti još jednu činjenicu: iako objekti klase naslednice sadrže privatne članove iz nasleđene klase, u klasi naslednici ti članovi *nisu* direktno dostupni. Prema tome, privatni članovi neke klase ne mogu se direktno koristiti nigde, čak ni u klasi naslednici, osim u samoj klasi u kojoj su definisani. Naravno, ako je potrebno, pristup privatnim poljima neke klase iz drugih nepovezanih klasa treba obezbediti javnim geter i seter metodima. A ako treba omogućiti restriktivniji pristup privatnim poljima samo iz klasa naslednica, onda geter i seter metodi za ta polja treba da budu definisani kao zaštićeni.

4.3 Službena reč `super`

Klasa `B` koja nasleđuje postojeću klasu `A` može imati dodatne ili zamenjene članove (polja i metode) u odnosu na nasleđenu klasu `A`. Za korišćenje

dodatnih članova u klasi naslednici B nema nejasnoća, jer takvi članovi ne postoje u nasleđenoj klasi A, pa se mogu nedvosmisleno prepoznati na osnovu svojih prostih imena. U slučaju zamenjenih članova koji su definisani u klasi naslednici B, koji dakle već postoje u nasleđenoj klasi A, može se postaviti pitanje na šta se misli kada se u klasi naslednici B koriste njihova imena: da li na primerak definisan u klasi naslednici B ili na onaj definisan u nasleđenoj klasi A? Generalan odgovor na ovo pitanje je da se korišćenje prostog imena nekog člana klase uvek odnosi na primerak u klasi u kojoj je taj član definisan. Ali, u klasi naslednici B, isti član koji postoji i u nasleđenoj klasi A ne zamenjuje taj član iz nasleđene klase A, nego ga samo *zaklanja*.

Zaklanjanje polja u klasi naslednici B se retko koristi, ali ako je to slučaj, onda se može smatrati da objekat klase naslednice B sadrži zapravo dva polja sa istim imenom: jedno koje je definisano u samoj klasi naslednici B i drugo koje je definisano u nasleđenoj klasi A. Novo polje u klasi naslednici B ne zamenjuje staro polje sa istim imenom u nasleđenoj klasi A, nego ga samo zaklanja time što se podrazumeva da se svaka prosta upotreba tog imena u klasi naslednici B odnosi na primerak polja definisanog u toj klasi naslednici.

Slično, ako se u klasi naslednici B definiše metod koji ima isti potpis kao neki metod u nasleđenoj klasi A, metod iz nasleđene klase A je zaklonjen u klasi naslednici B na isti način. U ovom slučaju se još kaže da metod u klasi naslednici B *nadjačava* metod iz nasleđene klase A.

Ako je potrebno, zaklonjen član (polje ili metod) iz nasleđene klase A se i dalje može koristiti u klasi naslednici B uz službenu reč super:

```
super.zaklonjen-član
```

Prema tome, službena reč super se koristi za pristup članovima nasleđene klase koji su zaklonjeni članovima klase naslednice. Na primer, super.x se uvek odnosi na objektno polje sa imenom x u nasleđenoj klasi. Isto tako, super.m(...) se uvek odnosi na poziv nadjačanog metoda m() iz nasleđene klase.

Kao jedan primer primene reči super, razmotrimo klasu Dete koju nasleđuje klasa Učenik. Klasa Dete služi, recimo, za registrovanje podataka pojedine dece, dok klasa Učenik služi za registrovanje podataka pojedinih učenika osnovnih škola. Budući da je svaki učenik osnovne škole ujedno i

dete, prirodno je da klasa `Učenik` nasleđuje klasu `Dete`:

```
public class Dete {

   private String ime;
   private int uzrast;

   // Konstruktor
   public Dete(String ime, int uzrast) {
      this.ime = ime;
      this.uzrast = uzrast;
   }

   // Metod prikaži()
   public void prikaži() {
      System.out.println();
      System.out.println("Ime:    " + ime);
      System.out.println("Uzrast: " + uzrast);
   }
}

public class Učenik extends Dete {

   private String škola;
   private int razred;

   // Konstruktor
   public Učenik(String ime, int uzrast,
                 String škola, int razred) {
      super(ime,uzrast);
      this.škola = škola;
      this.razred = razred;
      System.out.println("Konstruisan učenik ... ");
      prikaži();
   }

   // Metod prikaži()
   public void prikaži() {
```

```
        super.prikaži();
        System.out.println("Škola:  " + škola);
        System.out.println("Razred: " + razred);
    }
}
```

U klasi `Dete` su definisana dva privatna polja `ime` i `uzrast`, konstruktor za inicijalizovanje ovih polja objekta deteta, kao i metod `prikaži()` koji prikazuje individualne podatke nekog deteta. U proširenoj klasi `Učenik` su dodata dva privatna polja `škola` i `razred`, konstruktor za inicijalizovanje svih polja objekta učenika, kao i drugi metod `prikaži()` koji prikazuje individualne podatke nekog učenika.

Kako metod `prikaži()` u klasi `Učenik` ima isti potpis kao metod sa istim imenom u klasi `Dete`, u klasi `Učenik` ova „učenička" verzija tog metoda nadjačava njegovu „dečju" verziju sa istim potpisom. Zato se poziv metoda `prikaži()` na kraju konstruktora klase `Učenik` odnosi na taj metod koji je definisan u klasi `Učenik`.

Metod `prikaži()` u klasi `Učenik` je predviđen za prikazivanje podataka o učeniku. Ali kako objekat učenika sadrži privatna polja `ime` i `uzrast` nasleđene klase `Dete`, ovaj metod nema pristup tim poljima i ne može direktno prikazati njihove vrednosti. Zbog toga se naredbom

```
super.prikaži();
```

u metodu `prikaži()` klase `Učenik` najpre poziva metod `prikaži()` nasleđene klase `Dete`. Ovaj nadjačan metod ima pristup poljima `ime` i `uzrast`, pa može prikazati njihove vrednosti. Na primer, izvršavanjem naredbe

```
Učenik u = new Učenik("Laza Lazić", 16, "gimnazija", 2);
```

dobijaju se sledeći podaci na ekranu:

```
Konstruisan učenik ...

Ime:    Laza Lazić
Uzrast: 16
Škola:  gimnazija
Razred: 2
```

Primetimo još da bi bilo pogrešno da se u metodu `prikaži()` klase `Uče-`

nik poziva nadjačan metod klase Dete bez reči super ispred metoda prikaži(). To bi onda značilo da se zapravo rekurzivno poziva metod prikaži() klase Učenik, pa bi se dobio beskonačan lanac poziva tog metoda.

Ovaj mali primer nasleđivanja pokazuje, u stvari, glavnu primenu službene reči super u opštem slučaju. Naime, ona se koristi u slučajevima kada novi metod sa istim potpisom u klasi naslednici ne treba da potpuno zameni funkcionalnost nasleđenog metoda, već novi metod treba da *proširi* funkcionalnost nasleđenog metoda. U novom metodu se tada može koristiti službena reč super za pozivanje nadjačanog metoda iz nasleđene klase, a dodatnim naredbama se može proširiti njegova funkcionalnost. U prethodnom primeru je metod prikaži() za prikazivanje podataka o učeniku proširivao funkcionalnost metoda prikaži() za prikazivanje podataka o detetu.

U prethodnom primeru bi se moglo prigovoriti da nismo ni morali da koristimo reč super. Naime, da smo dozvolili da pristup poljima ime i uzrast nasleđene klase Dete bude javan ili zaštićen, u metodu prikaži() proširene klase Učenik mogli bismo da direkno prikažemo njihove vrednosti. Ali obratite pažnju na to da smo korišćenjem reči super u prethodnom primeru mogli da proširimo funkcionalnost metoda prikaži() klase Učenik čak i da nismo znali kako je napisan nadjačan metod u klasi Dete. A to je tipični slučaj u praksi, jer programeri vrlo često proširuju nečije gotove klase čiji izvorni tekst ne poznaju ili ne mogu da menjaju.

* * *

Radi kompletnosti, pomenimo na kraju i mogućnost sprečavanja da neki metod u nasleđenoj klasi bude nadjačan. To se obezbeđuje navođenjem modifikatora final u zaglavlju metoda. Na primer:

```
public final void nekiMetod() { ... }
```

Podsetimo se da se modifikatorom final za promenljive (i polja) onemogućava promena njihove vrednosti nakon njihove inicijalizacije. U slučaju metoda, modifikatorom final se onemogućava nadjačavanje takvog metoda u bilo kojoj klasi koja nasleđuje onu u kojoj je metod definisan.

U stvari, modifikator final se može koristiti i za klase. Ako se final nalazi u zaglavlju definicije neke klase, to znači da se ta klasa ne može naslediti. Na primer:

```
public final class NekaKlasa { ... }
```

Drugačije gledano, final klasa je konačni završetak grane u stablu neke hijerarhije klasa. Na neki način final klasa se može smatrati generalizacijom final metoda, jer svi njeni metodi automatski postaju final metodi. Ali s druge strane to ne važi za njena polja — ona ne dobijaju nikakvo dodatno specijalno značenje u final klasi.

Razlozi zbog kojih se obično koriste final klase i metodi su bezbednost i objektno orijentisani dizajn. Na primer, ako su neke klase ili metodi toliko važni da bi njihovo menjanje moglo ugroziti ispravan rad celog programa, onda ih treba definisati sa modifikatorom final. Upravo je ovo razlog zašto su standardne klase System i String definisane da budu final klase. Drugo, ako neka klasa koncepcijski nema smisla da ima naslednicu, ona se može definisati da bude final klasa. Razloge za upotrebu modifikatora final u svakom konkretnom slučaju treba ipak dobro odmeriti, jer taj modifikator ima negativističku ulogu ograničavanja podrazumevanih mehanizama u Javi.

Konstruktori u klasama naslednicama

Konstruktor tehnički ne pripada klasi, pa se ni ne može naslediti u klasi naslednici. To znači da, ako nijedan konstruktor nije definisan u klasi naslednici, toj klasi se dodaje podrazumevani konstruktor bez obzira na konstruktore u nasleđenoj klasi. Drugim rečima, ako se želi neki poseban konstruktor u klasi naslednici, on se mora pisati od početka.

To može predstavljati problem ukoliko treba ponoviti sve inicijalizacije onog dela konstruisanog objekta klase naslednice koje obavlja konstruktor nasleđene klase. To može biti i nemoguć zadatak, ukoliko nisu poznati detalji rada konstruktora nasleđene klase (jer izvorni tekst nasleđene klase nije dostupan), ili konstruktor nasleđene klase inicijalizuje privatna polja nasleđene klase.

Rešenje ovog problema se sastoji u mogućnosti pozivanja konstruktora nasleđene klase upotrebom službene reči super. Ova mogućnost je povezana sa ulogom reči super za pozivanje nadjačanog metoda nasleđene klase u smislu da se poziva konstruktor nasleđene klase. Međutim, način pozivanja konstruktora nasleđene klase je drugačiji od uobičajenog poziva nadja-

čanog metoda, jer se ni konstruktori ne pozivaju kao ostali metodi. Poziv konstruktora nasleđene klase u opštem obliku izgleda kao da je reč `super` ime tog metoda, iako ona to nije:

```
super(lista-argumenata)
```

Primetimo da je na ovaj način, u konstruktoru klase `Učenik` prethodnog primera, pozvan konstruktor nasleđene klase `Dete` radi inicijalizacije polja `ime` i `uzrast` konstruisanog objekta klase `Učenik`:

```
super(ime,uzrast);
```

Obratite pažnju na to da poziv konstruktora nasleđene klase upotrebom reči `super` mora biti prva naredba u konstruktoru klase naslednice. Radi potpunosti napomenimo i da ako se konstruktor nasleđene klase ne poziva eksplicitno na ovaj način, onda se automatski poziva podrazumevani konstruktor nasleđene klase bez argumenata.

U stvari, potpun postupak za međusobno pozivanje konstruktora i inicijalizaciju objektnih polja prilikom konstruisanja nekog objekta operatorom `new` sastoji se od ovih koraka:

- Ako je prva naredba pozvanog konstruktora obična naredba, odnosno nije poziv drugog konstruktora pomoću službenih reči `this` ili `super`, implicitno se dodaje poziv `super()` radi pozivanja (podrazumevanog) konstruktora nasleđene klase bez argumenata. Posle završetka tog poziva se inicijalizuju objektna polja i nastavlja se sa izvršavanjem aktuelnog konstruktora.
- Ako prva naredba pozvanog konstruktora jeste poziv konstruktora nasleđene klase pomoću službene reči `super`, poziva se taj konstruktor nasleđene klase. Posle završetka tog poziva se inicijalizuju objektna polja i nastavlja se sa izvršavanjem aktuelnog konstruktora.
- Ako prva naredba pozvanog konstruktora jeste poziv preopterećenog konstruktora pomoću službene reči `this`, onda se poziva odgovarajući preopterećeni konstruktor klase. Posle završetka tog poziva se odmah nastavlja sa izvršavanjem aktuelnog konstruktora. Poziv konstruktora nasleđene klase se izvršava unutar preopterećenog konstruktora, bilo eksplicitno ili implicitno, tako da se i inicijalizacija objektnih polja nasleđene klase tamo završava.

4.4 Klasa `Object`

Glavna odlika objektno orijentisanog programiranja je mogućnost nasleđivanja klasa i definisanja nove klase na osnovu postojeće klase. Nova klasa nasleđuje sve atribute i mogućnosti postojeće klase, ali ih može modifikovati ili dodati nove.

Objektno orijentisani jezik Java je karakterističan po tome što sadrži specijalnu klasu `Object` koja se nalazi na vrhu ogromne hijerarhije klasa koja obuhvata sve druge klase u Javi. Klasa `Object` definiše osnovne mogućnosti koje moraju imati svi objekti: proveravanje jednakosti objekta sa drugim objektima, generisanje jedinstvenog kodnog broja objekta, konvertovanja objekta u string, kloniranje objekta i tako dalje.

Stablo nasleđivanja u Javi je organizovano tako da svaka klasa, osim specijalne klase `Object`, jeste (direktna ili indirektna) naslednica neke klase. Naime, neka klasa koja nije eksplicitno definisana kao naslednica druge klase, automatski postaje naslednica klase `Object`. To znači da definicija svake klase koja ne sadrži reč `extends`, na primer,

```
public class NekaKlasa { ... }
```

potpuno je ekvivalentna sa:

```
public class NekaKlasa extends Object { ... }
```

Kako je svaka klasa u Javi naslednica klase `Object`, to prema principu podtipa znači da promenljiva deklarisanog tipa `Object` može zapravo sadržati referencu na objekat bilo kog tipa. Ova činjenica je iskorišćena u Javi za realizaciju standardnih struktura podataka koje su definisane tako da sadrže elemente tipa `Object`. Ali kako je svaki objekat ujedno i instanca klase `Object`, ove strukture podataka mogu zapravo sadržati objekte proizvoljnog tipa.

U klasi `Object` je definisano nekoliko objektnih metoda koji nasleđivanjem postaju deo svake klase. Ovi metodi se zato mogu bez izmena koristiti u svakoj klasi, ali se mogu i nadjačati ukoliko njihova osnovna funkcionalnost nije odgovarajuća. U nastavku su pomenuti samo oni metodi iz klase `Object` koji su korisni za jednostavnije primene — ostali metodi se koriste za složenije, višenitne programe i njihov opis se može naći u zvaničnoj dokumentaciji jezika Java.

equals()

Metod `equals()` vraća logičku vrednost tačno ili netačno prema tome da li je neki objekat koji je naveden kao argument metoda jednak onom objektu za koji je metod pozvan. Preciznije, ako su `o1` i `o2` promenljive klasnog tipa `Object`, u klasi `Object` je metod `equals()` definisan tako da je `o1.equals(o2)` ekvivalentno sa `o1 == o2`. Ovaj metod dakle jednostavno proverava da li dve promenljive klasnog tipa `Object` ukazuju na isti objekat. Ovaj pojam jednakosti dva objekta neke druge konkretnije klase obično nije odgovarajući, pa u toj klasi treba nadjačati ovaj metod radi realizacije pojma jednakosti njenih objekata.

Na primer, dva deteta predstavljena klasom `Dete` iz prethodnog odeljka mogu se smatrati jednakim ukoliko imaju isto ime i uzrast. Zato u klasi `Dete` treba nadjačati osnovni metod `equals()` na sledeći način:

```
public boolean equals(Object o) {
   if (o == null || !(o instanceof Dete))
      return false;

   // Konverzija parametra metoda u tip Dete
   Dete drugoDete = (Dete)o;

   if (this.ime == drugoDete.ime &&
       this.uzrast == drugoDete.uzrast)
      // Drugo dete ima isto ime i uzrast kao ovo dete,
      // pa se podrazumeva da su ta deca jednaki objekti.
      return true;
   else
      return false;
}
```

Drugi primer je metod `equals()` u standardnoj klasi `String` kojim se dva stringa klase `String` smatraju jednakim ukoliko se sastoje od tačno istih nizova znakova u istom redosledu:

```
String ime = "Pera";
. . .
if (ime.equals(korisničkoIme))
   login(korisničkoIme);
```

```
. . .
```

Obratite pažnju na to da ovo nije logički ekvivalentno sa ovim testom:

```
if (ime == korisničkoIme)  // Pogrešno!
```

Ovom `if` naredbom se proverava da li dve promenljive klasnog tipa `String` ukazuju na isti string, što je dovoljno ali nije neophodno da dva stringa budu jednaka.

`hashCode()`

Metod `hashCode()` kao rezultat daje kodni broj objekta za koji se poziva. Ovaj „nasumično izabran" ceo broj (koji može biti negativan) služi za identifikaciju objekta i popularno se naziva njegov *heš kôd*. Heš kodovi se koriste za rad sa objektima u takozvanim *heš tabelama*. To su naročite strukture podataka koje omogućavaju efikasno čuvanje i nalaženje objekata.

Osnovni metod `hashCode()` u klasi `Object` proizvodi jedinstven heš kôd za svaki objekat. Drugim rečima, jednaki objekti imaju isti heš kôd i različiti objekti imaju različite heš kôdove. Pri tome se pod jednakošću objekata podrazumeva ona jednakost koju realizuje metod `equals()` klase `Object`. Zbog toga, ukoliko se u nekoj klasi nadjačava metod `equals()`, mora se nadjačati i metod `hashCode()`. Uslov koji nadjačana verzija metoda `hashCode()` mora zadovoljiti, radi ispravne realizacije heš tabela, donekle je oslabljen u odnosu na onaj koji zadovoljava podrazumevana verzija klase `Object`. Naime, (novo) jednaki objekti moraju imati isti heš kôd, ali različiti objekti ne moraju dobiti različit heš kôd.

`toString()`

Metod `toString()` kao rezultat daje string (objekat tipa `String`) koji predstavlja objekat za koji je metod pozvan. Značaj metoda `toString()` se sastoji u tome što ukoliko se neki objekat koristi u kontekstu u kojem se očekuje string, recimo kada se objekat prikazuje na ekranu, onda se taj objekat automatski konvertuje u string pozivanjem metoda `toString()`.

Verzija metoda `toString()` koja je definisana u klasi `Object`, za objekat za koji je ovaj metod pozvan, proizvodi string koji se sastoji od imena klase

kojoj taj objekat pripada, zatim znaka `@` i, na kraju, heš koda tog objekta u heksadekadnom zapisu. Na primer, ako promenljiva `klinac` ukazuje na neki objekat klase `Dete` iz prethodnog odeljka, onda bi se kao rezultat poziva

```
klinac.toString()
```

dobio string oblika `"Dete@62f72617"`. Pošto ovo nije mnogo korisno, u sopstvenim klasama treba definisati nov metod `toString()` koji će nadjačati ovu nasleđenu verziju. Na primer, u klasi `Dete` iz prethodnog odeljka može se dodati sledeći metod:

```
public String toString() {
   return ime + " (" + uzrast + ")";
}
```

Metod `toString()` se automatski poziva za konvertovanje objekta u string ukoliko se neki objekat koristi u kontekstu u kojem se očekuje string. Na primer, izvršavanjem programskog fragmenta

```
Dete klinac = new Dete("Aca", 10);
System.out.print("Moj mali je " + klinac);
System.out.println(", ljubi ga majka.");
```

na ekranu se primenom prethodnog nadjačanog metoda `toString()` dobija rezultat:

```
Moj mali je Aca (10), ljubi ga majka.
```

`clone()`

Metod `clone()` proizvodi kopiju objekta za koji je pozvan. Pravljenje kopije nekog objekta izgleda prilično jednostavan zadatak, jer na prvi pogled treba samo kopirati vrednosti svih polja u drugu instancu iste klase. Ali postavlja se pitanje šta ako polja originalnog objekta sadrže reference na objekte neke druge klase? Kopiranjem ovih referenci će polja kloniranog objekta takođe ukazivati na iste objekte druge klase. To možda nije efekat koji se želi — možda svi objekti druge klase na koje ukazuju polja kloniranog objekta treba takođe da budu nezavisne kopije.

Rešenje ovog problema nije tako jednostavno i prevazilazi okvire ove knjige. Ovde ćemo samo pomenuti da se rezultati prethodna dva pristupa

kloniranja nazivaju *plitka kopija* i *duboka kopija* originalnog objekta. Verzija metoda `clone()` u klasi `Object`, koju nasleđuju sve klase, proizvodi plitku kopiju objekta.

4.5 Polimorfizam

Tri stuba na koja se oslanja objektno orijentisano programiranje su enkapsulacija, nasleđivanje i polimorfizam. Do sada smo upoznali prva dva koncepta, a u ovom odeljku se više pažnje posvećuje polimorfizmu.

Pre svega, reč polimorfizam ima korene u grčkom jeziku i otprilike izražava „mnoštvo oblika". (*Poli* na grčkom znači mnogo i *morf* znači oblik.) U Javi se polimorfizam manifestuje na više načina. Jedna forma polimorfizma je princip podtipa po kojem promenljiva klasnog tipa može sadržati reference na objekte svog deklarisanog tipa i svakog njegovog podtipa.

Druga forma polimorfizma odnosi se na različite metode koji imaju isto ime. U jednom slučaju, različiti metodi imaju isto ime, ali različite liste parametara. To su onda *preopterećeni* metodi o kojima smo govorili na strani 45. Verzija preopterećenog metoda koja se zapravo poziva u konkretnoj naredbi poziva tog metoda, određuje se na osnovu liste argumenata poziva. To se može razrešiti u fazi prevođenja programa, pa se kaže da se za pozive preopterećenih metoda primenjuje *statičko vezivanje* (ili *rano vezivanje*).

Posmatrajmo primer klase `A` i njene naslednice klase `B` u kojima je definisano nekoliko verzija preopterećenog metoda `m()`. Sve verzije ovog metoda samo prikazuju poruku na ekranu koja identifikuje pozvani metod:

```
public class A {
   public void m() {
      System.out.println("m()");
   }
}

public class B extends A {
   public void m(int x) {
      System.out.println("m(int x)");
   }
```

```
    public void m(String y) {
        System.out.println("m(String y)");
    }
}
```

U klasi `A` je definisan preopterećen metod `m()` bez parametara. Klasa `B` nasleđuje ovaj metod i dodatno definiše još dve njegove verzije:

```
m(int x)
m(String y)
```

Primetimo da ove obe verzije metoda `m()` imaju po jedan parametar različitog tipa. U sledećem programskom fragmentu se pozivaju sve tri verzije ovog metoda za objekat klase `B`:

```
B b = new B();
b.m();
b.m(17);
b.m("niska");
```

Java prevodilac može da, na osnovu argumenata koji su navedeni u pozivima metoda `m()`, nedvosmisleno odredi koju verziju preopterećenog metoda treba izvršiti u svakom slučaju. To potvrđuje i rezultat izvršavanja prethodnog programskog fragmenta koji se dobija na ekranu:

```
m()
m(int x)
m(String y)
```

U drugom mogućem slučaju za više metoda sa istim imenom, različiti metodi imaju isti potpis (isto ime i iste liste parametara) u klasama naslednicama. To su onda *nadjačani* metodi o kojima smo govorili u odeljku 4.3. Kod nadjačanih metoda se odluka o tome koji metod treba zapravo pozvati donosi u vreme izvršavanja na osnovu stvarnog tipa objekta za koji je metod pozvan. Ovaj mehanizam vezivanja poziva metoda za odgovarajuće telo metoda koje će se izvršiti, naziva se *dinamičko vezivanje* (ili *kasno vezivanje*) pošto se primenjuje u fazi izvršavanja programa, a ne u fazi prevođenja programa.

Da bismo ovo razjasnili, razmotrimo sledeći primer hijerarhije klasa kućnih ljubimaca:

```
public class KućniLjubimac {
   public void koSamJa() {
      System.out.println("Ja sam kućni ljubimac.");
   }
}

public class Pas extends KućniLjubimac {
   public void koSamJa() {
      System.out.println("Ja sam pas.");
   }
}

public class Mačka extends KućniLjubimac {
   public void koSamJa() {
      System.out.println("Ja sam mačka.");
   }
}
```

U baznoj klasi `KućniLjubimac` je definisan metod `koSamJa()`, a on je nadjačan u izvedenim klasama `Pas` i `Mačka` kako bi se prikazala odgovarajuća poruka na ekranu zavisno od vrste kućnog ljubimca. Obratite pažnju na to da ako je `ljubimac` promenljiva klasnog tipa `KućniLjubimac`, onda se poziv metoda `koSamJa()` u naredbi

```
ljubimac.koSamJa();
```

ne može razrešiti u fazi prevođenja programa. To je posledica principa podtipa, jer promenljiva `ljubimac` tokom izvršavanja programa može ukazivati na objekat bilo kog tipa iz hijerarhije kućnih ljubimaca: `KućniLjubimac`, `Pas` ili `Mačka`. Zbog toga se odluka o tome koju verziju metoda `koSamJa()` treba pozvati mora odložiti za fazu izvršavanja programa, jer će se samo tada znati stvarni tip objekta na koji ukazuje promenljiva `ljubimac`. A na osnovu tog tipa će se onda pozvati i odgovarajuća verzija metoda `koSamJa()`. Zbog toga, posle izvršavanja programskog fragmenta

```
KućniLjubimac ljubimac1 = new KućniLjubimac();
KućniLjubimac ljubimac2 = new Pas();
KućniLjubimac ljubimac3 = new Mačka();
ljubimac1.koSamJa();
```

```
ljubimac2.koSamJa();
ljubimac3.koSamJa();
```

na ekranu se dobija ovaj rezultat:

```
Ja sam kućni ljubimac.
Ja sam pas.
Ja sam mačka.
```

Obratite pažnju u ovom primeru na to da su definisane tri promenljive istog tipa `KućniLjubimac`: `ljubimac1`, `ljubimac2` i `ljubimac3`. Međutim, samo prva promenljiva ukazuje na objekat tipa `KućniLjubimac`, a ostale dve ukazuju na objekte različitih podtipova tog tipa. Na osnovu rezultata izvršavanja programskog fragmenta može se zaključiti da se pozivaju različite verzije metoda `koSamJa()` za ove tri promenljive. Naime, poziva se ona verzija koja odgovara tipu objekata na koje stvarno ukazuju tri promenljive istog deklarisanog tipa `KućniLjubimac`.

Dinamičko vezivanje metoda obavlja se dakle po jednostavnom pravilu: nadjačan metod koji se poziva za neku klasnu promenljivu zavisi od stvarnog tipa objekta na koji ukazuje ta promenljiva, bez obzira na njen deklarisan tip. Zahvaljujući ovom aspektu polimorfizma može se na efikasan način manipulisati velikim brojem objekata u istoj hijerarhiji klasa. Tako, ako jedan niz predstavlja populaciju svih kućnih ljubimaca koje drže stanovnici nekog naselja od 1000 stanovnika, na primer,

```
KućniLjubimac[] ljubimci = new KućniLjubimac[1000];
```

i ako pojedinačni elementi niza ukazuju na objekte tipova `KućniLjubimac`, `Pas` ili `Mačka`, onda se pojedinačne vrste ljubimaca mogu prikazati u običnoj petlji:

```
for (KućniLjubimac ljubimac : ljubimci) {
   ljubimac.koSamJa();
}
```

Pored toga, ako se hijerahiji klasa kućnih ljubimaca kasnije doda nova izvedena klasa, na primer,

```
public class Kanarinac extends KućniLjubimac {
   public void koSamJa() {
      System.out.println("Ja sam kanarinac.");
```

```
        }
    }
```

i ako neki elementi niza `ljubimci` mogu zato ukazivati i na objekte nove klase `Kanarinac`, onda prethodna petlja za prikazivanje vrste svih kućnih ljubimaca ne mora uopšte da se menja. Ova opšta mogućnost, da se može pisati programski kôd koji će ispravno uraditi nešto što nije čak ni predviđeno u vreme njegovog pisanja, jeste verovatno najznačajnija osobina polimorfizma.

GLAVA 5

Posebne klase i interfejsi

Pored običnih klasa koje smo do sada upoznali, u Javi se mogu koristiti i druge vrste klasa čija prava vrednost dolazi do izražaja tek u složenijim primenama objektno orijentisanog programiranja. U ovom poglavlju se govori o takvim posebnim vrstama klasa.

U stvari, najpre se govori o jednostavnom konceptu nabrojivih tipova koji su važni i za svakodnevno programiranje. Nabrojivim tipom se predstavlja mali (ograničen) skup vrednosti. Mogućnost definisanja nabrojivih tipova je relativno skoro dodata programskom jeziku Java od verzije 5, ali mnogi programeri smatraju da su nabrojivi tipovi trebali biti deo Jave od samog početka.

Druga tema ovog poglavlja su apstraktne klase. U objektno orijentisanom programiranju je često potrebno definisati opšte mogućnosti datog apstraktnog entiteta bez navođenja konkretnih implementacionih detalja svake od njegovih mogućnosti. U takvim slučajevima se može koristiti apstraktna klasa na vrhu hijerarhije klasa za navođenje najopštijih zajedničkih mogućnosti u vidu apstraktnih metoda, dok se u klasama naslednicama apstraktne klase ovi apstraktni metodi mogu nadjačati konkretnim metodima sa svim detaljima.

Treći deo ovog poglavlja posvećen je važnom konceptu interfejsa. Interfejsi u Javi su još jedan način da se opiše *šta* objekti u programu mogu da urade, bez definisanja načina koji pokazuje *kako* treba to da urade. Neka klasa može implementirati jedan ili više interfejsa, a objekti ovih implementirajućih klasa onda poseduju sve mogućnosti koje su navedene u implementiranim interfejsima.

Na kraju, ugnježđene klase su tehnički nešto složeniji koncept — one se definišu unutar drugih klasa i njihovi metodi mogu koristiti polja obuhvatajućih klasa. Ugnježđene klase su naročito korisne za pisanje programskog koda koji služi za rukovanje događajima grafičkog korisničkog interfejsa.

Java programeri verovatno neće morati da pišu svoje apstraktne klase, interfejse i ugnježđene klase sve dok ne dođu do tačke pisanja vrlo složenih programa ili biblioteka klasa. Ali čak i za svakodnevno programiranje, a pogotovo za programiranje grafičkih aplikacija, svi programeri moraju da razumeju ove naprednije koncepte, jer se na njima zasniva upotreba mnogih standardnih tehnika u Javi. Prema tome, da bi Java programeri bili uspešni u svom poslu, moraju poznavati pomenute naprednije mogućnosti objektno orijentisanog programiranja u Javi bar na osnovnom nivou korišćenja.

5.1 Nabrojivi tipovi

Programski jezik Java sadrži osam ugrađenih primitivnih tipova i potencijalno veliki broj drugih tipova koji se u programu definišu klasama. Vrednosti primitivnih tipova, kao i dozvoljene operacije nad njima, ugrađeni su u sâm jezik. Vrednosti klasnih tipova su objekti definisanih klasa, a dozvoljene operacije nad njima su definisani metodi u tim klasama. U Javi postoji još jedan način za definisanje novih tipova koji se nazivaju *nabrojivi tipovi* (engl. *enumerated types* ili kraće *enums*).

Pretpostavimo da je u programu potrebno predstaviti četiri godišnja doba: proleće, leto, jesen i zimu. Ove vrednosti bi se naravno mogle kodirati celim brojevima 1, 2, 3 i 4, ili slovima `P`, `L`, `J` i `Z`, ili čak objektima neke klase. Ali nedostatak ovakvog pristupa je to što je svako kodiranje podložno greškama. Naime, u programu je vrlo lako uvesti neku pogrešnu vrednost za godišnje doba: recimo, broj 0 ili slovo `z` ili dodatni objekat pored četiri „pravih". Bolje rešenje je definisanje novog nabrojivog tipa:

```
public enum GodišnjeDoba {PROLEĆE, LETO, JESEN, ZIMA}
```

Na ovaj način se definiše nabrojivi tip `GodišnjeDoba` koji se sastoji od četiri vrednosti čija su imena `PROLEĆE`, `LETO`, `JESEN` i `ZIMA`. Po konvenciji kao i

za obične konstante, vrednosti nabrojivog tipa se pišu velikim slovima i sa donjom crtom ukoliko se sastoje od više reči.

U pojednostavljenom opštem obliku, definicija nabrojivog tipa se piše na sledeći način:

```
enum ime-tipa {lista-konstanti}
```

U definiciji nabrojivog tipa, njegovo ime se navodi iza službene reči `enum` kao prost identifikator *`ime-tipa`*. Vrednosti nabrojivig tipa navode se, odvojene zapetama, unutar vitičastih zagrada kao *`lista-konstanti`*. Pri tome, svaka konstanta u listi mora biti prost identifikator.

Definicija nabrojivog tipa ne može stajati unutar metoda, jer nabrojivi tip mora biti član klase. Tehnički, nabrojivi tip je specijalna vrsta ugnježđene klase koja sadrži konačan broj instanci (konstanti) koje su navedene unutar vitičastih zagrada u definiciji nabrojivog tipa.[1] Tako, u prethodnom primeru, klasa `GodišnjeDoba` ima tačno četiri instance i nije moguće konstruisati nove.

Nakon što se definiše nabrojivi tip, on se može koristiti na svakom mestu u programu gde je dozvoljeno pisati tip podataka. Na primer, mogu se deklarisati promenljive nabrojivog tipa:

```
GodišnjeDoba modnaSezona;
```

Promenljiva `modnaSezona` može imati jednu od četiri vrednosti nabrojivog tipa `GodišnjeDoba` ili specijalnu vrednost `null`. Ove vrednosti se promenljivoj `modnaSezona` mogu dodeliti naredbom dodele, pri čemu se mora koristiti puno ime konstante tipa `GodišnjeDoba`. Na primer:

```
modnaSezona = GodišnjeDoba.LETO;
```

Promenljive nabrojivog tipa se mogu koristiti i u drugim naredbama gde to ima smisla. Na primer:

```
if (modnaSezona == GodišnjeDoba.LETO)
   System.out.println("Krpice za jun, jul i avgust.");
else
   System.out.println(modnaSezona);
```

[1] Definicija nabrojivog tipa ne mora biti ugnježđena u nekoj klasi, već se može nalaziti u svojoj posebnoj datoteci. Na primer, definicija nabrojivog tipa `GodišnjeDoba` može stajati u datoteci `GodišnjeDoba.java`.

Iako se nabrojive konstante tehnički smatraju da su objekti, primetimo da se u `if` naredbi ne koristi metod `equals()` nego operator ==. To je zato što svaki nabrojivi tip ima fiksne, unapred poznate instance i zato je za proveru njihove jednakosti dovoljno uporediti njihove reference. Primetimo isto tako da se drugim metodom `println()` prikazuje samo prosto ime konstante nabrojivog tipa (bez prefiksa `GodišnjeDoba`).

Nabrojivi tip je zapravo klasa, a svaka konstanta nabrojivog tipa je `public final static` polje te klase, iako se nabrojive konstante ne pišu sa ovim modifikatorima. Vrednosti ovih polja su reference na objekte koji pripadaju klasi nabrojivog tipa, a jedan takav objekat odgovara svakoj nabrojivoj konstanti. To su jedini objekti klase nabrojivog tipa i novi se nikako ne mogu konstruisati. Prema tome, ovi objekti predstavljaju moguće vrednosti nabrojivog tipa, a nabrojive konstante su polja koje ukazuju na te objekte.

Pošto se tehnički podrazumeva da nabrojivi tipovi predstavljaju klase, oni pored nabrojivih konstanti mogu sadržati polja, metode i konstruktore kao i obične klase. Naravno, konstruktori se jedino pozivaju kada se konstruišu objekti-konstante koje su navedene kao vrednosti nabrojivih tipova. Ukoliko nabrojivi tip sadrži dodatne članove, oni se navode iza liste konstanti koja se mora završiti tačkom-zapetom. Na primer:

```
public enum GodišnjeDoba {
   PROLEĆE('P'), LETO('L'), JESEN('J'), ZIMA('Z');

   private char skraćenica;                // polje

   private GodišnjeDoba(char skraćenica) {  // konstruktor
      this.skraćenica = skraćenica;
   }

   public char getSkraćenica() {           // geter metod
      return skraćenica;
   }
}
```

Svi nabrojivi tipovi nasleđuju jednu klasu `Enum` i zato se u radu s njima mogu koristiti metodi koji su definisani u klasi `Enum`. Najkorisniji od njih je metod `toString()` koji vraća ime nabrojive konstante u obliku stringa.

Na primer:

```
modnaSezona = GodišnjeDoba.LETO;
System.out.println(modnaSezona);
```

U drugom redu ovog fragmenta u naredbi `println`, kao i za svaku promenljivu klasnog tipa, implicitno se poziva metod `modnaSezona.toString()` i dobija se kao rezultat string `"LETO"` (koji se na ekranu prikazuje bez navodnika).

Obrnutu funkciju metoda `toString()` u klasi `Enum` obavlja statički metod `valueOf()` čije je zaglavlje:

```
static Enum valueOf(Class nabrojivi-tip, String konstanta)
```

Metod `valueOf()` kao rezultat vraća nabrojivu konstantu datog nabrojivog tipa sa datim imenom u obliku stringa. Na primer:

```
Scanner tastatura = new Scanner(System.in);
System.out.print("Unesite godišnje doba: ");
String g = tastatura.nextLine();

modnaSezona = (GodišnjeDoba) Enum.valueOf(
                    GodišnjeDoba.class, g.toUpperCase());
```

Ukoliko se prilikom izvršavanja ovog programskog fragmenta preko tastature unese string `zima`, promenljiva `modnaSezona` kao vrednost dobija nabrojivu konstantu `GodišnjeDoba.ZIMA`.

Objektni metod `ordinal()` u klasi `Enum` kao rezultat daje redni broj nabrojive konstante, brojeći od nule, u listi konstanti navedenih u definiciji nabrojivog tipa. Na primer, `GodišnjeDoba.PROLEĆE.ordinal()` daje vrednost 0 tipa `int`, `GodišnjeDoba.LETO.ordinal()` daje vrednost 1 tipa `int` i tako dalje.

Najzad, svaki nabrojivi tip ima statički metod `values()` koji kao rezultat vraća niz svih vrednosti nabrojivog tipa. Na primer, niz `gd` od četiri elementa sa vrednostima

```
GodišnjeDoba.PROLEĆE
GodišnjeDoba.LETO
GodišnjeDoba.JESEN
GodišnjeDoba.ZIMA
```

može se definisati na sledeći način:

```
GodišnjeDoba[] gd = GodišnjeDoba.values();
```

Metod `values()` se često koristi u paru sa `for-each` petljom kada je potrebno obraditi sve konstante nabrojivog tipa. Na primer, izvršavanjem ovog programskog fragmenta,

```
enum Dan {
   PONEDELJAK, UTORAK, SREDA, ČETVRTAK, PETAK, SUBOTA, NEDELJA};

for (Dan d : Dan.values()) {
   System.out.print(d);
   System.out.print(" je dan pod rednim brojem ");
   System.out.println(d.ordinal());
}
```

na ekranu se kao rezultat dobija:

```
PONEDELJAK je dan pod rednim brojem 0
UTORAK je dan pod rednim brojem 1
SREDA je dan pod rednim brojem 2
ČETVRTAK je dan pod rednim brojem 3
PETAK je dan pod rednim brojem 4
SUBOTA je dan pod rednim brojem 5
NEDELJA je dan pod rednim brojem 6
```

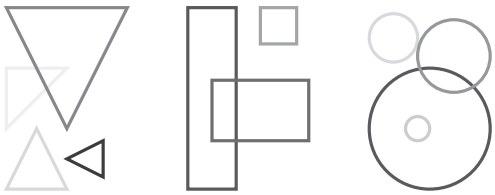
Nabrojivi tipovi se mogu koristiti i u paru sa naredbom `switch`. Preciznije, izraz u naredbi `switch` može biti nabrojivog tipa. Konstante uz klauzule `case` onda moraju biti nabrojive konstante odgovarajućeg tipa, pri čemu se moraju pisati njihova prosta imena a ne puna. Na primer:

```
switch (modnaSezona) {
   case PROLEĆE:  // pogrešno je GodišnjeDoba.PROLEĆE
      System.out.println("Krpice za mart, april i maj.");
      break;
   case LETO:
      System.out.println("Krpice za jun, jul i avgust.");
      break;
   case JESEN:
      System.out.println("Krpice za sept., okt. i nov.");
```

```
            break;
        case ZIMA:
            System.out.println("Krpice za dec., jan. i feb.");
            break;
    }
```

5.2 Apstraktne klase

Da bismo bolje razumeli apstraktne klase, razmotrimo jedan program koji služi za crtanje raznih geometrijskih oblika na ekranu među kojima su mogući trouglovi, pravougaonici i krugovi. Primer objekata ovog programa je prikazan na slici 5.1.

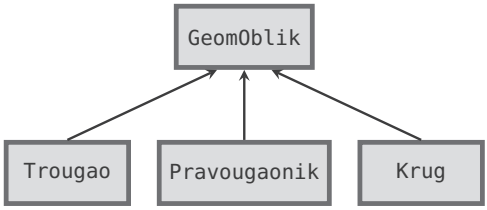


Slika 5.1: Geometrijski oblici programa za crtanje.

Nakon analize problema i imajući u vidu objektno orijentisani pristup, recimo da je izabrano da se definišu tri klase `Trougao`, `Pravougaonik` i `Krug` koje će predstavljati tri tipa mogućih geometrijskih oblika u programu. Pored toga, ove tri klase će nasleđivati zajedničku klasu `GeomOblik` u kojoj su definisana zajednička svojstva koje poseduju sva tri oblika. Klasnim dijagramom na slici 5.2 prikazano je stablo nasleđivanja klase `GeomOblik`.

Zajednička klasa `GeomOblik` može imati, između ostalog, objektna polja za boju, poziciju na ekranu i veličinu nekog geometrijskog oblika, kao i objektni metod za crtanje određenog geometrijskog oblika. Na primer:

```
public class GeomOblik {
    .
    . // Opšta polja i metodi geometrijskog oblika
    .
```



Slika 5.2: Hijerarhija klasa geometrijskih oblika.

```
    public void nacrtaj() {
        .
        . // Naredbe za crtanje geometrijskog oblika
        .
    }
}
```

Postavlja se pitanje kako napisati metod `nacrtaj()` u klasi `GeomOblik`? Problem je u tome što se svaki tip geometrijskih oblika crta na drugačiji način. To praktično znači da svaka klasa za trouglove, pravougaonike i krugove mora imati sopstveni metod `nacrtaj()` koji nadjačava verziju tog metoda u klasi `GeomOblik`:

```
public class Trougao extends GeomOblik {
    .
    . // Ostala polja i metodi
    .
    public void nacrtaj() {
        .
        . // Naredbe za crtanje trougla
        .
    }
}

public class Pravougaonik extends GeomOblik {
    .
    . // Ostala polja i metodi
    .
```

```
    public void nacrtaj() {
        .
        . // Naredbe za crtanje pravougaonika
        .
    }
}

public class Krug extends GeomOblik {
    .
    . // Ostala polja i metodi
    .
    public void nacrtaj() {
        .
        . // Naredbe za crtanje kruga
        .
    }
}
```

Ako je definisana promenljiva klasnog tipa `GeomOblik`, na primer,

```
GeomOblik oblik;
```

onda promenljiva `oblik` u programu može ukazivati, prema principu podtipa, na objekat svakog od podtipova `Trougao`, `Pravougaonik` ili `Krug`. Tokom izvršavanja programa, vrednost promenljive `oblik` se može menjati i ona može čak ukazivati na objekte ovih različitih tipova u različitom trenutku. Ali zbog polimorfizma, kada se izvršava naredba poziva

```
oblik.nacrtaj();
```

onda se poziva metod `nacrtaj()` iz odgovarajuće klase kojoj pripada objekat na koji trenutno ukazuje promenljiva `oblik`. Primetimo da se samo na osnovu teksta programa ne može reći koji će se geometrijski oblik nacrtati prethodnom naredbom, jer to zavisi od tipa objekta na koji ukazuje promenljiva `oblik` u trenutku izvršavanja te naredbe. Ali ono što je bitno je da mehanizam dinamičkog vezivanja metoda u Javi obezbeđuje da se pozove pravi metod `nacrtaj()` i tako nacrta odgovarajući geometrijski oblik na ekranu.

Ovo što je do sada rečeno ipak nije odgovorilo na naše polazno pitanje:

kako napisati metod `nacrtaj()` u klasi `GeomOblik`, odnosno šta u njemu treba uraditi za crtanje opšteg geometrijskog oblika? Odgovor je zapravo vrlo jednostavan: metod `nacrtaj()` u klasi `GeomOblik` ne treba da radi ništa!

Klasa `GeomOblik` služi samo za predstavljanje geometrijskih oblika u opštem smislu i zato ne postoji način da se nacrta nešto što nije potpuno poznato. Samo specifični, konkretni oblici kao što su trouglovi, pravougaonici i krugovi mogu da se nacrtaju. Ali ako metod `nacrtaj()` u klasi `GeomOblik` ne treba da radi ništa, to onda otvara pitanje zašto uopšte moramo imati taj metod u klasi `GeomOblik`?

Metod `nacrtaj()` ne može se prosto izostaviti iz klase `GeomOblik`, jer onda za promenljivu `oblik` tipa `GeomOblik` naredba poziva

```
oblik.nacrtaj();
```

ne bi bila ispravna. To je zato što bi se prilikom prevođenja programa proveravalo da li klasa `GeomOblik`, koja je deklarisani tip promenljive `oblik`, sadrži metod `nacrtaj()`. Ako to nije slučaj, prevođenje ne bi uspelo i ne bi se dobila izvršna verzija programa.

S druge strane, verzija metoda `nacrtaj()` iz klase `GeomOblik` se u programu neće nikad ni pozivati. Naime, ako se bolje razmisli, u programu ne postoji potreba da se ikad konstruiše stvarni objekat tipa `GeomOblik`. Ima razona da se deklariše neka promenljiva tipa `GeomOblik` (kao što je to `oblik`), ali objekat na koji ona ukazuje uvek će pripadati nekoj klasi koja nasleđuje klasu `GeomOblik`. Zato se za klasu `GeomOblik` kaže da je to apstraktna klasa.

Apstraktna klasa je klasa koja se ne koristi za konstruisanje objekata, nego samo kao osnova za nasleđivanje. Drugim rečima, apstraktna klasa služi samo za predstavljanje zajedničkih svojstava svih njenih klasa naslednica. Za klasu koja nije apstraktna se kaže da je *konkretna klasa*. U programu dakle mogu se konstruisati samo objekti koji pripadaju konkretnim klasama, ali ne i apstraktnim. Promenljive čiji deklarisani tip predstavlja neka apstraktna klasa mogu zato ukazivati samo na objekte koji pripadaju konkretnim klasama naslednicama te apstraktne klase.

Slično, za metod `nacrtaj()` u klasi `GeomOblik` kaže se da je *apstraktni metod*, jer on nije ni predviđen za pozivanje. U stvari, nema ništa logičnog što bi on mogao praktično da uradi, jer stvarno crtanje obavljaju verzije tog metoda koje ga nadjačavaju u klasama koje nasleđuju klasu `GeomOblik`. On

se mora nalaziti u klasi `GeomOblik` samo da bi se na neki način obezbedilo da svi objekti nekog podtipa od `GeomOblik` imaju svoj metod `nacrtaj()`. Pored toga, uloga apstraktnog metoda `nacrtaj()` u klasi `GeomOblik` je da definiše zajednički oblik zaglavlja svih stvarnih verzija tog metoda u konkretnim klasama koje nasleđuju klasu `GeomOblik`. Ne postoji dakle nijedan razlog da apstraktni metod `nacrtaj()` u klasi `GeomOblik` sadrži ikakve naredbe.

Klasa `GeomOblik` i njen metod `nacrtaj()` su po svojoj prirodi dakle apstraktni entiteti. Ta činjenica se programski može označiti dodavanjem modifikatora `abstract` u zaglavlju definicije klase ili metoda. Dodatno, u slučaju apstraktnog metoda, telo takvog metoda kojeg čini blok naredbi između vitičastih zagrada se ne navodi, nego se zamenjuje tačkom-zapetom. Zato nova, proširena definicija klase `GeomOblik` kao apstraktne klase može imati sledeći oblik:

```
public abstract class GeomOblik {   // apstraktna klasa

   private Color boja;

   public void oboji(Color novaBoja) {
      boja = novaBoja;
      nacrtaj();
   }

   public abstract void nacrtaj();  // apstraktni metod
   .
   . // Ostala polja i metodi
   .
}
```

U opštem slučaju, klasa sa jednim apstraktnim metodom, ili više njih, mora se i sama definisati da bude apstraktna. (Klasa se može definisati da bude apstraktna čak i ako nema nijedan apstraktni metod.) Pored apstraktnih metoda, apstraktna klasa može imati konkretne metode i polja. Tako, u prethodnom primeru apstraktne klase `GeomOblik`, polje `boja` i metod `oboji()` su konkretni članovi te apstraktne klase.

Klasa koja je u programu definisana da bude apstraktna ne može se in-

stancirati, odnosno nije dozvoljeno operatorom `new` konstruisati objekte te klase. Na primer, ako se u programu napiše `new GeomOblik()`, dobija se poruka o grešci prilikom prevođenja programa.

Iako se apstraktna klasa ne može instancirati, ona može imati konstruktore. Kao i kod konkretne klase, ukoliko nijedan konstruktor nije eksplicitno definisan u apstraktnoj klasi, automatski joj se dodaje podrazumevani konstruktor bez parametara. Ovo ima smisla, jer apstraktna klasa može imati konkretna polja koja možda treba inicijalizovati posebnim vrednostima, a ne onim podrazumevanim. Naravno, konstruktore apstraktne klase nije moguće pozivati neposredno iza operatora `new` za konstruisanje objekata apstraktne klase, nego samo (direktno ili indirektno) koristeći službenu reč `super` u konstruktorima klasa koje nasleđuju apstraktnu klasu.

Mada se objekti apstraktne klase ne mogu konstruisati, naglasimo još jednom da se klasne promenljive mogu deklarisati da budu tipa neke apstraktne klase. Takve promenljive međutim moraju ukazivati na objekte konkretnih klasa koje su naslednice apstraktne klase. Na primer:

```
GeomOblik oblik = new Pravougaonik();
```

Apstraktni metodi služe kao „čuvari mesta" za metode koji će biti definisani u klasama naslednicama. Da bi neka klasa koja nasleđuje apstraknu klasu bila konkretna, ta klasa-naslednica mora sadržati definicije konkretnih nadjačanih verzija svih apstraktnih metoda nasleđene klase. To znači da ukoliko se proširuje neka apstraktna klasa, onda postoje dve mogućnosti. Prva je da se u klasi-naslednici ostavi da neki ili svi apstraktni metodi budu nedefinisani. Onda je i nova klasa apstraktna i zato se mora navesti službena reč `abstract` u zaglavlju njene definicije. Druga mogućnost je da se u klasi-naslednici definišu svi nasleđeni apstraktni metodi i onda je ta nova klasa jedna obična konkretna klasa.

5.3 Interfejsi

Neki objektno orijentisani jezici (na primer, C++) dozvoljavaju da klasa može direktno proširivati dve klase ili čak više njih. Ovo takozvano *višestruko nasleđivanje* nije dozvoljeno u Javi, već neka klasa u Javi može direkt-

no proširivati samo jednu klasu. Međutim, u Javi postoji koncept *interfejsa* koji obezbeđuje slične mogućnosti kao višestruko nasleđivanje, ali ne povećava značajno složenost programskog jezika.

Pre svega, otklonimo jednu terminološku nepreciznost. Termin „interfejs" se ponekad koristi u vezi sa slikom jednog metoda kao crne kutije. Interfejs nekog metoda se sastoji od informacija koje je potrebno znati radi ispravnog pozivanja tog metoda u programu. To su ime metoda, tip njegovog rezultata i redosled i tip njegovih parametara. Svaki metod ima i implementaciju koja se sastoji od tela metoda u kojem se nalazi niz naredbi koje se izvršavaju kada se metod pozove.

Pored ovog značenja u vezi sa metodima, termin „interfejs" u Javi ima i dodatno, potpuno različito koncepcijsko značenje za koje postoji službena reč `interface`. Da ne bude zabune, naglasimo da se u ovoj knjizi podrazumeva ovo značenje kada se govori o interfejsu. Interfejs u ovom smislu se sastoji od skupa apstraktnih metoda (tj. interfejsa objektnih metoda), bez ikakve pridružene implementacije. (U stvari, Java interfejsi mogu imati i `static final` konstante, ali to nije njihova prvenstvena odlika.) Neka klasa *implementira* dati interfejs ukoliko sadrži definicije svih (apstraktnih) metoda tog interfejsa.

Interfejs u Javi nije dakle klasa, već skup mogućnosti koje implementirajuće klase moraju imati. Razmotrimo jedan jednostavan primer interfejsa u Javi. Metodom `sort()` iz klase `Arrays` može se sortirati niz objekata, ali pod jednim uslovom: ti objekti moraju biti uporedivi. Tehnički to znači da ti objekti moraju pripadati klasi koja implementira interfejs `Comparable`. Ovaj interfejs je u Javi definisan na sledeći način:

```
public interface Comparable {
   int compareTo(Object o);
}
```

Kao što se iz ovog primera vidi, definicija interfejsa u Javi je vrlo slična definiciji klase, osim što se umesto reči `class` koristi reč `interface` i definicije svih metoda u interfejsu se izostavljaju. Svi metodi nekog interfejsa automatski postaju javni. Zbog toga nije neophodno (ali nije ni greška) navesti specifikator pristupa `public` u zaglavlju deklaracije nekog metoda u interfejsu.

U specifičnom primeru interfejsa `Comparable`, neka klasa koja imple-

mentira taj interfejs mora definisati metod `compareTo()`, a taj metod mora imati parametar tipa `Object` i vratiti rezultat tipa `int`. Naravno, ovde postoji dodatni semantički uslov koji se implicitno podrazumeva, ali se njegova ispunjenost ne može obezbediti u interfejsu: u izrazu `x.compareTo(y)` metod `compareTo()` mora zaista uporediti dva objekta `x` i `y` dajući rezultat koji ukazuje koji od njih je veći. Preciznije, ovaj metod treba da vrati negativan broj ako je `x` manje od `y`, nulu ako su jednaki i pozitivan broj ako je `x` veće od `y`.

Uz definicije svih metoda interfejsa, svaka klasa koja implementira neki interfejs mora sadržati i eksplicitnu deklaraciju da ona implementira dati interfejs. To se postiže pisanjem službene reči `implements` i imena tog interfejsa iza imena klase koja se definiše. Na primer:

```
public class Radnik implements Comparable {

   String ime;    // ime i prezime radnika
   double plata;  // plata radnika
   .
   . // Ostala polja i metodi
   .
   public int compareTo(Object o) {

      Radnik drugiRadnik = (Radnik) o;

      if (this.plata < drugiRadnik.plata) return -1;
      if (this.plata > drugiRadnik.plata) return +1;
      return 0;
   }
}
```

Obratite pažnju na to da se radnici u metodu `compareTo()` klase `Radnik` upoređuju na osnovu njihove plate. (Ako objekti nisu jednaki, pozitivne ili negativne vrednosti koje su rezultat metoda `compareTo()` nisu bitni.) Isto tako, za ovaj konkretan metod se mora navesti specifikator pristupa `public`, iako to nije bilo obavezno za njegovu apstraktnu deklaraciju u interfejsu `Comparable`.

Pretpostavimo da u programu konkretna klasa `Radnik` koja implementi-

ra interfejs `Comparable` služi za predstavljanje pojedinih radnika neke firme. Ako dodatno pretpostavimo da su svi radnici predstavljeni nizom baznog tipa `Radnik`, onda se ovi radnici mogu jednostavno sortirati po rastućem redosledu njihovih plata:

```
Radnik[] radnici = new Radnik[500];
   .
   . // Druge naredbe
   .
Arrays.sort(radnici);
```

Napomenimo još da se u vezi sa interfejsima često koristi jedan kraći terminološki izraz. Naime, kaže se da neki objekat implementira interfejs, iako se zapravo misli da taj objekat pripada klasi koja implementira dati interfejs. Tako, u prethodnom primeru može se reći da objekat tipa `Radnik` implementira interfejs `Comparable`.

Osobine interfejsa

Po svojoj glavnoj nameni, interfejsi sadrže jedan ili više apstraktnih metoda (ali mogu imati i konstante). Interfejsi se slično klasama pišu u datotekama sa ekstenzijom `java`, a prevedena verzija (bajtkod) interfejsa se nalazi u datoteci sa ekstenzijom `class`. Ono što je isto tako važno naglasiti jeste šta interfejsi ne mogu imati. Interfejsi ne mogu imati objektna polja, niti metodi interfejsa mogu imati implementaciju. Dodavanje polja i implementacije metoda je zadatak klasa koje implementiraju interfejs. Prema tome, interfejsi su slični apstraktnim klasama bez objektnih polja.

Iako su slični (apstraktnim) klasama, interfejsi nisu klase. To znači da se interfejsi ne mogu koristiti za konstruisanje objekata. Pisanjem, na primer,

```
new Comparable()
```

proizvodi se greška u programu.

Slično kao za klase, definicijom nekog interfejsa se u programu tehnički uvodi novi klasni tip podataka. Vrednosti tog klasnog tipa su objekti klasa koje implementiraju definisani interfejs. To znači da se ime interfejsa može koristiti za tip promenljive u naredbi deklaracije, kao i za tip formalnog parametra i za tip rezulatata u definiciji nekog metoda. Prema tome, u Javi

tip može biti klasa, interfejs ili jedan od osam primitivnih tipova. To su jedine mogućnosti, ali samo klase služe za konstruisanje novih objekata.

U programu se dakle mogu deklarisati klasne promenljive čiji tip predstavlja neki interfejs, na primer:

```
Comparable x;  // OK
```

Pošto se interfejsi implementiraju, a ne nasleđuju kao apstraktne klase, interfejsna promenljiva (tj. promenljiva čiji je tip neki interfejs) može ukazivati samo na objekte onih klasa koje implementiraju dati interfejs. Tako, za prethodni primer klase `Radnik` koja implementira interfejs `Comparable` možemo pisati:

```
Comparable x = new Radnik();  // OK
```

Dalje, poznato je da se operator `instanceof` može koristiti za proveravanje da li neki objekat pripada specifičnoj klasi. Taj operator se može koristiti i za proveravanje da li neki objekat implementira dati interfejs:

```
if (nekiObjekat instanceof Comparable) . . .
```

Kao što se mogu praviti hijerarhije klasa, tako se mogu proširivati i interfejsi. Interfejs koji proširuje drugi interfejs nasleđuje sve njegove deklarisane metode. Hijerarhije interfejsa su nezavisne od hijerarhija klasa, ali imaju istu ulogu: to je način da se od nekog najopštijeg entiteta na vrhu hijerarhije izrazi veći stepen specijalizacije kako se prelazi na donje nivoe hijerarhije. Na primer:

```
public interface Radio {
   void uključi();
   void isključi();
}

public interface KlasičniRadio extends Radio {
   void izaberiStanicu();
}
public interface InternetRadio extends Radio {
   void izaberiSajt();
}
```

Za razliku od klase koja može direktno proširiti samo jednu klasu, in-

terfejs može direktno proširivati više interfejsa. U tom slučaju se iza reči `extends` navode svi nazivi interfejsa koji se proširuju, razdvojeni zapetama.

Mada interfejsi ne mogu imati objektna polja i statičke metode, u njima se mogu nalaziti konstante. Na primer:

```
public interface KlasičniRadio extends Radio {
   double B92 = 92.5;
   double STUDIO_B = 99.1;
   void izaberiStanicu();
}
```

Slično kao što se za metode interfejsa podrazumeva da su `public` metodi, sva navedena polja interfejsa automatski postaju `public static final` konstante.

U uvodnom delu o interfejsima smo spomenuli da u Javi nije omogućeno višestruko nasleđivanje klasa. Ali to se može nadomestiti time što klase mogu istovremeno *implementirati* više interfejsa. U deklaraciji jedne takve klase se nazivi interfejsa koji se implementiraju navode razdvojeni zapetama. Na primer, u Javi je definisan interfejs `Cloneable` tako da, po konvenciji, klase koje implementiraju ovaj interfejs treba da nadjačaju metod `clone()` klase `Object`. Zato ukoliko želimo da se objekti neke klase mogu, recimo, klonirati i upoređivati, u definiciji klase tih objekata treba navesti da se implementiraju interfejsi `Cloneable` i `Comparable`. Na primer:

```
public class Radnik implements Cloneable, Comparable { ... }
```

Naravno, telo ove klase `Radnik` sada mora sadržati definicije metoda `clone()` i `compareTo()`.

Interfejsi i apstraktne klase

Pažljiviji čitaoci se verovatno pitaju zašto su potrebni interfejsi kada se oni naizgled mogu potpuno zameniti apstraktnim klasama. Zašto recimo interfejs `Comparable` ne bi mogao biti apstraktna klasa? Na primer:

```
abstract class Comparable { // Zašto ne?
   public abstract int compareTo(Object o);
}
```

Onda bi klasa `Radnik` mogla prosto da nasledi ovu apstraktnu klasu i da definiše metod `compareTo()`:

```
public class Radnik extends Comparable { // Zašto ne?

   String ime;    // ime i prezime radnika
   double plata;  // plata radnika
   .
   . // Ostala polja i metodi
   .
   public int compareTo(Object o) {

      Radnik drugiRadnik = (Radnik) o;

      if (this.plata < drugiRadnik.plata) return -1;
      if (this.plata > drugiRadnik.plata) return +1;
      return 0;
   }
}
```

Glavni problem kod korišćenja apstraktnih klasa za izražavanje apstraktnih mogućnosti objekata je to što neka klasa u Javi može direktno naslediti samo jednu klasu. Ako pretpostavimo da klasa `Radnik` već nasleđuje jednu klasu, recimo `Osoba`, onda klasa `Radnik` ne može naslediti i drugu:

```
public class Radnik extends Osoba, Comparable  // GREŠKA
```

Ali neka klasa u Javi može implementirati više interfejsa (sa nasleđivanjem jedne klase ili bez toga):

```
public class Radnik extends Osoba implements Comparable  // OK
```

5.4 Ugnježđene klase

Do sada smo naučili da se u Javi neke programske konstrukcije mogu ugnježđavati, na primer upravljačke naredbe, dok se druge ne mogu ugnježđavati, na primer metodi. Postavlja se sada pitanje da li se klase, kao najsloženiji vid programske celine, mogu ugnježđavati i da li to ima smisla?

Odgovor na oba pitanja je, možda malo iznenađujuće, potvrdan.

Ugnježđena klasa je klasa koja je definisana unutar druge klase. Ovakav pristup u dizajnu programa je koristan uglavnom iz tri razloga:

1. Kada je neka klasa pomoćna za glavnu klasu i ne koristi se van te veće klase, nema razloga da pomoćna klasa bude izdvojena kao samostalna klasa.
2. Kada neka klasa služi za konstruisanje samo jednog objekta te klase, prirodnije je takvu klasu definisati (na kraći način) baš na mestu konstruisanja njenog jedinstvenog objekta.
3. Metodi ugnježđene klase mogu koristiti sva polja obuhvatajuće klase — čak i njena privatna polja, kao i obrnuto.

Tema ugnježđenih klasa je prilično složena i svi njeni detalji prevazilaze okvire ove knjige. Zato ćemo ovde pomenuti samo one mogućnosti ugnježđavanja klasa u Javi koje su dovoljne za razumevanje jednostavnijih programskih tehnika.[2] Primena ugnježđenih klasa je naročito korisna kod grafičkog programiranja o čemu se govori u poglavlju 6. U tom poglavlju ćemo upoznati i praktične primere u kojima se najčešće koriste ugnježđene klase.

Pre svega, neka klasa se može definisati unutar druge klase na isti način na koji se definišu uobičajeni članovi spoljašnje klase: polja i metodi. To znači da se ugnježđena klasa može deklarisati sa ili bez modifikatora `static`, pa se prema tome razlikuju *statičke* i *objektne* ugnježđene klase:

```
class SpoljašnjaKlasa {
   .
   . // Ostala polja i metodi
   .
   static class StatičkaUgnježđenaKlasa {
      . . .
   }

   class ObjektnaUgnježđenaKlasa {
      . . .
```

[2] U stvari, pravilnije je možda govoriti o ugnježđavanju tipova, a ne klasa, jer se mogu definisati i interfejsi unutar klasa. Tu dodatnu komplikaciju nećemo razmatrati u knjizi.

```
    }
}
```

Statičke i objektne ugnježđene klase se smatraju običnim članovima obuhvatajuće klase, što znači da se za njih mogu koristiti specifikatori pristupa `public`, `private` i `protected`, ili nijedan od njih. (Podsetimo se da se obične klase mogu definisati samo sa specifikatorom pristupa `public` ili bez njega.)

Druga mogućnost kod ugnježđavanja klasa je definisanje jedne klase unutar nekog metoda (tačnije, unutar bloka naredbi) druge klase. Takve ugnježđene klase mogu biti *lokalne* ili *anonimne*. Ove ugnježđene klase imaju koncepcijski slične osobine kao lokalne promenljive metoda.[3]

Statičke ugnježđene klase

Definicija statičke ugnježđene klase ima isti oblik kao definicija obične klase, osim što se nalazi unutar druge klase i sadrži modifikator `static`.[4] Statička ugnježđena klasa je deo statičke strukture obuhvatajuće klase, logički potpuno isto kao statička polja i metodi te klase. Kao i statički metod, na primer, statička ugnježđena klasa nije vezana za objekte obuhvatajuće klase i postoji nezavisno od njih.

U metodima obuhvatajuće klase se statička ugnježđena klasa može koristiti za konstruisanje objekata na uobičajeni način. Ako nije deklarisana kao privatna, statička ugnježđena klasa se takođe može koristiti izvan obuhvatajuće klase, ali se onda tačka-notacijom mora navesti njeno članstvo u obuhvatajućoj klasi.

Na primer, pretpostavimo da klasa `KoordSistem` predstavlja koordinatni sistem u dvodimenzionalnoj ravni. Pretpostavimo još da klasa `KoordSistem` sadrži statičku ugnježđenu klasu `Duž` koja predstavlja jednu duž između dve tačke. Na primer:

[3] Nestatičke ugnježđene klase (tj. objektne, lokalne i anonimne) često se nazivaju *unutrašnje klase*.

[4] Nabrojivi tip definisan unutar neke klase smatra se statičkom ugnježđenom klasom, iako se u njegovoj definiciji ne navodi modifikator `static`.

```
public class KoordSistem {
   .
   . // Ostala polja i metodi
   .
   // Ugnježđena klasa koja predstavlja duž u ravni od
   // tačke (x1,y1) do tačke (x2,y2)
   public static class Duž {

      double x1, y1;
      double x2, y2;
   }
}
```

Objekat klase `Duž` bi se unutar nekog metoda klase `KoordSistem` konstruisao izrazom `new Duž()`. Izvan klase `KoordSistem` bi se morao koristiti izraz `new KoordSistem.Duž()`.

Obratite pažnju na to da kada se prevede klasa `KoordSistem`, dobijaju se zapravo dve datoteke bajtkoda. Iako se definicija klase `Duž` nalazi unutar klase `KoordSistem`, njen bajtkod se smešta u posebnu datoteku čije je ime `KoordSistem$Duž.class`. Bajtkod klase `KoordSistem` se smešta, kao što je to uobičajeno, u datoteku `KoordSistem.class`.

Statička ugnježđena klasa je vrlo slična regularnoj klasi na najvišem nivou. Jedna od razlika je to što se njeno ime hijerarhijski nalazi unutar prostora imena obuhvatajuće klase, slično kao što se ime regularne klase nalazi unutar prostora imena odgovarajućeg paketa. Pored toga, statička ugnježđena klasa ima potpun pristup statičkim članovima obuhvatajuće klase, čak i ako su oni deklarisani kao privatni. Obrnuto je takođe tačno: obuhvatajuća klasa ima potpun pristup svim članovima ugnježđene klase. To može biti jedan od razloga za definisanje statičke ugnježđene klase, jer tako jedna klasa može dobiti pristup privatnim članovima druge klase a da se pri tome ti članovi ne otkrivaju drugim klasama.

Objektne ugnježđene klase

Objektna ugnježđena klasa je klasa koja je definisana kao član obuhvatajuće klase bez službene reči `static`. Ako je statička ugnježđena klasa analog-

na statičkom polju ili statičkom metodu, onda je objektna ugnježđena klasa analogna objektnom polju ili objektnom metodu. Zbog toga je objektna ugnježđena klasa vezana zapravo za objekat obuhvatajuće klase.

Nestatički članovi neke obične klase određuju šta će se nalaziti u svakom konstruisanom objektu te klase. To važi i za objektne ugnježđene klase, bar logički, odnosno može se smatrati da svaki objekat obuhvatajuće klase ima sopstveni primerak objektne ugnježđene klase. Ovaj primerak ima pristup do svih posebnih primeraka polja i metoda nekog objekta, čak i onih koji su deklarisani kao privatni. Dva primerka objektne ugnježđene klase u različitim objektima se razlikuju, jer koriste različite primerke polja i metoda u različitim objektima. U stvari, pravilo na osnovu kojeg se može odlučiti da li ugnježđena klasa treba da bude statička ili objektna je jednostavno: ako ugnježđena klasa mora koristiti neko objektno polje ili objektni metod obuhvatajuće klase, ona i sama mora biti objektna.

Objektna ugnježđena klasa se izvan obuhvatajuće klase mora koristiti uz konstruisani objekat obuhvatajuće klase u formatu:

```
promenljiva.ObjektnaUgnježđenaKlasa
```

U ovom zapisu, *promenljiva* označava ime promenljive koja ukazuje na objekat obuhvatajuće klase. Ova mogućnost se u praksi ipak retko koristi, jer se objektna ugnježđena klasa obično koristi samo unutar obuhvatajuće klase, a onda je dovoljno navesti njeno prosto ime.

Da bi se konstruisao objekat koji pripada nekoj objektnoj ugnježđenoj klasi, mora se najpre imati objekat njene obuhvatajuće klase. Unutar obuhvatajuće klase se konstruiše objekat koji pripada njenoj objektnoj ugnježđenoj klasi za objekat na koji ukazuje promenljiva `this`. Objekat koji pripada objektnoj ugnježđenoj klasi se permanentno vezuje za objekat obuhvatajuće klase i ima potpun pristup svim članovima tog objekta.

Pokušajmo da ovo razjasnimo na jednom primeru. Posmatrajmo klasu koja predstavlja igru tablića sa kartama. Ova klasa može imati objektnu ugnježđenu klasu koja predstavlja igrače tablića:

```
public class Tablić {

   private Karta[] špil;  // špil karata za this igru tablića
```

```
    private class Igrač {  // jedan igrač this igre tablića
       . . .
    }
    .
    . // Ostali članovi klase Tablić
    .
}
```

Ako je `igra` promenljiva tipa `Tablić`, onda objekat igre tablića na koji `igra` ukazuje, koncepcijski sadrži svoj primerak objektne ugnježđene klase `Igrač`. U svakom objektnom metodu klase `Tablić`, novi igrač bi se na uobičajeni način konstruisao izrazom `new Igrač()`. (Ali izvan klase `Tablić` bi se novi igrač konstruisao izrazom `igra.new Igrač()`.) Unutar klase `Igrač`, svi objektni metodi imaju pristup objektnom polju `špil` obuhvatajuće klase `Tablić`.

Jedna igra tablića koristi svoj špil karata i ima svoje igrače; igrači neke druge igra tablića koriste drugi špil karata. Ovaj prirodan efekat je upravo postignut time što je klasa `Igrač` definisana da nije statička. Objekat klase `Igrač` u prethodnom primeru predstavlja igrača jedne konkretne igre tablića. Da je klasa `Igrač` bila definisana kao statička ugnježđena klasa, onda bi ona predstavljala igrača tablića u opštem smislu, nezavisno od konkretne igre tablića.

Lokalne klase

Lokalna klasa je ona koja je definisana unutar nekog metoda druge klase.[5] Pošto se svi metodi moraju nalaziti unutar neke klase, lokalne klase moraju biti ugnježđene unutar obuhvatajuće klase. Zbog toga lokalne klase imaju slične osobine kao objektne ugnježđene klase, mada predstavljaju potpuno različitu vrstu ugnježđenih klasa. Lokalne klase u odnosu na objektne ugnježđene klase stoje otprilike kao lokalne promenljive u odnosu na objektna polja klase.

Kao i neka lokalna promenljiva, lokalna klasa je vidljiva samo unutar metoda u kojem je definisana. Na primer, ako se neka objektna ugnježđena klasa koristi samo unutar jednog metoda svoje obuhvatajuće klase, obično

[5] Tačnije, lokalna klasa se može definisati i unutar nekog bloka naredbi u metodu.

ne postoji razlog zbog kojeg se ona ne može definisati unutar tog metoda, a ne kao članica obuhvatajuće klase. Na taj način se definicija klase pomera još bliže mestu gde se koristi, što skoro uvek doprinosi boljoj čitljivosti programskog koda.

Neke od značajnih osobina lokalnih klasa su:

- Slično objektnim ugnježđenim klasama, lokalne klase su vezane za obuhvatajući objekat i imaju pristup svim njegovim članovima, čak i privatnim. Prema tome, objekti lokalne klase su pridruženi jednom objektu obuhvatajuće klase čija se referenca u metodima lokalne klase može koristiti putem promenljive složenog imena `ObuhvatajućaKlasa.this`.

- Slično pravilu za oblast važenja lokalne promenljive, lokalna klasa važi samo unutar metoda u kojem je definisana i njeno ime se nikad ne može koristiti van tog metoda. Obratite ipak pažnju na to da objekti lokalne klase konstruisani u njenom obuhvatajućem metodu mogu postojati i nakon završetka izvršavanja tog metoda.

- Slično lokalnim promenljivim, lokalne klase se ne mogu definisati sa modifikatorima `public`, `private`, `protected` ili `static`. Ovi modifikatori mogu stajati samo uz članove klase i nisu dozvoljeni za lokalne promenljive ili lokalne klase.

- Pored članova obuhvatajuće klase, lokalna klasa može koristiti lokalne promenljive i parametre metoda pod uslovom da su oni deklarisani sa modifikatorom `final`. Ovo ograničenje je posledica toga što životni vek nekog objekta lokalne klase može biti mnogo duži od životnog veka lokalne promenljive ili parametra metoda u kojem je lokalna klasa definisana. (Podsetimo se da je životni vek lokalne promenljive ili parametra nekog metoda jednak vremenu izvršavanja tog metoda.) Zbog toga lokalna klasa mora imati svoje privatne primerke svih lokalnih promenljivih ili parametara koje koristi (što se automatski obezbeđuje od strane Java prevodioca). Jedini način da se obezbedi da lokalne promenljive ili parametri metoda budu isti kao privatni primerci lokalne klase jeste da se zahteva da oni budu `final`.

Anonimne klase

Anonimne klase su lokalne klase bez imena. One se uglavnom koriste u slučajevima kada je potrebno konstruisati samo jedan objekat neke klase. Umesto posebnog definisanja te klase i zatim konstruisanja njenog objekta samo na jednom mestu, oba efekta se istovremeno mogu postići pisanjem operatora `new` u jednom od dva specijalna oblika:

```
new natklasa(lista-argmenata) {
      metodi-i-promenljive
}
```

ili

```
new interfejs() {
      metodi-i-konstante
}
```

Ovi oblici operatora `new` se mogu koristiti na svakom mestu gde se može navesti običan operator `new`. Obratite pažnju na to da dok se definicija lokalne klase smatra naredbom nekog metoda, definicija anonimne klase uz operator `new` je formalno izraz. To znači da kombinovani oblik operatora `new` može biti u sastavu većeg izraza gde to ima smisla.

Oba prethodna izraza služe za definisanje bezimene nove klase čije se telo nalazi između vitičastih zagrada i istovremeno se konstruiše objekat koji pripada toj klasi. Preciznije, prvim izrazom se proširuje *`natklasa`* tako što se dodaju navedeni *`metodi-i-promenljive`*. Argumenti navedeni između zagrada se prenose konstruktoru *`natklase`* koja se proširuje. U drugom izrazu se podrazumeva da anonimna klasa implementira *`interfejs`* definisanjem svih njegovih deklarisanih metoda i da proširuje klasu `Object`. U svakom slučaju, glavni efekat koji se postiže je konstruisanje posebno prilagođenog objekta, baš na mestu u programu gde je i potreban.

Anonimne klase se često koriste za postupanje sa događajima koji nastaju pri radu programa koji koriste grafički korisnički interfejs. O detaljima rada sa elementima grafičkog korisničkog interfejsa biće više reči u poglavlju 6 koje je posvećeno grafičkom programiranju. Sada ćemo samo radi ilustracije, bez ulaženja u sve finese, pokazati programski fragment u kojem se konstruiše jednostavna komponenta grafičkog korisničkog interfejsa i definišu metodi koji se pozivaju kao reakcija na događaje koje proizvodi

ta komponenta. Taj grafički element je dugme, označeno prigodnim tekstom, koje proizvodi razne vrste događaja zavisno od korisničke aktivnosti s mišom nad njim.

U primeru u nastavku, najpre se konstruiše grafički element dugme koje je objekat standardne klase `Button` iz paketa `java.awt`. Nakon toga se za novokonstruisano dugme poziva metod `addMouseListener()` kojim se registruje objekat koji može da reaguje na događaje izazvane klikom miša na dugme, prelaskom strelice miša preko dugmeta i tako dalje. Tehnički to znači da argument poziva metoda `addMouseListener()` mora biti objekat koji pripada klasi koja implementira interfejs `MouseListener`. Bez možda potpunog razumevanja svih ovih detalja o kojima se više govori u poglavlju 6, ovde je za sada važno primetiti samo to da nam treba jedan jedini objekat posebne klase koju zato možemo definisati da bude anonimna klasa. Pored toga, specijalni oblik izraza sa operatorom `new` za konstruisanje tog objekta možemo pisati u samom pozivu metoda `addMouseListener()` kao njegov argument:

```
Button dugme = new Button("Pritisni me");  // dugme sa tekstom

dugme.addMouseListener(new MouseListener() { // početak tela
                                             // anonimne klase
   // Svi metodi interfejsa MouseListener se moraju definisati
   public void mouseClicked(MouseEvent e) {
      System.out.println("Kliknuto na dugme");
   }
   public void mousePressed(MouseEvent e) { }
   public void mouseReleased(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) { }
   public void mouseExited(MouseEvent e) { }
});  // srednja zagrada i tačka-zapeta završavaju poziv metoda!
```

Budući da anonimne klase nemaju ime, nije moguće definisati konstruktore za njih. Anonimne klase se obično koriste za proširivanje jednostavnih klasa čiji konstruktori nemaju parametre. Zato su zagrade u definiciji anonimne klase često prazne, kako u slučaju proširivanja klasa tako i u slučaju implementiranja interfejsa.

Sve anonimne klase obuhvatajuće klase prilikom prevođenja daju po-

sebne datoteke u kojima se nalazi njihov bajtkod. Na primer, ako je `NekaKlasa` ime obuhvatajuće klase, onda datoteke sa bajtkodom njenih anonimnih klasa imaju imena `NekaKlasa$1.class`, `NekaKlasa$2.class` i tako dalje.

GLAVA 6

Grafičko programiranje

Do sada smo upoznali samo proste, konzolne programe u Javi koji ulazne podatke učitavaju preko tastature i za izlazne podatke koriste tekstualni prikaz na ekranu. Međutim, svi moderni računari rade pod operativnim sistemima koji se zasnivaju na grafičkim prozorima. Zbog toga su današnji korisnici računara naviknuti da sa programima komuniciraju preko grafičkog korisničkog interfejsa (engl. *graphical user interface*, skraćeno GUI). Grafički korisnički interfejs se odnosi na vizuelni deo programa koji korisnik vidi na ekranu i preko koga može upravljati programom tokom njegovog izvršavanja.

Programski jezik Java omogućava naravno pisanje ovakvih grafičkih programa. U stvari, jedan od osnovnih ciljeva Jave od samog početka je bio da se obezbedi relativno lako pisanje grafičkih programa. Ovaj naglasak na grafičkim programima je zapravo razlog zašto su zanemareni konzolni programi i, recimo, način učitavanja podataka u njima nije tako jednostavan. U ovom poglavlju ćemo se upoznati sa osnovnim principima pisanja grafičkih programa u Javi.

6.1 Grafički programi

Grafički programi se znatno razlikuju od konzolnih programa. Najveća razlika je u tome što se konzolni program izvršava sinhrono (od početka do kraja, jedna naredba iza druge). U konzolnim programima se zato programski određuje trenutak kada korisnik treba da unese ulazne podatke i

kada se prikazuju izlazni podaci. Nasuprot tome, kod grafičkih programa je korisnik taj koji određuje kada hoće da unese ulazne podatke i najčešće on sâm određuje kada se prikazuju izlazni podaci programa. Za grafičke programe se zato kaže da su vođeni događajima (engl. *event-driven programs*). To znači da korisnikove aktivnosti, poput pritiska tasterom miša na neki grafički element ekrana ili pritiska na taster tastature, generišu događaje na koje program mora reagovati na odgovarajući način. Pri tome naravno, ovaj model rada programa u Javi je prilagođen objektno orijentisanom načinu programiranja: događaji su objekti, boje i fontovi su objekti, grafičke komponente su objekti, a na pojavu nekog događaja se pozivaju objektni metodi neke klase koji propisuju reakciju programa na taj događaj.

Grafički programi imaju mnogo bogatiji korisnički interfejs nego konzolni programi. Taj interfejs se može sastojati od grafičkih komponenti kao što su prozori, meniji, dugmad, polja za unos teksta, trake za pomeranje teksta i tako dalje. (Ove komponente se u Windows okruženju nazivaju i kontrole.) Grafički programi u Javi svoj korisnički interfejs grade programski, najčešće u toku inicijalizacije glavnog prozora programa. To obično znači da glavni metod Java programa konstruiše jednu ili više od ovih komponenti i prikazuje ih na ekranu. Nakon konstruisanja neke komponente, ona sledi svoju sopstvenu logiku kako se crta na ekranu i kako reaguje na događaje izazvane nekom korisnikovom akcijom nad njom.

Pored običnih, „pravih" ulaznih podataka, grafički programi imaju još jedan njihov oblik — događaje. Događaje proizvode razne korisnikove aktivnosti s fizičkim uređajima miša i tastature nad grafičkim elementima programa. Tu spadaju na primer izbor iz menija prozora na ekranu, pritisak tasterom miša na dugme, pomeranje miša, pritisak nekog tastera miša ili tastature i slično. Sve ove aktivnosti generišu događaje koji se prosleđuju grafičkom programu. Program prima događaje kao ulaz i obrađuje ih u delu koji se naziva *rukovalac događaja* (engl. *event handler* ili *event listener*). Zbog toga se često kaže da je logika grafičkih programa zasnovana na događajima korisničkog interfejsa.

Glavna podrška za pisanje grafičkih programa u Javi se sastoji od dve standardne biblioteke klasa. Starija biblioteka se naziva AWT (skraćeno od engl. *Abstract Window Toolkit*) i njene klase se nalaze u paketu `java.awt`. Karakteristika ove biblioteke koja postoji od prve verzije programskog je-

zika Java jeste to što obezbeđuje upotrebu minimalnog skupa komponenti grafičkog interfejsa koje poseduju sve platforme koje podržavaju Javu. Klase iz ove biblioteke se oslanjaju na grafičke mogućnosti konkretnog operativnog sistema, pa su AWT komponente morale da imaju najmanji zajednički imenitelj mogućnosti koje postoje na svim platformama. Zbog toga se za njih često kaže da izgledaju „podjednako osrednje" na svim platformama.

To je bio glavni razlog zašto su od verzije Java 1.2 mnoge klase iz biblioteke AWT ponovo napisane tako da ne zavise od konkretnog operativnog sistema. Nova biblioteka za grafičke programe u Javi je nazvana Swing i njene klase se nalaze u paketu `javax.swing`. Swing komponenete su potpuno napisane u Javi i zato podjednako izgledaju na svim platformama. Možda još važnije, te komponente rade na isti način nezavisno od operativnog sistema, tako da su otklonjeni i svi problemi u vezi sa različitim suptilnim greškama AWT komponenti prilikom izvšavanja na različitim platformama.

Obratite pažnju na to da biblioteka Swing nije potpuna zamena za biblioteku AWT, nego se nadovezuje na nju. U biblioteci Swing se nalaze grafičke komponente sa većim mogućnostima, ali se od biblioteke AWT nasleđuje osnovna arhitektura kao što je, recimo, mehanizam rukovanja događajima. U stvari, kompletno grafičko programiranje u Javi se bazira na biblioteci JFC (skraćeno od engl. *Java Foundation Classes*), koja obuhvata Swing/AWT i sadrži još mnogo više od toga.

Grafički programi u Javi se uglavnom zasnivaju na biblioteci Swing iz više razloga:

- Biblioteka Swing sadrži bogatu kolekciju GUI komponenti kojima se lako manipuliše u programima.
- Biblioteka Swing se minimalno oslanja na osnovni operativni sistem računara.
- Biblioteka Swing pruža korisniku konzistentan utisak i način rada nezavisno od platforme na kojoj se program izvršava.

Mnoge komponente u biblioteci Swing imaju svoje stare verzije u biblioteci AWT, ali imena odgovarajućih klasa su pažljivo birana kako ne bi dolazilo do njihovog sukoba u programima. (Imena klasa u biblioteci

Swing obično počinju velikim slovom J, na primer `JButton`.) Zbog toga je bezbedno u Java programima uvoziti oba paketa `java.awt` i `javax.swing`, što se obično i radi, jer je često sa novim komponentama iz biblioteke Swing potrebno koristiti i osnovne mehanizme rada s njima iz biblioteke AWT.

Primer: grafički ulaz/izlaz programa

Da bismo pokazali kako se lako može realizovati grafički ulaz i izlaz programa u Javi, u nastavku je u listingu 6.1 prikazan mali program u kojem se od korisnika najpre zahteva nekoliko ulaznih podataka, a zatim se na ekranu ispisuje poruka dobrodošlice za korisnika. Pri tome, za ove ulazne i izlazne podatke programa koriste se grafički dijalozi. Ovaj program nema svoj glavni prozor radi jednostavnosti, nego samo koristi komponentu okvira za dijalog kako bi se korisniku omogućilo da unese ulazne podatke, kao i to da se u grafičkom okruženju prikažu izlazni podaci.

Za grafičke dijaloge se koristi standardna klasa `JOptionPane` i dva njena statička metoda `showInputDialog()` i `showMessageDialog()`. Oba metoda imaju nekoliko preopterećenih verzija, a u programu se koristi ona koja u oba slučaja sadrži četiri parametra: prozor kome dijalog pripada, poruka koja se prikazuje u dijalogu, naslov dijaloga i vrsta dijaloga. Kako ovaj jednostavan program nema glavni prozor, prvi argument ovih metoda u svim slučajevima je `null`, drugi i treći argumenti su stringovi odgovarajućeg teksta, dok četvrti argument predstavlja konstantu izabranu od onih koje su takođe definisane u klasi `JOptionPane`.

Listing 6.1: Program *Zdravo* sa grafičkim dijalozima za ulaz/izlaz

```
import javax.swing.*;

public class ZdravoGUI {

   public static void main(String[] args) {

      String ime = JOptionPane.showInputDialog(null,
                        "Kako se zovete?",
                        "Grafički ulaz",
```

```
                        JOptionPane.QUESTION_MESSAGE);
        String godine = JOptionPane.showInputDialog(null,
                        "Koliko imate godina?",
                        "Grafički ulaz",
                        JOptionPane.QUESTION_MESSAGE);

        int god = Integer.parseInt(godine);

        String poruka = "Zdravo " + ime + "!\n";
        poruka += god + " su najlepše godine.";

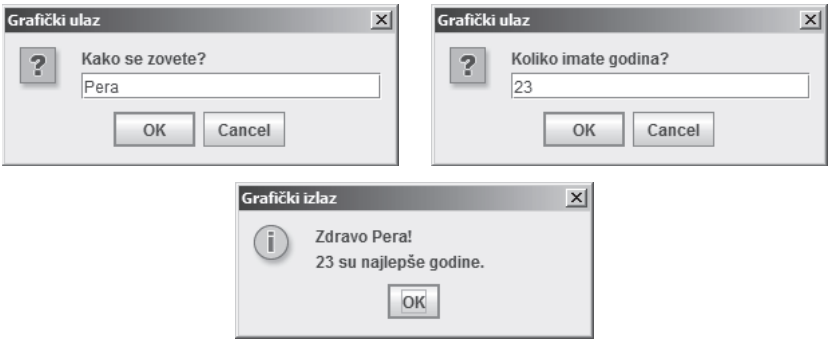
        JOptionPane.showMessageDialog(null,
                        poruka,
                        "Grafički izlaz",
                        JOptionPane.INFORMATION_MESSAGE);
        System.exit(0);
    }
}
```

Vraćena vrednost metoda `showInputDialog()` jeste string koji je korisnik uneo u polju prikazanog grafičkog dijaloga. Primetimo da je za godine korisnika potrebno taj string konvertovati u celobrojnu vrednost primenom odgovarajućeg metoda `parseInt()` omotačke klase `Integer`. Izvršavanjem programa se dobijaju tri okvira za dijalog koja su prikazana na slici 6.1.

6.2 Grafički elementi

Grafički elementi koji se u Javi mogu koristiti za pravljenje korisničkog interfejsa programa pripadaju trima glavnim kategorijama:

1. **Prozori**. To su obične provaougaone oblasti koje se mogu nacrtati na ekranu i dalje se dele u dve vrste:
   - **Okviri**. To su permanenti prozori koji su predviđeni da ostaju na ekranu za sve vreme izvršavanja programa.



Slika 6.1: Tri okvira za dijalog grafičkog programa `ZdravoGUI`.

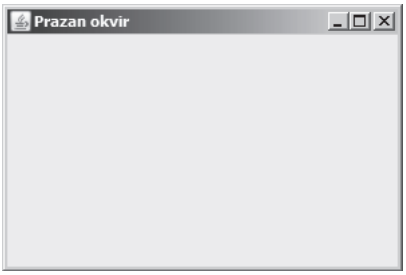
- **Dijalozi**. To su privremeni prozori koji se mogu skloniti sa ekrana tokom izvršavanja programa.

2. **Komponente**. To su vizuelni elementi koji imaju veličinu i poziciju na ekranu i koji mogu proizvoditi događaje.

3. **Kontejneri**. To su komponente koje mogu obuhvatati druge komponente.

Kontejner najvišeg nivoa u Javi se naziva *okvir* (engl. *frame*). Okvir je dakle nezavisan prozor koji se ne može nalaziti unutar drugog prozora, pa se skoro uvek koristi kao glavni prozor programa. Okvir poseduje nekoliko ugrađenih mogućnosti: može se otvoriti i zatvoriti, može mu se promeniti veličina i, najvažnije, može sadržati druge GUI komponente kao što su dugmad, meniji i tako dalje. Svaki okvir na vrhu obavezno sadrži dugme sa ikonom, polje za naslov i tri manipulativna dugmeta za minimizovanje, maksimizovanje i zatvaranje okvira.

U biblioteci Swing se nalazi klasa `JFrame` koja služi za konstruisanje i podešavanje okvira koji ima pomenute standardne mogućnosti. Okvir tipa `JFrame` na početku ne sadrži druge komponente, nego se one moraju programski dodati nakon konstruisanja okvira. Radi ilustracije, u listingu 6.2 je dat jednostavan program koji prikazuje prazan okvir na ekranu, a njegov izgled je prikazan na slici 6.2.



Slika 6.2: Prazan okvir na ekranu.

Listing 6.2: Prikaz praznog okvira

```
import javax.swing.*;

public class PrazanOkvir {

   public static void main(String[] args) {

      JFrame okvir = new JFrame("Prazan okvir");
      okvir.setSize(300, 200);
      okvir.setLocation(100, 150);
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
  }
}
```

Razmotrimo malo detaljnije naredbe metoda `main()` ovog programa za prikaz praznog okvira. Na samom početku se okvir konstruiše pozivom konstruktora klase `JFrame`:

```
JFrame okvir = new JFrame("Prazan okvir");
```

Konstruktor klase `JFrame` ima nekoliko preopterećenih verzija, a ovde je korišćen onaj koji ima parametar za naslov okvira koji se prikazuje na vrhu okvira. Svaki novokonstruisani pravougaoni okvir ima podrazumeva-

nu, prilično beskorisnu veličinu širine 0 i visine 0 piksela.[1] Zbog toga se u listingu 6.2 nakon konstruisanja okvira odmah određuje njegova veličina (300×200 piksela) i njegova pozicija na ekranu (gornji levi ugao okvira je u tački sa koordinatama 100 i 150 piksela). Objektni metodi u klasi `JFrame` za podešavanje veličine okvira i njegove pozicije na ekranu su `setSize()` i `setLocation()`:

```
okvir.setSize(300, 200);
okvir.setLocation(100, 150);
```

Naredni korak je određivanje reakcije programa kada korisnik zatvori okvir pritiskom na dugme `X` u gornjem desnom uglu okvira. U ovom primeru se prosto završava rad programa:

```
okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

U složenijem programu sa više okvira verovatno ne treba završiti njegov rad samo zbog toga što je korisnik zatvorio jedan od okvira. Inače, ukoliko se eksplicitno ne odredi šta treba dodatno uraditi kada se okvir zatvori, program se ne završava nego se samo okvir „sakriva" tako što se učini nevidljivim.

Konstruisanjem okvira se on automatski ne prikazuje na ekranu. Novi okvir se nalazi u memoriji i nevidljiv je kako bi mu se u međuvremenu mogle dodati druge komponente, pre prvog prikazivanja na ekranu. Da bi se okvir prikazao na ekranu, za njega se poziva metod `setVisible()` klase `JFrame` sa argumentom `true`:

```
okvir.setVisible(true);
```

Obratite pažnju u listingu 6.2 na to da se nakon ove naredbe završava metod `main()`, ali se time ne završava rad programa. Program nastavlja da se izvršava (tačnije, i dalje se izvršava jedna njegova nît za obradu događaja) sve dok se ne zatvori njegov glavni okvir ili se ne izvrši metod `System.exit()` usled neke greške.

Sama klasa `JFrame` ima samo nekoliko metoda za menjanje izgleda okvira. Ali u bibliotekama AWT i Swing je nasleđivanjem definisana relativno velika hijerarhija odgovarajaućih klasa tako da klasa `JFrame` nasleđuje mno-

[1] Sve grafičke dimenzije u Javi se navode u jedinicama koje označavaju pojedinačne tačke na ekranu. Ove tačke se nazivaju *pikseli*.

ge metode od svojih natklasa. Na primer, neki od značajnijih metoda za podešavanje okvira koji se mogu naći u raznim natklasama od `JFrame`, pored onih već pomenutih, jesu:

- Metod `setLocationByPlatform()` sa argumentom `true` služi da se operativnom sistemu prepusti izbor, po svom nahođenju, pozicije (ali ne i veličine) okvira na ekranu.
- Metod `setBounds()` služi za pozicioniranje okvira i određivanje njegove veličine na ekranu u jednom koraku. Na primer, dve naredbe

  ```
  okvir.setSize(300, 200);
  okvir.setLocation(100, 150);
  ```

  mogu se zameniti jednom:

  ```
  okvir.setBounds(100, 150, 300, 200);
  ```

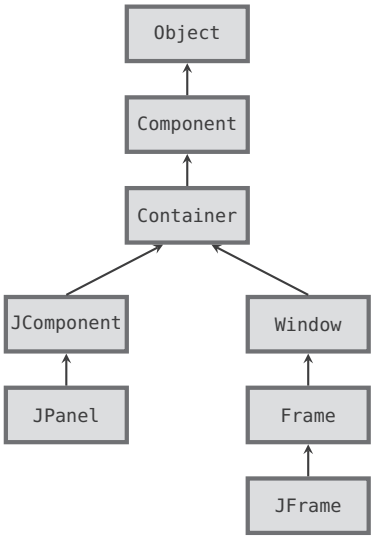
- Metod `setTitle()` služi za promenu teksta u naslovu okvira.
- Metod `setResizable()` sa argumentom logičkog tipa određuje da li se može promeniti veličina okvira.
- Metod `setIconImage()` služi za određivanje slike ikone na vrhu okvira.
- Metod `setLayout()` služi za određivanje načina na koji se komponente razmeštaju unutar okvira.

Kontejneri i komponente

Elementi na kojima se zasniva grafičko programiranje u Javi su kontejneri i komponente. Kontejneri se mogu prikazati na ekranu i služe za grupisanje komponenti, a komponente se jedino mogu prikazati unutar nekog kontejnera. Dugme je primer jedne komponente, dok je okvir primer kontejnera. Da bi se prikazalo dugme, ono se najpre mora dodati nekom okviru i zatim se taj okvir mora prikazati na ekranu.

Mogući kontejneri i komponente u Javi su naravno predstavljeni odgovarajućim klasama čija je hijerarhija definisana u bibliotekama AWT i Swing. Na slici 6.3 je prikazan mali deo ovog stabla nasleđivanja koji je relevantan za osnovne grafičke elemente. Uzgred, nalaženje nekog metoda

koji je potreban za rad sa grafičkim elementima je otežano zbog više nivoa nasleđivanja. Naime, traženi metod ne mora biti u klasi koja predstavlja odgovarajući element, već se može nalaziti u nekoj od nasleđenih klasa na višem nivou. Zato je dobro snalaženje u Java dokumentaciji od velike važnosti za grafičko programiranje.



Slika 6.3: Hijerarhija klasa AWT/Swing grafičkih elemenata.

Klase `Component` i `Container` u biblioteci AWT su apstraktne klase od kojih se grana cela hijerarhija. Primetimo da je `Container` dete od `Component`, što znači da neki kontejner jeste i komponenta. Kako se komponenta može dodati u kontejner, ovo dalje implicira da se jedan kontejner može dodati u drugi kontejner. Mogućnost ugnježđavanja kontejnera je važan aspekt pravljenja dobrog i lepog grafičkog korisničkog interfejsa.

Klasa `JComponent` u biblioteci Swing je dete klase `Container` i predak svih komponenti predstavljenih klasama `JButton`, `JLabel`, `JComboBox`, `JMenuBar` i tako dalje. Prema tome, sve Swing komponente su i kontejneri, pa se mogu međusobno ugnježđavati.

Dva osnovna kontejnera za komponente u biblioteci Swing su *okvir* i *panel* predstavljeni klasama `JFrame` i `JPanel`. Okvir je, kao što je već spomenuto, predviđen da bude prozor najvišeg nivoa. Njegova struktura je rela-

tivno složena i svi njeni detalji prevazilaze ciljeve ove knjige. Napomenimo samo da jedna od posledica specijalnog statusa okvira kao glavnog prozora jeste to što se on ne može dodavati drugim kontejnerima i komponentama, iako je klasa `JFrame` naslednica od `Container` i dakle od `Component`.

Prostija verzija kontejnera opšte namene je panel koji se može dodavati drugim kontejnerima. Panel je nevidljiv kontejner i mora se dodati kontejneru najvišeg nivoa, recimo okviru, da bi se mogao videti na ekranu. Jedan panel se može dodavati i drugom panelu, pa se onda ugnježđavanjem panela može na relativno lak način postići grupisanje komponenti i njihov željeni raspored unutar nekog kontejnera višeg nivoa. Ova primena panela kao kontejnera je najrasprostranjenija u programima, ali budući da je klasa `JPanel` naslednica klase `JComponent`, panel ujedno predstavlja i Swing komponentu, što znači da panel može služiti i za prikazivanje (crtanje) informacija.

Komponenta je vizuelni grafički objekat koji služi za prikazivanje (crtanje) informacija. Programeri mogu definisati sopstvene komponente na jednostavan način — proširivanjem klase `JComponent`. Biblioteka Swing sadrži i veliki broj gotovih komponenti koje su predstavljene odgovarajućim klasama naslednicama klase `JComponent`. Na primer, standardnim komponentama kao što su dugme, oznaka, tekstalno polje, polje za potvrdu, radio dugme i kombinovano polje odgovaraju, redom, klase `JButton`, `JLabel`, `JTextField`, `JCheckBox`, `JRadioButton` i `JComboBox`. (Ovo su samo neke od osnovnih komponenti, a potpun spisak svih komponenti u bibliotekama AWT i Swing je daleko veći.) Na slici 6.4 je prikazan izgled ovih komponenti unutar glavnog okvira programa.



Slika 6.4: Neke standardne GUI komponente iz biblioteke Swing.

U svakoj od klasa standardnih komponenti definisano je nekoliko konstruktora koji se koriste za konstruisanje objekata konkretnih komponenti.

Na primer, komponente na slici 6.4 mogu se konstruisati na sledeći način:

```
// Dugme sa tekstom "OK"
JButton dugmeOK = new JButton("OK");

// Oznaka sa tekstom "Unesite svoje ime: "
JLabel oznakaIme = new JLabel("Unesite svoje ime: ");

// Tekstualno polje sa tekstom "Ovde upišite svoje ime"
JTextField tekstPoljeIme =
               new JTextField("Ovde upišite svoje ime");

// Polje za potvrdu sa tekstom "Student"
JCheckBox potvrdaStudent = new JCheckBox("Student");

// Radio dugme sa tekstom "Zaposlen"
JRadioButton radioDugmeZaposlen =
               new JRadioButton("Zaposlen");

// Kombinovano polje sa tekstom godine studija
JComboBox kombPoljeGodina =
               new JComboBox(new String[]{"I godina",
                 "II godina", "III godina", "IV godina"});
```

Dodavanje komponente nekom kontejneru, kao i dodavanje jednog kontejnera drugom kontejneru, postiže se metodom `add()`. Ovaj metod ima više preopterećenih verzija, ali u najjednostavnijem obliku se kao njegov argument navodi komponenta ili kontejner koji se dodaju kontejneru. Na primer, ako promenljiva `panel` ukazuje na panel tipa `JPanel`, onda se standardna komponenta dugme može dodati tom panelu naredbom:

```
panel.add(new JButton("Test"));
```

Paneli se mogu nalaziti unutar okvira ili drugog panela, pa se prethodni panel sa dugmetom može dodati jednom okviru tipa `JFrame`, recimo, na koji ukazuje promenljiva `okvir`:

```
okvir.add(panel);
```

Koordinate grafičkih elemenata

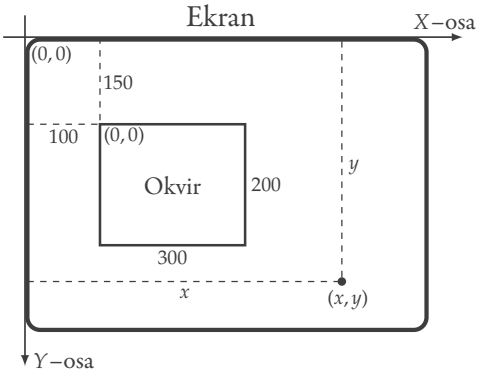
Sve komponente i kontejneri su pravougaonog oblika koji imaju veličinu (širinu i visinu) i poziciju na ekranu. Ove vrednosti se u Javi izražavaju u jedinicama koje označavaju najmanje tačke za crtanje na ekranu. Naziv za ove jedinice je *pikseli*, što je kovanica od engleskog termina *picture element*.

Ekran računara se fizički sastoji od dvodimenzionalne matrice tačaka sa određenim brojevima redova i kolona koji zavise od rezolucije ekrana. Tako, na primer, rezolucija ekrana 1024×768 označava da je ekran podeljen u 1024 reda i 768 kolona u čijim presecima se nalazi po jedna tačka (piksel) za crtanje slike. Ovakva fizička organizacija ekrana se logički predstavlja koordinatnim sistemom koji obrazuju pikseli. Koordinatni početak $(0, 0)$ se smatra da se nalazi u gornjem levom uglu ekrana. Svaka tačka na ekranu je logički dakle piksel sa koordinatama (x, y), gde je x broj piksela desno i y broj piksela dole od koordinatnog početka (gornjeg levog ugla ekrana).

Pretpostavimo da je u programu konstruisan okvir tipa `JFrame` na koji ukazuje promenljiva `okvir` i da mu je metodom `setBounds()` određena pozicija i veličina:

```
okvir.setBounds(100, 150, 300, 200);
```

Gornji levi ugao ovog okvira se nalazi u tački sa koordinatama $(100, 150)$ u odnosu na gornji levi ugao ekrana. Širina okvira je 300 piksela i visina okvira je 200 piksela. Izgled celog ekrana za ovaj primer je prikazan na slici 6.5.



Slika 6.5: Koordinatni sistem ekrana i komponenti.

Kada se komponenta doda nekom kontejneru, njena pozicija unutar kontejnera se određuje na osnovu gornjeg levog ugla kontejnera. Drugim rečima, gornji levi ugao kontejnera je koordinatni početak (0, 0) relativnog koordinatnog sistema kontejnera, a pozicije komponente se određuje u odnosu na koordinatni početak kontejnera, a ne ekrana.

U narednom primeru se najpre konstruiše dugme, zatim mu se određuju pozicija i veličina i, na kraju, dodaje se okviru:

```
JButton dugme = new JButton("OK"); // dugme sa tekstom OK
dugme.setBounds(20, 100, 60, 40);  // granice tog dugmeta
okvir.add(dugme);  // dodavanje tog dugmeta u okvir
```

Gornji levi ugao ovog dugmeta se nalazi u tački (20, 100) relativno u odnosu na gornji levi ugao okvira kome se dodaje. (Ovde se pretpostavlja da se za okvir ne koristi nijedan od unapred raspoloživih načina za razmeštanje komponenti.) Širina dugmeta je 60 piksela i njegova visina je 40 piksela.

Relativni koordinatni sistem se koristi zato što se prozori mogu pomerati na ekranu. U prethodnom primeru je gornji levi ugao okvira potencijalno promenljiv, ali relativna pozicija dugmeta unutar okvira se ne menja. Na taj način su programeri oslobođeni velike obaveze da preračunavaju relativne pozicije komponenti unutar kontejnera u njihove apsolutne koordinate na ekranu kako se prozor pomera.

Raspored komponenti unutar kontejnera

Prilikom dodavanja komponenti u kontejner, one se unutar kontejnera razmeštaju na određen način. To znači da se po unapred definisanom postupku, koji interno obavlja tzv. *layout manager*, određuje veličina i pozicija komponenti unutar kontejnera. Svakom kontejneru je pridružen podrazumevani način razmeštanja komponenti unutar njega, ali se to može promeniti ukoliko se željeni način razmeštanja navede kao argument metoda `setLayout()` klase `Container`:

```
public void setLayout(LayoutManager m)
```

Iz zaglavlja ovog metoda se vidi da je postupak za razmeštanje komponenti u Javi predstavljen objektom tipa `LayoutManager`. `LayoutManager` je interfejs koji sve klase odgovorne za razmeštanje komponenti moraju im-

plementirati. To nije mali zadatak, ali u principu programeri mogu pisati sopstvene postupke za razmeštanje komponenti. Mnogo češći slučaj je ipak da se koristi jedan od gotovih načina koji su raspoloživi u bibliotekama AWT i Swing:

- *FlowLayout*
- *GridLayout*
- *BorderLayout*
- *GridBagLayout*
- *CardLayout*
- *BoxLayout*
- *SpringLayout*
- *OverlayLayout*

U opštem slučaju dakle, komponente se smeštaju unutar kontejnera, a njegov pridruženi postupak razmeštanja određuje poziciju i veličinu komponenti u kontejneru. Detaljan opis svih raspoloživih postupaka za razmeštanje komponenti u Javi bi oduzeo mnogo prostora u ovoj knjizi, pa ćemo u nastavku više pažnje posvetiti samo onim osnovnim.

FlowLayout. Ovo je najjednostavniji postupak za razmeštanje komponenti. Komponente se smeštaju sleva na desno, kao reči u rečenici, onim redom kojim su dodate u kontejner. Svaka komponenta dobija svoju prirodnu veličinu i prenosi se u naredni red ako ne može da stane do širine kontejnera. Pri tome se može kontrolisati da li komponente treba da budu levo poravnate, desno poravnate, ili poravnate po sredini, kao i to koliko treba da bude njihovo međusobno horizontalno i vertikalno rastojanje.

Na primer, u programu u listingu 6.3 se jednom panelu dodaju šest dugmadi koja se automatski razmeštaju po *FlowLayout* postupku, jer je to podrazumevani način za razmeštanje komponenti u svakom panelu.

Listing 6.3: Postupak *FlowLayout* za razmeštanje komponenti

```
import javax.swing.*;

public class TestFlowLayout {

   public static void main(String[] args) {

      // Konstruisanje okvira
      JFrame okvir = new JFrame("Test FlowLayout");
      okvir.setSize(300, 200);
      okvir.setLocation(100, 150);
```

```
        okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Konstruisanje šest dugmadi
        JButton crvenoDugme = new JButton("Crveno");
        JButton zelenoDugme = new JButton("Zeleno");
        JButton plavoDugme = new JButton("Plavo");
        JButton narandžastoDugme = new JButton("Narandžasto");
        JButton beloDugme = new JButton("Belo");
        JButton crnoDugme = new JButton("Crno");

        // Konstruisanje panela za dugmad
        JPanel panel = new JPanel();

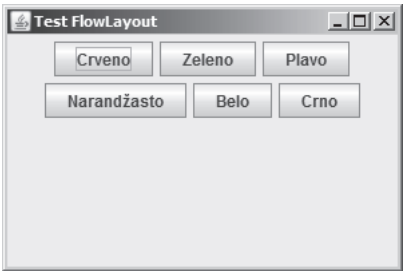
        // Smeštanje dugmadi u panel
        panel.add(crvenoDugme);
        panel.add(zelenoDugme);
        panel.add(plavoDugme);
        panel.add(narandžastoDugme);
        panel.add(beloDugme);
        panel.add(crnoDugme);

        // Smeštanje panela u okvir
        okvir.add(panel);
        okvir.setVisible(true);
    }
}
```

Na slici 6.6 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što se može videti sa te slike, komponente su poravnate po sredini i prenose se u novi red kada nema više mesta u panelu.

Pored toga, ukoliko se pomeranjem ivica okvira promeni njegova veličina, komponente u okviru se automatski razmeštaju poštujući *FlowLayout* postupak. Ova činjenica je ilustrovana na slici 6.7.

GridLayout. Ovim postupkom se kontejner deli u matricu polja po redovima i kolonama. Jedna komponenta se smešta u jedno od ovih polja



Slika 6.6: Panel sa šest dugmadi razmeštenih po *FlowLayout* postupku.



Slika 6.7: Automatsko razmeštanje dugmadi zbog promene veličine okvira.

tako da sve komponente imaju istu veličinu. Brojevi redova i kolona polja za smeštanje komponenti se određuju argumentima konstruktora klase `GridLayout`, kao i međusobno horizontalno i vertikalno rastojanje između polja. Komponente se u kontejner smeštaju sleva na desno u prvom redu, pa zatim u drugom redu i tako dalje, onim redom kojim se dodaju u kontejner. Na primer, u programu u listingu 6.4 jednom panelu se dodaju šest dugmadi koja se u njemu razmeštaju po *GridLayout* postupku.

Listing 6.4: Postupak *GridLayout* za razmeštanje komponenti

```
import javax.swing.*;
import java.awt.*;

public class TestGridLayout {

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

```
        // Konstruisanje okvira
        JFrame okvir = new JFrame("Test GridLayout");
        okvir.setSize(300, 200);
        okvir.setLocation(100, 150);
        okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Konstruisanje više dugmadi
        JButton crvenoDugme = new JButton("Crveno");
        JButton zelenoDugme = new JButton("Zeleno");
        JButton plavoDugme = new JButton("Plavo");
        JButton narandžastoDugme = new JButton("Narandžasto");
        JButton beloDugme = new JButton("Belo");
        JButton crnoDugme = new JButton("Crno");

        // Konstruisanje panela za dugmad
        JPanel panel = new JPanel();

        // Pridruživanje postupka GridLayout za razmeštanje
        // komponenti u panel u 3 reda i 2 kolone, sa horizontalnim
        // i vertikalnim rastojanjem 5 i 10 piksela između njih
        panel.setLayout(new GridLayout(3, 2, 5, 10));

        // Smeštanje dugmadi u panel
        panel.add(crvenoDugme);
        panel.add(zelenoDugme);
        panel.add(plavoDugme);
        panel.add(narandžastoDugme);
        panel.add(beloDugme);
        panel.add(crnoDugme);

        // Smeštanje panela u okvir
        okvir.add(panel);
        okvir.setVisible(true);
    }
}
```

Na slici 6.8 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što se može videti sa te slike, komponente su iste veličine i smeštene su u 3 reda i 2 kolone. Horizontalno i vertikalno rastojanje između njih je 5 i 10 piksela. Obratite pažnju na to kako su ove vrednosti navedene u pozivu metoda `setLayout()` za panel:

```
new GridLayout(3, 2, 5, 10)
```

Ovim izrazom se konstruiše objekat tipa `GridLayout`, pri čemu su željeni brojevi redova i kolona, kao i horizontalno i vertikalno rastojanje između njih, navode kao argumenti konstruktora klase `GridLayout`.



Slika 6.8: Panel sa šest dugmadi razmeštenih po *GridLayout* postupku..

U slučaju postupka *GridLayout*, ukoliko se promeni veličinu okvira, matrični poredak komponenti se ne menja, odnosno broj redova i kolona polja ostaje isti, kao i rastojanje između njih. Ova činjenica je ilustovana na slici 6.9.



Slika 6.9: Promena veličine okvira ne utiče na razmeštanje komponenti.

BorderLayout. Ovim postupkom se kontejner deli u pet polja: istočno, zapadno, severno, južno i centralno. Svaka komponenta se dodaje u jedno od ovih polja metodom `add()` koristeći dodatni argument koji predstavlja jednu od pet konstanti:

- `BorderLayout.EAST`
- `BorderLayout.WEST`
- `BorderLayout.NORTH`
- `BorderLayout.SOUTH`
- `BorderLayout.CENTER`

Komponente se smeštaju u njihovoj prirodnoj veličini u odgovarajuće polje. Pri tome, severno i južno polje se mogu automatski horizontalno raširiti, istočno i zapadno polje se mogu automatski vertikalno raširiti, dok se centralno polje može automatski raširiti u oba pravca radi popunjavanja praznog prostora. Na primer, u programu u listingu 6.5 jednom panelu se dodaju pet dugmadi koja se u njemu razmeštaju po *BorderLayout* postupku.

Listing 6.5: Postupak *BorderLayout* za razmeštanje komponenti

```
import javax.swing.*;
import java.awt.*;

public class TestBorderLayout {

   public static void main(String[] args) {

      // Konstruisanje okvira
      JFrame okvir = new JFrame("Test BorderLayout");
      okvir.setSize(300, 200);
      okvir.setLocation(100, 150);
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Konstruisanje pet dugmadi
      JButton istočnoDugme = new JButton("Istočno");
      JButton zapadnoDugme = new JButton("Zapadno");
      JButton severnoDugme = new JButton("Severno");
      JButton južnoDugme = new JButton("Južno");
      JButton centralnoDugme = new JButton("Centralno");
```

```
        // Konstruisanje panela za dugmad
        JPanel panel = new JPanel();

        // Pridruživanje postupka BorderLayout za razmeštanje
        // komponenti u panel, sa horizontalnim i vertikalnim
        // rastojanjem 5 i 10 piksela između njih
        panel.setLayout(new BorderLayout(5, 10));

        // Smeštanje dugmadi u panel
        panel.add(istočnoDugme, BorderLayout.EAST);
        panel.add(zapadnoDugme, BorderLayout.WEST);
        panel.add(severnoDugme, BorderLayout.NORTH);
        panel.add(južnoDugme, BorderLayout.SOUTH);
        panel.add(centralnoDugme, BorderLayout.CENTER);

        // Smeštanje panela u okvir
        okvir.add(panel);
        okvir.setVisible(true);
    }
}
```

Na slici 6.10 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što se može videti sa te slike, severno i južno dugme se automatski horizontalno prostiru celom širinom okvira, zapadno i istočno dugme se automatski vertikalno prostiru po raspoloživoj visini, dok je centralno dugme automatski prošireno u oba pravca.

Kod postupka *BorderLayout* nije neophodno da se svako polje popuni nekom komponentom. U slučaju da neko polje nije popunjeno, ostala polja će automatski zauzeti njegovo mesto prema tome u kom pravcu mogu da se proširuju. Na primer, ako se u prethodnom programu ukloni istočno dugme, onda će se centralno dugme proširiti udesno da bi se zauzeo slobodan prostor. Isto tako, ukoliko se promeni veličina okvira, komponente se automatski proširuju u dozvoljenom pravcu. Ova činjenica je ilustrovana na slici 6.11.



Slika 6.10: Panel sa pet dugmadi razmeštenih po *BorderLayout* postupku.



Slika 6.11: Promena veličine okvira izaziva proširivanje komponenti u dozvoljenom pravcu.

6.3 Definisanje grafičkih komponenti

Ukoliko je potrebno prikazati informacije na poseban način koji se ne može postići unapred definisanim komponentama, moraju se definisati nove grafičke komponente. To se postiže definisanjem klase nove komponente tako da bude naslednica klase `JComponent`. Pri tome se mora nadjačati metod `paintComponent()` klase `JComponent`, a u nadjačanoj verziji ovog metoda se mora definisati šta se crta u novoj komponenti. Prema tome, opšta šema definicije klase nove komponente je:

```
public class GrafičkaKomponenta extends JComponent {
    .
    . // Ostala polja i metodi
```

```
        .
        public void paintComponent(Graphics g) {
            .
            . // Naredbe za crtanje u komponenti
            .
        }
    }
```

U opštem slučaju, svaka grafička komponenta je odgovorna za sopstveno crtanje koje se izvodi u metodu `paintComponent()`. To važi kako za standardne, tako i za nove komponente koje se posebno definišu. Doduše, standardne komponente se samo dodaju odgovarajućem kontejneru, jer je za njih u biblioteci AWT/Swing već definisano šta se crta na ekranu.

Obratite pažnju na to da se prikazivanje informacija u nekoj komponenti izvodi jedino crtanjem u toj komponenti. (Čak i prikazivanje teksta u komponenti izvodi se njegovim crtanjem.) Ovo crtanje se obavlja u komponentinom metodu `paintComponent()` koji ima jedan parametar tipa `Graphics`. U stvari, kompletno crtanje u ovom metodu mora se izvoditi samo preko tog njegovog parametra tipa `Graphics`.

Klasa `Graphics` je apstraktna klasa koja u programu obezbeđuje grafičke mogućnosti nezavisno od fizičkog grafičkog uređaja računara. Svaki put kada se neka komponenta (standardna ili programerski definisana) prikaže na ekranu, JVM automatski konstruiše objekat tipa `Graphics` koji predstavlja grafički kontekst za tu komponentu na konkretnoj platformi. Da bi se bolje razumela svrhu klase `Graphics`, dobra analogija je ukoliko se jedna grafička komponenta uporedi sa listom papira, a odgovarajući objekat tipa `Graphics` sa olovkom ili četkicom: metodi klase `Graphics` (mogućnosti olovke ili četkice) mogu se koristiti za crtanje u komponenti (na listu papira).

Klasa `Graphics` sadrži objektne metode za crtanje stringova, linija, pravougaonika, ovala, lukova, poligona i poligonalnih linija. Na primer, za crtanje stringa (tj. prikazivanje teksta) služi metod

```
public void drawString(String s, int x, int y)
```

gde su `x` i `y` (relativne) koordinate mesta početka stringa `s` u komponenti. Slično, za crtanje pravougaonika služi metod

```
public void drawRect(int x, int y, int w, int h)
```

gde su x i y (relativne) koordinate gornjeg levog ugla pravougaonika, a w i h njegova širina i visina. (Detalje o ostalim metodima čitaoci mogu potražiti u Java dokumentaciji.) Radi ilustracije, u listingu 6.6 je prikazan program kojim se crta prigodan tekst i pravougaonik unutar okvira. Dobijeni rezultat izvršavanja ovog programa je prikazan na slici 6.12.

Listing 6.6: Test klase `Graphics`

```
import javax.swing.*;
import java.awt.*;

public class TestGraphics {

   public static void main(String[] args) {

      // Konstruisanje okvira
      JFrame okvir = new JFrame("Test Graphics");
      okvir.setSize(300, 200);
      okvir.setLocation(100, 150);
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Konstruisanje komponente
      GrafičkaKomponenta komp = new GrafičkaKomponenta();

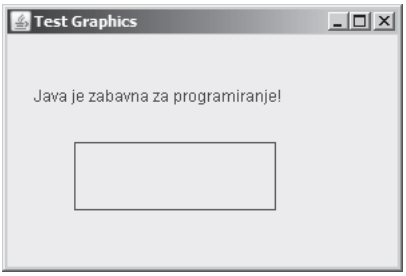
      // Smeštanje komponente u okvir
      okvir.add(komp);
      okvir.setVisible(true);
  }
}

class GrafičkaKomponenta extends JComponent {

   public void paintComponent(Graphics g) {

      g.drawString("Java je zabavna za programiranje!", 20, 50);
      g.drawRect(50, 80, 150, 50);
```

```
    }
}
```



Slika 6.12: Rezultat izvršavanja programa u listingu 6.6.

Primetimo da se program sastoji od glavne klase `TestGraphics` i klase nove komponente `GrafičkaKomponenta`. Definicija klase `GrafičkaKomponenta` ne sadrži nijedan specifikator pristupa kako bi se obe klase u ovom jednostavnom programu mogle nalaziti u istoj datoteci. Naime, ukoliko datoteka sa Java kodom sadrži više klasa, onda samo jedna od njih može biti `public`.

Obratite posebno pažnju na to da se u programu nigde eksplicitno ne poziva metod `paintComponent()` klase `GrafičkaKomponenta`. U stvari, ovaj metod nikad ne treba pozivati direktno, jer se on automatski poziva kada treba prikazati glavni okvir programa. Preciznije, svaki put kada se prikazuje glavni okvir programa, bez obzira na razlog, JVM implicitno poziva za izvršavanje metode `paintComponent()` svih komponenti u tom okviru.

Koji su mogući razlozi za (ponovno) prikazivanje okvira programa? Naravno, glavni razlog je kada se okvir prikazuje po prvi put nakon pokretanja programa. Ali i druge akcije korisnika izazivaju automatsko crtanje komponenti pozivom metoda `paintComponent()`. Na primer, minimizovanjem pa otvaranjem okvira programa, njegov sadržaj se mora ponovo nacrtati. Ili, ako se otvaranjem drugog prozora prekrije (delimično ili potpuno) postojeći okvir, pa se ovaj okvir ponovo izabere u fokusu, tada se sadržaj postojećeg okvira mora opet nacrtati.

U klasi `JComponent` je dakle definisan metod

```
protected void paintComponent(Graphics g)
```

koji svaka nova komponenta treba da nadjača da bi se obezbedilo njeno crtanje. JVM automatski poziva ovu nadjačanu verziju svaki put kada komponentu treba (ponovno) prikazati. Pri tome, JVM automatski konstruiše i propisno inicijalizuje objekat `g` tipa `Graphics` za svaku vidljivu komponentu i prenosi ga metodu `paintComponent()` radi crtanja u komponenti, nezavisno od konkretnog grafičkog uređaja na kome se fizički izvodi crtanje.

Pošto metod `paintComponent()` ne treba direktno pozivati za crtanje u komponenti, šta uraditi u slučaju kada je neophodno hitno prikazati sadržaj komponente negde u sredini nekog drugog metoda? Prikazivanje komponente se može zahtevati metodom

```
public void repaint()
```

koji je definisan u klasi `Component`, pa ga može koristiti svaka komponenta. U stvari, pozivom ovog metoda se ne izvršava neposredno crtanje komponente, nego se samo obaveštava JVM da je potrebno ponovo nacrtati komponentu. JVM će ovaj zahtev registrovati i pozvati metod `paintComponent()` komponente što pre može, odnosno čim obradi eventualne prethodne zahteve.

6.4 Klase za crtanje

Osnovna klasa `Graphics` za crtanje u Javi postoji od početne verzije jezika. Ona sadrži metode za crtanje teksta, linija, provougaonika i drugih geometrijskih oblika. Ipak te grafičke operacije su prilično ograničene jer, na primer, nije moguće menjati debljinu linija ili rotirati geometrijske oblike.

Zbog toga je od verzije Java 1.2 dodata klasa `Graphics2D` sa mnogo većim grafičkim mogućnostima. Uz to su dodate i druge prateće klase, pa se cela kolekcija dodatih klasa jednim imenom naziva biblioteka Java 2D. U ovom odeljku se govori samo o osnovnim svojstvima biblioteke Java 2D, a više informacije o njoj čitaoci mogu potražiti u dodatnoj literaturi.

Klasa `Graphics2D` nasleđuje klasu `Graphics`, što znači da se svi postojeći metodi klase `Graphics` mogu primeniti i na objekte klase `Graphics2D`. Dodatno, metodu `paintComponent()` se od verzije Java 1.2 prenosi zapravo objekat stvarnog tipa `Graphics2D`, a ne tipa `Graphics`. To je neophodno kako bi se u ovom metodu mogle koristiti složenije grafičke operacije definisane u klasi `Graphics2D`.

Primetimo da se ovim ne narušava definicija metoda `paintComponent()` u klasi `JComponent`, jer na osnovu principa podtipa za nasleđivanje, parametar tipa `Graphics` ovog metoda može ukazivati na objekat podtipa `Graphics2D`. Međutim, za korišćenje svih grafičkih mogućnosti klase `Graphics2D` u metodu `paintComponent()`, mora se izvršiti ekplicitna konverzija tipa njegovog parametra na uobičajeni način:

```
public void paintComponent(Graphics g) {

   Graphics2D g2 = (Graphics2D) g;
   .
   . // Koristiti metode za crtanje objekta g2, a ne g
   .
}
```

Crtanje u biblioteci Java 2D se generalno zasniva na geometrijskim objektima klasa iz paketa `java.awt.geom` koje implementiraju interfejs `Shape`. Među ovim klasama se nalaze sledeće apstraktne klase:

- `Line2D` za linije;
- `Rectangle2D` za pravougaonike;
- `Ellipse2D` za elipse i krugove;
- `Arc2D` za lukove; i
- `CubicCurve2D` za takozvane Bezijeove krive.

U klasi `Graphics2D`, između ostalog, definisani su i metodi

```
void draw(Shape s)
void fill(Shape s)
```

koji služe za crtanje konture geometrijskog oblika `s` i za popunjavanje njegove unutrašnjosti koristeći aktuelna svojstva grafičkog konteksta tipa `Graphics2D`.

Jedno poboljšanje koje donosi biblioteka Java 2D su tačnije jedinice u kojima se navode koordinate grafičkih elemenata. Koordinate na stari način se predstavljaju celobrojnim vrednostima za označavanje piksela na ekranu. Nasuprot tome, u biblioteci Java 2D se koriste realni brojevi za koordinate koji ne moraju označavati piksele, već uobičajene dužinske jedinice (na primer, milimetre ili inče).

U internim izračunavanjima biblioteke Java 2D koriste se realni brojevi obične preciznosti tipa `float`. Ta preciznost je sasvim dovoljna za označavanje piksela na ekranu ili papiru. Pored toga, izračunavanja sa `float` vrednostima se brže izvode nego sa `double` vrednostima, a i `float` vrednosti zauzimaju duplo manje memorije od `double` vrednosti. Međutim, u Javi se podrazumeva da su realni literali tipa `double`, a ne tipa `float`, pa se mora dodavati slovo `F` na kraju konstante tipa `float`. Sledeće naredba dodele, na primer,

```
float x = 2.34;   // GREŠKA!
```

pogrešna je zato što se podrazumeva da je realni broj 2.34 tipa `double`. Ova naredba ispravno napisana je:

```
float x = 2.34F;  // OK
```

Pored toga, u radu sa nekim grafičkim operacijama je potrebno koristiti eksplicitnu konverziju iz tipa `double` u tip `float`. Sve ovo je razlog zašto se u biblioteci Java 2D mogu naći dve verzije za svaki geometrijski element: jedna verzija koristi koordinate tipa `float` i druga verzija koristi koordinate tipa `double`.

Razmotrimo primer geometrijskog oblika pravougaonika koji je predstavljen klasom `Rectangle2D`. Ovo je apstraktna klasa koju nasleđuju dve konkretne klase:[2]

- `Rectangle2D.Float`
- `Rectangle2D.Double`

Kada se konstruiše pravougaonik tipa `Rectangle2D.Float` navode se koordinate njegovog gornjeg levog ugla i njegova širina i visina kao vrednosti tipa `float`. Za pravougaonik tipa `Rectangle2D.Double` ove veličine se izražavaju kao vrednosti tipa `double`:

[2] To su zapravo statičke ugnježđene klase unutar apstraktne klase `Rectangle2D` da se ne bi koristila imena kao što su recimo `FloatRectangle2D` i `DoubleRectangle2D`.

```
Rectangle2D.Float p1 =
      new Rectangle2D.Float(50.0F, 100.0F, 22.5F, 45.5F);
Rectangle2D.Double p2 =
      new Rectangle2D.Double(50.0, 100.0, 22.5, 45.5);
```

U stvari, pošto klase `Rectangle2D.Float` i `Rectangle2D.Double` nasleđuju zajedničku klasu `Rectangle2D` i metodi u ove dve klase nadjačavaju metode u klasi `Rectangle2D`, nema potrebe pamtiti tačan tip pravougaonika. Naime, na osnovu principa podtipa za nasleđivanje, promenljiva klasnog tipa `Rectangle2D` može ukazivati na pravougaonike oba tipa:

```
Rectangle2D p1 =
      new Rectangle2D.Float(50.0F, 100.0F, 22.5F, 45.5F);
Rectangle2D p2 =
      new Rectangle2D.Double(50.0, 100.0, 22.5, 45.5);
```

Klase `Rectangle2D.Float` i `Rectangle2D.Double` se dakle moraju koristiti samo za konstruisanje pravougaonika, dok se za dalji rad sa pravougaonicima može svuda koristiti promenljiva zajedničkog nadtipa `Rectangle2D`.

Ovo što se odnosi za pravougaonike važi potpuno isto i za ostale geometrijske oblike u biblioteci Java 2D. Pored toga, umesto korišćenja odvojenih x i y koordinata, tačke koordinatnog sistema se mogu predstaviti na više objektno orijentisan način pomoću klase `Point2D`. Zato mnogi konstruktori i metodi klasa u biblioteci Java 2D imaju parametre tipa `Point2D`, jer je obično prirodnije u geometrijskim izračunavanjima koristiti tačke klase `Point2D`.

Konstruisanje tačaka tipa `Point2D` izvodi se slično kao i za druge geometrijske oblike: dve klase `Point2D.Float` i `Point2D.Double` su naslednice klase `Point2D` i služe za konstruisanje tačaka sa (x, y) koordinatama tipa `float` i `double`:

```
Point2D t1 = new Point2D.Float(10F, 20F);
Point2D t2 = new Point2D.Double(10, 20);
```

Boje

Prilikom crtanja se obično koriste razne boje radi postizanja određenog vizuelnog efekta. U Javi se mogu koristiti dva načina za predstavljanje boja:

RGB (skraćenica od engl. *red, green, blue*) i HSB (skraćenica od engl. *hue, saturation, brightness*). Podrazumevani model je RGB u kojem se svaka boja predstavlja pomoću tri broja iz intervala 0–255 koji određuju stepen crvene, zelene i plave boje u nekoj boji.

Boja u Javi je objekat klase `Color` iz paketa `java.awt` i može se konstruisati navođenjem njene crvene, zelene i plave komponente, na primer:

```
Color nekaBoja = new Color(r,g,b);
```

gde su `r`, `g` i `b` brojevi (tipa `int` ili `float`) iz intervala 0–255. Nove boje se često ne moraju posebno konstruisati, jer su u klasi `Color` definisane statičke konstante čije vrednosti predstavljaju uobičajene boje:

- `Color.WHITE`
- `Color.BLACK`
- `Color.RED`
- `Color.GREEN`
- `Color.BLUE`
- `Color.CYAN`
- `Color.MAGENTA`
- `Color.YELLOW`
- `Color.PINK`
- `Color.ORANGE`
- `Color.LIGHT_GRAY`
- `Color.GRAY`
- `Color.DARK_GRAY`

Alternativni model boja je HSB u kojem se svaka boja predstavlja pomoću tri broja za njenu nijansu (ton), zasićenost i sjajnost. Nijansa (ton) je osnovna boja koja pripada duginom spektru boja. Potpuno zasićena boja je čista boja, dok se smanjivanjem zasićenosti dobijaju prelivi kao da se čista boja meša sa belom bojom. Najzad, sjajnost određuje stepen osvetljenosti boje. U Javi se ova tri HSB elementa svake boje predstavljaju realnim brojevima tipa `float` iz intervala 0.0F–1.0F. (Podsetimo se da se literali tipa `float` pišu sa slovom F na kraju da bi se razlikovali od realnih brojeva tipa `double`.) U klasi `Color` je definisan statički metod `getHSBColor()` koji služi za konstruisanje HSB boje, na primer:

```
Color nekaBoja = Color.getHSBColor(h,s,b);
```

Između modela RGB i HSB nema bitne razlike. Oni su samo dva različita načina za predstavljanje istog skupa boja, pa se u programima obično koristi podrazumevani RGB model.

Jedno od svojstava objekta tipa `Graphics2D` (ili `Graphics`) za crtanje u komponenti jeste aktuelna boja u kojoj se izvodi crtanje. Ako je `g2` objekat tipa `Graphics2D` koji predstavlja grafički kontekst komponente, onda se njegova aktuelna boja za crtanje može izabrati metodom `setPaint()`:

```
g2.setPaint(c);
```

Argument c ovog metoda je objekat boje tipa `Color`. Ako treba crtati, recimo, crvenom bojom u nekoj komponenti, onda treba pisati

```
g2.setPaint(Color.RED);
```

pre naredbi za crtanje u metodu `paintComponent()` te komponente. Na primer:

```
g2.setPaint(Color.RED);
g2.drawString("Ceo disk će biti obrisan!", 50, 100);
```

U grafičkom kontekstu komponente se koristi ista boja za crtanje sve dok se ona eksplicitno ne promeni metodom `setPaint()`. Ako se želi dobiti aktuelna boja grafičkog konteksta komponente, koristi se metod `getPaint()`. Rezultat ovog metoda je objekat tipa `Paint`, a `Paint` je interfejs koga implementira klasa `Color`:

```
Color aktuelnaBoja = (Color) g2.getPaint();
```

Metod `getPaint()` može biti koristan u slučajevima kada je potrebno privremeno promeniti boju za crtanje, a zatim se vratiti na prvobitnu boju:

```
Color staraBoja = (Color) g2.getPaint();
g2.setPaint(new Color(0, 128, 128)); // zeleno-plava boja
. . .
g2.setPaint(staraBoja);
```

Sve komponente imaju pridružene dve boje za pozadinu i lice („prednju stranu"). Generalno, pre bilo kakvog crtanja, cela komponenta se boji bojom njene pozadine. Doduše ovo važi samo za neprozirne (engl. *opaque*) komponente, jer postoje i prozirne (engl. *transparent*) komponente kod kojih se boja pozadine ne koristi. Za menjanje boje pozadine komponente služi metod `setBackground()` koji je definisan u klasi `Component`:

```
NovaKomponenta k = new NovaKomponenta();
k.setBackground(Color.PINK);
```

Metodom `setBackground()` se, u stvari, samo određuje svojstvo boje pozadine komponente, odnosno time se ne postiže automatsko crtanje njene boje pozadine na ekranu. Da bi se boja pozadine komponente zaista promenila na ekranu, to se mora obezbediti u samoj komponenti u njenoj

nadjačanoj verziji metoda `paintComponent()`, na primer:

```
public void paintComponent(Graphics g) {

   Graphics2D g2 = (Graphics2D) g;

   if (isOpaque()) {  // komponenta neprozirna?
      g2.setPaint(getBackground());           // boja pozadine
      g2.fillRect(0,0,getWidth(),getHeight()); // obojiti pozadinu
      // Za preciznije bojenje pravougaonika komponente:
      // g2.fill(new
      //    Rectangle2D.Double(0,0,getWidth(),getHeight()));
      g2.setPaint(getForeground());
   }
   .
   . // Ostale naredbe za crtanje u komponenti
   .
}
```

Boja lica komponente slično se može izabrati metodom `setForeground()` iz klase `Component`. Ovim metodom se određuje podrazumevana boja za crtanje u komponenti, jer kada se konstruiše grafički kontekst komponente, njegova aktuelna boja za crtanje dobija vrednost boje lica komponente. Obratite ipak pažnju na to da su boje pozadine i lica zapravo svojstva komponente, a ne njenog grafičkog konteksta.

Fontovi

Font predstavlja izgled znakova teksta određenog pisma — jedan isti znak obično izgleda drugačije u različitim fontovima. Raspoloživi fontovi zavise od konkretnog računara, jer neki dolaze sa operativnim sistemom, dok su drugi komercijalni i moraju se kupiti.

U Javi, font je određen imenom, stilom i veličinom. Pošto lista imena stvarnih fontova zavisi od računara do računara, u biblioteci AWT/Swing je definisano pet *logičkih* imena fontova:

- Serif
- Dialog
- SansSerif
- DialogInput
- Monospaced

Nazivom „Serif" označava se font u kojem znakovi imaju crtice (serife) na svojim krajevima koje pomažu očima da se lakše prati tekst. Na primer, glavni tekst ove knjige je pisan takvim fontom, što se može zaključiti po, recimo, horizontalnim crticama na dnu slova n. Nazivom „SansSerif" označava se font u kojem znakovi nemaju ove crtice. „Monospaced" označava font u kojem svi znakovi imaju jednaku širinu. (Programski tekstovi u ovoj knjizi su na primer pisani ovakvim fontom.) „Dialog" i „DialogInput" su fontovi koji se obično koriste u okvirima za dijalog.

Logička imena fontova se uvek mapiraju na fontove koji zaista postoje na računaru. U Windows sistemu, recimo, fontu "SansSerif" odgovara stvarni font Arial.

Stil fonta se određuje korišćenjem statičkih konstanti koje su definisane u klasi `Font`:

- `Font.PLAIN`
- `Font.ITALIC`
- `Font.BOLD`
- `Font.BOLD + Font.ITALIC`

Najzad, veličina fonta je ceo broj, obično od 10 do 36, koji otprilike predstavlja visinu najvećih znakova fonta. Jedinice u kojima se meri veličina fonta su takozvane tipografske tačke: jedan inč (2,54 cm) ima 72 tačke. Podrazumevana veličina fonta za prikazivanje (crtanje) teksta je 12.

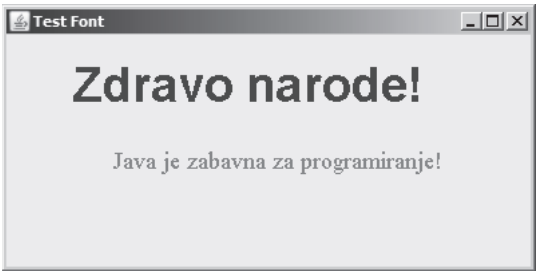
Da bi se neki tekst prikazao u određenom fontu, mora se najpre konstruisati odgovarajući objekat klase `Font` iz paketa `java.awt`. Novi font se konstruiše navođenjem njegovog imena, stila i veličine u konstruktoru:

```
Font običanFont = new Font("Serif", Font.PLAIN, 12);
Font sansBold24 = new Font("SansSerif", Font.BOLD, 24);
```

Grafički kontekst svake komponente ima pridružen font koji se koristi za prikazivanje (crtanje) teksta. Aktuelni font grafičkog konteksta se može menjati metodom `setFont()`. Obrnuto, vrednost aktuelnog fonta se može dobiti metodom `getFont()` koji vraća objekat tipa `Font`. Na primer, u sledećoj komponenti se prikazuje prigodan tekst čiji je izgled pokazan na slici 6.13:

```
public class TestFontKomponenta extends JComponent {

   public void paintComponent(Graphics g) {
```



Slika 6.13: Primer teksta u različitim fontovima.

```
        Graphics2D g2 = (Graphics2D) g;

        Font sansBold36 = new Font("SansSerif", Font.BOLD, 36);
        g2.setFont(sansBold36);    // veliki font za tekst
        g2.drawString("Zdravo narode!", 50, 50);
        Font serifPlain18 = new Font("Serif", Font.PLAIN, 18);
        g2.setFont(serifPlain18);  // manji font za tekst
        g2.setPaint(Color.RED);    // crvena boja za crtanje
        g2.drawString("Java je zabavna za programiranje!", 80, 100);
    }
}
```

Svaka komponenta ima takođe svoj pridružen font koji se može zadati objektnim metodom `setFont()` iz klase `Component`. Kada se konstruiše grafički kontekst za crtanje u komponenti, njegov pridružen font dobija vrednost aktuelnog fonta komponente.

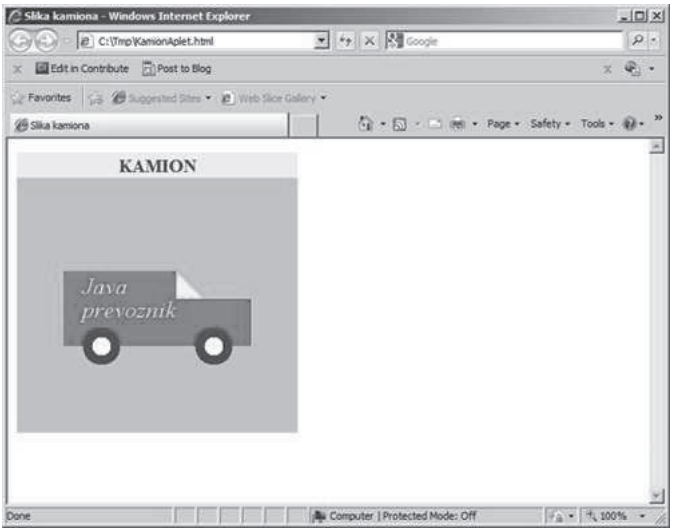
Primer: aplet za crtanje kamiona

Aplet je grafički Java program koji se izvršava u veb čitaču (Explorer, Mozilla, Chrome, ...) prilikom prikazivanja neke veb strane. Naime, pored instrukcija za prikaz teksta, slika i drugog sadržaja, veb strane mogu imati instrukcije za izvršavanje specijalnog Java programa koji se popularno naziva aplet. Izvršavanje apleta unutar veb strane je omogućeno time što svi moderni veb čitači sadrže u sebi Java interpretator (JVM) pod čijom se kontrolom izvršava aplet.

Pisanje apleta nije mnogo drugačije od pisanja običnih grafičkih programa u Javi. Naime, struktura jednog apleta je suštinski ista kao struktura okvira klase `JFrame`, a i sa događajima se kod obe vrste programa postupa na isti način. Praktično postoje samo dve glavne razlike između apleta i običnog programa. Jedna razlika je posledica činjenice da aplet zavisi od izgleda veb strane, pa se njegove pravougaone dimenzije ne mogu programski odrediti, nego se zadaju instrukcijama za opis veb strane koja sadrži aplet.

Druga, bitnija razlika je to što se aplet predstavlja klasom koja nasleđuje klasu `JApplet`. U klasi `JApplet` je definisano nekoliko objektnih metoda koji su jedinstveni za aplete. Najvažniji od njih je metod `init()` koji se početno poziva prilikom izvršavanja apleta od strane brauzera. Metod `init()` kod apleta odgovara u neku ruku metodu `main()` kod običnih programa. Metod `init()` služi dakle za inicijalizaciju vizuelne strukture apleta (kao i postupka rukovanja sa događajima) radi obezbeđivanja željene funkcionalnosti apleta.

Da bismo ilustrovali način na koji se pišu apleti u Javi, u listingu 6.7 u nastavku je dat primer apleta za crtanje kamiona. Izgled veb strane sa ovim apletom je prikazan na slici 6.14.



Slika 6.14: Veb strana sa apletom za crtanje kamiona.

Slika kamiona u apletu se crta u klasi `Kamion` koja predstavlja uobičajenu grafičku komponentu običnog programa. Sam aplet je predstavljen klasom `KamionAplet` u kojoj je nadjačan metod `init()` klase `JApplet`. Metod `init()` klase `KamionAplet` sadrži standardne naredbe kojima se apletu dodaju željene grafičke komponente. To su u ovom primeru naslov apleta kao instanca klase `JLabel` i slika kamiona kao instanca klase `Kamion`.

Listing 6.7: Aplet za crtanje kamiona

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class KamionAplet extends JApplet {

   public void init() {

      setLayout(new BorderLayout());

      JLabel naslov = new JLabel("KAMION", SwingConstants.CENTER);
      naslov.setFont(new Font("Serif", Font.BOLD, 20));
      add(naslov, BorderLayout.NORTH);

      JComponent fap = new Kamion();
      add(fap, BorderLayout.CENTER);
   }
}

class Kamion extends JComponent {

   public void paintComponent(Graphics g) {

      Graphics2D g2 = (Graphics2D) g;

      // Bojenje pozadine
      Color bledoPlava = new Color(0.75f, 0.750f, 1.0f);
      g2.setColor(bledoPlava);
```

```
        g2.fill(new Rectangle2D.Double(0,0,300,300));

        // Crtanje šasije kamiona
        g2.setColor(Color.RED);
        g2.fill(new Rectangle2D.Double(50,100,120,80));
        g2.fill(new Rectangle2D.Double(170,130,80,50));

        // Crtanje kabine kamiona
        Polygon trougao = new Polygon();
        trougao.addPoint(170,100);
        trougao.addPoint(170,130);
        trougao.addPoint(200,130);
        g2.setColor(Color.YELLOW);
        g2.fillPolygon(trougao);

        // Crtanje zadnjeg točka
        g2.setColor(Color.DARK_GRAY);
        g2.fill(new Ellipse2D.Double(70,160,40,40));
        g2.setColor(Color.WHITE);
        g2.fill(new Ellipse2D.Double(80,170,20,20));

        // Crtanje prednjeg točka
        g2.setColor(Color.DARK_GRAY);
        g2.fill(new Ellipse2D.Double(190,160,40,40));
        g2.setColor(Color.WHITE);
        g2.fill(new Ellipse2D.Double(200,170,20,20));

        // Crtanje logoa na stranici kamiona
        g2.setFont(new Font("Serif", Font.ITALIC, 25));
        g2.setColor(Color.WHITE);
        g2.drawString("Java",70,125);
        g2.drawString("prevoznik",70,150);
    }
}
```

Izvršavanje apleta se odvija unutar veb čitača, pa pored pisanja samog apleta, u odgovarajućoj veb strani je potrebno navesti instrukcije za izvrša-

vanje apleta. Ove instrukcije su deo standardnog HTML jezika koji služi za opis izgleda bilo koje veb strane. Na primer, opisu veb strane sa apletom za crtanje kamiona potrebno je na željenom mestu dodati takozvani `<applet>` tag:

```
<applet code = "KamionAplet.class" width=300 height=300>
</applet>
```

U ovom primeru su navedene minimalne informacije koje su potrebne za izvršavanje apleta, jer širi opis `<applet>` taga i HTML jezika svakako prevazilazi okvire ove knjige. Za početno upoznavanje sa apletima je zato dovoljno znati da se prethodnim primerom `<applet>` taga u prozoru za prikaz cele veb strane zauzima na odgovarajućoj poziciji okvir veličine 300×300 piksela. Pri tome, kada veb čitač prikazuje tu veb stranu, u predviđenom okviru se prikazuje rezultat izvršavanja Java bajtkoda koji se nalazi u datoteci `KamionAplet.class`.

6.5 Rukovanje događajima

Programiranje grafičkih programa se zasniva na događajima. Grafički program nema metod `main()` od kojeg počinje izvršavanje programa redom od prve naredbe do kraja. Grafički program umesto toga reaguje na različite vrste događaja koji se dešavaju u nepredvidljivim trenucima i po redosledu nad kojim program nema kontrolu.

Događaje direktno generiše korisnik svojim postupcima kao što su pritisak na taster tastature ili miša, pomeranje miša i slično. Ovi događeji se prenose grafičkoj komponenti u prozoru programa u kojoj se desio događaj tako da se sa gledišta programa može smatrati i da komponente indirektno generišu događaje. Standardna komponenta dugme recimo generiše događaj svaki put kada se pritisne na dugme. Svaki generisan događaj ne mora izazvati reakciju programa — program može izabrati da ignoriše neke događaje.

Događaju su u Javi predstavljeni objektima koji su instance klasa naslednica početne klase `EventObject` iz paketa `java.util`. Kada se desi neki događaj, JVM konstruiše jedan objekat koji grupiše relevantne informacije

o događaju. Različiti tipovi događaja su predstavljeni objektima koji pripadaju različitim klasama. Na primer, kada korisnik pritisne jedan od tastera miša, konstruiše se objekat klase pod imenom `MouseEvent`. Taj objekat sadrži relavantne informacije kao što su izvor događaja (tj. komponenta u kojoj se nalazi pokazivač miša u trenutku pritiska na taster miša), koordinate tačke komponente u kojoj se nalazi pokazivač miša i taster na mišu koji je pritisnut. Slično, kada se klikne na standardnu komponentu dugme, generiše se *akcijski* događaj i konstruiše odgovarajući objekat klase `ActionEvent`. Klase `MouseEvent` i `ActionEvent` (i mnoge druge) pripadaju hijerarhji klasa na čijem vrhu se nalazi klasa `EventObject`.

Nakon konstruisanja objekta koji opisuje nastali događaj, taj objekat se kao argument prenosi posebno predviđenom metodu za obradu određenog tipa događaja. Programer utiče na obradu događaja pisanjem ovog metoda kojim se definiše šta treba uraditi kada se događaj desi.

Objektno orijentisan model rukovanja događajima u Javi se zasniva na dva glavna koncepta:

1. **Izvor događaja (engl. *event source*).** Svaka grafička komponenta u kojoj se desio događaj je izvor događaja. Na primer, dugme koje je pritisnuto je izvor akcijskog događaja tipa `ActionEvent`. Izvorni objekat nekog događaja u programu se može dobiti pomoću metoda `getSource()`. Ovaj objektni metod se nalazi u klasi `EventObject` od koje počinje cela hijerarhija klasa događaja. Na primer, ako je `e` događaj tipa `EventObject` (ili nekog specifičnijeg podtipa kao što je `ActionEvent`), onda se referenca na objekat u kojem je nastao taj događaj može dobiti pomoću `e.getSource()`.

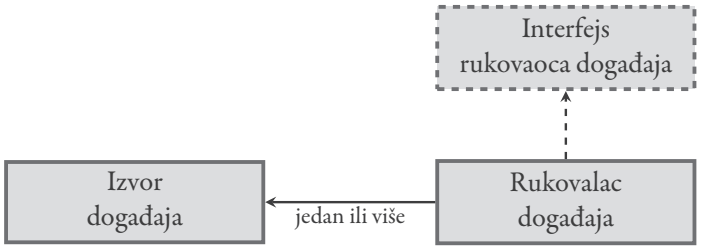
2. **Rukovalac događaja (engl. *event listener*).** Rukovalac događaja je objekat koji sadrži poseban metod za obradu onog tipa događaja koji može da se desi u izvoru događaja. Da bi se taj metod u rukovaocu događaja zaista pozvao za obradu nastalog događaja, rukovalac događaja mora zadovoljiti dva uslova:

   a) Rukovalac događaja mora biti objekat klase koja implementira specijalni interfejs kako bi se obezbedilo da rukovalac događaja sadrži odgovarajući metod za obradu događaja.

   b) Rukovalac događaja se mora eksplicitno pridružiti izvoru doga-

đaja kako bi se znalo po nastanku događaja kome treba preneti konstruisani objekat događaja za obradu.

Na slici 6.15 je prikazan odnos između klasa i interfejsa za rukovanje događajima u Javi.



Slika 6.15: Odnos između klasa i interfejsa za rukovanje događajima.

Opšti mehanizam rukovanja događajima u Javi može se opisati u glavnim crtama na sledeći način:

- Rukovalac događaja je objekat klase koja implementira specijalni interfejs rukovaoca događaja.
- Izvor događaja je objekat kome se pridružuju objekti rukovalaca događaja da bi im se preneli objekti nastalih događaja.
- Izvor događaja prenosi objekte događaja svim pridruženim rukovaocima događaja kada se događaj stvarno desi.
- Objekat rukovaoca događaja koristi informacije iz dobijenog objekta događaja da bi se odredila odgovarajuća reakcija na nastali događaj.

Razmotrimo sada konkretnije kako se ovaj mehanizam rukovanja događajima reflektuje na primeru standardne komponente dugmeta. Do sada smo naučili samo kako se dugme prikazuje u prozoru programa. Kako su pažljivi čitaoci možda već primetili u prethodnim primerima, ako se na neko takvo dugme pritisne tasterom miša, ništa se ne dešava. To je zato što mu nismo pridružili nijedan rukovalac događajima koje dugme proizvodi kada se na njega pristisne tasterom miša.

Kada se tasterom miša pritisne na dugme, ono proizvodi događaj tipa `ActionEvent`. Rukovalac ovim tipom događaja mora implementirati specijalni interfejs `ActionListener` koji sadrži samo jedan apstraktni metod čije je ime `actionPerformed()`:

```
package java.awt.event;
public interface ActionListener extends java.util.EventListener {
   public void actionPerformed(ActionEvent e);
}
```

Rukovalac akcijskog događaja kojeg proizvodi dugme zbog toga mora biti objekat klase koja implementira interfejs `ActionListener`:

```
public class RukovalacDugmeta implements ActionListener {
   .
   . // Ostala polja i metodi
   .
   public void actionPerformed(ActionEvent e) {
      .
      . // Reakcija na pritisak tasterom miša na dugme
      .
   }
}
```

U programu još treba konstruisati dugme i rukovalac njegovim akcijskim događajem, kao i uspostaviti vezu između njih:

```
JButton dugme = new JButton("OK");
ActionListener rukovalacDugmeta = new RukovalacDugmeta(...);
dugme.addActionListener(rukovalacDugmeta);
```

Nakon izvršavanja ovog programskog fragmenta, svaki put kada se klikne na `dugme`, odnosno kada se desi akcijski događaj u dugmetu sa oznakom OK, o tome se „izveštava" objekat na koji ukazuje `rukovalacDugmeta`. Sam čin izveštavanja se sastoji od konstruisanja objekta događaja `e` tipa `ActionEvent` i pozivanja metoda

```
rukovalacDugmeta.actionPerformed(e)
```

kome se kao argument prenosi novo konstruisani objekat događaja `e`.

Izvoru događaja može biti po potrebi pridruženo više rukovalaca događaja. Ako je izvor događaja neko dugme, na primer, onda bi se pozivao

metod `actionPerformed()` svih pridruženih rukovalaca akcijskog događaja koji nastaje kada se tasterom miša pritisne na to dugme.

U Javi se koristi konvencija za imena klasa određenih događaja i interfejsa rukovalaca tih događaja. Ime klase događaja je standardnog oblika `<Tip>Event`, a ime interfejsa rukovaoca odgovarajućeg događaja je oblika `<Tip>Listener`. Tako, akcijski događaj je predstavljen klasom `ActionEvent`, a događaj proizveden mišem je predstavljen klasom `MouseEvent`. Interfejsi rukovalaca tih događaja su zato `ActionListener` i `MouseListener`. Sve klase događaja i interfejsi rukovalaca događaja se nalaze u paketu `java.awt.event`.

Pored toga, metodi kojima se rukovalac događaja pridružuje izvoru događaja radi obrade određenog tipa događaja, po konvenciji imaju standardna imena u obliku `add<Tip>Listener()`. Zato, na primer, klasa `Button` (i njena naslednica `JButton`) sadrži metod

```
public void addActionListener(ActionListener a);
```

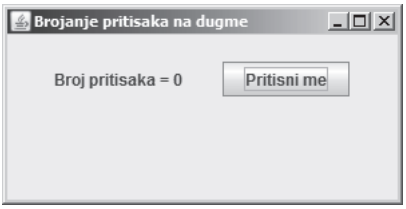
koji je korišćen u prethodnom primeru da bi se dugmetu pridružio rukovalac akcijskog događaja koji dugme može proizvesti.

Bez sumnje, pisanje Java programa u kojima se reaguje na događaje obuhvata mnogo detalja. Ponovimo zato u glavnim crtama osnovne delove koji su neophodni za takav program:

1. Radi korišćenja klasa događaja i interfejsa rukovalaca događaja, na početku programa se mora nalaziti paket `java.awt.event` sa deklaracijom `import`.
2. Radi definisanja objekata koji obrađuju određeni tip događaja, mora se definisati njihova klasa koja implementira odgovarajući interfejs rukovaoca događaja. To je, na primer, interfejs `ActionListener` za akcijske događaje.
3. U toj klasi se moraju definisati svi metodi koji se nalaze u interfejsu koji se implementira. Ovi metodi se pozivaju kada se desi specifični događaj određene komponente.
4. Rukovalac događaja koji je instanca ove klase mora se pridružiti komponenti čiji se događaji obrađuju. Ovo se radi odgovarajućim metodom određene komponente; to je, na primer, metod `addActionListener()` za dugme.

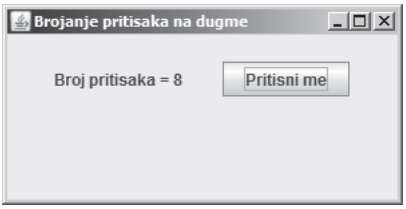
Primer: brojanje pritisaka tasterom miša na dugme

Da bismo ilustrovali kompletan primer rukovanja događajima u Javi, u nastavku je u listingu 6.8 prikazan jednostavan grafički program koji reaguje na pritiske tasterom miša na dugme. U programu se prikazuje okvir sa standardnim dugmetom i oznakom koja prikazuje broj pritisaka tasterom miša na to dugme. Početni prozor programa je prikazan na slici 6.16.



Slika 6.16: Početni prozor za brojanje pritisaka tasterom miša na dugme.

Svaki put kada se tasterom miša pritisne na dugme, rukovalac događaja koji je pridružen dugmetu biva obavešten o događaju tipa `ActionEvent`. Ovaj rukovalac događaja kao odgovor na to treba da odbrojava ukupan broj ovih događaje i promeni tekst oznake koji prikazuje taj broj. Na primer, ako je tasterom miša korisnik pritisnuo 8 puta na dugme, prozor programa treba da izgleda kao što je to prikazano na slici 6.17.



Slika 6.17: Izgled prozora nakon 8 pritisaka tasterom miša na dugme.

Kada se tasterom miša pritisne na dugme u prozoru programa, rukovalac koji obrađuje ovaj događaj treba da uveća brojač pritisaka na dugme za jedan i da prikaže njegov novi sadržaj. Ako se za vrednosti brojača koristi polje `brojač` i za prikaz teksta koristi komponenta `oznaka` tipa `JLabel`,

onda je rukovalac akcijskog događaja ovog dugmeta jedna instanca sledeće klase `RukovalacDugmeta`:

```
class RukovalacDugmeta implements ActionListener {

   private int brojač;  // brojač pritisaka na dugme

   public void actionPerformed(ActionEvent e) {
      brojač++;
      oznaka.setText("Broj pritisaka = " + brojač);
   }
}
```

Glavni okvir programa je kontejner koji sadrži standardne komponente za oznaku i dugme. Tekst oznake služi za prikazivanje odgovarajućeg broja pritisaka na dugme, a dugme je vizuelni element koji se pritiska tasterom miša i proizvodi akcijske događaje. Glavni okvir programa je zato predstavljen klasom `DugmeBrojačOkvir` koja proširuje klasu `JFrame`. Pored toga, ova klasa glavnog okvira sadrži polje `oznaka` tipa `JLabel` i konstruktor za inicijalizaciju okvira:

```
class DugmeBrojačOkvir extends JFrame {

   private JLabel oznaka;  // tekst broja pritisaka na dugme
   . . .

   // Konstruktor
   public DugmeBrojačOkvir() {
      setTitle("Brojanje pritisaka na dugme ");
      setSize(300, 150);
      setLayout(new FlowLayout(FlowLayout.CENTER, 30, 20));

      oznaka = new JLabel("Broj pritisaka = 0");
      add(oznaka);
      JButton dugme = new JButton("Pritisni me");
      add(dugme);
      dugme.addActionListener(new RukovalacDugmeta());
   }
}
```

U konstruktoru klase `DugmeBrojačOkvir` se najpre određuje naslov, veličina i način razmeštanja komponenti u glavnom okviru. Zatim se konstruišu komponente za oznaku i dugme sa prigodnim tekstom i dodaju okviru. Na kraju, konstruisanom dugmetu se metodom `addActionListener()` pridružuje rukovalac akcijskog događaja koji je instanca prethodne klase `RukovalacDugmeta`.

Obratite pažnju na to da klasa `DugmeBrojačOkvir` mora da sadrži posebno polje `oznaka` tipa `JLabel`. To je zato što rukovalac događaja dugmeta treba da promeni tekst oznake kada se tasterom miša pritisne na dugme. Pošto se komponenta za oznaku mora nalaziti u kontejneru-okviru, polje `oznaka` ukazuje na oznaku u okviru kako bi rukovalac događaja dugmeta preko tog polja imao pristup do oznake u okviru.

Ali to stvara mali problem — u klasi `RukovalacDugmeta` se ne može koristiti polje `oznaka`, jer je ono privatno polje u klasi `DugmeBrojačOkvir`. Postoji više rešenja ovog problema, od najgoreg da se ovo polje učini javnim, do najboljeg da klasa `RukovalacDugmeta` bude ugnježđena u klasi `DugmeBrojačOkvir`.

Ovo je inače česta primena ugnježđenih klasa o kojima smo govorili u odeljku 5.4. Objekti koji obrađuju događaje moraju obično da izvedu neki postupak koji utiče na druge objekte u programu. To se onda može elegantno rešiti ukoliko se klasa rukovalaca događaja definiše unutar klase objekata čije stanje rukovalac događaja treba da menja.

U listingu 6.8 je prikazan kompletan program za brojanje pritisaka tasterom miša na dugme u kome je klasa `RukovalacDugmeta` ugnježđena u klasu `DugmeBrojačOkvir`.

Listing 6.8: Brojanje pritisaka tasterom miša na dugme

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class TestDugmeBrojač {

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

```
      DugmeBrojačOkvir okvir = new DugmeBrojačOkvir();
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
   }
}

// Klasa za glavni okvir sa dugmetom i oznakom
class DugmeBrojačOkvir extends JFrame {

   private JLabel oznaka;  // tekst broja pritisaka na dugme

   // Ugnježđena klasa rukovaoca akcijskog događaja
   private class RukovalacDugmeta implements ActionListener {

      private int brojač;  // brojač pritisaka na dugme

      public void actionPerformed(ActionEvent e) {
         brojač++;
         oznaka.setText("Broj pritisaka = " + brojač);
      }
   }

   // Konstruktor
   public DugmeBrojačOkvir() {
      setTitle("Brojanje pritisaka na dugme ");
      setSize(300, 150);
      setLayout(new FlowLayout(FlowLayout.CENTER, 30, 20));

      oznaka = new JLabel("Broj pritisaka = 0");
      add(oznaka);
      JButton dugme = new JButton("Pritisni me");
      add(dugme);
      dugme.addActionListener(new RukovalacDugmeta());
   }
}
```

Primetimo da se ovaj program može još više pojednostaviti. Naime, klasa `RukovalacDugmeta` se koristi samo jedanput u poslednjoj naredbi konstruktora `DugmeBrojačOkvir()` radi konstruisanja objekta za obradu događaja koje proizvodi dugme u prozoru programa. To znači da je potreban samo *jedan* objekat klase `RukovalacDugmeta` u programu, pa se ova klasa može definisati da bude anonimna ugnježđena klasa. Prethodni program sa ovom izmenom postaje znatno kraći, i zato jasniji, što se možemo uveriti iz listinga 6.9.

Listing 6.9: Brojanje pritisaka tasterom miša na dugme, kraća verzija

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class TestDugmeBrojač {

   public static void main(String[] args) {

      DugmeBrojačOkvir okvir = new DugmeBrojačOkvir();
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
   }
}

// Klasa za glavni okvir sa dugmetom i oznakom
class DugmeBrojačOkvir extends JFrame {

   private JLabel oznaka;  // tekst broja pritisaka na dugme

   // Konstruktor
   public DugmeBrojačOkvir() {
      setTitle("Brojanje pritisaka na dugme ");
      setSize(300, 150);
      setLayout(new FlowLayout(FlowLayout.CENTER, 30, 20));

      oznaka = new JLabel("Broj pritisaka = 0");
```

```
      add(oznaka);
      JButton dugme = new JButton("Pritisni me");
      add(dugme);

      // Dodavanje objekta za rukovanje događajem dugmeta kao
      // instance anonimne klase koja implementira ActionListener
      dugme.addActionListener(new ActionListener() {

         private int brojač;  // brojač pritisaka na dugme

         public void actionPerformed(ActionEvent e) {
            brojač++;
            oznaka.setText("Broj pritisaka = " + brojač);
         }
      });
   }
}
```

Adapterske klase

Nisu svi događaji tako jednostavni za obradu kao pritisak tasterom miša na dugme. Na primer, interfejs `MouseListener` sadrži pet metoda koji se pozivaju kao reakcija na događaje proizvedene mišem:

```
public interface MouseListener extends java.util.EventListener {
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
   public void mouseClicked(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
}
```

Metod `mousePressed()` se poziva čim se pritisne jedan od tastera miša, a metod `mouseReleased()` se poziva kada se otpusti taster miša. Treći metod `mouseClicked()` se poziva kada se taster miša pritisne i brzo otpusti, bez pomeranja miša. (U stvari, to izaziva pozivanje sva tri metoda `mousePressed()`, `mouseReleased()` i `mouseClicked()`, tim redom.) Metodi `mou-`

seEntered() i mouseExited() se pozivaju čim pokazivač miša uđe u pravougaonu oblast neke komponente i čim je napusti.

U programu se obično ne mora voditi računa o svim mogućim događajima koji se proizvode mišem, ali se u klasi rukovalaca ovim događajima koja implementira interfejs MouseListener moraju definisati svih pet metoda. Naravno, oni metodi koji nisu potrebni se definišu tako da imaju prazno telo metoda.

Drugi primer je komponenta za okvir tipa JFrame koja je izvor događaja tipa WindowEvent. Klasa rukovalaca ovog događaja mora implementirati interfejs WindowListener koji sadrži sedam metoda:

```
public interface WindowListener extends java.util.EventListener {
   public void windowOpened(WindowEvent e);
   public void windowClosing(WindowEvent e);
   public void windowClosed(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowActivated(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
}
```

Na osnovu imena ovih metoda se lako može zaključiti kada se koji metod poziva, osim možda za windowIconified() i windowDeiconified() koji se pozivaju kada se okvir minimizuje i otvara iz minimizovanog stanja.

Ukoliko je u programu potrebno omogućiti korisniku da se predomisli kada pokuša da zatvori glavni okvir programa, onda bi trebalo definisati klasu rukovaoca događaja tipa WindowEvent koja implementira interfejs WindowListener. U toj klasi bi zapravo trebalo definisati samo odgovarajući metod windowClosing(), a ostali metodi bi trebalo da budu trivijalno definisali sa praznim telom:

```
public class ZatvaračOkvira implements WindowListener {

   public void windowClosing(WindowEvent e) {

      if (. . .)          // korisnik želi da završi program?
         System.exit(0);  // ako je to slučaj, završiti program
   }
```

```
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

Da se ne bi formalno pisali nepotrebni metodi sa praznim telom, svaki interfejs rukovaoca događaja koji sadrži više od jednog metoda ima svoju *adaptersku klasu* koja ga implementira tako što su svi njegovi metodi trivijalno definisani sa praznim telom. Tako, klasa `MouseAdapter` ima svih pet trivijalno definisanih metoda interfejsa `MouseListener`, dok klasa `WindowAdapter` ima sedam trivijalno definisanih metoda interfejsa `WindowListener`.

To znači da adapterska klasa automatski zadovoljava tehnički uslov koji je u Javi potreban za klasu koja implementira neki interfejs, odnosno adapterska klasa definiše sve metode interfejsa koji implementira. Ali pošto svi metodi adapterske klase zapravo ništa ne rade, ove klase su korisne samo za nasleđivanje. Pri tome, proširivanjem adapterske klase se mogu definisati (nadjačati) samo neki, potrebni metodi odgovarajućeg interfejsa rukovaoca događaja, a ne svi. To je i razlog zašto za interfejse koji sadrže samo jedan metod (na primer, `ActionListener`) nema potrebe za adapterskom klasom.

Ako se dakle nasleđuje adapterska klasa `WindowAdapter` u prethodnom primeru klase `ZatvaračOkvira`, onda treba nadjačati samo jedan metod te adapterske klase, `windowClosing()`, koji je bitan u primeru:

```
public class ZatvaračOkvira extends WindowAdapter {

   public void windowClosing(WindowEvent e) {

      if (. . .)          // korisnik želi da završi program?
         System.exit(0);  // ako je to slučaj, završiti program
   }
}
```

Rukovalac tipa `ZatvaračOkvira` može se na standardan način pridružiti nekom okviru radi obrade njegovih događaja koje proizvodi:

```
okvir.addWindowListener(new ZatvaračOkvira());
```

Sada kada `okvir` proizvede neki događaj tipa `WindowEvent`, poziva se jedan od sedam metoda pridruženog rukovaoca tipa `ZatvaračOkvira`. Šest od tih sedam metoda ne radi ništa, odnosno odgovarajući događaji se zanemaruju. Ako se okvir zatvara, poziva se metod `windowClosing()` koji pozivom metoda `System.exit(0)` završava program, ako to korisnik odobri.

Ako se konstruiše samo jedan objekat klase `ZatvaračOkvira` u programu, ova klasa ne mora čak ni da se posebno definiše, nego može biti anonimna ugnježđena klasa:

```
okvir.addWindowListener(new WindowAdapter() {

   public void windowClosing(WindowEvent e) {

      if (. . .)          // korisnik želi da završi program?
         System.exit(0);  // ako je to slučaj, završiti program
   }
});
```

Primer: uslovno zatvaranje prozora programa

U ovom primeru je prethodni program, koji broji pritiske tasterom miša na dugme, proširen tako da se glavni okvir programa ne zatvara bezuslovno kada korisnik pritisne na dugme `X` u gornjem desnom uglu okvira. U tom slučaju će se pojaviti okvir sa porukom, prikazan na slici 6.18, koji korisniku pruža šansu da odustane od zatvaranja glavnog okvira i da nastavi rad.



Slika 6.18: Okvir sa porukom za završetak rada programa.

Jedina izmena koja je neophodna u programu za brojanje pritisaka tasterom miša na dugme jeste dodavanje rukovaoca događaja zatvaranja glavnog okvira. U programu se primenjuje postupak koji smo opisali u prethodnom odeljku za adaptersku klasu `WindowAdapter`. Ako se podsetimo, glavnom okviru se pridružuje rukovalac događaja zatvaranja okvira, a taj rukovalac je instanca anonimne ugnježđene klase. U toj klasi se nadjačava metod `windowClosing()` koji prikazuje okvir sa prigodnom porukom i prekida rad programa ukoliko korisnik to potvrdi.

Za dobijanje potvrde od korisnika da se završi rad, u programu se koristi metod `showOptionDialog()` iz klase `JOptionPane`. Svi detalji ovog metoda nisu bitni za obradu događaja glavnog okvira programa i mogu se naći u zvaničnoj dokumentaciji jezika Java. Primetimo ipak da se podrazumevana aktivnost prilikom zatvaranju okvira mora izabrati tako da se ne radi ništa:

```
okvir.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

To je zato što se ova aktivnost uvek izvodi nakon što neki rukovalac obradi događaj zatvaranja okvira, pa bi inače glavni okvir postao nevidljiv kada korisnik odustane od završetka programa. Naime, podrazumevana aktivnost prilikom zatvaranja svakog okvira je da se okvir učini nevidljivim.

U listingu 6.10 je prikazan kompletan program koji, pored prikazivanja broja pritisaka tasterom miša na dugme, omogućava korisniku da se predomisli kada pritisne na dugme `X` u gornjem desnom uglu glavnog okvira programa.

Listing 6.10: Uslovno zatvaranje prozora programa

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class TestDugmeBrojač {

   public static void main(String[] args) {

      DugmeBrojačOkvir okvir = new DugmeBrojačOkvir();
      okvir.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
      // Pridruživanje rukovaoca događaja zatvaranja okvira kao
      // instance anonimne klase koja proširuje WindowAdapter
      okvir.addWindowListener(new WindowAdapter() {

         public void windowClosing(WindowEvent e) {

            Object[] opcija = {"Da", "Ne"};
            int izabranaOpcija = JOptionPane.showOptionDialog(null,
                           "Zaista želite da završite program?",
                           "Kraj rada programa",
                           JOptionPane.DEFAULT_OPTION,
                           JOptionPane.WARNING_MESSAGE,
                           null, opcija, opcija[0]);
            if (izabranaOpcija == 0)  // korisnik želi da završi?
               System.exit(0);        // ako da, završiti program
         }
      });
      okvir.setVisible(true);
   }
}

// Klasa za glavni okvir sa dugmetom i oznakom
class DugmeBrojačOkvir extends JFrame {

   private JLabel oznaka;    // oznaka u okviru

   // Konstruktor
   public DugmeBrojačOkvir() {
      setTitle("Brojanje pritisaka na dugme ");
      setSize(300, 150);
      setLayout(new FlowLayout(FlowLayout.CENTER, 30, 20));

      oznaka = new JLabel("Broj pritisaka = 0");
      add(oznaka);
      JButton dugme = new JButton("Pritisni me");
      add(dugme);
```

```
      // Dodavanje objekta za rukovanje događajem dugmeta kao
      // instance anonimne klase koja implementira ActionListener
      dugme.addActionListener(new ActionListener() {

         private int brojač; // brojač pritisaka na dugme

         public void actionPerformed(ActionEvent e) {
            brojač++;
            oznaka.setText("Broj pritisaka = " + brojač);
         }
      });
   }
}
```

GLAVA 7

Programske greške

Naglasak u prethodnim poglavljima knjige bio je na mogućnostima programskog jezika Java za pisanje ispravnih programa. Ali kao što su čitaoci mogli primetiti, programiranje je intelektualna aktivnost koja je, nažalost, vrlo podložna greškama. Dodatno, sasvim mala nepreciznost u programu može dovesti do prevremenog prekida izvršavanja programa ili do, još gore, normalnog završetka programa, ali sa potpuno pogrešnim rezultatima. Posledice loših programa mogu biti manje ozbiljne (npr. izgubljeno vreme i nerviranje u običnim primenama jer se program „ukočio" ili „puca"), vrlo ozbiljne (npr. izgubljen novac i ugled u bankarskim i telekomunikacionim primenama), pa čak i katastrofalne (npr. gubitak ljudskih života u medicinskim i saobraćajnim primenama). Zbog toga se u ovom poglavlju govori o sredstvima koja stoje Java programerima na raspolaganju radi rešavanja ovog opšteg problema programiranja.

7.1 Ispravni i robustni programi

Ispravnost programa je najvažniji uslov koji svi programi moraju da ispunjavaju, jer je, naravno, besmisleno da programi ne obavljaju zadatak za koji su predviđeni. Ali skoro isto tako važan uslov za dobre programe je da budu robustni. To znači da oni u neočekivanim okolnostima tokom svog izvršavanja treba da reaguju na razuman način.

Razmotrimo konkretnije jedno programsko rešenje problema sortiranja, odnosno program koji treba da učita neki niz brojeva od korisnika i da

prikaže te iste brojeve u rastućem redosledu. Taj program je ispravan ukoliko svoj zadatak obavlja za bilo koji niz brojeva koje korisnik unese. Ovaj program je robustan ukoliko se razumno postupa sa potencijalnim korisnikovim greškama. Na primer, kada korisnik pogrešno unese nenumeričku vrednost za neki ulazni broj, program treba da prikaže odgovarajuću poruku o grešci i da prekine svoje izvršavanje na kontrolisan način.

Treba naglasiti da, iako svaki program mora biti apsolutno ispravan, svaki program ne mora biti potpuno robustan. Na primer, neki mali pomoćni program koji smo napisali isključivo za svoje potrebe ne mora biti toliko robustan kao neki komercijalni program koji se koristi u poslovne svrhe. Očigledno je da nema svrhe ulagati dodatni trud u povećanje robustnosti pomoćnog programa koji jedino sami koristimo, jer verovatno nećemo napraviti grešku prilikom njegovog korišćenja. A ukoliko se to nenamerno i dogodi, pošto dobro poznajemo svoj program, znaćemo kako da reagujemo na grešku pri njegovom izvršavanju. S druge strane, izvršavanje komercijalnog programa nije pod kontrolom programera i takav program mogu koristiti osobe različitog nivoa računarske pismenosti. Zbog toga se za izvršavanje komercijalnog programa mora pretpostaviti minimalističko okruženje i moraju se predvideti sve vanredne okolnosti kako bi i najmanje sofisticirani korisnici mogli pravilno reagovati na greške.

U Javi je od početka prepoznata važnost ispravnih i robustnih programa, pa su u sam jezik ugrađeni mehanizmi koji najčešće greške programera svode na najmanju meru. To je posebno obezbeđeno sledećim svojstvima programskog jezika Java:

- Sve promenljive moraju biti deklarisane. U (starijim) programskim jezicima kod kojih promenljive ne moraju da se deklarišu, promenljive se uvode kada se prvi put koriste u programu. Iako ovaj način rada sa promenljivim izgleda pogodniji za programere, on je čest uzrok teško prepoznatljivih grešaka kada se nenamerno pogrešno napiše ime neke promenljive. To je isto kao da se uvodi potpuno različita promenljiva, što može izazvati neočekivane rezultate čiji uzrok nije lako otkriti. Kada sve promenljive moraju unapred biti deklarisane, kompajler može prilikom prevođenja programa da otkrije da pogrešno napisana promenljiva nije deklarisana i da ukaže na nju. To onda znači da programer lako može ispraviti takvu grešku.

- Indeksi elemenata niza u Javi se proveravaju da li leže u dozvoljenim granicama. Nizovi se sastoje od elemenata koji su numerisani od nule do nekog maksimalnog indeksa. Svako korišćenje elementa niza čiji je indeks van ovog intervala biće otkriveno od strane Java interpretatora (JVM) i izvršavanje programa biće nasilno prekinuto, ukoliko se na neki drugi način ne postupi u programu. U drugim jezicima (kao što su C i C++) prekoračenje granica niza se ne proverava i zato je u tim jezicima moguće da se, recimo, upiše neka vrednost u memorijsku lokaciju koja ne pripada nizu. Kako ta lokacija služi za potpuno druge svrhe, posledice upotrebe nepostojećeg elementa niza su vrlo nepredvidljive. Ova programska graška se popularno naziva *prekoračenje bafera* (engl. *buffer overflow*) i može se iskoristiti u maliciozne svrhe u raznim „virusima". Greške ovog tipa nisu dakle moguće u Java programima, jer je nemoguć nekontrolisan pristup memorijskim lokacijama koje ne pripadaju nizu.
- Direktno manipulisanje pokazivačima u Javi nije dozvoljeno. U drugim jezicima je u suštini moguće obezbediti da pokazivačka promenljiva ukazuje na proizvoljnu memorijsku lokaciju. Ova mogućnost s jedne strane daje veliku moć programerima, ali je s druge strane vrlo opasna, jer ukoliko se pokazivači koriste na pogrešan način to dovodi do nepredvidljivih rezultata. U Javi neka promenljiva klasnog tipa (tj. pokazivačka promenljiva) ili može ukazivati na valjani objekat u memoriji ili takva promenljiva može sadržati specijalnu vrednost `null`. Svako korišćenje pokazivača čija je vrednost `null` biće otkrivena od strane Java interpretatora i izvršavanje programa biće nasilno prekinuto, ukoliko se na neki drugi način ne postupi u programu. A ukoliko pokazivač ukazuje na ispravnu memorijsku lokaciju, pošto aritmetičke operacije sa pokazivačima nisu dozvoljene, u Javi je nemoguće nekontrolisano pristupiti pogrešnim delovima programske memorije.
- U Javi se automatski oslobađa memorija koju zauzimaju objekti koji nisu više potrebni u programu. U drugim jezicima su sami programeri odgovorni za oslobađanje memorije koju zauzimaju nepotrebni objekti. Ukoliko to ne učine (iz nemarnosti ili iz neznanja), nepotrebno zauzeta memorija može se brzo uvećati do nivoa kada je

dalje izvršavanje samog programa (ili drugih programa u računaru) nemoguće, jer u računaru nema više slobodne glavne memorije. Ova programska graška se popularno naziva *curenje memorije* (engl. *memory leak*) i u Javi se mnogo teže mogu nenamerno izazvati greške ovog tipa.

Svi potencijalni izvori grešaka programera ipak nisu mogli biti potpuno eliminisani u Javi, jer bi njihovo otkrivanje znatno usporilo izvršavanje programa. Jedna oblast gde su greške moguće su numerička izračunavanja koja ne podležu nikakvim proverama. Neki primeri takvih grešaka u Javi su:

- Kod prekoračenja opsega za cele brojeve mogu se dobiti neočekivani rezultati: na primer, zbir 2147483647 + 1 dva broja tipa `int` kao rezultat daje najmanji negativan broj –2147483648 tipa `int`.
- Nedefinisane vrednosti realnih izraza kao što su deljenje sa nulom ili kvadratni koren negativnog broja (na primer, $\sqrt{-4}$), kao i rezultati prekoračenja opsega brojeva za tipove `float` i `double`, predstavljeni su specijalnim konstantama u klasi `Double`: `POSTIVE_INFINITY`, `NEGATIVE_INFINITY` i `NaN`.
- Zbog približnog predstavljanja realnih brojeva (na primer, realan broj $1/3 = 0.333\cdots$ sa beskonačno decimala zaokružuje se na broj 0.3333333 sa 7 decimala tipa `float` ili na broj 0.333333333333333 sa 15 decimala tipa `double`), greške usled zaokruživanja realnih brojeva mogu se akumulirati i dovesti do netačnih rezultata.

Robustni programi, kao što je već spomenuto, moraju „preživeti" vanredne okolnosti tokom svog izvršavanja. Za pisanje takvih programa može se primeniti opšti pristup koji nije specifičan za Javu i koji se sastoji u tome da se predvide sve izuzetne okolnosti u programu i uključe provere za svaki potencijalni problem. Na primer, ako se u programu koriste elementi niza *a*, onda se u Javi proveravaju njihovi indeksi da li leže u dozvoljenim granicama i ukoliko to nije slučaj, izvršavanje programa se nasilno prekida. Međutim, da bismo takve greške sami otkrili i kontrolisali, u programu možemo postupiti na sledeći način:

```
if (i < 0 || i > a.length) {
   // Obrada greške nedozvoljenog indeksa i
   System.out.println("GREŠKA: pogrešan indeks " + i + " niza a");
   System.exit(0);
}
else {
   .
   . // Normalna obrada elementa a[i]
   .
}
```

U ovom primeru se prikazuje vrednost indeksa niza *a* koji leži u nedozvoljenim granicama i prekida se ceo program. Ponekad takvo postupanje sa greškom nije odgovarajuće, jer je možda u nekom drugom delu programa moguć oporavak od greške na manje brutalan način. Tako, u metodu u kojem se koriste elementi niza *a* možda treba samo vratiti indikaciju da je došlo do greške. Prethodni programski fragment zato postaje:

```
if (i < 0 || i > a.length) {
   // Obrada greške nedozvoljenog indeksa i
   return -1;
}
else {
   .
   . // Normalna obrada elementa a[i]
   .
}
```

U ovom primeru se vraća specijalna vrednost –1 kao indikacija da je došlo do greške i prepušta se pozivajućem metodu da proveri ovaj rezultat i da shodno tome postupi na način koji je možda prikladniji od prostog prekida izvršavanja.

Glavni nedostaci opšteg pristupa za sprečavanje potencijalnih grešaka su:

- Teško je predvideti sve potencijalne probleme u programu, a nekad je to i nemoguće.
- Nije uvek jasno kako treba postupati u slučaju nekog problema.

- Program postaje komplikovan splet „pravih” naredbi i `if` naredbi. Na taj način logika čak i jednostavnih programa postaje teško razumljiva, a to dalje izaziva mnogo ozbiljnije teškoće kod menjanja programa.

U Javi je uzeta u obzir manjkavost ovog ad-hok pristupa za pisanje robustnih programa na drugim jezicima, pa je predviđen poseban mehanizam koji obezbeđuje postupanje sa greškama u programu na više strukturiran način. Taj mehanizam se zasniva na konceptu *izuzetaka* i naziva se *rukovanje izuzecima* (engl. *exception handling*).

7.2 Izuzeci

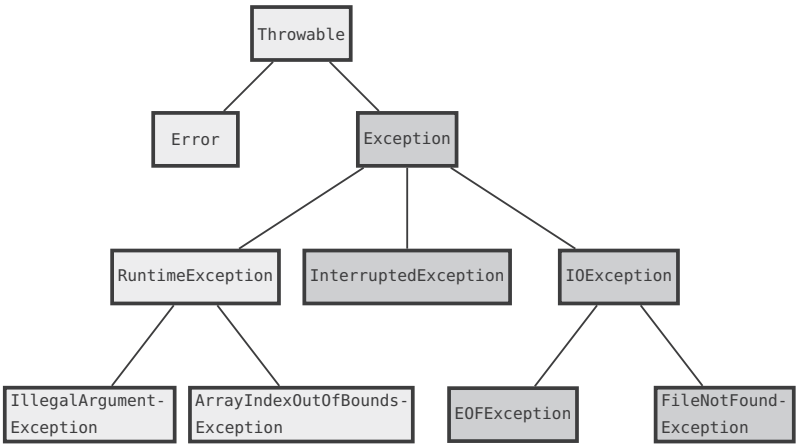
Izuzetak (engl. *exception*) je zapravo skraćeni termin za izuzetnu okolnost koja se može desiti tokom izvršavanja programa. Izuzeci su opštiji pojam od programerskih grešaka i obuhvataju sve vanredne događaje koji program mogu skrenuti sa normalnog toka izvršavanja. Tu spadaju recimo i hardverske greške na koje programer nema uticaja.

Terminologija u vezi sa izuzecima je prilično kolokvijalna, pa se tako kaže da je izuzetak *izbačen* kada se izuzetna okolnost desi tokom izvršavanja programa i „izbaci ga iz koloseka”. Ukoliko se u programu ne postupi sa izuzetkom na neki način kada se izuzetak desi, program se momentalno prekida i završava se njegovo izvršavanje. Postupak rukovanja izuzetkom u programu sastoji se od dve aktivnosti: izbačeni izuzetak se najpre *hvata* i zatim se izuzetak *obrađuje*.

Opšti koncept izuzetaka je u Javi naravno prilagođen objektno orijentisanoj metodologiji. To znači da se izuzeci predstavljaju objektima određenih klasa koji se konstruišu u trenutku nastanka izuzetne okolnosti tokom izvršavanja programa. Ovi objekti izuzetaka sadrže opis uzroka koji je doveo do izbacivanja izuzetka, listu metoda koji su bili aktivni kada se izuzetak dogodio (engl. *method call stack*) i slične bliže informacije o stanju programa u trenutku prekida njegovog izvršavanja.

Svi objekti izuzetka moraju pripadati nekoj klasi koja je naslednica standardne klase `Throwable` iz paketa `java.lang`. Ova klasa se nalazi na vrhu složene hijerarhije klasa kojima su predstavljeni različiti tipovi izuzetaka.

Na slici 7.1 je prikazan samo mali deo ove hijerarhije sa nekim karakterističnim klasama.



Slika 7.1: Deo hijerarhije klasa izuzetaka.

Kao što je prikazano na slici 7.1, klasa `Throwable` ima dve direktne klase naslednice, `Error` i `Exception`. Ove dve klase imaju svoje unapred definisane mnoge druge klase naslednice. Pored toga, ukoliko postojeće klase nisu odgovarajuće za konkretnu primenu, u programu se mogu definisati nove klase izuzetaka radi predstavljanja posebnih tipova grešaka.

Klasa `Error` i njene naslednice predstavljaju vrlo ozbiljne greške unutar samog Java interpretatora. To su obično fatalne greške koje treba da izazovu prekid izvršavanja programa pošto ne postoji razuman način da se program oporavi od tih grešaka. Na primer, izuzetak tipa `ClassFormatError` dešava se kada Java interpretator otkrije neispravan format bajtkoda u datoteci koja sadrži prevedenu klasu. Ako je takva klasa deo programa koji se izvršava, onda sigurno nema smisla da se nastavi dalje izvršavanje tog programa.

U opštem slučaju dakle, izuzeci tipa (ili podtipa od) `Error` nisu predviđeni da budu uhvaćeni i obrađeni u programu, nego samo da izazovu prevremen prekid programa. S druge strane, izuzeci tipa (ili podtipa od) `Exception` jesu izuzeci koji su predviđeni da budu uhvaćeni i obrađeni u programu na odgovarajući način, kako ne bi izazvali nasilan prekid programa. Ovi izuzeci u mnogim slučajevima predstavljaju greške u programu

ili ulaznim podacima koje se mogu očekivati i od kojih se program može oporaviti na razuman način.

Među naslednicama klase `Exception` je klasa `RuntimeException` koja opisuje najčešće greške koje se mogu desiti tokom izvršavanja programa. Tu spadaju greške usled nedozvoljenog argumenta metoda, prekoračenja granice niza, korišćenja nepostojeće reference i tako dalje. Opštije, izuzeci tipa `RuntimeException` predstavljaju programske greške koje su posledica propusta programera i zato ih treba ispraviti u programu.

Izuzeci tipa `RuntimeException` i `Error` (i njihovih podtipova) imaju zajedničku osobinu da se mogu zanemariti u programu u smislu da se svesno prenebregava činjenica da će se program nasilno prekinuti ukoliko se takvi izuzeci dese tokom izvršavanja. To znači da njihovo hvatanje i obrada u programu nije obavezna, već se programeru ostavlja izbor da li je bolje u programu reagovati na ove izuzetke ili prosto dozvoliti da se izvršavanje programa nasilno prekine.

S druge strane, zajednička osobina izuzetaka klase `Exception` i svih njenih klasa naslednica (osim `RuntimeException`) jeste to da se tim izuzecima mora posvetiti posebna pažnja u programu. To znači da ukoliko u programu postoji mogućnost njihovog izbacivanja, oni se u programu moraju hvatati i obraditi na način o kome ćemo govoriti u nastavku. Neke od klasa izuzetaka čije je rukovanje u programu obavezno prikazane su na slici 7.1 tamnijom bojom.

Klasa `Throwable` sadrži nekoliko objektnih metoda koji se mogu koristiti sa svakim objektom izuzetka. Na primer, ako je `e` promenljiva tipa `Throwable` koja sadrži referencu na neki objekat izuzetka, onda:

- `e.getMessage()` kao rezultat daje tekstualni opis greške koja je dovela do izbacivanja izuzetka.
- `e.printStackTrace()` prikazuje listu aktivnih metoda u trenutku izbacivanja izuzetka.
- `e.toString()` kao rezultat daje reprezentaciju objekta izuzetka u obliku stringa.

Naredba try

Za rukovanje izuzecima u programu, odnosno za njihovo hvatanje i obradu, koristi se posebna naredba `try`. Nepotpun oblik ove naredbe sastoji se od dva bloka, `try` i `catch`, koji se pišu jedan iza drugog:

```
try {
    .
    . // Naredbe koje mogu izazvati izbacivanje izuzetaka
    .
}
catch (tip-izuzetka promenljiva) {
    .
    . // Naredbe kojima se obrađuje izuzetak tipa tip-izuzetka
    .
}
```

Pojednostavljen opis izvršavanja naredbe `try` može se podeliti u više koraka:

1. Izvršavaju se naredbe u telu bloka `try`.
   - Ako se tokom izvršavanja ovih naredbi u telu bloka `try` *ne* izbaci nijedan izuzetak ili se izbaci izuzetak različitog tipa od onog navedenog u zagradi iza klauzule `catch`, izvršavanje tela bloka `catch` se potpuno preskače.
   - Ako se tokom izvršavanja naredbi u telu bloka `try` izbaci izuzetak tipa navedenog u zagradi iza klauzule `catch`, odmah se prekida dalje izvršavanje naredbi u telu bloka `try` i prelazi se na izvršavanje naredbi u telu bloka `catch`.
2. Izvršavanje programa se nastavlja iza tela bloka `catch`, ako drugačije nije određeno.

Na primer, za inicijalizovanje *i*-tog elementa niza *a* primenom naredbe `try` može se napisati:

```
try {
   a[i] = 0;
}
catch (ArrayIndexOutOfBoundsException e) {
```

```
    System.out.println("Indeks " + i + " niza a je izvan granica");
    e.printStackTrace();
}
```

U ovom `try-catch` bloku se najpre pokušava izvršavanje bloka naredbi iza klauzule `try`. Kako se taj blok ovde sastoji samo od jedne naredbe, to znači da se najpre pokušava dodeljivanje vrednosti 0 elementu niza *a* sa indeksom *i*. Ako se ta naredba dodele uspešno izvrši, odnosno pri njenom izvršavanju se ne izbaci nijedan izuzetak, izvršavanje bloka `catch` se prosto preskače i izvršavanje programa se normalno nastavlja od mesta neposredno iz bloka `catch`. Ako se pak naredba dodele u bloku `try` ne može uspešno izvršiti usled nepravilnog indeksa niza, izbacuje se izuetak tipa

```
ArrayIndexOutOfBoundsException
```

koji se hvata blokom `catch` i izvršavaju se naredbe u telu tog bloka. Tačnije, pre izvršavana tela bloka `catch` konstruiše se objekat koji predstavlja izbačeni izuzetak i njegova referenca se dodeljuje promenljivoj `e` koja se koristi u telu bloka `catch`. Drugim rečima, ovde će se izvršavanjem bloka `catch` najpre prikazati poruka o grešci prekoračenja granica niza *a* i zatim će se prikazati lista aktivnih metoda u trenutku izbacivanja izuzetka. Nakon toga se izvršavanje programa nastavlja od mesta neposredno iza bloka `catch`.

U opštem slučaju, vitičaste zagrade u blokovima `try` i `catch` su obavezne i ne mogu se izostaviti iako sadrže samo jednu naredbu. Telo bloka `catch` se naziva *rukovalac izuzetka* (engl. *exception handler*) odgovarajućeg tipa, jer je to deo programa u kome se postupa sa izuzetkom onog tipa koji je naveden u zagradi iza službene reči `catch`.

Primetimo da se u prethodnom primeru hvata i obrađuje samo izuzetak izbačen usled prekoračenja granice niza, mada generalno jedan blok `try` može imati više pridruženih blokova `catch` kako bi se moglo rukovati izuzecima više različitih tipova koji se mogu izbaciti u jednom bloku `try`. Tako se prethodni primer može proširiti, recimo, na sledeći način:

```
try {
   a[i] = 0;
}
catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Indeks " + i + " niza a je izvan granica");
```

```
    e.printStackTrace();
catch (NullPointerException e) {
    System.out.println("Niz a ne postoji");
    e.printStackTrace();
}
```

U ovom fragmentu se prilikom izvršavanja naredbe dodele `a[i] = 0` u bloku `try`, hvataju i obrađuju dve vrste grešaka: prekoračenje granica niza i korišćenje nepostojeće reference. Ukoliko se ne desi nijedna greška u bloku `try`, oba bloka `catch` se preskaču. Ako se izbaci izuzetak usled prekoračenja granica niza, izvršava se samo prvi blok `catch`, a drugi se preskače. Ako se izbaci izuzetak usled korišćenja nepostojeće reference, izvršava se samo drugi blok `catch`, a prvi se preskače.

Obe klase `ArrayIndexOutOfBoundsException` i `NullPointerException` su naslednice klase `RuntimeException`, pa se prethodni primer može kraće zapisati na sledeći način:

```
try {
    a[i] = 0;
}
catch (RuntimeException e) {
    System.out.print("Greška " + e.getMessage());
    System.out.println(" u radu sa nizom a");
    e.printStackTrace();
}
```

U ovom fragmentu se klauzulom `catch` hvataju svi izuzeci koji pripadaju klasi `RuntimeException` ili bilo kojoj od njenih klasa naslednica. Ovaj princip važi u opštem slučaju i pokazuje zašto su klase izuzetaka organizovane u hijerarhiju klasa. Naime, to obezbeđuje veću fleksibilnost u rukovanju izuzecima, jer programeri imaju mogućnost da u posebnim slučajevima hvataju specifične tipove izuzetaka, a i da hvataju mnogo šire tipove izuzetaka ukoliko nije bitan njihov poseban tip. Primetimo da je zbog ovoga moguće da izbačeni izuzetak odgovara većem broju rukovalaca, ako je više blokova `catch` pridruženo jednom bloku `try`. Na primer, izuzetak tipa `NullPointerException` bi se uhvatio klauzulama `catch` za tipove `NullPointerException`, `RuntimeException`, `Exception` ili `Throwable`. U tom slučaju, izvršava se samo prvi blok `catch` u kojem se hvata izuzetak.

Uzgred, mada prethodni primeri nisu mnogo realistični, oni pokazuju zašto rukovanje izuzecima tipa `RuntimeException` nije obavezno u programu. Jednostavno, mnogo grešaka ovog tipa se može desiti tokom izvršavanja programa, pa bi svakako bilo frustrirajuće ukoliko bi se morali pisati `try-catch` blokovi svaki put kada se, recimo, koristi neki niz u programu. Pored toga, za sprečavanje programskih grešaka sličnog tipa kao što su one usled prekoračenja granica niza ili korišćenja nepostojeće reference, bolje je primeniti pažljivo programiranje nego rukovanje izuzecima.

Klauzula `finaly`

Dosadašnji opis naredbe `try` nije potpun, jer ta naredba u Javi može imati dodatnu klauzulu `finaly`. Razlog za ovu mogućnost je taj što u nekim slučajevima kada se dogodi greška prilikom izvršavanja programa, potrebno je uraditi neki postupak kojim se stanje programa mora vratiti na konzistentno stanje pre pojave greške. Tipičan primer za ovo su situacije u programu kada su zauzeti neki računarski resursi koji se moraju osloboditi nakom otkrivanja greške. Ukoliko se, na primer, prilikom čitanja ili pisanja neke datoteke na disku dogodi ulazno/izlazna greška zbog koje se ne može nastaviti rad sa datotekom, ova datoteka se mora zatvoriti kako bi se oslobodili računarski resursi koji su u programu rezervisani za rad sa datotekom.

U ovakvim slučajeva se `try-catch` bloku u Javi može dodati još jedan blok u kome se izvršavaju naredbe bez obzira na to da li se greška u bloku `try` dogodila ili ne. Taj blok naredbi nije obavezan i navodi se na kraju `try-catch` bloka iza službene reči `finaly`. Prema tome, najopštiji oblik naredbe `try` ima sledeći izgled:

```
try {
   .
   . // Naredbe koje mogu izazvati izbacivanje izuzetaka
   .
}
catch (. . .) {
   .
   . // Naredbe koje obrađuju izuzetak određenog tipa
```

```
    .
}
.
. // Ostali catch blokovi
.
finally {
    .
    . // Naredbe koje se uvek izvršavaju
    .
}
```

Ako je klauzula `finaly` pridružena `try-catch` bloku, blok `finaly` se uvek izvršava prilikom izvršavanja `try-catch` bloka, bez obzira na to da li je izuzetak izbačen u bloku `try` ili da li je izbačeni izuzetak uhvaćen i obrađen u bloku `catch`. Preciznije, ako nije izbačen nijedan izuzetak u bloku `try`, blok `finaly` se izvršava nakon poslednje naredbe bloka `try`; ako je izbačen neki izuzetak u bloku `try`, blok `finaly` se izvršava nakon izvršavanja odgovarajućeg bloka `catch` ili odmah ukoliko izuzetak nije uhvaćen nijednim blokom `catch`.

Čest primer oslobađanja zauzetih resursa sreće se kod mrežnog programiranja kada je potrebno na početku otvoriti mrežnu konekciju radi komuniciranja sa udaljanim računarom i na kraju zatvoriti tu konekciju. Programski detalji ovih aktivnosti nisu važni, ali robustan način rada programa preko jedne mrežne konekcije može se šematski predstaviti na sledeći način:

```
try {
    // Otvoriti konekciju (i zauzeti računarske resurse)
}
catch (IOException e) {
    // Prijaviti grešku da uspostavljanje veze nije uspelo
    return;  // dalji rad je nemoguć
}

// U ovoj tački je konekcija sigurno otvorena
try {
    // Normalan rad preko otvorene konekcije
}
```

```
catch (IOException e) {
   // Prijaviti ulazno/izlaznu grešku (dalji rad je nemoguć)
}
finally {
   // Zatvoriti konekciju (i osloboditi resurse)
}
```

U ovom fragmentu se prvom naredbom `try` obezbeđuje to da mrežna konekcija bude svakako otvorena kada se komunicira sa udaljenim računarom u nastavku. Klauzulom `finaly` u drugoj naredbi `try` se obezbeđuje to da mrežna konekcija sigurno bude zatvorena, bez obzira na to da li je došlo do prekida veze ili ne u toku normalnog rada preko otvorene konekcije.

Radi potpunosti opisa naredbe `try`, napomenimo još da u njoj nisu ni klauzule `catch` obavezne. Preciznije, naredba `try` se mora pisati bilo sa jednom ili više klauzula `catch`, ili samo sa jednom klauzulom `finaly` ili sa obe od ovih mogućnosti.

Programsko izbacivanje izuzetka

Ponekad sama logika programa može ukazivati da neka grana izvršavanja dovodi do izuzetne okolnosti (greške), ali se na tu okolnost ne može na razuman način reagovati tamo gde se to logički otkriva. U Javi se na takvom mestu može programski izbaciti izuzetak sa ciljem da taj izuzetak bude obrađen u nekom drugom delu programa u kojem se to može smislenije uraditi. Programsko izbacivanje izuzetka postiže se naredbom `throw` koja ima opšti oblik:

```
throw objekat-izuzetka
```

Ovde *`objekat-izuzetka`* mora pripadati klasi `Throwable` ili nekoj njenoj naslednici. (Obično je to naslednica od klasa `Exception`.) Na primer, ukoliko se u nekoj tački programa otkrije da dolazi do deljenja sa nulom, može se izbaciti izuzetak ukoliko nije jasno da li na tom mestu treba potpuno prekinuti izvršavanje ili prekinuti samo aktuelni metod ili nešto treće:

```
throw new ArithmeticException("Deljenje s nulom")
```

Da bi se ukazalo da u telu nekog metoda može doći do izbacivanja izuzetka, zaglavlju metoda se može dodati klauzula `throws`.[1] Na primer, ukoliko prilikom izvršavanja metoda `nekiMetod()` može doći do izbacivanja izuzetka tipa `ArithmeticException`, to se može naznačiti na sledeći način:

```
public void nekiMetod(...) throws ArithmeticException {
   . . .
}
```

Ukoliko u metodu može doći do izbacivanja izuzetka više tipova, ovi tipovi se navode iza klauzule `throws` razdvojeni zapetama.

Mogućnost programskog izbacivanja izuzetaka naredbom `throw` korisna je naročito za pisanje metoda opšte namene koji se mogu koristiti u različitim programima. U tom slučaju, autor metoda ne može na razuman način postupiti sa potencijalnom greškom, jer ne zna tačno kako će se metod koristiti. Naime, ukoliko se samo prikaže poruka o grešci i nastavi izvršavanje metoda, to obično dalje izaziva ozbiljnije greške. Prikazivanje poruke o grešci i prekid izvršavanja programa je skoro isto tako neprihvatljivo, jer se možda u nekom drugom delu programa greška može na zadovoljavajući način amortizovati.

Metod u kome se dogodila greška mora na neki način o tome obavestiti drugi metod koji ga poziva. U programskim jezicima koji nemaju mehanizam izuzetaka, praktično jedini način za ovo je vraćanje neke specijalne vrednosti kojom se ukazuje da metod nije normalno završio svoj rad. Međutim, ovo ima smisla samo ako se u jednom metodu nakon svakog poziva drugog metoda proverava njegova vraćena vrednost. Kako je ove provere često vrlo lako prevideti, izuzeci su efikasnije sredstvo prinude za programere da obrate pažnju na potencijalne greške.

Primer: koreni kvadratne jednačine

Da bismo prethodnu diskusiju potkrepili jednim praktičnim primerom, pretpostavimo da treba napisati metod kojim se rešava kvadratna jednačina opšteg oblika $ax^2+bx+c = 0$. Drugim rečima, za date konstante a, b, c,

[1] Obratite pažnju na razliku između reči `throw` i `throws` — prva se koristi za naredbu, a druga za klauzulu u zaglavlju metoda.

traženi metod kao rezultat treba da vrati realne korene x_1 i x_2 odgovarajuće kvadratne jednačine.

Ukoliko radi jednostavnosti pretpostavimo da se x_1 i x_2 vraćaju kao vrednosti dva javna polja objekta posebne klase `Koren`, jedno robustno rešenje u kojem se otkrivaju sve potencijalne greške jeste sledeći metod:

```
public Koren kvadratnaJednačina(double a, double b, double c)
   throws IllegalArgumentException {

   if (a == 0)
      throw new IllegalArgumentException("Nije kvadratna jednačina")
   else if (b*b-4*a*c < 0)
      throw new IllegalArgumentException("Nisu realni koreni")
   else {
      Koren k = new Koren();
      k.x1 = -b + Math.sqrt(b*b-4*a*c)/(2*a);
      k.x2 = -b - Math.sqrt(b*b-4*a*c)/(2*a);
      return k;
   }
}
```

Primetimo da u ovom metodu nije unapred jasno kako treba postupiti u slučajevima kada se realna rešenja kvadratne jednačine ne mogu izračunati. Naime, pitanje je da li treba odmah prekinuti izvršavanje, ili pre toga treba prikazati poruku o grešci na ekranu, ili treba samo vratiti indikator o grešci, ili treba uraditi nešto sasvim drugo? Zbog toga se u metodu koristi mogućnost programskog izbacivanja izuzetka u nadi da će potencijalni izuzetak obraditi neki rukovalac izuzecima u drugom metodu koji poziva metod `kvadratnaJednačina()`.

7.3 Postupanje sa izuzecima

Usled neke vanredne okolnosti prilikom izvršavanja programa, mogući izvori izbacivanja izuzetka su:

- Java interpretator (JVM) pod čijom se kontrolom izvršava program;
- sam program kada se izvršava naredba `throw` u nekom metodu.

Nezavisno od izvora, nakon izbacivanja izuzetka u jednom delu programa može doći do propagiranja izuzetka i njegovog hvatanja u potpuno drugom delu programa. U Javi se sa izbačenim izuzetkom u oba prethodna slučaja postupa na isti način o kome se detaljnije govori u nastavku.

Pretpostavimo da metod `A` poziva metod `B` i da se u telu metoda `B` izbacuje izuzetak prilikom izvršavanja neke naredbe. Drugim rečima, definicije ovih metoda imaju otprilike sledeći oblik:

```
void A( ... ) {
   . . .
   B( ... )  // poziv metoda B
   . . .
}

void B( ... ) {
   . . .
   greška!   // naredba koja je uzrok izuzetka
   . . .
}
```

Mehanizam rukovanja izuzetkom koji je izbečen u metodu `B` zasniva se najpre na tome da li je izuzetak izbačen unutar nekog `try-catch` bloka. Ako je to slučaj, struktura metoda `B` ima otprilike sledeći oblik:

```
void B( ... ) {
   . . .
   try {
      . . .
      greška!   // naredba koja je uzrok izuzetka
      . . .
   }
   catch ( ... ) {
      . . .
   }
   finally {
      . . .
   }
   . . .
}
```

Ako je dakle izuzetak izbačen unutar nekog `try-catch` bloka u metodu `B`, postupanje sa izuzetkom dalje zavisi od moguća dva slučaja:

1. Izuzetak je uhvaćen nekim `catch` blokom. U ovom slučaju,
   a) izvršava se odgovarajući blok `catch`;
   b) izvršava se blok `finally` ukoliko je naveden;
   c) nastavlja se normalno izvršavanje metoda `B` iza bloka `finally` (ili iza bloka `catch` ukoliko blok `finally` nije naveden).
2. Izuzetak nije uhvaćen nijednim `catch` blokom. Onda se sa izuzetkom postupa na isti način kao da je izuzetak izbačen izvan nekog `try-catch` bloka, o čemu se govori u nastavku.

Primetimo da se zbog ovog drugog slučaja, opis celokupnog mehanizma rukovanja izuzecima može pojednostaviti. Naime, prethodna dva slučaja se kraće razlikuju tako što se kaže da je izbačeni izuzetak uhvaćen ili nije uhvaćen. U prvom slučaju se dakle podrazumeva da je izuzetak izbačen unutar nekog bloka `try` i uhvaćen i obrađen u odgovarajućem bloku `catch`. Drugi slučaj podrazumeva da je ili izuzetak izbačen unutar nekog bloka `try`, ali nije uhvaćen nijednim pridruženim `catch` blokom, ili je izuzetak izbačen izvan bilo kog bloka `try` (i naravno nije uhvaćen nijednim `catch` blokom).

U drugom slučaju kada izbačeni izuzetak nije uhvaćen u metodu `B`, izvršavanje tog metoda se momentalno prekida i metod `A` koji je pozvao metod `B` dobija šansu da obradi izbačeni izuzetak. Pri tome se smatra da je naredba poziva metoda `B` u metodu `A` ona koja je uzrok izbacivanja izuzetka u metodu `A`. Za rukovanje originalnim izuzetkom u metodu `A` primenjuje se isti postupak koji je primenjen na prvobitnom mestu izbacivanja izuzetka u metodu `B`. Naime, naredba poziva metoda `B` u metodu `A` može se nalaziti unutar nekog `try-catch` bloka ili ne. U slučaju da se ona nalazi i da je originalni izuzetak uhvaćen i obrađen, postupanje s njim se završava i nastavlja se normalno izvršavanje metoda `A` iza `try-catch` bloka.

U slučaju da originalni izuzetak nije uhvaćen u metodu `A`, izvršavanje tog metoda se prekida kod naredbe poziva metoda `B` i metod koji je pozvao metod `A` dobija šansu da obradi originalni izuzetak. Opštije, lanac poziva metoda se „odmotava" i pri tome svaki metod u tom lancu dobija šansu da obradi originalni izuzetak ukoliko to nije urađeno u prethodnom metodu.

Prema tome, postupanje sa izbačenim originalnim izuzetkom se nastavlja na isti način tokom vraćanja duž lanca poziva metoda, sve dok se pri tome izuzetak ne obradi u nekom metodu ili dok se ne dođe i prekine izvršavanje početnog metoda `main()`. U ovom drugom slučaju Java interpretator prekida izvršavanje programa i prikazuje standardnu poruku o grešci.

Obavezno i neobavezno rukovanje izuzecima

U Javi su izuzeci generalno podeljeni u dve kategorije prema tome da li programeri u svojim programima moraju obratiti manju ili veću pažnju na mogućnost izbacivanja izuzetaka. To praktično znači da jednu kategoriju izuzetaka čine oni čije rukovanje nije obavezno, dok drugu čine oni čije rukovanje jeste obavezno u programima u kojima se mogu desiti.

Izuzeci čije rukovanje nije obavezno u programima nazivaju se *neproveravani izuzeci* (engl. *unchecked exceptions*), a izuzeci čije rukovanje jeste obavezno u programima nazivaju se *proveravani izuzeci* (engl. *checked exceptions*). Ova, pomalo zbunjujuća, terminologija zapravo govori da li Java kompajler prilikom prevođenja programa proverava postojanje rukovalaca odgovarajuće vrste izuzetaka — za neproveravane izuzetke ne proverava, dok za proveravane izuzetke proverava postojanje njihovih rukovalaca u programu i bez njih ne dozvoljava uspešno prevođenje programa.

U neproveravane izuzetke spadaju izuzeci koji pripadaju klasama `Error`, `RuntimeException` ili nekim njihovim klasama naslednicama. Osnovno svojstvo neproveravanih izuzetaka je dakle da ih programeri mogu (ali ne moraju) zanemariti u svojim programima, jer kompajler ne proverava da li program sadrži rukovaoce ovih izuzetaka. Drugim rečima, za neproveravane izuzetke se u programu ne moraju (ali mogu) pisati rukovaoci tim izuzecima, niti se moraju (ali mogu) pisati klauzule `throws` u zaglavlju metoda u kojima može doći do izbacivanja neproveravanih izuzetaka.

Na primer, budući da je klasa `IllegalArgumentException` izvedena od `RuntimeException`, izuzeci usled nepravilnog argumenta metoda pripadaju kategoriji neproveravanih izuzetaka. Zato se u metodu za izračunavanje korena kvadratne jednačine na strani 263 ne mora pisati klauzula `throws` u zaglavlju tog metoda (ali može, kao što je to tamo navedeno radi jasnoće). Isto tako, greške usled prekoračenja granica nekog niza su neproveravani

izuzeci, pa se manipulisanje elementima nizova u programu ne mora obavljati unutar `try-catch` blokova.

Proveravani izuzeci su oni koji pripadaju klasi `Exception` i njenim naslednicama (osim klase `RuntimeException`). U proveravane izuzetke spadaju greške čija je priroda nastanka izvan kontrole programera (recimo, to su ulazno-izlazne greške), odnosno to su greške koje se potencijalno ne mogu izbeći, ali ih robustan program mora na neki način obraditi. Da bi se zato program u kome se mogu desiti takve greške učinio robustnijim, kompajler proverava da li postoje rukovaoci tih grešaka i ne dozvoljava uspešno prevođenje programa ukoliko to nije slučaj.

Programeri su dakle primorani da obrate pažnju na proveravane izuzetke i moraju ih u programu obraditi na jedan od dva načina:

1. Direktno — pisanjem rukovaoca proveravanog izuzetka, tj. navođenjem naredbe koja može izbaciti proveravani izuzetak unutar bloka `try` i hvatanjem tog izuzetka u pridruženom bloku `catch`.
2. Indirektno — delegiranjem drugom delu programa da rukuje proveravanim izuzetkom, tj. pisanjem klauzule `throws` za proveravani izuzetak u zaglavlju metoda koji sadrži naredbu koja može izbaciti proveravani izuzetak.

U prvom od ovih slučajeva, proveravani izuzetak biće uhvaćen u metodu u kojem je nastao, pa nijedan drugi metod koji poziva izvorni metod neće ni znati da se desio izuzetak. U drugom slučaju, proveravani izuzetak neće biti uhvaćen u metodu u kojem je nastao, a kako se taj izuzetak ne može zanemariti, svaki metod koji poziva izvorni metod mora rukovati originalnim izuzetkom. Zbog toga, ako metod `A` poziva drugi metod `B` sa klauzulom `throws` u svom zaglavlju za proveravani izuzetak, onda se ovaj izuzetak mora obraditi u metodu `A` opet na jedan od dva načina:

1. Pisanjem rukovaoca proveravanog izuzetka u metodu `A`, odnosno navođenjem naredbe poziva metoda `B` koji je potencijalni izvor proveravanog izuzetka unutar bloka `try` i hvatanjem tog izuzetka u pridruženom bloku `catch`.
2. Pisanjem klauzule `throws` za originalni proveravani izuzetak u zaglavlju pozivajućeg metoda `A` radi daljeg delegiranja rukovanja tim izuzetkom.

Da bismo ovu diskusiju ilustrovali primerom, razmotrimo postupak otvaranja datoteke na disku koja se čita u programu. U Javi su ulazne datoteke programa u sistemu datoteka na disku predstavljene (između ostalog) objektima klase `FileInputStream` iz paketa `java.io`. Prema definiciji jedne preopterećene verzije konstruktora ove klase, njegov parametar je ime datoteke koja se otvara za čitanje. Ovaj konstruktor dodatno u svom zaglavlju sadrži klauzulu `throws` u obliku:

```
throws FileNotFoundException
```

Kako je klasa `FileNotFoundException` naslednica klase `Exception` (videti sliku 7.1), to znači da konstruktor klase `FileInputStream` izbacuje proveravani izuzetak kada se datoteka sa datim imenom ne može otvoriti za čitanje. Najčešći uzrok toga je da se tražena datoteke ne nalazi na disku, pa otuda dolazi ime klase `FileNotFoundException` za izuzetke ovog tipa.

Znajući ovo, kao i to da se proveravani izuzeci ne mogu ignorisati u programu, čitanje neke datoteke na disku mora se rešiti na jedan od dva načina. Prvi način je da se u metodu u kojem se obavlja čitanje datoteke, recimo `čitajDatoteku()`, napiše rukovalac izuzetka tipa `FileNotFoundException`. Na primer:

```
void čitajDatoteku (String imeDatoteke) {

   FileInputStream datoteka;
   . . .
   try {
      datoteka = new FileInputStream(imeDatoteke);
   }
   catch (FileNotFoundException e) {
      System.out.println("Datoteka " + imeDatoteke
                                     + " ne postoji");
      return;
   }
   // Naredbe za čitanje datoteke
   . . .
}
```

Drugi način je da se rukovanje izuzetkom tipa `FileNotFoundException` odloži i to prepusti svakom metodu koji koristi metod `čitajDatoteku()` za

čitanje datoteke. Na primer:

```
void čitajDatoteku (String imeDatoteke)
   throws FileNotFoundException {

   FileInputStream datoteka;
   . . .
   datoteka = new FileInputStream(imeDatoteke);
   // Naredbe za čitanje datoteke
   . . .
}
```

7.4 Definisanje klasa izuzetaka

Izuzeci olakšavaju pisanje robustnih programa, jer obezbeđuju strukturiran način za jasno realizovanje glavne programske logike, ali ujedno i za obradu svih izuzetnih slučajeva u `catch` blokovima naredbe `try`. Java sadrži priličan broj unapred definisanih klasa izuzetaka za predstavljanje raznih potencijalnih grešaka. Ove standardne klase izuzetaka treba uvek koristiti kada adekvatno opisuju vanredne situacije u programu.

Međutim, ukoliko nijedna standardna klasa nije odgovarajuća za neku potencijalnu grešku, programeri mogu definisati svoje klase izuzetaka radi tačnijeg predstavljanja te greške. Nova klasa izuzetaka mora biti naslednica klase `Throwable` ili neke njene naslednice. Generalno, ukoliko se ne želi da se zahteva obavezno rukovanje novim izuzetkom, nova klasa izuzetaka se piše tako da proširuje klasu `RuntimeException` ili neku njenu naslednicu. Ali ukoliko se želi obavezno rukovanje novim izuzetkom, nova klasa izuzetaka treba da proširuje klasu `Exception` ili neku njenu naslednicu.

U narednom primeru je definisana klasa izuzetaka koja nasleđuje klasu `Exception` i stoga izuzeci tog tipa zahtevaju obavezno rukovanje u programima u kojima se mogu desiti:

```
public class PogrešanPrečnikIzuzetak extends Exception {

   // Konstruisanje objekta izuzetka
   // koji sadrži datu poruku o grešci
```

```
    public PogrešanPrečnikIzuzetak(String poruka) {
      super(poruka);
    }
}
```

Klasa `PogrešanPrečnikIzuzetak` opisuje greške koje se mogu desiti usled određivanja negativnog prečnika za krugove u radu, recimo, programa koji manipuliše geometrijskim oblicima. Ova klasa sadrži samo konstruktor kojim se konstruiše objekat izuzetka sa datom porukom o grešci. U konstruktoru se naredbom

```
super(poruka);
```

poziva konstruktor nasleđene klase `Exception` kojim se navedena poruka u zagradi prenosi u konstruisani objekat izuzetka. (Klasa `Exception` sadrži konstruktor `Exception(String poruka)` za konstruisanje objekta izuzetka sa navedenom porukom.)

Klasa `PogrešanPrečnikIzuzetak` nasleđuje metod `getMessage()` od nasleđene klase `Exception` (kao uostalom i ostale metode i polja). Tako, ako promenljiva `e` ukazuje na neki objekat tipa `PogrešanPrečnikIzuzetak`, onda se pozivom `e.getMessage()` može dobiti poruka o grešci koja je navedena u konstruktoru klase `PogrešanPrečnikIzuzetak`.

Ali glavna poenta klase `PogrešanPrečnikIzuzetak` je prosto to što ona postoji — činom izbacivanja objekta tipa `PogrešanPrečnikIzuzetak` ukazuje se da se desila greška u programu usled pogrešnog prečnika kruga. Ovo se postiže naredbom `throw` u kojoj se navodi objekat izuzetka koji se izbacuje. Na primer:

```
throw new PogrešanPrečnikIzuzetak("Negativan prečnik kruga");
```

ili

```
throw new PogrešanPrečnikIzuzetak(
            "Nedozvoljena vrednost prečnika: " + prečnik);
```

Ako se naredba `throw` ne nalazi unutar naredbe `try` kojom se hvata greška usled pogrešnog prečnika kruga, onda se za metod koji sadrži takvu naredbu `throw` mora naznačiti da se u njemu može desiti greška usled pogrešnog prečnika kruga. To se postiže pisanjem odgovarajuće klauzule `throws` u zaglavlju tog metoda. Na primer:

```
public void promeniPrečnik(double p)
            throws PogrešanPrečnikIzuzetak {
   . . .
}
```

Primetimo da klauzula `throws` u zaglavlju metoda `promeniPrečnik()` ne bi bila obavezna da je klasa `PogrešanPrečnikIzuzetak` bila definisana tako da bude naslednica klase `RuntimeException` umesto klase `Exception`, jer u tom slučaju rukovanje izuzecima tipa `PogrešanPrečnikIzuzetak` ne bi bilo obavezno.

Radi ilustracije, u nastavku je naveden primer kompletne klase `Krug` u kojoj se koristi nova klasa izuzetaka `PogrešanPrečnikIzuzetak`:

```
public class Krug {

   private double prečnik;

   public Krug(double p)
          throws PogrešanPrečnikIzuzetak {

      promeniPrečnik(p);
   }

   public void promeniPrečnik(double p)
               throws PogrešanPrečnikIzuzetak {

      if (p >= 0)
         prečnik = p;
      else
         throw new PogrešanPrečnikIzuzetak(
                     "Negativan prečnik kruga");
   }

   public double getPrečnik() {

      return prečnik;
   }
```

```
    public double površina() {

        return Math.PI * prečnik * prečnik;
    }
}
```

U programu u kome se koristi ova klasa `Krug`, naredbe koje mogu dovesti do greške tipa `PogrešanPrečnikIzuzetak` treba navesti unutar bloka `try` kojim se rukuje greškama usled pogrešnog prečnika kruga. Na primer:

```
public class TestKrug {

    public static void main(String[] args) {

        try {
            System.out.println("Konstruisanje prvog kruga ...");
            Krug k1 = new Krug(5);
            k1.promeniPrečnik(-1);
            System.out.println("Konstruisanje drugog kruga ...");
            Krug k2 = new Krug(-5);
        }
        catch (PogrešanPrečnikIzuzetak e) {
            System.out.println(e.getMessage());
        }
    }
}
```

U metodu `main()` se u bloku `try` konstruiše prvi krug i zatim se pokušava promena njegovog prečnika u negativnu vrednost. To izaziva grešku tipa `PogrešanPrečnikIzuzetak` kojom se rukuje u pridruženom bloku `catch`. Nakon toga se program završava i primetimo da se time preskače konstruisanje drugog kruga (što bi takođe izazvalo grešku tipa `PogrešanPrečnikIzuzetak`).

GLAVA 8

Programski ulaz i izlaz

Računarski programi su korisni samo ukoliko na neki način razmenjuju podatke sa svojim okruženjem. Pod time se podrazumava dobijanje (čitanje) podataka iz okruženja u program i predavanje (pisanje) podataka iz programa u njegovo okruženje. Ova dvosmerna razmena podataka programa sa okruženjem naziva se *ulaz* i *izlaz* programa (ili kraće *U/I*). U knjizi se do sada govorilo samo o jednoj vrsti ulaza i izlaza programa: interakciji čovek–program preko tekstualnog ili grafičkog interfejsa. Ali čovek je samo jedan primer izvora i korisnika informacija, jer računar može biti povezan sa vrlo različitim ulaznim i izlaznim uređajima. Zbog toga su moguća mnoga druga ulazno-izlazna okruženja za program u koja spadaju datoteke na disku, udaljeni računari, razni kućni uređaji i slično.

U Javi se programski ulaz i izlaz jednoobrazno zasnivaju na apstrakciji toka podataka u program i iz njega . Programi čitaju podatke iz ulaznih tokova i upisuju podatke u izlazne tokove. Ulazno/izlazni tokovi podataka su naravno predstavljeni objektima koji obezbeđuju metode za čitanje i pisanje podataka slične onima koji su već korišćeni u tekstualnim programima u knjizi. U stvari, objekti za standardni ulaz i standardni izlaz (`System.in` i `System.out`) primeri su tokova podataka.

Sve osnovne U/I klase u Javi nalaze se u paketu `java.io`. Od verzije 1.4 dodate su nove klase u paketima `java.nio` i `java.nio.channels` da bi se obezbedile nove mogućnosti i otklonili neki nedostaci osnovnog pristupa. Sa novim klasama uveden je koncept *kanala* kojim se u suštini obezbeđuju dodatne mogućnosti osnovnog načina rada sa tokovima podataka. Omogućeno je i „baferovanje" U/I operacija radi značajnog povećanja njihove

efikasnosti.

U ovom poglavlju se najpre govori o konceptu tokova podataka u Javi kako bi se razumeo opšti mehanizam pisanja i čitanja podataka u programima. Zatim se razmatra jedna primena tog koncepta za pisanje i čitanje datoteka na disku.

8.1 Tokovi podataka

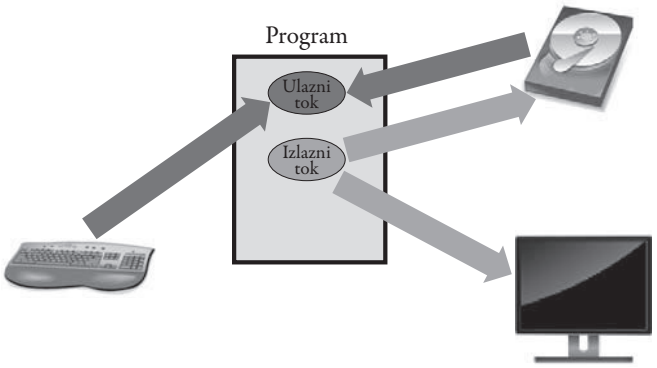
Ulaz i izlaz programa u Javi se konceptualno zasniva na jednoobraznom modelu u kome se U/I operacije obavljaju na konzistentan način nezavisno od konkretnih ulazno-izlaznih uređaja za koje se primenjuju. U centru ovog logičkog modela nalazi se pojam toka podataka kojim se apstrahuju vrlo raznorodni detalji rada svih mogućih U/I uređaja povezanih sa računarom. Na primer, u tekstualnom programu se izlazni tok koristi za ispisivanje podataka u konzolni prozor na ekranu, a ulazni tok se koristi za učitavanje podataka preko tastature.

Tok podataka (engl. *data stream*) je apstraktna reprezentacija prenosa podataka iz nekog izvora podataka u program ili iz programa u neki prijemnik podataka. Tok podataka se vizuelno može zamisliti kao *jednosmerni* i *sekvencijalni* protok bajtova u program ili iz njega. Na slici 8.1 je ilustrovana veza programa, tokova podataka i fizičkih uređaja.

Na jednom kraju ulaznog ili izlaznog toka uvek se nalazi program. Izlaz programa obavlja se tako što se podaci iz programa upisuju u *izlazni tok*. Izlazni tok na drugom kraju može biti povezan sa svakim uređajem kojem se bajtovi podataka mogu serijski preneti. U takve prijemnike spadaju datoteke na disku, udaljeni računari povezani mrežom, konzolni prozor na ekranu ili zvučna kartica povezana sa zvučnicima.[1]

Ulaz programa obavlja se tako što se podaci u programu čitaju iz *ulaznog toka*. Na drugom kraju ulaznog toka može se nalaziti, u principu, svaki serijski izvor bajtova podataka. To su obično datoteke na disku, udaljeni računari ili tastatura.

[1] Štampač je izlazni uređaj koji se ne može koristiti za izlazni tok. Štampač se u Javi smatra grafičkim uređajem i štampanje je konceptualno vrlo slično prikazivanju grafičkih elemenata na ekranu.



Slika 8.1: Programski ulaz i izlaz u Javi.

Binarni i tekstualni tokovi

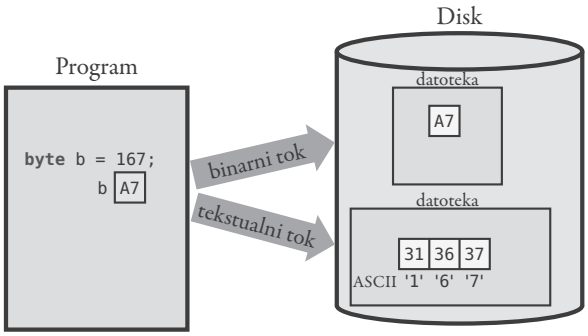
Kada se govori o ulazu i izlazu programa, pre svega treba imati na umu dve široke kategorije podataka koje odražavaju način na koji se oni predstavljaju: binarni podaci i tekstualni podaci. Binarni podaci se predstavljaju u binarnom obliku na isti način na koji se zapisuju unutar računara. Binarni podaci su dakle obični nizovi nula i jedinica čija interpretacija nije odmah prepoznatljiva. Tekstualni podaci se predstavljaju nizovima znakova čije se značenje obično može lako prepoznati. Ali pojedinačni znakovi u tekstualnom podatku se i sami zapisuju u binarnom obliku prema raznim šemama kodiranja. Prema tome, svi podaci u računarima su naravno binarni, ali se tekstualni podaci u ovom kontekstu smatraju da su na višem nivou interpretacije u odnosu na bezlični niz nula i jedinica binarnih podataka.

Da bi se ove dve kategorije ulazno-izlaznih podataka uzele u obzir, u Javi se posledično razlikuju dve vrste tokova: *binarni U/I tokovi* i *tekstualni U/I tokovi*. Za ove dve vrste tokova se primenjuju različiti postupci prilikom pisanja i čitanja podataka. Najvažnija razlika između binarnih i tekstualnih tokova je to što tekstualni tok obezbeđuje automatsko konvertovanje bajtova prilikom njihovog prenosa iz programa u spoljašnje okruženje ili u program iz spoljašnjeg okruženja. Ukoliko se koristi binarni tok, bajtovi se neizmenjeni prenose iz programa ili u njega.

Da bismo bolje razumeli razliku između binarnih i tekstualnih toko-

vima u Javi, razmotrimo jedan konkretan primer izlaza programa: pretpostavimo da se u programu ceo broj 167 tipa `byte` upisuje u datoteku na disku. Broj 167 je u programu zapisan unutar jednog bajta u binarnom obliku `10100111` (ili ekvivalentno `0xA7` u heksadecimalnom zapisu).

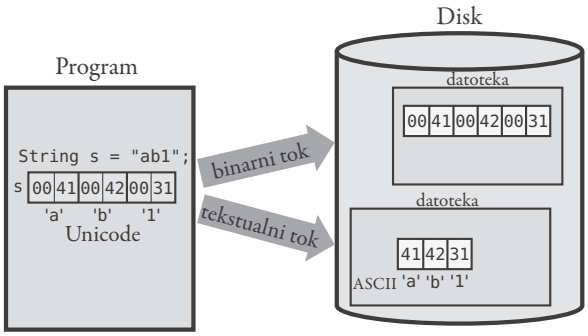
Ukoliko se za pisanje broja 167 koristi binarni tok, jedan bajt odgovarajuće binarne vrednosti `10100111` (`0xA7` heksadecimalno) biće upisan neizmenjen u datoteku. Ali ako se za njegovo pisanje koristi tekstualni tok, onda se broj 167 upisuje kao niz znakova `'1'`, `'6'` i `'7'`. Pri tome se svaki znak u nizu konvertuje specifičnom kodnom šemom u odgovarajući bajt koji se zapravo upisuje u datoteku na disku. Ako se, recimo, koristi ASCII šema, onda znaku `'1'` odgovara jedan bajt čija ASCII vrednost iznosi `0x31` u heksadecimalnom zapisu (ili `00110001` u binarnom obliku), znaku `'6'` odgovara bajt `0x36` i znaku `'7'` odgovara bajt `0x37`. Prema tome, za broj 167 tipa `byte` u datoteku se redom upisuju tri bajta `0x31`, `0x36` i `0x37`. Dva načina ispisivanja numeričke vrednosti u ovom primeru ilustrovana su na slici 8.2. (Sadržaj pojedinih bajtova na toj slici prikazan je u kraćem, heksadecimalnom zapisu.)



Slika 8.2: Upisivanje broja 167 u datoteku.

S druge strane, pretpostavimo da se u programu string `"ab1"` upisuje u datoteku na disku. Ovaj string je u programu zapisan kao niz znakova `'a'`, `'b'` i `'1'`, a svaki znak je predstavljen pomoću dva bajta prema Unicode šemi. Kako znaku `'a'` u ovoj šemi odgovaraju dva bajta `0x0041`, znaku `'b'` odgovaraju dva bajta `0x0042` i znaku `'1'` odgovaraju dva bajta `0x0031`, to je u programu string `"ab1"` predstavljen sa šest bajtova.

Ukoliko se za pisanje stringa `"ab1"` koristi binarni tok, ovih šest bajtova biće upisani neizmenjeni u datoteku. Ali ako se za pisanje koristi tekstualni tok, onda se svaki Unicode znak redom konvertuje na osnovu specifične kodne šeme i odgovarajući bajtovi se upisuje u datoteku na disku. Dakle, ukoliko se koristi podrazumevana ASCII šema za izlaznu datoteku, za string `"ab1"` se u datoteku redom upisuju tri bajta `0x41`, `0x42` i `0x31`, jer u ASCII šemi znakovima `'a'`, `'b'` i `'1'` odgovaraju bajtovi, respektivno, `0x41`, `0x42` i `0x31`. Dva načina ispisivanja stringa u ovom primeru ilustrovana su na slici 8.3, na kojoj je sadržaj pojedinih bajtova opet prikazan u heksadecimalnom zapisu.



Slika 8.3: Upisivanje stringa `"ab1"` u datoteku.

Za ulaz programa primenjuje se sličan princip za binarne i tekstualne tokove prilikom čitanja podataka, samo u obrnutom smeru. Tako, ako se neka datoteka sa ASCII znakovima čita sa diska korišćenjem tekstualnog toka, onda se ASCII bajtovi najpre konvertuju u Unicode znakove, a zatim se ovi Unicode znakovi pretvaraju u odgovarajuću vrednost određenog tipa navedenog u programu.

Binarni tokovi su mnogo efikasniji za ulaz ili izlaz programa, jer se kod njih ne gubi dodatno vreme potrebno za konverziju bajtova prilikom prenosa. To znači i da su podaci upisani korišćenjem binarnog toka portabilni, jer se mogu čitati neizmenjeni na računaru koji je različit od onog na kome su upisani. Ipak, binarni tokovi se mnogo ređe koriste, praktično samo kada se radi o velikim količinama prenosa podataka u specijalnim primenana

(na primer, kod baza podataka ili kod direktnih računar–računar komunikacija). Osnovni nedostatak binarnih tokova je to što su dobijeni binarni podaci nečitljivi za ljude, pa je potreban dodatni napor da bi se oni mogli interpretirati.

Hijerarhija klasa za ulaz i izlaz

Klase u Javi kojima se obezbeđuje ulaz i izlaz programa nalaze se uglavnom u paketu `java.io`. Pored nekih pomoćnih klasa, većina klasa u ovom paketu sačinjava hijerarhiju na čijem vrhu se nalaze četiri apstraktne klase za svaku vrstu tokova podataka:

- `InputStream` za ulazne binarne tokove
- `OutputStream` za izlazne binarne tokove
- `Reader` za ulazne tekstualne tokove
- `Writer` za izlazne tekstualne tokove

Klase koje nasleđuju ove apstraktne klase uglavnom proširuju osnovne mogućnosti rada sa tokovima podataka zavisno od fizičkih uređaja preko kojih se obavlja programski ulaz ili izlaz. U verziji Java 1.4 dodate su nove klase u paketima `java.nio` i `java.nio.channels` da bi se obezbedile nove i povećala efikasnot postojećih U/I operacija.

Sve klase binarnih tokova nasleđuju jednu od apstraktnih klasa `InputStream` ili `OutputStream`. Ako se numeričke vrednosti upisuju pomoću toka tipa `OutputStream`, rezultati će biti u binarnom obliku u kojem su te vrednosti zapisane u memoriji računara. Zbog toga su upisane vrednosti nečitljive za ljude, ali se mogu ispravno učitavati u program pomoću toka tipa `InputStream`. Pisanje i čitanje podataka na ovaj način je vrlo efikasno, jer ne dolazi ni do kakvog konvertovanja bajtova prilikom prenosa.

Sve klase tekstualnih tokova nasleđuju jednu od apstraktnih klasa `Reader` ili `Writer`. Ako se numeričke vrednosti upisuju pomoću toka tipa `Writer`, rezultati se konvertuju u nizove znakova koji predstavljaju te vrednosti u čitljivom obliku. Učitavanje brojeva pomoću toka tipa `Reader` obuhvata obrnuto prevođenje niza znakova u binarni zapis prema tipu odgovarajuće promenljive. Obratite pažnju na to da kod tekstualnih tokova obično dolazi do konvertovanja znakova i prilikom čitanja i pisanja stringova. U tom

slučaju se znakovi zapisani na spoljašnjem medijumu prema jednoj kodnoj šemi (ASCII ili neka šema za neenglesko pismo) konvertuju u Unicode znakove, ili obrnuto.

Osnovne U/I klase `InputStream`, `OutputStream`, `Reader` i `Writer` obezbeđuju samo vrlo primitivne U/I operacije. Na primer, klasa `InputStream` sadrži objektni metod

```
public int read() throws IOException
```

kojim se jedan bajt iz ulaznog binarnog toka čita u obliku celog broja iz opsega od 0 do 255. Ukoliko se naiđe na kraj ulaznog toka prilikom čitanja, metod `read()` kao rezultat vraća vrednost –1. Ako se čitanje ne može obaviti usled neke greške, izbacuje se izuzetak tipa `IOException`. Kako je izuzetak ovog tipa proveravan izuzetak, njime se mora rukovati u programu tako što se metod `read()` navodi unutar naredbe `try` ili se u zaglavlju drugog metoda koji poziva metod `read()` navodi klauzula `throws IOException`.

Za pisanje jednog bajta u izlazni binarni tok, u klasi `OutputStream` nalazi se metod:

```
public void write(int b) throws IOException
```

Primetimo da je parametar ovog metoda tipa `int` a ne `byte`, ali se pre same operacije upisivanja automatski izvršava konverzija tipa ovog parametra u `byte`. To praktično znači da se efektivno zapravo upisuje bajt najmanje težine argumenta metoda `write()`.

Klase `Reader` i `Writer` sadrže analogne metode `read()` i `write()` za čitanje i pisanje tekstualnih tokova, s tom razlikom što ovi metodi čitaju i pišu pojedinačne znakove, a ne bajtove. Vraćena vrednost metoda `read()` je –1 ukoliko se naiđe na kraj ulaznog toka prilikom čitanja, dok se u normalnom slučaju mora izvršiti konverzija tipa te vraćene vrednosti u `char` da bi se dobio pročitani znak.

Naravno, upotreba ovih primitivnih U/I operacija u programima bila bi za programere prilično mukotrpna. Zbog toga se u praksi skoro uvek koriste U/I operacija višeg nivoa koje su realizovane metodima u klasama koje su naslednice četiri osnovne klase na vrhu hijerarhije klasa za ulaz i izlaz programa.

Čitanje i pisanje binarnih podataka

Klase `InputStream` i `OutputStream` omogućavaju čitanje i pisanje samo pojedinačnih bajtova, ali ne i podataka koji su zapisani u složenijem formatu. Da bi se u programima omogućilo čitanje i pisanje vrednosti primitivnih tipova u internom binarnom obliku, u paketu `java.io` se nalaze klase `DataInputStream` i `DataOutputStream` koje su naslednice osnovnih klasa `InputStream` i `OutputStream`.

Klasa `DataOutputStream` sadrži metod `writeDouble(double x)` za pisanje realnih brojeva tipa `double`, metod `writeInt(int x)` za pisanje celih brojeva tipa `int`, i tako dalje za svaki primitivni tip u Javi. Pored toga, objekat tipa `OutputStream` može se „umotati" u objekat tipa `DataOutputStream` da bi se na objekat nižeg nivoa apstrakcije mogli primeniti ovi metodi višeg nivoa. To se postiže prostim navođenjem objekta tipa `OutputStream` da bude argument konstruktora klase `DataOutputStream`. Na primer, ako je promenljiva `b` tipa `OutputStream`, onda se može pisati

```
DataOutputStream a = new DataOutputStream(b);
```

da bi se objekat `b` „umotao" u objekat `a` i tako omogućilo pisanje vrednosti primitivnih tipova u internom formatu na uređaj predstavljen objektom `b` tipa `OutputStream`.

Za čitanje vrednosti primitivnih tipova zapisanih u internom binarnom obliku služi analogna klasa `DataInputStream`. Tako ova klasa sadrži metode `readDouble()`, `readInt()` i tako dalje za čitanje vrednosti svih primitivnih tipova. Slično, objekat tipa `InputStream` može se „umotati" u objekat tipa `DataInputStream` da bi se na objekat nižeg nivoa apstrakcije mogli primeniti ovi metodi višeg nivoa. To se isto postiže prostim navođenjem objekta tipa `InputStream` da bude argument konstruktora klase `DataInputStream`.

U nekim primenama treba čitati znakove iz binarnog toka tipa `InputStream` ili pisati znakove u binarni tok tipa `OutputStream`. Kako za čitanje i pisanje znakova služe klase `Reader` i `Writer`, u takvim slučajevima može se koristiti standardna mogućnost „umotavanja" jednog toka u drugi. Klase `InputStreamReader` i `OutputStreamWriter` namenjene su upravo za ovu mogućnost. Ako su `bIn` promenljiva tipa `InputStream` i `bOut` promenljiva tipa `OutputStream`, onda se naredbama

```
Reader zIn  = new InputStreamReader(bIn);
Writer zOut = new OutputStreamWriter(bOut);
```

konstruišu tekstualni tokovi koji se mogu koristiti za čitanje znakova iz binarnog toka `bIn` i pisanje znakova u binarni tok `bOut`.

Jedan primer ove tehnike može se naći u programima za čitanje znakova iz standardnog ulaza `System.in`, koji u Javi iz istorijskih razloga predstavlja binarni tok tipa `InputStream`. Da bi se zato olakšalo čitanje znakova, objekat standardnog ulaza može se „umotati" u tekstualni tok tipa `Reader`:

```
Reader zIn  = new InputStreamReader(System.in);
```

Drugi primer ove tehnike sreće se u mrežnom programiranju. Ulazni i izlazni tokovi podataka koji su povezani mrežnim konekcijama predstavljeni su binarnim a ne tekstualnim tokovima. Ali, radi lakšeg slanja i primanja tekstualnih podataka preko mreže, binarni tokovi se mogu „umotati" u tekstualne tokove. O mrežnom programiranju se više govori u poglavlju 10.

Čitanje i pisanje tekstualnih podataka

Klasama `InputStream` i `OutputStream` za binarne tokove odgovaraju klase `Reader` i `Writer` za tekstualne tokove, jer klase `Reader` i `Writer` omogućavaju čitanje i pisanje samo pojedinačnih znakova iz skupa Unicode dvobajtnih znakova. Slično kao za binarne tokove, u radu sa tekstualnim tokovima se dodatne mogućnosti za tokove nižeg nivoa apstrakcije mogu obezbediti „umotavanjem" tih tokova u druge tokova višeg nivoa. Dobijeni rezultat je takođe jedan tok podataka i zato se može koristiti za čitanje ili pisanje podataka, ali uz pomoć U/I operacija višeg nivoa.

`PrintWriter`. Klasa `PrintWriter` je naslednica klase `Writer` i sadrži metode za ispisivanje vrednosti svih osnovnih tipova podataka u uobičajenom obliku čitljivom za ljude. U stvari, ove metode smo već upoznali u radu sa standardnim izlazom `System.out`. Naime, ako je `out` promenljiva tipa `PrintWriter`, onda se za pisanje u tok na koji ukazuje ova promenljiva mogu koristiti sledeći metodi:

- `out.print(x)` — vrednost izraza `x` upisuje se u izlazni tok u obliku niza znakova. Izraz `x` može biti bilo kog tipa, primitivnog ili klasnog.

Objekat se reprezentuje nizom znakova koristeći njegov metod `toString()`, a vrednost `null` se reprezentuje nizom znakova `"null"`.

- `out.println()` — znak za kraj reda upisuje se u izlazni tok.
- `out.println(x)` — vrednost izraza `x` i zatim znak za kraj reda upisuju se u izlazni tok. Ovaj efekat je jednak naredbama `out.print(x)` i `out.println()` napisanim jedna iza druge.
- `out.printf(s,x1,x2, ...)` — vrednosti izraza `x1`, `x2` i tako dalje upisuju se na osnovu specifikacije formata `s` u izlazni tok. Prvi parametar `s` je string kojim se precizno opisuje kako se vrednosti argumenata `x1`, `x2` i tako dalje reprezentuju nizovima znakova radi upisivanja u izlazni tok. Mnogobrojna specifikacija formata ove reprezentacije je potpuno ista kao u slučaju standardnog izlaza `System.out`, a zainteresovani čitaoci mogu dodatne informacije o tome potražiti u dokumentaciji Jave.

Obratite pažnju na to da nijedan od ovih metoda ne izbacuje nijedan izuzetak tipa `IOException`. Umesto toga, ukoliko se desi neka greška prilikom izvršavanja nekog od ovih metoda, u klasi `PrintWriter` se interno hvataju svi U/I izuzeci i zatim se oni obrađuju tako što se menja vrednost jednog privatnog indikatora greške. Zbog toga klasa `PrintWriter` sadrži i metod

```
public boolean checkError()
```

kojim se u programu može proveriti vrednost indikatora greške kod pisanja podataka. Na ovaj način je omogućeno da se u programu ne mora na uobičajen način rukovati izuzecima usled izlaznih grešaka, ali u robustnim programima greške treba izbegavati korišćenjem metoda `checkError()`.

Da bi objekat tipa `PrintWriter` poslužio kao omotač za objekat osnovne klase `Writer`, postupa se na standardni način: konstruiše se novi objekat tipa `PrintWriter` koristeći objekat tipa `Writer` kao argument konstruktora. Na primer, ako je `zOut` tipa `Writer`, onda se može pisati:

```
PrintWriter out = new PrintWriter(zOut);
```

Kada se podaci upisuju u izlazni tok `out` korišćenjem nekog od metoda `print()` klase `PrintWriter`, onda se ti podaci zapravo upisuju u izlazni tok

koji je predstavljen objektom `zOut`. To može biti, recimo, datoteka na disku ili udaljeni računar u mreži.

`Scanner`. Možda je interesantno znati da na početku u Javi nije postojala standardna klasa, simetrična klasi `PrintWriter`, koja omogućava lako čitanje vrednosti osnovnih tipova podataka zapisanih u uobičajenom obliku čitljivom za ljude. Tek od verzije Java 5 je u paketu `java.util` dodata klasa `Scanner` koja ovaj problem rešava na relativno zadovoljavajući način.

Problem lakog čitanja ulaznih tekstualnih tokova mučio je naročito programere početnike, jer se za čitanje podataka iz standardnog ulaza moralo primeniti nekoliko „umotavanja" objekata jedan u drugi da bi se došlo do iole prihvatljivog nivoa za ulazne operacije. U paketu `java.io` nalazi se, na primer, standardna klasa `BufferedReader` koja sadrži metod za čitanje jednog celog reda znakova iz ulaznog toka:

```
public String readLine() throws IOException
```

Tako, da bi se iz standardnog ulaza čitao ceo jedan red tekstualnih podataka, objekat `System.in` tipa `InputStream` mora se najpre „umotati" u tok tipa `InputStreamReader`, a ovaj u tok tipa `BufferedReader`:

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

Nakon konstruisanja objekta `in` na ovaj način, u programu se naredbom `in.readLine()` može učitati ceo red teksta iz standardnog ulaza. Ovim postupkom se može čitati red-po-red iz standardnog ulaza, ali se ne obezbeđuje izdvajanje pojedinačnih podataka iz celih redova ulaznog teksta. Za tako nešto, ukoliko je potrebno u programu, moraju se pisati posebni metodi. Ovi problemi, otežani obaveznim rukovanjem izuzetka koji može izbaciti metod `readLine()`, doprineli su lošoj reputaciji ulaznih mogućnosti jezika Java.

Objekat tipa `Scanner` služi kao omotač za ulazni tok podataka. Izvor znakova navodi se u konstruktoru skenera i može biti tipa `InputStream`, `Reader`, `String` ili `File`. (Ako se koristi string za ulazni tok, znakovi stringa se čitaju redom od početka do kraja. O klasi `File` govori se u narednom odeljku.) Na primer, za čitanje podataka preko tastature koja je povezana

sa standardnim ulazom `System.in` tipa `InputStream`, u prethodnim poglavljima je često korišćena klasa `Scanner` na sledeći način:

```
Scanner tastatura = new Scanner(System.in);
```

Slično, ako je `zIn` tok tipa `Reader`, onda se za čitanje tog tekstualnog toka može konstruisati skener na sledeći način:

```
Scanner skener = new Scanner(zIn);
```

Prilikom čitanja toka podataka, skener deli ulazni niz znakova u *tokene*, odnosno grupe znakova koji čine jednu celinu. Tokeni su recimo grupa znakova koji predstavljaju jedan broj tipa `double` ili `int` ili sličnu literalnu vrednost. Podrazumeva se da su tokeni u ulaznom toku razdvojeni znakovima beline (razmaci, tabulatori ili novi redovi) koji se obično preskaču tokom čitanja. Klasa `Scanner` sadrži objektne metode za čitanje tokena različitih tipova:

- `next()` — čita sledeći token koji se kao rezultat vraća u obliku objekta tipa `String`.
- `nextInt()`, `nextDouble()` i tako dalje — čita sledeći token koji se konvertuje u vrednost tipa `int`, `double` i tako dalje. Postoje odgovarajući metodi za čitanje vrednosti svih primitivnih tipova podataka osim `Char`.
- `nextLine()` — čita sve znakove do sledećeg znaka za novi red. Pročitani znakovi se kao rezultat vraćaju u obliku objekta tipa `String`. Obratite pažnju na to da se ovaj metod ne zasniva na čitanje tokena, nego su eventualni znakovi razmaka i tabulatora deo vraćenog stringa. Pored toga, granični znak za novi red se čita, ali nije deo vraćenog stringa.

Ovi metodi u klasi `Scanner` mogu dovesti do izbacivanja izuzetaka. Tako, ukoliko se pokušava čitanje ulaznog toka iza njegovog kraja, izbacuje se izuzetak tipa `NoSuchElementException`. Ili, metod `nextInt()` izbacuje izuzetak tipa `InputMismatchException` ukoliko sledeći token ne predstavlja celobrojnu vrednost. Svi ovi izuzeci pripadaju kategoriji neproveravanih izuzetaka i zato ne zahtevaju obavezno rukovanje u programima.

U klasi `Scanner` se nalaze i metodi kojima se može proveravati da li se čitanjem došlo do kraja ulaznog toka, kao i to da li je sledeći token specifič-

nog tipa:

- `hasNext()` — vraća logičku vrednost tačno ili netačno zavisno od toga da li se u ulaznom toku nalazi bar još jedan token.
- `hasNextInt()`, `hasNextDouble()` i tako dalje — vraća logičku vrednost tačno ili netačno zavisno od toga da li sledeći token u ulaznom toku predstavlja vrednost specifičnog tipa.
- `hasNextLine()` — vraća logičku vrednost tačno ili netačno zavisno od toga da li se u ulaznom toku nalazi bar još jedan red znakova.

Čitanje i pisanje objekata

`Scanner`/`PrintWriter` i `DataInputStream`/`DataOutputStream` su klase u Javi koje obezbeđuju jednostavan ulaz/izlaz za vrednosti primitivnih tipova podataka. Ali budući da su programi u Javi objektno orijentisani, postavlja se pitanje kako se obavlja ulaz/izlaz za objekte?

Bez ugrađenog mehanizma u programskom jeziku, programski izlaz objekata mora se uraditi na indirektan način. To se svodi najpre na pronalaženje neke posebne forme predstavljanja objekata u programu preko vrednosti primitivnih tipova i zatim pisanja ovih vrednosti u binarni ili tekstualni tok. U obrnutom smeru, za programski ulaz objekata onda treba čitati odgovarajuće vrednosti primitivnih tipova i rekonstruisati originalne objekte. Ovaj postupak kodiranja i dekodiranja objekata primitivnim vrednostima radi pisanja i čitanja objekata u programu naziva se *serijalizacija* i *deserijalizacija* objekata.

Serijalizacija i deserijalizacija za kompleksne objekte nije nimalo jednostavan zadatak. Zamislimo samo šta bi sve u jednom grafičkom programu moralo da se uradi za serijalizaciju najprostijeg objekta tipa `JButton`. Naime, morale bi da se sačuvaju trenutne vrednosti svih atributa tog objekta: boja, font, oznaka, pozicija i tako dalje. Ako je neki atribut klasnog tipa (kao što je pozadina tipa `Color`), moraju se sačuvati i vrednosti atributa tog objekta. To nažalost nije kraj, jer ako je klasa naslednica druge klase (kao što je klasa `JButton` naslednica klase `AbstractButton`), onda se moraju čuvati i vrednosti atributa koji pripadaju nasleđenom delu objekta.

Srećom, u Javi postoji ugrađeni mehanizam za čitanje i pisanje objekata koji sledi opšte principe tokova podataka. Za ulaz i izlaz objekata sa au-

tomatskom serijalizacijom i deserijalizacijom koriste se klase `ObjectInputStream` i `ObjectOutputStream`. Klasa `ObjectInputStream` je naslednica klase `InputStream`, dok je analogno klasa `ObjectOutputStream` naslednica klase `OutputStream`. Tokovi tipa `ObjectInputStream` i `ObjectOutputStream` za čitanje i pisanje objekata konstruišu se kao omotači binarnih tokova tipa `InputStream` i `OutputStream` korišćenjem sledećih konstruktora:

```
public ObjectInputStream(InputStream bIn)
public ObjectOutputStream(OutputStream bOut)
```

U klasi `ObjectInputStream` za čitanje jednog objekta služi metod:

```
public Object readObject()
```

Slično, u klasi `ObjectOutputStream` za pisanje jednog objekta služi metod:

```
public void writeObject(Object o)
```

Svaki od ova dva metoda može izbaciti neke proveravane izuzetke i zato se pozivi ovih metoda moraju nalaziti unutar dela programa u kojima se tim izuzecima rukuje na odgovarajući način. Primetimo i da je vraćena vrednost metoda `readObject()` tipa `Object`, što znači da se ona obično mora eksplicitno konvertovati u drugi, korisniji klasni tip u programu.

U klasi `ObjectOutputStream` nalaze se i metodi za pisanje vrednosti primitivnih tipova u binarni tok kao što su `writeInt()`, `writeDouble()` i tako dalje. Slično, u klasi `ObjectInputStream` nalaze se i odgovarajući metodi za čitanje vrednosti primitivnih tipova. U stvari, klase `ObjectInputStream` i `ObjectOutputStream` po funkcionalnosti potpuno zamenjuju klase `DataInputStream` i `DataOutputStream`, tako da se ovaj drugi par klasa može zanemariti i koristiti samo prvi par.

Klase `ObjectInputStream` i `ObjectOutputStream` mogu se koristiti za čitanja i pisanje samo onih objekata koji implementiraju specijalni interfejs `Serializable`. To ipak nije veliko ograničenje, jer interfejs `Serializable` zapravo ne sadrži nijedan metod. Ovaj interfejs služi samo kao marker kojim se kompajleru ukazuje da su objekti implementirajuće klase predviđeni za ulaz/izlaz i da za njih zato treba dodati ugrađeni mehanizam za automatsku serijalizaciju i deserijalizaciju. U definiciji klase čiji su objekti predviđeni za ulaz/izlaz ne treba dakle implementirati nikakve dodatne metode, nego samo u zaglavlju definicije te klase treba dodati klauzulu `im-`

`plements Serializable`. Mnoge standardne klase u Javi su definisane sa ovom klauzulom kako bi se njihovi objekti mogli lako čitati i pisati u programima.

Primenom klasa `ObjectInputStream` i `ObjectOutputStream` za čitanje i pisanje objekata koriste se binarni tokovi. Drugim rečima, objekti se predstavljaju u binarnom obliku nečitljivom za ljude tokom čitanja i pisanja. To je dobro zbog efikasnosti, ali ima i svojih nedostataka. Prvo, binarni format objekata je specifičan za trenutnu verziju Jave koji se može promeniti u nekoj budućoj verziji. Drugo, taj format nije obično isti u drugim programskim jezicima, pa objekti upisani na spoljašnji medijum u Java programu ne mogu se lako čitati u programima koji su napisani na nekom drugom jeziku.

Zbog ovih razloga, binarni tokovi za čitanje i pisanje objekata korisni su samo za privremeno čuvanje objekata na spoljašnjim medijumima ili za prenos objekata iz jednog Java programa preko mreže drugom Java programu. Tekstualni tokovi su korisniji za trajnije čuvanje objekata ili za prenos objekata programima koji nisu napisani u Javi. U tom slučaju koriste se standardizovani načini za predstavljanje objekata u tekstualnom obliku (na primer, XML format).

8.2 Datoteke

Vrednosti koje su u toku rada programa nalaze u promenljivim, nizovima i objektima jesu privremenog karaktera i gube se po završetku programa. Da bi se te vrednosti trajno sačuvale, one se mogu upisati na neki spoljašnji uređaj permanentne memorije koja se deli u celine koje se zovu *datoteke* (engl. *file*). Datoteka predstavlja posebnu grupu podataka na disku, USB flešu, DVD-ju ili na nekom drugom tipu spoljašnjeg uređaja permanentne memorije. Pored toga što služe za trajno čuvanje podataka, datoteke se mogu prenositi na druge računare i kasnije čitati drugim programima.

Sistem datoteka na spoljašnjem uređaju podeljen je u *direktorijume* (ili *foldere*) koji sadrže datoteke, ali i druge direktorijume. Datoteke i direktorijumi se raspoznaju po jedinstvenim imenima kojima se nazivaju prilikom formiranja.

Programi mogu čitati podatke iz postojećih datoteka i upisivati podatke u postojeće ili nove datoteke. Programski ulaz i izlaz u Javi se standardno obavlja putem tokova podataka. Vrednosti iz programa se zapisuju u tekstualnom obliku u datoteci pomoću tekstualnog toka tipa `FileWriter`, a takve vrednosti zapisane u tekstualnom obliku u datoteci čitaju se pomoću tekstualnog toka tipa `FileReader`. Klasa `FileWriter` je naslednica klase `Writer` i klasa `FileReader` je naslednica klase `Reader`. Podsetimo se da se apstraktne klase `Reader` i `Writer` nalaze na vrhu hijerarhije klasa kojima se predstavljaju tekstualni tokovi podataka.

Čitanje i pisanje podataka u binarnom formatu za datoteke u Javi se obavlja uz pomoć klasa `FileInputStream` i `FileOutputStream` koje su naslednice apstraktnih klasa `InputStream` i `OutputStream`. U ovom odeljku, međutim, govori se samo o tekstualnom ulazu/izlazu za datoteke, jer je primena klasa `FileInputStream` i `FileOutputStream` potpuno analogna radu sa klasama `FileReader` i `FileWriter`. Sve ove klase se nalaze u paketu `java.io`.

Čitanje i pisanje datoteka

U klasi `FileReader` nalazi se konstruktor kojim se konstruiše ulazni tekstualni tok za čitanje podataka iz datoteke. Ime datoteke koja se želi čitati navodi se kao argument konstruktora. Ukoliko datoteka sa tim imenom ne postoji, konstruktor izbacije izuzetak tipa `FileNotFoundException` čije je rukovanje u programu obavezno.

Pretpostavimo da se na disku nalazi datoteka `ulaz.txt` sa podacima koje treba čitati u programu. U sledećem programskom fragmentu se konstruiše ulazni tok za tu datoteku:

```
FileReader ulaz;  // ulazni tok

// Konstruisanje ulaznog toka za datoteku ulaz.txt
try {
   ulaz = new FileReader("ulaz.txt");
}
catch (FileNotFoundException e) {
   .
```

```
    . // Rukovanje izuzetkom
    .
}
```

Obratite pažnju u ovom fragmentu na to da je promenljiva `ulaz` definisana izvan naredbe `try`. To je neophodno, jer bi inače ta promenljiva bila lokalna za blok `try` i ne bi se mogla koristiti izvan njega. Pored toga, kako je klasa `FileNotFoundException` naslednica klase `IOException`, u prethodnoj klazuli `catch` se može navesti ova roditeljka klasa. Opštije, skoro svaka specifična U/I greška može se hvatati u klazuli `catch` pisanjem najopštijeg tipa greške `IOException`.

Nakon uspešnog konstruisanja ulaznog toka tipa `FileReader` za datoteku `ulaz.txt`,[2] podaci iz ove datoteke mogu se redom čitati. Ali pošto klasa `FileReader` sadrži samo primitivne metode nasleđene od klase `Reader` za čitanje podataka, radi komfornijeg rada obično se koristi objekat tipa `Scanner` da bude omotač objekta tipa `FileReader`:

```
Scanner ulaz;  // ulazni tok

// Konstruisanje omotača ulaznog toka za datoteku ulaz.txt
try {
   ulaz = new Scanner(new FileReader("ulaz.txt"));
}
catch (FileNotFoundException e) {
   .
   . // Rukovanje izuzetkom
   .
}
```

Na ovaj način se datoteka `ulaz.txt` na disku može čitati korišćenjem naredbi `ulaz.nextInt()`, `ulaz.nextLine()` i slično, odnosno potpuno isto kako se koristi skener za čitanje vrednosti iz standardnog ulaza preko tastature.

Pisanje podataka u datoteku je vrlo slično čitanju podataka i ne zahteva neki dodatni napor. Izlazni tok za pisanje u datoteku konstruiše se da bude objekat tipa `FileWriter`. Pošto konstruktor klase `FileWriter` može izbaciti proveravan izuzetak tipa `IOException`, konstruisanje odgovaraju-

[2] Konstruisanje ulaznog toka za datoteku naziva se kolokvijalno *otvaranje datoteke*.

ćeg objekta izvodi se obično unutar naredbe `try`. Isto tako, radi lakšeg pisanja podataka analogno se koristi objekat tipa `PrintWriter` da bude omotač objekta tipa `FileWriter`. Na primer, ukoliko podatke iz programa treba upisati u datoteku `izlaz.txt`, to se može uraditi otprilike na sledeći način:

```
PrintWriter izlaz;  // izlazni tok

// Konstruisanje omotača izlaznog toka za datoteku izlaz.txt
try {
   izlaz = new PrintWriter(new FileWriter("izlaz.txt"));
}
catch (IOException e) {
   .
   . // Rukovanje izuzetkom
   .
}
```

Ukoliko na disku ne postoji datoteka pod imenom `izlaz.txt`, obrazuje se nova datoteka i u nju se upisuju podaci. Ukoliko datoteka sa ovim imenom već postoji, njen trenutni sadržaj biće obrisan bez upozorenja i biće zamenjen podacima koji se u programi ispisuju. Ovaj često neželjeni efekat može se izbeći prethodnom proverom da li izlazna datoteka već postoji na disku. (O ovome se detaljnije govori u narednom odeljku.) Greška tipa `IOException` može se dogoditi prilikom konstruisanja objekta izlaznog toga ukoliko, recimo, program nema prava da formira datoteke u aktuelnom direktorijumu ili je ceo disk zaštićen od pisanja (engl. *write-protected*).

Nakon završetka rada sa datotekom u programu, datoteku treba *zatvoriti* pozivom metoda `close()` za odgovarajući tok. Zatvorena datoteka se više ne može koristiti za čitanje ili pisanje, osim ako se ponovo ne otvori za novi tok podataka. Ukoliko se datoteka ne zatvori eksplicitno metodom `close()`, ona se automatski zatvara od strane JVM kada se program završi. Ovo ipak ne znači da operaciju zatvaranja datoteke ne treba uzeti ozbiljno, iz dva razloga.

Prvo, memorijski resursi rezervisani u programu za rad sa datotekom biće zauzeti za sve vreme rada programa, iako se možda mogu osloboditi mnogo pre kraja rada programa. Drugi, ozbiljniji nedostatak je to što u slučaju izlaznih datoteka neki podaci mogu biti izgubljeni. To se dešava zato

što podaci koji se u programu pišu u datoteku obične se radi efikasnosti ne upisuju odmah u nju, nego se privremeno upisuju u oblast glavne memorije koja se naziva *bafer*. Tek kad se popuni ceo bafer, podaci se iz njega fizički ispisuju u datoteku. Zato ukoliko se datoteka nasilno zatvori na kraju rada programa, neki podaci u delimično popunjenom baferom mogu ostati fizički neupisani u datoteci. Jedan od zadataka metoda `close()` za izlazni tok je upravo to da se svi podaci iz bafera upišu u datoteku. Primetimo i da ako je u programu potrebno isprazniti bafer bez zatvaranja datoteke, onda se za svaki izlazni tok može koristiti metod `flush()`.

Primer: sortiranje datoteke brojeva

U ovom kompletnom primeru se ilustruju osnovne tehnike rada sa datotekama u Javi o kojima smo govorili u prethodnim odeljcima. Pretpostavlja se da se na disku nalazi datoteka `podaci.txt` sa brojevima u proizvoljnom redosledu. Zadatak programa u listingu 8.1 je da brojeve iz te datoteke najpre učita u jedan niz, zatim sortira taj niz i, na kraju, upiše vrednosti rezultujućeg sortiranog niza u drugu datoteku `sort-podaci.txt`.

Listing 8.1: Sortiranje datoteke brojeva

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

public class SortiranjeDatoteke {

   public static void main(String[] args) {
      Scanner ulaz;       // ulazni tekstualni tok za čitanje
      PrintWriter izlaz;  // izlazni tekstualni tok za pisanje
      double[] brojevi = new double[1000];  // niz učitanih brojeva
      int k = 0;  // ukupan broj učitanih brojeva

      // Konstruisanje ulaznog toka za datoteku podaci.txt
      try {
         ulaz = new Scanner(new FileReader("podaci.txt"));
      }
      catch (FileNotFoundException e) {
```

```
   System.out.print("Ulazna datoteka podaci.txt ");
   System.out.println("nije nađena!");
   return;        // kraj rada programa zbog greške
}

// Konstruisanje izlaznog toka za datoteku sort-podaci.txt
try {
   izlaz = new PrintWriter(new FileWriter("sort-podaci.txt"));
}
catch (IOException e) {
   System.out.print("Otvaranje izlazne datoteke ");
   System.out.println("sort-podaci.txt nije uspelo!");
   System.out.println("Greška: " + e);
   ulaz.close();  // zatvoriti ulaznu datoteku
   return;        // kraj rada programa zbog greške
}

try {
   // Učitavanje brojeva iz ulazne datoteke u niz
   while (ulaz.hasNext()) {  // sve dok ima sledećeg broja
      brojevi[k] = ulaz.nextDouble();  // . . . učitati ga
      k = k + 1;
   }

   // Sortiranje učitanih brojeva u nizu u opsegu od 0 do k-1
   Arrays.sort(brojevi, 0, k);

   // Upisivanje sortiranih brojeva u izlaznu datoteku
   for (int i = 0; i < k; i++)
      izlaz.println(brojevi[i]);

   System.out.println("Ulazna datoteka je sortirana.");
}
catch (RuntimeException e) {
   System.out.println("Problem u radu sa nizom!");
   System.out.println("Greška: " + e.getMessage());
}
```

```
      catch (Exception e) {
         System.out.println("Problem pri čitanju/pisanju podataka!");
         System.out.println("Greška: " + e.getMessage());
      }
      finally {
         // Zatvaranje obe datoteke, u svakom slučaju
         ulaz.close();
         izlaz.close();
      }
   }
}
```

U ovom programu se pre samog čitanja i pisanja podataka najpre otvaraju ulazna i izlazna datoteka. Ukoliko se bilo koja od ovih datoteka ne može otvoriti, nema smisla nastaviti s radom i program se zato završava u rukovaocima ove greške uz prethodno prikazivanje odgovarajuće poruke.

Ako je sve u redu, u programu se redom učitavaju brojevi u jedan niz, taj niz se sortira u rastućem redosledu njegovih vrednosti i na kraju se ove vrednosti sortiranog niza redom upisuju u izlaznu datoteku. Ovaj postupak se izvršava unutar bloka `try` kako bi se otkrile moguće greške u radu sa nizom ili datotekama. Na primer, ukoliko ulazna datoteka sadrži više od 1000 brojeva kolika je dužina niza, doći će do izbacivanja izuzetka usled prekoračenja granice nize. Ova greška se onda hvata u prvoj klauzuli `catch` iza greške opšteg tipa `RuntimeException`.

Primetimo i praktičnu korisnost klauzule `finally` u programu, jer se naredbe unutar bloka `finally` obavezno izvršavaju nezavisno od toga da li je u bloku `try` došlo do neke greške ili ne. Zbog toga se sa sigurnošću može obezbediti da obe datoteke budu pravilno zatvorene na kraju programa.

8.3 Imena datoteka

Neka datoteka kao celina za grupu određenih podataka može se zapravo posmatrati iz dva ugla: jedan aspekt datoteke je njeno ime i drugi aspekt je njen sadržaj. Sadržaj datoteke čine njeni podaci koji se mogu čitati i upi-

sivati U/I operacijama o kojima je bilo reči u prethodnom delu poglavlja. Ime datoteke je do sada uzimano zdravo za gotovo kao običan niz znakova, ali to je ipak malo složeniji pojam. Zbog toga je u Javi na raspolaganju klasa `File` koja služi za manipulisanje imenima datoteka u sistemu datoteka na spoljašnjem uređaju nezavisno od konkretnih detalja.

Konceptualno, svaka datoteka se nalazi u nekom direktorijumu u sistemu datoteka. Jednostavno ime datoteke kao što je `podaci.txt` odnosi se na datoteku koja se nalazi u takozvanom *aktuelnom direktorijumu* (ili „radnom direktorijumu" ili „podrazumevanom direktorijumu"). Svaki program na početku izvršavanja ima pridružen aktuelni direktorijum koji je obično onaj u kome se nalazi sama datoteka sa bajtkodom programa. Aktuelni direktorijum međutim nije stalno svojstvo programa i može se promeniti tokom izvršavanja, ali prosta imena datoteka uvek ukazuju na datoteke u aktuelnom direktorijumu, ma koji on trenutno bio. Ukoliko se koristi datoteka koja nije u aktuelnom direktorijumu, mora se navesti njeno puno ime u obliku koji obuhvata njeno prosto ime i direktorijum u kome se datoteka nalazi.

Puno ime datoteke, u stvari, može se pisati na dva načina. *Apsolutno ime datoteke* jedinstveno ukazuje na datoteku među svim datotekama u sistemu datoteka. Kako je sistem datoteka hijerarhijski organizovan po direktorijumima, apsolutno ime datoteke određuje mesto datoteke tako što se navodi putanja niza direktorijuma koja vodi od početnog direktorijuma na vrhu hijerarhije do same datoteke. Sa druge strane, *relativno ime datoteke* određuje mesto datoteke tako što se navodi (obično kraća) putanja niza direktorijuma koja vodi od aktuelnog direktorijuma do same datoteke.

Način pisanja apsolutnog i relativnog imena datoteka zavisi donekle od operativnog sistema računara, pa to treba imati u vidu prilikom pisanja programa kako bi se oni mogli izvršavati na svakom računaru. Razlike u pisanju imena datoteka najbolje je pokazati na primerima:

- `podaci.txt` je prosto ime datoteke u svakom sistemu. Ovim načinom pisanja se određuje da se datoteka nalazi u aktuelnom direktorijumu.
- `/home/dzivkovic/java/podaci.txt` je apsolutno ime datoteke u sistemu UNIX. Ovim načinom pisanja se određuje kompletan niz direktorijuma koje treba preći da bi se od početnog direktorijuma do-

šlo do datoteke sa prostim imenom `podaci.txt`. Drugim rečima, ta datoteka se nalazi u direktorijumu `java`. Taj direktorijum se sa svoje strane nalazi u direktorijumu `dzivkovic`; direktorijum `dzivkovic` se nalazi u direktorijumu `home`; a direktorijum `home` se nalazi u početnom direktorijumu.

- `C:\users\dzivkovic\java\podaci.txt` je apsolutno ime datoteke u sistemu Windows. Mesto datoteke se određuje na sličan način kao u prethodnom slučaju. Primetimo da se u sistemu Windows imena direktorijuma u nizu razdvajaju znakom obrnute kose crte, a ne znakom kose crte.
- `java/podaci.txt` je relativno ime datoteke u sistemu UNIX. Ovim načinom pisanja se određuje putanja niza direktorijuma koju treba proći da bi se od aktuelnog direktorijuma došlo do datoteke sa prostim imenom `podaci.txt`. Drugim rečima, ta datoteka se nalazi u direktorijumu `java`, a taj direktorijum se nalazi u aktuelnom direktorijumu. U sistemu Windows, isto relativno ime datoteke zapisuje se na sličan način: `java\podaci.txt`.

Problem drugačijeg pisanja imena datoteka u različitim sistemima nije toliko izražen u grafičkim programima, jer umesto pisanja imena datoteke korisnik može željenu datoteku izabrati na lakši način putem grafičkog interfejsa. O ovome se više govori u nastavku poglavlja.

U programu je moguće dinamički saznati apsolutna imena dva posebna direktorijuma koja su važna za određivanje mesta datoteka: aktuelni direktorijum i korisnikov matični direktorijum. Ova imena pripadaju grupi *sistemskih svojstava* koja se mogu dobiti kao rezultat metoda `getProperty()` iz klase `System`:

- `System.getProperty("user.dir")` — Rezultat je string koji sadrži apsolutno ime aktuelnog direktorijuma.
- `System.getProperty("user.home")` — Rezultat je string koji sadrži apsolutno ime matičnog direktorijuma korisnika.

Na primer, izvršavanjem dve naredbe

```
System.out.println(System.getProperty("user.dir"));
System.out.println(System.getProperty("user.home"));
```

na ekranu se kao rezultat dobijaju dva stringa za aktuelni i matični direktorijum u trenutku izvršavanja tih naredbi:

```
D:\My Stuff\Java\Programs\TestDir
C:\Users\Dejan Zivkovic
```

Klasa `File`

Ime datoteke je u Javi konceptualno predstavljeno objektom koji pripada klasi `File` iz paketa `java.io`. Sama datoteka (tj. njen sadržaj) na koju se njeno ime odnosi može, ali i ne mora postojati na spoljašnjem uređaju. U klasi `File` se imena direktorijuma logički ne razlikuju od imena datoteka tako da se obe vrste imena uniformno tretiraju. Klasa `File` sadrži metode za određivanje raznih svojstava datoteka/direktorijuma, kao i metode za preimenovanje i brisanje datoteka/direktorijuma.

Klasa `File` sadrži jedan konstruktor kojim se objekat te klase konstruiše od imena tipa `String`:

```
public File(String ime)
```

Argument ovog konstruktora može biti prosto ime, relativno ime ili apsolutno ime. Na primer, izrazom `new File("podaci.txt")` konstruiše se objekat tipa `File` koji ukazuje na datoteku sa imenom `podaci.txt` u aktuelnom direktorijumu.

Drugi konstruktor klase `File` služi za konstruisanje objekta te klase koji ukazuje na relativno mesto datoteke:

```
public File(File dir, String ime)
```

Prvi parametar ovog konstruktora predstavlja ime direktorijuma u kome se nalazi datoteka. Drugi parametar predstavlja prosto ime datoteke u tom direktorijumu ili relativnu putanju od tog direktorijuma do datoteke.

Klasa `File` sadrži više objektnih metoda kojima se obezbeđuje lako manipulisanje datotekama u sistemu datoteka na disku. Ako je `datoteka` promenljiva tipa `File`, onda se najčešće koriste ovi od tih metoda:

- `datoteka.exists()` — Rezultat je logička vrednost tačno ili netačno zavisno od toga da li postoji datoteka sa imenom predstavljenim objektom klase `File` na koji ukazuje promenljiva `datoteka`.

- `datoteka.isDirectory()` — Rezultat je logička vrednost tačno ili netačno zavisno od toga da li objekat klase `File` na koji ukazuje promenljiva `datoteka` predstavlja direktorijum.
- `datoteka.delete()` — Briše se datoteka (ili direktorijum) iz sistema datoteka, ukoliko postoji. Rezultat je logička vrednost tačno ili netačno zavisno od toga da li je operacija uklanjanja uspešno izvršena.
- `datoteka.list()` — Ako promenljiva `datoteka` ukazuje na neki direktorijum, onda je rezultat niz tipa `String[]` čiji su elementi imena datoteka (i direktorijuma) u tom direktorijumu. U suprotnom slučaju se kao rezultat vraća vrednost `null`.

Sve klase koje služe za čitanje i pisanje datoteka imaju konstruktore čiji je parametar tipa `File`. Na primer, ako je `datoteka` promenljiva tipa `File` koja ukazuje na ime tekstualne datoteke koju treba čitati, onda se objekat tekstualnog toka tipa `FileReader` za njeno čitanje može konstruisati izrazom `new FileReader(datoteka)`.

Primer: lista svih datoteka u direktorijumu

U listingu 8.2 je prikazan primer kompletnog programa kojim se prikazuje lista svih datoteka i direktorijuma u zadatom direktorijumu.

Listing 8.2: Lista svih datoteka u direktorijumu

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

public class ListDir {

   public static void main(String[] args) {

      String imeDir;      // ime direktorijuma koje korisnik zadaje
      Scanner tastatura;  // ... i program učitava preko tastature
      File dir;           // objekat tipa File za taj direktorijum
      String[] datoteke;  // niz imena datoteka u tom direktorijumu

      tastatura = new Scanner(System.in);
```

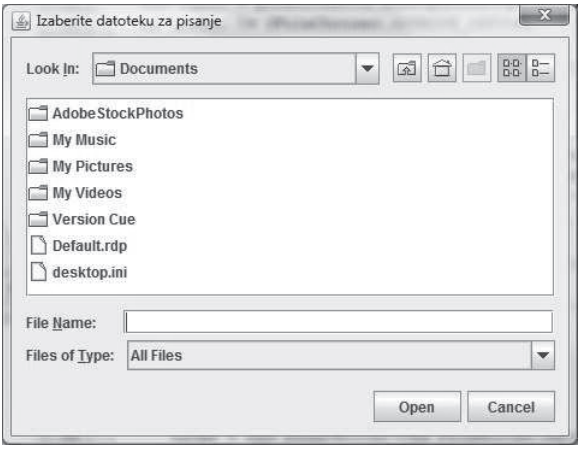
```
      System.out.print("Unesite ime direktorijuma: ");
      imeDir = tastatura.nextLine().trim();

      dir = new File(imeDir);
      if (!dir.exists())
         System.out.println("Takav direktorijum ne postoji!");
      else if (!dir.isDirectory())
         System.out.println("To nije direktorijum!");
      else {
         datoteke = dir.list();
         System.out.print("Datoteke u direktorijumu \"");
         System.out.println(dir + "\" su:");
         for (int i = 0; i < datoteke.length; i++)
            System.out.println("  " + datoteke[i]);
      }
   }
}
```

Izbor datoteka u grafičkim programima

U radu sa datotekama je skoro uvek potrebno obezbediti izbor željenog imena datoteke/ direktorijuma za ulaz ili izlaz programa. Na primer, u programu u listingu 8.2 iz prethodnog odeljka se preko tastature zadaje ime direktorijuma čiji se sadržaj prikazuje na ekranu. Nedostatak ovog načina izbora datoteka u tekstualnim programima, pored toga što je to podložno greškama, jeste to da korisnik mora poznavati detalje pisanja imena datoteka za konkretni računar.

U grafičkim programima se izbor datoteka može obezbediti na mnogo lakši način za korisnika putem posebnih komponenti. U Javi se grafički dijalog za izbor datoteke predstavlja klasom `JFileChooser` u biblioteci Swing. U prozoru za izbor datoteke se prikazuje lista datoteka i direktorijuma u nekom direktorijumu. Korisnik iz tog prozora onda lako može mišem izabrati jednu od prikazanih datoteka ili direktorijuma ili prelaziti iz jednog direktorijuma u drugi. Na slici 8.4 je prikazan izgled tipičnog prozora za izbor datoteke.



Slika 8.4: Grafički dijalog za izbor datoteke.

Najčešće korišćeni konstruktor klase `JFileChooser` je onaj koji nema nijedan parametar i onda se u prozoru za izbor datoteke početno prikazuje sadržaj korisnikovog matičnog direktorijuma:

```
public JFileChooser()
```

Druga dva konstruktora omogućavaju da se u prozoru za izbor datoteke prikaže početni direktorijum koji je različit od matičnog direktorijuma:

```
public JFileChooser(File dir)
public JFileChooser(String dir)
```

Konstruisani objekat klase `JFileChooser` koji predstavlja grafički prozor za izbor datoteke ne prikazuje se odmah na ekranu, nego se mora pozvati metod kojim se to postiže u željenoj tački programa. U stvari, postoje dva metoda za prikazivanje prozora za izbor datoteke, jer postoje dve vrste takvih prozora: prozor za otvaranje datoteke (engl. *open file dialog*) i prozor za čuvanje datoteke (engl. *save file dialog*). Prozor za otvaranje datoteke služi za izbor postojeće datoteke radi učitavanja podataka iz te datoteke u program. Prozor za čuvanje datoteke služi za izbor postojeće ili nepostojeće datoteke radi upisivanja podataka iz programa u tu datoteku. Metodi za prikazivanje ovih prozora su `showOpenDialog()` i `showSaveDialog()`. Obratite pažnju na to da se ovi metodi ne završavaju dok korisnik

ne izabere datoteku ili ne odustane od izbora.

Prozor za izbor datoteke uvek ima drugi prozor za koji je vezan. Taj „roditeljski" prozor je obično složenija grafička komponenta čiji jedan deo čini prozor za izbor datoteke. Roditeljski prozor se navodi kao argument metoda `showOpenDialog()` i `showSaveDialog()`. U prostim slučajevima kada prozor za izbor datoteke logički nema roditeljski prozor navodi se argument `null`, ali se i onda zapravo formira nevidljiva roditeljska komponenta.

Metodi `showOpenDialog()` i `showSaveDialog()` daju celobrojnu vrednost koja ukazuje na to šta je korisnik uradio sa prikazanim prozorom za izbor datoteke. Ta vrednost je jedna od statičkih konstanti koje su definisane u klasi `JFileChooser`:

- `CANCEL_OPTION`
- `ERROR_OPTION`
- `APPROVE_OPTION`

Ako je vraćena konstanta `JFileChooser.APPROVE_OPTION`, onda je korisnik izabrao datoteku u prozoru i njena reprezentacija tipa `File` može se dobiti pozivom metoda `getSelectedFile()` iz klase `JFileChooser`. U ostalim slučajevima datoteka nije izabrana u prozoru iz nekog razloga (na primer, korisnik je možda odustao od izbora klikom na dugme *Cancel*). Zbog toga u programu uvek treba proveriti vraćenu vrednost metoda za prikazivanje prozora za izbor datoteke i prema rezultatu te provere treba granati dalju logiku programa.

Klasa `JFileChooser` sadrži i metode kojima se prozor za izbor datoteke može dodatno konfigurisati pre nego što se prikaže na ekranu. Tako, metodom `setDialogTitle(String naslov)` određuje se tekst koji se pojavljuje u naslovu prozora za izbor datoteke. Metod `setSelectedFile(File ime)` služi da se u polju za ime datoteke prikaže datoteka koja se unapred bira ukoliko se drugačije ne izabere u prozoru za izbor datoteke. (Argument `null` ovog metoda označava da se nijedna datoteka ne podrazumeva da je prethodno izabrana i zato se prikazuje prazno polje za ime datoteke.)

U nastavku je prikazan tipičan postupak za čitanje neke tekstualne datoteke izabrane od strane korisnika:

```
// Biranje datoteke u prozoru za izbor
JFileChooser prozorIzbora = new JFileChooser();
prozorIzbora.setDialogTitle("Izaberite datoteku za čitanje");
prozorIzbora.setSelectedFile(null);  // ništa se unapred ne bira
```

```
int izbor = prozorIzbora.showOpenDialog(null);
if (izbor != JFileChooser.APPROVE_OPTION)
   return;  // korisnik je odustao od izbora
File datoteka = prozorIzbora.getSelectedFile();

// Otvaranje izabrane datoteke
Scanner ulaz;
try {
   ulaz = new Scanner(new FileReader(datoteka));
}
catch (Exception e) {
   JOptionPane.showMessageDialog(null,
      "Greška prilikom otvaranja datoteke:\n" + e);
   return;
}

// Čitanje podataka iz ulaznog toka i njihova obrada
try {
   .
   . // Naredbe za čitanje i obradu podataka
   .
}
catch (Exception e) {
   JOptionPane.showMessageDialog(null,
      "Greška prilikom čitanja datoteke:\n" + e);
}
finally {
   ulaz.close();
}
```

Postupak za izbor datoteke u koju treba upisivati podatke vrlo je sličan prethodnom postupku za čitanje podataka iz datoteke. Jedina razlika je u tome što se obično vrši dodatna provera da li izabrana datoteka već postoji. Naime, ukoliko izabrana datoteka postoji, njen sadržaj se zamenjuje upisanim podacima. Zbog toga korisnuku treba omogućiti da se predomisli u slučaju pogrešno izabrane datoteke, jer bi u suprotnom slučaju stari sadržaj postojeće datoteke bio bespovratno izgubljen. U nastavku je prikazan

tipičan postupak za pisanje neke tekstualne datoteke izabrane od strane korisnika:

```
// Biranje datoteke u prozoru za izbor
JFileChooser prozorIzbora = new JFileChooser();
prozorIzbora.setDialogTitle("Izaberite datoteku za pisanje");
prozorIzbora.setSelectedFile(null);  // ništa se unapred ne bira
int izbor = prozorIzbora.showOpenDialog(null);
if (izbor != JFileChooser.APPROVE_OPTION)
   return;  // korisnik je odustao od izbora
File datoteka = prozorIzbora.getSelectedFile();

// Proveravanje da li izabrana datoteka postoji
if (datoteka.exists()) { // zameniti postojeću datoteku?
   int odgovor = JOptionPane.showConfirmDialog(null,
                 "Datoteka \"" + datoteka.getName()
                 + "\" postoji.\nZameniti njen sadržaj?",
                 "Potvrdite zamenu datoteke",
                 JOptionPane.YES_NO_OPTION,
                 JOptionPane.WARNING_MESSAGE);
   if (odgovor != JOptionPane.YES_OPTION)
      return; // korisnik ne želi zamenu postojeće datoteke
}

// Otvaranje izabrane datoteke
PrintWriter izlaz;
try {
   izlaz = new PrintWriter(new FileWriter(datoteka));
}
catch (Exception e) {
   JOptionPane.showMessageDialog(null,
      "Greška prilikom otvaranja datoteke:\n" + e);
   return;
}

// Pisanje podataka u izlazni tok
try {
   .
```

```
    . // Naredbe za pisanje podataka
    .
}
catch (Exception e) {
    JOptionPane.showMessageDialog(null,
        "Greška prilikom pisanja u datoteku:\n" + e);
}
finally {
    izlaz.close();
}
```

Primer: kopiranje datoteke

Pravljenje identične kopije neke datoteke je česta aktivnost na računaru. To je razlog zašto svaki operativni sistem računara omogućava da se to uradi na lak način, bilo odgovarajućom komandom preko tastature ili postupkom „prevlačenja" miša koristeći grafički interfejs. U listingu 8.3 u nastavku prikazan je sličan Java program čiji je zadatak prepisivanje sadržaja jedne datoteke u drugu. Taj program je ilustrativan, jer mnoge operacije sa datotekama slede istu opštu logiku kopiranja jedne datoteke, osim što se podaci iz ulazne datoteke prethodno obrađuju na neki poseban način pre nego što se upišu u izlaznu datoteku.

Pošto program treba da kopira bilo koju datoteku, ne može se pretpostaviti da je originalna datoteka u tekstualnom obliku, nego se kopiranje mora izvršiti bajt po bajt. Zbog toga se ne mogu koristiti tekstualni tokovi tipa `Reader` i `Writer`, nego osnovni binarni tokovi `InputStream` i `OutputStream`. Ako je `original` promenljiva koja ukazuje na ulazni tok tipa `InputStream`, onda se naredbom `original.read()` učitava jedan bajt iz tog ulaznog toka. Metod `read()` vraća rezultat –1 kada su pročitani svi bajtovi iz ulaznog toga.

S druge strane, ako je `kopija` promenljiva koja ukazuje na izlazni tok tipa `OutputStream`, onda se naredbom `kopija.write(b)` upisuje jedan bajt u taj izlazni tok. Prema tome, osnovni postupak za kopiranje datoteke sastoji se od jedne obične petlje (koja se mora nalaziti unutar naredbe `try`, jer U/I operacije izbacuju proveravane izuzetke) u ovom obliku:

```
int b = original.read();
while(b >= 0) {
   kopija.write(b);
   b = original.read();
}
```

Da bi program bio što sličniji pravim komandama operativnih sistema za kopiranje datoteke, ime originalne datoteke i ime njene kopije navode se kao argumenti programa. Ovi argumenti su zapravo niz stringova koji se programu prenose preko parametra glavnog metoda `main()`. Tako, ako taj parametar ima uobičajeno ime `args` i ako je ime programa za kopiranje datoteke `KopiDat`, onda se u tom programu naredbom

```
java KopiDat orig.dat nova.dat
```

dobija vrednost elementa `args[0]` jednaka stringu `"orig.dat"` i vrednost elementa `args[1]` jednaka stringu `"nova.dat"`. Broj elemenata niza `args` određen je vrednošću polja `args.length`, kao što je to uobičajeno za svaki niz.

Da se nepažnjom ne bi izgubio sadržaj neke važne datoteke, u programu se kopiranje neće izvršiti ukoliko je za ime izlazne datoteke navedena neka postojeća datoteka. Međutim, da bi se kopiranje omogućilo i u tom slučaju, mora se navesti dodatna opcija `-u` (što podseća na „uvek") kao prvi argument programa. Prema tome, naredbom

```
java KopiDat -u orig.dat nova.dat
```

dobila bi se kopija `nova.dat` od originalne datoteke `orig.dat` bez obzira na to da li već postoji neka stara datoteka `nova.dat`.

Listing 8.3: Kopiranje datoteke

```
import java.io.*;
public class KopiDat {

   public static void main(String[] args) {
      String imeOriginala;
      String imeKopije;
      InputStream original;
      OutputStream kopija;
```

```
boolean uvek = false;
int brojBajtova;  // ukupan broj kopiranih bajtova

// Određivanje imena datoteka iz komandnog reda
if (args.length == 3 && args[0].equalsIgnoreCase("-u")) {
   imeOriginala = args[1];
   imeKopije = args[2];
   uvek = true;
}
else if (args.length == 2) {
   imeOriginala = args[0];
   imeKopije = args[1];
}
else {
   System.out.println(
      "Upotreba: java KopiDat <original> <kopija>");
   System.out.println(
      "   ili    java KopiDat -u <original> <kopija>");
   return;
}

// Konstruisanje ulaznog toka
try {
   original = new FileInputStream(imeOriginala);
}
catch (FileNotFoundException e) {
   System.out.print("Ulazna datoteka \"" + imeOriginala);
   System.out.println("\" ne postoji.");
   return;
}

// Proveravanje da li izlazna datoteka već postoji
File datoteka = new File(imeKopije);
if (datoteka.exists() && uvek == false) {
   System.out.print("Izlazna datoteka već postoji. ");
   System.out.println("Koristite opciju -u za njenu zamenu.");
   return;
```

```
}

// Konstruisanje izlaznog toka
try {
   kopija = new FileOutputStream(imeKopije);
}
catch (IOException e) {
   System.out.print("Izlazna datoteka \"" + imeKopije);
   System.out.println("\" ne može se otvoriti.");
   return;
}

// Bajt po bajt prepisivanje iz ulaznog toka u izlazni tok
brojBajtova = 0;
try {
   int b = original.read();
   while(b >= 0) {
      kopija.write(b);
      brojBajtova++;
      b = original.read();
   }
   System.out.print("Kopiranje završeno: kopirano ");
   System.out.println(brojBajtova + " bajtova.");
}
catch (Exception e) {
   System.out.print("Neuspešno kopiranje (kopirano ");
   System.out.println(brojBajtova + " bajtova).");
   System.out.println("Greška: " + e);
}
finally {
   try {
      original.close();
      kopija.close();
   }
   catch (IOException e) {
   }
}
```

```
    }
}
```

GLAVA 9

Programske niti

Računari mogu obavljati više različitih zadataka u isto vreme i ta njihova mogućnost se naziva *multitasking*. Ovaj način rada nije teško zamisliti kod računara sa više procesora, jer se različiti zadaci mogu paralelno izvršavati na odvojenim procesorima. Ali i računari sa jednim procesorom mogu raditi multitasking, doduše ne bukvalno nego simulirajući paralelnost naizmeničnim izvršavanjem više zadataka po malo.

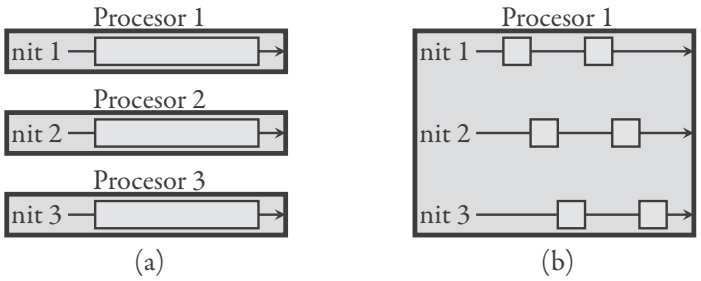
Slično računarima, posao jednog programa u Javi se može podeliti u više zadataka koji se paralelno izvršavaju. Jedan zadatak koji se paraleno izvršava sa drugim zadacima u okviru programa naziva se *nit izvršavanja* ili kraće samo *nit* (engl. *thread*). Analogno, ova mogućnost u Javi se naziva *multithreading*.

U ovom poglavlju se govori o osnovnim konceptima u vezi sa nitima izvršavanja, kao i o tome kako se pišu programi sa više niti u Javi. Nažalost, paralelno programiranje je još teže od običnog (jednonitnog) programiranja, jer kada se više niti izvršava istovremeno, njihov efekat na stanje programa je mnogo teže pratiti i moguće su potpuno nove vrste grešaka.

9.1 Osnovni pojmovi

Zadatak je programska celina koja se izvršava nezavisno od drugih delova programa. *Nit* je sled izvšavanja nekog zadatka, od početka do kraja, unutar programa. Nit je dakle programski mehanizam kojim se obezbeđuje nezavisno izvršavanje jednog zadatka u okviru programa. U jednom Java

programu se može pokrenuti više niti radi paralelnog izvršavanja. Više niti se u multiprocesorskom računaru mogu izvršavati istovremeno, dok se u jednoprocesorskom računaru može dobiti privid njihovog istovremenog izvršavanja. Način na koji se više niti mogu izvršavati u računaru ilustrovan je na slici 9.1.



Slika 9.1: Izvršavanje tri niti na (a) multiprocesorskom i (b) jednoprocesorskom računaru.

Kao što pokazuje slika 9.1 pod (b), više niti u jednoprocesorskom sistemu dele vreme jedinog procesora tako što ih procesor izvršava malo po malo. Na taj način se praktično dobija privid paralelnog izvršavanja zadataka, naročito ako su oni po prirodi raznovrsni. Na primer, procesor ne radi ništa dok se u jednom programu čeka da korisnik unese neke podatke preko tastature, pa za to vreme procesor može izvršavati neki drugi zadatak kao što je, recimo, crtanje nekog prozora na ekranu.

Više niti u programu doprinose većoj „živosti" i interaktivnosti programa. Tako, recimo, svaki dobar program za unos dokumenata omogućava da se dokument štampa na štampaču ili da se dokument sačuva na disku dok se istovremeno kuca na tastaturi. To je obezbeđeno time što se takav program sastoji od dve-tri niti u kojima se paralelno može štampati dokument, upisivati datoteka na disku i učitavati znakovi uneseni preko tastature ili izvršavati opcije izabrane mišem.

Svaki Java program poseduje bar jednu nit izvršavanja — kada se pokrene program, JVM automatski formira nit u kojoj se izvršava glavni metod `main()` navedene klase. U ovoj početnoj niti se zatim mogu pokrenuti druge niti radi paralelnog izvršavanja. Primetimo da se izvršavanje tih ni-

ti može nastaviti čak i posle završetka početne niti. Grafički Java program poseduje bar još jednu dodatnu nit izvršavanja u kojoj se odvija rukovanje događajima i crtanje komponenti na ekranu. Ova „grafička" nit se pokreće čim se konstruiše prvi grafički okvir u programu i ona se izvršava paralelno sa ostalim nitima. O ovoj niti se više govori na strani 335.

Jedna nit izvršavanja je u Javi predstavljena objektom koji pripada klasi `Thread` (ili klasi koja nasleđuje ovu klasu) iz paketa `java.lang`. Svrha ovog objekta je da se izvrši određen zadatak koji je obuhvaćen jednim metodom. Tačnije, jedan zadatak je instanca interfejsa `Runnable` u kojem je definisan samo jedan metod koji se izvršava u posebnoj niti. Kada se izvršavanje tog metoda završi, bilo regularno ili usled nekog neuhvaćenog izuzetka, završava se i izvršavanje njegove niti. Završena nit se više ne može aktivirati, a ni njen objekat se više ne može koristiti za pokretanje nove niti.

9.2 Zadaci i niti za paralelno izvršavanje

Da bi se definisao zadatak koji se može paralelno izvršavati, mora se najpre definisati klasa takvog zadatka. Zadaci za paralelno izvršavanje u Javi su objekti, kao i sve ostalo, ali njihova klasa mora implementirati poseban interfejs `Runnable` da bi se ti zadaci mogli paralelno izvršavati u nitima. Interfejs `Runnable` je prilično jednostavan i sadrži samo jedan metod: `public void run()`. Ovaj metod se mora implementirati u klasi zadatka kako bi se ukazalo da se zadatak izvršava u posebnoj niti i da bi se definisala funkcionalnost zadatka. Osnovni šablon za definisanje klase zadataka ima dakle sledeći oblik:

```
// Šablon za klasu zadatka koji se paralelno izvršava
public class KlasaZadatka implements Runnable {
   .
   . // Ostala polja i metodi
   .
   // Konstruktor za inicijalizaciju instance zadatka
   public KlasaZadatka(...) {
      . . .
   }
```

```
    // Implementacija metoda run() iz interfejsa Runnable
    public void run() {
        .
        . // Naredbe za izvršavanje zadatka
        .
    }
}
```

Konkretan zadatak čija je klasa definisana prema ovom šablonu može se zatim konstruisati na uobičajen način:

```
KlasaZadatka zadatak = new KlasaZadatka(...);
```

Zadatak se mora izvršiti u niti. Klasa `Thread` sadrži konstruktore za konstruisanje niti i mnoge korisne metode za kontrolisanje niti. Da bi se konstruisala nit za izvršavanje prethodno konstruisanog zadatka, može se pisati na primer:

```
Thread nit = new Thread(zadatak);
```

Međutim, konstruisanjem niti se automatski ne započinje izvršavanje odgovarajućeg zadatka. To se mora eksplicitno uraditi u odgovarajućoj tački programa, odnosno nakon konstruisanja niti mora se koristiti metod `start()` iz klase `Thread` radi pokretanja izvršavanja zadatka u niti:

```
nit.start();
```

Primetimo da se metodom `start()` zapravo počinje izvršavanje metoda `run()` koji je implementiran u klasi zadatka. Obratite pažnju i na to da se metod `run()` implicitno poziva od strane JVM — ako se metod `run()` pozove direktno u programu, on će se izvršiti u istoj niti u kojoj se izvršava njegov poziv, bez formiranja nove niti! Kompletan šablon za ceo postupak može se prikazati na sledeći način:

```
// Šablon za izvršavanje zadatka u niti
public class NekaKlasa {
    .
    . // Ostala polja i metodi
    .
    public void nekiMetod(...) {
        . . .
```

```
        // Konstruisanje instance klase KlasaZadatka
        KlasaZadatka zadatak = new KlasaZadatka(...);

        // Konstruisanje niti za izvršavanje te instance
        Thread nit = new Thread(zadatak);

        // Pokretanje niti za izvršavanje zadatka
        nit.start();
        . . .
    }
}
```

Primer: tri paralelne niti

U ovom primeru je u listingu 9.1 prikazan program u kojem se tri zadatka paralelno izvršavaju u svojim nitima:

- prvi zadatak prikazuje tekst „Java" 20 puta;
- drugi zadatak prikazuje tekst „C++" 20 puta;
- treći zadatak prikazuje brojeve od 1 do 20.

Pošto prvi i drugi zadatak imaju sličnu funkcionalnost, predstavljeni su jednom klasom `PrikazTeksta`. Ova klasa implementira interfejs `Runnable` definisanjem metoda `run()` za prikaz teksta određen broj puta. Treći zadatak je predstavljen klasom `PrikazBrojeva` koja takođe implementira interfejs `Runnable` definisanjem metoda `run()`, ali za prikaz niza brojeva od 1 do neke granice.

Listing 9.1: Tri paralelne niti

```
public class TriNiti {

   public static void main(String[] args) {

      // Konstruisanje tri zadatka
      Runnable java = new PrikazTeksta("Java",20);
      Runnable cpp = new PrikazTeksta("C++",20);
```

```
      Runnable niz = new PrikazBrojeva(20);

      // Konstruisanje tri niti izvršavanja
      Thread nitJava = new Thread(java);
      Thread nitCpp = new Thread(cpp);
      Thread nitNiz = new Thread(niz);

      // Pokretanje niti
      nitJava.start();
      nitCpp.start();
      nitNiz.start();
   }
}

// Zadatak za prikaz teksta određen broj puta
class PrikazTeksta implements Runnable {

   private String tekst; // tekst koji se prikazuje
   private int n;        // broj puta koliko se prikazuje

   // Konstruktor zadatka
   public PrikazTeksta(String tekst, int n) {
      this.tekst = tekst;
      this.n = n;
   }

   // Implementacija metoda run()
   public void run() {
     for (int i = 0; i < n; i++) {
        System.out.print(tekst + " ");
     }
   }
}

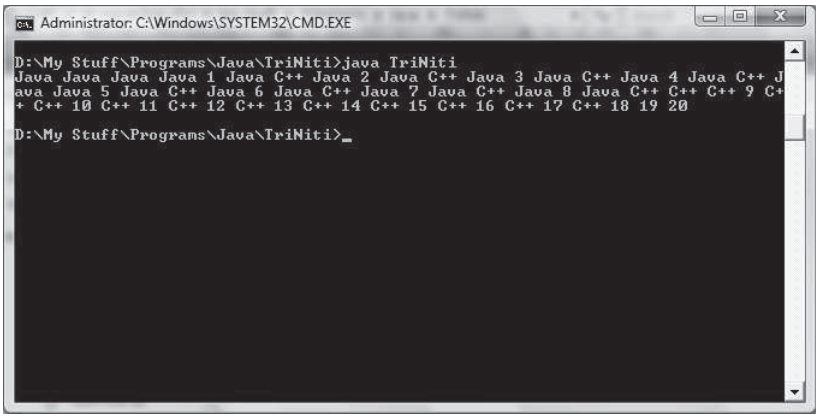
// Zadatak za prikaz niza brojeva od 1 do neke granice
class PrikazBrojeva implements Runnable {
```

```
    private int n;  // granica niza brojeva koji se prikazuju

    // Konstruktor zadatka
    public PrikazBrojeva(int n) {
        this.n = n;
    }

    // Implementacija metoda run()
    public void run() {
        for (int i = 1; i <= n; i++) {
            System.out.print(i + " ");
        }
    }
}
```

Ako bi se ovaj program u listingu 9.1 izvršio na troprocesorskom računaru, sve tri niti bi se (verovatno) izvršavale istovremeno. Ali ukoliko bi se ovaj program izvršio na jednoprocesorskom računaru, tri niti bi delile vreme jedinog procesora i naizmenično bi se prikazivao tekst i brojevi na ekranu. Na slici 9.2 je prikazan jedan mogući rezultat izvršavanja programa na jednoprocesorskom računaru.



Slika 9.2: Rezultat izvršavanja tri niti na jednoprocesorskom računaru.

Klasa `Thread`

Pored metoda `start()` kojim se započinje izvršavanje niti, odnosno kojim se zapravo pokreće metod `run()` odgovarajućeg zadatka radi izvršavanja u niti, klasa `Thread` sadrži i metode kojima se može dodatno kontrolisati izvršavanje neke niti. U stvari, klasa `Thread` implementira interfejs `Runnable` tako da se zajedno mogu definisati neka nit i zadatak koji se u njoj izvršava. Naime, ovaj drugi način za programiranje niti sastoji se od definisanja jedne klase koja nasleđuje klasu `Thread` i ujedno implementiranja metoda `run()` u toj novoj klasi kojim se određuje zadatak koji će se izvršiti u niti. Drugim rečima, kada se objekat niti nove klase pokrene, taj metod `run()` će se izvršiti u posebnoj niti. Šablon za drugi način programiranja niti ima dakle sledeći oblik:

```
// Šablon za definisanje klase niti i zadatka zajedno
public class NekaNit extends Thread {
   .
   . // Ostala polja i metodi
   .
   // Konstruktor
   public NekaNit(...) {
      . . .
   }

   // Implementacija metoda run() iz interfejsa Runnable
   public void run() {
      .
      . // Naredbe za izvršavanje zadatka
      .
   }
}
```

Ovako definisana nit, kao i zadatak koji se u njoj izvršava, mogu se zatim koristiti otprilike na sledeći način:

```
// Šablon za izvršavanje niti
public class NekaKlasa {
   .
   . // Ostala polja i metodi
```

```
    .
    public void nekiMetod(...) {
        . . .
        // Konstruisanje jedne niti
        NekaNit nit1 = new NekaNit(...);
        // Startovanje te niti
        nit1.start();
        . . .
        // Konstruisanje druge niti
        NekaNit nit2 = new NekaNit(...);
        // Startovanje te niti
        nit2.start();
        . . .
    }
}
```

Iako ovaj drugi način za programiranje niti izgleda kompaktniji, takav pristup obično nije dobar jer se u njemu mešaju pojmovi zadatka i mehanizma za izvršavanje tog zadatka. Zbog toga je u programu obično bolje odvojiti definiciju zadatka od niti u kojoj se izvršava.

Nepotpun spisak metoda u klasi `Thread` kojima se može kontrolisati izvršavanje niti je:

- `boolean isAlive()`
- `static void sleep(long milisec)`
- `void interrupt()`
- `static void yield()`
- `void join()`
- `void setPriority(int p)`

Metod `isAlive()` se koristi za proveru statusa izvršavanja neke niti. Ako je `t` objekat tipa `Thread`, onda `t.isAlive()` kao rezultat daje tačno ili netačno prema tome da li je nit `t` „živa". Nit je „živa" između trenutka njenog pokretanja i trenutka njenog završetka. Nakon završetka niti se kolokvijalno kaže da je ona „mrtva".

Statički metod `sleep()` prevodi nit u kojoj se taj metod izvršava u stanje „spavanja" za navedeni broj milisekundi. Nit koja je u stanju spavanja

je i dalje živa, ali se ne izvršava nego je blokirana. Dok se neka nit nalazi u stanju spavanja, računar može da izvršava neku drugu nit istog programa ili potpuno drugi program. Metod `sleep()` koristi se radi privremenog prekida izvršavanja neke niti i nastavljanja njenog izvršavanja posle isteka navedene dužine vremena. Ovaj metod može izbaciti proveravani izuzetak tipa `InterruptedException` koji zahteva obavezno rukovanje. U praksi to znači da se metod `sleep()` obično piše unutar naredbe `try` u kojoj se hvata potencijalni izuzetak tipa `InterruptedException`:

```
try {
   Thread.sleep(dužinaPauze);
}
catch (InterruptedException e) {
}
```

Na primer, ako se u primeru tri niti u listingu 9.1 na strani 315 izmeni metod `run()` za zadatak `PrikazBrojeva` na sledeći način:

```
public void run() {
   try {
      for (int i = 1; i <= n; i++) {
         System.out.print(i + " ");
         if (i > 10)
            Thread.sleep(1);
      }
   }
   catch (InterruptedException e) {
   }
}
```

onda se, nakon prikazivanja svakog broja većeg od 10, izvršavanje niti `nitNiz` prekida za (najmanje) jednu milisekundu.

Jedna nit može drugoj niti poslati signal prekida da druga nit prekine ono što trenutno radi i da uradi nešto sasvim drugo. Na primer, ukoliko je nit u stanju spavanja ili je blokirana iz nekog drugog razloga, to može biti signal da se ta nit „probudi" i uradi nešto što zahteva hitni intervenciju. Jedna nit šalje signal prekida drugoj niti `t` pozivanjem njenog metoda `t.interrupt()`. Detaljan opis mehanizma prekidanja niti prevazilazi okvire ove knjige, ali ipak treba znati da nit koja se prekida od strane druge niti

mora biti pripremljena da reaguje na poslati signal prekida.

Statički metod `yield()` je indikacija za JVM da je nit u kojoj se taj metod izvršava spremna da prepusti procesor drugim nitima radi izvršavanja. Na primer, ako se u primeru tri niti u listingu 9.1 na strani 315 izmeni metod `run()` za zadatak `PrikazBrojeva` na sledeći način:

```
public void run() {
   try {
      for (int i = 1; i <= n; i++) {
         System.out.print(i + " ");
         Thread.yield();
      }
   }
   catch (InterruptedException e) {
   }
}
```

onda se, nakon prikazivanja svakog broja, izvršavanje niti `nitNiz` prekida tako da se iza svakog broja prikazuje neki tekst iz druge niti. Obratite ipak pažnju na to da JVM nije obavezna da poštuje ove „lepe manire" neke niti i da po svom nahođenju može ignorisati poziv metoda `yield()` u niti i nastaviti njeno izvršavanje bez prekida. Zbog toga se ovaj metod retko koristi u praksi, osim za testiranje i rešavanje problema neefikasnosti u programu radi otklanjanja „uskih grla".

U nekim slučajevima je potrebno da jedna nit sačeka da se druga nit završi i da tek onda prva nit nastavi izvršavanje. Ova funkcionalnost se može postići metodom `join()` iz klase `Thread`. Ako je `t` jedna nit tipa `Thread` i u drugoj niti se izvršava `t.join()`, onda ova druga nit prelazi u stanje spavanja dok se prva nit `t` ne završi. Ako se nit `t` već završila, onda poziv `t.join()` nema efekta, odnosno nit u kojoj se izvršava taj poziv nastavlja izvršavanje bez prekida. Metod `join()` može izbaciti izuzetak tipa `InterruptedException` čije je rukovanje obavezno u programu.

Radi ilustracije, primetimo da se u primeru tri niti u listingu 9.1 na strani 315 sve niti zapravo pokreću u četvrtoj, glavnoj niti u kojoj se izvršava metod `main()`. Nakon toga se glavna nit odmah završava, skoro sigurno pre ostalih niti. Zato ukoliko je potrebno da se, recimo, prikaže ukupno vreme izvršavanja sve tri niti, to se mora uraditi na sledeći način:

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

   // Konstruisanje tri zadatka
   Runnable java = new PrikazTeksta("Java",20);
   Runnable cpp = new PrikazTeksta("C++", 20);
   Runnable niz = new PrikazBrojeva(20);

   // Konstruisanje tri niti izvršavanja
   Thread nitJava = new Thread(java);
   Thread nitCpp = new Thread(cpp);
   Thread nitNiz = new Thread(niz);

   long početak = System.currentTimeMillis();

   // Startovanje niti
   nitJava.start();
   nitCpp.start();
   nitNiz.start();

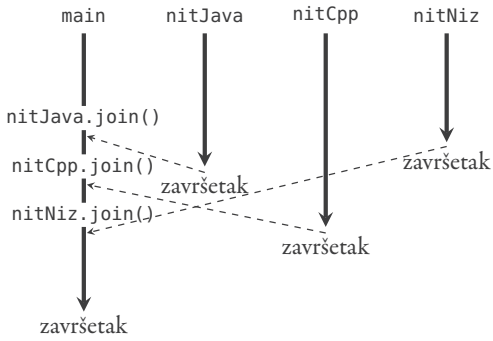
   try {
      nitJava.join(); // čekanje dok se nitJava ne završi
      nitCpp.join();  // čekanje dok se nitCpp ne završi
      nitNiz.join();  // čekanje dok se nitNiz ne završi
   }
   catch (InterruptedException e) {
   }

   // U ovoj tački su sve tri niti završene
   long ukupnoVreme = System.currentTimeMillis() - početak;
   System.out.println("\n");
   System.out.print("Ukupno vreme izvršavanja tri niti: ");
   System.out.println((ukupnoVreme/1000.0) + " sekundi.");
}
```

Izvršavanje ove verzije metoda `main()` ilustrovano je na slici 9.3. Primetimo da ovaj metod ne daje očekivani rezultat ukoliko u njemu neki od metoda `join()` izbaci izuzetak tipa `InterruptedException`. Takva moguć-

nost, ma kako ona mala bila, mora se predupredити i obezbediti sigurno registrovanje završetka neke niti otprilike na sledeći način:

```
while (nitJava.isAlive()) {
   try {
      nitJava.join();
   }
   catch (InterruptedException e) {
   }
}
```



Slika 9.3: Čekanje na završetak tri niti upotrebom metoda `join()`.

Svakoj niti JVM dodeljuje prioritet za izvršavanje. Jedna nit `t` na početku nasleđuje prioritet one niti u kojoj je nit `t` pokrenuta. Prioritet niti može se menjati metodom `setPriority()`, a aktuelna vrednost prioriteta niti može se dobiti metodom `getPriority()`. Prioriteti su celi brojevi od 1 do 10, a klasa `Thread` sadrži konstante `MIN_PRIORITY`, `NORM_PRIORITY` i `MAX_PRIORITY` koje odgovaraju prioritetima 1, 5 i 10. Glavna nit na početku dobija prioritet `Thread.NORM_PRIORITY`.

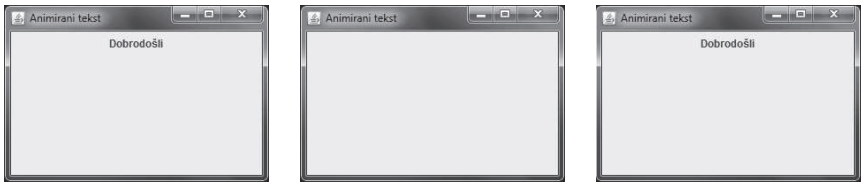
Od niti koje su spremne za izvršavanje, JVM radi izvršavanja uvek bira onu koja ima najviši prioritet. Ukoliko nekoliko spremnih niti imaju isti prioritet, one se na jednom procesoru izvršavaju na kružni način. Niti nižeg prioriteta mogu se izvršavati samo ukoliko nema spremnih niti višeg prioriteta. Na primer, ako se nakon startovanja tri niti u prethodnom primeru metoda `main()` doda naredba

```
nitJava.setPriority(Thread.MAX_PRIORITY);
```

onda se prva završava nit `nitJava`.

Primer: animirani tekst

Mehanizam paralelnog izvršavanja niti može se iskoristiti za realizovanje animacija na ekranu. Efekat animacije postiže se prikazivanjem više sličnih slika određenom brzinom što ljudsko oko, zbog svoje sporosti, registruje kao „pokretne slike". U grafičkom programu u listingu 9.2 proizvodi se efekat treptanja teksta tako što se tekst naizmenično prikazuje i briše. Usporen rad ovog programa, odnosno slike koje se prikazuju svake pola sekunde, ilustrovan je na slici 9.4.



Slika 9.4: Efekat treptanja teksta „Dobrodošli".

Klasa `AnimiraniTekst` u listingu 9.2 implementira interfejs `Runnable` i zato predstavlja klasu zadatka koji se može paralelno izvršavati u niti. U konstruktoru ove klase se konstruiše nit za ovaj zadatak i odmah se pokreće ta nit. Metod `run()` određuje kako se zadatak izvršava u niti: tekst u oznaci tipa `JLabel` se upisuje ukoliko ta oznaka ne sadrži tekst, a tekst u oznaci se briše ukoliko ta oznaka sadrži tekst. Na taj način se zapravo simulira treptanje teksta, ukoliko se upisivanje i brisanje teksta obavljaju dovoljno brzo (ali ne i prebrzo).

Listing 9.2: Animirani tekst

```
import javax.swing.*;

public class TestAnimiraniTekst {
```

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

      // Konstruisanje okvira
      JFrame okvir = new JFrame("Animirani tekst");
      okvir.setSize(300, 200);
      okvir.setLocation(100, 150);
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      okvir.add(new AnimiraniTekst());
      okvir.setVisible(true);
   }
}

class AnimiraniTekst extends JPanel implements Runnable {

   private JLabel oznaka = new JLabel("Dobrodošli", JLabel.CENTER);

   public AnimiraniTekst() {
      add(oznaka);
      new Thread(this).start();
   }

   // Prikazati ili obrisati tekst svakih 500 milisekundi
   // iz posebne niti, izvan grafičke niti za crtanje
   public void run() {
      try {
         while (true) {
            if (oznaka.getText() == null)
               oznaka.setText("Dobrodošli");
            else
               oznaka.setText(null);

            Thread.sleep(500);
         }
      }
      catch (InterruptedException ex) {
      }
```

```
    }
}
```

U ovom programu je važno primetiti da se beskonačna petlja za prikazivanje i brisanje trepćućeg teksta mora izvršavati u posebnoj niti, izvan grafičke niti u kojoj se izvodi crtanje na ekranu. U suprotnom slučaju, ukoliko bi se ta petlja izvršavala u konstruktoru klase `AnimiraniTekst` koji se izvršava u grafičkoj niti, izvršavanje tog konstruktora se ne bi nikad završilo. A to bi sa svoje strane u metodu `main()` glavne klase `TestAnimiraniTekst` sprečilo da se glavni okvir programa učini vidljivim, odnosno ništa se ne bi prikazalo na ekranu.

9.3 Sinhronizacija niti

Programiranje više niti u kojima se paralelno izvršavaju nezavisni zadaci nije toliko komplikovano. Prava teškoća nastaje kada niti moraju da uzajamno komuniciraju radi dobijanja konačnog rezultata programa. Jedan način na koji niti mogu da međusobno sarađuju je deljenjem računarskih resursa kao što su promenljive ili prozori na ekranu. Ali ako dve ili više niti moraju da koriste isti resurs, onda se mora obratiti pažnja na njihovo programiranje kako ne bi koristile taj resurs u isto vreme.

Da bismo ilustrovali ovaj problem, razmotrimo najobičniju naredbu dodele kojom se uvećeva neki brojač za jedan:

```
brojač = brojač + 1;
```

Izvršavanje ove naredbe izvodi se, u stvari, u tri koraka:

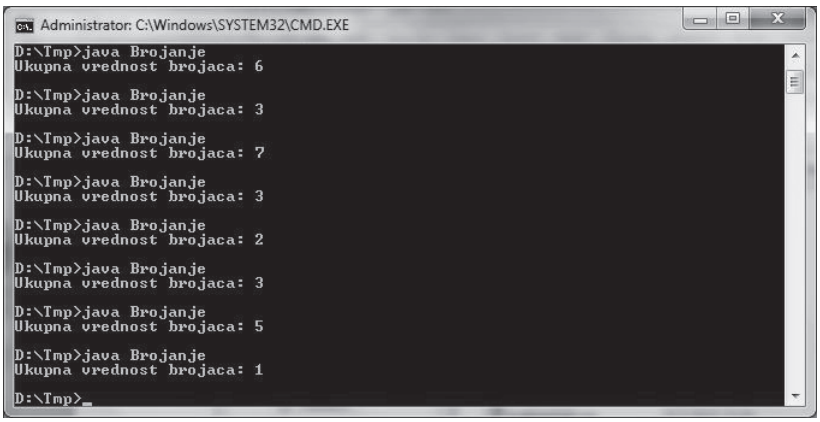
1. čitanje vrednosti promenljive `brojač`;
2. izračunavanje zbira te vrednosti i broja 1;
3. upisivanje tog zbira u istu promenljivu `brojač`.

Ako dve niti izvršavaju ovu naredbu redom jedna iza druge, onda će efekat toga biti uvećana vrednost brojača za 2. Međutim, ako dve niti izvršavaju ovu istu naredbu u isto vreme, onda efekat toga može biti neočekivan. Naime, pretpostavimo da je jedna nit završila samo prva dva od prethodna

tri koraka kada je bila prekinuta i da je druga nit počela da izvršava sva tri koraka. Pošto u prvoj niti nova vrednost brojača nije bila upisana, u drugoj niti će biti pročitana stara vrednost brojača i ona će biti uvećana i upisana kao nova vrednost brojača. Kada prva nit nastavi sa izvršavanjem, opet će stara vrednost brojača uvećana za jedan biti upisana kao nova vrednost brojača. Prema tome, efekat ovog mogućeg načina paralelnog izvršavanja biće vrednost brojača uvećana samo za 1, a ne za 2!

Ova vrsta greške kod paralelnog izvršavanja naziva se *stanje trke* (engl. *race condition*). Ona nastaje kada je jedna nit prekinuta u sredini izvršavanja neke složene operacije od više koraka, a druga nit može promeniti neku vrednost ili uslov od kojih zavisi izvršavanje prve niti. Termin za ovu vrstu greške potiče od toga što je prva nit „u trci" da završi sve korake pre nego što bude prekinuta od strane druge niti.

Da bismo ilustrovali problem trke niti, u programu u listingu 9.3 se konstruiše brojač `b` sa početnom vrednošću `0`, kao i `10` niti (zadataka) u kojima se tom brojaču dodaje jedinica. Nakon završetka paralelnog izvršavanja svih niti prikazuje se vrednost brojača, koja bi naravno trebalo da bude `10` da nema problema trke niti. Međutim, kako sve niti istovremeno menjaju isti brojač, rezultat je nepredvidljiv. Tako, jedan pogrešan rezultat izvršavanja ovog programa je prikazan na slici 9.5.



Slika 9.5: Problem trke niti je uzrok pogrešnog brojanja.

Listing 9.3: Trka niti za brojanje

```
public class Brojanje {

   public static void main(String[] args) {

      Brojač b = new Brojač();
      Thread[] niti = new Thread[10];

      for (int i = 0; i < niti.length; i++) {
         niti[i] = new Dodavanje1(b);
         niti[i].start();
      }
      for (int i = 0; i < niti.length; i++) {
         try {
            niti[i].join();
         }
         catch (InterruptedException e) {
         }
      }
      // U ovoj tački su sve niti sigurno završene
      System.out.print("Ukupna vrednost brojača: ");
      System.out.println(b.vrednost());
   }
}

class Dodavanje1 extends Thread implements Runnable {

   private Brojač b;
   public Dodavanje1(Brojač b) {
      this.b = b;
   }
   public void run() {
      b.dodaj1();
   }
}
```

```
class Brojač {

   private int brojač = 0;

   public void dodaj1() {

      // Naredba brojač = brojač + 1 je namerno razbijena na svoje
      // sastavne delove kako bi se istakao problem trke. Taj problem
      // je dodatno pojačan slučajnom zadrškom niti od 1 ms.
      int n = brojač;
      n = n + 1;
      if (Math.random() < 0.5) {
         try {
            Thread.sleep(1);
         }
         catch (InterruptedException ex) {
         }
      }
      brojač = n;
   }
   public int vrednost() {
      return brojač;
   }
}
```

Drugi slučaj greške trke niti može nastati u naredbi `if`, recimo u sledećem primeru u kome se izbegava deljenje sa nulom:

```
if (a != 0)
   c = b / a;
```

Ako se ova naredba izvršava u jednoj niti i vrednost promenljive `a` mogu promeniti više niti, onda može doći do stanja trke ukoliko se ne preduzmu određene mere za sprečavanje tog problema. Naime, može se desiti da neka druga nit dodeli vrednost nula promenljivoj `a` u momentu između trenutka kada se u prvoj niti proverava uslov `a != 0` i trenutka kada se stvarno obavlja deljenje `c = b / a`. To znači da u prvoj niti može doći do greške deljenja sa nulom, iako se u toj niti neposredno pre deljenja provera da promenljiva `a`

nije jednaka nuli!

Da bi se rešio problem trke niti koje se paralelno izvršavaju, mora se u višenitnom programu obezbediti da jedna nit dobije ekskluzivno pravo da koristi deljeni resurs u određenom trenutku. Taj mehanizam *uzajamne isključivosti* u Javi realizovan je preko *sinhronizovanih metoda* i *sinhronizovanih naredbi*. Deljeni resurs u metodu ili blok naredbi štiti se tako što se obezbeđuje da se izvršavanje sinhronizovanog metoda ili naredbe u jednoj niti ne može prekinuti od strane neke druge niti.

Problem trke niti u primeru programa u listingu 9.3 može se rešiti ukoliko se metodi za uvećavanje brojača i dobijanje njegove vrednosti u klasi `Brojač` definišu da budu sinhronizovani. U tu svrhu je dovoljno u zaglavlju ovih metoda na početku dodati samo modifikator `synchronized`:

```
class SinhroniBrojač {

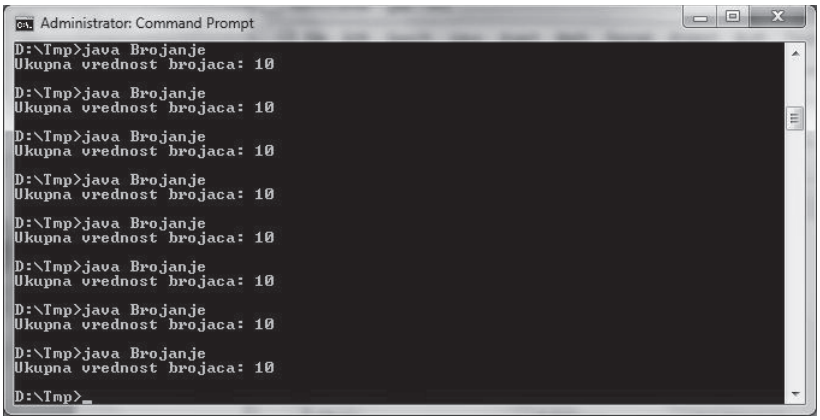
   private int brojač = 0;

   public synchronized void dodaj1() {
      brojač = brojač + 1;
   }

   public synchronized int vrednost() {
      return brojač;
   }
}
```

Ako je brojač `b` tipa `SinhroniBrojač`, onda se u svakoj niti za dodavanje 1 tom brojaču može izvršiti `b.dodaj1()` na potpuno bezbedan način. Činjenica da je metod `dodaj1()` sinhronizovan znači to da se samo u jednoj niti može izvršavati ovaj metod od početka do kraja. Drugim rečima, kada se u jednoj niti počne izvršavanje ovog metoda, nijedna druga nit ne može početi da izvršava isti metod dok se on ne završi u niti u kojoj je prvo počelo njegovo izvršavanje. Ovim se sprečava mogućnost istovremenog menjanja zajedničkog brojača u programu za brojanje i sada je njegov rezultat uvek tačan, kao što to pokazuje slika 9.6.

Primetimo da rešenje problema trke niti u klasi `SinhroniBrojač` suštinski zavisi od toga što je `brojač` privatna promenljiva. Na taj način se obezbe-



Slika 9.6: Problem trke niti je rešen sinhronizovanim metodima.

đuje da svaki pristup toj promenljivoj mora ići preko sinhronizovanih metoda u klasi `SinhroniBrojač`. Da je `brojač` bila javna promenljiva, onda bi neka nit mogla da zaobiđe sinhronizaciju, recimo, naredbom `b.brojač++`. Time bi se opet otvorila mogućnost stanja trke u programu, jer bi jedna nit mogla promeniti vrednost brojača istovremeno dok druga nit izvršava `b.dodaj1()`. Ovo pokazuje da sinhronizacija ne garantuje ekskluzivan pristup deljenom resursu, nego samo obezbeđuje uzajamnu isključivost između onih niti u kojima se poštuje princip sinhronizacije radi pristupa deljenom resursu.

Klasa `SinhroniBrojač` ne sprečava sve moguće probleme trke niti koji mogu nastati kada se koristi neki brojač `b` te klase. Pretpostavimo, na primer, da je za ispravan rad nekog metoda potrebno da vrednost brojača `b` bude jednaka nuli. To možemo pokušati da obezbedimo sledećom naredbom `if`:

```
if (b.vrednost() == 0)
   nekiMetod();
```

U ovom slučaju ipak može doći do stanja trke u programu ukoliko se u drugoj niti uveća vrednost brojača između trenutka kada se u prvoj niti proverava uslov `b.vrednost() == 0` i trenutka kada se izvršava `nekiMetod()`. Drugim rečima, za pravilan rad programa, prvoj niti je potreban ekskluziv-

ni pristup brojaču `b` za *sve* vreme izvršavanja naredbe `if`. Problem u ovom slučaju je to što se sinhronizacijom metoda u klasi `SinhroniBrojač` dobija ekskluzivni pristup brojaču `b` samo za vreme izvršavanja poziva metoda `b.vrednost()` unutar naredbe `if`. Ovaj problem se može rešiti tako što se naredba `if` napiše unutar sinhronizovane naredbe koja počinje službenom rečju `synchronized`:

```
synchronized (b) {
   if (b.vrednost() == 0)
      nekiMetod();
}
```

U ovom slučaju izvršavanje cele naredbe `if`, a u opštem slučaju čitavog bloka naredbi,[1] neće biti prekinuto u jednoj niti dok se potpuno ne završi. Obratite pažnju na to da je u ovom slučaju objekat `b` neka vrsta argumenta sinhronizovane naredbe. To nije slučajno, jer u opštem obliku sintaksa sinhronizovane naredbe je:

```
synchronized (objekat) {
   .
   . // Naredbe
   .
}
```

Sinhronizacija niti putem uzajamne isključivosti u Javi uvek se obavlja na osnovu nekog objekta. Ova činjenice se kolokvijalno izražava terminom da se vrši sinhronizacija „po" nekom objektu: na primer, prethodna naredba `if` je sinhronizovana po objektu `b`.

Kao što se kod sinhronizovane naredbe neki blok naredbi sinhronizuje po objektu, tako se i sinhronizovani metodi implicitno sinhronizuju po objektu. Naime, objektni (nestatički) metodi, kao što su oni u klasi `SinhroniBrojač`, sinhronizuju se po objektu za koji se pozivaju. U stvari, dodavanje modifikatora `synchronized` nekom objektnom metodu ekvivalentno je pisanju tela tog metoda unutar sinhronizovane naredbe sa argumentom `this`. Na primer, na slici 9.7 je sinhronizovani metod pod (a) ekvivalentan metodu pod (b). Statički metodi takođe mogu biti sinhronizovani. Oni

[1] Blok naredbi unutar sinhronizovane naredbe se naziva i *kritična oblast*.

se sinhronizuju po specijalnom objektu koji predstavlja klasu koja sadrži statički metod.

```
public synchronized void sMetod() {

    .
    . // Telo metoda
    .

}
```

(a)

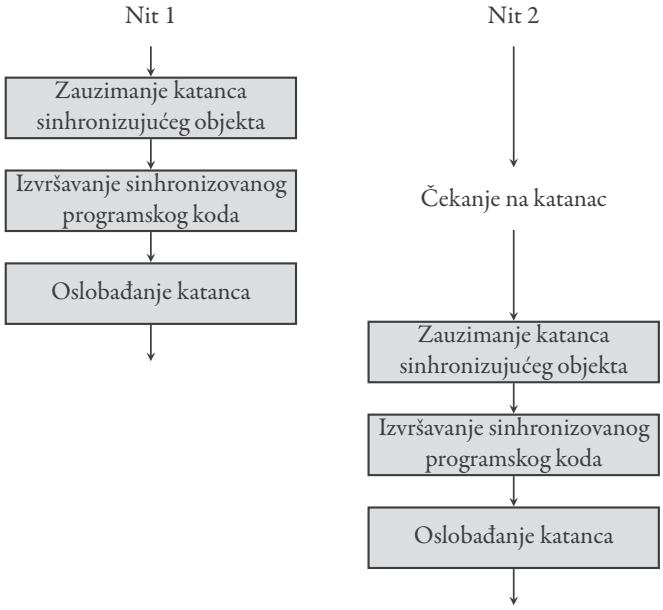
```
public void sMetod() {
   synchronized (this) {
      .
      . // Telo metoda
      .
  }
}
```

(b)

Slika 9.7: Sinhronizacija metoda po objektu za koji se poziva.

Opšte pravilo u Javi za sinhronizaciju niti je dakle da se u dvema nitima ne može istovremeno izvršavati blok naredbi koji je sinhronizovan po istom objektu. Ako je jedna nit sinhronizovana po nekom objektu i druga nit pokušava da se sinhronizuje po istom objektu, onda je druga nit primorana da čeka dok prva nit ne završi sa sinhronizujućim objektom.

Realizacija ovog načina sinhronizacije zasnovana je na modelu „katanca" (engl. *lock*). Naime, pored uobičajenih polja i metoda, svaki objekat u Javi sadrži i jedan katanac koji u svakom trenutku može „zauzeti" samo jedna nit. Da bi se u nekoj niti izvršila sinhronizovana naredba ili sinhronizovani metod, ta nit mora zauzeti katanac sinhronizujućeg objekta. Ako je taj katanac trenutno slobodan, onda ga nit zauzima i odmah počinje sa izvršavanjem odgovarajućeg bloka naredbi ili metoda. Nakon završetka izvršavanja sinhronizovane naredbe ili sinhronizovanog metoda, nit oslobađa katanac sinhronizujućeg objekta. Ako druga nit pokuša da zauzme katanac koji već drži prva nit, onda druga nit mora da čeka dok prva nit ne oslobodi katanac. U stvari, druga nit prelazi u stanje spavanja i neće biti probuđena sve dok katanac ne bude na raspolaganju. Ovaj postupak je ilustrovan na slici 9.8.



Slika 9.8: Sinhronizacija niti pomoću katanca.

Mrtva petlja

Sinhronizacijom se rešava problem trke niti, ali se i uvodi mogućnost pojave druge vrste greške koja se naziva *mrtva petlja* (engl. *deadlock*). Mrtva petlja nastaje kada jedna nit bezuspešno čeka na neki resurs koji nikada ne može dobiti. Do mrtve petlje može doći kada se niti sinhronizuju po nekoliko istih objekata. Onda se može desiti da, recimo, dve niti zauzimaju katance dva različita objekta i obe niti čekaju da druga nit oslobodi katanac drugog objekata kako bi mogle da nastave rad. Razmotrimo pažljivije primer na slici 9.9 na kojoj su prikazane dve niti A i B koje se međusobno sinhronizuju po dva objekta `o1` i `o2`.

Pretpostavimo da se kod prvih sinhronizovanih naredbi niti A i B tokom izvršavanja programa desio slučaj da je nit A zauzela katanac objekta `o1` i da je odmah iza toga nit B zauzela katanac objekta `o2`. Kada se tokom daljeg izvršavanja u ovim nitima dođe do drugih sinhronizovanih naredbi, niti A i B će da čekaju jedna drugu da oslobode katanac koji jedna nit tre-

nit A

```
synchronized (o1) {
   .
   . // naredbe
   .
   synchronized (o2) {
      .
      . // naredbe
      .
   }
}
```

nit B

```
synchronized (o2) {
   .
   . // naredbe
   .
   synchronized (o1) {
      .
      . // naredbe
      .
   }
}
```

Slika 9.9: Mrtva petlja ukoliko A zauzme `o1` i B zauzme `o2`.

ba i koji druga zauzima. Drugim rečima, nijedna nit ne može da nastavi sa izvršavanjem i obe su blokirane zbog mrtve petlje.

Da ne bi došlo do mrtve petlja niti koje se sinhronizuju po više objekata, u programu se moraju koristiti posebne tehnike. Na primer, sprečavanje pojave mrtve petlje niti može se lako postići jednostavnim *uređenjem resursa*. Ovaj pristup sastoji se u tome da se najpre svi sinhronizujući objekti urede po nekom redosledu i zatim da se obezbedi da sve niti zauzimaju katance ovih objekata tim redosledom.

U prethodnom primeru pretpostavimo da je, recimo, redosled sinhronizujućih objekata `o1` i `o2`. Onda se nit B mora prvo sinhronizovati po objektu `o1`, pa zatim po objektu `o2`. Kada nit A zauzme katanac objekta `o1`, nit B će morati da čeka na katanac objekta `o1` i neće moći da zauzme katanac objekta `o2` koji dalje treba niti A. Zato će nit A dalje moći da zauzme katanac objekta `o2` i da završi rad, kada će nit B nastaviti sa svojim radom. Time je dakle sprečena pojava mrtve petlje ove dve niti.

Nit za izvršavanje grafičkih događaja

Svaki program u Javi se sastoji od bar jedne niti u kojoj se izvršava metod `main()` glavne klase. U grafičkom programu se automatski dodatno pokreće posebna nit u kojoj se izvršava svako rukovanje GUI događajima i svako crtanje komponenti na ekranu. Konstrukcija posebne niti je ne-

ophodna kako bi se obezbedilo da svaki rukovalac događajima u grafičkim programima završi svoj rad pre sledećeg rukovaoca i da događaji ne prekidaju proces crtanja na ekranu. Ova „grafička" nit se naziva *nit za izvršavanje događaja* (engl. *event dispatcher thread*).

Da bi se kod grafičkih programa prevazišla mogućnost mrtve petlje niti u nekim slučajevima, potrebno je izvršiti dodatne naredbe u niti za izvršavanje grafičkih događaja. U tu svrhu se koriste statički metodi `invokeLater()` i `invokeAndWait()` iz klase `SwingUtilities` u paketu `javax.swing`. Naredbe koje treba izvršiti u niti za izvršavanje grafičkih događaja moraju se pisati unutar metoda `run()` jednog objekta tipa `Runnable`, a ovaj objekat se navodi kao argument metoda `invokeLater()` i `invokeAndWait()`.

Razlika između metoda `invokeLater()` i `invokeAndWait()` je u tome što se metod `invokeLater()` odmah završava nakon poziva metoda `run()`, bez čekanja da se on završi u niti za izvršavanje grafičkih događaja. S druge strane, metod `invokeAndWait()` se ne završava odmah nego tek nakon što se potpuno završi izvršavanje metoda `run()` u niti za izvršavanje grafičkih događaja.

Do sada su grafički programi u Javi bili pisani tako što su se u metodu `main()` napre konstruisali glavni okviri programa, a zatim su se ovi okviri prikazivali na ekranu tako što su se učinili vidljivim. Ovo je u redu za većinu programa, ali u nekim slučajevima to može dovesti do problema mrtve petlje. Zbog toga se, da bi grafički program bio apsolutno ispravan, taj postupak mora izvršiti u niti za izvršavanje grafičkih događaja na sledeći način:

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

   SwingUtilities.invokeLater(new Runnable() {
      public void run() {
         .
         . // Naredbe za konstruisanje okvira
         . // i njegovo podešavanje
         .
      }
   });
}
```

Obratite ovde pažnju na to da je argument metoda `invokeLater()` jedan objekat tipa `Runnable` koji pripada anonimnoj klasi. U listingu 9.4 je prikazan najjednostavniji grafički program u kome se glavni okvir pokreće iz niti za izvršavanje grafičkih događaja.

Listing 9.4: Nit za izvršavanje grafičkih događaja

```
import javax.swing.*;

public class GrafičkiProgram {

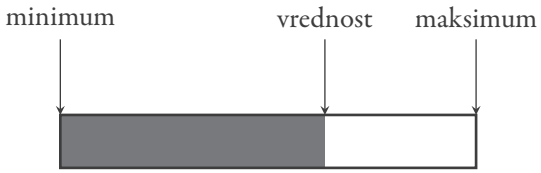
   public static void main(String[] args) {

      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            JFrame frame = new JFrame("Grafički program");
            frame.add(new JLabel(
                           "Test niti za izvršavanje događaja"));
            frame.setLocationRelativeTo(null);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(250, 200);
            frame.setVisible(true);
         }
      });
   }
}
```

Primer: prikaz procenta završetka zadatka

Ukoliko se u grafičkom programu neki postupak dugo izvršava, korisno je na neki način prikazati koliki deo tog postupka je završen kako bi korisnik imao povratnu informaciju o tome kada će se postupak otprilike završiti. Grafička komponenta `JProgressBar` u biblioteci Swing može poslužiti u takve svrhe, jer grafički prikazuje neku vrednost iz ograničenog intervala. Ovaj interval između neke minimalne i maksimalne vrednosti predstavljen je horizontalnom pravougaonom trakom koja se popunjava

sleva na desno (slika 9.10). Traka može biti prikazana i vertikalno, kada se popunjava odozdo na gore.



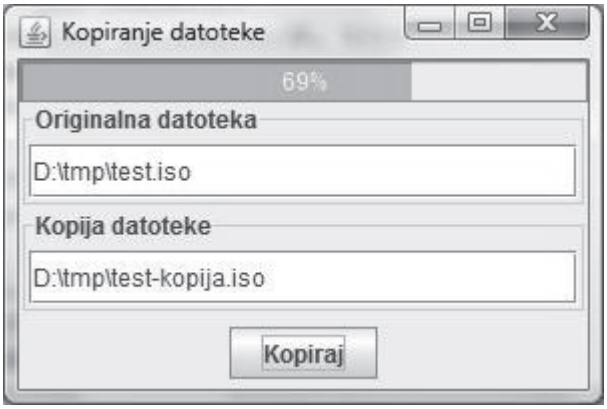
Slika 9.10: Traka za prikaz procenta završetka zadatka.

Klasa `JProgressBar` sadrži više metoda kojima se može manipulisati trakom za prikaz procenta završetka zadatka. Nepotpun spisak ovih metoda je:

- `JProgressBar()` — Konstruiše se horizontalna traka sa minimalnom vrednošću 0 i maksimalnom vrednosšću 100.
- `void setOrientation(int p)` — Menja se položaj trake u horizontalni ili vertikalni.
- `void setValue(int v)` — Menja se aktuelna vrednost trake.
- `void setString(String s)` — Menja se aktuelni tekst koji se prikazuje unutar trake.
- `void setStringPainted(boolean b)` — Menja se indikator čija vrednost ukazuje da li se tekst unutar trake prikazuje ili ne.

Traka za prikaz procenta završetka zadatka pokreće se često u posebnoj niti radi pokazivanja stanja u kome se nalazi izvršavanje zadataka drugih niti. Ovaj pristup je ilustrovan u narednom grafičkom programu u kome se kopira izabrana datoteka u drugu datoteku, uz istovremeni prikaz procenta završenog kopiranja. Izgled prozora ovog programa, sa trakom za prikaz procenta završenog kopiranja na vrhu, ilustrovan je na slici 9.11.

U glavnoj klasi `KopiranjeDatoteke` ovog programa koji je prikazan u listingu 9.5, formira se samo osnovni korisnički interfejs, bez obraćanja mnogo pažnje na ostale detalje oko otkrivanja grešaka i izveštavanja korisnika. U ovoj glavnoj klasi je definisana unutrašnja klasa `ZadatakKopiranja` koja



Slika 9.11: Kopiranje datoteke i procenat završenog kopiranja.

implementira interfejs `Runnable` radi kopiranja date datoteke. Kada se pritisne dugme za početak kopiranja, konstruiše se i pokreće se posebna nit u kojoj se izvodi zadatak kopiranja date datoteke. Kako se originalna datoteka kopira u drugu datoteku, povremeno se ažurira traka za prikaz procenta završetka kopiranja.

Listing 9.5: Prikaz procenta završenog kopiranja datoteke

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;

public class KopiranjeDatoteke extends JFrame {

   private JProgressBar traka = new JProgressBar();
   private JButton dugme = new JButton("Kopiraj");
   private JTextField original = new JTextField();
   private JTextField kopija = new JTextField();

   // Konstruktor
   public KopiranjeDatoteke() {
```

```
        JPanel panel1 = new JPanel();
        panel1.setLayout(new BorderLayout());
        panel1.setBorder(new TitledBorder("Originalna datoteka"));
        panel1.add(original, BorderLayout.CENTER);

        JPanel panel2 = new JPanel();
        panel2.setLayout(new BorderLayout());
        panel2.setBorder(new TitledBorder("Kopija datoteke"));
        panel2.add(kopija, BorderLayout.CENTER);

        JPanel panel3 = new JPanel();
        panel3.setLayout(new GridLayout(2, 1));
        panel3.add(panel1);
        panel3.add(panel2);

        JPanel panel4 = new JPanel();
        panel4.add(dugme);

        this.add(traka, BorderLayout.NORTH);
        this.add(panel3, BorderLayout.CENTER);
        this.add(panel4, BorderLayout.SOUTH);

        traka.setStringPainted(true); // prikazati tekst unutar trake

        dugme.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Konstruisanje i pokretanje niti za kopiranje datoteke
                new Thread(new ZadatakKopiranja()).start();
            }
        });
    }

    public static void main(String[] args) {

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JFrame okvir = new KopiranjeDatoteke();
```

```
            okvir.setLocationRelativeTo(null);
            okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            okvir.setTitle("Kopiranje datoteke");
            okvir.setSize(300, 200);
            okvir.setVisible(true);
         }
      });
   }

   // Unutrašnja klasa za zadatak kopiranja jedne datoteke i
   // ažuriranja procenta završenog kopiranja te datoteke
   class ZadatakKopiranja implements Runnable {
      private int p; // procenat završenog kopiranja

      public void run() {
         BufferedInputStream ulaz = null;
         BufferedOutputStream izlaz = null;
         try {
            // Konstruisanje ulaznog toka
            File imeUlaza = new File(original.getText().trim());
            ulaz = new BufferedInputStream(
                           new FileInputStream(imeUlaza));

            // Konstruisanje izlaznog toka
            File imeIzlaza = new File(kopija.getText());
            izlaz = new BufferedOutputStream(
                            new FileOutputStream(imeIzlaza));

            long n = ulaz.available();  // veličina ulazne datoteke
                                        // ... u bajtovima
            traka.setValue(0);          // početna vrednost trake

            long k = 0;  // aktuelni broj upisanih bajtova datoteke
            byte[] bafer = new byte[256];  // prostor za privremeno
                                           // ... učitane bajtove
            int b;  // broj privremeno učitanih/upisanih bajtova
```

```
            // Kopiranje datoteke i ažuriranje vrednosti trake
            while ((b = ulaz.read(bafer, 0, bafer.length)) != -1) {
               izlaz.write(bafer, 0, b);
               k = k + b;
               p = (int)(k * 100 / n);  // ažuriranje procenta
                                        // ... završenog kopiranja
               traka.setValue(p);       // ... i vrednosti trake
            }
         }
         catch (FileNotFoundException e) {
            e.printStackTrace();
         }
         catch (IOException e) {
            e.printStackTrace();
         }
         finally {
            try {
               if (ulaz != null)
                  ulaz.close();
               if (izlaz != null)
                  izlaz.close();
            }
            catch (Exception e) {
            }
         }
      }
   }
}
```

Obratite pažnju na to da je izvođenje kopiranja datoteke u posebnoj niti od suštinske važnosti za ispravnost ovog programa. U suprotnom slučaju, slika trake za prikaz procenta završenog kopiranja ne bi se ažurirala sve dok se ceo postupak kopiranja ne završi! Naime, obrada akcijskog događaja dugmeta za pokretanje kopiranja datoteke, odnosno izvršavanje metoda `actionPerformed()`, odvija se u istoj niti za izvršavanje grafičkih događaja kao i operacija crtanja trake za prikaz procenta završenog zadatka na ekra-

nu. To znači da slika trake u celom vremenskom intervalu od početka do završetka kopiranja datoteke ne bi mogla da se ažurira na ekranu (mada bi se sama vrednost trake povremeno ažurirala). Zbog toga, kopiranjem datoteke u posebnoj niti omogućeno je paralelno crtanje trake za prikaz procenta završenog kopiranja tako da slika trake zaista odražava trenutno stanje kopiranja datoteke.

9.4 Kooperacija niti

Niti mogu međusobno sarađivati i na druge načine osim deljenjem resursa. Na primer, jedna nit može proizvoditi neki rezultat koji koristi druga nit. To onda uslovljava određena ograničenja na poredak po kome se niti moraju paralelno izvršavati. Ako druga nit dođe do tačke u kojoj joj je potreban rezultat prve niti, druga nit možda mora da zastane i sačeka, jer prva nit još nije proizvela potreban rezultat. Pošto druga nit ne može da nastavi, ona prelazi u neaktivno, blokirano stanje. Ali onda mora postojati i neki način da druga nit bude obaveštena kada je rezultat prve niti spreman, kako bi se druga nit aktivirala i nastavila svoje izvršavanje.

U Javi se ova vrsta čekanja i obaveštavanja niti može postići metodima `wait()` i `notify()` koji su definisani kao objektni metodi u klasi `Object`. Opšti princip je da kada se u jednoj niti pozove metod `wait()` za neki objekat, ta nit prelazi u blokirano stanje dok se ne pozove metod `notify()` za isti objekat. Metod `notify()` mora se očigledno pozvati u drugoj niti, jer se prva nit ne izvršava u blokiranom stanju.

Tipični postupak je dakle da se u prvoj niti A pozove metod `wait()` kada je potreban rezultat druge niti B koji još nije spreman. S druge strane, kada se u niti B proizvede potreban rezultat, onda se poziva metod `notify()` da bi se nit A aktivirala i iskoristila raspoloživ rezultat. Pozivanje metoda `notify()` kada nit A ne čeka nije greška, već prosto takav poziv nema nikakvog efekta. Drugim rečima, ukoliko je `obj` neki objekat, u niti A se izvršava programski fragment koji ima otprilike ovaj oblik

```
if (!rezultatSpreman())  // rezultat nije spreman?
   obj.wait();           // čekati dok rezultat ne bude spreman
koristiRezultat();       // koristiti rezultat pošto je spreman
```

dok se istovremeno u niti B izvršava ovaj programski fragment:

```
proizvediRezultat();
obj.notify();  // poslati obaveštenje da je rezultat spreman
```

Pažljivi čitaoci su možda primetili da ovo rešenje nije potpuno ispravno, jer može dovesti do mrtve petlje. Naime, niti A i B se mogu izvršavati po sledećem redu:

1. u niti A se izvršava naredba `if` i nalazi se da rezultat nije spreman;
2. pre nego što se u niti A izvrši poziv `obj.wait()`, u niti B se proizvede potreban rezultat i izvrši se poziv `obj.notify()`;
3. u niti A se izvršava poziv `obj.wait()` radi čekanja na obaveštenje kada rezultat bude spreman.

Sada će u trećem koraku nit A čekati na obaveštenje koje nikad neće doći, jer se u niti B već izvršio poziv `obj.notify()`. Ovaj mogući način izvršavanja niti A i B dovodi dakle do beskonačnog čekanja niti A da nastavi rad.

Očigledno je da se cela naredba `if` u niti A mora izvršiti bez prekidanja i zato je neophodno sinhronizovati niti A i B. Drugime rečima, programske fragmente obe niti treba pisati unutar sihronizovanih naredbi. Za sinhronizujući objekat prirodno je koristiti isti objekat `obj` koji se koristi za pozivanje metoda `wait()` i `notify()`. U stvari, u Javi je ovo prirodno rešenje obavezno: u nekoj niti se mogu izvršiti metodi `obj.wait()` i `obj.notify()` samo za vreme kada je katanac objekta `obj` u posedu te niti. Ukoliko to nije slučaj, izbacuje se izuzetak tipa `IllegalMonitorStateException`. Rukovanje ovim izuzetkom nije obavezno i u programu se taj izuzetak obično ne hvata. Ali metod `wait()` može izbaciti proveravani izuzetak tipa `InterruptedException`, pa se poziv tog metoda mora nalaziti unutar naredbe `try` u kojoj se hvata ovaj izuzetak.

Prethodni primer niti A i B je zapravo uvodni deo opšteg problema proizvođač/potrošač u kome jedna nit proizvodi neki rezultat koji se „troši" u drugoj niti. Pretpostavimo da se rezultat proizvođača prenosi potrošaču preko deljive promenljive `rezultat` koja početno ima vrednost `null`. Nakon što proizvede potreban rezultat, proizvođač ga upisuje u tu promenljivu. Potrošač može proveriti da li je rezultat spreman proveravanjem te promenljive da li ima vrednost `null`. Ako se koristi objekat `katanac` za sinhronizaciju, a proizvodnja i korišćenje rezultata se simbolički predstavi me-

todima `proizvediRezultat()` i `koristiRezultat()`, onda proizvođač ima ovaj oblik:

```
r1 = proizvediRezultat(); // nije sinhronizovano!
synchronized (katanac) {
   rezultat = r1;
   katanac.notify();
}
```

S druge strane, potrošač ima ovaj oblik:

```
synchronized (katanac) {
   while (rezultat == null) {
      try {
         katanac.wait();
      }
      catch (InterruptedException e) {
      }
   }
   r2 = rezultat;
}
koristiRezultat(r2); // nije sinhronizovano!
```

Obratite pažnju na to da postupci proizvodnje i korišćenja rezultata nisu sinhronizovani kako bi se mogli paralelno izvršavati sa drugim nitima koje su možda sinhronizovane po istom objektu `katanac`. Pošto je `rezultat` deljiva promenljiva, njeno korišćenje mora biti sinhronizovano. Zato se svako pozivanje na promenljivu `rezultat` mora nalaziti unutar sinhronizovanih naredbi. Ali cilj je i da se obezbedi što veća paralelnost rada proizvođača i potrošača, odnosno da se što manje naredbi (ali ne preterano manje) nalazi u sinhronizovanim naredbama.

U prethodnom primeru, vrlo pažljivi čitaoci su možda primetili potencijalni problem sinhronizacije niti proizvođača i potrošača po istom objektu `katanac`. Naime, ako potrošač zauzme `katanac` i izvrši `katanac.wait()`, potrošač je blokiran dok proizvođač ne izvrši poziv `katanac.notify()`. Ali da bi proizvođač izvršio ovaj poziv, proizvođač mora zauzeti `katanac` koji već drži potrošač. Da li je ovo klasičan primer mrtve petlje dve niti? Odgovor je negativan, jer je `katanac.wait()` specijalan slučaj mehanizma sinhronizacije: odmah nakon izavršavanja `katanac.wait()` u niti potrošača,

ova nit automatski oslobađa katanac sinhronizujućeg objekta `katanac`. To daje priliku niti proizvođača da zauzme `katanac` i da se izvrši poziv `katanac.notify()` koji se nalazi unutar sinhronizovanog bloka. Nakon izvršavanja ovog sinhronizovanog bloka u niti proizvođača, `katanac` se vraća niti potrošača kako bi se ova nit mogla nastaviti.

Rešenje jednostavnog problema proizvođač/potrošač, u specijalnom slučaju u kome se proizvodi i koristi samo jedan rezultat, može se vrlo lako generalizovati. U opštem slučaju, jedan ili više proizvođača proizvodi više rezultata koji se koriste od strane jednog ili više potrošača. U tom slučaju se umesto samo jednog deljenog rezultata vodi evidencija o svim rezultatima koji su proizvedeni, a nisu još iskorišćeni. Ti rezulati mogu biti organizovani u obliku liste (dinamičkog niza) rezultata, pri čemu niti proizvođača dodaju proizvedene rezultate na kraj ove liste i niti potrošača uklanjaju rezultate sa početka ove liste radi korišćenja. Jedino vreme kada je neka nit prirodno blokirana i mora duže da čeka jeste kada nit potrošača pokušava da uzme jedan rezulat iz liste, a lista rezultata je prazna.

Pod pretpostavkom da `TipRezultata` predstavlja tip proizvedenih i korišćenih rezultata, opšti model proizvođač/potrošač može se lako obuhvatiti jednom klasom na sledeći način:

```
/**
 * Objekat tipa ProizvođačPotrošač predstavlja listu rezultata koji
 * se raspoloživi za korišćenje. Rezultati se u listu dodaju pozivom
 * metoda dodaj(), a iz liste se uklanjaju pozivom metoda ukloni().
 * Rezultati se dodaju u listu kako se proizvode, a uklanjaju iz
 * liste kako su potrebni za korišćenje.
 * Ako je lista rezultata prazna kada se pozove metod ukloni(), taj
 * metod se ne završava dok neki rezultat ne bude spreman.
 */

public class ProizvođačPotrošač {

   // Lista proizvedenih rezultata koji su spremni za korišćenje
   private ArrayList<TipRezultata> rezultati =
                                       new ArrayList<TipRezultata>();

   public void dodaj(TipRezultata r) {
```

```
      synchronized (rezultati) {
         rezultati.add(r);    // dodati rezultat u listu
         rezultati.notify();  // obavestiti nit koja čeka u metodu
                              // ... ukloni() da je rezultat spreman
      }
   }

   public TipRezultata ukloni() {

      TipRezultata r;
      synchronized (rezultati) {
         // Ako je lista prazna, čekati na obaveštenje iz metoda
         // dodaj() kada bar jedan rezultat bude spreman
         while (rezultati.size() == 0) {
            try {
               rezultati.wait();
            }
            catch (InterruptedException e) {
            }
         }
        // U ovoj tački je raspoloživ bar jedan rezultat
        r = rezultati.remove(0);
      }
      return r;
   }
}
```

Na kraju istaknimo još dva detalja u vezi sa metodima `wait()` i `notify()`. U programu se može desiti da je nekoliko niti blokirano čekajući na obaveštenje da mogu nastaviti sa radom. Poziv `obj.notify()` će aktivirati samo jednu od niti koje čekaju na `obj`. Metod `notifyAll()` služi za obaveštavanje svih niti: pozivom `obj.notifyAll()` aktiviraće se sve niti koje čekaju na `obj`.

Metod `wait()` se može pozvati i sa jednim argumentom koji predstavlja broj milisekundi. Nit u kojoj se izvršava `obj.wait(ms)` čekaće na obaveštenje da može nastaviti sa radom, ali najviše onoliko koliki je broj milisekundi naveden kao argument. Ako obaveštenje ne stigne u tom intervalu, nit će se

nakon isteka intervala aktivirati i nastaviti izvršavanje. U praksi se ova mogućnost najčešće koristi da bi se nit koja čeka aktivirala s vremena na vreme kako bi se u njoj mogao uraditi neki periodični zadatak — na primer, da bi se prikazala poruka da nit i dalje čeka na završetak određene aktivnosti.

GLAVA 10

Mrežno programiranje

Da bi računari mogli da međusobno šalju i primaju podatke, oni moraju biti fizički povezani u mrežu. Broj računara i njihova prostorna rasprostranjenost u mreži obično određuje to da li se mreža računara smatra lokalnom, regionalnom ili globalnom (Internet). Fizička povezanost računara ostvaruje se raznim sredstvima: žičanim kablovima, optičkim kablovima, satelitskim vezama, bežičnim vezama i tako dalje. Ali pored hardverske povezanosti, za komunikaciju računara u mreži je potrebna i njihova softverska povezanost, odnosno potrebni su programi u njima koji zapravo međusobno razmenjuju podatke. U ovom poglavlju se opširnije govori o pisanju takvih programa u Javi.

10.1 Komunikacija računara u mreži

Jedan program na nekom računaru komunicira sa drugim programom na drugom računaru u mreži na vrlo sličan način na koji se obavlja ulaz i izlaz programa preko datoteka. Mreža se, pojednostavljeno rečeno, smatra samo još jednim mogućim izvorom i prijemnikom podataka. Naravno, rad sa mrežom nije tako jednostavan kao rad sa datotekama na disku, ali u Javi se za mrežno programiranje koriste ulazni i izlazni tokovi podataka, baš kao što se to radi u opštem slučaju programskog ulaza i izlaza. Ipak treba imati u vidu da uspostavljanje veze između dva računara zahteva dodatni napor u odnosu na prosto otvaranje datoteke, jer dva računara na neki način moraju da se slože oko otvaranja konekcije između njih. Dodatno, slanje podata-

ka između dva računara zahteva sinhronizaciju kako bi jedan računar bio spreman da primi podatke koje drugi računar šalje.

Komunikacija računara u mreži odvija se na unapred precizno definisan način koji se naziva *protokol*. Dva osnovna protokola na kojima se zasniva komunikacija između računara nazivaju se *Internet Protocol* (*IP*) i *Transmission Control Protocol* (*TCP*). Ovaj par protokola je međusobno vrlo povezan tako da se obično zajednički označava skraćenicom *TCP/IP*. IP je protokol niskog nivoa za slanje i primanje podataka između dva računara u manjim delovima koji se nazivaju *paketi*. TCP je protokol višeg nivoa koji se oslanja na IP radi uspostavljanja logičke veze između dva računara i obezbeđivanja tokova podataka između njih. Pored toga, TCP je protokol kojim se garantuje prenos svih paketa od jednog do drugog računara i to onim redom kojim su poslati.

U stvari, u nekim specijalnim primenama, umesto TCP protokola može se koristiti drugi osnovni protokol koji se naziva *User Datagram Protocol* (*UDP*). UDP je efikasniji protokol od TCP-ja, ali se njegovom primenom ne garantuje pouzdana dvosmerna komunikacija. U Javi se mogu koristiti oba protokola TCP i UDP za mrežno programiranje, ali se u ovoj knjizi podrazumeva TCP protokol zbog njegove mnogo šire primene u praksi.

Pored korišćenja protokola, radi komuniciranja u mreži neki računar mora na neki način da naznači sa kojim drugim računarom od svih onih u mreži želi da komunicira. Zbog toga svaki računar u mreži ima jedinstvenu *IP adresu* po kojoj se razlikuje od drugih računara. IP adrese su 32-bitni celi brojevi koji se obično pišu u obliku četiri decimalna broja odvojena tačkama, na primer `192.168.1.8`. Svaki od ova četiri broja predstavlja 8-bitni ceo broj iz intervala od 0 do 255 u grupi od 32 bita redom sleva na desno. Tačnije, ovo su tradicionalne adrese koje se koriste u starijoj verziji 4 IP protokola, pa se ponekad radi preciznosti nazivaju IPv4 adrese. Najnovija verzija 6 IP protokola koristi 128-bitne cele brojeve za adrese koji se nazivaju IPv6 adrese.

Kako nije lako raditi sa toliko mnogo brojeva kojima se označavaju različiti računari, mnogi računari imaju i smislena imena koja se lakše pamte, na primer `java.sun.com` ili `www.singidunum.ac.rs`. Ova *domenska imena* su takođe jedinstvena, ali nisu zamena za IP adrese nego samo olakšavaju rad sa računarima u mreži. Kada jedan računar šalje zahtev drugom računa-

ru koristeći njegovo domensko ime radi upostavljanja veze, ovo domensko ime drugog računara mora se najpre prevesti u njegovu IP adresu i tek onda se može poslati zahtev na odgovarajuću IP adresu. Zadatak prevođenja domenskih imena u IP adrese obavljaju posebni računari u mreži koji se nazivaju *DNS serveri*.

Jedan računar može imati više IP adresa, kako IPv4 tako i IPv6, kao i više domenskih imena. Jedna od ovih IP adresa je standardno `127.0.0.1` koja se koristi kada jedan program treba da komunicira sa drugim programom na istom računaru. Ova IP adresa se naziva *adresa petlje* (engl. *loopback address*) i za nju se tradicionalno koristi domensko ime *localhost*.

Standardni paket u Javi koji sadrži klase za mrežno programiranje naziva se `java.net`. Klase u ovom paketu obezbeđuju dva načina mrežnog programiranja koji se grubo mogu nazvati veb programiranje i klijent-server programiranje. Prvi način je lakši način koji se zasniva na veb protokolima viskog nivoa koji se nalaze iznad osnovnih TCP/IP protokola. Veb protokoli kao što su HTTP, FTP i drugi koriste TCP/IP protokole za neposredno slanje i primanje podataka, ali za veb programiranje nije potrebno poznavanje svih detalja njihovog rada. Na ovaj način se u programu mogu realizovati mogućnosti slične onima koje imaju standardni veb čitači (Internet Explorer, Mozilla Firefox, ...). Glavne klase za ovaj stil mrežnog programiranja nalaze se u paketima `java.net.URL` i `java.net.URLConnection`. Objekat tipa `URL` predstavlja apstrakciju adrese nekog HTML dokumenta ili drugog resursa na vebu. Ova adresa se tehnički naziva *univerzalni lokator resursa*, a skraćenica URL potiče od engleskog termina *Universal Resource Locator* za tu adresu. S druge strane, objekat tipa `URLConnection` predstavlja otvorenu mrežnu konekciju sa nekim veb resursom.

Druga vrsta mrežnog programiranja u Javi je opštija, ali zato i mnogo nižeg nivoa. Ona se zasniva na TCP/IP protokolima i na modelu *klijent-server* za komunikaciju dva računara u mreži. Ipak, radi pojednostavljenja ovog načina programiranja, mnogi tehnički detalji TCP/IP protokola apstrahovani su konceptom *soketa* (engl. *socket*, u prevodu „utikač"). Program koristi soket za uspostavljanje konekcije sa drugim programom na drugom računaru u mreži. Komunikacija dva računara preko mreže obuhvata zapravo dva soketa, po jedan na svakom računaru koji međusobno komuniciraju. (Ovo je verovatno i razlog za termin „utikač" kojim se pokušava izraziti ana-

logija sa fizičkim stavljanjem kabla u jedan „utikač" na računaru radi povezivanja računara u mrežu.) Klasa `java.net.Socket` služi za predstavljanje soketa koji se koriste za mrežnu komunikaciju u programu. Odmah treba naglasiti da soketi koji pripadaju klasi `Socket` jesu objekti koji i nemaju baš mnogo veze sa predstavom stvarnih „utikača". Naime, neki program može imati nekoliko soketa u isto vreme i svaki od njih može povezivati taj program sa drugim programom koji se izvršava na različitom računaru u mreži. Pri tome, sve ove konekcije se uspostavljaju preko istih fizičkih veza između računara u mreži.

10.2 Veb programiranje

Svaki „resurs" na vebu (dokument, datoteka, slika, ...) ima svoju adresu koja ga jedinstveno označava. Na primer, osnovna stranica sajta za Javu ima adresu `http://java.sun.com/index.html`. Ova adresa se naziva *univerzalni lokator resursa* (engl. *Universal Resource Locator*) ili u žargonu kraće *url*. Univerzalni lokator resursa služi da bi se u veb čitaču tačno odredilo koji resurs se traži kako bi ga veb čitač mogao naći na vebu.

Objekat koji pripada klasi `URL` predstavlja jedinstvenu url adresu resursa na vebu. Taj objekat služi da se metodima iz klase `URLConnection` uspostavi konekcija sa datim resursom na odgovarajućoj adresi. Puna url adresa se obično navodi kao string, recimo `"http://java.sun.com/index.html"`. Puna url adresa resursa se može odrediti i relativno u odnosu na drugu url adresu koja se naziva *osnova* ili *kontekst*. Na primer, ako je osnova jednaka `http://java.sun.com/`, onda relativna url adresa u obliku `index.html` ukazuje na resurs čija je puna adresa `http://java.sun.com/index.html`.

Objekat klase `URL` nije običan string, već se može konstruisati na osnovu url adrese u obliku stringa. Objekat klase `URL` može se konstruisati i na osnovu drugog objekta tipa `URL`, koji predstavlja osnovu, kao i stringa koji određuje relativni url u odnosu na tu osnovu. Zaglavlja odogovarajućih konstruktora klase `URL` su:

```
public URL(String url)
       throws MalformedURLException
```

```
public URL(URL osnova, String relativni-url)
        throws MalformedURLException
```

Primetimo da oba konstruktora izbacuju poseban izuzetak ukoliko navedeni stringovi ne predstavljaju sintaksno ispravan url. Ovaj izuzetak je objekat klase `MalformedURLException` koja je naslednica klase `IOException`. To znači da se izuzetkom tipa `MalformedURLException` mora obavezno rukovati u programu. Drugim rečima, konstruktori klase `URL` moraju se nalaziti unutar naredbe `try` u kojoj se hvata ovaj izuzetak ili unutar metoda u čijem zaglavlju se nalazi klauzula `throws` kojom se naznačava mogućnost izbacivanja tog izuzetka.

Drugi konstruktor se naročito koristi za pisanje apleta. U klasi `JApplet` su definisana dva metoda koja kao razultat daju dve korisne osnove za relativni url. Prvi metod je `getDocumentBase()` čiji je rezultat objekat tipa `URL`. Ovaj objekat predstavlja url adresu veb strane koja sadrži aplet. Na taj način se u apletu mogu preuzeti dodatne datoteke koje se nalaze na istoj adresi. Pisanjem, na primer,

```
URL url = new URL(getDocumentBase(), "podaci.txt");
```

u apletu se dobija url koji ukazuje na datoteku `podaci.txt` koja se nalazi na istom računaru i u istom direktorijumu kao i HTML dokument koji sadrži aplet.

Drugi, sličan metod u klasi `JApplet` je `getCodeBase()` čiji je rezultat objekat tipa `URL` koji predstavlja url adresu datoteke sa bajtkodom apleta. Naime, lokacije datoteka koje sadrže opis veb strane na HTML jeziku i bajtkod apleta ne moraju biti iste.

U klasi `URL` je definisan metod `openConnection()` kojim se uspostavlja konekcija sa resursom na datoj url adresi. Ovaj metod kao rezultat daje objekat tipa `URLConnection` koji predstavlja otvorenu konekciju sa resursom. Taj objekat može se dalje koristiti za formiranje ulaznog toka radi čitanja podataka iz resursa na odgovarajućoj url adresi. Formiranje ulaznog toka preko otvorene konekcije postiže se metodom `getInputStream()`. Metodi `openConnection()` i `getInputStream()` mogu izbaciti proveravani izuzetak tipa `IOException` kojim se mora rukovati u programu. Na primer, u sledećem fragmentu se konstruiše (binarni) ulazni tok za čitanje resursa koji se nalazi na adresi datoj stringom `urlString`:

```
URL url = new URL(urlString);
URLConnection urlVeza = url.openConnection();
InputStream urlUlaz = urlVeza.getInputStream();
```

Dalji postupak čitanja podataka iz ulaznog toka odvija se na potpuno isti način kao što se to radi za datoteku na disku — na primer, „umotavanjem" binarnog toka u tekstualni tok mogu se čitati tekstualni podaci iz resursa na vebu.

Prilikom čitanja nekog resursa na vebu obično je korisno znati vrstu informacija koje sadrži taj resurs. Za tu svrhu služi metod `getContentType()` u klasi `URLConnection` koji kao rezultat vraća string koji opisuje sadržaj resursa na vebu. Ovaj string ima specijalni oblik koji se naziva *mime format*, na primer: „text/plain", „text/html", „image/jpeg", „image/gif" i tome slično. U opštem slučaju, mime format sastoji se od dva dela. Prvi deo opisuje opšti tip informacija koje sadrži resurs, recimo da li je to tekst ili slika („text" ili „image"). Drugi deo mime formata bliže određuje vrstu informacija unutar opšte kategorije iz prvog dela, recimo da li je tekst običan („plain") ili je slika u „gif" zapisu. Mime format je prvobitno uveden za opis sadržaja poruka elektronske pošte, ali se danas skoro univerzalno koristi za opis sadržaja bilo koje datoteke ili drugog resursa na vebu. (Naziv mime za ovaj format potiče od skraćenice engleskog termina *Multipurpose Internet Mail Extensions*.)

Vraćena vrednost metoda `getContentType()` može biti `null` ukoliko se ne može odrediti vrsta informacija koje sadrži resurs na vebu ili to nije još poznato u trenutku poziva metoda. Ovaj drugi slučaj se može desiti pre formiranja ulaznog toka, pa da se metod `getContentType()` obično poziva posle metoda `getInputStream()`.

Radi boljeg razumevanja prethodnih pojmova, u nastavku je prikazan metod za čitanje tekstualne datoteke na datoj url adresi. U tom metodu se otvara konekcija sa datim resursom, proverava se da li ovaj sadrži tekstualnu vrstu informacija i zatim se prikazuje njegov sadržaj na ekranu. Mnoge naredbe u ovom metodu mogu izbaciti izuzetke, ali se njima rukuje na jednostavan način tako što metod u zaglavlju sadrži klauzulu `throws IOException`. Time se glavnom programu prepušta odluka o tome šta konkretno treba uraditi u slučaju neke greške.

```
public void prikažiTekstURL(String urlString)
   throws IOException {

   // Otvoranje konekcije sa resursom na url adresi
   // i formiranje ulaznog toka preko te konekcije
   URL url = new URL(urlString);
   URLConnection urlVeza = url.openConnection();
   InputStream urlUlaz = urlVeza.getInputStream();

   // Proveravanje da li je sadržaj resursa neka forma teksta
   String mime = urlVeza.getContentType();
   if ((mime == null) || (mime.startsWith("text") == false))
      throw new IOException(
         "Url ne ukazuje na tekstualnu datoteku");

   // Kopiranje redova teksta iz ulaznog toka na ekran dok se
   // ne naiđe na kraj datoteke (ili se ne desi neka greška)
   BufferedReader ulaz;
   ulaz = new BufferedReader(new InputStreamReader(urlUlaz));

   while (true) {
      String jedanRed = ulaz.readLine();
      if (jedanRed == null)
         break;
      System.out.println(jedanRed);
   }
}
```

Obratite pažnju na to da kada se ovaj metod koristi za prikazivanje neke datoteke na vebu, mora se kao argument navesti pun oblik njene url adrese koji počinje prefiksom `http://`.

Primer: jednostavni veb čitač

Pored prikazivanja običnog teksta datoteke na datoj url adresi, u Javi se na jednostavan način može i interpretirati sadržaj neke datoteke napisan u HTML ili RTF formatu i prikazati „prava" slika tog sadržaja. Naime, u bi-

blioteci Swing se nalazi klasa `JEditorPane` koja predstavlja grafičku komponentu za automatsko prikazivanje kako datoteka sa običnim tekstom, tako i datoteka sa tekstom u HTML ili RTF formatu. Upotrebom ove komponente ne moraju se pisati eksplicitne naredbe za čitanje datoteke na datoj url adresi, nego se prikazivanje njenog interpretiranog sadržaja postiže jednostavnim pozivom metoda:

```
public void setPage(URL url) throws IOException
```

Komponenta tipa `JEditorPane` predstavlja zapravo prozor pravog editora teksta. Međutim, zbog relativno skromnih mogućnosti za uređivanje teksta, ova komponenta se skoro uvek koristi samo za prikazivanje teksta (tj. njeno svojstvo `Editable` obično se podešava da bude `false`).

Komponenta tipa `JEditorPane` proizvodi događaj tipa `HyperlinkEvent` kada se u njenom prozoru klikne na neki hiperlink. Objekat događaja ovog tipa sadrži url adresu izabranog hiperlinka, koja se onda može koristiti u rukovaocu ovog događaja za metod `setPage()` radi prikazivanja odgovarajuće veb stranice.

Ove mogućnosti klase `JEditorPane` iskorišćene su u jednostavnom programu koji je prikazan u listingu 10.1. Ovaj program poseduje najosnovnije funkcije pravog veb čitača, kao što se može videti sa slike 10.1 na kojoj je prikazan glavni prozor programa. Ukoliko se unese url adresa HTML datoteke u tekstualnom polju glavnog prozora programa i pritisne taster Enter, sadržaj te datoteke se prikazuje u okviru editora tipa `JEditorPane`. Pored toga, ukoliko se klikne na neki hiperlink unutar prikazanog sadržaja, prikazuje se novi sadržaj odgovarajućeg resursa.

Listing 10.1: Jednostavni veb čitač

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class VebČitač extends JFrame {
```



Slika 10.1: Jednostavni veb čitač.

```
private JLabel urlOznaka;       // oznaka url polja
private JTextField urlPolje;    // url polje
private JEditorPane urlStrana;  // sadržaj veb strane

// Konstruktor
public VebČitač(String naslov) {
   super(naslov);
   urlOznaka = new JLabel("URL: ");
   urlPolje = new JTextField();

   // Pridruživanje rukovaoca događaja pritiska na Enter
   urlPolje.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         try {
            URL url = new URL(urlPolje.getText().trim());
            urlStrana.setPage(url);  // prikaži HTML datoteku
         }
         catch (IOException ex) {
            System.out.println(ex);
         }
```

```
        }
    });

    urlStrana = new JEditorPane();
    urlStrana.setEditable(false);

    // Pridruživanje rukovaoca događaja pritiska na hiperlink
    urlStrana.addHyperlinkListener(new HyperlinkListener() {

       public void hyperlinkUpdate(HyperlinkEvent e) {
          if (e.getEventType() ==
                HyperlinkEvent.EventType.ACTIVATED)
             try {
                urlStrana.setPage(e.getURL());  // prikaži link
             }
             catch (IOException ex) {
                System.out.println(ex);
             }
       }
    });

    JPanel trakaURL = new JPanel();
    trakaURL.setLayout(new BorderLayout(3,3));
    trakaURL.add(urlOznaka, BorderLayout.WEST);
    trakaURL.add(urlPolje, BorderLayout.CENTER);

    JPanel sadržaj = new JPanel();
    sadržaj.setLayout(new BorderLayout(3,3));
    sadržaj.add(trakaURL, BorderLayout.NORTH);
    sadržaj.add(new JScrollPane(urlStrana), BorderLayout.CENTER);

    setContentPane(sadržaj);
    setBounds(50,100,800,600);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

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

```
        VebČitač jvč = new VebČitač("Jednostavni veb čitač");
        jvč.setVisible(true);
    }
}
```

10.3 Klijent-server programiranje

Da bi dva programa mogla da komuniciraju preko mreže, oba programa moraju da obrazuju po jedan soket koji predstavlja logičku krajnju tačku konekcije između programa. Nakon što se konstruišu soketi i uspostavi mrežna konekcija između njih, u programima se soketi dalje mogu koristiti kao obični izvori i prijemnici podataka na standardni način programskog ulaza/izlaza. Drugim rečima, za dvosmernu komunikaciju dva programa na različitim računarima, svaki program koristi ulazni i izlazni tok podataka preko uspostavljene konekcije između soketa. Podaci koje jedan program upisuje u svoj izlazni tok šalju se drugom računaru. Tamo se oni usmeravaju u ulazni tok programa na drugom kraju mrežne konekcije. Zato kada ovaj drugi program čita podatke iz svog ulaznog toka, on zapravo dobija podatke koji su primljeni preko mreže od prvog programa.

Čitanje i pisanje podataka preko otvorene mrežne konekcije nije dakle ništa drugačije od ulazno-izlaznih operacija za datoteke na disku. Najteži deo mrežnog programiranja je, u stvari, uspostavljanje same konekcije između dva programa. Taj postupak obuhvata dva soketa i sledi unapred određen niz koraka. Jedan program mora najpre konstruisati soket koji pasivno čeka dok ne stigne zahtev za uspostavljanje konekcije od nekog drugog soketa. Na suprotnoj strani potencijalne konekcije, drugi program konstruiše aktivni soket koji šalje zahtev za konekcijom pasivnom soketu koji čeka. Kada pasivni soket primi zahtev, on aktivnom soketu odgovara potvrdno na zahtev za konekcijom i time je konekcija uspostavljena.[1] Nakon toga, oba programa na logičkim krajevima konekcije mogu formirati ulazni i izlazni tok za primanje i slanje podataka preko otvorene konekcije.

[1] Pasivni soket može i odbiti zahtev aktivnog soketa za uspostavljanje konekcije.

Komunikacija između dva programa preko ovih tokova podataka se dalje odvija sve dok jedan od programa ne zatvori konekciju.

Program koji konstruiše soket koji čeka na zahtev za uspostavljanje konekcije naziva se *server*, a ovaj pasivni soket koji se u njemu koristi naziva se *serverski soket*. Drugi program koji se povezuje sa serverom naziva se *klijent*, a njegov aktivni soket kojim se šalje zahtev za uspostavljanje konekcije sa serverom naziva se *klijentski soket*. Osnovna ideja ovog modela *klijent-server programiranja* je da se server nalazi negde u mreži i da čeka na zahtev za uspostavljanje konekcije od nekog klijenta. Za server se pretpostavlja da nudi neku vrstu usluge koja je potrebna klijentima, a klijenti je mogu dobiti nakon uspostavljanja konekcije sa serverom.

U praksi je često potrebno da server može raditi sa više klijenata u isto vreme. Naime, kada jedan klijent uspostavi konekciju sa serverom, serverski soket ne obustavlja čekanje na nove zahteve za uspostavljanje konekcije od drugih klijenata. Zato dok se jedan klijent uslužuje (ili se to radi za više njih), drugi klijenti mogu uspostaviti konekciju sa serverom i istovremeno biti usluženi od strane servera. Kao što čitaoci verovatno pretpostavljaju, za obezbeđivanje ovog paralelizma u radu servera koriste se programske niti o kojima smo govorili u poglavlju 9.

Mada se to ne vidi zbog visokog nivoa veb programiranja, u klasi `URL` iz prethodnog odeljka koristi se zapravo klijentski soket za mrežnu komunikaciju. Na drugoj strani konekcije je serverski program koji prihvata zahteve za konekcijom od objekta tipa `URL`, zatim čita zahtev za specifičnom datotekom na serverskom računaru i, na kraju, odgovara tako što šalje sadržaj te datoteke klijentskom objektu tipa `URL`. Nakon prenosa svih podataka u datoteci, server zatvara konekciju.

Uspostavljanje konekcije između serveskog i klijentskog programa je dodatno iskomplikovano mogućnošću da na jednom računaru može raditi nekoliko programa koji istovremeno komuniciraju sa drugim programima preko mreže ili time što jedan program može komunicirati sa više programa preko mreže. Zbog toga se logički krajevi mrežne konekcije ne označavaju samo IP adresama odgovarajućih računara, nego i takozvanim brojem *porta*. Broj porta je običan 16-bitni ceo broj iz intervala od 0 do 65536, ali brojevi od 0 do 1024 su rezervisani za standardne veb servise. Prema tome, serverski program ne čeka prosto na zahtev za uspostavljanje konekcije od

klijenata — on čeka na te zahteve na specifičnom portu. Da bi uspostavio konekciju sa serverom, potencijalni klijent mora zato znati kako IP adresu (ili domensko ime) računara na kome server radi, tako i broj porta na kome server očekuje zahteve za uspostavljanje konekcije. Na primer, veb server obično očekuje zahteve na portu 80, dok emejl server to radi na portu 25. Drugim standardnim veb servisima su takođe dodeljeni standardni brojevi portova.

Serverski i klijentski soketi

U Javi se za uspostavljanje TCP/IP konekcije koriste klase `ServerSocket` i `Socket` iz paketa `java.net`. Objekat klase `ServerSocket` predstavlja serverski soket koji čeka na zahteve za uspostavljanje konekcije od klijenata. Objekat klase `Socket` predstavlja jednu krajnju tačku uspostavljene mrežne konekcije. Objekat ovog tipa može biti klijentski soket koji serveru šalje zahtev za uspostavljanje konekcije. Ali objekat tipa `Socket` može biti i soket koji server konstruiše radi uspostavljanja konekcije sa klijentom, odnosno radi obezbeđivanja druge logičke tačke jedne takve konekcije. Na taj način server može konstruisati više soketa koji odgovaraju višestrukim konekcijama sa različitim klijentima. Objekat tipa `ServerSocket` ne učestvuje neposredno u komunikaciji između servera i klijenata, već samo čeka na zahteve za uspostavljanje konekcije od klijenata i konstruiše soket tipa `Socket` radi uspostavljanja veze između servera i klijenta i omogućavanja njihove dalje komunikacije.

Kod pisanje serverskog programa mora se najpre konstruisati serverski soket tipa `ServerSocket` i navesti broj porta na koji server čeka zahteve za uspostavljanje konekcije. Na primer:

```
ServerSocket server = new ServerSocket(port);
```

Port u argumentu konstruktora klase `ServerSocket` mora biti broj tipa `int` iz intervala od 0 do 65536, ali generalno to treba biti broj veći od 1024. Konstruktor klase `ServerSocket` izbacuje izuzetak tipa `IOException` ukoliko se serverski soket ne može konstruisati i povezati sa navedenim portom. To obično znači da se navedeni port već koristi od strane nekog drugog servera.

Serverski soket tipa `ServerSocket` odmah nakon konstruisanja počinje da čeka na zahteve za uspostavljanje konekcije. Ali da bi se mogao prihva-

titi i odgovoriti na jedan takav tahtev, mora se koristiti metod `accept()` iz klase `ServerSocket`. Kada stigne zahtev za uspostavljanje konekcije, metodom `accept()` se taj zahtev prihvata, uspostavlja se konekcija sa klijentom i kao rezultat se vraća soket tipa `Socket` koji se dalje može koristiti za komunikaciju sa klijentom. Na primer, odmah nakon konstruisanja serverskog soketa `server`, odgovaranje na zahteve klijenata od strane servera postiže se naredbom:

```
Socket konekcija = server.accept();
```

Kada se pozove metod `accept()`, njegovo izvršavanje se *blokira* dok se ne primi zahtev za uspostavljanje konekcije, ili dok se ne desi neka greška kada se izbacuje izuzetak tipa `IOException`. Obratite pažnju na to da je onda blokiran i metod u kojem je pozvan metod `accept()`, odnosno da se u niti u kojoj se izvršavaju ti metodi ne može ništa dalje izvršavati. (Naravno, druge niti istog programa se normalno dalje izvršavaju.) Metod `accept()` se može pozivati više puta za isti serverski soket radi prihvatanja zahteva od više klijenata. Serverski soket nastavlja da čeka zahteve za uspostavljanje konekcije sve dok se ne zatvori pozivom metoda `close()`, ili dok se ne desi neka greška u programu.

Nakon uspostavljanja konekcije čiji su krajevi soketi tipa `Socket`, pozivom metoda `getInputStream()` i `getOutputStream()` iz klase `Socket` se za njih mogu formirati ulazni i izlazni tokovi radi prenosa podataka preko uspostavljene konekcije. Metod `getInputStream()` kao rezulta vraća objekat tipa `InputStream`, dok metod `getOutputStream()` kao rezulta vraća objekat tipa `OutputStream`.

Uzimajući u obzir sve do sada rečeno i pretpostavljajući da server treba da očekuje zahteve na portu 2345, osnovna struktura serverskog programa bi otprilike imala sledeći oblik:

```
int port = 2345;
try {
   ServerSocket server = new ServerSocket(port);
   while (true) {
      Socket konekcija = server.accept();
      InputStream in = konekcija.getInputStream();
      OutputStream out = konekcija.getOutputStream();
```

```
            .
            . // Koristiti ulazni i izlazni tok, in i out,
            . // za komunikaciju sa klijentom
            .
            konekcija.close();
        }
    }
    catch (IOException e) {
        System.out.println("Kraj rada servera usled greške: " + e);
    }
```

Na strani klijenta, soket koji je klijentski kraj konekcije konstruiše se pozivom konstruktora klase `Socket`:

```
public Socket(String računar, int port)
       throws IOException
```

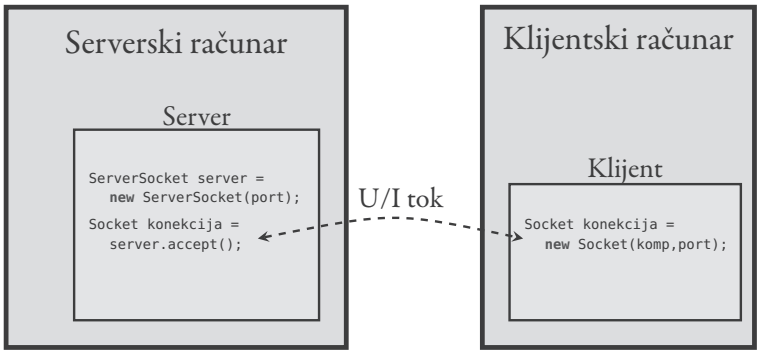
Prvi parametar ovog konstruktora je IP adresa ili domensko ime računara na kome se izvršava server sa kojim se želi uspostaviti konekcija. Drugi parametar je broj porta na kojem taj server očekuje zahteve za uspostavljanje konekcije od klijenata. Konstruktor klase `Socket` ne završava svoj rad sve dok se konekcija ne uspostavi ili dok se ne desi neka greška.

Pretpostavljajući da klijent uspostavlja konekciju sa serverom na adresi `192.168.1.8` i portu 2345, osnovna struktura klijentskog programa bi otprilike imala sledeći oblik:

```
String komp = "192.168.1.8";
int port = 2345;
try {
    Socket konekcija = new Socket(komp,port);
    InputStream in = konekcija.getInputStream();
    OutputStream out = konekcija.getOutputStream();
    .
    . // Koristiti ulazni i izlazni tok, in i out,
    . // za komunikaciju sa serverom
    .
    konekcija.close();
}
catch (IOException e) {
```

```
            System.out.println("Neuspešno uspostavljanje konekcije: " + e);
        }
```

Klijent-server model komunikacije prethodno opisanih prototipova servera i klijenta ilustrovan je na slici 10.2.

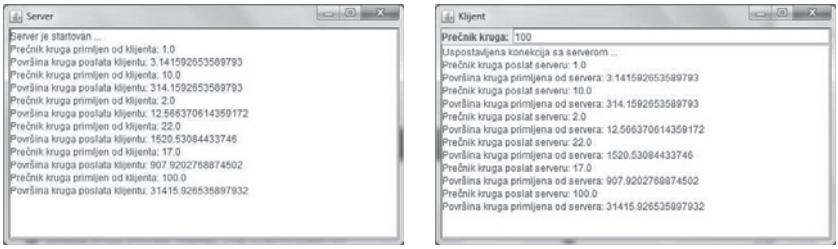


Slika 10.2: Model rada servera i klijenta.

Napomenimo da je u prethodnim prototipovima za serverski i klijentski program prikazano samo najosnovnije rukovanje greškama. U stvari, najteži deo mrežnog programiranja je pisanje robustnih programa, jer je prenos podataka preko mreže prilično nepouzdan. Zbog toga je vrlo verovatna pojava raznih vrsta grešaka u programu, što komplikuje postupak njihove obradi radi uspešnog oporavka programa ukoliko se desi neka greška.

Primer: jednostavni server i klijent

U ovom primeru su prikazani rudimentarni serverski i klijentski programi da bi se ilustrovala dvosmerna komunikacija između njih. Osnovna paradigma ovakve razmene podataka jeste da klijent šalje podatke serveru, a server prima te podatke, koristi ih za neko izračunavanje i dobijeni rezultat šalje nazad klijentu. U ovom primeru, klijent šalje prečnik jednog kruga, a server izračunava površinu tog kruga i šalje je klijentu. Prikaz rada servera i klijenta na ovaj način ilustrovan je na slici 10.3.



Slika 10.3: Dvosmerna komunikacija servera i klijenta.

Server u listingu 10.2 koristi uobičajeni postupak za uspostavljanje konekcije i komunikaciju sa klijentom. Nakon konstruisanja serverskog soketa na portu 2345, server prihvata i uspostavlja konekciju sa klijentom. Nakon što se uspostavi konekcija sa klijentom, preko te konekcije se konstruišu ulazni tok i izlazni tok tipa `DataInputStream` i `DataOutputStream` radi lakšeg primanja i slanja realnih vrednosti tipa `double`. Na kraju se obavlja komunikacija sa klijentom tako što se ponavljaju tri koraka: čitanje prečnika kruga iz ulaznog toka, izračunavanje površine kruga i upisivanje dobijenog rezultata u izlazni tok.

Listing 10.2: Jednostavni server

```
import java.io.*;
import java.net.*;
import java.awt.*;
import javax.swing.*;

public class Server {

   public static void main(String[] args) {

      ServerOkvir okvir = new ServerOkvir();
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
      okvir.startServer();
   }
}
```

```
class ServerOkvir extends JFrame {

   private JTextArea log; // oblast za prikaz rada servera

   // Konstruktor
   public ServerOkvir() {

      setTitle("Server");
      setSize(500, 300);

      log = new JTextArea();
      log.setEditable(false);

      // Dodavanje tekstualnog prikaza u okvir servera
      setLayout(new BorderLayout());
      add(new JScrollPane(log), BorderLayout.CENTER);
   }

   // Glavni metod servera za komunikaciju sa klijentom
   public void startServer() {

      log.append("Server je startovan ... " + '\n');
      try {
         // Konstruisanje serverskog soketa
         ServerSocket srvSoket = new ServerSocket(2345);

         // Čekanje na zahtev za konekcijom od klijenta
         Socket kliSoket = srvSoket.accept();
         srvSoket.close(); // serverski soket nije više potreban

         // Konstruisanje ulaznog i izlaznog toka za razmenu podataka
         DataInputStream odKlijenta = new DataInputStream(
            kliSoket.getInputStream());
         DataOutputStream kaKlijentu = new DataOutputStream(
            kliSoket.getOutputStream());
```

```
            while (true) {
               // Primanje prečnika od klijenta
               double r = odKlijenta.readDouble();
               log.append("Prečnik kruga primljen od klijenta: ");
               log.append(r + "\n");

               // Izračunavanje površine kruga
               double p = r * r * Math.PI;

               // Slanje površine kruga klijentu
               kaKlijentu.writeDouble(p);
               kaKlijentu.flush();
               log.append("Površina kruga poslata klijentu: ");
               log.append(p + "\n");
            }
         }
         catch(IOException e) {
            log.append("Prekid konekcije sa klijentom: " + e + "\n");
         }
      }
   }
```

Serverski program se mora pokrenuti pre nego što se pokrene klijentski program, jer naravno klijent ne može uspostaviti vezu sa serverom koji ne radi. Pošto se server i klijent izvršavaju na istom računaru u ovom primeru, klijent u listingu 10.3 uspostavlja konekciju sa računarom `localhost` na portu 2345. Nakon toga se preko te konekcije konstruišu ulazni i izlazni tokovi radi razmene podataka sa serverom. U klijentu se prečnik kruga koji se šalje serveru unosi u tekstualno polje. Pritisak tastera Enter u tom polju proizvodi akcijski događaj čiji rukovalac zapravo obavlja slanje i primanje podataka. Na taj način se serveru mogu slati više prečnika i od servera redom primati površine odgovarajućih krugova.

Listing 10.3: Jednostavni klijent

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Klijent {

   public static void main(String[] args) {

      KlijentOkvir okvir = new KlijentOkvir();
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
   }
}

class KlijentOkvir extends JFrame {

   private JTextField prečnikPolje; // polje za unos prečnika
   private JTextArea log;    // oblast za prikaz rada klijenta

   // Ulazno/izlazni tokovi
   private DataInputStream  odServera;
   private DataOutputStream kaServeru;

   // Konstruktor
   public KlijentOkvir() {

      setTitle("Klijent");
      setSize(500,300);

      prečnikPolje = new JTextField();
      prečnikPolje.setHorizontalAlignment(JTextField.LEFT);
      // Pridruživanje rukovaoca pritiska na taster Enter
      prečnikPolje.addActionListener(new RukovalacEntera());
```

```
        log = new JTextArea();
        log.setEditable(false);

        // Dodavanje oznake i tekstualnog polja u panel
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add(new JLabel("Prečnik kruga:  "),BorderLayout.WEST);
        panel.add(prečnikPolje,BorderLayout.CENTER);

        // Dodavanje panela i tekstualne oblasti u okvir klijenta
        setLayout(new BorderLayout());
        add(panel,BorderLayout.NORTH);
        add(new JScrollPane(log),BorderLayout.CENTER);


        try {
           // Konstruisanje klijentskog soketa za uspostavljanje
           // konekcije sa serverom "localhost" na portu 2345
           Socket kliSoket = new Socket("localhost",2345);
           // Socket kliSoket = new Socket("192.168.1.8",2345);
           log.append(
              "Uspostavljena konekcija sa serverom ... " + "\n");

           // Konstruisanje ulaznog toka za primanje podataka od servera
           odServera = new DataInputStream(
              kliSoket.getInputStream());

           // Konstruisanje izlaznog toka za slanje podataka ka serveru
           kaServeru = new DataOutputStream(
              kliSoket.getOutputStream());
        }
        catch (IOException e) {
           log.append(
              "Neuspešno uspostavljanje konekcije sa serverom: " + e);
        }
     }
```

```
   // Unutrašnja klasa rukovaoca događaja pritiska na taster Enter
   private class RukovalacEntera implements ActionListener {

      public void actionPerformed(ActionEvent ae) {

         try {
            // Pretvaranje prečnika kruga iz tekstualnog polja u broj
            double r =
               Double.parseDouble(prečnikPolje.getText().trim());

            // Slanje prečnika kruga serveru
            kaServeru.writeDouble(r);
            kaServeru.flush();
            log.append("Prečnik kruga poslat serveru: ");
            log.append(r + "\n");

            // Primanje površine kruga od servera
            double p = odServera.readDouble();
            log.append("Površina kruga primljena od servera: ");
            log.append(p + "\n");
         }
         catch (IOException e) {
            log.append("Greška u komunikaciji sa serverom: " + e);
         }
      }
   }
}
```

Obratite pažnju na to da kada se u serveru površina kruga šalje klijentu pozivom `kaKlijentu.writeDouble(p)`, ili kada se u klijentu prečnik kruga šalje serveru pozivom `kaServeru.writeDouble(r)`, onda se iza toga za odgovarajući izlazni tok poziva metod `flush()`. Metod `flush()` se nalazi u klasi svake vrste izlaznog toka i osigurava da se podaci upisani na jednom kraju izlaznog toka odmah pošalju drugom kraju izlaznog toka. Inače, zavisno od implementacije Jave, podaci ne moraju biti poslati odmah preko mreže, nego se mogu sakupljati sve dok se ne skupi dovoljna količina po-

dataka za slanje. Ovakav pristup se primenjuje radi optimizacije mrežnih operacija, ali se time može izazvati neprihvatljivo kašnjenje u prenosu podataka, pa čak i to da neki podaci ne budu preneseni kada se konekcija zatvori. Zbog toga metod `flush()` treba skoro uvek pozivati kada se koristi izlazni tok za slanje podataka preko mrežne konekcije. U suprotnom slučaju, program može raditi različito na različitim platformama računara.

10.4 Višenitno mrežno programiranje

Priroda mrežnih program često zahteva da se više zadataka obavlja u isto vreme. Jedan primer je slučaj više klijenata koji mogu uspostaviti konekciju sa istim serverom radi dobijanja nekih rezultata od njega. U ovom slučaju server mora istovremeno komunicirati sa više klijenata. Drugi tipični primer je asinhrona komunikacija dva programa preko mreže kada redosled primanja i slanja podataka ne mora biti naizmeničan. U ovom slučaju programi moraju zadatke primanja i slanja podataka obavljati paralelno, jer je vreme stizanja i slanja podataka međusobno nezavisno.

U ovim slučajevima je dakle neophodno korišćenje više niti za paralelno izvršavanje zadataka. U prvom primeru, istovremena komunikacija jednog servera i više klijenata odvija se u posebnim nitima. U drugom primeru, programi na oba kraja konekcije izvršavaju primanje i slanje podataka u posebnim nitima kako ne bi bili blokirani čekajući da podaci stignu kada druga strana nije spremna da ih pošalje. U ovom odeljku se redom govori o ova dva načina primene više programskih niti izvršavanja kod mrežnih programa.

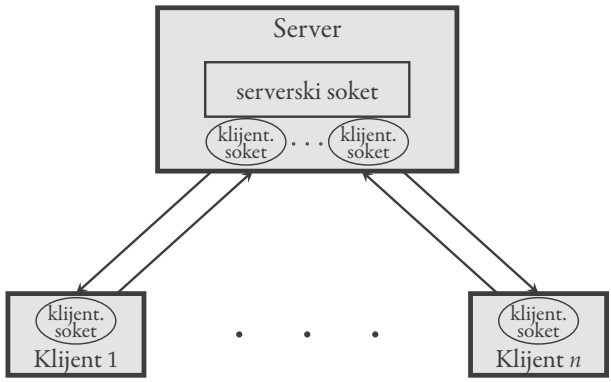
Višenitni server za više klijenata

Obično više klijenata istovremeno uspostavlja konekciju sa jednim serverom radi dobijanja nekih rezultata od njega. Na primer, klijenti u formi veb čitača mogu u isto vreme iz bilo kog kraja Interneta uspostaviti konekciju sa jednim veb serverom radi prikazivanja određenog veb sajta. Istovremena komunikacija servera i više klijenata u Javi može se ostvariti ukoliko

se koristi posebna nit za svaku konekciju. Opšti postupak za ovaj pristup ima sledeći oblik:

```
while (true) {
   Socket konekcija = serverskiSoket.accept();
   Thread nitKlijenta = new Thread(new KlasaZadatka(konekcija));
   nitKlijenta.start();
}
```

U svakoj iteraciji ove beskonačne `while` petlje uspostavlja se nova konekcija na zahtev od klijenta i kao rezultat se dobija jedan klijentski soket na serverskoj strani konekcije. Zatim se ovaj klijentski soket koristi za konstruisanje nove niti u kojoj se odvija komunikacija između servera i novog klijenta. Na taj način u isto vreme može postojati više konekcija koje su ilustrovane na slici 10.4.

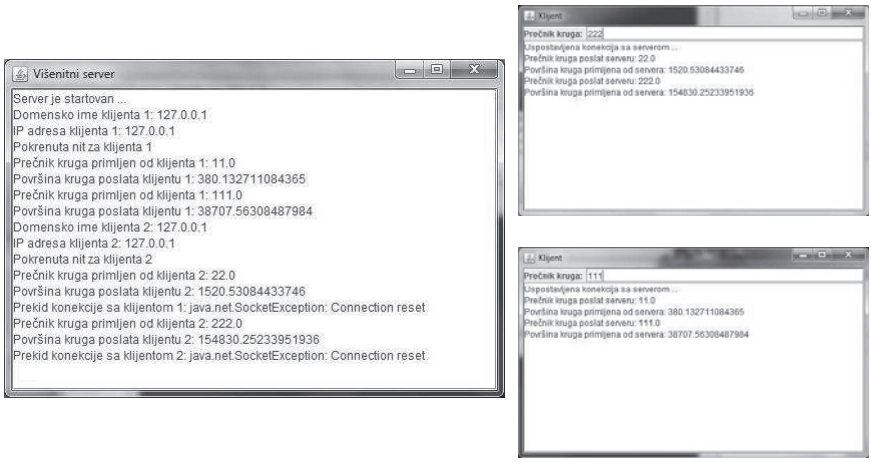


Slika 10.4: Server koji istovremeno uslužuje više klijenata.

Primer: jednostavni višenitni server

Da bismo bolje razumeli osnovni način rada višenitnog servera koji uslužuje više klijenata, u ovom primeru je modifikovan server prikazan u listingu 10.2 na strani 365 za izračunavanje površine kruga. Nakon uspostavljanja konekcije sa svakim novim klijentom, višenitni server u listingu 10.4 u nastavku pokreće posebnu nit za komunikaciju s njim. U toj niti se od

jednog klijenta neprekidno prima prečnik kruga i kao odgovor mu se šalje površina kruga. Pošto je protokol komunikacije između servera i klijenta nepromenjen, klijentski program je isti kao u listingu 10.3 na strani 367. Ilustracija rada ovog višenitnog servera i dva klijenta prikazana je na slici 10.5.



Slika 10.5: Višenitni server i dva klijenta.

Listing 10.4: Jednostavni višenitni server

```
import java.io.*;
import java.net.*;
import java.awt.*;
import javax.swing.*;

public class VišenitniServer {

   public static void main(String[] args) {

      VišenitniServerOkvir okvir = new VišenitniServerOkvir();
      okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      okvir.setVisible(true);
```

```
      okvir.startServer();
   }
}

class VišenitniServerOkvir extends JFrame {

   private JTextArea log; // oblast za prikaz rada servera

   // Konstruktor
   public VišenitniServerOkvir() {

      setTitle("Višenitni server");
      setSize(500, 300);

      log = new JTextArea();
      log.setEditable(false);

      // Dodavanje tekstualnog prikaza u okvir servera
      setLayout(new BorderLayout());
      add(new JScrollPane(log), BorderLayout.CENTER);
   }

   // Glavni metod servera za uspostavljanje konekcije sa klijentima
   public void startServer() {

      log.append("Server je startovan ... " + '\n');

      int k = 0; // broj klijenta
      try {
         // Konstruisanje serverskog soketa
         ServerSocket srvSoket = new ServerSocket(2345);

         while (true) {
            // Čekanje na zahtev za konekcijom od klijenta
            Socket kliSoket = srvSoket.accept();

            k++; // novi klijent je uspostavio konekciju
```

```
            // Nalaženje domenskog imena i IP adrese klijenta
            InetAddress inetAdresa = kliSoket.getInetAddress();
            log.append("Domensko ime klijenta " + k + ": "
               + inetAdresa.getHostName() + "\n");
            log.append("IP adresa klijenta " + k + ": "
               + inetAdresa.getHostAddress() + "\n");

            // Konstruisanje niti za komunikaciju sa klijentom
            Thread nitKlijenta = new Thread(
               new RukovalacKlijenta(kliSoket,k));

            // Startovanje niti za komunikaciju sa klijentom
            nitKlijenta.start();
            log.append("Pokrenuta nit za klijenta " + k + "\n");
         }
      }
      catch(IOException e) {
         log.append("Prekid rada servera: " + e + "\n");
      }
   }

   // Unutrašnja klasa za zadatak koji se obavlja
   // u posebnoj niti (komunikacija sa klijentom)
   private class RukovalacKlijenta implements Runnable {

      private Socket kliSoket;
      private int klijent;

      // Konstruktor
      public RukovalacKlijenta(Socket kliSoket, int klijent) {
         this.kliSoket = kliSoket;
         this.klijent = klijent;
      }

      // Izvršavanje niti
      public void run() {
```

```
            try {
               // Konstruisanje ulaznog i izlaznog toka
               // za razmenu podataka servera i klijenta
               DataInputStream odKlijenta = new DataInputStream(
                  kliSoket.getInputStream());
               DataOutputStream kaKlijentu = new DataOutputStream(
                  kliSoket.getOutputStream());

               while (true) {
                  // Primanje prečnika od klijenta
                  double r = odKlijenta.readDouble();
                  log.append("Prečnik kruga primljen od klijenta ");
                  log.append(klijent + ": " + r + "\n");

                  // Izračunavanje površine kruga
                  double p = r * r * Math.PI;

                  // Slanje površine kruga klijentu
                  kaKlijentu.writeDouble(p);
                  kaKlijentu.flush();
                  log.append("Površina kruga poslata klijentu ");
                  log.append(klijent + ": " + p + "\n");
               }
            }
            catch(IOException e) {
               log.append("Prekid konekcije sa klijentom ");
               log.append(klijent + ": " + e + "\n");
            }
         }
      }
   }
```

Obratite pažnju u ovom programu na prikazivanje IP adrese i domenskog imena klijenta nakon uspostavljanja konekcije za serverom. U tu svrhu se koristi klasa `InetAddress` kojom se u Javi predstavlja internet adresa računara. Internet adresa računara obuhvata njegovu IP adresu (kako IPv4

tako i IPv6) i njegovo domensko ime. Da bi se ove informacije dobile o klijentu, koristi se objektni metod `getInetAddress()` iz klase `Socket`. Naime, kada se pozove za neki klijentski soket, ovaj metod kao rezultat daje objekat tipa `InetAddress` koji predstavlja internet adresu računara na drugom kraju konekcije tog klijentskog soketa. Zato se u programu naredbom

```
InetAddress inetAdresa = kliSoket.getInetAddress();
```

dobija internet adresa klijentskog računara, a njeni pojedini delovi (IP adresa i domensko ime) izdvajaju se metodima `getHostAddress()` i `getHostName()` iz klase `InetAddress`.

Asinhrona komunikacija programa

U prethodnim odeljcima su pokazani primeri u kojima nije zapravo istaknuta jedna od osnovnih karakteristika mrežnih programa — asinhrona priroda njihove međusobne komunikacija. Iz perspektive programa na jednom kraju mrežne konekcije, poruke od programa na drugom kraju mogu stići u nepredvidljivim trenucima nad kojim program koji prima poruke nema uticaja. U nekim slučajevima, kao na primer kod slanja i primanja prečnika i površine kruga, moguće je ustanoviti protokol po kome se komunikacija odvija korak po korak na sinhroni način od početka do kraja. Glavni nedostatak sinhrone komunikacije jeste to što kada po protokolu program treba da primi podatke, on mora da čeka na njih i ne može ništa drugo da radi dok očekivani podaci ne stignu.

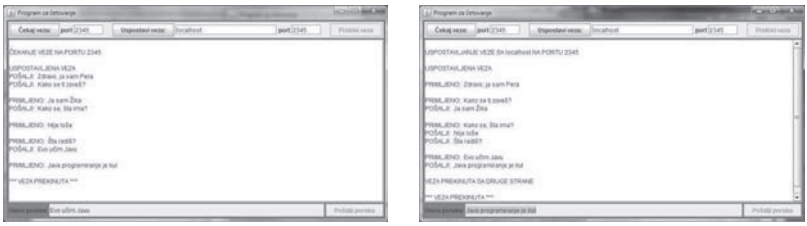
U Javi se asinhrona priroda mrežne komunikacije može realizovati upotrebom više programskih niti. Ako se podsetimo iz poglavlja 9, jedna nit izvršavanja u programu sastoji se od zasebnog niza naredbi koje se nezavisno i paralelno izvršavaju sa drugim programskim nitima. Zbog toga kada mrežni program koristi više paralelnih niti, neke niti mogu biti blokirane zbog čekanja da stignu podaci, ali druge niti mogu nastaviti sa izvršavanjem i obavljati bilo koji drugi korisni zadatak.

Primer: program za „četovanje”

Dva korisnika mogu sinhrono razmenjivati poruke preko mreže („četovati”) tako što jedan korisnik pošalje svoju poruku i zatim čeka da dobije

odgovor od drugog korisnika. Asinhroni način razmene poruka je mnogo bolji, jer onda jedan korisnik može slati poruke ne čekajući da dobije odgovor od druge strane. Po takvom protokolu onda, poruke koje asinhrono stižu od drugog korisnika prikazuju se čim stignu.

Grafički program u listingu 10.5 za asinhronu razmenu poruka zasniva se na posebnoj programskoj niti čiji je zadatak da čita poruke koje stižu sa drugog kraja mrežne konekcije. Čim se pročita jedna poruka u toj niti, ta poruka se prikazuje na ekranu i zatim se opet čeka da stigne sledeća poruka. Dok je nit za čitanje poruka blokirana čekajući da stigne neka poruka, druge niti u programu mogu paralelno nastaviti sa izvršavanjem. Specifično, nit za reagovanje na grafičke događaje izazvane aktivnostima korisnika nije blokirana i u njoj se mogu slati poruke čim ih korisnik napiše. Prikaz rada dva programa na različitim računarima za asinhronu razmenu poruka ilustrovan je na slici 10.6.



Slika 10.6: Asinhrono „četovanje" preko mreže.

Program za „četovanje" u listingu 10.5 može igrati ulogu servera ili klijenta zavisno od toga da li je korisnik pritisnuo dugme za čekanje na zahtev za uspostavljanje konekcije od udaljenog računara ili je pak pritisnuo dugme za uspostavljanje konekcije sa udaljenim računarom. Nakon uspostavljanja mrežne konekcije dva programa, svaki korisnik može drugom slati poruke koje se unose u predviđeno polje na dnu prozora. Pritisak na taster Enter ili na dugme za slanje poruke proizvodi događaj kojim se rukuje u grafičkoj niti tako što se poruka šalje korisniku na drugom kraju konekcije.

Dok se poruke šalju u grafičkoj niti, poruke se primaju u posebnoj niti koja se pokreće čim se pritisne na odgovarajuće dugme u prozoru za čekanje ili uspostavljanje konekcije. U ovoj drugoj niti se obavlja i početno usposta-

vljanje konekcije da bi se izbeglo blokiranje grafičkog korisničkog interfejsa ukoliko uspostavljanje konekcije potraje duže vreme.

U programu u listingu 10.5 koristi se unutrašnja klasa `RukovalacKonekcije` za realizaciju posebne niti u kojoj se upravlja konekcijom preko koje se primaju poruke. Ova klasa sadrži i nekoliko metoda koji se pozivaju u grafičkoj niti i koji se zato izvršavaju u toj drugoj niti. Najznačajniji među njima je metod za slanje poruka koji se poziva i izvršava u grafičkoj niti kao reakcija na aktivnost korisnika. Svi ovi metodi su sinhronizovani kako ne bi došlo do greške trke dve niti usled promene stanja konekcije u sredini odgovarajuće operacije.

Listing 10.5: Program za „četovanje"

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class ChatProgram extends JFrame {

   // Nabrojivi tipovi za tip programa i stanje konekcije
   private enum TipPrograma {SERVER,KLIJENT};
   private enum StanjeKonekcije {POVEZANO,PREKINUTO};

   // Podrazumevani parametri programa
   private TipPrograma program = null;
   private static String brojPorta = "2345";
   private static String udaljeniRačunar = "localhost";

   // Nit uspostavljene konekcije
   private RukovalacKonekcije konekcija;

   // Dugmad i tekstualna polja u prozoru programa
   private JButton serverDugme, klijentDugme;
   private JButton prekiniDugme, pošaljiDugme;
   private JTextField serverPortPolje;
```

```
private JTextField udaljeniRačunarPolje, udaljeniPortPolje;
private JTextField porukaPolje;
private JTextArea log;

// Konstruktor kojim se konstruiše gravni prozor,
// ali se prozor ne čini vidljivim zbog druge niti
public ChatProgram(String naslov) {

   super(naslov);

   ActionListener rukovalac = new RukovalacDugmadima();

   serverDugme = new JButton("Čekaj vezu:");
   serverDugme.addActionListener(rukovalac);

   klijentDugme = new JButton("Uspostavi vezu:");
   klijentDugme.addActionListener(rukovalac);

   prekiniDugme = new JButton("Prekini vezu");
   prekiniDugme.addActionListener(rukovalac);
   prekiniDugme.setEnabled(false);

   pošaljiDugme = new JButton("Pošalji poruku");
   pošaljiDugme.addActionListener(rukovalac);
   pošaljiDugme.setEnabled(false);

   porukaPolje = new JTextField();
   porukaPolje.addActionListener(rukovalac);
   porukaPolje.setEditable(false);

   log = new JTextArea(20,60);
   log.setLineWrap(true);
   log.setWrapStyleWord(true);
   log.setEditable(false);

   serverPortPolje = new JTextField(brojPorta,5);
   udaljeniPortPolje = new JTextField(brojPorta,5);
```

```
udaljeniRačunarPolje = new JTextField(udaljeniRačunar,18);

JPanel sadržaj = new JPanel();
sadržaj.setLayout(new BorderLayout(3,3));
sadržaj.setBackground(Color.GRAY);
sadržaj.setBorder(BorderFactory.createLineBorder(Color.GRAY,3));

JPanel trakaVeza = new JPanel();
trakaVeza.setLayout(new FlowLayout(FlowLayout.CENTER,3,3));
trakaVeza.add(serverDugme);
trakaVeza.add(new JLabel("port"));
trakaVeza.add(serverPortPolje);
trakaVeza.add(Box.createHorizontalStrut(12));
trakaVeza.add(klijentDugme);
trakaVeza.add(udaljeniRačunarPolje);
trakaVeza.add(new JLabel("port"));
trakaVeza.add(udaljeniPortPolje);
trakaVeza.add(Box.createHorizontalStrut(12));
trakaVeza.add(prekiniDugme);

JPanel trakaPoruka = new JPanel();
trakaPoruka.setLayout(new BorderLayout(3,3));
trakaPoruka.setBackground(Color.GRAY);
trakaPoruka.add(new JLabel("Unesi poruku:"), BorderLayout.WEST);
trakaPoruka.add(porukaPolje, BorderLayout.CENTER);
trakaPoruka.add(pošaljiDugme, BorderLayout.EAST);

sadržaj.add(trakaVeza, BorderLayout.NORTH);
sadržaj.add(new JScrollPane(log), BorderLayout.CENTER);
sadržaj.add(trakaPoruka, BorderLayout.SOUTH);

setContentPane(sadržaj);
pack();
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

addWindowListener(new WindowAdapter() {
   public void windowClosed(WindowEvent we) {
```

```
            if (konekcija != null &&
                  konekcija.stanje() == StanjeKonekcije.POVEZANO) {
               konekcija.zatvori();
            }
            System.exit(0);
         }
      });
   }

   // Glavni metod kojim se konstruiše i prikazuje glavni prozor
   public static void main(String[] args) {
      ChatProgram okvir = new ChatProgram("Program za četovanje");
      okvir.setVisible(true);
   }

   // Dodavanje poruke u oblast za prikaz rada programa
   private void prikaži(String poruka) {
      log.append(poruka);
      log.setCaretPosition(log.getDocument().getLength());
   }

   // Unutrašnja klasa rukovaoca događaja svih dugmadi u prozoru
   private class RukovalacDugmadima implements ActionListener {
      public void actionPerformed(ActionEvent ae) {

         Object izvorDogađaja = ae.getSource();

         if (izvorDogađaja == serverDugme) {
            if (konekcija == null ||
                  konekcija.stanje() == StanjeKonekcije.PREKINUTO) {
               String portString = serverPortPolje.getText();
               int port;
               try {
                  port = Integer.parseInt(portString);
                  if (port < 1025 || port > 65535)
                     throw new NumberFormatException();
               }
```

```
            catch (NumberFormatException e) {
               JOptionPane.showMessageDialog(ChatProgram.this,
                     portString + " nije ispravan broj porta.");
               return;
            }
            klijentDugme.setEnabled(false);
            serverDugme.setEnabled(false);
            program = TipPrograma.SERVER;
            konekcija = new RukovalacKonekcije(port);
         }
      }
      else if (izvorDogađaja == klijentDugme) {
         if (konekcija == null ||
               konekcija.stanje() == StanjeKonekcije.PREKINUTO) {
            String portString =
               udaljeniPortPolje.getText().trim();
            int port;
            try {
               port = Integer.parseInt(portString);
               if (port < 1025 || port > 65535)
                  throw new NumberFormatException();
            }
            catch (NumberFormatException e) {
               JOptionPane.showMessageDialog(ChatProgram.this,
                     portString + " nije ispravan broj porta.");
               return;
            }
            klijentDugme.setEnabled(false);
            serverDugme.setEnabled(false);
            program = TipPrograma.KLIJENT;
            konekcija = new RukovalacKonekcije(
               udaljeniRačunarPolje.getText().trim(),port);
         }
      }
      else if (izvorDogađaja == prekiniDugme) {
         if (konekcija != null)
            konekcija.zatvori();
```

```
        }
        else if (izvorDogađaja == pošaljiDugme ||
                    izvorDogađaja == porukaPolje) {
           if (konekcija != null &&
                  konekcija.stanje() == StanjeKonekcije.POVEZANO) {
              konekcija.pošalji(porukaPolje.getText());
              porukaPolje.selectAll();
              porukaPolje.requestFocus();
           }
        }
     }
  }

  // Unutrašnja klasa za nit u kojoj se upravlja
  // konekcijom preko koje se razmenjuju poruke
  private class RukovalacKonekcije extends Thread {

     private volatile StanjeKonekcije s;
     private String udaljeniRačunar;
     private int port;
     private ServerSocket srvSoket;
     private Socket kliSoket;
     private PrintWriter izlaz;    // izlazni tok za konekciju
     private BufferedReader ulaz;  // ulazni tok za konekciju

     // Konstruktor kada program radi kao server
     RukovalacKonekcije(int port) {
        this.port = port;
        prikaži("\nČEKANJE VEZE NA PORTU " + port + "\n");
        start();
     }

     // Konstruktor kada program radi kao klijent
     RukovalacKonekcije(String udaljeniRačunar, int port) {
        this.udaljeniRačunar = udaljeniRačunar;
        this.port = port;
        prikaži("\nUSPOSTAVLJANJE VEZE SA " + udaljeniRačunar);
```

```
   prikaži(" NA PORTU " + port + "\n");
   start();
}

// Trenutno stanje konekcije (metod se poziva iz druge niti)
synchronized StanjeKonekcije stanje() {
   return s;
}

// Slanje poruke preko uspostavljene konekcije
// (metod se poziva iz druge niti)
synchronized void pošalji(String poruka) {
   if (s == StanjeKonekcije.POVEZANO) {
      prikaži("POŠALJI:  " + poruka + "\n");
      izlaz.println(poruka);
      izlaz.flush();
      if (izlaz.checkError()) {
         prikaži("\n\nGREŠKA PRILIKOM SLANJA PODATAKA!");
         zatvori();
      }
   }
}

// Zatvaranje konekcije iz nekog razloga
// (metod se poziva iz druge niti)
synchronized void zatvori() {
   s = StanjeKonekcije.PREKINUTO;
   serverDugme.setEnabled(true);
   klijentDugme.setEnabled(true);
   prekiniDugme.setEnabled(false);
   pošaljiDugme.setEnabled(false);
   porukaPolje.setEditable(false);

   try {
      if (kliSoket != null)
         kliSoket.close();
      else if (srvSoket != null)
```

```
            srvSoket.close();
        }
        catch (IOException e) {
        }
    }

    // Glavni metod kojim se definiše nit izvršavanja
    public void run() {

        try {
            if (program == TipPrograma.SERVER) {
                srvSoket = new ServerSocket(port);
                kliSoket = srvSoket.accept();
                srvSoket.close();
            }
            else if (program == TipPrograma.KLIJENT) {
                kliSoket = new Socket(udaljeniRačunar,port);
            }

            // Priprema konekcije za razmenu poruka
            srvSoket = null;
            ulaz = new BufferedReader(
                new InputStreamReader(kliSoket.getInputStream()));
            izlaz = new PrintWriter(kliSoket.getOutputStream());
            s = StanjeKonekcije.POVEZANO;
            prikaži("\nUSPOSTAVLJENA VEZA\n");
            porukaPolje.setEditable(true);
            porukaPolje.setText("");
            porukaPolje.requestFocus();
            pošaljiDugme.setEnabled(true);
            prekiniDugme.setEnabled(true);

            // Čitanje jednog reda poruke od druge
            // strane sve dok je veza uspostavljena
            while (s == StanjeKonekcije.POVEZANO) {
                String red = ulaz.readLine();
                if (red == null) {
```

```
                prikaži("\nVEZA PREKINUTA SA DRUGE STRANE\n");
                s = StanjeKonekcije.PREKINUTO;
             }
             else
                prikaži("\nPRIMLJENO:  " + red + "\n");
          }
       }
       catch (ConnectException e) {
             prikaži("\n\n GREŠKA:  " + e + "\n");
       }
       catch (Exception e) {
          if (s == StanjeKonekcije.POVEZANO)
             prikaži("\n\n GREŠKA:  " + e + "\n");
       }
       // Na kraju izvršavanja "počistiti sve za sobom"
       finally {
          zatvori();
          prikaži("\n*** VEZA PREKINUTA ***\n");
          kliSoket = null;
          ulaz = null;
          izlaz = null;
          srvSoket = null;
       }
    }
  }
}
```

GLAVA 11

Generičko programiranje

Termin *generičko programiranje* odnosi se na pisanje programskog koda kojim se obezbeđuje uniformna obrada struktura podataka različitih tipova. Pretpostavimo da se u programu koristi dinamički niz koji se sastoji od elemenata određenog tipa, recimo `int`. Ukoliko je potrebno da elementi drugog dinamičkog niza budu drugog tipa, recimo `String`, deo programa u kome se obrađuje jedan takav niz bio bi identičan onom za elemente niza tip `int`, osim zamene jednog tipa drugim. Pisanje u suštini istog programskog koda više puta nije naravno odlika dobrog programiranja, jer pored gubitka vremena uvek postoji mogućnost grešaka kod prepisivanja.

Elegantno rešenje ovog problema u Javi je upotreba „parametrizovanih tipova" radi generalizovanja programskog koda. Naime, klase, interfejsi i metodi u Javi mogu biti definisani sa parametrima koji predstavljaju (klasne) tipove podataka. U ovom poglavlju se najpre govori o korišćenju takvih *generičkih* programskih celina koje su na raspolaganju programerima u standardnim bibliotekama, a na kraju će biti više reči o tome kako se definišu nove generičke klase, interfejsi i metodi.

11.1 Uvod

Poznato je da, na primer, dinamički niz tipa `ArrayList` može imati elemente bilo kog tipa. To je posledica činjenice što je u klasi `ArrayList` definisano da se svaki niz tipa `ArrayList` sastoji zapravo od elemenata tipa

`Object`. Kako se podrazumeva da je svaka klasa u Javi izvedena od najopštije klase `Object`, to na osnovu principa podtipa onda sledi da objekti bilo kog tipa mogu biti vrednosti elemenata niza tipa `ArrayList`.

Ali u Javi se može i suziti tip elemenata dinamičkog niza, recimo da bude `String`, ukoliko se konstruiše dinamički niz tipa `ArrayList<String>`. Ovo je moguće zato što je definicija klase `ArrayList` zapravo parametrizovana jednim tipom podataka i ima oblik `ArrayList<T>`. Dodatak `<T>` u definiciji klase predstavlja formalni generički tip koji se zamenjuje stvarnim konkretnim tipom prilikom korišćenja klase. (Po konvenciji, formalni generički tip označava se jednim velikim slovom.)

Dobra strana primene generičkog programiranja sastoji se u tome što se dobijaju opštije programske celine i time se poboljšava čitljivost programa u kome se koriste. Druga, možda važnija prednost generičkog programiranja jeste to što se time omogućava rano otkrivanje grešaka i tako program čini pouzdanijim. Na primer, ako se u programu koriste dva niza `a` i `b` čiji elementi treba da budu celi brojevi i ako je `a` niz tipa `ArrayList` i `b` niz tipa `ArrayList<Integer>`, onda se dodela vrednosti pogrešnog tipa elementima ovih nizova može otkriti u različitim fazama razvoja programa. Tako, naredba

```
a[0] = "greška";
```

ne može biti otkrivena kao pogrešna u prvoj fazi prevođenja programa, jer elementi niza `a` tipa `ArrayList` mogu biti bilo kog tipa i prevodilac ne može znati da vrednost elementa `a[0]` treba da bude celobrojnog tipa, a ne pogrešnog tipa `String`. Ova vrsta greške dakle može biti otkrivena tek u kasnijoj fazi — fazi izvršavanja programa, ali tada takva greška može izazvati i vrlo ozbiljne posledice. S druge strane, naredba

```
b[0] = "greška";
```

biće otkrivena kao pogrešna u ranoj fazi prevođenja programa, jer prevodilac može iskoristi činjenicu da elementi niza `b` tipa `ArrayList<Integer>` moraju biti tipa `Integer`.

Klasa `ArrayList` je samo jedna od više standardnih klasa koje se koriste za generičko programiranje u Javi. Ove gotove generičke klase i interfejsi nazivaju se jednim imenom „Sistem kolekcija u Javi” (engl. *Java Collection Framework*). Ovaj sistem je prošao kroz nekoliko faza razvoja od početne

verzije, a u Javi 5 su dodati parametrizovani tipovi kao što je `ArrayList<T>`. Tačnije, u Javi 5 su parametrizovane sve klase i interfejsi koje čine Sistem kolekcija, pa čak i neke klase koje nisu deo tog sistema. Zbog toga je u Javi moguće konstruisati generičke strukture podataka čiji se tipovi mogu proveriti tokom prevođenja programa, a ne tek u fazi izvršavanja programa. Treba ipak napomenuti da se parametrizovani tipovi mogu koristiti i bez parametara tako da je dozvoljeno koristiti, recimo, i običnu klasu `ArrayList`.

Ponekad treba imati na umu da je implementacija parametrizovanih tipova u Javi specifična po tome što se za svaku parametrizovanu klasu dobija samo jedna prevedena klasa. Na primer, postoji samo jedna prevedena klasa `ArrayList.class` za parametrizovanu klasu `ArrayList<T>`. Tako, parametrizovani tipovi `ArrayList<String>` i `ArrayList<Integer>`, kao i običan tip `ArrayList`, koriste jednu istu prevedenu klasu. Naime, konkretni tipovi `String` i `Integer`, koji su argumenti parametrizovanog tipa, služe samo kao informacija prevodiocu da ograniči tip objekata koji mogu biti članovi strukture podataka. Ti konkretni tipovi nemaju nikakvog efekta u fazi izvršavanja programa i čak nisu ni poznati u toku izvršavanja programa.

Ovo gubljenje informacije o tipu u fazi izvršavanja programa kod parametrizovanih tipova postavlja izvesna ograničenja u radu sa parametrizovanim tipovima. Na primer, logički uslov u sledećoj naredbi `if` nije ispravan:

```
if (a instanceof ArrayList<String>)  // GREŠKA!
```

Razlog za ovo je to što se operator `instanceof` primenjuje u fazi izvršavanja programa, a tada postoji samo običan tip `ArrayList`. Slično, ne može se konstruisati *običan* niz čiji je bazni tip `ArrayList<String>`, na primer:

```
new ArrayList<String>[100]  // GREŠKA!
```

jer se i operator `new` primenjuje u fazi izvršavanja programa, a tada ne postoji parametrizovani tip `ArrayList<String>`.

11.2 Sistem kolekcija u Javi

Generičke strukture podataka u Javi mogu se podeliti u dve kategorije: *kolekcije* i *mape*. Kolekcija je struktura podataka koja se posmatra samo kao

zajednička grupa nekih objekata, bez obraćanja mnogo pažnje na moguće dodatne međusobne odnose između objekata u grupi. Mapa je struktura podataka u kojoj se može prepoznati preslikavanje objekata jednog skupa u objekte drugog skupa. Telefonski imenik je primer jedne takve strukture podataka, jer u njemu su imena pretplatnika pridružena telefonskim brojevima čime je definisano preslikavanje skupa imena pretplatnika u skup telefonskih brojeva.

Kolekcije i mape su u Javi predstavljene parametrizovanim interfejsima `Collection<T>` i `Map<T,S>`. Slova T i S ovde stoje umesto bilo kog tipa osim nekog od primitivnih tipova. Primetimo još da parametrizovani tip `Map<T,S>` ima dva parametra tipa, T i S. (U opštem slučaju, parametrizovani tip može imati više parametara tipa odvojenih zapetama.)

Kolekcije

Kolekcije objekata u Javi se prema specifičnim međusobnim odnosima njihovih objekata dalje dele u dve podvrste: *liste* i *skupove*. Lista je kolekcija u kojoj su objekti linearno uređeni, odnosno definisan je redosled objekata kolekcije po kome je određen prvi objekat, drugi objekat i tako dalje. Preciznije, osnovno svojstvo liste je da se iza svakog objekta u listi, osim poslednjeg, nalazi tačno jedan drugi objekat liste. Osnovno svojstvo skupa kao kolekcije objekata je da ne postoje duplikati objekata u skupu. Naglasimo da se kod skupova u opštem slučaju ne podrazumeva da među članovima skupa postoji bilo kakav međusobni poredak.

Liste i skupovi su predstavljeni parametrizovanim interfejsima `List<T>` i `Set<T>` koji nasleđuju interfejs `Collection<T>`. Prema tome, svaki objekat koji implementira interfejs `List<T>` ili `Set<T>` automatski implementira i interfejs `Collection<T>`. Interfejs `Collection<T>` definiše opšte operacije koje se mogu primeniti na svaku kolekciju, dok interfejsi `List<T>` i `Set<T>` definišu dodatne operacije koje su specifične samo za liste i skupove.

Svaka aktuelna struktura podataka koja je kolekcija, lista ili skup mora naravno biti objekat konkretne klase koja implementira odgovarajući interfejs. Klasa `ArrayList<T>`, na primer, implementira interfejs `List<T>` i time takođe implementira interfejs `Collection<T>`. To znači da se svi metodi koji se nalaze u interfejsu `List<T>`, odnosno `Collection<T>`, mogu koristiti sa

objektima tipa, recimo, `ArrayList<String>`.

Interfejs `Collection<T>`. Interfejs `Collection<T>` sadrži metode za obavljanje osnovnih operacija nad bilo kojom kolekcijom objekata. Kako je kolekcija vrlo opšti pojam strukture podataka, to su i operacije koje se mogu primeniti nad svim kolekcijama vrlo generalne. One su generalne u smislu da se mogu primeniti nad raznim vrstama kolekcija koje sadrže razne tipove objekata. Ako pretpostavimo da je `k` objekat klase koja implementira interfejs `Collection<T>` za neki konkretan neprimitivni tip `T`, onda se za kolekciju `k` mogu primeniti sledeći metodi definisani u interfejsu `Collection<T>`:

- `k.size()` vraća celobrojnu vrednost tipa `int` koja predstavlja veličinu kolekcije `k`, odnosno aktuelni broj objekata koji se nalazi u kolekciji `k`.
- `k.isEmpty()` vraća logičku vrednost `true` ukoliko je kolekcija `k` prazna, odnosno ukoliko je njena veličina jednaka 0.
- `k.clear()` uklanja sve objekte iz kolekcije `k`.
- `k.add(o)` dodaje jedan objekat `o` kolekciji `k`. Tip objekta `o` mora biti `T`; ako to nije slučaj, proizvodi se sintaksna greška prilikom prevođenja. Metod `add()` vraća logičku vrednost koja ukazuje na to da li se dodavanjem datog objekta kao rezultat dobila modifikovana kolekcija. Na primer, dodavanje duplikata nekog objekta u skup ne proizvodi nikakav efekat i zato se originalna kolekcija ne menja.
- `k.contains(o)` vraća logičku vrednost `true` ukoliko kolekcija `k` sadrži objekat `o`. Objekat `o` može biti bilo kog tipa, jer ima smisla proveravati da li se objekat bilo kog tipa nalazi u kolekciji.
- `k.remove(o)` uklanja objekat `o` iz kolekcije `k`, ukoliko se dati objekat nalazi u kolekciji. Metod `remove()` vraća logičku vrednost koja ukazuje na to da li se dati objekat nalazio u kolekciji, odnosno da li se kao rezultat uklanjanja dobila modifikovana kolekcija. Tip objekta `o` *ne* mora biti `T`.
- `k.containsAll(kk)` vraća logičku vrednost `true` ukoliko kolekcija `k` sadrži sve objekte kolekcije `kk`. Kolekcija `kk` može biti bilo kog tipa oblika `Collection<T>`.

- `k.addAll(kk)` dodaje sve objekte kolekcije `kk` u kolekciju `k`. Kolekcija `kk` može biti bilo kog tipa oblika `Collection<T>`.
- `k.removeAll(kk)` uklanja sve objekte kolekcije `kk` iz kolekcije `k`. Kolekcija `kk` može biti bilo kog tipa oblika `Collection<T>`.
- `k.retainAll(kk)` zadržava samo objekte kolekcije `kk` u kolekciji `k`, a sve ostale uklanja iz kolekcije `k`. Kolekcija `kk` može biti bilo kog tipa oblika `Collection<T>`.
- `k.toArray()` vraća niz tipa `Object[]` koji sadrži sve objekte u kolekciji `k`. Obratite pažnju na to da je vraćeni niz tipa `Object[]`, a ne `T[]`, mada se njegov tip može eksplicitnom konverzijom promeniti u drugi tip. Na primer, ako su svi objekti u kolekciji `k` tipa `String`, onda se eksplicitnom konverzijom `(String[]) k.toArray()` dobija niz stringova koji sadrži sve stringove u kolekciji `k`.

Pošto se ovi metodi nalaze u interfejsu `Collection<T>`, oni moraju biti definisani u svakoj klasi koja implementira taj interfejs. To znači da funkcija ovih metoda zavisi od same implementacije u konkretnoj klasi i da prethodni opis njihove semantike ne mora biti tačan za svaku kolekciju. Ali, za sve standardne kolekcije u Javi, semantika metoda u interfejsu `Collection<T>` upravo je ona koja je gore navedena.

Potencijalni problem koji proističe iz opštosti interfejsa `Collection<T>` jeste i to što neki njegovi metodi ne moraju imati smisla za sve kolekcije. Veličina nekih kolekcija, na primer, ne može se promeniti nakon konstruisanja kolekcije, pa metodi za dodavanje i uklanjanje objekata u tim kolekcijama nemaju smisla. U takvim slučajevima, iako je u programu dozvoljeno da se pozivaju ovi metodi, u fazi izvršavanja programa oni izazivaju izuzetak tipa `UnsupportedOperationException`.

Jednakost i poređenje objekata kolekcije. U interfejsu `Collection<T>` nekoliko metoda zavisi od toga kako se određuje jednakost objekata u kolekciji. Tako se u metodima `k.contains(o)` i `k.remove(o)`, recimo, dobija odgovarajući rezultat prema tome da li se u kolekciji `k` pronalazi neki objekat koji je jednak datom objektu `o`. Ali, pod razumnom jednakošću objekata obično se ne podrazumeva ono što se dobija primenom operatora ==.

Operatorom == proverava se da li su dva objekta jednaka u smislu da zauzimaju isti memorijski prostor. To je u opštem slučaju sasvim različito od prirodne interpretacije jednakosti dva objekta u smislu da predstavljaju istu vrednost. Na primer, prirodno je smatrati da su dva stringa tipa `String` jednaka ukoliko sadrže isti niz znakova, dok je pitanje da li se oni nalaze u istoj memorijskoj lokaciji potpuno nevažno.

U klasi `Object`, koja se nalazi na samom vrhu hijerarhije klasa u Javi, definisan je metod `equals(o)` koji vraća tačno ili netačno prema tome da li je jedan objekat jednak drugom. Ovaj metod se koristi u mnogim standardnim klasama Sistema kolekcija u Javi radi utvrđivanja da li su dva objekta jednaka. U klasi `Object`, rezultat izraza `o1.equals(o2)` definisan je da bude ekvivalentan rezultatu izraza `o1 == o2`. Kao što smo napomenuli, ova definicija nije odgovarajuća za mnoge druge „prave" klase koje automatski nasleđuju klasu `Object`, pa metod `equals()` treba redefinisati (nadjačati) u njima. Zbog toga je ovaj metod u klasi `String` definisan tako da je `s.equals(o)` tačno ukoliko je objekat `o` tipa `String` i `o` sadrži isti niz znakova kao i string `s`.

U svojim klasama programeri bi takođe trebalo da definišu poseban metod `equals()` da bi se dobio očekivan rezultat prilikom određivanja jednakosti objekata tih klasa. U narednom primeru je u klasi `PunoIme` definisan takav metod kojim se korektno određuje da li su puna imena dve osobe jednaka:

```
public class PunoIme {

   private String ime, prezime;

   public boolean equals(Object o) {
      try {
         PunoIme drugo = (PunoIme)o;  // konverzija argumenta
         return ime.equals(drugo.ime) &&
                   prezime.equals(drugo.prezime);
      }
      catch (Exception e) {
         return false;  // ako je o == null ili
                        // o nije tipa PunoIme
```

```
        }
    }
    .
    . // Ostali metodi
    .
}
```

Sa ovako definisanim metodom `equals()`, klasa `PunoIme` se može potpuno bezbedno koristiti u kolekcijama. U suprotnom slučaju, metodi u interfejsu `Collection<PunoIme>` kao što su `contains()` i `remove()` ne bi dali tačan rezultat.

Pored jednakosti objekata, druga relacija među proizvoljnim objektima za koju ne postoji očigledna interpretacija jeste njihov poredak. Ovaj problem pojavljuje se naročito kod *sortiranja* objekata u kolekciji po rastućem ili opadajućem redosledu. Da bi se objekti mogli međusobno upoređivati, njihova klasa mora da implementira interfejs `Comparable`. U stvari, ovaj interfejs je definisan kao parametrizovan interfejs `Comparable<T>` kojim se obezbeđuje poređenje objekata tipa `T`. U interfejsu `Comparable<T>` definisan je samo jedan metod:

```
public int compareTo(T o)
```

Rezultat poziva `o1.compareTo(o2)` treba da bude negativan broj ukoliko se objekat `o1` nalazi pre objekta `o2` u relativnom poretku. Taj rezultat treba da bude pozitivan broj ukoliko se objekat `o1` nalazi iza objekta `o2` u relativnom poretku. Najzad, rezultat treba da bude nula ukoliko se objekti u svrhu poređenja smatraju jednakim. Obratite pažnju na to da ova jednakost ne mora biti ekvivalentna rezultatu izraza `o1.equals(o2)`. Na primer, ako je reč o sortiranju objekata tipa `PunoIme`, onda je uobičajeno da se to radi po prezimenima osoba. U tom slučaju, dva puna imena se u svrhu poređenja radi sortiranja smatraju jednakim ukoliko imaju isto prezime, ali to očigledno ne znači da su puna imena dve osobe jednaka.

Klasa `String` implementira interfejs `Comparable<String>` i definiše metod `compareTo()` za poređenje stringova na razuman način. Na osnovu toga može se definisati i poređenje punih imena tipa `PunoIme` radi sortiranja:

```
public class PunoIme implements Comparable<PunoIme> {
```

```
    private String ime, prezime;

    public int compareTo(PunoIme drugo) {

       if (prezime.compareTo(drugo.prezime) < 0)
          return -1;
       else if (prezime.compareTo(drugo.prezime) > 0)
          return +1;
       else
          return ime.compareTo(drugo.ime);
    }
    .
    . // Ostali metodi
    .
}
```

Drugi način poređenja objekata u Javi sastoji se od primene posebnog objekta koji direktno izvodi poređenje. Ovaj objekat se naziva komparator i mora pripadati klasi koja implementira interfejs `Comparator<T>`, gde je `T` tip objekata koji se upoređuju. U interfejsu `Comparator<T>` definisan je jedan metod:

```
public int compare(T o1, T o2)
```

Ovaj metod upoređuje dva objekta tipa `T` i treba da vrati broj koji je negativan, pozitivan ili nula prema tome da li je objekat `o1` „manji", „veći" ili „jednak" objektu `o2`. Komparatori su korisni u slučajevima kada treba porediti objekte koji ne implementiraju interfejs `Comparable` ili kada je potrebno definisati različite relacije poretka nad istom kolekcijom objekata.

Primitivni tipovi i generičke kolekcije. Generičkim kolekcijama u Javi ne mogu pripadati vrednosti primitivnih tipova nego samo klasnih tipova. Ali ovo ograničenje može se skoro potpuno prevazići upotrebom klasa „omotača" primitivnih tipova.

Podsetimo se da je za svaki primitivni tip definisan odgovarajući klasni tip-omotač: za `int` je to klasa `Integer`, za `boolean` je to klasa `Boolean`, za `char` je to klasa `Character` i tako dalje. Tako, jedan objekat tipa `Integer` sadrži vrednost tipa `int` i služi kao omotač za nju. Na taj način se primi-

tivne vrednosti mogu koristiti tamo gde su neophodni objekti u Javi, što je slučaj kod generičkih kolekcija. Na primer, lista celih brojeva može biti predstavljena kolekcijom tipa `ArrayList<Integer>`. U klasi `Integer` su na prirodan način definisani metodi `equals()`, `compareTo()` i `toString()`, a slične osobine imaju i druge klase omotači.

Podsetimo se i da se u programu izvodi automatska konverzija između vrednosti primitivnog tipa i objekata odgovarajuće klase omotača. Zbog toga u radu sa objektima koji su omotači primitivnih vrednosti skoro da i nema razlike u odnosu na rad sa primitivnim vrednostima. To važi i za generičko programiranje, jer nakon konstruisanja kolekcije sa objektima koji pripadaju omotačkoj klasi nekog primitivnog tipa, ta kolekcija se može koristiti skoro na isti način kao da sadrži vrednosti primitivnog tipa. Na primer, ako je `k` kolekcija tipa `Collection<Integer>`, onda se mogu koristiti metodi `k.add(23)` ili `k.remove(17)`. Iako se vrednosti primitivnog tipa `int` ne mogu dodavati kolekciji `k` (ili uklanjati iz nje), broj 23 se automatski konvertuje u omotački objekat `new Integer(23)` i dodaje kolekciji `k`. Ovde ipak treba imati u vidu da konverzija utiče na efikasnost programa tako da je upotreba, recimo, običnog niz tipa `int[]` mnogo efikasnija od upotrebe kolekcije tipa `ArrayList<Integer>`.

Iteratori. U interfejsu `Collection<T>` su definisani neki opšti metodi koji se mogu primeniti za svaku kolekciju objekata. Postoje međutim i generičke operacije koje nisu deo tog interfejsa, jer priroda tih operacija zavisi od konkretne implementacije neke kolekcije. Takva je recimo operacija prikazivanja *svih* objekata u kolekciji. Ova operacija zahteva sistematično „posećivanje" svih objekata u kolekciji nekim redom, počinjući od nekog objekta u kolekciji i prelazeći s jednog objekta na drugi dok se ne iscrpu svi objekti. Ali pojam prelaska s jednog objekta na drugi u kolekciji može imati različitu interpretaciju u konkretnoj implemantaciji međusobnih veza objekata kolekcije. Na primer, ako je kolekcija implementirana kao običan niz objekata, onda je to prosto uvećanje indeksa niza za jedan; ako je kolekcija implementirana kao povezana struktura objekata, onda je to praćenje reference (pokazivača) na sledeći objekat.

Ovaj problem se u Javi može rešiti na opšti način pomoću *iteratora*. Iterator je objekat koji služi upravo za prelazak s jednog objekta na drugi u

kolekciji. Različite vrste kolekcija mogu imati iteratore koji su implementirani na različite načine, ali svi iteratori se koriste na isti način i zato se njihovom upotrebom dobijaju generički postupci.

U interfejsu `Collection<T>` nalazi se metod koji služi za dobijanje iteratora bilo koje kolekcije. Ako je `k` kolekcija, onda poziv `k.iterator()` kao rezultat daje iterator koji se može koristiti za pristup svim objektima kolekcije `k`. Iterator se može zamisliti kao jedan oblik generalizovanog pokazivača koji najpre ukazuje na početni objekat kolekcije, a zatim se može pomerati s jednog objekta na sledeći objekat u kolekciji.

Iteratori su definisani parametrizovanim interfejsom `Iterator<T>`. Ako je `k` objekat klase koja implementira interfejs `Collection<T>` za neki specifični tip `T`, onda `k.iterator()` kao rezultat daje iterator tipa `Iterator<T>`, gde je `T` isti konkretni tip za koji je implementiran interfejs `Collection<T>`. U interfejsu `Iterator<T>` definisana su samo tri metoda. Ako `iter` ukazuje na objekat klase koja implementira interfejs `Iterator<T>`, onda se za taj iterator mogu pozivati ovi metodi:

- `iter.next()` vraća sledeći objekat u kolekciji i pomera iterator za jedno mesto. Vraćena vrednost ovog metoda je tipa `T`. Primetimo da nije moguće pristupiti nekom objektu kolekcije bez pomeranja iteratora na sledeće mesto. Ukoliko su iskorišćeni svi objekti kolekcije, poziv ovog metoda proizvodi izuzetak tipa `NoSuchElementException`.

- `iter.hasNext()` vraća logičku vrednost tačno ili netačno prema tome da li ima preostalih objekata u kolekciji koji nisu korišćeni. U programu se ovaj metod obično poziva pre metoda `iter.next()` da ne bi došlo do izbacivanja izuzetka tipa `NoSuchElementException`.

- `iter.remove()` uklanja objekat koji je vraćen poslednjim pozivom metoda `iter.next()`. Poziv metoda `iter.remove()` može dovesti do izbacivanja izuzetka tipa `UnsupportedOperationException` ukoliko se iz kolekcije ne mogu uklanjati objekti.

Prethodno pomenuti problem prikazivanja objekata neke kolekcije može se lako generički rešiti upotrebom iteratora. Na primer, ako je `k` kolekcija stringova tipa `Collection<String>`, onda jr rezultat poziva `k.iterator()` jedan iterator tipa `Iterator<String>` za kolekciju `k`, pa se može pisati:

```
// Dobijanje iteratera za kolekciju
Iterator<String> iter = k.iterator();

// Pristupanje svakom elementu kolekcije po redu
while (iter.hasNext()) {
   String elem = iter.next();
   System.out.println(elem);
}
```

Sličan generički postupak može se primeniti i za druge oblike rada sa kolekcijama. Tako, u narednom primeru se uklanjaju sve vrednosti `null` iz neke kolekcije `k` tipa, recimo, `Collection<File>` (ukoliko je operacija uklanjanja moguća u toj kolekciji):

```
Iterator<File> iter = k.iterator();

while (iter.hasNext()) {
   File elem = iter.next();
   if (elem == null)
      iter.remove();
}
```

Primena iteratora radi obavljanja neke operacije nad svim objektima kolekcije može se izbeći ukoliko se koristi `for-each` petlja. Pored nizova i nabrojivih tipova, `for-each` petlja može se koristiti i za obradu svih objekata neke kolekcije. Za kolekciju `k` tipa `Collection<T>`, opšti postupak upotrebe `for-each` petlje ima ovaj oblik:

```
for (T x : k) {  // za svaki objekat x tipa T u kolekciji k
   .
   . // Obrada objekta x
   .
}
```

Kontrolnoj promenljivi `x` u ovoj petlji redom se dodeljuje svaki objekat iz kolekcije `k` i zatim se izvršava telo petlje. Pošto su objekti u kolekciji `k` tipa `T`, promenljiva `x` mora biti definisana da bude istog tipa `T`. Na primer, prikazivanje imena svih datoteka u kolekciji `dir` tipa `Collection<File>` može se izvesti na sledeći način:

```
for (File datoteka : dir) {
   if (datoteka != null)
      System.out.println(datoteka.getName());
}
```

Naravno, ekvivalentno rešenje ovog primera može se napisati i upotrebom iteratora i `while` petlje, ali se korišćenjem `for-each` petlje dobija razumljiviji program.

Liste

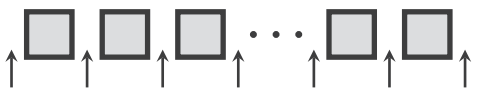
Lista na apstraktnom nivou predstavlja kolekciju objekata koji su linearno uređeni. Sekvencijalni poredak elemenata liste definisan je time što je određen prvi element liste, iza njega drugi element liste i tako dalje sve do poslednjeg elementa liste. U Javi, ovaj naopštiji koncept liste objekata tipa `T` obuhvaćen je parametrizovanim interfejsom `List<T>`. Interfejs `List<T>` nasleđuje interfejs `Collection<T>` i sadrži dodatne metode koji omogućavaju pristup elementima liste prema njihovoj numeričkoj poziciji u listi. U nastavku je prikazan spisak ovih metoda za listu `l` tipa `List<T>`. Primetimo da njihova funkcionalnost zavisi od metoda `size()` u interfejsu `Collection<T>`.

- `l.get(i)` vraća objekat tipa `T` koji se nalazi na *i*-toj poziciji u listi `l`. Indeks `i` je broj tipa `int` koji mora biti u intervalu 0,1,...,`l.size()`−1, jer se u suprotnom izbacuje izuzetak tipa `IndexOutOfBoundsException`.
- `l.set(i,o)` zamenjuje objekat na *i*-toj poziciji u listi `l` datim objektom `o`. Objekat `o` mora biti tipa `T`. Izvršavanjem ovog metoda se ne menja veličina liste `l`, niti dolazi do pomeranja njenih drugih elemenata.
- `l.add(i,o)` dodaje u listu `l` dati objekat `o` na *i*-toj poziciji. Objekat `o` mora biti tipa `T`. Izvršavanjem ovog metoda se veličina liste `l` uvećava za jedan, a elementi iza *i*-te pozicije u listi se pomeraju za jedno mesto udesno da bi se umetnuo novi element. Indeks `i` mora biti ceo broj u intervalu od `0` do `l.size()`, a ukoliko je jednak `l.size()`, novi element se dodaje na kraj liste.

- `l.remove(i)` uklanja iz liste `l` objekat koji se nalazi na *i*-toj poziciji i vraća taj objekat kao rezultat metoda. Izvršavanjem ovog metoda se veličina liste `l` smanjuje za jedan, a elementi iza *i*-te pozicije u listi se pomeraju za jedno mesto ulevo da bi se popunilo mesto uklonjenog elementa. Indeks `i` mora biti ceo broj u intervalu od 0 do `l.size()`−1.
- `l.indexOf(o)` vraća ceo broj tipa `int` jednak poziciji datog objekta `o` u listi `l`. Ukoliko se objekat `o` ne nalazi u listi, vraćena vrednost je −1; ukoliko lista sadrži više objekata `o`, vraćena vrednost jednaka je poziciji prvog pojavljivanja tog objekta od početka liste. Objekat `o` može biti bilo kog tipa, ne samo tipa `T`.

Ako je `l` lista tipa `List<T>`, onda se pozivom `l.iterator()` dobija iterator tipa `Iterator<T>` za pristupanje elementima liste od početka do kraja. U interfejsu `List<T>` dodatno je definisan metod `listIterator()` koji kao rezultat daje iterator tipa `ListIterator<T>`. Interfejs `ListIterator<T>` sadrži ne samo uobičajene metode `hasNext()`, `next()` i `remove()`, nego i metode `hasPrevious()`, `previous()` i `add(o)` koji su dodatno namenjeni samo za rad sa listom elemenata. Ovi dodatni metodi omogućavaju da se duž liste prelazi od jednog do drugog elementa ne samo sleva na desno, nego i zdesna na levo, kao i da se pri tome modifikuje lista.

Da bi se bolje razumeo način rada iteratora za liste, najpre treba imati u vidu da iterator uvek pokazuje bilo između dva elementa u listi ili ispred prvog elementa liste ili iza poslednjeg elementa liste. Ovo je slikovito prikazano na slici 11.1 na kojoj su strelicama naznačene moguće pozicije iteratora između elemenata liste.



Slika 11.1: Pozicije iteratora u listi.

Ako je `iter` iterator tipa `ListIterator<T>`, naredbom `iter.next()` pomera se pokazivač iteratora za jedno mesto udesno i vraća element liste preko kojeg je pokazivač prešao. Slično, naredbom `iter.previous()` pomera

se pokazivač iteratora za jedno mesto ulevo i vraća element liste preko kojeg je pokazivač prešao. Naredbom `iter.remove()` uklanja se element iz liste koji je vraćen poslednjim pozivom metoda `next()` ili `remove()`. Najzad, naredbom `iter.add(o)` dodaje se dati objekat `o`, koji mora biti tipa `T`, u listu na mesto neposredno *ispred* trenutne pozicije pokazivača iteratora.

Radi praktične ilustracije rada sa iteratorom liste, razmotrimo problem sortirane liste stringova u rastućem redosledu i dodavanja novog stringa u tu listu tako da proširena lista bude i dalje sortirana. U nastavku je prikazan metod u kome se koristi iterator tipa `ListIterator` za nalaženje pozicije u listi sortiranih stringova `l` na kojoj treba dodati novi string `s`. U ovom metodu, pokazivač tog iteratora se pomera od početka liste `l` za jedno mesto preko svih elemenata koji su manji od novog elementa `s`. Kada se pri tome naiđe na prvi element liste koji je veći od novog elementa, na tom mestu se u listi dodaje novi element koristeći metod `add()` iteratora liste.

```
public void dodajSortLista(List<String> l, String s) {

   ListIterator<String> iter = l.listIterator();

   while (iter.hasNext()) {
      String elem = iter.next();
      if (s.compareTo(elem) <= 0) {
         // Novi string s treba da bude ispred elementa elem,
         // ali je pokazivač iteratora pomeren iza tog elementa.
         iter.previous();
         break;
      }
   }
   iter.add(s);
}
```

Klase `ArrayList` i `LinkedList`. Praktična reprezentacija opšteg pojma liste elemenata u Javi i definicija metoda u interfejsu `List<T>` zasniva se na dinamičkim nizovima i povezanim čvorovima. Ova dva načina obuhvaćena su dvema konkretnim klasama, `ArrayList` i `LinkedList`, koje su deo Sistema kolekcija u Javi.

Parametrizovane klase `ArrayList<T>` i `LinkedList<T>`, koje se nalaze u paketu `java.util`, predstavljaju listu elemenata pomoću dinamičkog niza i povezanih čvorova. Obe klase implementiraju interfejs `List<T>`, a time i `Collection<T>`. Jedan objekat tipa `ArrayList<T>` predstavlja linearno uređenu listu objekata tipa `T` koji se čuvaju kao elementi dinamičkog niza, odnosno dužina niza se automatski povećava kada je to neophodno prilikom dodavanja novih objekata u listu. Jedan objekat tipa `LinkedList<T>` takođe predstavlja linearno uređenu listu objekata tipa `T`, ali se ti objekti čuvaju u formi čvorova koji su međusobno povezani pokazivačima (referencama).

Kako obe klase `ArrayList<T>` i `LinkedList<T>` implementiraju interfejs `List<T>`, odnosno omogućavaju iste osnovne operacije nad listom, postavlja se pitanje zašto su potrebne dve implementacije liste umesto samo jedne od njih? Odgovor leži u efikasnosti pojedinih operacija u dvema reprezentacijama liste. Naime, za neke operacije su povezane liste efikasnije od nizova, dok za druge operacije nad listom važi suprotno. Zato u konkretnim primenama listi treba pažljivo odmeriti koje se operacije često koriste i prema tome odabrati optimalnu implementaciju liste.

Povezana lista predstavljena klasom `LinkedList<T>` efikasnija je u primenama kod kojih se elementi liste često dodaju ili uklanjaju u sredini liste. Ove operacije kod liste predstavljene dinamičkim nizom tipa `ArrayList<T>` zahtevaju pomeranje približno polovine elementa niza udesno ili ulevo radi pravljenja mesta za dodavanje novog elementa ili popunjavanje praznine nakon uklanjanja nekog elementa. S druge strane, lista predstavljena dinamičkim nizom efikasnija je kada treba pročitati ili promeniti proizvoljni element liste. U takvom slučaju se traženom elementu liste pristupa prostim korišćenjem njegovog indeksa u dinamičkom nizu, dok se kod povezane liste moraju redom ispitati svi elementi od početka do, u najgorem slučaju, kraja liste. Zbog toga, mada klase `ArrayList<T>` i `LinkedList<T>` implementiraju sve metode u interfejsu `List<T>`, neki od tih metoda efikasni su samo za pojedinu reprezentaciju liste.

U klasi `LinkedList<T>` se nalaze neki metodi koji nisu definisani u klasi `ArrayList<T>`. Tako, ako je `ll` lista tipa `LinkedList<T>`, onda se za taj objekat mogu koristiti i ovi metodi:

- `ll.getFirst()` vraća objekat tipa `T` koji se nalazi na samom početku liste `ll`. Pri tome lista ostaje nepromenjena, odnosno prvi element

liste se ne uklanja. A ukoliko je lista prazna, izbacuje se izuzetak tipa `NoSuchElementException`.

- `ll.getLast()` vraća objekat tipa `T` koji se nalazi na samom kraju liste `ll`. Pri tome lista ostaje nepromenjena, odnosno poslednji element liste se ne uklanja. A ukoliko je lista prazna, izbacuje se izuzetak tipa `NoSuchElementException`.
- `ll.removeFirst()` uklanja prvi element liste `ll` i vraća taj objekat tipa `T` kao rezultat. Izuzetak tipa `NoSuchElementException` izbacuje se ukoliko je lista prazna.
- `ll.removeLast()` uklanja poslednji element liste `ll` i vraća taj objekat tipa `T` kao rezultat. Ukoliko je lista prazna, izbacuje se izuzetak tipa `NoSuchElementException`.
- `ll.addFirst(o)` dodaje novi objekat `o`, koji mora biti tipa `T`, na početak liste `ll`.
- `ll.addLast(o)` dodaje novi objekat `o`, koji mora biti tipa `T`, na kraj liste `ll`.

Primer: sortiranje liste

Kao što je poznato, u klasi `Arrays` iz paketa `java.util` nalazi se statički metod `sort()` kojim se može efikasno sortirati standardni niz elemenata u rastućem redosledu. Tako, naredbom

```
Arrays.sort(a);
```

sortira se niz *a*. Elementi niza *a* mogu biti jednog od primitivnih tipova (osim `boolean`) ili mogu biti objekti klase koja implementira interfejs `Comparable`.

Svi metodi za efikasno sortiranje niza zasnivaju se na činjenici da se svakom elementu niza može lako pristupiti preko njegovog indeksa. Postavlja se pitanje da li se opšta lista elemenata može efikasno sortirati s obzirom na to da pristup svim elementima liste nije vremenski podjednak — na primer, da bi se koristio neki element u sredini liste moraju se redom proći svi elementi liste od njenog početka do tog elementa. To je suštinski različito od načina na koji se koriste elementi niza, gde je vreme pristupa prvom,

srednjem ili bilo kom elementu niza potpuno jednako zahvaljujući indeksima.

Jedan očigledan postupak za sortiranje liste je da se od date liste obrazuje niz kopirajući redom elemente liste u jedan niz, zatim da se efikasno sortira novoformirani niz i, na kraju, da se od sortiranog niza obrazuje sortirana lista kopirajući redom elemente niza. U stvari, sličan način se koristi u statičkom metodu `sort()` u klasi `Collections` iz paketa `java.util`. Ova klasa sadrži razne pomoćne metode za rad sa opštim kolekcijama, pa i listama. Tako, ukoliko je `l` lista tipa `List<T>`, naredbom

```
Collections.sort(l);
```

sortira se lista `l` u rastućem redosledu. Elementi liste `l` moraju biti objekti klase koja implementira interfejs `Comparable<T>`.

Da bismo pokazali da se lista elemenata može efikasno i direktno sortirati, u ovom primeru govorimo o postupku za sortiranje liste koji se naziva *sortiranje objedinjavanjem* (engl. *merge-sort*). Ovaj postupak je po prirodi rekurzivan i sastoji se od tri glavna koraka:

1. Data lista ℓ se najpre deli na dva dela u dve duplo manje liste ℓ_1 i ℓ_2.
2. Ove manje liste ℓ_1 i ℓ_2 se zatim rekurzivno sortiraju istim postupkom.
3. Sortirane manje liste ℓ_1 i ℓ_2, dobijene u prethodnom koraku, na kraju se objedinjuju u jednu sortiranu listu ℓ.

Efikasnost celog postupka sortiranja objedinjavanjem zasniva se upravo na brzini njegovog trećeg koraka. (Prvi korak se očigledno može brzo izvršiti alternativnim kopiranjem elemenata date liste u prvu ili drugu manju listu.) Naime, dve sortirane manje liste mogu se efikasno objediniti u jednu sortiranu listu sekvencijalnim uparivanjem elemenata prve i druge liste i kopiranjem manjeg elementa iz odgovarajuće liste u rezultujuću listu.

U nastavku je prikazan metod u kome se primenjuje ovaj pristup za sortiranje liste. Radi konkretnosti, uzeto je da se lista sastoji od celih brojeva, mada se metod može lako prilagoditi za listu čiji su elementi bilo kog konkretnog tipa koji implementira interfejs `Comparable<T>`. Još bolje, metod se može napisati tako da bude generički metod koji bi se onda mogao koristiti za liste parametrizovanog tipa. O tome se govori pri kraju ovog poglavlja na strani 439.

```
public static void mergeSort(List<Integer> l) {

   if (l.size() > 1) {  // lista l ima bar dva elementa

      // 1. Podeliti listu l u dve polovine l1 i l2
      List<Integer> l1 = new LinkedList<Integer>();
      List<Integer> l2 = new LinkedList<Integer>();

      ListIterator<Integer> it = l.listIterator();
      boolean dodajL1 = true;  // element liste l ide u l1 ili l2

      while (it.hasNext()) {
         Integer e = it.next();
         if (dodajL1)
            l1.add(e);
         else
            l2.add(e);
         dodajL1 = !dodajL1;
      }

      // 2. Rekurzivno sortirati liste l1 i l2
      mergeSort(l1);
      mergeSort(l2);

      // 3. Objediniti sortirane liste l1 i l2 u sortiranu listu l
      l.clear();

      ListIterator<Integer> it1 = l1.listIterator();
      ListIterator<Integer> it2 = l2.listIterator();

      Integer e1 = it1.next();  // l1 ima bar jedan element
      Integer e2 = it2.next();  // l2 ima bar jedan element

      while (true) {
         if (e1.compareTo(e2) <= 0) {
            l.add(e1);
            if (it1.hasNext())
```

```
                e1 = it1.next();
             else {
                l.add(e2);
                while (it2.hasNext())
                   l.add(it2.next());
                break;
             }
          }
          else {
             l.add(e2);
             if (it2.hasNext())
                e2 = it2.next();
             else {
                l.add(e1);
                while (it1.hasNext())
                   l.add(it1.next());
                break;
             }
          }
       }
    }
}
```

Skupovi

Skup je kolekcija objekata u kojoj nema duplikata, odnosno nijedan objekat u skupu se ne pojavljuje dva ili više puta. U Javi, ovaj naopštiji koncept skupa objekata tipa `T` obuhvaćen je parametrizovanim interfejsom `Set<T>`. Interfejs `Set<T>` nasleđuje interfejs `Collection<T>` i obezbeđuje da se nijedan objekat ne pojavljuje dvaput u skupu. Tako, ako je `s` objekat tipa `Set<T>`, onda naredba `s.add(o)` ne proizvodi nikakav efekat ukoliko se objekat `o` već nalazi u skupu `s`.

Praktična reprezentacija opšteg pojma skupa elemenata u Javi i definicija metoda u interfejsu `Set<T>` zasniva se na binarnim stablima i heš tabelama. Ova dva načina obuhvaćena su dvema konkretnim generičkim klasama `TreeSet<T>` i `HashSet<T>` u paketu `java.util`.

Skup predstavljen klasom `TreeSet` ima dodatnu osobinu da su njegovi elementi uređeni u rastućem redosledu. Pored toga, iterator za takav skup uvek prolazi kroz elemente skupa u ovom rastućem redosledu. Ovo sa druge strane ograničava vrstu objekata koji mogu pripadati skupu tipa `TreeSet`, jer se njegovi objekti moraju upoređivati da bi se odredio njihov rastući redosled. Praktično to znači da objekti u skupu tipa `TreeSet<T>` moraju implementirati interfejs `Comparable<T>` tako da relacija `o1.compareTo(o2)` ima razumnu interpretaciju za svaka dva objekta `o1` i `o2` u skupu. Alternativno, kada se konstruiše skup tipa `TreeSet<T>` može se kao parametar konstruktora navesti objekat tipa `Comparator<T>`. U tom slučaju se za poređenje objekata u konstruisanom skupu koristi metod `compare()` komparatora.

U klasi `TreeSet` se ne koristi metod `equals()` za utvrđivanje jednakosti dva objekta u skupu, nego metod `compareTo()` (ili `compare()`). To ponekad može dovesti do nelogičnosti, jer jednakost objekata na osnovu rezultata metoda `compareTo()` ne mora biti ekvivalentna njihovoj „prirodnoj" jednakosti koja se dobija kao rezultat metoda `equals()`. Na primer, dva objekta koji predstavljaju adrese prirodno su jednaka ukoliko su svi delovi adresa jednaki, odnosno jednaki su i ulica i broj i grad i poštanski broj. Ako bi se adrese upoređivale metodom `compareTo()` samo na osnovu, recimo, poštanskog broja, onda bi se sve adrese sa istim poštanskim brojem smatrale jednakim. To bi onda značilo da bi se u skupu tipa `TreeSet` mogla nalaziti samo jedna adresa sa određenim poštanskim brojem, što verovatno nije pravi način za predstavljanje skupa adresa.

Ovaj potencijalni problem u radu sa skupovima tipa `TreeSet` treba imati na umu i stoga treba obezbediti da metod `compareTo()` bude definisan na razuman način za objekte konkretnog skupa. Srećom, o ovome ne treba brinuti za objekte standardnih tipova `String`, `Integer` i mnogih drugih, jer se za njih poklapa prirodna jednakost i ona dobijena na osnovu metoda `compareTo()`.

Činjenica da su elementi skupa tipa `TreeSet` sortirani i da takav skup ne sadrži duplikate može biti vrlo korisna u nekim primenama. Pretpostavimo da je, recimo, `k` neka kolekcija stringova tipa `Collection<String>`. (Tip objekata `String` u kolekciji nije bitan, nego to da je za njega pravilno definisan metod `compareTo()`.) Ako je potrebno sortirati objekte u kolek-

ciji k i ukloniti duplikate u njoj, to se može uraditi na jednostavan način pomoću jednog skupa tipa TreeSet:

```
TreeSet<String> s = new TreeSet<String>();
s.addAll(k);
```

Drugom naredbom u ovom primeru se svi objekti kolekcije k dodaju skupu s. Pri tome, kako je s tipa TreeSet, duplikati u skupu s se uklanjaju i elementi tog skupa se uređuju u rastućem redosledu.

Elementi skupa mogu se lako dodeliti nekoj drugoj strukturi podataka. Tako, nastavljajući prethodni primer, od elemenata skupa s može se formirati sortirana lista tipa ArrayList<String>:

```
TreeSet<String> s = new TreeSet<String>();
s.addAll(k);
ArrayList<String> l = new ArrayList<String>();
l.addAll(s);
```

U stvari, svaka klasa u Sistemu kolekcija u Javi poseduje konstruktor koji ima parametar tipa Collection. Ukoliko se koristi taj konstruktor za konstruisanje nove kolekcije, svi elementi kolekcije koja je navedena kao argument konstruktora dodaju se inicijalno novo konstruisanoj kolekciji. Na primer, ako je k tipa Collection<String>, onda se izrazom

```
new TreeSet<String>(k)
```

konstruiše skup tipa TreeSet<String> koji ima iste elemente kao kolekcija k, ali bez duplikata i sa sortiranim elementima. To znači da se četiri naredbe prethodnog primera za formiranje sortirane liste stringova mogu kraće zapisati koristeći samo jednu naredbu:

```
ArrayList<String> l =
   new ArrayList<String>(new TreeSet<String>(k));
```

* * *

U klasi HashSet je primenjen potpuno drugačiji način predstavljanja skupa elemenata u Javi. U ovom slučaju se elementi skupa čuvaju u strukturi podataka koja se naziva *heš tabela*. Bez ulaženja u mnogo detalja, osnovna odlika ove strukture podataka jeste da ona obezbeđuje vrlo efikasne operacije nalaženja, dodavanja i uklanjanja elemenata. Te operacije su mnogo brže, u proseku, od sličnih operacije za skupove predstavljene klasom Tree-

`Set`. S druge strane, objekti nekog skupa tipa `HashSet` nisu uređeni u posebnom redosledu i ne moraju da implementiraju interfejs `Comparable`. Zato jedan iterator za skup tipa `HashSet` obilazi elemente tog skupa u nepredvidljivom redosledu, koji čak može biti potpuno drugačiji nakon dodavanja novog elementa skupu. Za određivanje jednakosti dva elementa u skupu tipa `HashSet` koristi se metod `equals()`. Prema tome, efikasniju klasu `HashSet` u odnosu na klasu `TreeSet` treba koristiti za predstavljanje skupa elementa ukoliko za njegove elemente nije definisana relacija poređenja veće ili manje, ili to nije važno u konkretnoj primeni.

Mape

Mapa je generalizacija računarskog koncepta niza elemenata, odnosno realizacija matematičkog pojma preslikavanja ili funkcije. Naime, niz a od n elemenata može se smatrati preslikavanjem brojeva $0, 1, \ldots, n-1$ u odgovarajuće elemente niza: ako je i neki od ovih brojeva, njegova slika je element a_i. Dve fundamentalne operacije nad nizom elemenata su *get*, koja za dati indeks i vraća vrednost elementa a_i, kao i *put*, koja datu novu vrednost za dati indeks i upisuje kao novu vrednost elementa a_i.

Po ovoj analogiji, mapa je određena preslikavanjem objekata proizvoljnog tipa `T`, a ne celih brojeva $0, 1, \ldots, n-1$ kao kod niza, u objekte potencijalno različitog tipa `S`. Isto tako, osnovne operacije *get* i *put* analogno se proširuju nad objektima tipa `T` kojima su pridruženi objekti tipa `S`. Prema tome, mapa koncepcijski podseća na niz, osim što za indekse ne služe celi brojevi nego objekti proizvoljnog tipa.

Objekti koji u mapi služe kao indeksi nazivaju se *ključevi*, dok se objekti koji su pridruženi ključevima nazivaju *vrednosti*. Jedna mapa može se dakle smatrati skupom „pridruživanja", pri čemu je svako pridruživanje određeno parom ključ/vrednost. Obratite pažnju na to da svaki ključ može odgovarati najviše jednoj vrednosti, ali jedna vrednost može biti pridružena većem broju različitih ključeva.

U Javi, mape su predstavljene interfejsom `Map<T,S>` iz paketa `java.util`. Ovaj interfejs je parametrizovan dvoma tipovima: prvi parametar `T` određuje tip objekata koji predstavljaju ključeve mape, a drugi parametar `S` određuje tip objekata koji su vrednosti mape. Mapa tipa `Map<Date,Boolean>`, na

primer, definiše preslikavanje ključeva tipa `Date` u vrednosti tipa `Boolean`. U mapi tipa `Map<String,String>` ključevi i vrednosti su istog tipa `String`.

U interfejsu `Map<T,S>` nalaze se metodi `get()` i `put()`, kao i drugi opšti metodi za rad sa mapama. Tako, ako je `m` promenljiva tipa `Map<T,S>` za neke specifične tipove `T` i `S`, onda se mogu koristiti sledeći metodi:

- `m.get(k)` vraća objekat tipa `S` koji je pridružen datom ključu `k` u mapi `m`. Ključ `k` ne mora biti objekat isključivo tipa `T`, nego može biti bilo koji objekat. Ako nijedna vrednost nije pridružena ključu `k` u mapi, vraćena vrednost je `null`. Primetimo da je efekat zapisa `m.get(k)` sličan rezultatu zapisa `a[k]` za običan niz `a` ukoliko je `k` indeks tog niza.
- `m.put(k,v)` u mapi `m` pridružuje datu vrednost `v` datom ključu `k`. Ključ `k` mora biti tipa `T` i vrednost `v` mora biti tipa `S`. Ako mapa već sadrži par sa datim ključem, onda u tom paru nova vrednost zamenjuje staru. Ovo je slično efektu naredbe `a[k] = v` za običan niz `a`.
- `m.remove(k)` uklanja par ključ/vrednost u mapi `m` koji odgovara datom ključu `k`. Ključ `k` ne mora biti objekat isključivo tipa `T`, nego može biti bilo koji objekat.
- `m.clear()` uklanja sve parove ključ/vrednost koje sadrži mapa `m`.
- `m.containsKey(k)` vraća logičku vrednost tačno ukoliko mapa `m` sadrži par ključ/vrednost koji odgovara datom ključu `k`. Ključ `k` ne mora biti objekat isključivo tipa `T`, nego može biti bilo koji objekat.
- `m.containsValue(v)` vraća logičku vrednost tačno ako je data vrednost `v` pridružena nekom ključu u mapi `m`. Vrednost `v` ne mora biti objekat isključivo tipa `S`, nego može biti bilo koji objekat.
- `m.size()` vraća celobrojnu vrednost tipa `int` koja predstavlja broj parova ključ/vrednost u mapi `m`.
- `m.isEmpty()` vraća logičku vrednost tačno ukoliko je mapa `m` prazna, odnosno ne sadrži nijedan par ključ/vrednost.
- `m.putAll(mm)` prepisuje u mapu `m` sve parove ključ/vrednost koje sadrži druga mapa `mm` tipa `Map<T,S>`.

Dve praktične implementacije interfejsa `Map<T,S>` u Javi obuhvaćene su klasama `TreeMap<T,S>` i `HashMap<T,S>`. Parovi ključ/vrednost u mapi koja je predstavljena klasom `TreeMap` čuvaju se u strukturi binarnog stabla. Pri tome su parovi mape sortirani prema njihovim ključevima, te zato ovi ključevi moraju biti uporedivi. To znači da bilo tip ključeva `T` mora implementirati interfejs `Comparable<T>` ili se mora navesti komparator za poređenje ključeva kao parametar konstruktora klase `TreeMap`. Obratite pažnju na to da se, slično skupu tipa `TreeSet`, za mapu tipa `TreeMap` koristi metod `compareTo()` radi utvrđivanja da li su dva ključa jednaka. Ovo treba imati na umu prilikom korišćenja mape tipa `TreeMap`, jer ukoliko se prirodna jednakost ključeva ne podudara sa onom koju indukuje metod `compareTo()`, to može imati iste neželjene posledice o kojima smo govorili kod skupova.

U drugoj implementaciji, parovi ključ/vrednost u mapi koja je predstavljena klasom `HashMap` čuvaju se u heš tabeli. Zbog toga parovi takve mape nisu uređeni po nekom posebnom redosledu ključeva, odnosno ključevi takve mape ne moraju biti uporedivi. Ali s druge strane, klasa ključeva mora imati dobre definicije metoda `equals()` i `hashCode()`, što je obezbeđeno u svim standardnim klasama u Javi.

Većina operacija u radu sa mapom nešto je efikasnija ukoliko se koristi klasa `HashMap` u odnosu na klasu `TreeMap`. Zato je obično bolje koristiti klasu `HashMap` ukoliko u programu nema potrebe za sortiranim redosledom ključeva koji obezbeđuje klasa `TreeMap`. Specifično, ukoliko se u programu koriste samo operacije *get* i *put* za neku mapu, dovoljno je koristiti klasu `HashMap`.

Primer: telefonski imenik kao mapa

Jedan dobar primer koncepta mape u programiranju je struktura podataka koju obrazuje telefonski imenik. Telefonski imenik je kolekcija kontakata koji se sastoje od dva podatka: imena osobe i njenog telefonskog broja. Dodatno, radi jednostavnosti, ovde se podrazumeva najprostiji slučaj imenika u kome svaka osoba može imati samo jedan telefonski broj. Osnovne operacije nad takvim telefonskim imenikom su:

- Dodavanje novog kontakta (imena osobe i telefonskog broja) u imenik. Ako se data osoba već nalazi u imeniku, onda se njen stari broj

zamenjuje novim.

- Uklanjanje jednog kontakta sa datim imenom osobe iz imenika.
- Nalaženje telefonskog broja u imeniku za dato ime osobe.

Telefonski imenik kao struktura podataka može se prosto realizovati kao dinamički niz elemenata, pri čemu svaki element niza predstavlja jedan kontakt imenika. Za takav niz onda osnovne operacije nad telefonskim imenikom predstavljaju obične operacije dodavanja, uklanjanja i nalaženja elemenata jednog niza. Ako se podsetimo, ovaj pristup je naime primenjen u programu u listingu 3.5 na strani 131.

Elegantnije rešenje, međutim, može se dobiti ukoliko se primeti da je jednostavni telefonski imenik upravo jedna mapa kojom su telefonski brojevi pridruženi imenima osoba. U nastavku je prikazana klasa u kojoj je imenik predstavljen mapom tipa `Map<String,String>`. Dodavanje nove stavke u imenik je onda obična operacija *put* za mapu, dok nalaženje telefonskog broja u imeniku odgovara operaciji *get*.

```
import java.util.*;

public class TelImenik {

   private Map<String,String> imenik;

   // Konstruktor
   public TelImenik() {
      imenik = new HashMap<String,String>();
   }

   public String nađiBroj(String imeOsobe) {
      return imenik.get(imeOsobe);
   }

   public void dodajKontakt(String imeOsobe, String brojOsobe) {
      imenik.put(imeOsobe,brojOsobe);
   }

   public void ukloniKontakt(String imeOsobe) {
```

```
            imenik.remove(imeOsobe);
        }
    }
```

Pogledi na mape i kolekcije

Tehnički jedna mapa nije kolekcija i zato za mape nisu definisane sve operacije koje su moguće za kolekcije. Specifično, za mape nisu definisani iteratori. S druge strane, u primenama je često potrebno redom obraditi sve parove ključ/vrednost koje sadrži jedna mapa. U Javi se ovo može uraditi na indirektan način preko takozvanih *pogleda*. Ako je `m` promenljiva tipa `Map<T,S>`, onda se pozivom metoda `keySet()` za mapu `m`

```
m.keySet()
```

dobija skup objekata koje čine ključevi u svim parovima ključ/vrednost mape `m`. Objekat koji se vraća pozivom metoda `keySet()` implementira interfejs `Set<T>`. To je dakle skup čiju su elementi svi ključevi mape.

Metod `keySet()` je specifičan po tome što njegov rezultat nije nezavisan skupovni objekat tipa `Set<T>`, nego se pozivom `m.keySet()` dobija „pogled" na aktuelne ključeve koji se nalaze u mapi `m`. Ovaj pogled na mapu implementira interfejs `Set<T>`, ali na poseban način tako da metodi definisani u tom interfejsu direktno manipulišu ključevima u mapi. Tako, ukoliko se ukloni neki ključ iz pogleda, taj ključ (zajedno sa pridruženom vrednošću) zapravo se uklanja iz same mape. Slično, nije dozvoljeno dodavanje ključa nekom pogledu, jer dodavanje ključa mapi bez njegove pridružene vrednosti nije moguće. S druge strane, kako se pozivom `m.keySet()` ne konstruiše novi skup, metod `keySet()` je vrlo efikasan.

Budući da su za skupove definisani iteratori, iterator za skup ključeva u mapi može se iskoristiti da bi se redom obradili svi parovi koje sadrži mapa. Na primer, ako je `m` mapa tipa `Map<String,Double>`, onda se na sledeći način mogu prikazati svi parovi ključ/vrednost u mapi `m`:

```
Set<String> ključevi = m.keySet();  // skup ključeva mape
Iterator<String> ključIter = ključevi.iterator();

System.out.println("Mapa sadrži sledeće parove ključ/vrednost:");
```

```
// Prikazivanje para (ključ,vrednost) za
// sve ključeve u skupu ključeva mape
while (ključIter.hasNext()) {
   String ključ = ključIter.next(); // sledeći ključ u mapi
   Double vrednost = m.get(ključ);  // pridružena vrednost ključu
   System.out.println("(" + ključ + "," + vrednost + ")");
}
```

Bez korišćenja iteratora, isti zadatak se može elegantnije rešiti pomoću `for-each` petlje:

```
System.out.println("Mapa sadrži sledeće parove ključ/vrednost:");

// Prikazivanje para (ključ,vrednost) za
// sve ključeve u skupu ključeva mape
for (String ključ : m.keySet()) {
   Double vrednost = m.get(ključ);
   System.out.println("(" + ključ + "," + vrednost + ")");
}
```

Ako je mapa tipa `TreeMap`, onda su ključevi u mapi sortirani i zato se korišćenjem iteratora za skup ključeva mape dobijaju ključevi u rastućem redosledu. Za mapu tipa `HashMap` ključevi se dobijaju u proizvoljnom, nepredvidljivom redosledu.

U interfejsu `Map<T,S>` definisana su još dva pogleda. Ako je `m` mapa tipa `Map<T,S>`, prvi pogled se dobija metodom

```
m.values()
```

koji vraća objekat tipa `Collection<S>` koji sadrži sve vrednosti iz parova ključ/vrednost u mapi `m`. Ovaj objekat je kolekcija a ne skup, jer rezultat može sadržati duplikate elemenata pošto u mapi ista vrednost može biti pridružena većem broju ključeva.

Drugi pogled se dobija metodom

```
m.entrySet()
```

koji daje skup čiji su elementi svi parovi ključ/vrednost u mapi `m`. Ovi elementi su objekti tipa `Map.Entry<T,S>` koji je definisan kao statički ugnježđeni interfejs unutar interfejsa `Map<T,S>`. Zbog toga se njegovo puno ime

zapisuje tačka-notacijom, ali to ne znači da se ne može koristiti na isti način kao bilo koje drugo ime tipa. Primetimo da je rezultat metoda `m.entrySet()` skup čiji su elementi tipa `Map.Entry<T,S>`. Ovaj skup je dakle tipa

```
Set<Map.Entry<T,S>>
```

odnosno parametar tipa u ovom slučaju je i sam jedan parametrizovan tip.

Skup koji je rezultat poziva metoda `m.entrySet()` nosi zapravo iste informacije o parovima ključ/vrednost kao i mapa `m`, ali taj skup pruža različit pogled na njih obezbeđujući drugačije operacije. Svaki element ovog skupa je objekat tipa `Map.Entry` koji sadrži jedan par ključ/vrednost u mapi. U interfejsu `Map.Entry` definisani su metodi `getKey()` i `getValue()` za dobijanje ključa i vrednosti koji čine par, kao i metod `setValue(v)` za promenu vrednosti određenog para u novu vrednost `v`. Obratite pažnju na to da se korišćenjem metoda `setValue(v)` za objekat tipa `Map.Entry` modifikuje sama mapa, odnosno to je ekvivalentno korišćenju metoda `put()` za mapu uz odgovarajući ključ.

Radi ilustracije, prikazivanje svih parova ključ/vrednost u mapi može se efikasnije rešiti primenom skupovnog pogleda na mapu nego izdvajanjem njenog skupa ključeva, kao što je to urađeno u prethodnom primeru. Razlog veće efikasnosti ovog pristupa je u tome što se ne mora koristiti metod `get()` radi traženja vrednosti koja je pridružena svakom ključu. Ako je mapa `m` tipa `Map<String,Double>` kao u prethodnom primeru, onda se prikazivanje njenog sadržaja može efikasno izvesti na sledeći način:

```
Set<Map.Entry<String,Double>> parovi = m.entrySet();
Iterator<Map.Entry<String,Double>> parIter = parovi.iterator();

System.out.println("Mapa sadrži sledeće parove ključ/vrednost:");

// Prikazivanje para (ključ,vrednost) za
// sve ključeve u skupu ključeva mape
while (parIter.hasNext()) {
   Map.Entry<String,Double> par = parIter.next();
   String ključ = par.getKey();       // izdvojiti ključ
   Double vrednost = par.getValue();  // izdvojiti vrednost
   System.out.println("(" + ključ + "," + vrednost + ")");
}
```

Naravno, isti zadatak se može kraće rešiti `for-each` petljom:

```
System.out.println("Mapa sadrži sledeće parove ključ/vrednost:");

// Prikazivanje para (ključ,vrednost) za
// sve ključeve u skupu ključeva mape
for (Map.Entry<String,Double> par : m.entrySet())
   System.out.println(
      "(" + par.getKey() + "," + par.getValue() + ")");
```

Treba naglasiti da se u Javi mogu koristiti pogledi ne samo za mape, nego i za neke vrste kolekcija. Tako, *podlista* u interfejsu `List<T>` definisana je kao pogled na deo liste. Ako je `l` lista tipa `List<T>`, onda se metodom

```
l.subList(i,j)
```

dobija pogled na deo liste `l` koji obuhvata elemente koji se nalaze na pozicijama između celobrojnih indeksa `i` i `j`. (Možda malo neobično, podlista sadrži *i*-ti element ali ne sadrži *j*-ti element liste.) Ovaj pogled omogućava rad sa podlistom na isti način kao sa običnom listom, odnosno mogu se koristiti sve operacije koje su definisane za listu. Ta podlista, međutim, nije neka nezavisna lista, jer promene izvršene nad elementima podliste zapravo se odražavaju na originalnu listu.

Pogledi se mogu obrazovati i radi predstavljanja određenih podskupova sortiranih skupova. Ako je `s` skup tipa `TreeSet<T>`, onda se metodom

```
s.subSet(e1,e2)
```

dobija *podskup* tipa `Set<T>` koji sadrži sve elemente skupa `s` koji se nalaze između elemenata `e1` i `e2`. (Preciznije, podskup sadrži sve one elemente skupa `s` koji su veći ili jednaki elementu `e1` i koji su striktno manji od elementa `e2`.) Parametri `e1` i `e2` moraju biti objekti tipa `T`. Na primer, ako je `knjige` skup tipa `TreeSet<String>` koji sadrži naslove svih knjiga u nekoj biblioteci, onda se izrazom `knjige.subSet("M","N")` obrazuje podskup svih knjiga čiji naslovi počinju slovom M. Ovaj podskup predstavlja pogled na originalni skup, odnosno kod formiranja podskupa ne dolazi do kopiranja elemenata skupa. Isto tako, sve promene izvršene nad podskupom, kao što su dodavanje ili uklanjanje elemenata, reflektuju se zapravo na originalni skup.

Za skupove se mogu koristiti još dva pogleda. Podskup `s.headSet(e)`

sadrži sve elemente skupa `s` koji su striktno manji od elementa `e`, dok podskup `s.tailSet(e)` sadrži sve elemente skupa `s` koji su veći ili jednaki elementu `e`.

Pored pogleda za opšte mape koji su pomenuti na početku ovog odeljka, za mape tipa `TreeMap<T,S>` sa sortiranim ključevima mogu se koristiti još tri pogleda kojima se obrazuju *podmape* date mape. Podmapa je konceptualno slična podskupu i predstavlja podskup ključeva originalne mape zajedno sa pridruženim vrednostima. Ako je `m` mapa tipa `TreeMap<T,S>`, onda se metodom

```
s.subMap(k1,k2)
```

dobija podmapa koja sadrži sve parove ključ/vrednost iz mape `m` čiji su ključevi veći ili jednaki ključu `k1` i striktno su manji od ključa `k2`. Druga dva pogleda `m.headMap(k)` i `m.tailMap(k)` za mape definisani su na sličan način kao pogledi `headSet` i `tailSet` za skupove.

Da bismo ilustrovali rad sa podmapama, pretpostavimo da je `imenik` mapa tipa `TreeMap<String,String>` čiji su ključevi imena osoba, a vrednosti su njihovi jedinstveni telefonski brojevi. Prikazivanje telefonskih brojeva svih osoba iz imenika čije ime počinje slovom M može se izvesti na sledeći način:

```
Map<String,String> mParovi = imenik.subMap("M","N");
if (mParovi.isEmpty()) {
   System.out.println("Nema osobe čije ime počinje na M.");
}
else {
   System.out.println(
      "Tel. brojevi osoba čija imena počinju na M:");
   for (Map.Entry<String,String> par : mParovi.entrySet())
      System.out.println(par.getKey() + ": " + par.getValue());
}
```

Primer: pravljenje indeksa knjige

Objekti neke kolekcije ili mape mogu biti bilo kog tipa. To znači da elementi kolekcije ili mape mogu i sami biti kolekcije ili mape. Indeks knjige je prirodan primer jedne takve složene strukture podataka.

Indeks u knjizi se sastoji od liste važnijih izraza koji se pojavljuju u knjizi. Radi lakšeg nalaženja ovih izraza u knjizi, uz svaki izraz se nalazi lista njegovih referenci, tj. lista brojeva strana knjige na kojima se taj izraz pojavljuje. Ukoliko se indeks programski formira skeniranjem teksta knjige, potrebno je indeks predstaviti strukturom podataka koja omogućava efikasno dodavanje željenog izraza u indeks. Na kraju, formirani indeks treba prikazati (odštampati) tako da njegovi izrazi budu prikazani po azbučnom redu.

S obzirom na mnogo detalja celog problema pravljenja indeksa knjige koji mogu zakloniti suštinu, ovde ćemo razmotriti samo bitne delove rešenja ovog problema u Javi primenom generičkih struktura podataka. Kako su izrazi u knjizi međusobno različiti, indeks se može predstaviti mapom u kojoj se lista referenci pridružuje svakom izrazu. Izrazi su dakle ključevi mape, a vrednosti pridružene tim ključevima su liste referenci za odgovarajući izraz. Zahtev da prikazivanje formiranog indeksa bude po azbučnom redu izraza ostavlja jedini izbor u pogledu tipa mape koja predstavlja indeks: `TreeMap`. Ako se podsetimo, tip `TreeMap` za razliku od tipa `HashMap` obezbeđuje lako dobijanje ključeva mape u sortiranom redosledu.

Vrednost pridružena svakom ključu mape koja reprezentuje indeks knjige treba da bude lista brojeva strana za odgovarajući izraz. Brojevi strana u svakoj ovoj listi su međusobno različiti, tako da je to zapravo skup brojeva, a ne lista u smislu strukture podataka. Pored toga, brojevi strana za svaki izraz se prikazuju u rastućem redosledu, što znači da ovaj skup brojeva treba da bude tipa `TreeSet`. Brojevi u ovom skupu su primitivnog tipa `int`, ali pošto elementi generičke strukture podataka u Javi moraju biti objekti, za brojeve skupa se mora koristiti omotačka klasa `Integer`.

Sve u svemu, indeks knjige se može predstaviti mapom tipa `TreeMap` čije ključeve čine izrazi tipa `String`, a odgovarajuće vrednosti ključeva čine skupovi brojeva tipa `TreeSet<Integer>`. Indeks knjige je dakle mapa sledećeg tipa:

```
TreeMap<String,TreeSet<Integer>>
```

Formiranje indeksa knjige u programu počinje od prazne mape koja predstavlja indeks. Zatim se redom čita tekst knjige i usput se ovoj mapi dodaje svaki važniji izraz i aktuelni broj strane knjige na kojoj se izraz ot-

krije. Na kraju, nakon pregledanja cele knjige, sadržaj mape se prikazuje ili štampa ili upisuje u datoteku. Razmotrimo kako se svaki od ova tri koraka može realizovati u programu ukoliko zanemarimo pitanje otkrivanja važnijih izraza koji se dodaju u indeks. Za taj deo obrade teksta knjige se obično autoru ostavlja da u samom tekstu knjige na specijalan način ručno označi sve izraze koji se trebaju naći u indeksu, pa se onda njihovo izdvajanje u programu svodi na prosto otkrivanje specijalne oznake uz važniji izraz.

Konstruisanje prazne mape koja predstavlja indeks na početku nije ništa drugo nego konstruisanje objekta `indeks` relativno složenijeg tipa koji smo prethodno objasnili:

```
TreeMap<String,TreeSet<Integer>> indeks;
indeks = new TreeMap<String,TreeSet<Integer>>();
```

Pretpostavimo da se zatim tokom skeniranja teksta knjige u programu na nekoj strani otkrije važniji izraz koji treba dodati indeksu. Ako je izraz predstavljen promenljivom `izraz` tipa `String` i broj strane promenljivom `ref` tipa `int`, onda metod kojim se u indeks dodaje izraz i jedna njegova referenca ima sledeći oblik:

```
public void dodajIndeks(String izraz, int ref) {

   TreeSet<Integer> skupRef;  // skup referenci za dati izraz

   skupRef = indeks.get(izraz);

   if (skupRef == null) {  // prva referenca za dati izraz
      TreeSet<Integer> prvaRef = new TreeSet<Integer>();
      prvaRef.add(ref);
      indeks.put(izraz,prvaRef);
   }
   else
      skupRef.add(ref);
}
```

U ovom metodu se najpre pozivom `index.get(izraz)` ispituje da li se u do sada formiranom indeksu nalazi dati izraz. Rezultat ovog poziva je vrednost `null` ili neprazan skup referenci koje su do tada pronađene za dati izraz. U prvom slučaju se dati izraz ne nalazi u indeksu, odnosno radi se o

prvoj referenci za dati izraz, pa se dati izraz i novi skup od jedne reference dodaju u indeks. U drugom slučaju se dobija postojeći skup referenci za dati izraz, pa se nova referenca samo dodaje tom skupu.

Na kraju, nakon formiranja kompletnog indeksa, njegov sadržaj treba prikazati (odštampati). Za taj zadatak treba azbučnim redom prikazati svaki ključ i u nastavku njegov rezultujući skup referenci u rastućem redosledu. Ovo se može postići dvema ugnježđenim `for-each` petljama. U spoljašnjoj petlji se redom prolazi kroz sve parove ključ/vrednost mape koja reprezentuje indeks i prikazuje ključ aktuelnog para. U unutrašnjoj petlji se za taj ključ prikazuje njegova vrednost, odnosno redom celi brojevi u skupu referenci u rastućem redosledu. Metod u kojem se primenjuje ovaj postupak za prikazivanje indeksa ima sledeći oblik:

```
public void prikažiIndeks() {

   // Prikazivanje listi referenci za sve izraze u indeksu
   for (Map.Entry<String,TreeSet<Integer>> par :
                                          indeks.entrySet()) {
      String izraz = par.getKey();
      TreeSet<Integer> skupRef = par.getValue();
      System.out.print(izraz);   // prikaži izraz
      for (int ref : skupRef) {  // i njegovu listu referenci
         System.out.print(" " + ref);
      }
      System.out.println();
   }
}
```

Obratite pažnju u ovom metodu na tip parova mape koja reprezentuje indeks:

```
Map.Entry<String,TreeSet<Integer>>
```

Ovaj naizled prilično komplikovan tip je zapravo lako protumačiti ukoliko se podsetimo da su u mapi tipa `Map<T,S>` njeni parovi tipa `Map.Entry<T,S>`. Prema tome, parametri tipa u `Map.Entry<String,TreeSet<Integer>>` jednostavno su preuzeti iz deklaracije promenljive `indeks`.

11.3 Definisanje generičkih klasa i metoda

Do sada smo upoznali kako se koriste postojeće generičke klase, interfejsi i metodi koji su deo Sistema kolekcija u Javi. Druga strana generičkog programiranja je pisanje novih generičkih klasa, interfejsa i metoda. Mada potreba za time dolazi do izražaja tek u složenijim softverskim primenama, ovaj drugi aspekt generičkog programiranja obezbeđuje pravljenje vrlo opštih programskih biblioteka koje se mogu višekratno koristiti.

Generička klasa je klasa sa jednim parametrom tipa, ili više njih. Na primer, definicija generičke klase za predstavljanje para koji sadrži dva objekta istog tipa je:

```
public class Par<T> {
   private T prvi;
   private T drugi;

    // Konstruktori
   public Par() {
      this.prvi = null;
      this.drugi = null;
   }
   public Par(T prvi, T drugi) {
      this.prvi = prvi;
      this.drugi = drugi;
   }

    // Get i set metodi
   public T getPrvi() {
      return prvi;
   }
   public T getDrugi() {
      return drugi;
   }

   public void setPrvi(T x) {
      prvi = x;
   }
```

```
    public void setDrugi(T x) {
       drugi = x;
    }
}
```

Generička klasa `Par` iza svog imena sadrži parametar tipa `T` u uglastim zagradama `< >`. Ovaj parametar tipa u definiciji klase koristi se na isti način kao i običan tip: može se upotrebiti za tipove polja i lokalnih promenljivih, kao i za tipove rezultata metoda.

Generička klasa se u programu koristi navođenjem konkretnog tipa umesto parametra tipa. Na primer, konkretna klasa `Par<String>` može se smatrati običnom klasom koja ima dva privatna polja tipa `String`, zatim dva konstruktora `Par<String>()` i `Par<String>(String,String)`, kao i metode:

```
String getPrvi()
String getDrugi()
void setPrvi(String)
void setDrugi(String)
```

Na sličan način mogu se zamisliti i druge konkretne klase koje se dobijaju od generičke klase `Par<T>`, recimo, `Par<Double>` ili `Par<Color>`. Može se dakle smatrati, mada ne potpuno tačno, da je generička klasa jedna matrica za pravljenje običnih klasa.

Radi ilustracije korišćenja generičke klase `Par<T>`, u nastavku je definisan i testiran metod kojim se istovremeno određuje najmanji i najveći element celobrojnog niza:

```
public class MinMax {

   public static void main(String[] args) {

      int[] a = {17, 5, 1, 4, 8, 23, 11, 5};
      Par<Integer> mm = minmax(a);
      System.out.println("min = " + mm.getPrvi());
      System.out.println("max = " + mm.getDrugi());
   }
```

```
   public static Par<Integer> minmax(int[] a) {

      if (a == null || a.length == 0)
         return null;

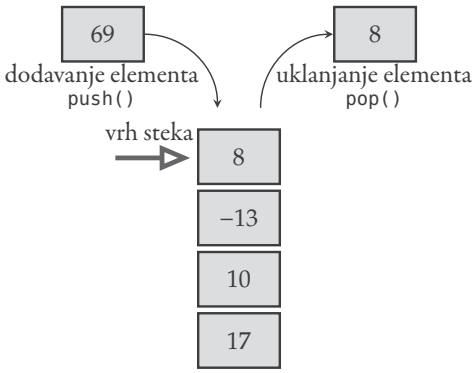
      int min = a[0], max = a[0];
      for (int i = 1; i < a.length; i++) {
         if (min > a[i])
            min = a[i];
         if (max < a[i])
            max = a[i];
      }
      return new Par(new Integer(min),new Integer(max));
   }
}
```

Da bismo još bolje razumeli koncept generičkih klasa, razmotrimo drugi primer strukture podataka koja se naziva *stek*. Stek se sastoji od niza elemenata „poređanih" u vertikalnom položaju tako da se elementi steka zamišljaju kao da su naslagani jedan na drugi od dna ka vrhu. Najbolji intuitivni model steka kao strukture podataka je grupa naslaganih poslužavnika u restoranu ili stog sena. Ono što karakteriše strukturu podataka steka je isto ono što karakteriše ove fizičke modele steka: samo vrh steka je (lako) pristupačan. Drugim rečima, novi element se steku dodaje samo na vrhu i iz steka se uklanja samo element koji se nalazi na vrhu. Operacija dodavanja novog elementa na vrh steka tradicionalno se naziva *push*, a operacija uklanjanja elementa sa vrha steka naziva se *pop*.

Stek na apstraktnom nivou može sadržati elemente bilo kog tipa. Ako su to, recimo, celi brojevi, onda su izgled jednog takvog steka i osnovne operacije nad njim ilustrovani na slici 11.2.

Realizacija steka pomoću povezane liste je očigledno lako izvodljiva ukoliko se zamisli da vrh steka odgovara početku liste. Na primer, stek brojeva definisan je sledećom klasom u kojoj je dodat metod za proveru da li je stek prazan, jer uklanjanje elementa iz praznog steka nije ispravno:

```
   public class StekBrojeva {
```



Slika 11.2: Osnovne operacije nad stekom.

```
    private LinkedList<Integer> elementi =
                                    new LinkedList<Integer>();

    public void push(Integer elem) {
       elementi.addFirst(item);
    }

    public Integer pop() {
       return elementi.removeFirst();
    }

    public boolean isEmpty() {
       return (elementi.size() == 0);
    }
}
```

Problem sa ovim pristupom je u tome što stek generalno može sadržati elemente bilo kog tipa. Ako nam treba stek čiji su elementi stringovi, realni brojevi, grafičke komponente ili koji su bilo kog drugog tipa, onda bismo morali da pišemo posebnu klasu za svaki tip elemenata steka. Pri tome, sve te klase bi bile skoro identične, osim što bi tip `Integer` elemenata steka brojeva bio zamenjen drugim tipom.

Da bi se izbeglo ovo ponavljanje programskog koda, i sve potencijalne

greške koje to donosi, može se definisati generička klasa `Stek` za predstavljanje steka bilo kog tipa elemenata. Način na koji se pišu generičke klase je jednostavan: konkretan tip `Integer` elemenata steka zamenjuje se parametrom tipa `T` i taj parametar tipa dodaje se imenu klase u zagradama `<T>`:

```
public class Stek<T> {

   private LinkedList<T> elementi = new LinkedList<T>();

   public void push(T elem) {
      elementi.addFirst(item);
   }

   public T pop() {
      return elementi.removeFirst();
   }

   public boolean isEmpty() {
      return (elementi.size() == 0);
   }
}
```

Obratite pažnju na to da se parametar tipa `T` unutar generičke klase `Stek` koristi potpuno isto kao obično ime tipa. Tako, ono se koristi kao tip rezultata metoda `pop()`, kao tip parametra `elem` metoda `push()` i čak kao konkretan tip elemenata povezane liste u `LinkedList<T>`. Generička klasa `Stek<T>` se u primenama može konkretizovati za predstavljanje steka čiji su elementi bilo kog tipa: `Stek<Integer>`, `Stek<String>`, `Stek<JButton>` i slično. Drugim rečima, generička klasa `Stek` koristi se na isti način kao postojeće generičke klase kao što su `LinkedList` i `HashSet`. Naredbom, na primer,

```
Stek<Double> s = new Stek<Double>();
```

konstruiše se stek realnih brojeva i definiše promenljiva `s` koja ukazuje na njega.

Mada je ime parametra tipa `T` uobičajeno u definiciji generičke klase, ono je potpuno proizvoljno i može se koristiti bilo koje ime. Ime parametra tipa u definiciji klase ima istu ulogu kao ime formalnog parametra

u definiciji metoda, odnosno parametar tipa se zamenjuje konkretnim tipom kada se generička klasa koristi za definisanje tipa promenljivih ili za konstruisanje objekata.

Generički interfejsi u Javi se definišu na sličan način. Slično se definišu i generičke klase i interfejsi koji imaju dva ili više parametra tipa. Tipičan primer je definicija generičke klase za predstavljanje para koji sadrži dva objekta potencijalno različitih tipova. Jednostavna verzija ove klase je:

```
public class Par<T,S> {
   private T prvi;
   private S drugi;

    // Konstruktor
   public Par(T prvi, S drugi) {
      this.prvi = prvi;
      this.drugi = drugi;
   }
   .
   . // Ostali metodi
   .
}
```

Ova klasa se zatim može koristiti za definisanje tipa promenljivih i za konstruisanje objekata, na primer:

```
Par<String,Color> boja =
                     new Par<String,Color>("Crvena", Color.RED);
Par<Double,Double> tačka =
                     new Par<Double,Double>(1.7,23.69);
```

Obratite pažnju na to da imenu konstruktora klase `Par<T,S>` u njegovoj definiciji nisu dodati parametri tipova. To je zato što je formalno ime klase `Par`, a ne `Par<T,S>`. Opšte pravilo je da se parametri tipova nikad ne dodaju imenima metoda ili konstruktora, nego samo imenima klasa i interfejsa.

Generički metodi

Pored generičkih klasa i interfejsa, u Javi se mogu pisati i generički metodi. Da bismo pokazali kako se to radi, razmotrimo problem pretrage niza

radi određivanja da li se data vrednost nalazi u datom nizu. Kako tip elemenata niza i tip date vrednosti nisu bitni, osim što moraju biti jednaki, prirodno se nameće generičko rešenje:

```
public static boolean nađi(T x, T[] a) {  // skoro ispravno
   for (T elem : a)
      if (x.equals(elem))
         return true;
   return false;
}
```

U ovom metodu, tip elemenata niza i tip tražene vrednosti označeni su parametrom tipa `T`. Kako prilikom poziva ovog metoda svaki konkretni tip koji zamenjuje `T` sadrži (nasleđeni) metod `equals()`, generički metod `nađi()` je logički ispravan. Međutim, ovako kako je napisan neće proći sintaksnu kontrolu, jer će Java prevodilac podrazumevati da je `T` neki konkretan tip čija definicija nije navedena. Zbog toga se mora na neki način ukazati da je `T` parametar tipa, a ne konkretan tip. Pošto je to upravo svrha dodavanja oznake `<T>` iza imena klase u definiciji generičke klase, slično se postupa i za generičke metode. Kod generičkih metoda, oznaka `<T>` navodi se u zaglavlju metoda iza svih modifikatora i ispred tipa rezultata metoda:

```
public static <T> boolean nađi(T x, T[] a) {  // ispravno
   for (T elem : a)
      if (x.equals(elem))
         return true;
   return false;
}
```

Ovaj metod se može koristiti za različite konkretne tipove nizova, na primer:

```
String[] jezici = {"Fotran", "Cobol", "Pascal", "C", "Ada"};
if (nađi("Java",jezici)) . . .
```

Ili, ako je `brojevi` niz tipa `Integer[]`, onda se pozivom

```
nađi(17,brojevi)
```

određuje da li se vrednost 17 nalazi među elementima niza `brojevi`. U ovom slučaju se koristi autopakovanje, odnosno 17 se automatski konvertuje u vrednost tipa `Integer`, jer se u Javi generičko programiranje primenjuje

samo za objekte.

Obratite pažnju na to da generički metodi mogu da se definišu kako u generičkim klasama tako i u običnim klasama. Primetimo i da u prethodnim primerima nije naveden konkretni tip koji zamenjuje parametar tipa u pozivu metoda, jer prevodilac ima dovoljno informacija da sam o tome ispravno zaključi. U vrlo retkim situacija kada to nije moguće, konkretni tip se u pozivu metoda piše neposredno ispred imena metoda, na primer:

```
if (<String>nađi("Java",jezici)) . . .
```

Metod `nađi()` se može koristiti za nizove bilo kog tipa. Ali vrlo sličan metod može se napisati i za nalaženje objekta u bilo kojoj kolekciji:

```
public static <T> boolean nađi(T x, Collection<T> k) {
   for (T elem : k)
      if (x.equals(elem))
         return true;
   return false;
}
```

Pošto je `Collection<T>` generički tip, ovaj metod je vrlo generalan. Zaista, može se koristiti za dinamički niz tipa `ArrayList` čiji su elementi tipa `Date`, skup tipa `TreeSet` čiji su elementi tipa `String`, listu tipa `LinkedList` čiji su elementi tipa `JButton` i tako dalje.

11.4 Ograničenja za parametar tipa

Parametar tipa u dosadašnjim primerima generičkog programiranja može se zameniti bilo kojim konkretnim tipom. To s jedne strane doprinosi opštosti napisanih programskih celina, ali s druge strane dovodi i do ograničenja jer se mogu koristiti samo one mogućnosti koje poseduju svi objekti. Na primer, sa onim što znamo do sada ne možemo napisati generički metod u kojem se objekti upoređuju metodom `compareTo()`, jer taj metod nije definisan za objekte svih konkretnih klasa koje potencijalno mogu zameniti parametar tipa generičkog metoda. Kod definisanja generičkih klasa potrebno je dakle na neki način ograničiti mogućnost zamene formalnog parametra tipa proizvoljnim aktuelnim tipom.

Generičko programiranje u Javi omogućava postavljanje ograničenja za parametre tipova na dva načina. Jedan način služi za definisanje *ograničenih tipova*, dok drugi uvodi takozvane *džoker-tipove*.

Ograničeni tipovi

Da bismo bolje razumeli ograničene tipove, pokušajmo najpre da napišemo generički metod kojim se određuje najmanji element niza:

```
public static <T> T min(T[] a) {  // skoro ispravno
   if (a == null || a.length == 0)
      return null;
   T minElem = a[0];
   for (int i = 1; i < a.length; i++)
      if (minElem.compareTo(a[i]) > 0)
         minElem = a[i];
   return minElem;
}
```

Problem s ovim generičkim metodom je to što je u njemu promenljiva `minElem` tipa `T`, odnosno može sadržati referencu na objekat bilo koje konkretne klase koja zamenjuje `T` u pozivu generičkog metoda. Ali kako možemo znati da takva klasa sadrži metod `compareTo()` kojim se `minElem` dalje upoređuje sa svakim elementom niza? Pošto je metod `compareTo()` definisan u interfejsu `Comparable`, potrebno je dakle da se paramatar tipa ograniči tako da se može zameniti samo klasama koje implementiraju interfejs `Comparable`. Način na koji se ovo postiže u Javi je navođenjem ograničenja za paramatar tipa u zaglavlju metoda:

```
public static <T extends Comparable> T min(T[] a)  // ispravno
```

Sada se generički metod `min()` može pozivati samo za nizove čiji elementi pripadaju klasama koje implementiraju interfejs `Comparable`.[1] To su, recimo, klase `Integer`, `String`, `Date` i tako dalje, ali ne i, recimo, `JButton`. Obratite pažnju na to da se ovde za ograničenje tipa koristi službena reč `extends`, a ne `implements`, iako se radi o interfejsu `Comparable`. To je opšte

[1] Sliku dodatno komplikuje činjenica da je `Comparable` i sam jedan generički interfejs, ali to je ovde manje važno.

pravilo, jer se time izražava uslov da parametar tipa treba da bude podtip nekog tipa.

Ova vrsta ograničenja za parametar tipa naziva se *ograničeni tip*. Ograničeni tipovi se mogu koristiti umesto formalnog parametra tipa ne samo za definisanje generičkih metoda kao u prethodnom primeru, nego i za definisanje generičkih klasa ili interfejsa.

Radi ilustracije, uzmimo da grupu grafičkih komponenti istog tipa u programu treba predstaviti jednom generičkom klasom `GrupaKomponenti`. Tako bi, recimo, klasa `GrupaKomponenti<JButton>` predstavljala grupu dugmadi, dok bi klasa `GrupaKomponenti<JPanel>` predstavljala grupu panela. Generička klasa treba dodatno da sadrži metode kojima bi se određene operacije primenljivale na sve komponente u grupi. Na primer, metod za iscrtavanje svih komponenti mogao bi imati ovaj oblik:

```
public void nacrtajSve() {
   .
   . // Pozivanje metoda repaint()
   . // za svaku komponentu u grupi
   .
}
```

Definicija parametrizovane klase u obliku `GrupaKomponenti<T>` nije dobra zbog sličnog problema koji smo sreli u prvoj verziji metoda `min()` u prethodnom primeru. Naime, metod `repaint()` je definisan za sve objekte tipa `JComponent`, ali ne i za objekte proizvoljnog tipa. Nema dakle smisla dozvoliti klase kao što su `GrupaKomponenti<String>` ili `GrupaKomponenti<Integer>`, jer klase `String` i `Integer` nemaju metod `repaint()`.

Potrebno je, drugim rečima, ograničiti formalni parametar tipa u definiciji parametrizovane klase `GrupaKomponenti<T>` tako da se `T` može zameniti samo klasom `JComponent` ili nekom njenom naslednicom. Rešenje se sastoji u pisanju ograničenog tipa „`T extends JComponent`" umesto prostog tipa `T` u definiciji klase:

```
public class GrupaKomponenti<T extends JComponent> {

   private ArrayList<T> komponente;  // niz komponenti u grupi

   public void nacrtajSve() {
```

```
        for (JComponent k : komponente)
            if (k != null)
                k.repaint();
    }

    // Dodavanje nove komponente u grupu
    public void dodaj(T k) {
        komponente.add(k);
    }
    .
    . // Ostali metodi
    .
}
```

Zapis „`T extends OsnovniTip`" u opštem slučaju označava neki tip `T` koji je jednak tipu `OsnovniTip` ili je podtip od tipa `OsnovniTip`. Posledica toga je da svaki objekat tipa `T` jesto ujedno i objekat tipa `OsnovniTip`, pa se sve operacije koje su definisane za objekte tipa `OsnovniTip` mogu primeniti i na objekte tipa `T`. Tip `OsnovniTip` ne mora biti ime klase, već može biti sve što predstavlja neki stvarni tip. To može biti i, recimo, neki interfejs ili čak neki parametrizovani tip.

Džoker-tipovi

Druga vrsta ograničenja za parametre tipova kod generičkog programiranja u Javi jesu takozvani *džoker-tipovi*. Džoker-tip se ne može koristiti kao formalni parametar tipa klase ili interfejsa, ali se može koristiti za tipove promenljivih ili, mnogo češće, za parametre u listi parametara generičkih metoda.

Pre nego što možemo razumeti džoker-tipove, moramo poznavati još jedan detalj u vezi sa nasleđivanjem za generičke tipove. Posmatrajmo jednu klasu i neku njenu naslednicu — radi određenosti, uzmimo klasu `Službenik` i njenu podklasu `Šef`. U kontekstu generičkih tipova može se postaviti pitanje da li se ovaj njihov odnos prenosi na konkretne klase koje se dobijaju zamenom parametrizovanih tipova? Na primer, da li je `Par<Šef>` podklasa od `Par<Službenik>`? Odgovor je negativan, iako se to možda ko-

si sa intuicijom. Naime, opšte pravilo kod generičkih klasa je da ne postoji nikakva veza između `GenKlasa<A>` i `GenKlasa<B>` bez obzira na to u kakvoj su vezi konkretne klase `A` i `B`.[2]

Ovo pravilo u nekim slučajevima ima negativan efekat na prednosti generičkog programiranja. Na primer, metod za prikazivanje para službenika može se napisati otprilike na ovaj način:

```
public static void prikažiKolege(Par<Službenik> p) {

   Službenik prvi = p.getPrvi();
   Službenik drugi = p.getDrugi();
   .
   . // Prikazivanje imena prvog i drugog službenika
   .
}
```

Ali ovaj metod se ne može koristiti za prikazivanje para šefova, iako su šefovi takođe službenici:

```
Par<Šef> dvašefa = new Par<Šef>(direktor,načelnik);
prikažiKolege(dvašefa); // GREŠKA!
```

Naime, konkretne klase `Par<Službenik>` i `Par<Šef>` ne stoje ni u kakvoj međusobnoj vezi, iako su instance iste generičke klase `Par<T>`, a klasa `Šef` je podklasa od `Službenik`. Zato se promenljiva `dvašefa` tipa `Par<Šef>` ne može koristiti kao argument u pozivu metoda `prikažiKolege()` čiji je parametar tipa `Par<Službenik>`. Rešenje za ovo prilično neprirodno ograničenje su džoker-tipovi:

```
public static void prikažiKolege(Par<? extends Službenik> p) {

   Službenik prvi = p.getPrvi();
   Službenik drugi = p.getDrugi();
   .
   . // Prikazivanje imena prvog i drugog službenika
   .
}
```

[2] Ovo pravilo je neophodno zbog konzistentnosti sistema tipova u Javi.

Zapis „`? extends Službenik`" u tipu parametra ovog metoda označava svaki tip koji je jednak klasi `Službenik` ili je podklasa od klase `Službenik`. Zbog toga džoker-tip

```
Par<? extends Službenik>
```

obuhvata klase `Par<Službenik>` i `Par<Šef>`, ali ne i recimo `Par<String>`. To znači da, pošto je parametar metoda `prikažiKolege()` upravo ovog džoker-tipa, argument u pozivu metoda `prikažiKolege()` može biti bilo kog od dva tipa `Par<Službenik>` ili `Par<Šef>`.

Kao drugi primer, razmotrimo pitanje dodavanja steku svih objekata koji se nalaze u datoj kolekciji. Preciznije, ako je `s` stek tipa `Stek<T>` i `k` kolekcija tipa `Collection<T>`, onda metodom `s.addAll(k)` treba dodati sve objekte iz `k` na `s`. Objekti kolekcije `k` su istog tipa `T` kao objekti steka, mada ako bolje razmislimo mogu biti opštiji. Naime, ako je `S` podklasa od `T`, onda kolekcija `k` može biti tipa `Collection<S>`. To ima smisla, jer je svaki objekat tipa `S` automatski i tipa `T`, pa se zato može dodati steku `s`.

Povećanje opštosti metoda `addAll()` može se postići primenom džoker-tipa za parametar tog metoda. Nova verzija generičke klase `Stek` koja se definisana ranije u ovom odeljku ima ovaj oblik:

```
public class Stek<T> {

   private LinkedList<T> elementi = new LinkedList<T>();

   public void push(T elem) {
      elementi.addFirst(item);
   }

   public T pop() {
      return elementi.removeFirst();
   }

   public boolean isEmpty() {
      return (elementi.size() == 0);
   }

   public void addAll(Collection<? extends T> k) {
```

```
        // Dodavanje svih elemenata kolekcije na vrh steka
        for (T elem : k)
            push(elem);
    }
}
```

Obratite pažnju na to da se ovde džoker-tipovi kombinuju sa generičkim klasama. U prethodnoj definiciji generičke klase `Stek<T>`, parametar `T` ima ulogu specifičnog, mada nepoznatog imena tipa. Unutar te klase džoker-tip „`? extends T`" označava naravno `T` ili neki izvedeni tip od `T`. Kada se konstruiše konkretni stek tipa recimo `Stek<Službenik>`, onda se tip `T` zamenjuje sa `Službenik` i džoker-tip „`? extends T`" u definiciji metoda `addAll()` postaje „`? extends Službenik`". To znači da se ovaj metod može primeniti kako za kolekciju objekata tipa `Službenik`, tako i za kolekciju objekata tipa `Šef`.

U definiciji metoda `addAll()` se koristi `for-each` petlja za postupno dodavanje elemenata kolekcije na vrh steka. Moguću sumnju u ispravnost ove petlje izaziva to što se pojedinačnim elementima kolekcije redom pristupa preko promenljive `elem` tipa `T`, a elementi kolekcije mogu biti tipa `S`, gde je `S` podklasa od `T`. Ali to prema principu podtipa u Javi nije greška, jer se objekti tipa `S` mogu dodeliti promenljivoj tipa `T` ukoliko je `S` podklasa od `T`.

U zapisu oblika „`? extends T`" umesto `T` može stajati interfejs, a ne isključivo klasa. Primetimo da se koristi reč `extends`, a ne `implements`, čak i ukoliko je `T` interfejs. Radi ilustracije, podsetimo se da je `Runnable` interfejs koji predstavlja zadatak koji se može izvršavati unutar posebne niti paralelno sa drugim zadacima. Jedan generički metod za paralelno izvršavanje svih zadataka tipa `Runnable` u datoj kolekciji zadataka je:

```
public void izvršiParalelno(
               Collection<? extends Runnable> zadaci) {

    for (Runnable z : zadaci) {
        Thread nit = new Thread(z);  // posebna niti za svaki
                                     // ... zadatak u kolekciji
        nit.start();                 // pokreni izvršavanje niti
    }
```

```
}
```

Metod `addAll()` u generičkoj klasi `Stek<T>` dodaje sve objekte iz neke kolekcije na stek. Pretpostavimo da je potrebno napisati metod kojim se obavlja suprotna operacija: sve objekte iz steka treba dodati datoj kolekciji. Rešenje koje odmah pada na um je metod čije je zaglavlje:

```
public void addAllTo(Collection<T> k)
```

Ovaj metod međutim nije dovoljno generalan, jer se može primeniti samo za kolekcije čiji su elementi istog tipa `T` kao i elementi steka. Opštije rešenje bi bio metod kojim se elementi steka mogu dodati kolekciji čiji su elementi nekog tipa `S` koji je nadtip od `T`.

Ovaj odnos tipova izražava se drugom vrstom džoker-tipa „`? super T`" koja označava bilo `T` ili neku nadklasu od `T`. Na primer,

```
Collection<? super Šef>
```

obuhvata tipove `Collection<Šef>` i `Collection<Službenik>`.

Koristeći ovu vrstu džoker-tipa za parametar metoda `addAllTo()`, kompletna generička klasa za strukturu podataka steka ima ovaj oblik:

```
public class Stek<T> {

   private LinkedList<T> elementi = new LinkedList<T>();

   public void push(T elem) {
      elementi.addFirst(item);
   }

   public T pop() {
      return elementi.removeFirst();
   }

   public boolean isEmpty() {
      return (elementi.size() == 0);
   }

   public void addAll(Collection<? extends T> k) {
      // Dodavanje svih elemenata kolekcije na vrh steka
```

```
        for (T elem : k)
            push(elem);
    }

    public void addAllTo(Collection<? super T> k) {
        // Uklanjanje elemenata sa steka i dodavanje kolekciji
        while (!isEmpty()) {
            T elem = pop();
            k.add(elem);
        }
    }
}
```

Napomenimo još da se može koristiti i treća vrsta džoker-tipa u obliku „<?>" kojim se označava nepoznat tip i u suštini to zamenjuje zapis „<? extends Object>". Na primer, generički metod za prikazivanje bilo koje kolekcije objekata ne bi bio dobro napisan u ovom obliku:

```
public void prikažiKolekciju(Collection<Object> k) {
    for (Object elem : k)
        System.out.println(elem);
}
```

Problem je to što tip parametra Collection<Object> ovog metoda nije ni u kakvoj vezi sa, recimo, tipom Collection<Integer>. To znači da se ovaj metod ne može koristiti za prikazivanje kolekcije celih brojeva, a ni za prikazivanje kolekcije objekata bilo kog drugog tipa različitog od baš tipa Object. Mnogo bolje rešenje se dobija primenom neograničenog džoker-tipa:

```
public void prikažiKolekciju(Collection<?> k) {
    for (Object elem : k)
        System.out.println(elem);
}
```

Zaključimo na kraju da se ograničeni tipovi i džoker-tipovi, iako su povezani u određenom smislu, koriste na vrlo različite načine. Ograničeni tip se može koristiti samo kao formalni parametar tipa u definiciji generičkog metoda, klase ili interfejsa. Džoker-tip se najčešće koristi za određiva-

nje tipova formalnih parametara u definiciji generičkih metoda i ne može se koristiti kao formalni parametar tipa. Još jedna sintaksna razlika je to što se kod ograničenih tipova, za razliku od džoker-tipova, uvek koristi reč `extends`, nikad `super`.

Primer: generičko sortiranje liste

Na strani 405 je pokazan jedan efikasan postupak (*merge-sort*) za sortiranje liste elemenata. U stvari, elementi liste su bili celobrojnog tipa, mada pažljivija analiza tog postupka otkriva da se od celobrojnog tipa jedino koristi mogućnost poređenja vrednosti celobrojnog tipa. Elementi liste koja se sortira mogu zato biti i realni brojevi i stringovi i, najopštije, bilo koji objekti za koje je definisan metod `compareTo()`. Drugim rečima, elementi liste mogu biti objekti bilo koje klase koja implementira interfejs `Comparable`.

Opšti metod za sortiranje liste treba dakle samo da sprovede ovo ograničenje za tip elemenata liste, a sve ostale izmene se praktično sastoje od pisanja parametra tipa umesto konkretnog tipa `Integer`. Ali ukoliko ovo pokušamo da rešimo koristeći džoker-tip za elemente liste, dobijamo generički metod čiji je početak:

```
public static void mergeSort(List<? extends Comparable> l) {

   if (l.size() > 1) {  // lista l ima bar dva elementa

      // 1. Podeliti listu l u dve polovine l1 i l2
      List<***> l1 = new LinkedList<***>(); // tip elemenata l1?
      List<***> l2 = new LinkedList<***>(); // tip elemenata l2?
      .
      .
      .
   }
}
```

Ovo pokazuje da smo na pogrešnom putu, jer nemamo konkretno ime za nepoznat tip elemenata liste označen zapisom „`? extends Comparable`". A ime tog tipa je neophodno radi konstruisanja listi `l1` i `l2` čiji elementi moraju biti istog tipa kao i elementi liste koja se sortira. U ovom slučaju

dakle moramo koristiti ograničen tip za elemente liste, jer se onda dobija konkretno ime nepoznatog tipa koje se može koristiti u telu metoda. Ovo rešenje je primenjeno u generičkom metodu za sortiranje liste (*merge-sort*) koji je u nastavku prikazan u celosti.

```
public static <T extends Comparable<T>>
              void mergeSort(List<T> l) {

   if (l.size() > 1) {  // lista l ima bar dva elementa

      // 1. Podeliti listu l u dve polovine l1 i l2
      List<T> l1 = new LinkedList<T>();
      List<T> l2 = new LinkedList<T>();

      ListIterator<T> it = l.listIterator();
      boolean dodajL1 = true;  // element liste l ide u l1 ili l2

      while (it.hasNext()) {
         T e = it.next();
         if (dodajL1)
            l1.add(e);
         else
            l2.add(e);
         dodajL1 = !dodajL1;
      }

      // 2. Rekurzivno sortirati liste l1 i l2
      mergeSort(l1);
      mergeSort(l2);

      // 3. Objediniti sortirane liste l1 i l2 u sortiranu listu l
      l.clear();

      ListIterator<T> it1 = l1.listIterator();
      ListIterator<T> it2 = l2.listIterator();

      T e1 = it1.next();  // l1 ima bar jedan element
      T e2 = it2.next();  // l2 ima bar jedan element
```

```
        while (true) {
           if (e1.compareTo(e2) <= 0) {
              l.add(e1);
              if (it1.hasNext())
                 e1 = it1.next();
              else {
                 l.add(e2);
                 while (it2.hasNext())
                    l.add(it2.next());
                 break;
              }
           }
           else {
              l.add(e2);
              if (it2.hasNext())
                 e2 = it2.next();
              else {
                 l.add(e1);
                 while (it1.hasNext())
                    l.add(it1.next());
                 break;
              }
           }
        }
     }
  }
```

Obratite pažnju na to da je parametrizovan ograničen tip prethodnog generičkog metoda naveden u obliku „`T extends Comparable<T>`", a ne u prostom obliku „`T extends Comparable`". Razlog je to što objekti liste tipa `T` treba zapravo da implementiraju parametrizovan interfejs `Comparable<T>`. Primetimo i da je ovo primer zapisa „`T extends OsnovniTip`" za ograničeni tip u kojem je `OsnovniTip` jedan parametrizovan tip, a ne obična klasa.

Literatura

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Prentice Hall, fourth edition, 2006.

[2] Harvey Deitel and Paul Deitel. *Java How to Program*. Prentice Hall, seventh edition, 2007.

[3] David J. Eck. *Introduction to Programming Using Java*. Free version at http://math.hws.edu/javanotes, fifth edition, 2006.

[4] David Flanagan. *Java in a Nutshell*. O'Reilly, fifth edition, 2005.

[5] Mark Guzdial and Barbara Ericson. *Introduction to Computing and Programming in Java: A Multimedia Approach*. Prentice Hall, 2007.

[6] Cay S. Horstmann and Gary Cornell. *Core Java, Volume I–Fundamentals*. Prentice Hall PTR, eighth edition, 2007.

[7] Cay S. Horstmann and Gary Cornell. *Core Java, Volume II–Advanced Features*. Prentice Hall PTR, eighth edition, 2008.

[8] Ivor Horton. *Beginning Java 2*. Wiley Publishing, JDK 5 edition, 2005.

[9] Jonathan Knudsen and Patrick Niemeyer. *Learning Java*. O'Reilly, third edition, 2005.

[10] Daniel Liang. *Introduction to Java Programming, Comprehensive Version*. Prentice Hall, seventh edition, 2008.

[11] Richard F. Raposa. *Java in 60 Minutes a Day*. Wiley Publishing, 2003.

[12] Matthew Robinson and Pavel Vorobiev. *Java Swing*. Manning, second edition, 2003.

[13] Herbert Schildt. *Java Programming Cookbook*. McGraw-Hill, 2007.

[14] Dejan Živković. *Osnove Java programiranja*. Univerzitet Singidunum, 2012.

[15] Dejan Živković. *Osnove Java programiranja – Zbirka pitanja i zadataka sa rešenjima*. Univerzitet Singidunum, 2010.

Indeks

A

ActionEvent, 233
ActionListener, 235
adapterske klase, 242
adresa petlje, 351
    localhost, 351
aktuelni (radni) direktorijum, 296
anonimne klase, 191
aplet, 228
apstraktne klase, 173
apstraktni metodi, 176
argumenti metoda, 36
ArrayList, 126
ArrayList<T>, 128, 403
Arrays, 105
asinhrona komunikacija programa, 377
asocijativnost operatora, 14

B

beskonačna petlja, 28
biblioteka AWT, 196
biblioteka Swing, 197
binarni U/I tokovi, 277
    InputStream, 280
    OutputStream, 280
blok naredbi, 19
boje, 223
break naredba, 34

C

Collection<T>, 392, 393
    List<T>, 392
    Set<T>, 392
Color, 224
Comparable, 179
Comparable<T>, 396
Comparator<T>, 409
Component, 204
Container, 204
continue naredba, 34
curenje memorije, 83, 252

D

DataInputStream, 282
DataOutputStream, 282
datoteka, 289
    File, 296
    ime, 295
    JFileChooser, 300
    sadržaj, 295
definicija metoda, 36
definisanje grafičkih komponenti, 216
definisanje klasa izuzetaka, 270
definisanje metoda
    specifikatori pristupa, 40
dinamički nizovi, 122, 125
dinamičko (kasno) vezivanje, 162
direktorijum, 289
    aktuelni (radni), 296
DNS server, 351
do-while naredba, 30
dokumentacija Jave, 58
domensko ime, 350
dvodimenzionalni nizovi, 112
džoker-tipovi, 433

E
eksplicitna konverzija tipa, 16
enkapsulacija (učaurivanje), 83
EventObject, 232

F
File, 296, 298
FileInputStream, 290
FileOutputStream, 290
final
  za klase, 154
  za konstante, 12
  za metode, 154
finaly klauzula, 260
folder, 289
fontovi, 226
  Font, 227
for naredba, 32

G
generička klasa, 423
  parametar tipa, 423
generički metod, 428
  parametar tipa, 429
generičko programiranje, 389
geteri, 85
grafički elementi, 199
  dijalozi, 200
  komponente, 200
  kontejneri, 200
  okviri, 199
  prozori, 199
grafički korisnički interfejs, 195
grafički programi, 195
Graphics, 217
Graphics2D, 220

H
HashMap<T,S>, 413
HashSet<T>, 408
heš tabela, 410
hijerarhija klasa, 142
hip-memorija, 68

I
ime datoteke, 295
  apsolutno, 296
  File, 296
  JFileChooser, 300
  relativno, 296
indeks knjige, 419
InputStream, 280
  DataInputStream, 282
  FileInputStream, 290
  ObjectInputStream, 288
InputStreamReader, 282
interfejs, 178
  implementacija, 179
  osobine, 181
Internet Protocol (IP), 350
  IP adresa, 350
IP adresa, 350
  adresa petlje, 351
  domensko ime, 350
    DNS server, 351
iterator, 398
Iterator<T>, 399
izraz, 12
  konverzija tipa, 14
  operatori, 13
    asocijativnost, 14
    prioritet, 13
izuzetak, 254
  definisanje klase, 270
  klauzula throws, 268
  mehanizam postupanja, 264
  naredba throw, 262
  neproveravani, 267
  proveravani, 267
  rukovalac, 258
  rukovanje, 254
    hvatanje i obrađivanje, 254
    obavezno i neobavezno, 267

`Throwable`, 254
`try-catch` naredba, 257
izvor događaja, 233

J

Java bajtkod, 2
Java virtuelna mašina, 2
`JButton`, 205
`JCheckBox`, 205
`JComboBox`, 205
`JComponent`, 204
`JEditorPane`, 356
`JFileChooser`, 300
`JFrame`, 200
naslеđeni metodi, 203
`JLabel`, 205
`JOptionPane`, 198, 246
`JPanel`, 204
`JProgressBar`, 337
`JRadioButton`, 205
`JTextField`, 205

K

katanac, 333
klasa
adapterska, 242
anonimna, 191
apstraktna, 173
generička, 423
omotač primitivnog tipa, 397
polja, 50
članovi, 51, 63
nestatički (objektni), 63
statički (klasni), 63
klase za crtanje, 220
klasni dijagram, 142
klijent, 360
klijent-server programiranje, 359
klijentski soket, 360
kolekcija, 392
iterator, 398
jednakost i poređenje objekata, 394
lista, 392
pogled na, 415
skup, 392
komentari, 5
dokumentacioni, 6
kompajleri, 2
komponente, 203
raspored unutar kontejnera, 208
konkretne klase, 176
konstante, 12
konstruktor, 76
u klasama naslednicama, 155
kontejneri, 203
raspored komponenti, 208
konverzija tipa, 14
kooperacija niti, 343
koordinate grafičkih elemenata, 207
kritična oblast, 332

L

`LayoutManager`, 208
`LinkedList<T>`, 403
`List<T>`, 392
lista, 392, 401
`ArrayList<T>`, 403
`LinkedList<T>`, 403
`List<T>`, 392
localhost, 351
lokalne klase, 189

M

`Map.Entry<T,S>`, 416
`Map<T,S>`, 392, 411, 413
`HashMap<T,S>`, 413
`Map.Entry<T,S>`, 416
`TreeMap<T,S>`, 413
mapa, 411
pogled na, 415
merge sort, 406, 440
metod, 36

apstraktni, 176
argumenti, 36
definicija, 36
klauzula `throws`, 268
generički, 428
geter, 85
konstruktor, 76
nadjačavanje, 151
parametri, 36
potpis, 45
pozivanje, 36
preopterećeni, 45
promenljiv broj argumenata, 110
rekurzivni, 47
rezultat, 36
seter (mutator), 85
sinhronizovan, 330
vraćanje rezultata, 42
mime format, 354
`MouseAdapter`, 244
`MouseEvent`, 233
`MouseListener`, 236, 242
mrežno programiranje, 349
klijent-server, 351, 359
port, 360
soket, 351
veb programiranje, 352
višenitno, 371
mrtva petlja, 334
multitasking, 311
multithreading, 311

N

nabrojivi tipovi, 168
nadjačavanje metoda, 151
naredba deklaracije promenljive, 18
naredba dodele, 17
naredba povratka, 43
naredbe grananja
`if-else` naredba, 21
`switch` naredba, 26
ugnježđavanje, 22
naredbe ponavljanja
`do-while` naredba, 30
`for-each` petlja, 109
`for` naredba, 32
kontrolna promenljiva (brojač), 33
`while` naredba, 28
nasleđivanje klasa, 135
neproveravani izuzeci, 267
nit, *Vidi* nit izvršavanja
nit izvršavanja, 311
`JProgressBar`, 337
kooperacija, 343
multithreading, 311
sinhronizacija, 326
`Thread`, 313
za grafičke događaje, 335
zadatak, 311
`Runnable`, 313
niz, 93
bazni tip, 94
dužina, 93
dvodimenzionalni, 112
matrica (tabela), 113
elementi, 93
indeks, 93
inicijalizacija, 97
jednodimenzionalni, 93
`length`, 97
višedimenzionalni, 113
nizovi
dinamički, 122, 125

O

`Object`, 157
`ObjectInputStream`, 288
`ObjectOutputStream`, 288
objekat
instanca klase, 63
jednakost i poređenje, 394
`Comparable<T>`, 396
`Comparator<T>`, 409

konstrukcija i inicijalizacija, 74
`Object`, 157
objektne ugnježđene klase, 187
objektno orijentisano programiranje, 61
oblast važenja imena, 54
ograničeni tipovi, 431
okvir, 200
operator `new`, 68, 78
operatori, 13
`OutputStream`, 280
`DataOutputStream`, 282
`FileOutputStream`, 290
`ObjectOutputStream`, 288
`OutputStreamWriter`, 282
označena petlja, 35

P

paket, 58, 350
`import`, 59
`package`, 60
anonimni, 60
korišćenje, 58
parametar tipa, 423, 429
ograničenja, 430
džoker-tipovi, 433
ograničeni tipovi, 431
parametri metoda, 36
parametrizovani tipovi, 128, 389
sistem kolekcija, 390
petlja, 28
beskonačna, 28
označena, 35
telo petlje, 28
ugnježđavanje, 30
uslov nastavka (ili prekida), 28
piksel, 202, 207
podlista, 418
podskup, 418
pogledi na mape i kolekcije, 415
`Map.Entry<T,S>`, 416
podlista, 418
podskup, 418
`Point2D`, 223
polimorfizam, 161
polja, 19, 50
pop, 425
port, 360
potpis metoda, 45
pozivanje metoda, 36, 40
prenošenje argumenata po vrednosti, 41
prazna naredba, 25
prekoračenje bafera, 251
preopterećeni metodi, 45, 161
prevodioci, 2
princip podtipa, 127, 147
`PrintWriter`, 283
prioritet operatora, 13
program
asinhrona komunikacija, 377
grafički, 195
greške, 249
ispravan i robustan, 249
klijent, 360
server, 360
slobodan format, 7
ulaz i izlaz, 275
hijerarhija klasa, 280
vođen događajima, 196
programiranje
generičko, 389
mrežno, 349
objektno orijentisano, 61
programske greške, 249
curenje memorije, 252
izuzeci, 254
korišćenje pokazivača `null`, 251
nedeklarisane promenljive, 250
prekoračenje bafera, 251
promenljiva, 10
alociranje, 53
dealociranje, 53
dužina trajanja, 53
globalna, 51

zaklonjena, 55
lokalna, 19, 20, 51
oblast važenja, 20
protokol, 350
Internet Protocol (IP), 350
paketi, 350
Transmission Control Protocol (TCP), 350
User Datagram Protocol (UDP), 350
proveravani izuzeci, 267
push, 425

R

raspored komponenti, 208
BorderLayout, 214
FlowLayout, 209
GridLayout, 210
postupak za razmeštanje, 208
`Reader`, 280
`InputStreamReader`, 282
`Rectangle2D`, 222
referenca (pokazivač), 68
rekurzivni metodi, 47
bazni slučaj, 48
`return` naredba, 43
rezultat metoda, 36
rukovalac događaja, 196, 233
rukovalac izuzetka, 258
rukovanje događajima, 232
`Runnable`, 313

S

sakupljanje otpadaka, 82
`Scanner`, 285
server, 360
višenitni, 371
serverski soket, 360
`ServerSocket`, 361
`Set<T>`, 392, 408
seteri (mutatori), 85
sinhronizacija niti, 326
greška trke niti, 327
katanac, 333
mrtva petlja, 334
uzajamna isključivost, 330
sinhronizovani metodi i naredbe, 330
sistem kolekcija, 390
`Collection<T>`, 392
kolekcije i mape, 391
`Map<T,S>`, 392
skup, 392, 408
`HashSet<T>`, 408
`Set<T>`, 408
`TreeSet<T>`, 408
službene (rezervisane) reči, 11
`Socket`, 361
soket, 351
klijentski, 360
serverski, 360
`ServerSocket`, 361
`Socket`, 361
sortiranje objedinjavanjem, 406
standardne komponente, 205
`JButton`, 205
`JCheckBox`, 205
`JComboBox`, 205
`JLabel`, 205
`JRadioButton`, 205
`JTextField`, 205
stanje trke, 327
statičke ugnježđene klase, 186
statičko (rano) vezivanje, 161
stek, 425
pop, 425
push, 425
strukture podataka, 93
stek, 425
`super`, 150
pozivanje konstruktora nasleđene klase, 155
pristup zaklonjenim članovima nasleđene klase, 151

T
tekstualni U/I tokovi, 277
  Reader, 280
  Writer, 280
telo petlje, 28
this, 87
  implicitni argument metoda, 88
  pozivanje preopterećenog konstruktora, 90
Thread, 313, 318
throw naredba, 262
Throwable, 254
  Error, 255
  Exception, 255
throws klauzula, 268
tip podataka, 8
  klasni, 8
  omotači primitivnih tipova, 397
  parametrizovani, 128, 389
  primitivni, 8
    void, 9
    celobrojni, 8
    logički, 9
    realni, 8
    znakovni, 9
tok podataka, 276
  binarni i tekstualni, 277
  izlazni, 276
  ulazni, 276
Transmission Control Protocol (TCP), 350
TreeMap<T,S>, 413
TreeSet<T>, 408
try-catch naredba, 257
  klauzula finaly, 260

U
ugnježđene klase, 184
  lokalne i anonimne, 186
  statičke i objektne, 185
uklanjanje objekata, 81
Unicode, 9
univerzalni lokator resursa, 351
URL, 351
User Datagram Protocol (UDP), 350
uslov nastavka (ili prekida), 28
uzajamna isključivost, 330

V
veb programiranje, 352
viseći pokazivači, 82
višedimenzionalni nizovi, 113
višekratna upotreba, 79
višestruko nasleđivanje, 178

W
while naredba, 28
WindowAdapter, 244
WindowListener, 243
Writer, 280
  OutputStreamWriter, 282
  PrintWriter, 283

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд

004.438JAVA(075.8)
004.42:004.738.5(075.8)

ЖИВКОВИЋ, Дејан, 1956-
Java programiranje / Dejan Živković. - 5. izd. - Beograd : Univerzitet Singidunum, 2021 (Beograd : BiroGraf). - VI, 451 str. : ilustr. ; 24 cm

Tiraž 1.500. - Bibliografija: str. 443-444. - Registar.

ISBN 978-86-7912-521-7

a) Програмски језик "Java" b) Интернет - Програмирање

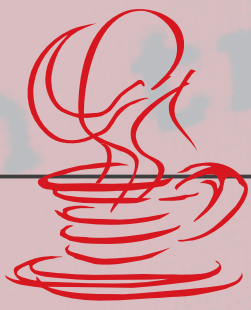
COBISS.SR-ID 30969097

© 2021.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.

www.singidunum.ac.rs

9 788679 125217



Dejan Živković

JAVA PROGRAMIRANJE

OBJEKTNO ORIJENTISANI KONCEPT PROGRAMIRANJA

Programski jezik Java je stekao veliku popularnost zbog svoje elegancije i savremene koncepcije. Programi napisani u Javi mogu se zato naći u mobilnim uređajima, Internet stranama, multimedijalnim servisima, distribuiranim informacionim sistemima i mnogim drugim okruženjima ljudskog delovanja. Cilj ove knjige je da programerima pomogne da što lakše nauče Javu kako bi u svakodnevnom radu mogli da koriste moćan i moderan alat.

Java je objektno orijentisan programski jezik, stoga se u prvom delu ove knjige govori o bitnim konceptima objektno orijentisanog programiranja, kao i o načinu na koji su ti koncepti realizovani u Javi. Java je i programski jezik opšte namene koji se koristi za rešavanje zadataka iz različitih oblasti primene. Programi napisani u Javi mogu biti obični tekstualni ili grafički programi, ali i složenije aplikacije zasnovane na principima mrežnog programiranja, višenitnog programiranja, generičkog programiranja i mnogih drugih posebnih tehnika.

Naglasak u ovoj knjizi je uglavnom na naprednijim mogućnostima Jave o kojima se govori u drugom delu knjige. Ali zbog toga se od čitalaca očekuje određeno predznanje o osnovnim konceptima programiranja. Specifično, pretpostavlja se da čitaoci dobro poznaju osnovne elemente programiranja kao što su promenljive i tipovi podataka. S tim u vezi se podrazumeva poznavanje osnovnih upravljačkih programskih konstrukcija u koje spadaju naredbe dodele vrednosti promenljivim, naredbe grananja i naredbe ponavljanja. Na kraju, neophodno je i poznavanje načina za grupisanje ovih prostih upravljačkih konstrukcija u složenije programske strukture, od kojih su najznačajniji potprogrami (obični i rekurzivni). Kratko rečeno, od čitalaca se očekuje da su ranije koristili neki programski jezik i da su solidno ovladali proceduralnim načinom programiranja.

Univerzitet Singidunum