Dejan Živković

OSNOVE JAVA PROGRAMIRANJA

UNIVERZITET SINGIDUNUM
FAKULTET ZA INFORMATIKU I MENADŽMENT

Beograd, 2009

OSNOVE JAVA PROGRAMIRANJA
drugo izdanje

Autor: Prof. dr Dejan Živković
Fakultet za informatiku i menadžment, Univerzitet Singidunum

Recenzent: Prof. dr Dragan Cvetković
Fakultet za informatiku i menadžment, Univerzitet Singidunum

Izdavač: UNIVERZITET SINGIDUNUM
FAKULTET ZA INFORMATIKU I MENADŽMENT
Danijelova 32, 11000 Beograd

Za izdavača: Prof. dr Milovan Stanišić

Tehnička obrada: Dejan Živković

Dizajn korica: Milan Nikolić

Godina izdanja: 2009.

Tiraž: 500 primeraka

Štampa: ČUGURA print, Beograd
www.cugura.rs

ISBN: 978-86-7912-189-9

Sadržaj

Spisak programa

Predgovor

Ova knjiga je namenjena čitaocima koji žele da nauče Java programiranje. Knjiga ne predstavlja uvod u objektno orijentisano programiranje, mada su objašnjeni neki osnovni koncepti te tehnologije kako bi se uspostavila zajednička terminologija. Obim i sadržaj knjige je prilagođen nastavnom planu odgovarajućeg predmeta na Fakultetu za informatiku i menadžment Univerziteta Singidunum u Beogradu.

Programski jezik Java je sam po sebi obiman i pokriva širok krug oblasti od standradnih tekstualnih i grafičkih programa do složenijih tehnika kao što su Internet programiranje, rad sa bazama podataka, višenitno programiranje i tako dalje. Pošto je knjiga zamišljena kao fakultetski udžbenik za prvo upoznavanje sa Javom, moralo se odustati od složenijih tema i svesti materijal na razumnu meru. Zbog toga je u knjizi samo „zagrebana površina" svih mogućnosti programskog jezika Java. Knjiga ipak pruža solidnu osnovu za dalje učenje Jave, ukoliko čitaoci žele da se profesionalno bave Java programiranjem. Informacije o dodatnim mogućnostima Jave se mogu pronaći u literaturi koja je navedena na kraju knjige.

Java ima mnoge jezičke osobine svojstvene većini programskih jezika. Java će izgledati poznato C i C++ programerima, jer su mnoge programske konstrukcije preuzete skoro bez izmene iz tih jezika. Zbog toga je predznanje čitalaca iz ovih jezika velika olakšica za učenje Jave. To međutim nije preduslov za uspešno savladavanje materijala ove knjige.

Glavni cilj pisanja ove knjige je predstavljanje osnova Java programiranja na najjasniji način. Zbog toga je u tekstu poštovano pedagoško načelo da izlaganje materijala svake oblasti treba da ide od poznatog ka nepoznatom i od konkretnog ka apstraktnom. Uz to, tekst je propraćen velikim brojem slika i urađenih primera kojima se konkretno ilustruju svi novouvedeni pojmovi. A kao što je to već uobičajeno za računarske knjige, za predstavljanje programskog teksta se koristi font sa slovima `iste širine`.

Zahvalnost. Izražavam najdublju zahvalnost svim osobama koji su mi pomogli u pripremi ove knjige. To se posebno odnosi na kolege koji su pročitali radne delove knjige i svojim sugestijama pomogli da konačna verzija bude bolja. Zahvalnost dugujem i studentima za uočene greške u prethodnom izdanju, a posebno Ivanu Proletu koji je s velikom pažnjom pročitao tekst.

Na kraju, bio bih veoma zahvalan i čitaocima na njihovom mišljenju o knjizi. Sve primedbe i pronađene greške se mogu poslati elektronskom poštom na adresu `dzivkovic@singidunum.ac.rs`.

Dejan Živković
Beograd, Srbija
jun 2009.

Glava

1

Uvod u Java programiranje

Sveprisutni uređaji koje nazivamo računarima imaju široku primenu u našem svakodnevnom životu. Iako postoji više vrsta računara opšte namene (super računari, stoni računari, prenosivi računari), kao i specijalnih računara ugrađenih u druge uređaje (mobilne telefone, automobile, avione, kosmičke letelice i tako dalje), svi računari rade na sličan način.

U ovom poglavlju ćemo se upoznati sa načinom rada računara i programa, kao i sa osnovama Java programiranja.

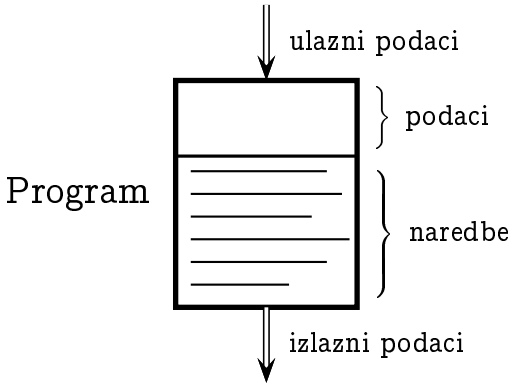
1.1 Računari i programi

Računari su elektronski uređaji koji se sastoje od ***hardvera*** i ***softvera***. Pod hardverom se podrazumevaju fizički delovi računara među kojima spadaju procesor, memorija, disk, tastatura, monitor i tako dalje. Softver obuhvata sve programe u računaru čijim izvršavanjem računar može obavljati neki koristan posao za ljude. Specijalni podskup tih programa, koji se naziva ***operativni sistem***, služi za upravljanje radom hardverskih delova kako bi računar funkcionisao kao celina.

Osnovni princip rada računara se sastoji dakle od izvršavanja ***programa*** čime se postiže odgovarajuća funkcionalnost. Program se sastoji od niza ***naredbi*** koje računar izvršava redom, jednu po jednu. Proces pisanja programa određene namene za računare se naziva ***programiranje***.

Dobra analogija za računarski program i računar koji ga izvršava jeste recept za neko jelo i kuvar koji sprema jelo po tom receptu. Recept se

sastoji od niza koraka (to su naredbe programa), a kuvar sprema jelo redom odrađujući svaki korak recepta (kao što računar redom izvršava naredbe programa). Na kraju se kao rezultat dobija određeno jelo, recimo pica (što odgovara određenoj funkcionalnosti programa, recimo pretraživanju Interneta). Primetimo da pravi kuvar može spremiti razna jela po različitim receptima, što odgovara računaru opšte namene koji može izvršiti različite programe (čak i više njih ,,istovremeno"). Kod specijalnih računara ugrađenih u druge uređaje je princip rada isti, osim što oni izvršavaju jedan jedini program (što bi odgovaralo priučenom kuvaru koji celog života pravi npr. sendviče po jednom receptu). Grubi izgled računarskog programa je prikazan na slici 1.1.



SLIKA 1.1: Pojednostavljena slika računarskog programa.

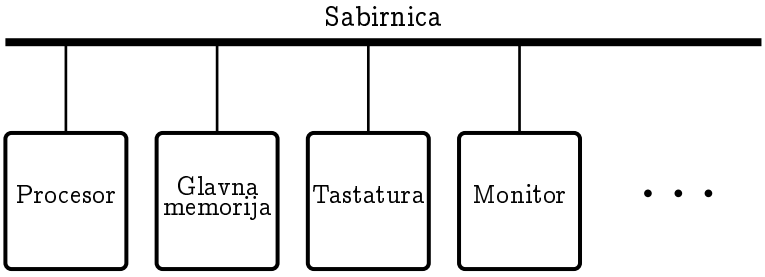
Računarski program se, kao što vidimo na slici 1.1, ne sastoji samo od naredbi koje izvršava računar. Svaki program dodatno sadrži radni prostor u kojem se čuvaju podaci koje program koristi i obrađuje. Na primer, neki program za pregledanje Interneta treba, između ostalog, obuhvatati prostor u kome se čuva adresa sajta koji korisnik želi da trenutno poseti, zatim prostor u kome se nalaze razne poruke koje se prikazuju u toku rada, kao i prostor za tekst aktuelne Web stranice koja se prikazuje. Podaci koje program mora dobiti od korisnika radi daljeg rada se nazivaju ***ulazni podaci*** (na primer, adresa željenog sajta u prethodnom primeru). Podaci koji predstavljaju rezultat rada programa na osnovu ulaznih podataka se nazivaju ***izlazni podaci*** (na primer, tekst aktuelne stranice u prethodnom primeru). Podaci neophodni za rad programa koji nisu ni ulazni ni izlazni, ponekad se nazivaju ***radni***

podaci (na primer, razne poruke programa u prethodnom primeru).

Pisanje ispravnih programa nameće dakle poznavanje odgovora na više pitanja, među kojima se izdvajaju ova tri:

- Koje se naredbe mogu koristiti u sastavu programa?
- Koji je pravilan redosled kojim naredbe treba navesti u programu da bi se postigla određena funkcionalnost prilikom izvršavanja programa?
- Koje vrste podataka program može koristiti u svom radu?

Ali pre nego što možemo odgovoriti na ova pitanja, potrebno je imati opštu sliku o glavnim delovima hardvera računara i njihovim osnovnim mogućnostima. Pojednostavljeni model svakog računara je prikazan na slici 1.2.



SLIKA 1.2: Funkcionalna šema računara.

Kao što možemo videti sa opšte slike 1.2, funkcionalni delovi svakog računara su:

- **Procesor**. To je jedini deo računara koji izvršava naredbe programa. Procesor se sastoji od relativno malog broja registara, koji predstavljaju njegovu brzu internu radnu memoriju, kao i od aritmetičko-logičke jedinice, u kojoj se zapravo izvršavaju naredbe. Procesor dakle predstavlja „mozak" računara u kojem se obavlja glavna funkcija računara — izvršavanje naredbi programa.
- **Glavna memorija**. To je deo računara u kojem se nalazi program prilikom njegovog izvršavanja. Glavna memorija se sastoji od relativno velikog broja lokacija u kojima se mogu čuvati pojedini podaci i naredbe. Da bi neki program mogao da se izvrši, on se prethodno mora preneti u glavnu memoriju računara. Glava memorija je dakle

„pasivni" uređaj čija je jedina namena da sadrži program spreman za izvršavanje.

- **Ulazno/izlazni uređaji**. To su delovi računara u koje spadaju tastatura, monitor, disk, zvučnici i tako dalje. Ulaznim uređajima se ulazni podaci programa prenose u program, a izlaznim uređajima se izlazni podaci (rezultati) iz programa prikazuju u obliku pogodnom za ljude.

- **Sabirnica (magistrala)**. To je deo računara koji obuhvata elektronska kola za međusobno povezivanje svih ostalih delova računara. Sabirnicom se prenose različite vrste podataka i signala između ostalih delova kako bi se obezbedio pravilan rad računara kao jedne celine.

Procesori računara su nažalost napravljeni tako da mogu izvršavati jedino naredbe koje su napisane na vrlo prostom jeziku koji se naziva ***mašinski jezik***. Taj jezik se sastoji od elementarnih operacija koje se uglavnom predviđene za neposredni rad sa hardverom računara. Primeri ovih primitivnih operacija su prenošenja podataka iz memorije u registre procesora, kao i obrnuto, zatim osnovne aritmetičke operacije nad registrima procesora, promena broja sledeće naredbe u programu koju treba izvršiti i tako dalje.

Operacije mašinskog jezika su izražene u binarnom zapisu koji se sastoji samo od dve binarne cifre, nule i jedan. Drugim rečima, operacije mašinskog jezika predstavljaju najprostije nizove nula i jedinica. Pored toga, podaci kojima se manipuliše u programu se predstavljaju binarnim brojevima. To znači da je program koji procesor izvršava ništa drugo nego jedan vrlo dugačak niz nula i jedinica. Računari mogu direktno raditi sa binarnim ciframa, jer se nule i jedinice mogu lako elektronski predstaviti prekidačkim kolima: uključen prekidač je zamena za jedinicu i isključen prekidač je zamena za nulu.

Na kraju zaključimo da je rad svakog računara zasnovan na vrlo jednostavnom principu:

- Glavna memorija računara sadrži naredbe i podatke programa na mašinskom (binarnom) jeziku.

- Procesor iz memorije redom uzima naredbe programa jednu za drugom i izvršava ih dok ne dođe do poslednje naredbe.

– To se radi mehanički i naredbe se „slepo" izvršavaju tačno onako kako su napisane, bez ikakvog razumevanja šta naredbe znače ili da li imaju smisla.

1.2 Java virtuelna mašina

Svaki tip procesora (Intel, PowerPC, Sparc) ima svoj mašinski jezik i može izvršiti neki program samo ako je taj program izražen na tom jeziku. Iako je teoretski moguće program napisati koristeći mašinski jezik, to je vrlo teško i podložno greškama čak i za jednostavne programe. U praksi se zato gotovo svi programi pišu na jezicima koji su prilagođeni ljudima. Takvi programski jezici se nazivaju ***jezici visokog nivoa***.

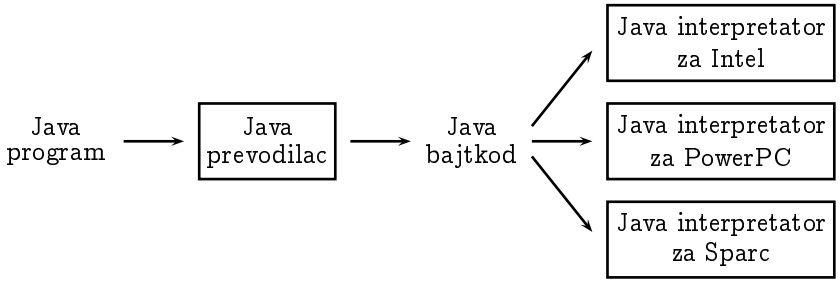
Primeri jezika visokog nivoa su Java, C, C++, C#, Pascal i tako dalje. Programi napisani na nekom jeziku visokog nivoa se ne mogu direktno izvršiti na računaru. Takav program se mora najpre prevesti na mašinski jezik. To prevođenje se ne obavlja ručno od strane ljudi, nego to rade specijalni računarski programi koji se zovu ***prevodioci*** (ili ***kompajleri***). Prevodilac dakle uzima program na jeziku visokog nivoa i prevodi ga u izvršni program na mašinskom jeziku. Prevedeni program na mašinskom jeziku se zatim može izvršiti proizvoljan broj puta, naravno pod uslovom da se to radi na jednoj vrsti računara sa odgovarajućim procesorom. Da bi se program izvršio na računaru sa drugim tipom procesora, on se mora ponovo prevesti na odgovarajući mašinski jezik koristeći drugi prevodilac.

Alternativa prevođenju programa na jeziku visokog nivoa radi njegovog izvršavanja je postupak koji se naziva ***interpretiranje***. Umesto prevodioca, koji prevodi ceo program, koristi se ***interpretator*** koji istovremeno prevodi i izvršava program postupno naredba-po-naredba, kako to logika programa nalaže. Interpretator je program koji radi slično kao procesor u ciklusu „uzmi naredbu i izvrši je". Preciznije, kod izvršavanja programa, interpretator redom uzima po jednu naredbu programa na visokom nivou, određuje šta je neophodno za njeno izvršavanje i, na kraju, izvršava odgovarajuće mašinske instrukcije radi toga. Obratite pažnju na to da ako se po logici programa neka naredba ne izvršava, ona se nikad neće ni prevesti. Sa druge strane, ako se po logici programa neka naredba izvršava više puta, ona će se i prevoditi više puta. Ovo je razlog zašto je izvršavanje programa njegovim interpretiranjem obično sporije od izvršavanja istog programa koji je prethodno preveden na mašinski jezik.

U stvari, interpretator ne mora da „razume" baš neki jezik visokog nivoa, već može interpretirati i mašinski jezik drugog tipa procesora. U tom slučaju se izvršni program na mašinskom jeziku za određeni tip procesora može izvršiti pomoću interpretatora na računaru sa drugim tipom procesora. Na primer, na običnom PC računaru (sa Intelovim procesorom) možemo izvršiti mašinski program za Macintosh računare (sa PowerPC procesorom) uz pomoć interpretatora na PC-ju koji interpretira mašinski jezik PowerPC procesora. Tada obično govorimo o simulaciji jednog tipa računara na drugom tipu računaru. Naravno, simulacija je inherentno sporija od direktnog izvršavanja prevedenog programa, jer se jedna mašinska operacija nekog procesora obično izvršava pomoću više mašinskih operacija drugog procesora.

U slučaju programskog jezika Java se koristi pristup koji je drugačiji od uobičajenog i sastoji se od kombinacije prevođenja i interpretiranja. 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 (JVM)***. Mašinski jezik Java virtuelne mašine se naziva ***Java bajtkod***.

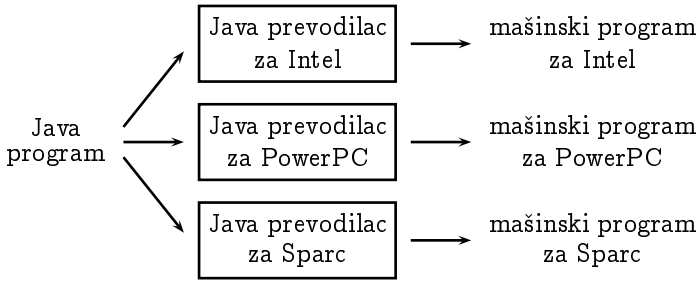
Prema tome, Java program se prevodi u Java bajtkod koji se ne može direktno izvršiti na računaru. Za izvršavanje Java programa prevedenog u Java bajtkod je potreban interpretator Java bajtkoda na računaru na kojem želimo izvršiti Java program. Ova šema je prikazana na slici 1.3.



SLIKA 1.3: Izvršavanje Java programa.

Kao što vidimo sa slike 1.3, potrebni su različiti interpretatori Java bajtkoda za svaki tip računara, ali je dovoljno samo jedno prevođenje Java programa u Java bajtkod. To je zapravo jedno od glavnih svojstava Jave u odnosu na druge jezike: isti prevedeni Java program se može izvršiti na različitim tipovima računara.

Može se postaviti pitanje zašto nam je uopšte potreban posredni Java bajtkod, odnosno mašinski jezik virtuelnog računara? Drugim rečima, zašto ne bismo mogli imati Java prevodilac za mašinski jezik različitih tipova stvarnih procesora i prevedeni Java program onda izvršiti na odgovarajućem računaru? Ova klasična šema je prikazana na slici 1.4.



SLIKA 1.4: Klasičan način izvršavanja Java programa.

Ima više razloga zašto nije izabran klasičan pristup za izvršavanje Java programa. Pre svega, prevodilac za Java jezik je vrlo složen program, jer je i sama Java složen programski jezik visokog nivoa. Sa druge strane, interpretator Java bajtkoda je prilično jednostavan program, jer je svaki mašinski jezik po svojoj prirodi veoma ograničen. To znači da je pisanje interpretatora Java bajtkoda za novi tip računara mnogo lakše od pisanja Java prevodioca za isti računar.

Pored toga, mnogi Java programi se izvršavaju tako što se automatski dobijaju preko mreže. To odmah povećava bezbednosne rizike, jer se ne može obezbediti apsolutna pouzdanost udaljenih računara sa kojih se takvi programi prenose. Ali kada se takav program izvršava pod kontrolom interpretatora Java bajtkoda, zaštita od mogućih štetnih posledica se može pružiti dodatnim proverama od strane interpretatora.

Na kraju napomenimo 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 programi na drugom programskom jeziku visokog nivoa bi se mogli prevoditi u Java bajtkod. Ipak, kombinacija Java jezika i Java bajtkoda omogućuje programiranje na modernom, objektno orijentisanom jeziku i istovremeno bezbedno izvršavanje napisanih programa na različitim tipovima računara.

1.3 Razvoj Java programa

Za učenje nekog programskog jezika je vrlo važno početi pisanje programe na tom jeziku od samog početka. Koncepti i konstrukcije svakog programskog jezika se najbolje uče kroz praktičnu primenu. Da bismo ovo mogli primeniti za Java jezik, moramo najpre upoznati tehničke detalje razvoja Java programa.

Postupak razvoja Java programa obuhvata uglavnom tri koraka: unošenje, prevođenje i ispravljanje Java programa. Obratite pažnju na to da se pod ovim ne podrazumeva najvažniji korak smišljanja programa i njegovog pisanja obično na papiru. To je kreativni proces kome je posvećen ostatak knjige. Ovde se dakle pretpostavlja da smo već napisali neki program i da želimo da ga unesemo i izvršimo na računaru.

Razvoj Java programa je mehanički postupak na računaru, mada često mukotrpan, koji se može obavljati u okviru dva osnovna razvojna okruženja: tekstualnog i grafičkog. Dok je tekstualno okruženje relativno standardno (konzolni prozor za Unix, DOS prozor za Windows i slična mogućnost za druge operativne sisteme), na raspolaganju su nam mnoga grafička okruženja, kako besplatna tako i komercijalna. Pošto zbog njihove brojnosti ovde ne možemo dati potpun opis svih okruženja, u našem izlaganju ćemo se ograničiti na operativni sistem Windows. Pored toga, od grafičkih razvojnih okruženja pomenućemo samo zvanični besplatni softver NetBeans. Ovim nećemo mnogo izgubiti na kompletnosti, jer sva okruženja rade na otprilike sličan način. Zbog toga nije teško preći na drugo okruženje ukoliko znate jedno, jer su osnovni koncepti zajednički za sve.

Radi praktičnog ilustrovanja celog postupka razvoja Java programa, koristićemo kao radni primer jednostavan program *Zdravo* koji je naveden u listingu 1.1. Naravno da se u ovoj fazi učenja jezika Java ne očekuje da čitaoci razumeju ovaj program. Dovoljno je za sada samo znati da se od korisnika programa najpre zahteva nekoliko ulaznih podataka, a da se zatim na ekranu ispisuje poruka dobrodošlice za korisnika.

LISTING 1.1: Program *Zdravo*.

```
// Prvi Java program

import java.util.*;
public class Zdravo
```

```
{
   public static void main(String[] args)
   {
      Scanner tastatura = new Scanner(System.in);
      System.out.print("Kako se zovete: ");
      String ime = tastatura.nextLine();
      System.out.print("Koliko imate godina: ");
      int god = tastatura.nextInt();
      System.out.println("Zdravo " + ime + "!");
      System.out.println(god + " su najlepše godine.");
   }
}
```

Tekstualno razvojno okruženje

Prvi korak razvoja Java programa u ovoj vrsti okruženja je unošenje teksta željenog programa. Za to može poslužiti bilo koji tekstualni editor, na primer Notepad koji se dobija uz Windows ili neki složeniji, programerski editor koji se besplatno može dobiti na Internetu. Obratite pažnju na to da se ne mogu koristiti programi za obradu dokumenata (kao što je Word), jer oni dodaju svoje specijalne znakove za formatiranje teksta.

Koristeći bilo koji editor treba najpre upisati tekst radnog programa *Zdravo* i sačuvati ga u datoteci pod imenom `Zdravo.java`. Ime datoteke koja sadrži tekst Java programa nije proizvoljno i mora biti sastavljeno iz dva dela razdvojena tačkom. Prefiks imena datoteke (u ovom slučaju `Zdravo`) mora tačno odgovarati imenu klase koja se nalazi u datoteci (to ime je ono koje sledi iza reči `class` u tekstu programa). (O klasama će mnogo više biti reči u daljem izlaganju.) Sufiks (ili ekstenzija) imena datoteke mora biti `java`. Zapamtite da je ovo opšte pravilo i da važi za svaki Java program.

Nakon što je tekst programa *Zdravo* sačuvan u datoteci, program se može prevesti i izvršiti. Prevođenje nekog Java programa se obavlja Java prevodiocem, dok se izvršavanje (interpretiranje) dobijenog Java bajtkoda obavlja Java interpretatorom. Java prevodilac se pokreće komandom `javac`, a Java interpretator se pokreće komandom `java`.

Da biste dakle radni program *Zdravo* (ili bilo koji drugi Java program uz odgovarajuće izmene) razvili u tekstualnom razvojnom okruženju, postupite na sledeći način:

1. Otvorite prozor nekog editora i pažljivo unesite tekst programa *Zdravo* na strani 8. Na kraju to sačuvajte u datoteci `Zdravo.java`.

2. Otvorite DOS prozor i promenite radni direktorijum da bude onaj u kojem ste sačuvali datoteku `Zdravo.java`.

3. Prevedite program *Zdravo* koji se nalazi u datoteci `Zdravo.java` komandom

   ```
   javac Zdravo.java
   ```

   Rezultat prevođenja programa *Zdravo* je njegov bajtkod koji se dobija u datoteci `Zdravo.class`.

4. Izvršite (interpretirajte) dobijeni bajtkod programa *Zdravo* komandom

   ```
   java Zdravo
   ```

Ako je prevođenje ili interpretiranje programa bilo neuspešno u trećem ili četvrtom koraku, to znači da ste pogrešno uneli tekst programa *Zdravo*. Ispravite greške u prozoru editora, sačuvajte izmene u istoj datoteci `Zdravo.java` i ponovite prevođenje i izvršavanje programa. Za ovo je najbolje držati oba prozora, editora i DOS-a, otvorenim a prelaskom iz jednog u drugi ispravljati i izvršavati program koji se razvija.

Grafičko razvojno okruženje

NetBeans IDE (Integrated Development Environment) je integrisano grafičko razvojno okruženje koje je namenjeno razvoju različitih vrsta programa. Ovde ćemo predstaviti osnovne mogućnosti tog alata za razvoj Java programa i pokazati sve neophodne korake na primeru radnog programa *Zdravo*.

Neophodni softver. Za razvoj Java programa u okruženju NetBeans je potrebno imati dva softverska paketa instalirana na računaru:

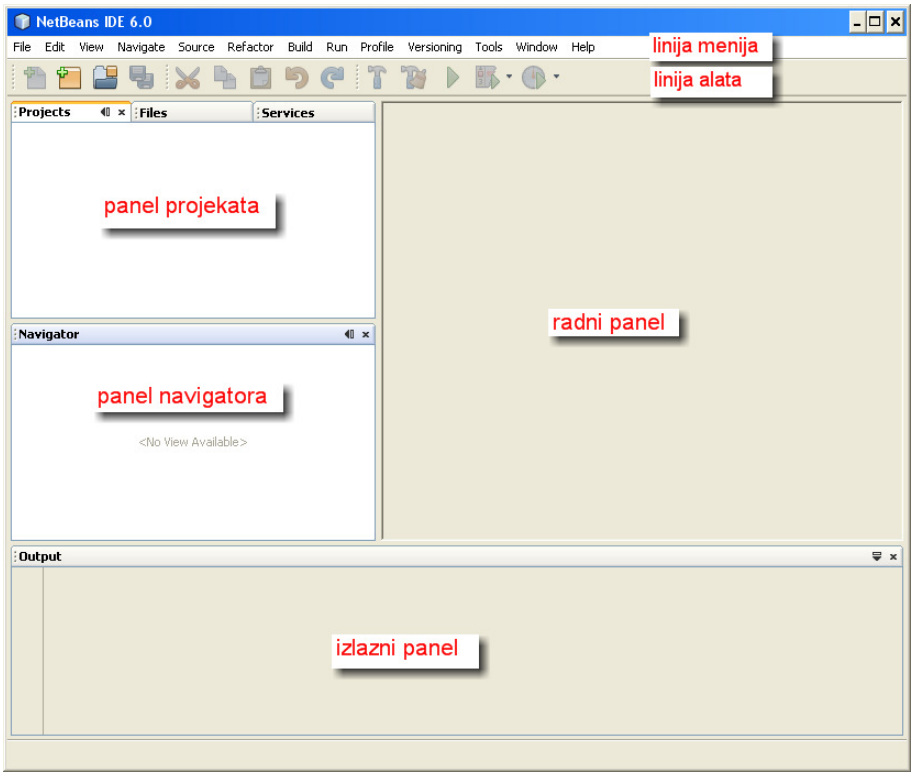
- J2SE JDK 6 (Java Development Kit, version 6)
- NetBeans IDE 6.0

Ovi softverski paketi su besplatni i mogu se preuzeti na adresama:

- `http://java.sun.com/javase/downloads/index.jsp`
- `http://download.netbeans.org/netbeans/6.0/final/`

Instalacija ovog neophodnog softvera je jednostavna — svodi se na prihvatanje ponuđenih podrazumevanih opcija i prelaska na sledeće korake dok se ne završi instalacija. Ipak, da ne biste naknadno morali da ručno konfigurišete pojedine parametre, obratite pažnju na redosled instaliranja: najpre instalirajte J2SE, a nakon toga NetBeans.

Osnovne mogućnosti okruženja NetBeans. Glavni prozor okruženja NetBeans (slika 1.5) izgleda slično drugim grafičkim okruženjima za razvoj programa na drugim programskim jezicima (na primer, MS Visual Studio).



Slika 1.5: Glavni prozor okruženja NetBeans.

Glavni prozor se sastoji od linije menija i linije alata na vrhu prozora, dok preostali deo prozora dele četiri panela u kojima se prikazuju informacije relevantne za aktuelni program koji se razvija. Sa leve strane se

nalaze panel projekata i panel navigatora, sa desne strane se nalazi radni panel, a na dnu se nalazi izlazni panel.

Namena pojedinih delova glavnog prozora okruženja NetBeans su:

- Linija menija sadrži sve menije iz kojih se mogu birati opcije za postojeće funkcije NetBeans-a. Osnovni meniji su:
  - Meni *File* je namenjen za formiranje novih i otvaranje postojećih projekata, za formiranje novih i otvaranje postojećih fajlova, kao i za ostale standardne aktivnosti u vezi sa fajlovima od kojih se sastoji Java aplikacija.
  - Meni *Edit* ima potpuno istu funkciju za rad sa tekstom programa kao istoimeni meniji u ostalim grafičkim okruženjma (na primer: *undo*, *redo*, *copy*, *paste*, *find*, *replace*).
  - Meni *Build* je namenjen za prevođenje Java programa koji se sastoji od jedne datoteke ili više njih.
  - Meni *Run* je namenjen za izvršavanje prevedenog programa. Pored toga, u ovom meniju se može pokrenuti debager radi otklanjanja grešaka, testiranja i detaljnog praćenja stanja promenjivih u toku rada programa.
- Linija sa alatima sadrži ikone alata koji omogućavaju brži pristup različitim funkcijama NetBeans-a nego preko menija.
- Panel projekata sadrži kartice *Projects*, *Files* i *Services* u kojima se hijerarhijski prikazuju projekti, datoteke i servisi koji su trenutno aktuelni za program.
- Panel navigatora prikazuje detalje stavki koje su izabrane u jednoj od tri pomenute kartice panela projekata.
- U desnom, radnom panelu se prikazuje sadržaj datoteka koje su izabrane u jednoj od kartica panela projekata. Sadržaj otvorenih datoteka u radnom panelu se može menjati, jer se oni automatski otvaraju u integrisanom editoru NetBeans-a.
- U donjem, izlaznom panelu se prikazuju sve važne poruke tokom prevođenja programa, kao i ulazno/izlazni podaci samog programa tokom njegovog izvršavanja.

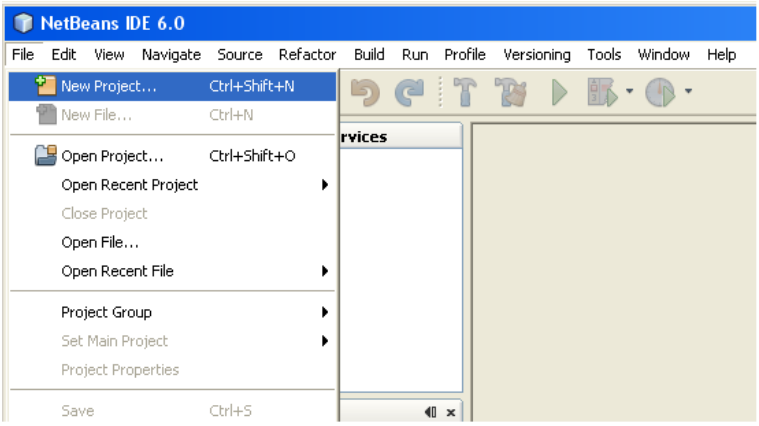
Pravljenje projekta. Kao i u drugim grafičkim okruženjima, osnovna jedinica rada u okruženju NetBeans je ***projekat***. Projekat odgovara, grubo

govoreći, jednom Java programu i sastoji se od njegovih izvornih datoteka, pridruženih informacija o tome kako ih treba prevesti i izvršiti, kao i od ostalih tehničkih detalja o putanjama, dokumentaciji i slično. Sve ove podatke NetBeans smešta u direktorijumu projekta (sa poddirektorijumima) koji se pravi na osnovu imena projekta. Da bi se omogućilo grupisanje više povezanih projekata na jednom mestu, svi direktorijumi takvih projekata se inicijalno prave u drugom datom direktorijumu koji se obično naziva ***radno mesto*** (engl. *workplace* ili *workbench*).

Naravno, o svemu ovome vodi računa uglavnom sâm softver, pa za razne vrste uobičajenih Java projekata (obična aplikacija, biblioteka klasa, Web aplikacija i tako dalje) postoje standardni šabloni od kojih se inicijalno pravi odgovarajući projekat. Nakon početnog formiranja projekta na osnovu odgovrajućeg šablona, projekat se otvara u glavnom prozoru NetBeans-a tako što se njegova logička struktura prikazuje u kartici *Projects* i struktura datoteka u kartici *Files* panela projekata. Pored toga, u radnom panelu se otvara glavna datoteka projekta koji sadrži početni izvorni tekst Java programa.

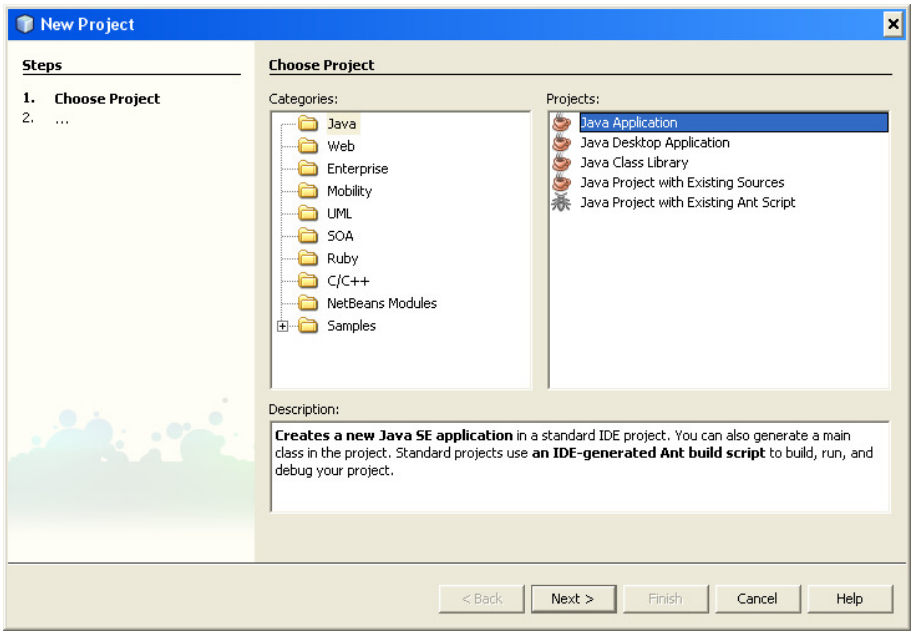
Da biste napravili novi projekat nakon pokretanja NetBeans-a, postupite na sledeći način (pritiskom na dugme Next odgovarajućeg prozora u svakom koraku, prelazite na sledeći korak):

1. U glavnom prozoru izaberite opciju File > New Project, kao što je to prikazano sa slici 1.6.



Slika 1.6: Izbor novog projekta.

2. U otvorenom prozoru New Project izaberite kategoriju Java i zatim opciju Java Application, kao što je to prikazano sa slici 1.7.

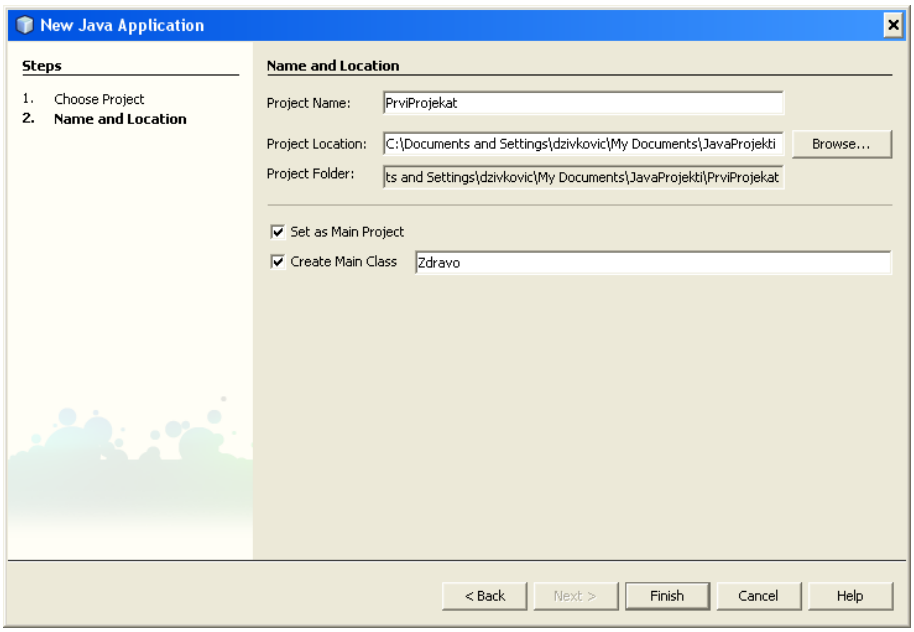


SLIKA 1.7: Izbor kategorije Java programa koji se pravi.

3. U otvorenom prozoru New Java Application treba upisati sledeće podatke o novom projektu (što je ilustrovano sa slici 1.8):

   - U polju Project Name upišite `PrviProjekat` za ime projekta.
   - U polju Project Location, pritiskom na dugme Browse izaberite direktorijum radnog prostora gde želite da se nalaze svi projekti. Obratite pažnju na to da taj direktorijum mora biti prethodno napravljen (u primeru je to `JavaProjekti`).
   - Ostavite polje Set as Main Project izabrano.
   - U polju Create Main Class upišite `Zdravo` za ime glavne klase Java programa.

Nakon što pritisnete dugme Finish, završava se formiranje projekta Zdravo i on se otvara u glavnom prozoru NetBeans-a (slika 1.9). Pri tome,

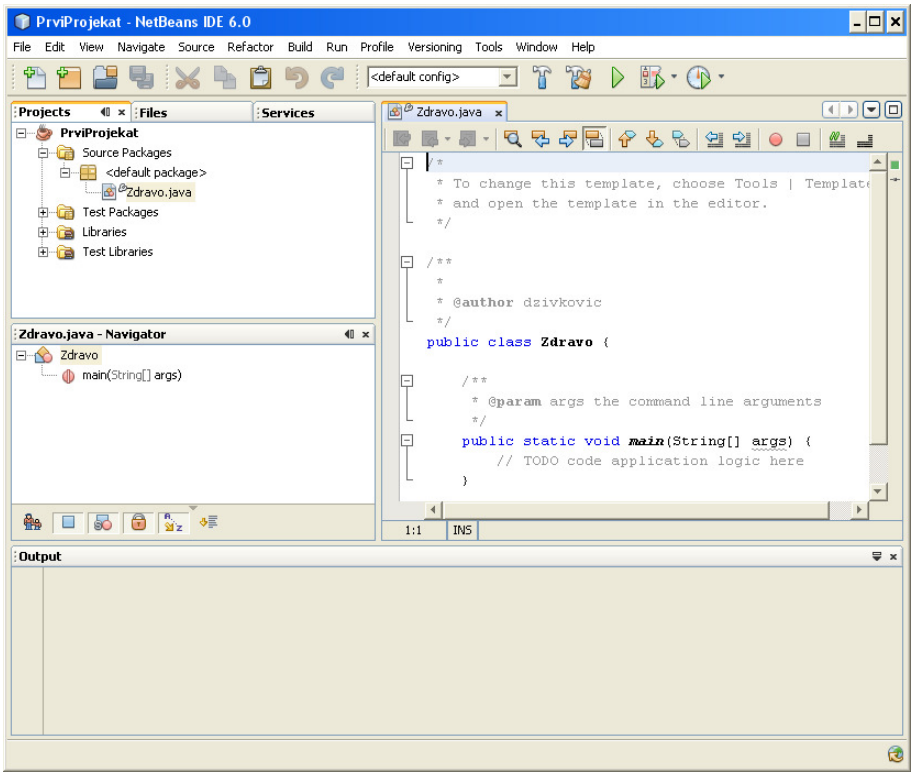


SLIKA 1.8: Izbor podataka za novi projekat.

- u panelu projekata se prikazuje stablasta struktura delova novog projekta (datoteke i paketi izvornog teksta, biblioteke koje su potrebne i tako dalje);
- u radnom panelu se otvara datoteka `Zdravo.java` pod kontrolom integrisanog editora;
- u panelu navigatora se prikazuje struktura elemenata izabrane klase koja se može iskoristiti za brzo prelaženje sa jednog elementa na drugi u izvornom tekstu radnog panela.

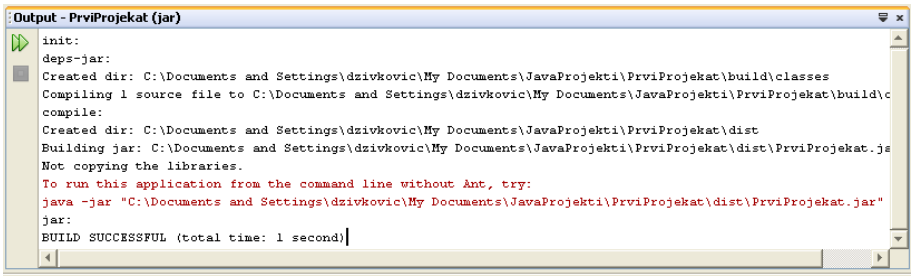
Unošenje izvornog teksta programa. Pošto ste u postupku pravljenja projekta izabrali polje Create Main Class, u okruženju NetBeans se formira kostur glavne klase Zdravo. Automatski generisan tekst u datoteci `Zdravo.java` zamenite tekstom radnog programa *Zdravo* na slici 1.1. Promene u datoteci `Zdravo.java` sačuvajte birajući opciju File > Save.

Prevođenje programa. Da biste preveli Java program, izaberite opciju Build > Build Main Project iz linije menija u glavnom prozoru okruženja



SLIKA 1.9: Otvaranje novog projekta.

NetBeans. Obaveštenja o statusu postupka prevođenja mogu se videti u izlaznom panelu. U konkretnom slučaju projekta Zdravo, sadržaj izlaznog panela izleda slično onome što je prikazano na slici 1.10.

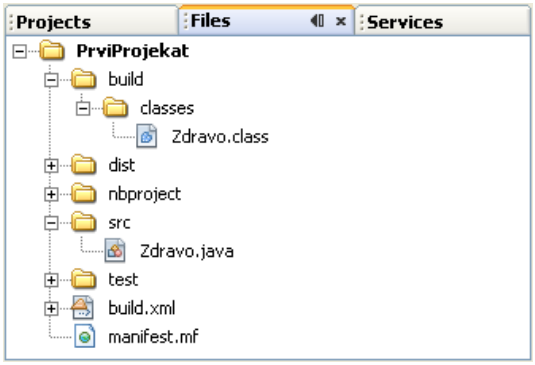


SLIKA 1.10: Status prevođenja Java programa u izlaznom panelu.

Ako se rezultat prevođenja završava porukom BUILD SUCCESSFUL,

to znači da je prevođenje bilo uspešno i da se program može izvršiti. U suprotnom, ako se rezultat prevođenja završava porukom BUILD FAILED, program sadrži greške koje se moraju ispraviti. Greške se u izlaznom panelu prikazuju u obliku hiperveza čijim praćenjem možete preći u izvorni tekst radnog panela na mestu odgovarajuće greške. Na taj način možete ispraviti sve greške i ponovo prevesti program izborom opcije Build > Build Main Project. Ovaj postupak morate ponoviti sve dok prevođenje ne bude uspešno.

U konkretnom primeru, uspešno prevedeni projekat se sastoji od bajtkoda u datoteci `Zdravo.class`. Lokacije datoteka raznih komponenti projekta Zdravo možete pregledati otvaranjem kartice Files panela projekata i proširivanjem pojedinih čvorova prikazane strukture. Jedan primer ove mogućnosti je prikazan na slici 1.11.

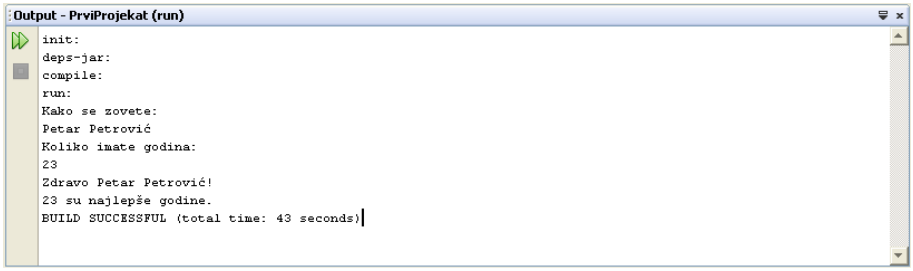


Slika 1.11: Datoteke raznih komponenti projekta.

Izvršavanje programa. Uspešno preveden program se izvršava izborom opcije Run > Run Main Project u glavnom prozoru NetBeans-a. Ulazno/izlazni podaci tokom izvršavanja programa se takođe prikazuju u izlaznom panelu tog prozora. Jedan primer izvršavanja programa *Zdravo* je prikazan na slici 1.12.

Ukoliko želite da program izvršite više puta (možda sa promenjenim ulaznim podacima), to postižete ponovnim biranjem opcije Run > Run Main Project.

Pored ovih osnovnih mogućnosti, grafičko okruženje NetBeans pruža još mnogo toga što je potrebno za razvoj složenih i profesionalnih apli-



SLIKA 1.12: Izvršavanje programa *Zdravo*.

kacija. Dodatne informacije o tome možete naći u priručniku *Using NetBeans IDE 5.5* i drugim materijalima na sajtu `www.netbeans.org`.

Glava

2

Uvod u programski jezik Java

U ovom poglavlju će čitaoci moći da se upoznaju sa osnovnim elementima programskog jezika Java, kao i sa paradigmom objektno orijentisanog programiranja na kojoj se Java zasniva.

2.1 Kratak istorijat jezika

Programski jezik Java je nastao 1990. god. u kompaniji Sun Microsystems kao pilot projekat jezika za programiranje ,,pametnih" kućnih uređaja (mobilnih telefona, TV uređaja, raznih plejera i slično). Kao što to obično biva, od početne zamisli do prve javne verzije jezika je cirkulisalo nekoliko probnih verzija i usput je dolazilo do promena ciljeva jezika. Java je zvanično objavljena 1995. god. i istovremeno je izdata njena besplatna implementacija. Prva verzija je dalje pretrpela mnogobrojne izmene, tako da je danas aktuelna šesta verzija jezika (Java 6). Međutim, ni tu nije kraj i jezik se i dalje aktivno razvija.

U stvari, danas je preciznije govoriti o Java platformi umesto o Java jeziku, jer se pod tim podrazumeva i veliki broj softverskih komponenti (Java API, *Java Application Programming Interface*) koji idu uz sam jezik. Ne samo to, da bi se olakšalo snalaženje u stotinama dodatnih paketa, Java platforma je podeljena u tri edicije prema vrsti programa kojima je namenjena:

- Java SE (Standard Edition)

- Java ME (Micro Edition)
- Java EE (Enterprise Edition)

Za standardne programe na stonim računarima se najčešće koristi Java SE. Edicija Java ME je namenjena pisanju programa za uređaje sa smanjenim hardverskim mogućnostima, a Java EE najzahtevnijim programima sa distribuiranim, servisno orijentisanim režimom rada.

Jedan od ciljeva jezika Java od samog njegovog nastanka je bio olakšano Internet programiranje. Zbog toga je ustanovljena posebna terminologija za kategorije Java programa koji koriste Internet tehnologije. Tako obično možemo razlikovati tri vrste Java programa:

- **Aplikacija.** Pod tim se podrazumeva samostalni uobičajeni program. Takav program se izvršava (interpretira) pod kontrolom interpretatora Java virtuelne mašine na jednom računaru.
- **Aplet.** To je program koji se ugrađuje u neku veb stranu (slično kao slike, zvuk i ostali elementi) i izvršava ga brauzer klijenta (na primer, Internet Explorer). Ono što izdvaja ove programe je njihova automatska distribucija prilikom otvaranja veb strane, kao i ograničene mogućnosti zbog bezbednosti.
- **Servlet i JSP (Java Server Pages).** To je program koji omogućava dinamičko generisanje veb strana nekog sajta i izvršava ga veb server tog sajta.

Treba naglasiti da je sav softver u vezi sa jezikom Java beplatan od samog početka. To je umnogome uticalo na njegovu popularnost, ali i doprinelo njegovom razvoju zahvaljujući usvajanju konstruktovnih kritika pojedinih elemenata jezika od strane mnogobrojnih korisnika. Najnovije verzije Java platforme i obimna dokumentacija su slobodno dostupni na zvaničnom sajtu `java.sun.com`.

2.2 Objekti, klase i nasleđivanje

Java je objektno orijentisani jezik. To znači da pisanje Java programa zahteva poznavanje osnovnih principa objektno orijentisanog programiranja. To su objekti, klase i nasleđivanje. Kako se oni ne koriste kod proceduralnog načina programiranja (na koji su čitaoci možda naviknuti),

najpre ćemo ukratko objasniti nove pojmove objektnog razmišljanja potrebne za pisanje Java programa.

Objektno orijentisan pristup programiranju se zasniva na manipulisanju ***objektima***. To znači da se ostvarivanje programske logike postiže definisanjem raznih objekata (ko šta radi) i njihovom interakcijom (šta ko radi). Objekti su auto, zgrada, bankovni račun, student i slično, odnosno sve stvari koje poseduju neku funkcionalnost ili nad kojima se može izvoditi neka aktivnost u određenoj situaciji. Ove ideje je najbolje potkrepiti jednim primerom iz svakodnevnog života, jer se u objektno orijentisanoj metodologiji teži da se rešavanjem problema simulira pristup koji se primenjuje u realnim situacijama.

Razmotrimo zato kako radi, recimo, tipični restoran. Kada gost dođe u restoran, on izabere sto i poručuje jelo i piće iz jelovnika. Jedan kelner prima porudžbinu i donosi poručeno jelo i piće. Kuvar sprema jelo i šanker priprema piće. Na kraju, isti kelner donosi račun i gost ga plaća.

Koji objekti učestvuju u radu rastorana? To su očigledno gost, kelner, kuvar, šanker, ali to nije sve što je dovoljno za rad restorana. Druge neophodne stvari su sto, jelovnik, jelo, piće i račun. Sve su to objekti čijim se međusobnim manipulisanjem realizuje rad restorana.

U primeru vidimo da postoje razni tipovi objekata. Njihova imena (gost, kelner, jelovnik, račun ...) nama otprilike govore šta je njihova uloga, ali računari ne znaju šta se podrazumeva pod tim imenima. Da bismo to opisali za računare, definišemo ***klase*** objekata. Klasa u Javi opisuje računaru koje podatke svi objekti klase imaju i šta takvi objekti mogu da urade. Svaki objekat jedne klase ima iste podatke (stvari koje ga obeležavaju) i iste mogućnosti (stvari koje može uraditi). Na primer, svaki objekat klase porudžbina treba da sadrži gosta koji je napravio određenu porudžbinu i spisak jela i pića sa cenama od kojih se porudžbina sastoji, kao i mogućnost da se izračuna ukupna vrednost porudžbine.

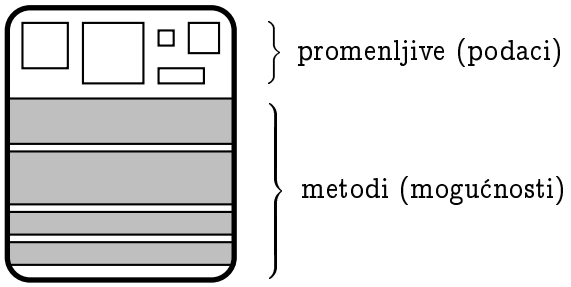
Primetimo da može postojati više objekata iste klase. Restoran može imati dva kuvara, tri kelnera, dvadeset jela na meniju, četiri gosta u datom trenutku i tako dalje. Zašto restoran nema samo jednog zaposlenog, jer bi u principu sve poslove kuvara, šankera i kelnera mogao obavljati jedan čovek? To bi možda bilo izvodljivo ako restoran ima samo jednog gosta, ali je praktično nemoguće ako ima više njih. Funkcionisanje restorana je podeljeno u više poslova kako bi se oni izvodili efikasnije i bolje. Slično tome, objektno orijentisani programi distribuiraju zadatke na više obje-

kata kako bi programi bili efikasniji i bolji. Pored toga, na taj način se programi mogu lakše menjati i proširivati.

Šta je objekat?

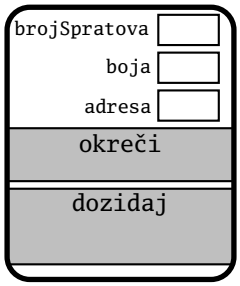
Objekat predstavlja entitet iz stvarnog života sa svojim jedinstvenim identitetom. Tako, student, sto, stolica, taster na tastaturi, telefon i tako dalje mogu se posmatrati kao objekti. Pored ovih „fizičkih" stvari, pojam objekta obuhvata i apstraktne entitete kao što su krug, kredit, brak i tako dalje. Jedinstven identitet objekta je određen njegovim obeležjima (atributima) i ponašanjem. Na primer, student se može identifikovati po imenu i prezimenu, fakultetu koji studira, broju indeksa i slično, a njegovo ponašanje se sastoji od mogućnosti da polaže ispite iz određenih predmeta, da bude član studentskih organizacija i slično.

Softverski objekti oponašaju stvarne objekte po tome što imaju pridružene podatke (koji određuju šta su obeležja ili atributi objekta) i mogućnosti (koji određuju šta objekat može da uradi). Podaci koje objekti sadrže se predstavljaju *promenljivim* (poljima), a njihove mogućnosti *metodima* (procedurama). Opšti izgled nekog objekta je dat na slici 2.1.



Slika 2.1: Tipični objekat.

Objekti iste vrste se definišu koristeći zajedničku klasu. Na primer, svaki objekat klase `Zgrada` može imati promenljive, recimo, `brojSpratova`, `boja` i `adresa` čije vrednosti obeležavaju konkretnu zgradu. Pored toga, svaki takav objekat može imati metode, recimo, `okreči` i `dozidaj` kojima se može nešto uraditi sa konkretnom zgradom. Softverski oblik objekta klase `Zgrada` je prikazan na slici 2.2.



SLIKA 2.2: Jedan objekat klase Zgrada.

Šta je klasa?

Klasa je opis objekata sa zajedničkim svojstvima. Definicijom neke klase se određuju polja (promenljive) i metodi (procedure) koje poseduju svi objekti te klase. Klase dakle definišu šablon kako izgledaju pojedini objekti tih klasa.

Vrlo je važno razumeti da se u Java programu pišu klase, a objekti se ne pišu nego se konstruišu na osnovu tih klasa. Dobra analogija klase i objekata u Javi je građevinski nacrt zgrade na papiru i izgrađenih zgrada u arhitekturi i građevinarstvu:

- Nacrt zgrade za arhitekte je ono što je klasa za programere.
- Izgrađena zgrada na osnovu datog nacrta za arhitekte je ono što je konstruisan objekat na osnovu date klase za programere.

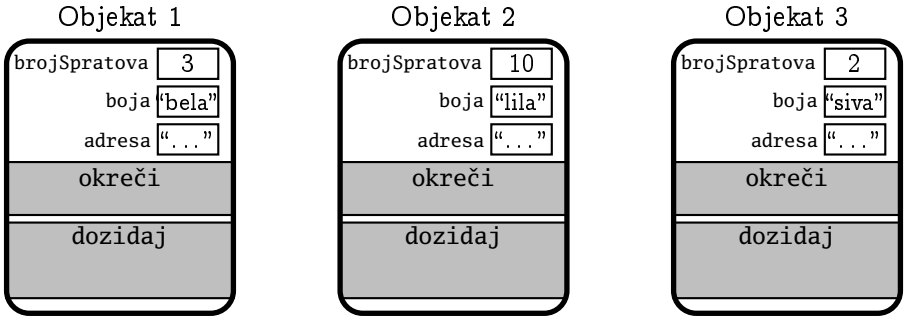
Pored toga, kao što se na osnovu jednog nacrta zgrade može izgraditi više zgrada, tako se i na osnovu jedne klase može konstruisati više objekata.

Ako posmatramo primer klase Zgrada o kojoj smo govorili u prethodnom odeljku, njena definicija bi u Javi izgledala otprilike ovako:

```
class Zgrada
{
   int brojSpratova;
   String boja;
   String adresa;
      ⋮
   public void okreči() { ... };
```

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

Detalji definicije ove klase za sada nisu važni, već je bitno to da na osnovu nje možemo konstruisati više objekata klase Zgrada. Ako konstruišemo, recimo, tri objekta te klase sa svojim individualnim podacima, onda je njihov izgled u memoriji računara prikazan na slici 2.3.



SLIKA 2.3: Tri objekta klase Zgrada.

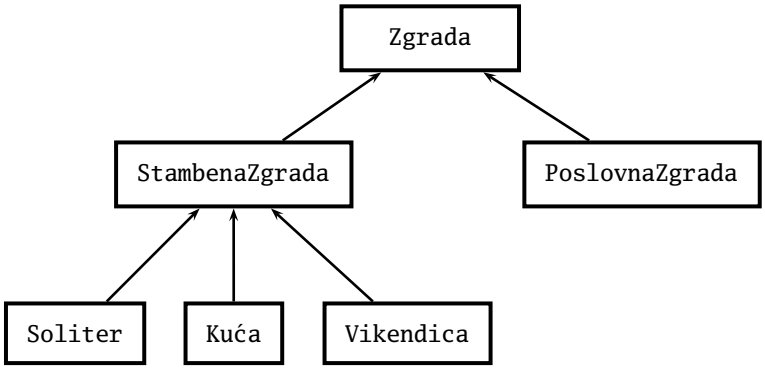
Primetimo sa slike 2.3 da svaki konstruisani objekat neke klase ima svoje primerke (objektnih) promenljivih i metoda te klase. To je glavna odlika objekata i jedna od karakteristika koja objektno orijentisano programiranje razlikuje od proceduralnog programiranja.

Šta je nasleđivanje?

Jedna od prednosti objektno orijentisanog programiranja u odnosu na proceduralno programiranje ogleda se u olakšanom višekratnom korišćenju postojećeg programskog koda. To je omogućeno primenom koncepta ***nasleđivanja*** klasa čime se postiže formiranje novih klasa na osnovu postojećih. Pri tome nova klasa proširuje postojeću klasu i nasleđuje sve promenljive i metode postojeće klase.

Nasleđivanjem se uspostavlja hijerarhijska relacija između srodnih klasa. Na primer, na osnovu klase Zgrada možemo formirati stablastu strukturu klasa-naslednica prikazanu na slici 2.4. Na toj slici je strelicom od jedne do druge klase označen odnos nasleđivanja klasa, odnosno da klasa na početku strelice proširuje klasu na koju strelica pokazuje.

Definicija klase StambenaZgrada, koja prema ovoj slici proširuje klasu Zgrada, u Javi bi izgledala otprilike ovako:

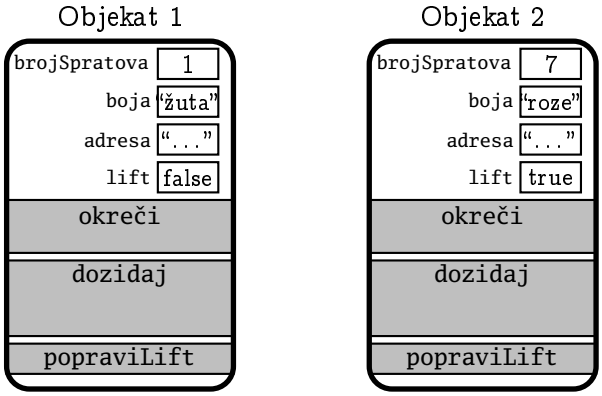


SLIKA 2.4: Nasleđivanje i hijerarhijska struktura klasa.

```
class StambenaZgrada extends Zgrada
{
   boolean lift;
      ⋮
   public void popraviLift() { ... };
}
```

Detalji ove definicije opet nisu važni, već treba samo primetiti da se nova klasa StambenaZgrada definiše na isti način kao obična klasa, osim što se postojeća klasa koja se proširuje navodi iza reči extends. Pri tome, svi objekti klase StambenaZgrada, pored definisanih promenljivih i metoda u klasi StambenaZgrada, sadrže (nasleđuju) i sve definisane promenljive i metode u klasi Zgrada. Ako konstruišemo, recimo, dva objekta klase StambenaZgrada sa svojim individualnim podacima, onda je njihov izgled u memoriji računara prikazan na slici 2.5.

Na kraju napomenimo da terminologija u vezi sa nasleđivanjem nije standardizovana, pa je u upotrebi više sinonima za iste pojmove. Na primer, za postojeću klasu koja se proširuje se kaže da je to bazna klasa, natklasa ili klasa-roditelj, dok se za proširenu klasu koriste termini izvedena klasa, potklasa ili klasa-dete.



SLIKA 2.5: Dva objekta klase StambenaZgrada.

2.3 Paketi i dokumentacija

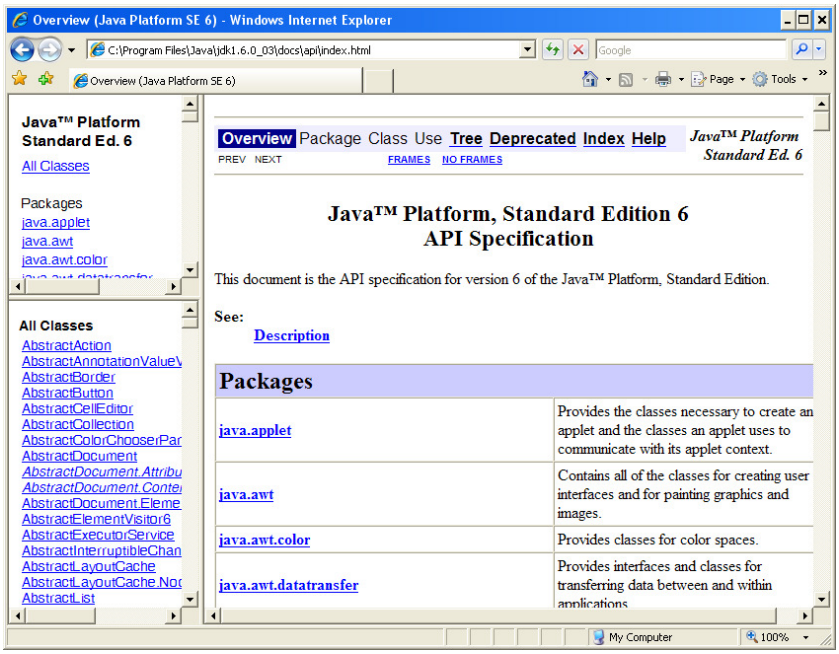
Java platforma (Java virtuelna mašina i Java API) sadrži veliki broj unapred napisanih klasa koje se mogu koristiti u programima. Da bi se olakšalo nalaženje i upotreba potrebnih klasa, sve klase Java platforme su grupisane u pakete, analogno organizaciji datoteka i direktorijuma u okviru sistema datoteka na disku.

Paket u Javi je dakle kolekcija klasa koje su namenjene jednoj vrsti posla i koje zato čine funkcionalnu celinu. Drugim rečima, jedan paket predstavlja biblioteku klasa. Neki od osnovnih paketa su:

- `java.lang`
- `java.util`
- `java.io`
- `java.net`
- `java.awt`
- `java.math`

Ostale pakete (njih preko 200) ne vredi nabrajati, a pogotovo nije izvodljivo opisati namenu svih klasa u njima. Obično se sadržaj paketa može prepoznati na osnovu njegovog imena, ali tačno poznavanje svake klase je praktično nemoguće. Zbog toga je neophodno da se Java programeri odmah naviknu na korišćenje dokumentacije, koja je besplatno raspoloživa u elektronskoj formi sa zvaničnog Java sajta. Dokumentacija je napisana u formatu pogodnom za čitanje u nekom brauzeru, tako da se jednostavnim izborom hiperveza u tekstu može brzo pronaći ono što se traži. Na slici 2.6 je prikazana početna strana Java dokumentacije u Internet Exploreru.

Pored toga što u programima olakšavaju nalaženje i upotrebu klasa iz Java API, paketi imaju još dve dobre strane:



Slika 2.6: Početna strana dokumentacije za Java API.

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

Prilikom definisanja nove klase u Javi, na jednostavan način se mogu koristi druge klase samo iz istog paketa. Klase iz drugog paketa se mogu koristiti uz navođenje punog imena klase koje se sastoji od dva dela, imena paketa i imena klase, razdvojena tačkom. Na primer, za rad sa datumima postoji klasa `Date` koja se nalazi u paketu `java.util`. Ako je potrebna, ta klasa se u programu može bez ograničenja upotrebiti pod imenom `java.util.Date`.

Međutim, upotreba dugačkog imena klase zahteva više pisanja, povećava mogućnost slovnih grešaka i smanjuje čitljivost programa. Zbog toga postoji mogućnost „uvoženja" pojedinih klasa na početku programa. Iza toga se u programu može koristiti jednostavno ime klase bez imena paketa

u kome se ona nalazi. Ova mogućnost se postiže pisanjem deklaracije import na početku teksta klase. Na primer, ako želimo da definišemo klasu MojaKlasa koja koristi objekat klase Date, umesto da pišemo:

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

možemo kraće pisati:

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

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

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

U stvari, to je način koji se najčešće koristi u Java programima. Napomenimo da se time ne povećava veličina programa, odnosno sve klase iz nekog paketa se fizički ne dodaju programu. Deklaracija import govori samo 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.

U mogućem (retkom) slučaju kada dve različite klase u dva paketa imaju isto ime, skraćivanje imena klasa nije moguće. Tada ime jedne klase

možemo pisati u skraćenom obliku (njenim „uvoženjem" bilo direktno ili preko džoker-znaka), dok za drugu klasu moramo koristiti puno ime.

Na kraju, naglasimo da se osnovni paket `java.lang` automatski „uvozi". To znači da se podrazumeva da se na početku teksta Java koda uvek nalazi deklaracija `import java.lang.*`.

Definisanje paketa

Svaka klasa u Javi mora pripadati nekom paketu. Kada definišemo neku klasu, ako se drugačije ne navede, klasa će pripadati jednom podrazumevanom paketu bez imena. To je takozvani anonimni paket kojem automatski pripadaju sve klase koje nisu eksplicitno dodate nekom imenovanom paketu.

Ako želimo da klasu dodamo imenovanom paketu, koristimo deklaraciju `package` kojom se definiše paket kome pripada klasa. Ova deklaracija se navodi na samom početku teksta klase (čak pre deklaracije `import`). Na primer, ako želimo da klasa `MojaKlasa` bude deo paketa `mojpaket`, to definišemo na sledeći način:

```
package mojpaket;
import java.util.*;
class MojaKlasa { ... }
```

Navođenje imenovanog paketa kojem pripada klasa zahteva posebnu organizaciju datoteka sa tekstom klase. Pored toga, prevođenje klase koja pripada imenovanom paketu se ne može obaviti iz direktorijuma u kojem se nalazi datoteka sa tekstom te klase.

U prethodnom primeru, datoteka `MojaKlasa.java` koja sadrži tekst klase `MojaKlasa` se mora nalaziti u posebnom direktorijumu čije ime je `mojpaket`. U opštem slučaju, ime posebnog direktorijuma mora biti isto kao ime paketa kojem klasa pripada. Pored toga, prevođenje i izvršavanje klase `MojaKlasa` se mora obaviti iz direktorijuma na prvom prethodnom nivou od direktorijuma `mojpaket`.

U grafičkom razvojnom okruženju se o svim ovim detaljima brine odgovarajući softver. U tekstualnom razvojnom okruženju programer nema pomoć i sâm mora voditi računa o ovome.

2.4 Logička organizacija programa

Pre nego što možemo govoriti o detaljima Java programiranja, treba da se upoznamo sa osnovnim pravilima jezika Java koja se moraju poštovati. Pre svega, svi programski elementi u Javi, osim komentara i deklaracija `package` i `import`, moraju se nalaziti u okviru neke klase. To razlikuje Javu od drugih (objektno orijentisanih) jezika kod kojih neki elementi mogu postojati izvan definicije klase.

Dalje, na logičkom nivou, Java program se sastoji od jedne klase ili više njih. Pri tome, izvorni tekst svake klase se piše u posebnoj datoteci čije ime mora biti isto kao ime klase koja se definiše. Naposletku, ekstenzija imena te datoteke koja sadrži neku klasu mora biti `.java`.

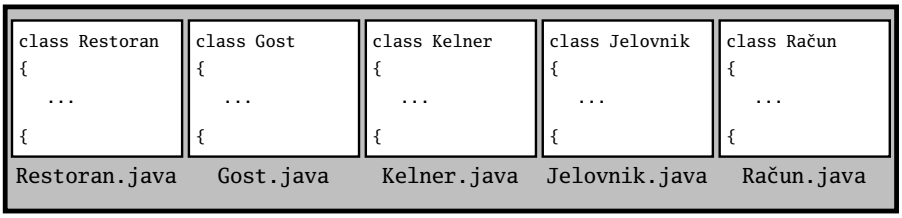
Razmotrimo jedan konkretan primer Java programa koji simulira rad nekog restorana. Pretpostavimo da smo u programu predvideli pet klasa čiji objekti međusobnim manipulisanjem obezbeđuju programsko simuliranje rada restorana:

`Restoran` `Gost` `Kelner` `Jelovnik` `Račun`

Tada se ceo program sastoji od pet datoteka, jer definicije ovih klasa treba da budu u u posebnim datotekama:

- `Restoran.java`
- `Gost.java`
- `Kelner.java`
- `Jelovnik.java`
- `Račun.java`

Slikoviti izgled ovog programa je prikazan na slici 2.7.



SLIKA 2.7: Java program od pet klasa za simuliranje rada restorana.

Dalji postupak prevođenja i izvršavanja tog programa objasnićemo za tekstualno razvojno okruženja. U slučaju kada radimo u grafičkom razvojnom okruženju se primenjuje sličan postupak, ali na pojednostavljen način zavisno od konkretnog softvera. Dakle, svih pet klasa moramo

prevesti u Java bajtkod pomoću Java prevodioca. Kao što znamo iz odeljka 1.3, komande za to su:

- `javac Restoran.java`
- `javac Gost.java`
- `javac Kelner.java`
- `javac Jelovnik.java`
- `javac Račun.java`

Time se dobija Java bajtkod ovih klasa u odgovarajućim datotekama:

- `Restoran.class`
- `Gost.class`
- `Kelner.class`
- `Jelovnik.class`
- `Račun.class`

Najzad, izvršavanje celog programa se sastoji od interpretiranja bajkoda logički početne klase programa. Ako pretpostavimo da je u ovom primeru to klasa `Restoran`, tada se komandom

```
java Restoran
```

započinje izvršavanje programa. Naravno, ovim se interpretira bajtkod klase `Restoran`, a prema logici programa će se onda po potrebi u odgovarajućem trenutku interpretirati bajtkodovi ostalih klasa programa.

Primetimo da se kao argument Java prevodioca mora navesti puno ime datoteka sa ekstenzijom `.java`, dok se kao argument Java interpretatora *ne* sme navesti ekstenzija `.class` datoteke koja sadrži bajtkod željene klase.

2.5 Prvi Java program

Posvetimo sada malo više pažnje programu *Zdravo* koji smo koristili u odeljku 1.3 kao radni primer za opis razvojnih okruženja. Sada ćemo na tom programu pokazati ono što smo već naučili o Javi, ali i pokušati da ga bolje razumemo kroz neke uvodne napomene o jeziku Java. Ipak i dalje nemojte očekivati da ćete baš sve razumeti o tom programu — naizgled trivijalan, program koristi neke elemente o kojima ćemo detaljno govoriti tek kroz nekoliko poglavlja ove knjige.

Da bismo mogli da lakše pratimo program *Zdravo*, najpre na ovom mestu ponavljamo njegov tekst.

LISTING 2.1: Program *Zdravo*.

```
// Prvi Java program

import java.util.*;
public class Zdravo
{
   public static void main(String[] args)
   {
      Scanner tastatura = new Scanner(System.in);
      System.out.print("Kako se zovete: ");
      String ime = tastatura.nextLine();
      System.out.print("Koliko imate godina: ");
      int god = tastatura.nextInt();
      System.out.println("Zdravo " + ime + "!");
      System.out.println(god + " su najlepše godine.");
   }
}
```

Prvi red programa *Zdravo* počinje simbolom //. Takva konstrukcija se u Javi naziva ***komentar*** i potpuno se preskače prilikom prevođenja programa. Za računar dakle komentari kao da ne postoje, ali to ne znači da su oni nevažni. Komentari se pišu za ljude i služe za objašnjenje delova programa drugim programerima. Bez komentara je vrlo teško razumeti kako radi neki program, a to je naravno neophodno za one koji će menjati program. Pisanje dobrih i kratkih komentara je odlika vrhunskih programera i nikako se ne sme olako shvatiti.

Iza reda sa komentarom u programu *Zdravo* uočavamo deklaraciju `import` kojom se programu *Zdravo* dodaju sve klase iz paketa `java.util`. Ovo je potrebno zbog toga što se u programu koristi klasa `Scanner` iz tog paketa. Klasa `Scanner` služi za pretvaranje otkucanih znakova na tastaturi u tipove podataka koje Java razume. U programu se koristi i klasa `System` iz paketa `java.lang`, ali se za taj paket ne mora pisati deklaracija `import` pošto se to podrazumeva i bez posebnog navođenja. Klasa `System` služi za povezivanje programskih ulaznih podataka sa standardnim ulazom (tastaturom) i programskih izlaznih podataka sa standardnim izlazom (monitorom).

Sve iza deklaracije `import` u programu *Zdravo* je definicija jedine klase od koje se taj program sastoji. Znamo da se u opštem slučaju Java program sastoji od više klasa, ali ovaj jednostavni program predstavlja

degenerativni slučaj. Definicija neke klase u Javi počinje službenom rečju `class` i imenom klase, a zatim se telo klase navodi između otvorene i zatvorene vitičaste zagrade, { i }. U telu klase se definišu promenljive (polja) i metodi koje sadrže svi objekti definisane klase.

U programu *Zdravo* vidimo da je definisana klasa pod nazivom `Zdravo`. (Ispred reči `class` se nalazi druga službena reč `public` koja predstavlja modifikator pristupa i ukazuje da se objekti definisane klase mogu konstruisati u bilo kojoj drugoj klasi.) Svaka klasa međutim ne čini program, odnosno nije moguće svaku klasu početno izvršiti primenom Java interpretatora. Da bi to bilo moguće, klasa mora da sadrži specijalni metod (proceduru) `main()` u obliku:

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

Prvi red definicije metoda `main()` je u ovom trenutku teško objasniti i razumeti — za sada je najbolje sve njegove elemente shvatiti kao obavezne. Iza toga između otvorene i zatvorene vitičaste zagrade dolazi telo metoda `main()`, u kojem se navodi niz naredbi koje će se izvršiti kada se metod `main()` pozove za izvršavanje. A to se implicitno radi kada se komandom

```
Java Zdravo
```

želi izvršiti program *Zdravo*. Preciznije, ovom komandom počinje interpretiranje bajtkoda klase `Zdravo` automatskim pozivom njenog metoda `main()` za izvršavanje. To znači da se onda redom izvršavaju naredbe u telu tog metoda.

U telu metoda `main()` se svaka naredba nalazi u posebnom redu i završava tačkom-zapetom. U Javi, svaka prosta naredba se mora završiti tačkom-zapetom. To je zapravo način na koji se proste naredbe razlikuju, a ne po tome da li se nalaze u posebnim redovima ili ne. Naime, u Javi je dopušteno da se više naredbi pišu u jednom redu. Složene (kompozitne) naredbe se grade od prostih i složenih naredbi prema specijalnim pravilima o kojima ćemo detaljno govoriti u poglavlju 4.

Postupak koji se primenjuje u metodu `main()` klase `Zdravo` predstavlja tipični postupak za interaktivni ulaz/izlaz programa. Pošto se sličan pristup može iskoristiti i u drugim programima, sada ćemo ukratko

objasniti naredbe kojima je to realizovano u ovom primeru. U Javi se podaci na ekranu najčešće prikazuju dvema naredbama:

- `System.out.print( ... )`
- `System.out.println( ... )`

Podaci koji se žele prikazati na ekranu se navode kao argumenti u zagradama umesto tri tačke, a u `println` slučaju se nakon njihovog prikazivanja kursor dodatno pomera u sledeći red na ekranu. Na primer, naredbom

```
System.out.print("Kako se zovete: ")
```

prikazuje se tekst "`Kako se zovete: `" od trenutne pozicije kursora na ekranu i kursor se postavlja neposredno iza toga. Slično, naredbom

```
System.out.println("Zdravo " + ime + "!")
```

prikazuje se odgovarajući tekst i kursor se pomera na početak sledećeg reda ekrana.

Učitavanje ulaznih podataka programa u Javi nije tako jednostavno i zahteva primenu objektno orijentisanih tehnika. Zato se prvom naredbom metoda `main()` u klasi `Zdravo`

```
Scanner tastatura = new Scanner(System.in)
```

najpre konstruiše (operatorom `new`) objekat klase `Scanner` povezan sa ***standardnim ulazom***, koji se u Javi predstavlja objektom `System.in`. Standardni ulaz je apstraktni model učitavanja podataka preko fizičkog uređaja tastature, pa smo novokonstruisani objekat nazvali `tastatura`. Primenom raznih metoda klase `Scanner` na ovaj objekat se mogu zatim učitavati vrednosti odgovarajućeg tipa. Tako se primenom metoda `nextLine()` na objekat `tastatura` u naredbi

```
String ime = tastatura.nextLine()
```

učitava niz znakova od kojih se sastoji ime korisnika, dok se primenom metoda `nextInt()` na objekat `tastatura` u naredbi

```
int god = tastatura.nextInt()
```

učitava ceo broj koji predstavlja broj godina korisnika.

Na kraju treba još naglasiti da je Java programski jezik ***slobodnog formata***. To znači da ne postoje ograničenja na izgled samog programa,

odnosno na način na koji se pojedine sastavne konstrukcije pišu u okviru programa. 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 jednom 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, i tako dalje.

Na primer, sa gledišta Java prevodioca, prihvatljiv izgled programa *Zdravo* je i ovakav njegov oblik:

```
// Prvi Java program
    import java.util.*;  public class Zdravo {    public static
void
       main(String[] args) {Scanner tastatura = new
Scanner(System.in); System.out.print("Kako se zovete: ");
  String ime = tastatura.nextLine(); System.out.print
("Koliko imate godina: ");int god=tastatura.nextInt();System.out.
println(
   "Zdravo " + ime + "!"); System.out.println
                (god
+
             " su najlepše godine.");}}
```

Naravno, ova „ispravna" verzija programa *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 i, generalno, da se struktura programa učini što jasnijom za čitaoce.

Uz pridržavanje ovih preporuka i upotrebu komentara, razumljivost komplikovanih programa se može znatno poboljšati. Obratite pažnju na to da je ovo važno ne samo za čitaoce, već i za autore programa, jer posle kratkog vremena će i sâm autor lako zaboraviti način na koji je neki

problem prvobitno rešen. Zato ukoliko je kasnije potrebno modifikovati neki program, treba se podsetiti i ponovo tačno razumeti kako on radi.

Glava

3

Osnovni elementi jezika Java

Od ovog poglavlja počinjemo sistematsko izučavanje programskog jezika Java. Ali da bismo odmah mogli pisati najjednostavnije programe u Javi, moramo se upoznati sa osnovnim programskim elementima koji su neophodni za pisanje kompletnih programa. Pošto slični koncepti postoje u svakom programskom jeziku, čitaoci koji poznaju neki drugi programski jezik imaće manje teškoća u savladavanju materijala ovog poglavlja.

3.1 Imena

Programiranje se u svojoj suštini svodi na davanje imena. Ime u programu može da označava razne stvari: memorijsku lokaciju, datoteku na disku čiji je sadržaj dokument, slika ili zvučni zapis, ali neko ime može biti zamena i za apstraktne koncepte kao što su niz naredbi (metod) ili novi tip podataka (klasa). Zbog svoje važnosti u programiranju, imena imaju poseban tehnički termin i zovu se ***identifikatori***.

Imena se ne mogu davati proizvoljno, već postoje striktna pravila kako se ona grade. U Javi se ime sastoji od niza znakova, jednog ili više njih. Ovi znakovi od kojih se sastoji ime moraju biti slova, cifre ili donja crta (_). Pored toga, ime mora početi slovom ili donjom crtom.[1] Na primer, neka ispravna imena u Javi su:

[1] U stvari, identifikator u Javi može sadržati i znak za dolar ($), kao i počinjati s tim znakom, ali se upotreba tog znaka ne preporučuje.

- `tastatura`
- `brojStudenata`
- `n`
- `broj_studenata`
- `Zdravo`
- `N`
- `x15`
- `jedno_VRLO_dugačko_ime`

Treba naglasiti da razmaci nisu dozvoljeni u imenima. Tako, na primer, ispravno ime u Javi je `broj_studenata`, ali `broj studenata` nije ispravno. Velika i mala slova se razlikuju u Javi tako da se `Zdravo`, `zdravo`, `ZDRAVO` i `zdRaVO` smatraju različitim imenima. Izvesne reči su rezervisane za specijalnu namenu u Javi tako da se one ne mogu koristiti za imena koja daje programer. U ove ***službene*** (ili ***rezervisane***) reči spadaju `import`, `class`, `public`, `static`, `if`, `else`, `while`, sve ukupno oko pedesetak takvih reči. Službene reči u programima u ovoj knjizi su naglašene podebljanim slovima.

Pored ovih formalnih pravila koje propisuje sâm jezik Java, postoje preporuke o tome kako treba birati dobra imena u programima. Pre svega, imena trebaju biti smislena da bismo na osnovu njih mogli lako zaključiti šta označavaju. U programu *Zdravo* na primer, objekat za učitavanje ulaznih podataka nazvali smo `tastatura`, jer to zaista i predstavlja, mada smo ga mogli zvati i `xy` ili `bzvz`. Davanje smislenih imena znatno doprinosi boljoj čitljivosti programa.

Dodatna konvencija koje se pridržavaju Java programeri radi bolje čitljivosti programa je: imena klasa počinju velikim slovom, imena promenljivih i metoda počinju malim slovom, a imena konstanti se sastoje od svih velikih slova. Kada se ime sastoji od nekoliko reči, recimo `brojStudenata`, onda se svaka reč od druge piše sa početnim velikim slovom (a i prva ako se radi o klasi). Java programeri uglavnom ne koriste donju crtu u imenima, osim u nekim izuzetnim slučajevima i to samo na prvom mestu.

Što se tiče upotrebe imena u programu radi označavanja programskih elemenata, često se koriste ***složena imena*** koja se sastoje od nekoliko običnih imena razdvojenih tačkama. Složena imena se ponekad nazivaju i ***kvalifikovana imena***, a notacija kojom se pišu se naziva ***tačka-notacija***. Već smo videli primer ove vrste imena u programu *Zdravo*: `System.out.println`.

Složena imena su potrebna zato što u Javi neke stvari mogu da obuhvataju druge stvari. Složeno ime je onda kao putokaz do neke stvari kroz jedan ili više nivoa obuhvatanja. Tako ime `System.out.println` ukazuje

da nešto pod nazivom `System` sadrži nešto pod nazivom `out` koje sadrži nešto pod nazivom `println`.

3.2 Tipovi podataka i literali

Programi manipulišu podacima koji se nalaze u delu programa predviđenom za njihovo čuvanje. Vrste podataka sa kojima Java programi mogu da rade se dele u dve glavne kategorije: one tipove podatka koji su unapred „ugrađeni" u jezik i nove tipove podataka koje programer može sâm da definiše. Prva grupa tipova podataka se naziva ***primitivni tipovi podataka***, dok u drugu grupu spadaju ***klasni tipovi podataka***.

Primitivnim tipovima podataka su obuhvaćeni osnovni podaci u Javi koji nisu objekti. Tu spadaju celi brojevi, realni brojevi, alfabetski znakovi i logičke vrednosti. Svi primitivni tipovi podataka imaju tačno određenu veličinu memorijske lokacije u bajtovima za zapisivanje odgovarajućih vrednosti. Odatle onda sledi da svi primitivni tipovi podataka imaju tačno definisan opseg vrednosti koje im pripadaju.

U Javi postoji osam primitivnih tipova podataka čija imena su `byte`, `short`, `int`, `long`, `float`, `double`, `char` i `boolean`. Prva četiri tipa služe za rad sa celim brojevima (brojevima bez decimala kao što su 17, -1234 i 0) i razlikuju se po veličini memorijske lokacije koja se koristi za binarni zapis celih brojeva:

- Tip `byte` koristi jedan bajt (8 bitova) za binarni zapis celih brojeva. Prema tome, vrednosti ovog tipa su celi brojevi u opsegu od -2^8 do $2^8 - 1$ (ili od -128 do 127), uključujući ove krajnje granice.
- Tip `short` koristi dva bajta (16 bitova) za binarni zapis celih brojeva. Vrednosti ovog tipa su celi brojevi u opsegu od -2^{16} do $2^{16} - 1$ (ili od -32768 do 32767).
- Tip `int` koristi četiri bajta (32 bita) za binarni zapis celih brojeva. Vrednosti ovog tipa su celi brojevi u opsegu od -2^{32} do $2^{32} - 1$ (ili od -2147483648 do 2147483647).
- Tip `long` koristi osam bajtova (64 bita) za binarni zapis celih brojeva. Vrednosti ovog tipa su celi brojevi u velikom opsegu od -2^{64} do $2^{64} - 1$ (ili od -9223372036854775808 do 9223372036854775807).

Ove detaljne činjenice naravno ne morate znati napamet, nego služe samo da dobijete osećaj o magnitudi celih brojeva sa kojima možete raditi

u programu. U najvećem broju slučajeva se koristi tip `int` kao srednje rešenje koje pokriva većinu namena.

Tipovi `float` i `double` služe za predstavljanje realnih brojeva (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 `float` koristi četiri bajta za binarni zapis realnih brojeva. Vrednosti ovog tipa su realni brojevi u opsegu $\pm 10^{38}$ i mogu imati oko 8 značajnih cifara. (To znači da bi ovim tipom brojevi 32.3989231134 i 32.3989234399 bili predstavljeni istim brojem 32.398923.)
- Tip `double` koristi osam bajtova za binarni zapis realnih brojeva. Vrednosti ovog tipa su realni brojevi u opsegu $\pm 10^{308}$ i mogu imati oko 15 značajnih cifara.

Sem u posebnim slučajevima kada je važna ušteda memorijskog prostora, u programima se obično koristi tip `double` za rad sa realnim brojevima.

Tip `char` koristi dva bajta za binarni zapis alfabetskih znakova. Vrednosti ovog tipa su pojedinačni znakovi kao što su mala i velika slova (`A`, `a`, ...), cifre (0, 1, ...), znakovi punktuacije (*, ?, razmak, ...) i neki specijalni znakovi (za novi red, za tabulator, ...). Način na koji se ovi znakovi predstavljaju u binarnom obliku se naziva Unicode, a ovaj standard je izabran u Javi zbog internacionalizacije. Od samog nastanka, cilj jezika Java je bio pisanje programa za razne govorne jezike (a ne samo za engleski). Standard Unicode je idealan iz tog aspekta, jer se u njegovoj šemi mogu predstaviti znakovi iz praktično svih pisama u današnjoj upotrebi na svetu.

Za pisanje vrednosti tipa `char` u programu se mora željeni znak staviti pod jednostruke apostrofe: na primer, `'A'`, `'3'` ili `'*'`. Bez ovih apostrofa, `A` bi se razumeo kao identifikator, 3 kao ceo broj tri i * kao znak za množenje. Jednostruki apostrofi nisu deo vrednosti tipa `char`, nego su samo notacija za pisanje konkretnih znakova tog tipa.

Konkretne vrednosti bilo kog tipa se u programu nazivaju ***literali***. Literali su dakle način za pisanje konkretnih vrednosti nekog tipa u programu. Pomenuli smo da se literali znakovnog tipa `char` zapisuju sa jednostrukim apostrofima. Neki specijalni znakovi tog tipa se pišu sa

obrnutom kosom crtom (\) na početku. Na primer, znak za novi red se piše `'\n'`, znak za tabulator se piše `'\t'`, a sâm znak za obrnutu kosu crtu se piše `'\\'`. Primetimo da iako pišemo dva znaka između apostrofa u ovim specijalnim slučajevima, predstavljena vrednost ovim literalima je samo jedan znak tipa `char`.

Numerički literali, odnosno pisanje brojeva u Java programu je složenije nego što bismo to mogli očekivati. Brojeve naravno možemo pisati na uobičajeni način na koji smo navikli (uz malu razliku da se koristi tačka umesto decimalnog zareza): na primer, 107, `23.55` ili `-0.5`. Ali postoje i druge mogućnosti koje su ponekad bolje od svakodnevnog načina i treba ih poznavati.

Realni brojevi se u Javi mogu pisati u eksponencijalnom obliku: na primer, `0.32e4` ili `12.345e-78`. Delovi `e4` i `e-78` ovih primera označavaju stepen broja 10 tako da `0.32e4` označava realni broj $0.32 \cdot 10^4 = 3200.0$ i `12.345e-78` je zapravo broj $12.345 \cdot 10^{-78}$. Ovaj eksponencijalni način pisanja brojeva je pogodan za zapis vrlo velikih ili vrlo malih realnih brojeva.

Svaki numerički literal koji sadrži decimalnu tački ili eksponent, tj. predstavlja realni broj, po automatizmu se smatra vrednošću tipa `double`.[2] Da bismo ovo promenili tako da navedeni realni broj bude vrednost tipa `float`, na kraju literala treba dodati veliko slovo `F` ili malo slovo `f`. Na primer, `1.2F` ili `1.2f` označava realni broj 1.2 tipa `float`.

Čak i za celobrojne literale nije sve tako jednostavno. Obični celobrojni literali kao što su 6945 ili -32 predstavljaju cele brojeve tipa `byte`, `short` ili `int` zavisno od njihove veličine. Ako se želi ceo broj tipa `long`, mora se dodati slovo `L` na kraju: na primer, `6945L` ili `-32L`.

Dodatna mogućnost je izražavanje celih brojeva u oktalnom ili heksadekadnom zapisu. U oktalnom brojnom sistemu (sa osnovom 8), koriste se cifre od 0 do 7: na primer, broj 17 u oktalnom zapisu je (uobičajeni) broj 15 u dekadnom zapisu (sa osnovom 10). U Javi, numerički literal koji počinje cifrom 0 se smatra brojem u oktalnom zapisu. Na primer, literal 045 predstavlja (dekadni) ceo broj 37, a ne 45.

[2] Ovo pravilo je uzrok glupih grešaka u Javi kod pisanja `float x = 1.2`. Kako je broj 1.2 tipa `double` i promenljiva `x` tipa `float`, a uz to automatsko pretvaranje vrednosti tipa `double` u tip `float` nije dozvoljeno, napisana dodela vrednosti nije ispravna. Ispravno je `float x = 1.2f`.

U heksadekadnom brojnom sistemu (sa osnovom 16), koriste se cifre od 0 do 9 i slova A, B, C, D, E, F (ili odgovarajuća mala slova). Ova slova predstavljaju brojeve od 10 do 15. Na primer, broj 2A u heksadekadnom zapisu je (uobičajeni) broj 42 u dekadnom zapisu. U Javi, celobrojni literali u heksadekadnom zapisu počinju sa 0x ili 0X: na primer, 0x2A ili 0xFF70.

Heksadekadni zapis se može koristiti i kod znakovnih literala za označavanje proizvoljnih Unicode znakova. Ovi literali počinju sa \u i sadrže još četiri heksadekadne cifre. Na primer, znakovni literal '\uC490' označava slovo "Đ".

Poslednji primitivni tip podataka `boolean` služi za predstavljanje samo dve logičke vrednosti: tačno i netačno. Literali kojima se u Javi označavaju ove vrednosti su `true` i `false` (bez apostrofa). Logičke vrednosti u programima se najčešće dobijaju kao rezultat izračunavanja relacijskih operacija.

Jedna karakteristika primitivnih tipova podataka je da se njima predstavljaju proste vrednosti (brojevi, znakovi i logičke vrednosti) koje se ne mogu dalje deliti. Druga karakteristika je da su nad njima definisane različite operacije koje se mogu primenjivati nad vrednostima određenog tipa. Na primer, za vrednosti numeričkih tipova (celi i realni brojevi) su definisane uobičajene aritmetičke operacije sabiranja, oduzimanja, množenja i deljenja. Oznake ovih operacija u Javi su uobičajene: +, −, * i /.

Treba naglasiti da operacija deljenja celih brojeva kao rezultat daje opet ceo broj (tj. decimalni deo rezultata se odbacuje): na primer, kao rezultat izraza 17/5 se dobija broj 3, a ne 3.4. U Javi postoji i operacija izračunavanja ostatka pri celobrojnom deljenju koja se označava znakom procenta %. Na primer, kao rezultat izraza 17 % 5 se dobija broj 2, odnosno ostatak pri celobrojnom deljenju 17/5.

Za sve primitivne tipove podataka su definisane relacijske operacije kojima se mogu upoređivati dve vrednosti jednog istog tipa. Šest standardnih relacijskih operacija i njihove oznake u Javi su:

- manje od: <
- manje ili jednako: <=
- veće od: >
- veće ili jednako: >=
- jednako: ==
- nejednako: !=

Obratite pažnju na to da se za operaciju provere jednakosti pišu za-

pravo dva znaka jednakosti (==). Isto tako, svi relacijski operatori kao rezultat daju logičku vrednost tačno ili netačno. Na primer, rezultat izraza 17 != 5 je logička vrednost tačno, dok 'A' == 'a' kao rezultat daje netačno.

Svi ostali tipovi podataka u Javi pripadaju kategoriji klasnih tipova podataka kojima se predstavljaju objekti. Objekti su ,,složeni" podaci po tome što se sastoje od sastavnih delova kojima se može nezavisno manipulisati. O klasnim tipovima podataka biće mnogo više reči u nastavku knjige. Za sada ćemo pomenuti jedan važan, unapred definisan klasni tip: String.

Objekti tipa String se nazivaju ***stringovi*** (ili ***niske***) i predstavljaju nizove znakova. Literale kojima se označavaju stringovi smo već koristili u programu *Zdravo* iz odeljka 2.5:

```
"Kako se zovete: "
"Koliko imate godina: "
```

Dvostruki apostrofi na početku i na kraju niza znakova (stringa) su neophodni kod pisanja string literala, ali ti apostrofi ne pripadaju stringu. Broj znakova u stringu je dužina stringa koja može biti nula ili više. Na primer, dužina praznog stringa je nula (taj string se označava literalom " "), a dužina stringa "ovo je string" je 13 (razmak se naravno računa kao regularan znak).

Unutar string literala se mogu koristiti specijalni znakovi \n, \t, \\ i ostali, kao i Unicode znakovi koji počinju sa \u (recimo \uC490). Ako je dvostruki apostrof deo stringa, on se mora se pisati u obliku \". String, na primer,

```
On reče: "Zdravo svima!"
```

sa znakom za novi red na kraju, označava se literalom:

```
"On reče: \"Zdravo svima!\"\n"
```

Nad stringovima je dozvoljena operacija ***spajanja*** (ili ***konkatenacije***) stringova. Oznaka za tu operaciju u Javi je znak +, jer podseća na ,,sabiranje" stringova. Na primer, rezultat izraza

```
"Java" + "programiranje"
```

jeste string "Javaprogramiranje". U opštem slučaju, rezultat operacije spajanja dva stringa je novi string koji se sastoji od znakova prvog stringa

iza kojih se dodaju znakovi drugog stringa. Primetimo da se razmaci ne dodaju automatski, nego se to mora postići spajanjem stringa koji se sastoji od znaka za razmak na odgovarajućem mestu. Na primer,

```
"Java" + " " + "programiranje" + " je " + "zabavno"
```

kao rezultat daje string "Java programiranje je zabavno".

Pošto su stringovi u Javi pravi objekti (a ne proste vrednosti), oni poseduju mnogo više mogućnosti za rad sa njima, ali i osobenosti svojstvene objektima u opštem slučaju. O tome se mnogo više govori u nastavku ove knjige.

3.3 Promenljive

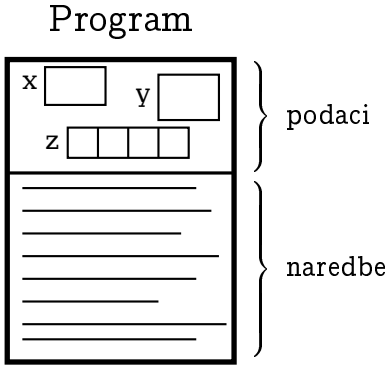
Pomenuli smo da programi manipulišu podacima i u prethodnom odeljku smo upoznali osnovne tipove podataka koji se mogu koristiti u Javi. Sada je vreme da detaljnije upoznamo način na koji se sa takvim podacima može manipulisati.

Podaci programa se čuvaju u memorijskim ćelijama u posebnom delu programa. Programer ne mora (a i ne može) da vodi računa o tačnim memorijskim mestima (adresama) gde se nalaze podaci, već se za memorijske ćelije koriste simbolička imena preko kojih se u programu ukazuje na podatke u njima. Ova imena kojima su u programu pridruženi podaci se nazivaju ***promenljive***.

Promenljivu je najbolje zamisliti kao jednu kutiju u koju se mogu čuvati podaci koji se koriste u programu. Promenljiva direktno upućuje na tu kutiju i samo indirektno na podatke koje se nalaze u kutiji. Pošto se podaci u kutiji mogu menjati tokom izvršavanja programa, promenljiva može ukazivati na različite podatke u različitim trenucima izvršavanja programa, ali se promenljiva uvek odnosi na istu kutiju.[3] Ponekad ipak može doći do zabune, jer kada se promenljiva koristi u programu na jedan način, ona ukazuje na kutiju, a kada se koristi na drugi način, ona ukazuje na podatke u kutiji.

Dopunjena predstava računarskog programa, sa delom za podatke u kojem promenljive služe kao ćelije za čuvanje podataka, prikazana je na slici 3.1.

[3] Ova činjenica da promenljiva može ukazivati na promenljive podatke tokom izvršavanja programa je razlog za termin *promenljiva*.



SLIKA 3.1: Slika računarskog programa sa tri promenljive x, y i z.

Jedini način u Javi da se dodeli neka vrednost promenljivoj — to jest, da se neki podatak „stavi" u kutiju koju promenljiva predstavlja — jeste pomoću ***naredbe dodele***. Naredba dodele ima opšti oblik:

```
ime = izraz;
```

gde *ime* predstavlja ime odgovarajuće promenljive, a *izraz* je konstrukcija kojom se navodi ili izračunava neka vrednost. Kada se izvršava neka naredba dodele prilikom izvršavanja programa, najpre se izračunava *izraz* i zatim se dobijena vrednost dodeljuje promenljivoj nazvanoj *ime*. Na primer, razmotrimo konkretnu naredbu dodele:

```
kamatnaStopa = 0.08;
```

U ovom slučaju, *ime* je kamatnaStopa i *izraz* je literal 0.08. To znači da se ova naredba dodele izvršava tako što se broj 0.08 dodeljuje promenljivoj kamatnaStopa (tj. broj 0.08 se upisuje u kutiju sa imenom kamatnaStopa, zamenjujući bilo koji njen stari sadržaj).

Razmotrimo sada malo složeniju naredbu dodele, koja se možda nalazi negde dalje u istom programu:

```
kamata = kamatnaStopa * kredit;
```

Ovde se izraz kamatnaStopa * kredit dodeljuje promenljivoj kamata. Na desnoj strani znaka =, imena kamatnaStopa i kredit predstavljaju promenljive, a znak * predstavlja operaciju množenja. Zato se ova naredba dodele izvršava tako što se najpre izračunava izraz na desnoj strani znaka jednakosti. To znači da se množe *vrednosti* koje se trenutno nalaze

u promenljivim kamatnaStopa i kredit. Rezultat toga se zatim upisuje u promenljivu kamata koja se nalazi na levoj strani znaka jednakosti.

Obratite pažnju na to da kada se promenljiva pojavljuje u izrazu, onda se umesto te promenljive za izračunavanje izraza zamenjuje vrednost koja se trenutno nalazi u promenljivoj. U ovom slučaju, promenljiva se odnosi na vrednost koja se trenutno nalazi u kutiji nazvanoj po toj promenljivoj. Sa druge strane, kada se promenljiva pojavljuje na levoj strani znaka jednakosti u naredbi dodele, vrednost izraza na desnoj strani znaka jednakosti se dodeljuje toj promenljivoj. U ovom slučaju, promenljiva se odnosi na kutiju nazvanoj po toj promenljivoj u koju treba upisati vrednost.

Naglasimo još nešto važno u vezi sa naredbom dodele. Znak jednakosti koji se pojavljuje u naredbi dodele služi samo da razdvoji dva sastavna dela ove naredbe. Znak jednakosti ovde ima potpuno drugačije značenje od onog u matematici, gde označava činjenicu da su dva elementa jednaka. Na primer, računar izvršava naredbu dodele x = 1 tako što broj 1 dodeljuje promenljivoj x. Sličan zapis u matematici $x = 1$ označava činjenicu da je element x jednak broju 1. Da bismo bolje uočili razliku, naglasimo da je potpuno ispravno u programu pisati naredbu dodele oblika x = x + 1. Ona će se izvršiti tako što će trenutna vrednost promenljive x biti uvećana za 1 i taj rezultat izraza na desnoj strani znaka = biće dodeljen (istoj) promenljivoj x koja se nalazi na levoj strani znaka =. Ovom naredbom dodele se prosto prethodna vrednost promenljive x uvećava za 1. Sa druge strane, zapis $x = x + 1$ u matematici je besmislen, jer nikad neki element x ne može biti jednak samom sebi uvećanom za 1.

Promenljiva se može koristiti u programu samo ako je prethodno ***deklarisana*** (ili ***definisana***) u programu. Deklaracijom promenljive se navodi njen tip i njeno ime. Tip promenljive određuje zapravo tip podataka koje promenljiva može sadržati. Ta inofrmacija je potrebna radi rezervisanja memorijske lokacije (kutije) čija veličina je dovoljna za smeštanje podataka odgovarajućeg tipa. Prilikom deklaracije promenljive se može dodatno dodeliti početna vrednost promenljivoj naredbom dodele. Opšti oblik ***naredbe deklaracije promenljivih*** je:

```
tip ime;
```

ili

```
tip ime = izraz;
```

gde je *tip* jedan od primitivnih ili objektnih tipova, *ime* je ime promenljive koja se deklariše, a *izraz* se izračunava da bi se dodelila početna vrednost promenljivoj koja se deklariše. Na primer:

```
int godina;
long n = 0L;
float t;
double x = 3.45;
String imePrezime = "Pera " + "Perić";
```

Ovim se deklarišu promenljive godina, n, t, x i imePrezime odgovarajućeg tipa. To znači da se za svaku od njih rezerviše potreban memorijski prostor i uspostavlja veza između imena i rezervisanog prostora. Promenljivim n, x i imePrezime se dodatno dodeljuju vrednosti navedene iza znaka = u njihovim deklaracijama.

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 x mogu biti samo realni brojevi tipa double — nikad celi brojevi, znakovi, logičke vrednosti ili vrednosti nekog klasnog tipa.

Naredba deklaracije promenljivih može biti, u stvari, malo složenija od oblika koji smo naveli. Naime, iza tipa promenljive se može pisati lista više promenljivih (sa ili bez početnih vrednosti) odvojenih zapetama. Time se prosto deklariše (i inicijalizuje) više promenljivih istog tipa. Na primer:

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

Iako ova mogućnost donosi uštedu u pisanju, jedna od odlika dobrog stila programiranja je da se promenljive posebno deklarišu. Pri tome od ovoga pravila treba odstupiti samo ako je više promenljivh povezano na neki način. U prethodnom primeru to verovatno važi recimo za promenljive ime, srednjeIme i prezime. Druga odlika dobrog stila programiranja je i da se uz deklaracije promenljivih navede komentar kojim se opisuje njihovo značenje u programu. Na primer:

```
double p;  // iznos kredita (principal)
```

```
double k;  // mesečna kamatna stopa
int m;     // broj mesečnih rata
double r;  // iznos mesečne rate
```

Promenjive se mogu deklarisati unutar metoda ili unutar klasa. Kada se deklarišu unutar klasa radi opisa atributa objekata, promenljive se nazivaju ***polja***. Za sada ćemo koristiti samo promenljive u užem smislu kada se deklarišu unutar metoda (tačnije unutar metoda `main()`). 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.

Deklaracije promenljivih se mogu pisati bilo gde unutar metoda, uz poštovanje pravila da svaka promenljiva mora biti deklarisana pre nego što se koristi u nekom izrazu. Iako Java daje veliku slobodu programerima u vezi sa mestom deklarisanja promenljivih, ne postoji opšteprihvaćeni konsenzus oko toga šta je dobar stil programiranja. Neki programeri vole da sve promenljive deklarišu na samom početku metoda. Drugi pak deklarišu promenljive tek kada je to neophodno u okviru metoda, odnosno kada se prvi put koriste. Neki srednji pristup je da se promenljive važne za logiku metoda deklarišu (uz opis šta označavaju) na početku metoda, a sve ,,pomoćne" promenljive tek kada se prvi put koriste.

Primer: pretvaranje Farenhajtovih u Celzijusove stepene

Da bismo primenili programske elemente Jave koje smo naučili do sada, sada ćemo napisati kompletan program kojim se dati stepeni Farenhajta pretvaraju u stepene Celzijusa. Preciznije, ulaz za program je broj stepeni Farenhajta koji se dobija od korisnika, a izlaz je odgovarajući broj stepeni Celzijusa koji se prikazuje na ekranu.

Za pisanje ovog programa je neophodno znati formulu za pretvaranje Farenhajtovih u Celzijusove stepene. (Programeri moraju biti svestrano obrazovani — pored odličnog poznavanja programiranja, oni moraju dobro razumeti i druge oblasti iz kojih rešavaju probleme na računaru!) Ako sa f označimo broj stepeni Farenhajta, onda se odgovarajući broj stepeni Celzijusa c dobija po formuli:

$$c = \frac{5(f - 32)}{9}.$$

Primenom ove formule je sada lako napisati traženi program. U programu se koriste dve celobrojne promenljive koje služe za čuvanje svih podataka programa: promenljiva f za broj stepeni Farenhajta koji korisnik želi da pretvori u stepene Celzijusa i promenljiva c za izračunati broj stepeni Celzijusa po prethodnoj formuli.

Ulazno/izlazni postupak u programu je skoro isti kao u programu *Zdravo*. Učitavanje broja stepeni Farenhajta od korisnika se vrši primenom metoda nextInt() na objekat koji predstavlja tastaturu, dok se prikazivanje izračunatog broja stepeni Celzijusa vrši primenom metoda println() na objekat koji predstavlja ekran. (Ovaj postupak će biti mnogo jasniji, nadamo se, nakon čitanja sledećeg odeljka u knjizi.)

LISTING 3.1: Pretvaranje Farenhajtovih u Celzijusove stepene.

```
import java.util.*;

public class PretvaranjeStepeni {

   public static void main(String[] args) {
      int f;  // broj stepeni Farenhajta
      int c;  // broj stepeni Celzijusa

      // Ulaz programa se dobija preko tastature
      Scanner tastatura = new Scanner(System.in);

      // Učitavanje stepena Farenhajta od korisnika
      System.out.print("Koliko stepeni Farenhajta? ");
      f = tastatura.nextInt();

      // Izračunavanje stepena Celzijusa po formuli
      c = 5*(f - 32)/9;

      // Prikazivanje rezultata na ekranu
      System.out.print(f + " stepeni Farenhajta = ");
      System.out.println(c + " stepeni Celzijusa");
   }
}
```

Obratite pažnju na to da smo znak { u programu pisali na kraju odgovarajućeg reda, a ne kao što smo do sada taj znak pisali sam za sebe u novom redu. Naime, jedna od preporuka dobrog stila Java programairanja

je da se znak { navodi na kraju reda iza kojeg sledi, a odgovarajući znak } da se piše propisno poravnat u novom redu. Ove preporuke ćemo se i ubuduće pridržavati.

Da biste ovaj Java program izvršili na računaru, potrebno je primeniti uobičajeni postupak koji se sastoji od tri koraka:

1. Unošenje teksta programa u računar.
2. Prevođenje unetog programa u njegov izvršni oblik (bajtkod).
3. Izvršavanje (interpretiranje) dobijenog bajtkoda programa.

Ovaj postupak u tekstualnom i grafičkom okruženju smo detaljno opisali u odeljku 1.3. Uz očigledna prilagođavanja, taj postupak se primenjuje za izvršavanje svakog Java programa, pa ga ubuduće nećemo posebno isticati u primerima drugih Java programa.

3.4 Neke korisne klase

Metodi u Javi su potprogrami koji obavljaju neki specifični zadatak. Drugačije rečeno, to je niz naredbi i promenljivih koje predstavljaju neku funkcionalnu celinu sa svojim posebnim imenom. Izvršavanje metoda, odnosno niza naredbi koje metod predstavlja, postiže se navođenjem imena (i argumenata) metoda u tekstu programa na mestu gde je potrebno uraditi zadatak koji metod obavlja. Ovo se naziva ***pozivanje*** metoda i predstavlja samo jedan, lakši aspekt korišćenja metoda u Javi.

Drugi aspekt *definisanja* metoda podleže složenijim pravilima o kojima će biti reči u poglavlju 5. Java sadrži veliki broj standardnih klasa sa unapred napisanim metodima koji se mogu koristiti u programima. Ovi metodi se mogu pozivati bez ikakvog razumevanja kako su oni napisani ili kako oni rade — dovoljno je samo znati šta oni rade. To je, u stvari, prava suština metoda: metod je ,,crna kutija" koja se može koristiti baz poznavanja šta se nalazi unutra.

Klase u Javi imaju zapravo dve različite uloge. Prva uloga klasa je da budu okvir za grupisanje polja (promenljivih) i metoda koji čine neku logičku celinu. Ova polja i metodi se nazivaju ***statički članovi*** klase. Na primer, u svakoj klasi od koje počinje izvršavanje programa se mora nalaziti metod `main()` koji je statički član te klase. Statički član klase se prepoznaje po službenoj reči `static` u definiciji tog člana.

Druga uloga klasa je da budu sredstvo za opis objekata. U ovoj važnijoj ulozi, neka klasa definiše polja i metode koje imaju svi objekti koji pripadaju toj klasi. Klasa na ovaj način definiše tip podataka u tehničkom smislu, a vrednosti tog tipa su objekti definisane klase. Na primer, standardna klasa `String` u jeziku Java definiše tip podataka sa istim imenom `String`, a vrednosti ovog tipa su nizovi znakova. Recimo, jedna vrednost ovog tipa je predstavljena literalom `"Java je kul!"`.

Prema tome, svaki metod pripada ili nekoj klasi ili nekom objektu. Klase sadrže metode koji su njeni statički članovi. Klase takođe opisuju objekte i time indirektno metode koji pripadaju tim objektima.

Ova dvostruka uloga klasa u Javi može biti zbunjujuća za početnike, ali u praksi se ove dve uloge retko mešaju za istu klasu. Tako recimo klasa `String` sadrži nekoliko statičkih metoda, ali prvenstveno služi za definisanje velikog broja metoda koji pripadaju objektima tipa `String`. Sa druge strane, standardna klasa `Math` sadrži samo statičke metode koji obavljaju uobičajene matematičke funkcije.

Da bismo bolje razumeli ove detalje, u nastavku ćemo opisati neke ,,ugrađene" metode i klase koji mogu biti od koristi Java programerima.

Klasa `System`

Klase mogu imati kako statičke metode, tako i statička polja (promenljive). Jedno od statičkih polja klase `System` je `out`. Pošto se ovo polje nalazi u klasi `System`, njegovo puno ime koje se mora koristiti u programima je `System.out`. Polje `System.out` ukazuje na objekat koji u Javi predstavlja ***standardni izlaz***. Standardni izlaz je apstraktni model prikazivanja raznih tipova podataka na fizičkom uređaju ekrana. Zato ovaj objekat standardnog izlaza sadrži metode za prikazivanje vrednosti na ekranu. Jedan od njih je metod `print`. Složeno ime `System.out.print` se dakle odnosi na metod `print` u objektu na koga ukazuje statičko polje `out` u klasi `System`. Potpuno isto važi i za drugi metod `println`: složeno ime `System.out.println` se odnosi na metod `println` u objektu na koga ukazuje statičko polje `out` u klasi `System`.

U odeljku 2.5 smo se upoznali sa osnovnim mogućnostima metoda `print` i `println` za prikazivanje vrednosti na ekranu. Podsetimo se da je njihova jedina funkcionalna razlika u tome što metod `println` dodatno

pomera kursor na početak sledećeg reda ekrana nakon prikazivanja neke vrednosti.

Argument ovih metoda, odnosno vrednosti koje se žele prikazati na ekranu, u programu se navodi u zagradama. Zagrade se pri pozivanju nekog metoda uvek moraju pisati, čak i ako metod nema argumenata: na primer,

```
System.out.println()
```

ukoliko samo želimo da kursor pomerimo za jedan red. Primetimo da zbog toga nikad ne možemo biti u dilemi da li neko složeno ime predstavlja promenljivu ili metod. Ako se iza složenog imena nalazi leva zagrada, onda ono predstavlja metod; u suprotnom slučaju, ono predstavlja promenljivu.

Jedan problem sa metodima `print` i `println` je to što su prikazani brojevi ponekad u formatu koji nije pregledan. Na primer, izvršavanjem naredbi

```
double kamatnaStopa = 20.0/12;  // kamatnaStopa = 1.666···
System.out.println("Mesečna kamata: " + kamatnaStopa);
```

na ekranu bismo dobili sledeći rezultat:

```
Mesečna kamata: 1.6666666666666667
```

Pošto je promenljiva `kamatnaStopa` tipa `double`, njena vrednost se izračunava sa 15–16 decimala i sve se one prikazuju na ekranu. Naravno, u većini slučajeva toliki broj decimala u izlaznim podacima samo smeta, pa je potrebno na neki način imati veću kontrolu nad formatom prikazanih brojeva.

Zato je od verzije Java 5.0 dodat metod `printf` koji ima slične mogućnosti kao ista funkcija u jeziku C. Broj opcija metoda `System.out.printf` za formatizovanje podataka je vrlo velik, ali ćemo ovde pomenuti samo nekoliko. Ako niste C programer, dodatne informacije potražite u dokumentaciji.

Metod `System.out.printf` može imati jedan argumenat ili više njih razdvojenih zapetama. Prvi argument je string (objekat tipa `String`) kojim se određuje format izlaznih podataka. Preostali argumenti predstavljaju vrednosti koje se prikazuju. Ako bismo prethodni primer napisali koristeći metod `printf` umesto `println`:

```
double kamatnaStopa = 20.0/12;  // kamatnaStopa = 1.666···
System.out.printf("Mesečna kamata: %5.2f\n", kamatnaStopa);
```

na ekranu bismo dobili sledeći rezultat:

```
Mesečna kamata:  1.67
```

Željeni izlazni format neke vrednosti se navodi u prvom argumentu metoda printf zapisom koji počinje znakom procenta (%), završava se određenim slovom, a između mogu biti još neke informacije. Slovo na kraju ukazuje na tip podatka koji se prikazuje, pri čemu se d koristi za cele brojeve, f za realne brojeve, s za stringove i tako dalje. Između početnog znaka % i slova na kraju može se navesti minimalan broj mesta za ispis podataka.

Na primer, razlika između %d i %8d je to što se u prvom slučaju prikazuje ceo broj sa onoliko cifara koliko ih ima (računajući i znak minus za negativne brojeve), dok se u drugom slučaju koristi tačno osam mesta. U ovom drugom slučaju, ako broj ima manje od osam cifara, dodaju se prazna mesta ispred cifara broja kako bi se popunilo svih osam mesta; ako broj ima više od osam cifara, koristi se tačno onoliko mesta koliko broj zapravo ima cifara.

U sledećem konkretnom primeru, izvršavanjem naredbi:

```
int n = 23;
System.out.printf("1. broj: %d\n", n);
System.out.printf("2. broj: %8d\n", n);
```

na ekranu dobijamo:

```
1. broj: 23
2. broj:       23
```

Primetimo da se u drugom slučaju dodaju šest praznih mesta (razmaka) ispred broja 23 kako bi se on prikazao u polju dužine osam mesta.

Pored dela koji počinje znakom % za opis formata podataka, prvi argument metoda printf može sadržati druge znakove (uključujući specijalne znakove koji počinju sa \). Ovi znakovi se na ekranu prikazuju neizmenjeni i mogu poslužiti da se prikaže proizvoljni tekst koji bliže opisuje podatke. Sledeće naredbe, na primer:

```
int n = 23;
System.out.printf("Kvadrat broja %d je %8d\n", n, n*n);
```

na ekranu proizvode prikaz teksta:

```
Kvadrat broja 23 je       529
```

U ovom primeru, format %d se odnosi na vrednost drugog argumenta metoda printf (promenljiva n), a format %8d se odnosi na vrednost trećeg argumenta (izraz n*n). Ovaj primer takođe pokazuje da argumenti metoda printf mogu biti proizvoljni izrazi.

Za prikazivanje realnih brojeva se dodatno može navesti željeni broj decimala prikazanog broja. Ovu mogućnost smo koristili u naredbi

```
System.out.printf("Mesečna kamata: %5.2f\n", kamatnaStopa);
```

gde smo vrednost promenljive kamatnaStopa prikazali u formatu %5.2f. Deo između % i slova f se u opštem slučaju sastoji od ukupnog broja mesta i broja decimala razdvojenih tačkom. Tako 5.2 znači da se realan broj koji je sadržaj promenljive kamatnaStopa prikazuje u polju od pet mesta sa dve decimale.

Vrlo veliki i vrlo mali realni brojevi se mogu prikazati u eksponencijalnom obliku ukoliko se koristi slovo e na kraju. Tako se broj 0.3333^{23} u formatu %8.1e prikazuje u obliku 0.3e23. Ako se koristi slovo g umesto slova e, onda se mali realni brojevi prikazuje u normalnom obliku i veliki realni brojevi u eksponencijalnom obliku.

Pomenimo još slovo s koje se može koristiti za bilo koji tip podataka. Tim slovom se neka vrednost prikazuje u svom podrazumevanom, neformatizovanom obliku. Na primer, %10s znači da se neka vrednost prikazuje u polju od (minimalno) deset mesta.

Klasa System sadrži i metod exit koji ponekad može biti koristan. U programu se ovaj metod mora naravno pisati punim imenom System.exit. Ovaj metod bezuslovno prekida izvršavanje programa i koristi se u situacijama kada program treba prekinuti pre kraja rada metoda main. Metod exit ima jedan celobrojni parametar (iz istorijskih razloga), tako da poziv ovog metoda može biti u obliku System.exit(0) ili System.exit(-1). Navedeni argument u zagradi je predviđen da bliže opiše razlog prekida izvršavanja programa. Tako 0 znači da je program normalno završio, a neki drugi ceo broj da je došlo do greške prilikom izvršavanja. U praksi se ipak vrednost argumenta obično ne uzima u obzir.

Klasa Math

Svaki metod obavlja specifični zadatak. Rezultat nekih metoda je izračunavanje određene vrednosti koja se zatim na neki način koristi u programu. Tada kažemo da metod ***vraća*** vrednost.

Za mnoge matematičke funkcije u Javi postoje metodi koji izračunavaju i vraćaju odgovarajuću vrednost. Ti metodi su grupisani u klasi Math kao njeni statički članovi. Na primer, za izračunavanje kvadratnog korena nekog broja služi metod sqrt: ako je x neka numerička vrednost, Math.sqrt(x) izračunava i vraća kvadratni koren te vrednosti.

Poziv metoda Math.sqrt(x) predstavlja vrednost tipa double i može se navesti svuda gde se neka vrednost tog tipa može koristiti. Na primer, rezultat kvadratnog korena se na ekranu može prikazati naredbom:

```
System.out.println(Math.sqrt(x));
```

Ili poziv Math.sqrt(x) može biti deo naredbe dodele na desnoj strani znaka = radi upisivanja vrednosti izračunatog kvadratnog korena u neku promenljivu:

```
double stranica = Math.sqrt(x);
```

Nepotpun spisak statičkih metoda u klasi Math za neke važnije matematičke funkcije je:

- Math.abs(x) izračunava apsolutnu vrednost od x.
- Math.sin(x), Math.cos(x) i Math.tan(x) izračunavaju odgovarajuće trigononometrijske funkcije od x. (Ugao x se navodi u radijanima, a ne u stepenima.)
- Math.exp(x) izračunava broj $e = 2.71828\cdots$ na stepen x.
- Math.log(x) izračunava logaritam broja x za osnovu e.
- Math.pow(x,y) izračunava broj x na stepen y.
- Math.floor(x) odbacuje decimale realnog broja x, a Math.round(x) zaokružuje broj x na najbliži ceo broj. Na primer, Math.floor(5.76) vraća broj 5.0, a Math.round(5.76) vraća broj 6.
- Math.random() vraća (pseudo) slučajni realni broj iz intervala $[0, 1)$.

U svim ovim metodima, parametar u zagradama može biti bilo kog numeričkog tipa. Osim za Math.abs(x) i Math.round(x), vraćena vrednost za sve metode je tipa double. Za Math.abs(x) je vraćena vrednost istog tipa kao tip parametra x, dok je za Math.round(x) vraćena vrednost celobrojnog tipa long. Primetimo da metod Math.random() nema parametre.

Klasa Math sadrži i nekoliko statičkih polja kojima su predstavljene poznate matematičke konstante. Tako, Math.PI predstavlja broj $\pi =$

$3.14159\cdots$, dok `Math.E` predstavlja broj $e = 2.71828\cdots$. Ova statička polja su tipa `double` i naravno daju samo približnu vrednost odgovarajućih konstanti koje imaju beskonačno decimala.

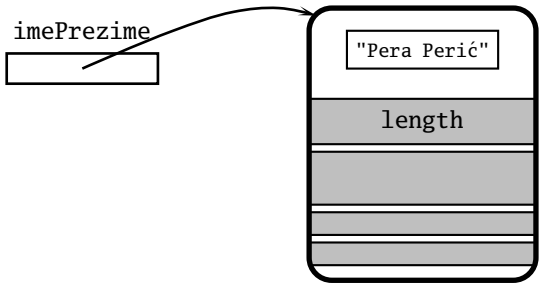
Klasa `String`

Vrednosti tipa `String` su objekti koje zovemo stringovi (niske). Objekat klase `String` sadrži podatke — nizove znakova koji čine string, ali i metode kojima se može manipulisati stringovima: na primer, metod `length()` koji daje broj znakova u stringu.

Ako pretpostavimo da smo deklarisali promenljivu `imePrezime` da bude tipa `String` i dodelili joj vrednost naredbom:

```
String imePrezime = "Pera Perić";
```

onda relevantni deo memorije računara izgleda otprilike onako kao što je to prikazano na slici 3.2.



SLIKA 3.2: Stanje relevantnog dela memorije posle izvršavanja naredbe `String imePrezime = "Pera Perić";`.

Zapis `imePrezime.length()` predstavlja poziv metoda `length` za objekat stringa na koga ukazuje promenljiva `imePrezime`. Rezultat ovog poziva je ceo broj 10 koji predstavlja dužinu (broj znakova) stringa `"Pera Perić"`. Opštije, za svaku promenljivu `s` tipa `String`, rezultat poziva `s.length()` je ceo broj tipa `int` jednak broju znakova stringa u objektu na koga ukazuje promenljiva `s`. Obratite pažnju na to da metod `length` nema eksplicitni argument na koji se primenjuje, već se izračunava dužina stringa koji se nalazi u implicitnom argumentu na koji ukazuje promenljiva `s`.

Primetimo da je precizna objektna terminologija vrlo nezgrapna. Naime, neka promenljiva tipa `String` ne sadrži niz znakova, nego samo ukazuje na objekat koji u sebi sadrži konkretni niz znakova kao podatke. Radi kraćeg izražavanja, međutim, ovo se zanemaruje i govori se kao da je vrednost takve promenljive sâm string. Tako bismo u prethodnom primeru kazali da promenljiva `imePrezime` ima vrednost (koja je niz znakova) `"Pera Perić"`, ili čak još kraće da je string `imePrezime` jednak `"Pera Perić"`. Napomenimo da ova terminološka nepreciznost nije svojtvena samo objektima tipa `String`, nego se primenjuje i za objekte bilo kog tipa ukoliko ne dolazi do zabune.

Pored metoda `length()`, u standardnoj klasi `String` je definisan veliki broj dodatnih metoda. Sledi opis onih najčešće korišćenih, uz pretpostavku da su `s1` i `s2` promenljive tipa `String`:

- `s1.equals(s2)` vraća logičku vrednost tipa `boolean`: `true` (tačno) ako se string `s1` sastoji od tačno istog niza znakova kao string `s2`, a `false` (netačno) u suprotnom slučaju.
- `s1.equalsIgnoreCase(s2)` proverava kao prethodni metod da li su stringovi `s1` i `s2` jednaki, ali se pri tome mala slova ne razlikuju od velikih.
- `s1.charAt(n)`, gde je `n` ceo broj, vraća znak tipa `char` koji se nalazi u n-toj poziciji stringa `s1`. Pozicije niza znakova su numerisane od 0, stoga `s1.charAt(0)` daje prvi znak stringa `s1`, `s1.charAt(1)` daje drugi znak stringa `s1` i tako dalje. Poslednji znak stringa `s1` se nalazi u poziciji `s1.length()`-1.
- `s1.substring(n,m)`, gde su `n` i `m` celi brojevi, vraća podniz tipa `String` koji se sastoji od znakova stringa `s1` u pozicijama n, $n+1$, $\ldots$, $m-1$. Obratite pažnju na to da se znak u poziciji m stringa `s1` ne uključuje.
- `s1.indexOf(s2)` vraća ceo broj koji predstavlja poziciju prvog pojavljivanja podniza `s2` u stringu `s1`. Ako se podniz `s2` ne nalazi u `s1`, vraćena vrednost je -1. Slično se može koristiti i `s1.indexOf(ch)` radi dobijanja pozicije prvog pojavljivanja znaka `ch` u stringu `s1`.
- `s1.compareTo(s2)` vraća ceo broj koji predstavlja rezultat upoređivanja stringova `s1` i `s2`. Ako su `s1` i `s2` jednaki, vraćena vrednost je 0. Ako je `s1` manje od `s2`, vraćena vrednost je ceo broj manji od nule. Najzad, ako je `s1` veće od `s2`, vraćena vrednost je ceo

broj veći od nule. (Relacije ,,manje od" i ,,veće od" za stringove odgovaraju njihovom leksikografskom redosledu. U stvari, redosled stringova se malo komplikovanije određuje, ali za većinu primena je leksikografski redosled dovoljan.)

- `s1.toUpperCase()` vraća kao rezultat novi string koji je jednak stringu `s1`, osim što se sva mala slova u `s1` pretvaraju u velika. Na primer, ako string `s1` ima vrednost `"Jaje"`, rezultat poziva `s1.toUpperCase()` je string `"JAJE"`. Za obrnut zadatak pretvaranja velikih slova u mala slova se koristi `s1.toLowerCase()`.
- `s1.trim()` vraća kao rezultat novi string koji je jednak stringu `s1`, osim što se nevidljivi znakovi sa početka i kraja stringa `s1` odstranjuju. Na primer, ako `s1` ima vrednost `"␣␣to␣je:␣"`, rezultat poziva `s1.trim()` je string `"to␣je:"`.

Obratite pažnju na to da se implicitni argument (string `s1`) kod poziva `s1.toUpperCase()`, `s1.toLowerCase()` i `s1.trim()` *ne* modifikuje, već se konstruiše novi string koji se vraća kao rezultat. Da bi se promenila vrednost stringa `s1`, mora se koristiti naredba dodele, na primer:

```
s1 = s1.toUpperCase();
```

Vratimo se na kraju ovog odeljka operaciji spajanja stringova koju smo spomenuli u odeljku 3.2. Oznaka za tu operaciju je znak + i već smo je koristili u programima. Na primer, izvršavanje naredbe

```
System.out.println("Zdravo " + ime + "!");
```

sastoji se od dva međukoraka. Najpre se tri stringa navedena između zagrada kao argument metoda `println` (`"Zdravo "`, vrednost promenljive `ime` i `"!"`), spajaju jedan iza drugog i dobija se jedan, novi string. Zatim se taj rezultujući string prikazuje na ekranu.

Ova operacija se, u stvari, može koristiti za spajanje ne samo dva stringa (ili više njih), nego i za spajanje stringa i vrednosti bilo kog tipa. Na primer, ako celobrojna promenljiva `god` ima vrednost 23, onda je rezultat izraza

```
god + " su najlepše godine."
```

jednak stringu `"23 su najlepše godine."`. Pri izračunavanju ovog rezultata se celobrojna vrednost 23 promenljive `god` najpre pretvara u odgovarajući niz znakova `"23"`, a zatim se taj string spaja sa stringom `" su najlepše godine."`

Ova mogućnost se često koristi u `print` metodima za prikazivanje vrednosti bilo kog tipa. Na primer:

```
System.out.println("Mesečna rata: " + r);
```

prikazuje realnu vrednost promenljive `r` tipa `double` uz prigodni tekst.

Obratite ipak pažnju na to da kod operacije spajanja bar jedan od operanada mora biti string. Tako, u prethodnom primeru promenljive `god`, rezultat izraza `god + 1` je očekivano ceo broj 24. Sa druge strane, dodavanjem praznog stringa dobijamo vrlo sličan izraz `"" + god + 1`, ali potpuno drugi rezultat: string "231". Naime, u prvom izrazu se znak + interpretira kao operator sabiranja brojeva, jer su oba operanda brojevi. U drugom izrazu se pak znak + interpretira kao operator spajanja stringova, jer se kao jedan operand pojavljuje string. Zato se prazan string spaja sa stringom "23" i dobija isti string "23", da bi se taj međurezultat spojio na kraju sa stringom "1" i dobio konačan rezultat "231".

Klasa **Scanner**

Jedna od poteškoća na koju nailaze oni koji uče programiranje u Javi je to što se njene mogućnosti za učitavanje vrednosti osnovnih tipova podataka zasnivaju na naprednim, objektno orijentisanim tehnikama. Da bi se to donekle popravilo, od verzije Java 5 je u paketu `java.util` dodata klasa `Scanner` koja olakšava učitavanje ulaznih podataka u Java programima. Učitavanje vrednosti u Java programu ipak nije tako jednostavno kao njihovo prikazivanje, jer metodi koji su definisani u klasi `Scanner` nisu statički nego objektni. To znači da je za njihovu primenu potrebno najpre konstruisati objekat tipa `Scanner`.

Za konstruisanje jednog objekta bilo kog klasnog tipa *`Klasa`* služi operator `new` koji ima opšti oblik:

`new` *`Klasa`*`( ... )`

gde je *`Klasa`*`( ... )` poziv specijalnog metoda koji se naziva *konstruktor* klase čije je ime *`Klasa`*. O konstruktorima će biti više reči u odeljku 6.3. Za sada je dovoljno znati da je to specijalni metod neke klase kojim se vrši konstruisanje objekta te klase u memoriji i njihova inicijalizacija. Rezultat ovog metoda je referenca (pokazivač) na konstruisani objekat u memoriji. Na primer, pošto je `String` regularna klasa u Javi, naredbom dodele:

```
String imePrezime = new String("Pera Perić");
```

najpre se u memoriji konstruiše novi objekat klase `String` i inicijalizuje stringom `"Pera Perić"` koji je naveden između zagrada kao argument konstruktora klase `String`. Vraćena vrednost operatora new je referenca na novokonstruisani objekat i ona se zatim dodeljuje promenljivoj `imePrezime`. Drugim rečima, efekat izvršavanja ove naredbe dodele je isti onaj koji je prikazan na slici 3.2. U stvari, raniji oblik naredbe dodele koji smo koristili:

```
String imePrezime = "Pera Perić";
```

predstavlja samo mogućnost jednostavnijeg pisanja umesto prethodnog proširenog oblika.

Ako operator new primenimo za konstruisanje objekta klase `Scanner`, možemo pisati recimo:

```
Scanner tastatura = new Scanner(System.in);
```

Ovom naredbom dodele se konstruiše objekat klase `Scanner` i referenca na njega se dodeljuje promenljivoj `tastatura`. Argument konstruktora klase `Scanner` naveden u zagradama jeste objekat `System.in`. Time se novokonstruisani objekat klase `Scanner` povezuje sa ***standardnim ulazom*** kao izvorom podataka koji će se čitati. Standardni ulaz je apstraktni model učitavanja raznih tipova podataka preko fizičkog uređaja tastature. On je u Javi predstavljen objektom na koga ukazuje statičko polje `in` u klasi `System`, baš kao što je standardni izlaz predstavljen objektom na koga ukazuje statičko polje `out` u klasi `System`.

Klasa `Scanner` sadrži metode koji se mogu primeniti na neki objekat te klase radi učitavanja podataka različitog tipa. Tako, uz prethodnu definiciju promenljive `tastatura`, imamo:

- `tastatura.next()` — učitava se sledeći niz znakova preko tastature do prve beline(znaka razmaka, tabulatora ili novog reda) i vraća se kao vrednost tipa `String`.
- `tastatura.nextInt()`, `tastatura.nextDouble()` i tako dalje — učitava se sledeća vrednost preko tastature tipa `int`, `double` i tako dalje. Postoje odgovarajući metodi za čitanje vrednosti svih primitivnih tipova podataka.
- `tastatura.nextLine()` — učitava se sledeći niz znakova preko tastature do kraja reda (znaka novog reda) i vraća se kao vrednost

tipa String. Obratite pažnju na to da se ovim metodom učitavaju eventualni znakovi razmaka i tabulatora do kraja reda, kao i na to da se završni znak novog reda čita, ali nije deo vraćenog stringa.

Neke od ovih metoda smo koristili u programu *Zdravo* iz odeljka 2.5. Na primer, naredbom:

```
int god = tastatura.nextInt();
```

učitava sa ceo broj preko tastature i dodeljuje celobrojnoj promenljivoj god.

U klasi Scanner su definisani i metodi kojima se može proveravati to da li su ulazni podaci raspoloživi, kao i to da li su oni određenog tipa. Na primer:

- `tastatura.hasNext()` — vraća logičku vrednost tipa boolean, i to true (tačno) ako je neki ulazni niz znakova, različit od znakova beline, raspoloživ preko tastature.
- `tastatura.hasNextInt()`, `tastatura.hasNextDouble()` i tako dalje — vraća logičku vrednost tipa boolean, i to true (tačno) ako je odgovarajuća vrednost tipa int, double i tako dalje, raspoloživa preko tastature.
- `tastatura.hasNextLine()` — vraća logičku vrednost tipa boolean, i to true (tačno) ako je ulazni niz znakova raspoloživ preko tastature do kraja reda.

Na kraju napomenimo da smo ovde obradili samo osnovne mogućnosti klase Scanner za učitavanje podataka preko tastature. Ova klasa je mnogo opštija i može poslužiti za učitavanje podataka i iz drugih izvora podataka kao što su datoteke, mrežne konekcije ili čak stringovi. Opšti koncept je ipak isti, pa više detalja o dodatnim mogućnosti klase Scanner potražite u zvaničnoj dokumentaciji.

Klase omotači

Osam primitivnih tipova podataka u Javi nisu klase, a ni vrednosti primitivnih tipova nisu objekti. Ponekad je ipak potrebno tretirati primitivne vrednosti kao da su to objekti. To se ne može direktno uraditi, ali se vrednost primitivnog tipa može „umotati" u objekat odgovarajuće ***klase omotača***. Za svaki od osam primitivnih tipova postoji odgovarajuća

klasa omotač: Byte, Short, Integer, Long, Float, Double, Character i Boolean.

Ove klase u programu prvenstveno služe za konstruisanje objekata koji predstavljaju vrednosti primitivnih tipova, mada te klase sadrže i neke korisne statičke članove za rad sa vrednostima primitivnih tipova. Glavna namena klasa omotača za, recimo, klasu Double ogleda se u tome što svaki njen objekat sadrži objektnu promenljivu tipa double. Takav objekat je „omotač" za vrednost tipa double. Na primer, objekat-omotač za vrednost 5.220931 tipa double se može konstruisati operatorom new:

```
Double d = new Double(5.220931);
```

Objektna promenljiva d predstavlja iste informacije kao primitivna promenljiva čija je vrednost 5.220931, ali u formi objekta. Ako treba koristiti vrednost tipa double koju sadrži objekat-omotač, to se mora uraditi primenom metoda doubleValue() klase Double na taj objekat, na primer:

```
double x = d.doubleValue() * 3.14;
```

Slično, vrednost tipa int može se „umotati" u objekat tipa Integer, vrednost tipa char u objekat tipa Character i tako dalje. U stvari, od verzije Java 5.0 je moguća automatska konverzija između primitivnih tipova i odgovarajućih klasa omotača. To znači da ako se koristi, recimo, vrednost tipa int u kontekstu u kojem je potreban objekat tipa Integer, ta int vrednost će se automatski pretvoriti u Integer objekat. Na primer, možemo pisati:

```
Integer broj = 23;
```

i to će se automatski interpretirati kao da smo napisali:

```
Integer broj = new Integer(23);
```

Ova mogućnost se naziva ***autopakovanje*** (engl. *autoboxing*). Naglasimo da se automatska konverzija primenjuje i u drugom smeru. Tako, ako promenljiva d ukazuje na objekat klase Double, možemo kraće pisati:

```
double x = d * 3.14;
```

U ovom primeru, prilikom izračunavanja izraza na desnoj strani, vrednost tipa double koja se nalazi u objektu na koga ukazuje d će se automatski ***raspakovati*** i pomnožiti sa 3.14.[4]

Autopakovanje i raspakivanje u Javi se može primeniti u svim slučajevima kada to ima smisla. U stvari, to svojstvo nam omogućava da možemo skoro potpuno zanemariti razlike između primitivnih tipova podataka i objekata.

Druga namena klasa omotača se ogleda u tome što obuhvataju neke korisne statičke članove koji olakšavaju rad sa primitivnim vrednostima. Klasa Integer, na primer, sadrži konstante MIN_VALUE i MAX_VALUE koje su jednake minimalnoj i maksimalnoj vrednosti tipa int, odnosno brojevima -2147483648 i 2147483647. Ovo je korisno jer je svakako lakše zapamtiti imena nego numeričke vrednosti. Iste konstante se nalaze i u klasama Byte, Short, Long, Float, Double i Character.

Doduše, u klasama Float i Double konstanta MIN_VALUE ima drugačiju interpretaciju i predstavlja najmanju *pozitivnu* vrednost koja iznosi $4.9 \cdot 10^{-324}$. To je dakle najmanja moguća pozitivna realna vrednost različita od nule, pošto se realni brojevi u računaru predstavljaju sa ograničenom preciznošću i ne mogu se predstaviti mali brojevi proizvoljno bliski nuli.

Klasa Double sadrži i neke konstante koje ne predstavljaju realne brojeve u matematičkom smislu. Konstanta POSITIVE_INFINITY označava beskonačnu vrednost koja se može dobiti kao rezultat nekih izraza. Na primer, deljenje pozitivnog broja sa nulom ili proizvod 1e200 * 1e200 koji prevazilazi vrednost MAX_VALUE, u Javi su definisani i kao rezultat daju POSITIVE_INFINITY. Slično važi i za konstantu NEGATIVE_INFINITY. Konstanta NaN (skraćenica od engl. *not a number*) predstavlja nedefinisanu vrednost koja se dobija, recimo, izračunavanjem kvadratnog korena od negativnog broja ili deljenjem nule sa nulom.

Klase omotači sadrže i neke statičke metode za rad sa odgovarajućim tipovima podataka. Tako klasa Integer sadrži metod parseInt() koji pretvara string u vrednost tipa int. Na primer, Integer.parseInt("17") kao rezultat daje ceo broj 17 tipa int. Ako argument metoda parseInt() nije niz znakova koji predstavljaju ceo broj, u programu nastaje greška. Slično, klasa Double sadrži metod parseDouble() koji pretvara string u

[4]Klase omotači sadrže objektne metode kojima se može i „ručno" raspakovati numerički objekat. Tako, d.doubleValue() daje vrednost tipa double koja se nalazi u objektu na koga ukazuje d.

vrednost tipa double. Na primer, Double.parseDouble("1.548e100") kao rezultat daje realni broj 154.8 tipa double.

Statički metodi sličnih funkcija u ovim klasama, ali koji kao rezultat daju objekat odgovarajuće klase, nazivaju se valueOf(). Na primer, Integer.valueOf("17") kao rezultat daje objekat klase Integer koji sadrži celobrojnu vrednost 17. Slično, Double.valueOf("1.548e100") kao rezultat daje objekat klase Double koji sadrži realni broj 154.8.

Klasa Character sadrži, između ostalog, veliki broj statičkih metoda kojima se može ispitivati dati znak tipa char. To se može pokazati vrlo korisnim u programima koji obrađuje tekstualne podatke, pogotovo na stranim jezicima sa „egzotičnim" pismom. (Podsetimo se da su znakovi u Javi predstavljeni na način propisan Unicode standardom koji sadrži ogroman broj glifova.) Na primer, metodi isLetter(), isLetterOrDigit(), isLowerCase() i tako dalje kao rezultat daju logičku vrednost tačno ili netačno prema tome da li dati argument tipa char predstavlja znak koji je slovo, slovo ili cifra, malo slovo i tako dalje.

3.5 Izrazi

Podsetimo se da izraz može predstavljati jednu vrednost ili formulu na osnovu koje se izračunava jedna vrednost. Drugim rečima, izraz može biti literalna vrednost, promenljiva, poziv metoda, ili nekoliko od ovih stvari u kombinaciji sa operatorima kao što su + i <.

Primetimo da je vrednost izraza uvek tačno jedna vrednost: literalna vrednost, vrednost promenljive, rezultat poziva metoda, ili vrednost koja se dobija izračunavanjem izraza. Ta vrednost izraza se može dodeliti nekoj promenljivoj, prikazati na ekranu print naredbama, koristiti kao argument u pozivima metoda, ili kombinovati sa drugim vrednostima u građenju složenijeg izraza. Do sada smo izraze razmatrali na neformalan način, ali pošto oni predstavljaju važan koncept programiranja, sada ćemo im posvetiti više pažnje.

Osnovni gradivni elementi izraza su literali („bukvalne" vrednosti kao što su 17, 0.23, true ili 'A'), promenljive i pozivi metoda. Literali, promenljive i pozivi metoda su prosti izrazi, a složeniji izrazi se grade kombinovanjem prostih izraza sa operatorima (aritmetičkim, relacijskim i drugim). Kada se u složenom izrazu nalazi više operatora, njihov redosled

primene radi izračunavanja tog izraza se određuje na osnovu njihovog prioriteta. Na primer, izraz:

```
x + y * z
```

izračunava se tako što se najpre izračunava podizraz `y*z`, a zatim se njegov rezultat sabira sa `x`. To je zato što množenje predstavljeno operatorom `*` ima viši prioritet od sabiranja koje je predstavljeno operatorom `+`. Ako nam ovaj unapred definisan prioritet operatora ne odgovara, redosled izračunavanja izraza možemo promeniti upotrebom zagrada. Na primer, kod `(x+y)*z` bi se najpre izračunavao podizraz u zagradi `x+y`, a zatim bi se njegov rezultat množio sa `z`.

Aritmetički operatori

Osnovni aritmetički operatori u Javi odgovaraju matematičkim operacijama sabiranja, oduzimanja, množenja i deljenja sa oznakama `+`, `-`, `*` i `/`. Ovi operatori se mogu primenjivati na vrednosti bilo kog numeričkog tipa (`byte`, `short`, `int`, `long`, `float` i `double`). Ali kada se neka od odgovarajućih operacija stvarno izračunava za dve vrednosti, obe vrednosti moraju biti istog tipa. Na primer, kod izračunavanja `17.4+10` se najpre ceo broj 10 pretvara u ekvivalentni realni broj `10.0`, a zatim se izračunava `17.4+10.0`. Ovo se naziva ***konverzija tipa*** i predstavlja pretvaranje vrednosti „manjeg" tipa u ekvivalentnu vrednost „većeg" tipa. Konverzija tipa se automatski vrši prilikom izračunavanja izraza. O tome obično ne treba voditi računa u programu, jer ne dolazi do gubitka tačnosti.

Kada se jedan od osnovnih aritmetičkih operatora primenjuje na dve vrednosti istog tipa (nakon eventualne konverzije tipa), dobija se i rezultat istog tipa. Ako se množe dva cela broja tipa `int`, rezultat je ceo broj tipa `int`; ako se sabiraju dva realna broja tipa `double`, rezultat je realni broj tipa `double`. Ovo je prirodno pravilo, osim u slučaju operatora deljenja `/`. Kada se dele dva cela broja, rezultat toga je opet ceo broj i eventualne decimale rezultata se odbacuju. Na primer, `7/2` kao rezultat daje ceo broj 3, a ne realni broj `3.5`. Slično, ako je `n` celobrojna promenljiva tipa `int`, tada je i `1/n` ceo broj `int`. Tako, ako `n` ima vrednost 10, rezultat izračunavanja `1/n` je 0. (U stvari, rezultat izraza `1/n` biće 0 za svaku vrednost od `n` veću od jedan!)

Tačna vrednost celobrojnog deljenja se može dobiti na dva načina. Prvi način je da se jedan od operanada napiše kao realan broj. Na primer,

kod izračunavanja izraza `1.0/n`, zbog konverzije tipa se najpre vrednost promenljive `n` pretvara u realni broj, pa se kao rezultat dobija tačan realni broj. To znači da ako `n` ima vrednost 10, ta vrednost se pretvara u realni broj `10.0`, i stoga `1.0/10.0` daje tačan rezultat `0.1`.

Drugi način za dobijanje tačnog rezultata celobrojnog deljenja je korišćenje operatora ***eksplicitne konverzije tipa*** (engl. *type cast*). Oznaka tog operatora u Javi je ime odgovarajućeg tipa podataka u zagradama, što se piše ispred vrednosti koju želimo da pretvorimo u navedeni tip. Na primer, ako su `n` i `m` celobrojne promenljive tipa `int`, u izrazu:

```
(double)n/m
```

zahteva se pretvaranje vrednosti promenljive `n` u tip `double` pre njenog deljenja sa vrednošću promenljive `m`, tako da se na kraju dobija tačan rezultat deljenja tipa `double`.

Kao drugi primer eksplicitne konverzije tipa, razmotrimo postupak za generisanje slučajnog celog broja između 1 i 6. (Rešenje ovog problema se u praksi može iskoristiti za simuliranje ishoda bacanja kocke za igru.) Metod `Math.random()` se ne može neposredno primeniti, jer on daje slučajni *realni* broj između 0 i $0.9999\cdots$. Ali rezultat izraza `6*Math.random()` će biti slučajni realni broj između 0 i $5.9999\cdots$. Operator eksplicitne konverzije tipa `(int)` možemo zatim iskoristiti za dobijanje celog broja rezultata ovog izraza, jer se realni broj pretvara u ceo broj odbacivanjem decimalnog dela. Prema tome, izraz `(int)(6*Math.random())` kao rezultat daje jedan od celih brojeva 0, 1, 2, 3, 4 ili 5 na slučajan način. Pošto nam treba slučajni ceo broj između 1 i 6, na kraju možemo dodati 1: `(int)(6*Math.random()) + 1` je dakle konačno rešenje.

Eksplicitna konverzija tipa se može vršiti i između znakovnog tipa `char` i celobrojnog tipa `int`. Numerička vrednost nekog znaka je njegov Unicode kodni broj tako da, recimo, `(int)'+'` daje ceo broj 43 i `(char)97` daje znak `'a'`. (Doduše, konverzija iz tipa `char` u tip `int` se vrši automatski, pa je u prvom primeru njeno eksplicitno pisanje suvišno, jer bi se ionako izvršila iz konteksta.)

Pošto celobrojno deljenje daje samo celobrojni rezultat, Java ima operator za izračunavanje ostatka pri celobrojnom deljenju. Oznaka tog operatora je znak procenta `%` tako da, recimo, `7%2` kao rezultat daje 1, ili `3456%100` kao rezultat daje 56.

Prethodni aritmetički operatori su takozvani binarni operatori, jer se svaki od njih primenjuje na dve vrednosti. Java ima i nekoliko unarnih aritmetičkih operatora koji se primenjuju samo na jednu vrednost. Jedan smo već pomenuli — operator eksplicitne konverzije tipa, koji se primenjuje na jedan operand. Drugi je operator koji se zove unarni minus. On se označava isto kao operacija oduzimanja znakom –, ali se primenjuje na jedan operand. Na primer, –x ima istu vrednost kao $(-1)*x$, što znači da operacija unarnog minusa kao rezultat daje vrednost operanda sa suprotnim znakom. Tako, ako x ima vrednost 17, –x kao rezultat daje -17; ako x ima vrednost -17, –x kao rezultat daje 17. Kao kuriozitet pomenimo da u Javi postoji i operator koji se zove unarni plus. Tako možemo pisati +x, mada to u stvari ne menja ništa.

Unarni operatori inkrementiranja i dekrementiranja promenljive obezbeđuju kraći zapis za vrlo česte operacije u programiranju kojima se vrednost neke promenljive uvećava za 1 ili smanjuje za 1. Efekat uvećanja vrednosti neke promenljive x za 1 se može postići naredbom dodele x=x+1. Naime, na desnoj strani znaka = se najpre stara vrednost promenljive x uvećava za 1, pa se taj rezultat dodeljuje promenljivoj x kao nova vrednost (jer se ista promenljiva x nalazi na levoj strani znaka =). Slično, naredbom dodele x=x-1 se postiže smanjenje vrednosti promenljive x za 1.

Operatori inkrementiranja i dekrementiranja, čije su oznake ++ i --, predstavljaju kraći zapis za operacije uvećanja ili smanjenja za 1. Tako, x++ ili ++x ima isti efekat kao x=x+1, dok x-- ili --x ima isti efekat kao x=x-1. Na primer, ako promenljiva x ima vrednost 10, nakon izvršavanja x++ njena vrednost biće 11.

Pisanje simbola ++ ili -- ispred ili iza neke promenljive nije bitno ako se operatori inkrementiranja i dekrementiranja pojavljuju u sklopu samostalne naredbe: na primer, x++; ili ++x; (sa ; na kraju). Međutim, zapisi x++, ++x, x-- i --x se mogu koristiti kao samostalni izrazi ili biti deo složenijih izraza. Na primer, ispravno zapisane naredbe su:

```
y = x++;
z = (--x)*(y++);
System.out.println(--x-1);
```

U ovim slučajevima je to da li se ++ ili -- nalazi ispred ili iza neke promenljive bitno, jer proizvodi različit efekat. Na primer, efekat naredbe dodele y=x++; je najpre dodeljivanje *stare* vrednosti promenljive x pro-

menljivoj y, a zatim uvećavanje vrednosti promenljive x za 1. Na primer, ako x ima vrednost 10, nakon izvršavanja y=x++; će y imati vrednost 10 (staru vrednost x), dok će nova vrednost x biti 11. Sa druge strane, efekat naredbe dodele y=++x; je najpre uvećavanje vrednosti promenljive x za 1, a zatim dodeljivanje te *nove* vrednosti promenljive x promenljivoj y. Na primer, ako x ima vrednost 10, nakon izvršavanja y=++x; će x i y imati istu vrednost 11, jer se nova vrednost x dodeljuje y. U slučaju operatora dekrementiranja -- važi slično pravilo.

Ovaj efekat je, očigledno, prilično komplikovan čak i najprostijem slučaju, a kamoli u složenijim izrazima kada može biti potpuno nerazumljiv. Zbog toga se nikako ne preporučuje upotreba operatora inkrementiranja i dekrementiranja u izrazima, nego samo u samostalnim naredbama.

Relacijski operatori

Aritmetički operatori iz prethodnog odeljka služe za građenje *numeričkih* izraza, odnosno izraza koji kao rezultat daju numeričku vrednost. Druga vrsta izraza su *logički* izrazi koji kao rezultat daju logičku vrednost `true` (tačno) ili `false` (netačno).

Logički izrazi se obično koriste u naredbama grananja i ponavljanja o čemu ćemo govoriti u poglavlju 4. Druga, doduše ređa primena je u naredbama dodele kada se logička vrednost koja se dobija kao rezultat logičkog izraza dodeljuje logičkoj promenljivoj tipa `boolean`. Ovaj oblik naredbe dodele se konceptualno ne razlikuje od uobičajenijeg oblika kada se numerička vrednost koja se dobija kao rezultat numeričkog izraza dodeljuje numeričkoj promenljivoj.

Jedan od gradivnih elemenata logičkih izraza su relacijski operatori, koji se koriste radi upoređivanja da li su dve vrednosti jednake, nejednake, da li je jedna veća od druge i tako dalje. Relacijski operatori u Javi su ==, !=, <, >, <= i >=. Ako su x i y dve vrednosti, značenje ovih operatora je:

- x == y kao rezultat daje logičku vrednost `true` ako je x jednako y; u suprotnom slučaju je rezultat `false`.
- x != y kao rezultat daje logičku vrednost `true` ako x nije jednako y; u suprotnom slučaju je rezultat `false`.
- x < y kao rezultat daje logičku vrednost `true` ako je x manje od y; u suprotnom slučaju je rezultat `false`.

- x > y kao rezultat daje logičku vrednost true ako je x veće od y; u suprotnom slučaju je rezultat false.
- x <= y kao rezultat daje logičku vrednost true ako je x manje ili jednako y; u suprotnom slučaju je rezultat false.
- x >= y kao rezultat daje logičku vrednost true ako je x veće ili jednako y; u suprotnom slučaju je rezultat false.

Vrednosti x i y koje se upoređuju mogu biti bilo kog numeričkog tipa, ali i tipa char. Za znakove tipa char, operatori „manje" (<) i „veće" (>) su definisani prema numeričkim Unicode vrednostima znakova. Tako, 'a'<'z' kao rezultat daje true, jer je brojni kôd znaka 'a' manji od brojnog kôda znaka 'z' u Unicode šemi. Obratite ipak pažnju na to da redosled slova u ovoj šemi nije isto što i njihov alfabetski redosled, jer se velika slova nalaze ispred svih malih slova.

Vrednosti x i y mogu biti čak i logičke vrednosti u slučaju operatora „jednako" (==) i „nije jednako" (!=). Ispravno je, na primer, napisati naredbu dodele:

```
boolean istiZnak = ((x > 0) == (y > 0));
```

Napomenimo još da se relacijski operatori *ne* mogu koristiti za upoređivanje stringova (objekata tipa String). U stvari, operatori == i != mogu, ali je njihov efekat potpuno drugačiji od očekivanog. Na primer, operatorom == se upoređuje da li su reference na dva objekta tipa String jednake, a ne da li su nizovi znakova koje oni sadrže jednaki. Za upoređivanje stringove treba koristiti odgovarajuće metode u klasi String koje smo spomenuli u odeljku 3.4: equals, equalsIgnoreCase i compareTo.

Logički operatori

Logički operatori su za logičke izraze ono što su aritmetički operatori za numeričke izraze. Logički operatori se primenjuju na logičke vrednosti i kao rezultat daju isto tako logičku vrednost.

Logički operator I (ili konjunkcija) označava se simbolom &&. To je binarni logički operator koji kao rezultat daje true ako su oba operanda na koje se primenjuje jednaka true. U suprotnom slučaju (ako je jedan od operanada, ili oba, jednak false), rezultat operatora && je false. Na primer, kao rezultat izraza

```
(x == 0) && (y == 0)
```

dobija se logička vrednost `true` ako i samo ako je 0 vrednost obe promenljive `x` i `y`.

Logički operator ILI (ili disjunkcija) označava se simbolom `||` (dve uspravne crte). To je binarni logički operator koji kao rezultat daje `true` ako je bar jedan od operanada na koje se primenjuje, ili oba, jednak `true`. U suprotnom slučaju (ako su oba operanda jednaka `false`), rezultat operatora `||` je `false`. Na primer, kao rezultat izraza

```
(x == 0) || (y == 0)
```

dobija se logička vrednost `true` ako i samo ako je 0 vrednost bar jedne od promenljivih `x` ili `y` (ili obe promenljive).

Rezultat operatora `&&` i `||` se u Javi, u stvari, izračunava na kraći način ukoliko je to moguće. Naime, ako se nakon dobijanja logičke vrednosti prvog operanda može zaključiti koji je krajnji rezultat ovih operatora, drugi operand se uopšte ne izračunava. To u nekim slučajevima može biti važno! Razmotrimo sledeći izraz kao primer:

```
(x != 0) && (y/x > 1)
```

Ako `x` ima zaista vrednost 0, tada je količnik `y/x` matematički nedefinisan zbog deljenja 0, pa logička vrednost drugog operanda `(y/x>1)` ne bi mogla da se izračuna. Ali u slučaju kada je `x` jednako 0, drugi operand se uopšte neće ni izračunavati. Naime, tada je logička vrednost prvog operanda `(x!=0)` jednaka `false`, što znači da vrednost celog izraza u primeru mora biti `false`, bez obzira na vrednost drugog operanda. (Podsetimo se da logički operator `&&` kao rezultat daje `true` ako i samo ako su oba operanda jednaka `true`.)

U Javi postoji i unarni logički operator negacije koji se označava znakom `!` i piše se ispred jedinog logičkog operanda. Tako, `!x` kao rezultat daje `true` ako je vrednost operanda `x` jednaka `false`; a ako je vrednost operanda `x` jednaka `true`, onda je rezultat od `!x` jednak `false`. Drugim rečima, rezultat logičke negacije je suprotna logička vrednost od vrednosti operanda.

Operator izbora

Java ima i jedan ternarni operator (sa tri operanda) koji pominjemo samo radi kompletnosti, jer je njegova čitljivost u programu diskutabilna.

Pošto se ovaj operator može lako zameniti naredbom grananja (o kojoj govorimo u narednom poglavlju), treba ga koristiti samo u izuzetnim slučajevima.

Operacija izbora se u opštem obliku piše na sledeći način:

logički-izraz ? $izraz_1$: $izraz_2$

Obratite pažnju na to da znak pitanja dolazi iza *logičkog-izraza* i da znak dve-tačke razdvaja $izraz_1$ i $izraz_2$. Izračunavanje ove operacije je posebno po tome što se vrši u dve faze. Najpre se izračunava *logički-izraz*, a zatim se rezultat dobija zavisno od njegove vrednosti: ako je ona `true` (tačno), konačni rezultat je izračunata vrednost $izraza_1$; u suprotnom slučaju, ako je ona `false` (netačno), konačni rezultat je izračunata vrednost $izraza_2$. Na primer, naredbom

```
n = (m%2 == 0) ? (2*m) : (m-1);
```

promenljivoj `n` se dodeljuje vrednost izraza `2*m` ako je vrednost promenljive `m` paran broj (to jest, ako je vrednost logičkog izraza `m%2 == 0` jednaka tačno), ili se promenljivoj `n` dodeljuje vrednost izraza `m-1` ako je vrednost promenljive `m` neparan broj (to jest, ako `m%2 == 0` daje netačno). Primetimo da zagrade oko izraza u ovom primeru nisu neophodne, ali je sa njima postignuta bolja čitljivost.

Operatori dodele

Do sada smo govorili o naredbi dodele kojom se vrednost nekog izraza na desnoj strani znaka = dodeljuje promenljivoj koja se nalazi na levoj strani znaka =. Međutim, znak = predstavlja zapravo (binarni) *operator* dodele u smislu da operacija dodele vrednosti daje rezultat, koji se eventualno može koristiti u sklopu nekog složenijeg izraza.

Vrednost operatora dodele u opštem obliku

ime = *izraz*

jednaka je vrednosti *izraza* na desnoj strani znaka jednakosti. Prema tome, efekat operatora dodele je dvojak: nakon izračunavanja *izraza* se njegova vrednost najpre dodeljuje promenljivoj *ime* i zatim se daje kao rezultat samog operatora =.

Na primer, ako bismo želeli da vrednost promenljive `y` dodelimo promenljivoj `x` i istovremeno testiramo da li je ta vrednost jednaka 0, mogli

bismo iskoristiti rezultat operatora dodele u sklopu operatora izbora (ili naredbe grananja) na sledeći način:

```
( (x = y) == 0 ) ? ( ... ) : ( ... )
```

Ovde se najpre izračunava operacija dodele x=y tako što se vrednost y dodeljuje x i ta vrednost se vraća kao rezultat ove operacije. Taj rezultat, tj. vrednost y odnosno x, zatim se relacijskim operatorom == proverava da li je jednaka 0. Zavisno od toga da li je to tačno ili netačno, na kraju se izračunava izraz iza znaka ? ili :, na način kako smo to već objasnili radi izračunavanja rezultata operatora izbora. Naravno, ovo je vrlo teško pratiti i zato ovu mogućnost kod dodele vrednosti treba koristiti samo u zaista izuzetnim prilikama. Mnogo razumljivije je prethodni primer raščlaniti u dva reda sa istim efektom, na primer:

```
x = y;
(x == 0) ? ( ... ) : ( ... )
```

ili, još bolje, umesto operatora izbora u drugom redu treba koristiti naredbu grananja o kojoj će više reči biti u odeljku 4.2.

Operator dodele u Javi ima, u stvari, nekoliko varijacija koje se mogu koristiti radi kraćeg pisanja.[5] Na primer:

```
ime += izraz
```

ekvivalentno je sa

```
ime = ime + izraz
```

Svaki binarni operator u Javi se može koristiti na sličan način sa operatorom dodele. Na primer:

```
x -= y;   // isto što i x = x - y;
x *= y;   // isto što i x = x * y;
x /= y;   // isto što i x = x / y;
n %= m;   // isto što i n = n % m; (n i m celobrojne promenljive)
p &&= q;  // isto što i p = p && q;(p i q logičke promenljive)
```

Kombinovani operator dodele += se može primeniti i na stringove. Podsetimo se da operator + u kontekstu stringova označava njihovo spajanje. Pošto je s+=t isto što i s=s+t, ako su s i t stringovi, onda je

[5] Sintaksa i semantika operatora dodele (sa varijacijama), kao i operatora izbora, preuzeti su iz jezika C.

nova vrednost stringa s jednaka njegovoj staroj vrednosti kojoj su dodati znakovi stringa t. Na primer, ako s ima vrednost "Dragan", onda se naredbom s+="a"; menja njegova vrednost u "Dragana".

Prioritet operatora

Ukoliko se u nekom izrazu pojavljuje više operatora i ako zagradama nije eksplicitno naznačen redosled njihovog izračunavanja, onda se oni primenjuju po unapred definisanom prioritetu. U tabeli 3.1 su dati svi do sada pomenuti operatori u opadajućem redosledu njihovog prioriteta — od onih najvišeg prioriteta koji se prvo primenjuju do onih najmanjeg prioriteta koji se poslednje primenjuju.

Unarni operatori:	+, −, ++, −−, eksplicitna konverzija
Množenje i deljenje:	*, /, %
Sabiranje i oduzimanje:	+, −
Relacijski operatori:	<, >, <=, >=
Jednakost i nejednakost:	==, !=
Logičko I:	&&
Logičko ILI:	||
Operator izbora:	?:
Operatori dodele:	=, +=, −=, *=, /=, %=

TABELA 3.1: Prioritet operatora u Javi od najvišeg do najnižeg.

Operatori u jednom redu tabele 3.1 imaju isti prioritet. Ukoliko se u nekom izrazu nalaze operatori istog prioriteta bez zagrada, onda se unarni operatori i operatori dodele izračunavaju redom zdesna na levo, dok se ostali operatori izračunavaju redom sleva na desno. Na primer, izraz x*y/z se izračunava postupno kao da stoji (x*y)/z, dok se x=y=z izračunava kao da stoji x=(y=z).

Pravila prioriteta operatora ne treba pamtiti napamet, jer se ona lako 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 redosled izračunavanja izraza koji se želi. Time se umnogome povećava čitljivost i smanjuje mogućnost suptilnih grešaka koje je teško otkriti.

Primer: vraćanje kusura

Problem vraćanja kusura je da se dati novčani iznos usitni sa *minimalnim* brojem novčića čije su vrednosti 1, 5, 10 i 25. Da bi ovaj problem uvek imao rešenje, podrazumeva se da na raspolaganju imamo neograničen broj svih novčića.

Postupak za rešavanje problema vraćanja kusura je onaj koji bismo primenili u svakodnevnom životu. Skoro bez razmišljanja, najpre bismo izabrali najveći novčić čija vrednost nije veća od datog iznosa. Zatim bismo izračunali preostali iznos koji treba usitniti — jednak razlici datog iznosa i vrednosti prvog novčića — i izabrali najveći novčić čija vrednost nije veća od tog preostalog iznosa. Ovaj postupak bismo ponovili sve dok ne dobijemo preostali iznos 0 za usitnjavanje.

Na primer, ako je iznos koji treba usitniti jednak 68, najpre biramo novčić čija je vrednost 25 (jer je on najveći i ne prevazilazi 68) i oduzimamo njegovu vrednost od 68 dobijajući preostali iznos jednak 43. Ponavljajući ovo za preostali iznos 43 koji treba usitniti, biramo ponovo najveći novčić koji ne prevazilazi 43, tj. opet jedan novčić čija je vrednost 25, i oduzimamo njegovu vrednost od 43 dobijajući sledeći preostali iznos jednak 18. Najveći novčić koji nije veći od 18 je sada onaj čija je vrednost 10, a preostali iznos za usitnjavanje je 8. Ovaj postupak nastavljamo sve dok ne dobijemo preostali iznos 0 koji treba usitniti. Na ovaj način dakle, iznos 68 ćemo usitniti u dva novčića vrednosti 25, jednog novčića vrednosti 10, jednog novčića vrednosti 5 i tri novčića vrednosti 1.

Da bismo ovaj postupak pretočili u Java program, primetimo da je broj najvećih novčića za neki (preostali) iznos jednak zapravo rezultatu deljenja tog iznosa sa vrednošću (aktuelno) najvećeg novčića, kao i da je preostali iznos jednak ostatku pri tom deljenju. Na primer, pošto za iznos 68 dva puta oduzimamo najveći novčić vrednosti 25, broj tih novčića za usitnjavanje je jednak `68 / 25` i preostali iznos 18 je jednak `68 % 25`.

U sledećem Java programu se za usitnjavanje datog novčanog iznosa koriste pet celobrojnih promenljivih za čuvanje potrebnih podataka. Promenljiva `iznos` ima višestruku ulogu. Ona sadrži početni, dati iznos za usitnjavanje, kao i sve preostale iznose koji se dobijaju u svakom koraku prethodno opisanog postupka nakon određivanja broja najvećeg novčića za aktuelni iznos. Iako smo za ove preostale iznose mogli uvesti nove promenljive, izabrali smo da „recikliramo" promenljivu `iznos` radi jed-

nostavnosti programa. (Druga, doduše ne tako važna korist jeste ušteda memorijskog prostora). Promenljive n25, n10, n5 i n1 sadrže izračunate brojeve novčića odgovarajuće vrednosti u rezultatu usitnjavanja.

LISTING 3.2: Vraćanje kusura.

```
import java.util.*;

public class Kusur {

   public static void main(String[] args) {
      int iznos;             // dati iznos za usitnjavanje
      int n25, n10, n5, n1;  // broj novčića usitnjenog iznosa

      // Ulaz programa se dobija preko tastature
      Scanner tastatura = new Scanner(System.in);

      // Učitavanje novčanog iznosa za usitnjavanje
      System.out.print("Unesite iznos za usitnjavanje: ");
      iznos = tastatura.nextInt();

      // Izračunavanje brojeva novčića usitnjenog iznosa
      n25 = iznos / 25;    // broj novčića vrednosti 25
      iznos = iznos % 25;  // preostali iznos za usitnjavanje
      n10 = iznos / 10;    // broj novčića vrednosti 10
      iznos = iznos % 10;  // preostali iznos za usitnjavanje
      n5 = iznos / 5;      // broj novčića vrednosti 5
      n1 = iznos % 5;      // broj novčića vrednosti 1

      // Prikazivanje rezultata na ekranu
      System.out.print("Minimalni broj novčića za ");
      System.out.println("dati iznos je: ");
      System.out.println(n25 + " novčića vrednosti 25");
      System.out.println(n10 + " novčića vrednosti 10");
      System.out.println(n5 + " novčića vrednosti 5");
      System.out.println(n1 + " novčića vrednosti 1");
   }
}
```

Primer: izračunavanje rate za kredit

Razmotrimo problem izračunavanja mesečne rate prilikom otplate uzetog kredita. Neka su p iznos uzetog kredita (principal), k godišnja kamatna stopa (pri čemu $k = 0.08$ označava kamatnu stopu od 8%) i r iznos mesečne rate za kredit. Koliki je iznos neotplaćenog kredita p_m posle m meseci?

Svakog meseca, iznos kredita se povećava usled zaračunavanja mesečne kamate i smanjuje usled otplate mesečne rate. Prema tome, ako uvedemo oznaku $a = 1 + k/12$, onda je:

- početni iznos kredita posle „nultog" meseca: $p_0 = p$
- iznos kredita posle prvog meseca: $p_1 = p_0 + (k/12)p_0 - r = ap_0 - r$
- iznos kredita posle drugog meseca: $p_2 = p_1 + (k/12)p_1 - r = ap_1 - r$
- i tako dalje ...
- iznos kredita posle m-tog meseca: $p_m = p_{m-1} + (k/12)p_{m-1} - r = ap_{m-1} - r$

Ako sada u izraz na desnoj strani za p_m zamenimo formulu za p_{m-1}, zatim u dobijeni izraz zamenimo formulu za p_{m-2}, i tako dalje obrnutim redom sve do formule za p_0, na kraju dobijamo formulu za iznos neotplaćenog kredita p_m posle m meseci:

$$p_m = p \cdot a^m - r \cdot \left(\frac{a^m - 1}{a - 1} \right), \quad a = 1 + \frac{k}{12}, \quad k \neq 0$$

Ako želimo da izvedemo formulu za aktuelni iznos mesečne rate r, treba pretpostaviti da se iznos neotplaćenog kredita p_m posle m meseci smanjio na 0. Zamenom $p_m = 0$ u prethodnu formulu za p_m i rešavanjem dobijene jednačine po r, dobijamo:

$$r = \frac{p \cdot a^m \cdot (a - 1)}{a^m - 1}, \quad a = 1 + \frac{k}{12}, \quad k \neq 0$$

Poznavajući ovu formulu, Java program koji izračunava mesečnu ratu za date vrednosti kredita, godišnje kamate i broja mesečnih rata, sada je lako napisati.

LISTING 3.3: Izračunavanje rate za kredit.

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

```
public class RataZaKredit {

   public static void main(String[] args) {

      // Ulaz programa se dobija preko tastature
      Scanner stdin = new Scanner(System.in);

      // Učitavanje kredita, kamate i broja rata
      System.out.print("Iznos kredita> ");
      double p = stdin.nextDouble();
      System.out.print("Godišnja kamata> ");
      double k = stdin.nextDouble();
      System.out.print("Broj mesečnih rata> ");
      int m = stdin.nextInt();

      // Izračunavanje mesečne rate
      double mk = k/12;  // mesečna kamata
      double aNaM = Math.pow(1 + mk, m);
      double r = (p * aNaM * mk) / (aNaM - 1);

      // Prikazivanje rezultata na ekranu
      System.out.println("Mesečna rata: " + r);
   }
}
```

Obratite pažnju u ovom programu na to da smo, radi promene, deklarisali promenljive tamo gde su potrebne. Da bismo izračunali vrednost a^m u formuli za mesečnu ratu, koristili smo funkciju `Math.pow()` za izračunavanje stepena u Javi. Pri tome smo rezultat poziva te funkcije sačuvali u promenljivoj `aNaM` da bismo malo ubrzali program ne pozivajući dva puta tu funkciju u narednoj naredbi dodele. Iz sličnog razloga smo uveli i promenljivu `mk` za vrednost mesečne kamate $k/12$, jer se ona dva puta koristi u daljem izračunavanju.

Glava

4

Upravljačke naredbe

U prethodnom poglavlju smo govorili o osnovnim elementima Java programa: promenljivim, izrazima, naredbama dodele i pozivima metoda. U ovom poglavlju ćemo se upoznati sa načinima kombinovanja ovih osnovnih elemenata radi pisanja složenijih programa sa interesantnijim funkcijama.

Mogućnost računara da obavlja vrlo složene zadatke se zasniva na samo nekoliko načina građenja složenih programskih struktura od prostih elemenata. U Javi postoje samo tri takve konstrukcije kojima se određuje tok izvršavanja programa: ***blok naredba***, ***naredbe grananja*** (`if` naredba, `if-else` naredba i `switch` naredba) i ***naredbe ponavljanja*** (`while` naredba, `do-while` naredba i `for` naredba).[1] Svaka od ovih upravljačkih konstrukcija smatra se jednom naredbom, ali je svaka zapravo *složena* naredba koja se sastoji od jedne ili više drugih (prostih ili složenih) naredbi. Stepen komponovanja složenih naredbi nije ograničen, tako da se mogu obrazovati upravljačke konstrukcije u programu za obavljanje najsloženijih zadataka.

Prema tome, naredbe u Javi mogu biti proste ili složene. Proste naredbe, u koje spadaju naredba dodele i naredba poziva metoda, predstavljaju osnovne gradivne elemente programa. Složene naredbe, u koje spadaju blok naredba, naredbe grananja i naredbe ponavljanja, predstavljaju načine za komponovanje prostih i složenih naredbi. Ove naredbe

[1] U stvari, dovoljne su samo tri od ovih naredbi (blok naredba, `if` naredba i `while` naredba) za pisanje bilo kog programa.

se nazivaju upravljačke naredbe jer one upravljaju redosledom kojim se naredbe programa izvršavaju.

Obratite pažnju na to da se jedino blok naredbom određuje sekvencijalni redosled izvršavanja naredbi od kojih se ona sastoji, kako smo to do sada podrazumevali za sve programe. Slika u opštem slučaju je ipak komplikovanija, jer se ostalim upravljačkim naredbama taj uobičajeni redosled izvršavanja može promeniti.

U nastavku ovog poglavlja ćemo posvetiti veću pažnju ovim i ostalim detaljima upravljačkih naredbi. Pri tome ćemo kroz primere opisati njihovu sintaksu (kako se ispravno pišu u programu) i semantiku (kako se izvršavaju u programu, odnosno kakav je njihov efekat).

4.1 Blok naredba

Blok naredba (ili kraće samo blok) je najprostiji način kombinovanja naredbi. Njena svrha je da samo grupiše niz naredbi u jednu naredbu. Opšti oblik bloka je:

```
{
   niz-naredbi
}
```

Drugim rečima, blok se sastoji od niza naredbi između otvorene vitičaste zagrade { i zatvorene vitičaste zagrade }.[2] Blok se obično nalazi unutar drugih složenih naredbi kada je potrebno da se više naredbi grupišu u jednu (složenu) naredbu. U opštem slučaju, međutim, blok se može nalaziti bilo gde je u programu moguće pisati neku naredbu.

Efekat izvršavanja blok naredbe je sekvencijalno izvršavanje naredbi od kojih se sastoji. Naime, kada se naiđe na otvorenu vitičastu zagradu bloka, sve naredbe u nizu koji sledi se izvršavaju redom jedna za drugom, dok se ne naiđe na zatvorenu vitičastu zagradu bloka.

Jedno mesto gde je blok neophodan je, kao što su pažljivi čitaoci možda primetili, metod `main()` programa. Niz naredbi od kojih se sastoji ovaj (a i svaki drugi) metod predstavlja zapravo blok, jer se taj niz nalazi unutar vitičastih zagrada. Na primer, blok koji predstavlja telo metoda `main()` programa *Zdravo* iz odeljka 2.5 je:

[2] Blok ne mora zapravo da sadrži nijednu naredbu, već samo par vitičastih zagrada. Takav blok se naziva ***prazan blok*** i ponekad može biti koristan u programu.

```
{
   Scanner tastatura = new Scanner(System.in);
   System.out.print("Kako se zovete: ");
   String ime = tastatura.nextLine();
   System.out.print("Koliko imate godina: ");
   int god = tastatura.nextInt();
   System.out.println("Zdravo " + ime + "!");
   System.out.println(god + " su najlepše godine.");
}
```

Ovaj blok čini niz od 7 naredbi koje se redom izvršavaju jedna za drugom od početka bloka (znaka {) do kraja bloka (znaka }).

U drugom primeru blok naredbe se zamenjuje vrednosti celobrojnih promenljivih x i y koje su deklarisane negde 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 naredbi koje se redom izvršavaju. Primetimo da je pomoćna promenljiva z deklarisana unutar ovog bloka (kao što su i potrebne promenljive deklarisane u prvom primeru 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 je potpuno nedostupna van tog bloka, jer se takva promenljiva „uništava" nakon izvršavanja njenog bloka. (Tačnije, memorija koju zauzima promenljiva se oslobađa za druge svrhe.) Pored toga što se malo štedi u memoriji, time se sprečavaju mnogo ozbiljniji problemi nenamerne upotrebe iste promenljive za druge svrhe. Za promenljivu deklarisana unutar nekog bloka se kaže da je ***lokalna*** za taj blok ili da je taj blok njena ***oblast važenja***. (Tačnije, oblast važenja lokalne promenljive je deo bloka od mesta deklaracije promenljive do kraja bloka.)

Blok naredba sama po sebi ne utiče na sekvencijalni tok izvršavanja programa. Ovaj „normalni" način izvršavanja programa se može promeniti naredbama grananja i ponavljanja, o kojima govorimo u nastavku.

4.2 Naredbe grananja

Naredbe grananja obezbeđuju alternativne puteve toka izvršavanja programa zavisno od vrednosti nekog izraza. Na primer, ako u nekoj tački programa korisnik treba da izabere neku opciju, tada zavisno od tog izbora izvršavanje programa treba da se nastavi potpuno različitim putevima. Naredbe grananja omogućavaju da se izabere jedan niz naredbi za izvršavanje, a drugi niz naredbi (ili više njih) da se potpuno preskoči i nikad ne izvrši.

U programu `if` i `if-else` naredba služe za izbor jednog od dva alternativna niza naredbi za izvršavanje zavisno od toga da li je vrednost datog logičkog izraza tačno ili netačno. Treća naredba grananja, `switch` naredba, služi za izbor jednog od više alternativnih nizova naredbi za izvršavanje zavisno od vrednosti datog celobrojnog ili znakovnog izraza.[3]

Naredbe `if` i `if-else`

Opšti oblik `if` naredbe u Javi je:

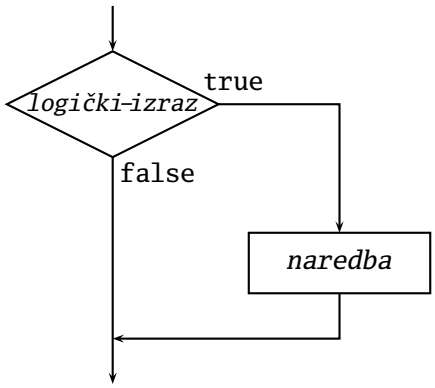
```
if ( logički-izraz )
   naredba
```

Izvršavanje `if` naredbe se izvodi tako što se najpre izračunava vrednost *logičkog-izraza* u zagradi. Zatim se, ako je ta vrednost `true`, izvršava *naredba* koja se nalazi unutar `if` naredbe; ako je ta vrednost `false`, ništa se dodatno ne izvršava, odnosno ova *naredba* se preskače. Time se u oba slučaja izvršavanje `if` naredbe završava i program se dalje nastavlja od naredbe koja sledi iza `if` naredbe. Slikovito, izvršavanje `if` naredbe možemo predstaviti blok-dijagramom koji je prikazan na slici 4.1.

Često je u programu potrebno uraditi nešto drugo kada je vrednost logičkog izraza jednaka `false`, a ne samo preskočiti naredbu unutar `if` naredbe. Za takve slučajeve služi `if-else` naredba. Opšti oblik `if-else` naredbe je:

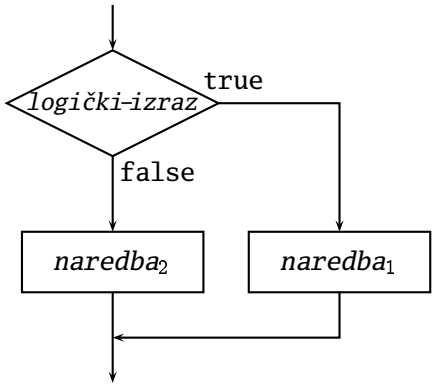
```
if ( logički-izraz )
   naredba1
else
   naredba2
```

[3] Ili izraza čija je vrednost nabrojivog tipa, o čemu do sada nismo govorili.



SLIKA 4.1: Izvršavanje `if` naredbe.

Izvršavanje `if-else` naredbe je slično izvršavanju `if` naredbe, osim što u slučaju kada je vrednost *logičkog-izraza* u zagradi jednaka `false`, izvršava se $naredba_2$ i preskače $naredba_1$. U drugom slučaju, kada je vrednost *logičkog-izraza* jednaka `true`, izvršava se $naredba_1$ i preskače $naredba_2$. Time se u oba slučaja izvršavanje `if-else` naredbe završava i program se dalje nastavlja od naredbe koja sledi iza `if-else` naredbe. Ovo je predstavljeno blok-dijagramom na slici 4.2.



SLIKA 4.2: Izvršavanje `if-else` naredbe.

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

U sledećem primeru se koristi if naredba:

```
if ( n % 2 == 0 )  // n je paran broj?
   System.out.println("n je paran broj");
System.out.println("n = " + n);
```

Kada se ovaj programski fragment izvršava, najpre se izračunava logički izraz n%2==0 u zagradi if naredbe. To znači da se izračunava da li je ostatak od n podeljeno sa 2 jednako nuli, ili ekvivalentno da li je n paran broj. Ako je to true (tačno), izvršava se naredba:

```
System.out.println("n je paran broj");
```

unutar if naredbe, odnosno prikazuje se tekst "n je paran broj" na ekranu. Time se ujedno i završava izvršavanje if naredbe. Ako logički izraz n%2==0 daje false (netačno), ništa se dalje ne radi i odmah se završava izvršavanje if naredbe. Nakon završetka izvršavanja if naredbe u oba slučaja, izvršavanje programa se nastavlja od naredbe koja sledi iza if naredbe, odnosno u ovom primeru je to naredba:

```
System.out.println("n = " + n);
```

kojom se prikazuje vrednost promenljive n na ekranu.

Obratite pažnju na to da smo naredbu:

```
System.out.println("n je paran broj");
```

malo uvukli unutar if naredbe čiji je sastavni deo. Za Java prevodilac je ovo bez značaja, ali smo iskoristili slobodni format jezika Java da bismo naznačili da je ta naredba deo složene naredbe i da se neće uvek izvršiti. Ovo je jedna od odlika dobrog stila programiranja.

U prethodnom primeru if naredbe se ništa ne izvršava kada je n neparan broj. Ukoliko je potrebno nešto uraditi i u tom slučaju, mora se koristiti if-else naredba. Na primer:

```
if ( n % 2 == 0 )  // n je paran broj?
   System.out.println("n je paran broj");
 else
   System.out.println("n je neparan broj");
System.out.println("n = " + n);
```

Ovde se opet najpre izračunava logički izraz n%2==0 u zagradi if-else naredbe. Ako to daje true (tačno), izvršava se naredba:

```
System.out.println("n je paran broj");
```

i preskače se else deo. Ali ako logički izraz u zagradi daje false (netačno), preskače se if deo i izvršava naredba:

```
System.out.println("n je neparan broj");
```

u else delu. Nakon završetka izvršavanja if-else naredbe u oba slučaja, izvršavanje programa se nastavlja od naredbe koja sledi iza if-else naredbe, odnosno i u ovom primeru je to naredba:

```
System.out.println("n = " + n);
```

kojom se prikazuje vrednost promenljive n na ekranu.

Obratimo pažnju na još neke detalje naredbi if i if-else. Pre svega, *naredba*, $naredba_1$ ili $naredba_2$ u opštem obliku ovih naredbi može biti blok naredba tako da if naredba može imati ovaj oblik:[4]

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

a if-else naredba ovaj oblik:

```
if ( logički-izraz ) {
   niz-naredbi
}
else {
   niz-naredbi
}
```

U sledećem primeru se koristi ovakav oblik if naredbe da bi se zamenile vrednosti celobrojnih promenljivih x i y, ali samo ako je početno x veće od y:

```
if ( x > y ) {
   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
}
```

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

Primetimo da posle izvršavanja ove if naredbe možemo biti sigurni da je vrednost promenljive x sigurno manja ili jednaka vrednosti promenljive y.

U sledećem programskom fragmentu se računa obim kruga za dati poluprečnik. Pri tome se koristi if-else naredba radi provere ispravnosti vrednosti poluprečnika koju unosi korisnik:

```
Scanner tastatura = new Scanner(System.in);
System.out.print("Unesite poluprečnik: ");
double r = tastatura.nextDouble();
if ( r <= 0 )
   System.out.println("Greška: poluprečnik nije pozitivan!");
else {
   double obim = 2*r*Math.PI;
   System.out.print("Obim kruga tog poluprečnika ");
   System.out.println("je: " + obim);
}
```

U ovom primeru se if deo if-else naredbe sastoji od jedne naredbe i zato ne mora da bude unutar para vitičastih zagrada. (Ne mora, ali može! U stvari, neki programeri uvek koriste blokove za sastavne delove if i if-else naredbi, čak i kada se oni sastoje od samo jedne naredbe.) Sa druge strane, else deo if-else naredbe u primeru se sastoji od tri naredbe i zato se mora nalaziti unutar para vitičastih zagrada u formi bloka.

Drugi detalj u vezi sa opštim oblikom if i if-else naredbi je da *naredba*, $naredba_1$ ili $naredba_2$ mogu biti bilo koje naredbe jezika Java. To specifično znači da one mogu biti isto tako if i if-else naredbe. Jedan primer ove mogućnosti koja se naziva ***ugnježđavanje*** jeste:

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

Međutim, pošto je Java jezik slobodnog formata, ovo se skoro uvek piše u obliku:

```
if ( logički-izraz1 )
```

```
    naredba_1
else if ( logički-izraz_2 )
    naredba_2
else
    naredba_3
```

Ovaj oblik naredbe grananja omogućuje izbor jedne od tri alternative za izvršavanje: *naredbe*$_1$, *naredbe*$_2$ ili *naredbe*$_3$. Naime, na uobičajeni način, najpre se izračunava *logički-izraz*$_1$. Ako je njegova vrednost `true`, izvršava se *naredba*$_1$ i preskače sve ostalo. Ako je njegova vrednost `false`, preskače se *naredba*$_1$ i izvršava se druga, ugnježđena `if-else` naredba. To znači da se izračunava *logički-izraz*$_2$ i, zavisno od toga da li je njegova vrednost `true` ili `false`, zatim se izvršava *naredba*$_2$ ili *naredba*$_3$.

Pošto nivo ugnježđavanja može biti proizvoljan, ništa nas ne sprečava da nastavimo taj postupak i tako dobijemo naredbu višestrukog grananja u opštem obliku:

```
if ( logički-izraz_1 )
    naredba_1
else if ( logički-izraz_2 )
    naredba_2
else if ( logički-izraz_3 )
    naredba_3
        ⋮
else if ( logički-izraz_n )
    naredba_n
else
    naredba_{n+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 `true`. Zatim se izvršava njegova pridružena naredba i preskače se sve ostalo. Ako vrednost nijednog logičkog izraza nije `true` (tj. svi logički izrazi daju `false`), izvršava se naredba u `else` delu. U stvari, ovaj `else` deo nije obavezan, pa ako nije naveden i nijedan logički izraz nije `true`, nijedna naredba 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 njenog

izvršavanja, osim što se izvršava blok (niz naredbi) pridružen prvom logičkom izrazu koji ima vrednost true.

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 kod ugnježđavanja koji se popularno naziva „viseći" else deo. Posmatrajmo sledeći programski fragment:

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

Ovako kako je napisan, fragment sugeriše da imamo jednu if–else naredbu čiji se if deo sastoji od druge if naredbe. 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, efekat datog fragmenta je isti kao da smo napisali:

```
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. Ako x ima vrednost manju od 0, izvršavanje pod pretpostavkom

prvobitne interpretacije bi dovelo do prikazivanja teksta "Drugi slučaj" na ekranu. Ali pravi efekat je zapravo da se preskače ugnježđena if-else naredba, odnosno ništa se ne dobija na ekranu.

Ukoliko se zaista želi efekat koji sugeriše prvobitna interpretacija, ugnježđena if naredba se može pisati unutar bloka:

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

ili se ona može pisati u obliku if-else naredba čiji se else deo sastoji od takozvane ***prazne naredbe*** označene samo tačkom-zarez:

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

Primer: sortiranje tri vrednosti

Radi ilustracije if i if-else naredbi, sada ćemo pokazati jednostavan program koji sortira neke tri vrednosti. Preciznije, program najpre od korisnika dobija tri ulazne celobrojne vrednosti u proizvoljnom redosledu, a zatim ih prikazuje u rastućem redosledu. Na primer, ako su vrednosti 69, 17 i 23 ulaz programa, unete tim redom, onda izlaz programa trebaju da budu te vrednosti u rastućem redosledu: 17, 23 i 69.

Ako pretpostavimo da su x, y i z celobrojne promenljive koje sadrže tri ulazne vrednosti, onda za prikazivanje njihovih vrednosti u rastućem redosledu možemo postupiti na više načina. Jedan način je da najpre odredimo gde se, recimo, vrednost promenljive x nalazi u rastućem nizu. Ona dolazi na prvom mestu ako je manja i od y i od z. Ona dolazi na poslednjem mestu ako je veća i od y i od z. U preostalom slučaju, ona je u sredini. Zatim, u svakom od ova tri slučaja, treba odrediti međusobni redosled vrednosti promenljivih y i od z.

Sledi kompletan Java program u kojem se primenjuje ovaj pristup.

LISTING 4.1: Sortiranje tri vrednosti.

```
import java.util.*;

public class Sort3 {

   public static void main(String[] args) {

      // Učitavanje tri vrednosti preko tastature
      Scanner tastatura = new Scanner(System.in);
      System.out.print("Unesite tri broja: ");
      int x = tastatura.nextInt();
      int y = tastatura.nextInt();
      int z = tastatura.nextInt();

      // Prikazivanje sortiranih vrednosti na ekranu
      System.out.print("Rastući redosled unetih brojeva je: ");
      if (x < y && x < z) {       // x je na prvom mestu
         if (y < z)
            System.out.println( x + " " + y + " " + z );
         else
            System.out.println( x + " " + z + " " + y );
      }
      else if (x > y && x > z) {  // x je na poslednjem mestu
         if (y < z)
            System.out.println( y + " " + z + " " + x );
         else
            System.out.println( z + " " + y + " " + x );
      }
      else {                      // x je u sredini
         if (y < z)
            System.out.println( y + " " + x + " " + z );
         else
            System.out.println( z + " " + x + " " + y );
      }
   }
}
```

Obratite pažnju u programu na to da su vitičaste zagrade nepotrebne u svakom od tri slučaja u kojem se određuje međusobni redosled y i z. To

je zato što se međusobni redosled y i z određuje jednom if naredbom, a blok je neophodan samo ako bi se to radilo pomoću dve ili više naredbi. S druge strane, blok se može sastojati samo od jedne naredbe, pa smo to ovde iskoristili radi bolje preglednosti.

Isti problem se u programiranju može često rešiti na više načina. Da bismo ovu činjenicu ilustrovali za sortiranje tri vrednosti, pokazaćemo drugi način na koji to možemo rešiti. Na primer, možemo početi sa ispitivanjem da li je x veće od y. Ako je to slučaj, zamenjujemo vrednosti ovih promenljivih da bismo dalje osigurali suprotan slučaj. Onda upoređivanjem z sa x ili y možemo lako dobiti rastući redosled tri ulazne vrednosti. Relevantni deo programa u kojem je primenjen ovaj pristup ima oblik:

```
if (x > y) {  // zameniti vrednosti x i y
   int a;     // pomoćna promenljiva
   a = x;
   x = y;
   y = a;
}
// x je sigurno ispred y
if (z < x)       // z je na prvom mestu
   System.out.println( z + " " + x + " " + y );
else if (z > y)  // z je na poslednjem mestu
   System.out.println( x + " " + y + " " + z );
else             // z je u sredini
   System.out.println( x + " " + z + " " + y );
```

Naredba switch

Treća naredba grananja, switch naredba, koristi se mnogo manje od prethodne dve, ali u nekim slučajevima prirodnije odražava logiku višestrukog grananja. Najčešći oblik switch naredbe je:

```
switch ( izraz ) {
   case konstanta1 :
      niz-naredbi1
      break;
   case konstanta2 :
      niz-naredbi2
      break;
```

```
        ⋮
    case konstanta_n :
        niz-naredbi_n
        break;
    default:
        niz-naredbi_{n+1}
}
```

Ovom naredbom se najpre izračunava *izraz* koji može biti celobrojnog ili znakovnog tipa. Zatim se zavisno od dobijene vrednosti izraza izvršava *niz-naredbi* koji je pridružen jednom od case delova 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 *niz-naredbi* pridružen pronađenom case delu. Poslednji slučaj u switch naredbi može opciono biti default deo koji se izvršava ukoliko izračunata vrednost *izraza* nije jednaka nijednoj *konstanti* u case delovima. Najzad, ukoliko default deo nije naveden i izračunata vrednost *izraza* nije jednaka nijednoj *konstanti* u case delovima, ništa se dodatno ne izvršava i izvršavanje switch naredbe se odmah završava.

Na kraju svakog case dela se obično, ali ne uvek, navodi break naredba. Efekat break naredbe je da se prekine izvršavanje odgovarajućeg case dela, a time i cele switch naredbe. Ako se na kraju case dela koji se izvršava ne nalazi break naredba, prelazi se na izvršavanje narednog case dela unutar switch naredbe. Ovaj postupak se nastavlja sve dok se ne naiđe na prvu break naredbu ili se ne dođe do kraja switch naredbe.

U nekom case delu switch naredbe se može potpuno izostaviti odgovarajući *niz-naredbi* i break naredba. Ako iza tog „praznog" case dela dolazi drugi „pravi" case deo, onda *niz-naredbi* ovog drugog dela odgovara dvema konstantama. To prosto znači da će se ovaj *niz-naredbi* izvršiti kada je vrednost *izraza* jednaka jednoj od te dve konstante.

Naredni, prilično veštački primer pokazuje neke od prethodno pomenutih mogućnosti switch naredbe. Obratite pažnju i na to da se konstante u case delovima mogu pisati u proizvoljnom redosledu, pod uslovom da su sve različite.

```
// n je celobrojna promenljiva
switch ( 2*n ) {
   case 1:
      System.out.println("Ovo se neće nikad izvršiti,");
```

```
            System.out.println("jer je 2n paran broj!");
            break;
         case 2:
         case 4:
         case 8:
            System.out.println("n ima vrednost 1, 2 ili 4,");
            System.out.println("jer je 2n jednako 2, 4 ili 8.");
            break;
         case 6:
            System.out.println("n ima vrednost 3.");
            break;
         case 5:
         case 3:
            System.out.println("I ovo je nemoguće!");
            break;
      }
```

Primer: rad sa menijem

Jedna od čestih primena switch naredbe je u programima čiji se rad zasniva na menijima. Program koji sledi prikazuje početni meni sa opcijama na ekranu i od korisnika očekuje izbor jedne od opcija. Zavisno od izabrane opcije, program zatim prikazuje odgovarajuću poruku. Naravno, ovo nije mnogo praktičan primer, ali pokazuje postupak koji se lako može prilagoditi u stvarnim programima.

LISTING 4.2: Rad sa menijem.

```
import java.util.*;

public class Meni {

   public static void main(String[] args) {

      // Prikazivanje menija na ekranu
      System.out.println("Opcije menija su:");
      System.out.println("    1. Predjelo");
      System.out.println("    2. Supe i čorbe");
      System.out.println("    3. Glavno jelo");
      System.out.println("    4. Salate");
      System.out.println("    5. Poslastice");
```

```
      System.out.println("    6. Piće");
      System.out.print("Unesite broj opcije koju želite: ");

      // Učitavanje izabrane opcije menija
      Scanner tastatura = new Scanner(System.in);
      int brojOpcije = tastatura.nextInt();

      switch (brojOpcije) {
         case 1:
            System.out.println("Izabrali ste \"Predjelo\".");
            break;
         case 2:
            System.out.println("Izabrali ste \"Supe i čorbe\".");
            break;
         case 3:
            System.out.println("Izabrali ste \"Glavno jelo\".");
            break;
         case 4:
            System.out.println("Izabrali ste \"Salate\".");
            break;
         case 5:
            System.out.println("Izabrali ste \"Poslastice\".");
            break;
         case 6:
            System.out.println("Izabrali ste \"Piće\".");
            break;
         default:
            System.out.println("Greška: pogrešna opcija!");
      }
   }
}
```

Primer: određivanje sutrašnjeg datuma

U ovom primeru ćemo pokazati malo složeniji Java program kojim se određuje sutrašnji datum za neki dati datum. Preciznije, ulaz programa su tri cela broja koja određuju ispravan datum u formatu dan, mesec, godina: na primer, "16 4 2008". Izlaz program treba da budu tri broja koja predstavljaju sutrašnji datum za dati datum, odnosno u ovom primeru je to "17 4 2008".

Sutrašnji datum je lako odrediti ukoliko je dan datog datuma manji od broja dana u mesecu datog datuma — onda je samo dan sutrašnjeg datuma za jedan veći. Međutim, stvari se komplikuju kada je dan datog datuma jednak broju dana u mesecu datog datuma: na primer, "30 4 2008". Onda je dan sutrašnjeg datuma 1, ali je mesec sutrašnjeg datuma za jedan veći. Međutim, izuzetak od toga je kada je dati datum, recimo, "31 12 2008", jer su onda sutrašnji dan i mesec jednaki 1, ali je godina za jedan veća.

Da bismo otkrili o kojem se od ovih slučajeva radi u programu, najpre određujemo ukupan broj dana u mesecu datog datuma. U tu svrhu koristimo promenljivu maxDanaUMesecu koja dobija odgovarajuću vrednost switch naredbom na osnovu meseca datog datuma. Nakon što odredimo ukupan broj dana u aktuelnom mesecu, ispitivanjem da li je dan datog datuma manji od ukupnog broja dana u mesecu datog datuma, kao i da li je mesec datog datuma manji od decembra, nije teško odrediti sutrašnji datum.

LISTING 4.3: Određivanje sutrašnjeg datuma.

```
import java.util.*;

public class Sutra {

   public static void main(String[] args) {

      // Učitavanje datuma u formatu d m g preko tastature
      System.out.print("Unesite datum u formatu d m g: ");
      Scanner tastatura = new Scanner(System.in);
      int dan = tastatura.nextInt();    // dan datog datuma
      int mesec = tastatura.nextInt();  // mesec datog datuma
      int godina = tastatura.nextInt(); // godina datog datuma

      // Određivanje broja dana u aktuelnom mesecu
      int maxDanaUMesecu = 0;  // ukupan broj dana u datom mesecu
      switch (mesec) {
         case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            maxDanaUMesecu = 31;
            break;
         case 4: case 6: case 9: case 11:
            maxDanaUMesecu = 30;
            break;
```

```
         case 2:
            if (godina%4==0 && godina%100!=0 || godina%400==0)
               maxDanaUMesecu = 29;
            else
               maxDanaUMesecu = 28;
            break;
         default:
            System.out.println("Pogrešan mesec!\n");
            System.exit(0);
      }

      // Izračunavanje sutrašnjeg datuma
      if (dan < maxDanaUMesecu)
         dan++;
      else {
         dan = 1;
         if (mesec < 12)
            mesec++;
         else {
            mesec = 1;
            godina++;
         }
      }
      // Prikazivanje sutrašnjeg datuma na ekranu
      System.out.print("Sutrašnji datum je: ");
      System.out.println(dan + " " + mesec + " " + godina);
   }
}
```

Ovaj program zaslužuje nekoliko napomena:

- Celobrojne promenljive dan, mesec i godina služe za predstavljanje kako ulaznog datog datuma tako i izlaznog sutrašnjeg datuma, prosto radi jednostavnosti.
- U slučaju kada je februar mesec datog datuma, njegov ukupan broj dana moramo odrediti prema tome da li se radi o prestupnoj godini. U svakodnevnoj praksi smo navikli da smatramo da je godina prestupna ako je deljiva sa 4. U Javi se to može izaziti uslovom da je ostatak pri deljenju godine sa 4 jednak 0 (godina%4==0). Međutim, to nije potpuno tačan uslov, nego je godina prestupna ukoliko je ona ili deljiva sa 4 i nije deljiva sa 100 ili je ona deljiva

sa 400. U programu se koristi ovaj, tačan kriterijum za prestupnost neke godine. Obratite pažnju na zapis odgovarajućeg logičkog izraza i na njegovu ispravnost zahvaljujući pravilima prioriteta operatora koji učestvuju u njemu.

- U `default` slučaju `switch` naredbe se koristi metod `System.exit()` iz odeljka 3.4 za prekid rada programa, jer program ne može dalje nastaviti zbog pogrešnog ulaznog podatka.

- Najzad, pažljivi čitaoci su možda primetili naredbu deklaracije

```
int maxDanaUMesecu = 0;
```

na početku dela u kojem se određuje broj dana u aktuelnom mesecu. Naravno, deklaracija promenljive `maxDanaUMesecu` je na tom mestu neophodna, ali se postavlja pitanje da li je neophodna i njena inicijalizacija (0 je ovde proizvoljno izabrana vrednost)? Odgovor je potvrdan, jer se bez njene inicijalizacije dobija poruka o greški kod korišćenja promenljive `maxDanaUMesecu` u `if` naredbi za izračunavanje sutrašnjeg datuma.

Razlog ove neočekivane greške je taj što se u Javi vrednost promenljive može koristiti samo ako je promenljivoj prethodno *nesumnjivo* dodeljena neka vrednost. To znači da Java prevodilac prilikom prevođenja programa mora zaključiti da je promenljivoj prethodno dodeljena vrednost kada se ona koristi u programu. Nažalost, Java prevodilac ne poznaje dovoljno značenje pojedinih programskih elemenata (semantiku), već samo način njihovog ispravnog pisanja (sintaksu), pa ponekad može izvesti pogrešan zaključak.

U našem primeru, u `default` slučaju `switch` naredbe se ne dodeljuje nijedna vrednost promenljivoj `maxDanaUMesecu`, pa Java prevodilac pogrešno zaključuje da ona može biti nedefinisana u `if` naredbi iza `switch` naredbe. To naravno nije moguće, jer se u `default` slučaju `switch` naredbe program prekida i zato se nikad ne može izvršiti `if` naredba iza `switch` naredbe sa nedefinisanom vrednošću promenljive `maxDanaUMesecu`. Najjednostavniji način da se izbegne ovaj problem u ovom primeru je da se promenljivoj `maxDanaUMesecu` dodeli inicijalna vrednost, kako bi se Java prevodilac naterao da zaključi da ova promenljiva ima vrednost u svim slučajevima `switch` naredbe.

4.3 Naredbe ponavljanja

Naredbe ponavljanja obezbeđuju izvršavanje neke (blok) naredbe više puta. Broj ponavljanja te naredbe kod naredbi ponavljanja se kontroliše vrednošću jednog logičkog izraza. To znači da se naredba ponavlja sve dok je vrednost tog logičkog izraza jednaka `true`, a da se ponavljanje prekida u trenutku kada vrednost tog logičkog izraza postane `false`. Ponovljeno izvršavanje iste naredbe se može, u principu, izvoditi beskonačno, ali takva ***beskonačna petlja*** obično predstavlja grešku u programu.

Naredba `while`

Opšti oblik `while` naredbe je:

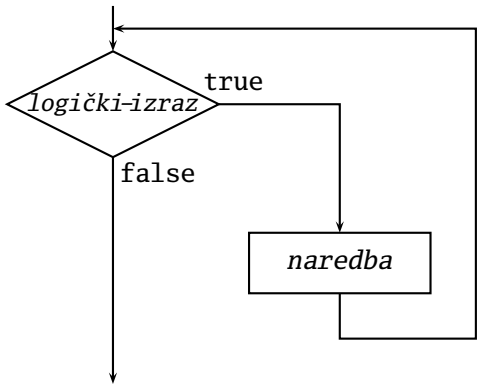
```
while ( logički-izraz )
   naredba
```

Pošto *naredba* unutar `while` naredbe može biti, a obično i jeste, blok naredba, `while` naredbe ima često ovaj oblik:

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

Izvršavanje `while` naredbe se izvodi tako što se najpre izračunava vrednost *logičkog-izraza* u zagradi. U jednom slučaju, ako je ta vrednost `false`, odmah se prekida izvršavanje `while` naredbe. To znači da se preskače ostatak `while` naredbe i program se normalno nastavlja od naredbe koja sledi iza `while` naredbe. U drugom slučaju, ako izračunavanje *logičkog-izraza* daje `true`, najpre se izvršava *naredba* ili *niz-naredbi* u bloku, a zatim se ponovo izračunava *logički-izraz* u zagradi i ponavlja isti postupak. To jest, ako *logički-izraz* daje `false`, prekida se izvršavanje `while` naredbe; ako *logički-izraz* daje `true`, izvršava se *naredba* ili *niz-naredbi* u bloku, opet se izračunava *logički-izraz* u zagradi i zavisno od njegove vrednosti se ili ponavlja ovaj postupak ili se prekida izvršavanje `while` naredbe.

Kratko rečeno, efekat `while` naredbe je ponavljanje jedne *naredbe* ili *niza-naredbi* u bloku sve dok je *logički-izraz* tačan. U trenutku kada on postane netačan, `while` naredba se završava i nastavlja se normalno izvršavanje programa od naredbe koja sledi iza `while` naredbe. Slikoviti prikaz izvršavanja `while` naredbe je prikazan blok-dijagramom na slici 4.3.



SLIKA 4.3: Izvršavanje while naredbe.

Kako se vidi sa slike 4.3, na desnoj strani blok-dijagrama toka izvršavanja while naredbe se obrazuje petlja, pa se često govori o while ***petlji***. Isto tako, *naredba* ili *niz-naredbi* bloka unutar while petlje se naziva ***telo petlje***, a jedno izvršavanje tela petlje se naziva jedna ***iteracija*** while petlje. Najzad, *logički-izraz* koji se izračunava na početku svake iteracije se naziva ***uslov prekida*** (ili ***nastavka***) petlje.

U sledećem prostom primeru while naredbe se na ekranu prikazuju brojevi 0, 1, 2, ..., 9 u jednom redu:

```
int broj = 0;         // početni broj koji se prikazuje
while (broj < 10) {  // ponoviti sve dok je broj manji od 10
   System.out.print(broj + " "); // prikazati broj na ekranu
   broj = broj + 1;              // preći na sledeći broj
}
System.out.println(); // pomeriti kursor u novi red
```

Ovde se na početku celobrojna promenljiva broj inicijalizuje prvom vrednošću 0. Zatim dolazi while naredba, kod koje se najpre određuje vrednost logičkog izraza broj<10. Vrednost ovog izraza je true, 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 while naredba je ponovno izračunavanje logičkog izraza broj<10. Vrednost ovog izraza je opet true, 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 vrednost 1 promenljive broj i ta vrednost se uvećava za 1. Nova vrednost promenljive broj je dakle 2. Pošto izvršavanje while naredbe još nije završeno, ponovo se izračunava logički izraz broj<10. Opet se dobija njegova vrednost true, 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 trenutna 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 false, jer 10 nije manje od 10, a to predstavlja uslov prekida izvršavanja while naredbe. Nakon toga se izvršavanje programa nastavlja od naredbe koja se nalazi iza while naredbe, što u našem primeru dovodi do pomeranja kursora na ekranu u novi red.

Na kraju, razjasnimo još neke detalje u vezi sa while naredbom. Prvo, šta se događa ako je *logički-izraz* (tj. uslov nastavka) while petlje netačan pri prvom njegovom izračunavanju, kada se telo petlje još nijedanput nije izvršilo? U tom slučaju, izvršavanje while petlje se odmah završava i zato se telo petlje nikad ne izvršava. To znači da broj iteracija while petlje može biti proizvoljan, uključujući nulu.

Drugo, šta se događa ako *logički-izraz* (uslov nastavka) while petlje postane netačan negde u sredini izvršavanja tela petlje? Da li se while petlja odmah tada završava? Odgovor je ne, nego se telo petlje izvršava do kraja. Tek kada se onda ponovo izračuna *logički-izraz* i dobije njegova netačna vrednost, dolazi do završetka izvršavanja while petlje.

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

Primer: izračunavanje akumulirane štednje

U sledećem Java programu se prikazuje iznos štednje koja se akumilira na kraju svake godine. Početni iznos štednje i godišnja kamata, kao i period izveštaja, dobijaju se od korisnika kao ulazne veličine programa.

LISTING 4.4: Izračunavanje akumulirane štednje.

```
import java.util.*;

public class Štednja {

   public static void main(String[] args) {

      double stanje;        // akumulirani iznos štednje
      double kamatnaStopa;  // godišnja kamatna stopa na štednju
      int period;           // broj godina za izveštaj

      // Učitavanje početnog depozita, kamate i perioda izveštaja
      Scanner tastatura = new Scanner(System.in);
      System.out.print ("Unesite početni depozit: ");
      stanje = tastatura.nextDouble();
      System.out.print ("Unesite godišnju kamatu: ");
      kamatnaStopa = tastatura.nextDouble();
      System.out.print ("Unesite broj godina izveštaja: ");
      period = tastatura.nextDouble();

      // Prikazivanje akumulirane štednje nakon svake godine
      int godina = 1;  // godina za koju se prikazuje štednja
      while (godina <= period) {
         // Izračunavanje kamate za ovu godinu
         int kamata = stanje * kamatnaStopa;
         // Dodavanje kamate na prethodno stanje
         stanje = stanje + kamata;
         // Prikazivanje novog stanja
         System.out.print("Štednja nakon " + godina + "iznosi €");
         System.out.printf("%1.2f\n", stanje);
         // Sledeća godina izveštaja
         godina++;
      }
   }
}
```

Primer: igra pogađanja broja

U ovom primeru ćemo napisati Java program koji igra igru pogađanja broja sa korisnikom. Program će izabrati slučajan ceo broj između 1 i

100, a korisnik treba da ga pogodi. Ako broj nije pogođen, program treba da pomogne korisniku prikazujući odgovarajuću poruku da je zamišljeni broj manji ili veći od pokušanog.

LISTING 4.5: Igra pogađanja broja.

```
import java.util.*;

public class PogađanjeBroja {

   public static void main(String[] args) {

      int zamišljenBroj; // broj koji je računar izabrao
      int pokušanBroj;   // broj koji je korisnik pokušao
      boolean pogodak;   // indikator da li je korisnik pogodio

      Scanner tastatura = new Scanner(System.in);

      zamišljenBroj = (int)(100*Math.random()) + 1;
      pogodak = false;  // broj nije pogođen na početku

      System.out.println("Zamislio sam ceo broj između 1 i 100.");
      System.out.println("Pokušajte da ga pogodite.\n");

      while (!pogodak) {
         System.out.print("Pogodite broj> ");
         pokušanBroj = tastatura.nextInt();
         if (pokušanBroj < zamišljenBroj)
            System.out.println("Zamislio sam veći broj :-(");
         else if (pokušanBroj > zamišljenBroj)
            System.out.println("Zamislio sam manji broj :-(");
         else {
            System.out.println("Bravo, pogodili ste broj :-)");
            pogodak = true;
         }
      }
   }
}
```

Obratite pažnju u programu na dve stvari. Prvo, slučajan broj između 1 i 100 je generisan slično načinu koji smo opisali na strani 66 za generisanje slučajnog broja između 1 i 6. Drugo, logička promenljiva pogodak

je potrebna da bismo prekinuli izvršavanje while petlje kada korisnik pogodi zamišljeni broj. Pošto se uslov prekida proverava na početku svake iteracije, pa i prve, neophodno je ovu promenljivu na početku inicijalizovati vrednošću false kako bi se prva iteracija svakako izvršila.

Primer: izračunavanje proseka niza brojeva

Tipični problem u programiranju koji se rešava petljom je sabiranje niza ulaznih brojeva. Klasičan način rešavanja ovog problema je da se najpre inicijalizuje konačna suma tih brojeva nulom, a zatim da se u svakoj iteraciji petlje dodaje sledeći član niza brojeva parcijalnoj sumi izračunatoj do tada za prethodne brojeve niza.

Da bismo ovo ilustrovali u Javi i da bi problem bio malo interesantniji, u primeru ćemo ovakav pristup primeniti zapravo za izračunavanje proseka niza ulaznih brojeva. Preciznije, korisnik treba da u sledećem programu unese ulazne brojeve niza jedan po jedan, a oni će se redom dodavati parcijalnoj sumi brojeva učitanih do tada. Istovremeno, pošto je za izračunavanje proseka potreban ukupan broj učitanih brojeva, njihovo brojanje se u programu izvodi uvećavanjem brojača za jedan kako se sledeći član niza učita u petlji. Na kraju, kada imamo konačnu sumu niza brojeva i njihov ukupan broj, prosek se lako dobija kao količnik ove dve vrednosti.

LISTING 4.6: Izračunavanje proseka niza brojeva.

```
import java.util.*;

public class Prosek {

   public static void main(String[] args) {

      double broj;    // aktuelni ulazni broj
      double suma;    // suma ulaznih brojeva
      int n;          // brojač ulaznih brojeva
      double prosek;  // prosek ulaznih brojeva

      // Inicijalizacija sume i brojača ulaznih brojeva
      suma = 0;
      n = 0;
```

```
      // Učitavanje, sabiranje i brojanje ulaznih brojeva
      Scanner tastatura = new Scanner(System.in);
      System.out.print("Unesite 1. broj: ");

      while (tastatura.hasNextDouble()) { // sledeći broj je unet?
         broj = tastatura.nextDouble();   // učitati ga
         suma = suma + broj;              // dodati ga sumi
         n++;                             // uvećati brojač za 1
         System.out.print("Unesite " + (n+1) + ". broj: ");
      }

      System.out.println();
      if (n == 0)
         System.out.println("GREŠKA: uneto je nula brojeva.");
      else {
         System.out.println("Uneto je " + n + " brojeva.");
         System.out.println("Prosek tih brojeva je: " + suma/n);
      }
   }
}
```

U programu smo za prestanak učitavanja niza brojeva iskoristili metod `hasNextDouble()` klase `Scanner` koji vraća logičku vrednost `true` ako je neka vrednost tipa `double` raspoloživa za učitavanje preko tastature. Korisnik treba da na tastatiri signalizira kraj ulaznog niza brojeva za koje želi prosek istovremenim pritiskom kombinacije tastera <CTRL> i Z. U tom trenutku će metod `hasNextDouble()` kao rezultat vratiti vrednost `false`, pa će time biti ispunjen uslov prekida `while` petlje.

Naredba `do-while`

U prethodnom odeljku smo videli da se kod `while` petlje uslov prekida petlje proverava na početku svake iteracije. Međutim, ponekad je prirodnije proveravati taj uslov na kraju svakog izvršavanja tela petlje. U takvim slučajevima se može koristiti `do-while` naredba koja je vrlo slična `while` naredbi, osim što su reč `while` i uslov prekida pomereni na kraj, a na početku se nalazi reč `do`. Opšti oblik `do-while` naredbe je:

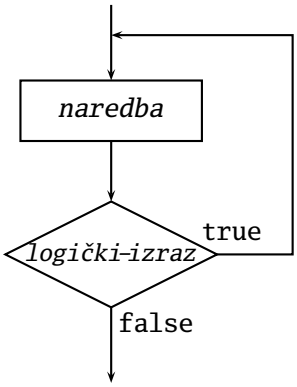
```
do
   naredba
while ( logički-izraz );
```

ili, ako je *naredba* unutar do-while naredbe jedna blok naredba:

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

Obratite pažnju na tačku-zarez na samom kraju. Znak za tačku-zarez je deo do-while naredbe i ne može se izostaviti. (Generalno, na kraju svake naredbe u Javi se mora nalaziti tačka-zarez (;) ili zatvorena vitičasta zagrada (}).)

Kod izvršavanja do-while naredbe se najpre izvršava telo petlje, odnosno *naredba* ili *niz-naredbi* u bloku. Zatim se izračunava vrednost *logičkogizraza* u zagradi. U jednom slučaju, ako je ta vrednost false, prekida se izvršavanje do-while naredbe i nastavlja se normalno izvršavanje programa od naredbe koja sledi iza reči while. U drugom slučaju, ako izračunavanje *logičkog-izraza* daje true, izvršavanje se vraća na početak tela petlje i ponavlja isti postupak od početka. Ovo je predstavljeno blok-dijagramom na slici 4.4.



SLIKA 4.4: Izvršavanje do-while naredbe.

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

naredba

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

Da bismo se uverili da je ponekad prirodnije koristiti do-while petlju umesto while petlje, razmotrimo primer programa za igru pogađanja broja. U tom primeru smo morali da veštački uvodimo promenljivu pogodak na osnovu koje smo prekidali igru ili nastavljali sa pogađanjem zamišljenog broja. Ako koristimo do-while petlju, ova veštačka promenljiva nam više nije potrebna, jer svakako moramo proći bar kroz jednu iteraciju i na njenom kraju znamo da li je zamišljeni broj pogođen ili ne. Na primer, relevantni deo nove verzije ovog programa je:

```
do {
   System.out.print("Pogodite broj> ");
   pokušanBroj = tastatura.nextInt();
   if (pokušanBroj < zamišljenBroj)
      System.out.println("Zamislio sam veći broj :-(");
   else if (pokušanBroj > zamišljenBroj)
      System.out.println("Zamislio sam manji broj :-(");
   else
      System.out.println("Bravo, pogodili ste broj :-)");
} while (pokušanBroj != zamišljenBroj);
```

Naredba for

Treća vrsta petlji u Javi se obično koristi kada je potrebno izvršiti telo petlje za sve vrednosti određene promenljive u nekom intervalu. Na primer, u programskom fragmentu kojim se na ekranu prikazuju brojevi 0, 1, 2, ..., 9 smo koristili while naredbu. Tu se zapravo radi o telu petlje koje se sastoji od prikazivanja promenljive broj, a to treba ponoviti za sve vrednosti te promenljive u intervalu od 0 do 9. Zato je u ovom primeru mnogo prirodnije koristiti for naredbu, jer je odgovarajući ekvivalentni fragment mnogo kraći i izražajniji:

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

Opšti oblik for naredbe je:

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

ili, ako je *naredba* unutar for naredbe jedna blok naredba:

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

Kod for petlje je kontrolnim delom u prvom redu 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 koja su međusobno razdvojena tačkom-zapetom:[5]

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

tako da je opšti oblik (proste) for naredbe zapravo:

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

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

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

Na samom početku se dakle izvršava deo *inicijalizacija*, i to samo jedanput. Dalje se, kao što je to uobičajeno, uslov nastavka petlje predstavljen *logičkim-izrazom* izračunava pre svakog izvršavanja tela petlje, a izvršavanje petlja se prekida kada *logički-izraz* daje false. Deo *završnica* se izvršava na kraju svakog izvršavanja tela petlje, neposredno posle izvršavanja *naredbe* i 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 da svaki od tri dela u kontrolnom delu može biti prazan (ali se tačka-zarezi ne mogu izostaviti). Ako je *logički-izraz* prazan, to se smatra kao da tu stoji true, pa se dobijena beskonačna petlja mora prekinuti na neki drugi način (recimo, break naredbom o kojoj se govori u narednom odeljku).

[5] Kontrolni deo od verzije Java 5.0 može imati još jedan oblik koji je prilagođen za rad sa strukturama podataka. O tome se govori u odeljku 7.2.

Obično se u delu *inicijalizacija* nekoj promenljivi dodeljuje početna vrednost, a u delu *završnica* se vrednost te promenljive uvećava ili smanjuje za određen korak. Takođe, vrednost te promenljive se proverava u uslovu nastavka petlje i petlja se završava kada taj uslov postane false. Promenljiva koja se na ovaj način koristi u for naredbi se naziva ***kontrolna promenljiva petlje*** ili kratko ***brojač***.

U našem uvodnom primeru for naredbe, promenljivu broj smo koristili upravo kao brojač. U tom primeru se odbrojavanje odvija u rastućem redosledu, mada je lako realizovati i odbrojavanje u opadajućem redosledu. Sledeći programski fragment prikazuje brojeve od 9 do 0:

```
for (int broj = 9; broj >= 0; broj--)
   System.out.print(broj + " ");
```

U stvari, kako *inicijalizacija* tako i *završnica* u kontrolnom delu for naredbe mogu imati nekoliko izraza razdvojenih zarezima. Na primer, u sledećem programskom fragmentu se 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;          //      i prelazak u novi red
}
```

Primer: određivanje da li je broj prost

Ceo broj veći od jedan je ***prost*** ako je deljiv samo sa samim sobom i jedinicom. Početak niza prostih brojeva je 2, 3, 5, 7, 11, ... U ovom primeru ćemo napisati Java program koji određuje da li je ulazni ceo broj prost ili ne.

Ako je n dati broj, očigledan postupak kojim se određuje da li je n prost broj je da se svi brojevi od 2 do $n-1$ redom provere da li dele n. Ako se pronađe bar jedan takav delioc, n nije prost; u suprotnom slučaju, n jeste prost. Ovu pretragu možemo skratiti ukoliko primetimo da ako broj n ima delioca d u intervalu $1 < d < n$, onda ima i delioca manjeg ili jednakog $\sqrt{n}$. Naime, to je broj d ili n/d. Dovoljno je dakle proveravati potencijalne delioce broja n samo iz intervala od 2 do $\sqrt{n}$ i na taj način ubrzati postupak.

LISTING 4.7: Određivanje da li je broj prost.

```
import java.util.*;

public class ProstBroj {

   public static void main(String[] args) {

      int broj;  // ulazni broj

      // Učitavanje broja preko tastature
      Scanner tastatura = new Scanner(System.in);
      System.out.print("Unesite ceo broj veći od 1: ");
      broj = tastatura.nextInt();

      int maxDelioc = (int)Math.sqrt(broj); // maksimalni delioc

      // Potraga za deliocem između 2 i maxDelioc
      for (int k = 2; k < maxDelioc; k++) {
         if (broj % k == 0) {  // da li je broj deljiv sa k?
            System.out.println("Taj broj nije prost.");
            System.out.println("Njegov delioc je " + k);
            System.exit(0);
         }
      }
      System.out.println("Taj broj je prost.");
   }
}
```

Primetimo u programu da kod izračunavanja maksimalnog delioca moramo da koristimo eksplicitnu konverziju tipa, jer metod `Math.sqrt()` kao rezultat vraća vrednost tipa `double`. Drugo, telo `for` petlje smo istakli omeđujući ga vitičastim zagradama, mada to nije neophodno jer se to telo sastoji samo od jedne, `if` naredbe.

Primer: prikazivanje tablice množenja

Upravljačke naredbe u Javi su naredbe koje sadrže druge naredbe. Specifično, upravljačke naredbe mogu sadržati druge upravljačke naredbe i tada govorimo o ugnježđavanju ovih naredbi jedne u drugu. Videli smo

primere if naredbi unutar drugih if naredbi i petlji, a sada ćemo ilustrovati mogućnost ugnježđavanja jedne petlje u drugu. U stvari, generalno je dozvoljena svaka kombinacija ugnježđavanja jedne upravljačke naredbe u drugu, kao i neograničen broj nivoa njihovog ugnježđavanja.

Da bismo dobro razumeli kako rade ugnježđene petlje, u ovom jednostavnom primeru ćemo napisati Java program koji prikazuje tablicu množenja u obliku:

```
 1   2   3   4   5   6   7   8   9
 2   4   6   8  10  12  14  16  18
 3   6   9  12  15  18  21  24  27
 4   8  12  16  20  24  28  32  36
 5  10  15  20  25  30  35  40  45
 6  12  18  24  30  36  42  48  54
 7  14  21  28  35  42  49  56  63
 8  16  24  32  40  48  56  64  72
 9  18  27  36  45  54  63  72  81
```

Postupak prikazivanja tablice množenja možemo rečima opisati na sledeći način: idući red po red, unutar svakog reda prikazati proizvod broja aktuelnog reda i broja kolone. Preciznije, za svaki broj reda od 1 do 9, prikazati proizvod broja aktuelnog reda i broja kolone za svaki broj kolone od 1 do 9. Ovo se programski može realizovati pomoću dve ugnježđene for petlje, pri čemu spoljašnja petlja ukazuje na broj aktuelnog reda, dok unutrašnja petlja prikazuje odgovarajuće proizvode po kolonama u tom redu.

LISTING 4.8: Prikazivanje tablice množenja.

```
import java.util.*;

public class TablicaMnozenja {

   public static void main(String[] args) {

      for (int brojReda = 1; brojReda < 10; brojReda++) {
         for (int brojKolone = 1; brojKolone < 10; brojKolone++)
            System.out.printf("%4d", brojReda * brojKolone);
         System.out.println();  // preći u novi red na ekranu
      }
   }
}
```

U ovom programu je važno uočiti da se u svakoj pojedinačnoj iteraciji spoljašnje for petlje izvršavaju sve iteracije unutrašnje for petlje. To znači da kada brojač spoljašnje petlje brojReda ima početnu vrednost 1, brojač unutrašnje petlje brojKolone će promeniti sve vrednosti od 1 do 10; kada brojač spoljašnje petlje brojReda ima drugu vrednost 2, brojač unutrašnje petlje brojKolone će ponovo promeniti sve vrednosti od 1 do 10; i tako dalje, kada brojač spoljašnje petlje brojReda ima vrednost 9, brojač unutrašnje petlje brojKolone će ponovo promeniti sve vrednosti od 1 do 10. Prema tome, ukupan broj iteracija unutrašnje petlje je 81, odnosno po devet iteracija unutrašnje petlje za svaku od devet iteracija spoljašnje petlje.

Obratite pažnju i na estetski lepo poravnate kolone prikazane tablice množenja. To smo postigli upotrebom specifikatora formata %4d kako bi se svaki broj u jednom redu prikazao u polju dužine četiri mesta.

Naredbe **break** i **continue**

Sintaksa while i do-while petlji omogućava da se uslov prekida tih petlji proverava na početku odnosno na kraju svake iteracije. Slično, iteracije for petlje se završavaju kada se dostigne 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 break naredba koja ima jednostavan oblik:

```
break;
```

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

U narednom fragmentu se proverava podatak koji korisnik unosi i ne dozvoljava se nastavak programa dok podatak nije ispravno unet:

```
while (true) {    // beskonačna petlja?
   System.out.print("Unesite pozitivan broj: ");
   int broj = tastatura.nextInt();
   if (broj > 0)  // uneti broj je ok, prekinuti petlju
      break;
   System.out.println("Greška: morate uneti broj > 0");
}
// Ovde se program nastavlja posle prekida petlje
```

Kako je u ovom primeru logički izraz while naredbe uvek true, to izgleda kao da se radi o beskonačnoj petlji[6] kojom se od korisnika stalno zahteva da unese pozitivan broj. Naime, ukoliko uneti broj nije pozitivan, prikazuje se poruka o grešci i ponavlja se telo petlje od početka učitavanjem novog broja. Međutim, u telu petlje se izvršava break naredba čim korisnik ispavno unese pozitivan broj, što dovodi do prekida izvršavanja petlje i normalnog nastavka programa.

Drugi primer korišćenja break naredbe predstavlja još jedan način za rešenje petlje u programu za igru pogađanja broja na strani 101:

```
while (true) {
   System.out.print("Pogodite broj> ");
   pokušanBroj = tastatura.nextInt();
   if (pokušanBroj < zamišljenBroj)
      System.out.println("Zamislio sam veći broj :-(");
   else if (pokušanBroj > zamišljenBroj)
      System.out.println("Zamislio sam manji broj :-(");
   else {
      System.out.println("Bravo, pogodili ste broj :-)");
      break;
   }
}
```

Pošto se petlje mogu ugnježđavati, postavlja se pitanje koju petlju break naredba prekida ako se nalazi unutar ugnježđene petlje? Pravilo je da break naredba prekida izvršavanje samo one petlje u kojoj se nalazi, ne i spoljašnju petlju koja sadrži ugnježđenu petlju.

U stvari, ako se koristi break naredba sa oznakom, može se prekinuti svaka ***označena*** petlja. Oznaka petlje se sastoji od identifikatora sa dvotačkom i navodi se ispred neke petlje. Na primer, početak označene while petlje sa oznakom gl_petlja može biti:

```
gl_petlja: while ...
```

Unutar tela ove petlje se zatim može koristiti naredba

```
break gl_petlja;
```

radi prekida ove označene petlje.

[6] Još jedan način za obrazovanje beskonačne petlje je pomoću for naredbe koja ima „prazan" kontrolni deo: for (;;) {...}.

U sledećem primeru se određuje da li dva stringa s1 i s2 sadrže neki isti znak. To se radi tako što se u spoljašnjoj for petlji sa oznakom sp, za svaki znak stringa s1, u ugnježđenoj for petlji proverava da li je dati znak jednak nekom znaku u stringu s2. Ako se zajednički znak pronađe, logička promenljiva nađenIstiZnak dobija vrednost true i break naredbom se prekida dalje traženje.

```
boolean nađenIstiZnak = false;   // s1 i s2 nemaju isti znak
sp: for (int i = 0; i < s1.length(); i++)
      for (int j = 0; j < s2.length(); j++)
         if (s1.charAt(i) == s2.charAt(j)) { // isti znakovi?
            nađenIstiZnak = true;
            break sp;
         }
// Ovde se logika programa može nastaviti na osnovu
// vrednosti logičke promenljive nađenIstiZnak.
```

U petljama se na sličan način može koristiti i continue naredba čiji je oblik takođe jednostavan:

```
continue;
```

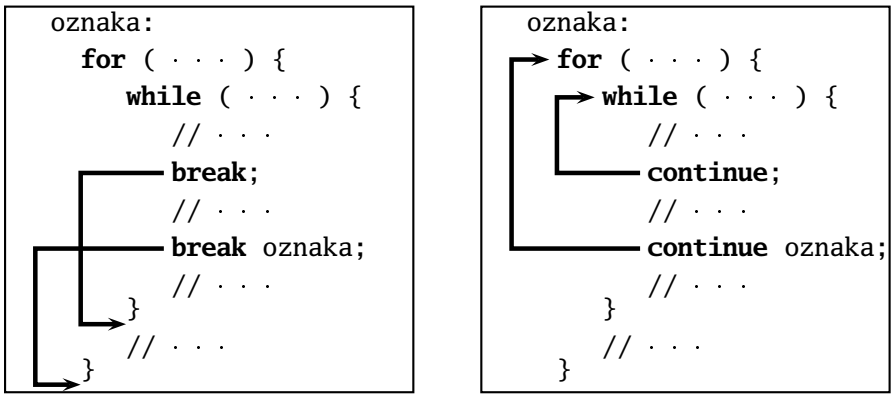
Izvršavanjem continue naredbe se samo preskače ostatak aktuelne iteracije petlje. To znači da se, umesto prekidanja izvršavanja cele petlje kao kod break naredbe, prekida izvršavanje samo aktuelne iteracije i nastavlja sa izvršavanjem naredne iteracije petlje (uključujući i izračunavanje uslova prekida radi provere da li treba uopšte nastaviti izvršavanje petlje).

U sledećem prostom primeru continue naredbe se na ekranu prikazuju samo parni brojevi iz intervala od 0 do 9 u jednom redu:

```
int broj = 0;          // početni broj intervala od 0 do 9
while ( broj < 10 ) {  // ponoviti sve dok je broj manji od 10
   if (broj % 2 != 0)  // broj je neparan?
      continue;        // preskočiti ostatak tela petlje
   System.out.print(broj + " "); // prikazati broj na ekranu
   broj = broj + 1;              // preći na sledeći broj
}
System.out.println();  // pomeriti kursor u novi red
```

Slično kao break naredba, ako se continue naredba nalazi u ugnježđenoj petlji, ona prekida aktuelnu iteraciju samo te petlje. Isto tako, može se koristiti continue naredba sa oznakom radi prekida izvršavanja aktuelne

iteracije bilo koje označene petlje. Efekat naredbi break i continue u ugnježđenim petljama ilustrovan je na slici 4.5.



SLIKA 4.5: Efekat naredbi break i continue u ugnježđenim petljama.

Pored toga što se naredbe break i continue mogu koristiti u svakoj vrsti petlje (while, do-while i for), videli smo da se break naredba može koristiti i za prekid izvršavanja switch naredbe. Pažljivi čitaoci bi mogli reći da smo break naredbu koristili i unutar if naredbe. To je dozvoljeno, ali samo ako se if naredba sama nalazi unutar neke petlje ili switch naredbe. U tom slučaju, break naredba prekida izvršavanje obuhvatajuće petlje ili switch naredbe, a ne if naredbe. Ako se if naredba ne nalazi unutar neke petlje ili switch naredbe, tada if naredba ne sme sadržati break naredbu.

Primer: određivanje da li je rečenica palindrom

Rečenica se naziva ***palindrom*** ako se ona isto čita sleva na desno i zdesna na levo, ne računajući razmake i znakove punktuacije, kao ni ne razlikujući mala i velika slova. Primeri palindroma su „Ana voli Milovana", „aroma sa mora", „radar" i tako dalje.

Sledeći program omogućava korisniku da unese više rečenica jednu po jednu, koje se redom proveravaju da li predstavljaju palindrom ili ne. Kraj rada se signalizira unošenjem „prazne" rečenice pritiskom na taster enter. („Prazna" rečenica jeste palindrom, jer nema ništa u njoj što bi protivrečilo tom njenom svojstvu.)

Osnovni postupak za određivanje da li je rečenica palindrom jeste da se redom ispitaju simetrični znakovi sa levog i desnog kraja rečenice. Ako su svi ti parovi znakova jednaki, rečenica jeste palindrom; ako se otkrije jedan par nejednakih znakova, rečenica nije palindrom.

LISTING 4.9: Određivanje da li je rečenica palindrom.

```
import java.util.*;

public class Palindrom {

   public static void main(String[] args) {

      Scanner tastatura = new Scanner(System.in);
      String red;  // ulazni red znakova rečenice

      novi_red: do {
         System.out.print("Unesite rečenicu (<enter> za kraj): ");
         red = tastatura.nextLine();

         // Konvertovanje svih znakova ulaznog reda u mala slova
         red = red.toLowerCase();

         // Zanemarivanje nevažećih znakova ulaznog reda
         String rečenica = "";
         for (int i = 0; i < red.length(); i++) {
             char znak = red.charAt(i);
             if (Character.isLetterOrDigit(znak))
                rečenica += znak;
         }

         // Proveravanje znakova rečenice sa dva kraja
         int l = 0;                    // indeks levog znaka
         int d = recenica.length() - 1; // indeks desnog znaka

         while (l < d) { // sve dok se indeksi ne preklapaju
            if (rečenica.charAt(l) != rečenica.charAt(d)) {
               System.out.println("To nije palindrom.");
               continue novi_red;  // nastavi sa novim ulazom
            }
            l++;  // pomeri levi indeks udesno
            d--;  // pomeri desni indeks ulevo
         }
```

```
         System.out.println("To jeste palindrom.");
      } while (red.length() != 0);
   }
}
```

U programu smo koristili metode `length()`, `toLowerCase()` i `charAt()` klase `String`, kao i metod `isLetterOrDigit()` klase `Character`. Iako su funkcije ovih metoda očigledne iz njihovih imena, čitaoci se o tome mogu podsetiti u odeljku 3.4.

Obratite pažnju i na korišćenje označene glavne petlje u kojoj se obrađuju ulazne rečenice. To je potrebno radi prekida aktuelne iteracije `continue` naredbom sa oznakom, čim se otkrije da ulazna rečenica nije palindrom.

Glava

5

Metodi

Metodi u Javi su potprogrami koje obavljaju neki specifični zadatak. Koncept potprograma je sveprisutan u programskim jezicima, ali se pojavljuje pod različitim imenima: metod, procedura, funkcija, rutina i slično. Njegova glavna svrha je to da se složeni program podeli u manje delove koji se mogu lakše i nezavisno razvijati.

Osim što je deo programa i što obično obavlja jednostavniji zadatak, potprogram ima sličnu logičku strukturu kao glavni program. Potprogram se sastoji od niza naredbi i promenljivih koje predstavljaju neku funkcionalnu celinu sa svojim posebnim imenom. Ime potprograma se može koristiti bilo gde u programu kao zamena za ceo niz naredbi od kojih se potprogram sastoji. Pored toga, ova mogućnost se može koristiti više puta, na različitim mestima u programu.

Potprogrami se mogu čak koristiti unutar drugih potprograma. To znači da se mogu napisati jednostavni potprogrami, koji se mogu iskoristiti u složenijim potprogramima, a i ovi se dalje mogu koristiti u još složenijim potprogramima. Na taj način, vrlo složeni programi se mogu postupno razvijati korak-po-korak, pri čemu je svaki korak relativno jednostavan. Koncept potprograma je dakle glavni alat kojim se programeri služe za hvatanje u koštac sa složenošću problema.

Potprogrami imaju, baš kao i programi, ulazne i izlazne podatke preko kojih komuniciraju sa okolnim (pot)programom. (Programi preko ulaznih i izlaznih podataka komuniciraju sa ljudima.) Ulazni podaci su parametri koje potprogram dobija od okolnog (pot)programa radi izvršenja svog

zadatka, a izlazni podaci predstavljaju rezultat rada potprograma.

Ulazni i izlazni podaci su deo veze jednog potprograma sa drugim potprogramima koja se često naziva ***interfejs*** (ili *ugovor*) potprograma. Interfejs pored ovoga uključuje ime i opis funkcije potprograma, odnosno sve što treba znati radi ispravne upotrebe potprograma u drugim potprogramima bez poznavanja same implementacije potprograma. U tom smislu se za potprogram kaže da predstavlja „crnu kutiju" čija je unutrašnjost (implementacija) sakrivena od spoljašnjosti, a interfejs je granica koja povezuje ova dva dela.

Korišćenje potprograma se u programiranju tehnički naziva ***pozivanje*** potprograma za izvršavanje. Svakim pozivom potprograma se dakle izvršava niz naredbi od kojih se potprogram sastoji. Implementacija potprograma, odnosno niz naredbi i ostali elementi koji čine potprogram, naziva se ***definicija*** (ili ***deklaracija***) potprograma. Obratite pažnju na to da za pozivanje potprograma nije potrebno znati njegovu definiciju, već samo njegov interfejs. Interfejs potprograma je obično jednostavan, a njegova definicija može biti vrlo komplikovana, tako da često ni ne želimo znati definiciju nekog potprograma koji koristimo.

Sada kada znamo opšte principe potprograma, možemo govoriti o njihovoj realizaciji u Javi. Termin koji se u Javi koristi za potprogram je ***metod***, u smislu postupak ili način za rešavanje nekog zadatka. Najpre ćemo se upoznati sa detaljima pisanja sopstvenih metoda, a zatim sa načinima pozivanja (sopstvenih ili tuđih) metoda.

5.1 Definisanje metoda

Važna odlika programskog jezika Java je da se definicija svakog metoda u Javi mora nalaziti unutar neke klase. Ovim se želela postići jasna hijerarhija programskog koda, jer metodi grupišu srodne naredbe i promenljive, klase grupišu srodne metode i polja i, na najvišem nivou, paketi grupišu srodne klase. Metod koji je član neke klase može biti statički (klasni) ili objektni (nestatički), prema tome da li pripada toj klasi kao celini ili pojedinačnim objektima te klase. Od ovoga zavisi kako se metod može pozivati, a njegova definicija sadrži ili ne sadrži samo jednu službenu reč kojom se to određuje.

Opšti oblik definicije metoda u Javi je:

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

Na primer, metod `main()` koji smo koristili su svim dosadašnjim programima je imao format:

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

U prvom redu ove definicije, službene reči `public` i `static` su primeri modifikatora iz opšteg oblika definicije metoda, `void` je tip rezultata, `main` je ime metoda i, na kraju, `String[] args` je lista od jednog parametra.

U opštem obliku definicije metoda, *niz-naredbi* između vitičastih zagrada se naziva ***telo metoda***. To su naredbe koje se izvršavaju kada se metod pozove i mogu biti bilo koje naredbe o kojima smo govorili u prethodnim poglavljima knjige. Taj deo definicije metoda je dakle unutrašnjost „crne kutije", ako tako zamišljamo metod. Prvi red definicije metoda (bez znaka {) naziva se ***zaglavlje metoda*** i predstavlja interfejs metoda. Sada ćemo objasniti pojedine delove u zaglavlju metoda, mada njihovo potpuno razumevanje ne treba očekivati pre kraja ovog poglavlja.

Deo *modifikatori* na početku zaglavlja nije obavezan, ali se može sastojati od jedne ili više službenih reči kojima se određuju izvesne karakteristike metoda. Na primer, službena reč `static` ukazuje da je metod statički, a ne objektni; službena reč `public` ukazuje da se metod može slobodno koristiti bilo gde. Ukupno je na raspolaganju oko desetak modifikatora i upoznaćemo ih u kontekstu daljeg teksta.

Ako metod izračunava jednu vrednost koja se vraća kao rezultat poziva metoda, onda deo *tip-rezultata* određuje tip podataka kojem pripada ta vrednost. Ako metod ne vraća nijednu vrednost, onda se deo *tip-rezultata* sastoji od službene reči `void`. (Kao da se želi ukazati da je tip podataka vraćene vrednosti nepostojeći.)

Sledećei deo *ime-metoda* predstavlja običan identifikator. Formalna pravila za davanje imena metodu su dakle uobičajena pravila za identifikatore u Javi o kojima smo govorili u odeljku 3.1. Podsetimo se samo da je dodatna konvencija za imena metoda ista kao za imena promenljivih: prva reč imena metoda počinju malim slovom, a ako se to ime sastoji od nekoliko reči, onda se svaka reč od druge piše sa početnim velikim

slovom. A drugo nepisano pravilo je da, naravno, imena metoda trebaju biti smislena, odnosno da imena metoda dobro označavaju šta je funkcija pojedinih metoda.

Na kraju zaglavlja metoda, deo *lista-parametara* u zagradi određuje sve parametre metoda. Parametri su ulazni podaci metoda koji se obrađuju naredbama u telu metoda. Na primer, pretpostavimo da se u klasi kojom se opisuje fizički objekat televizora nalazi metod `promeniKanal()` kojim se može promeniti kanal prikazivanja televizora. Da bi izvršio ovu svoju funkciju, ovaj metod mora imati broj kanala u koji želimo da televizor promeni prikazivanje. Ali taj broj nije unapred poznat, jer možda zavisi od onoga što korisnik navede prilikom izvršavanja. Zato metod `promeniKanal()` moramo pisati sa parametrom koji će tek u vreme izvršavanja tog metoda dobiti vrednost kanala u koji televizor treba da promeni prikazivanje. Pošto je ciljni kanal ceo broj, tip ovog parametra je `int`. Ako taj parametar nazovemo `ciljniKanal`, nepotpuna definicija metoda `promeniKanal()` može imati ovaj oblik:

```
public void promeniKanal (int ciljniKanal) {
   // Telo metoda
}
```

Zaglavlje ovog metoda čije je ime `promeniKanal` ukazuje da taj metod ima parametar pod imenom `ciljniKanal` tipa `int`. Jedan parametar predstavlja zapravo promenljivu, pa moramo navesti tip i ime svakog parametra. U telu metoda se parametar koristi kao obična promenljiva čija se konkretna vrednost ne poznaje. Parametar dobija specifičnu vrednost prilikom poziva metoda: na primer, `promeniKanal(8)`.

U opštem slučaju, lista parametara u zagradi zaglavlja nekog metoda može biti prazna (ali se zagrade moraju pisati), ili se može sastojati od jednog ili više parametara. Svaki parametar u listi je određen parom specifikacija u obliku:

tip-parametra ime-parametra

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

U nastavku navodimo još neke primere definicije metoda u Javi, bez naredbi u telu metoda koje određuju šta ti metodi rade:

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

  Ovde 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
  }
  ```

  Ovde 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
  }
  ```

  Ovde 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 ukazuje modifikatorom static). Modifikator public koji je korišćen u prvom primeru određuje da je metod odigrajPotez „javan", odnosno da se može pozivati bilo gde u programu, čak i izvan klase u kojoj je definisan. Srodni modifikator je private kojim se određuje 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 gde 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 primeru),
  ```

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, ali njihov međusobni redosled nije bitan. Konvencija je ipak da se najpre piše specifikator pristupa (ako ga ima), zatim reč `static` (ako je ima) i, na kraju, eventualno ostali modifikatori.

5.2 Pozivanje metoda

Definisanje nekog metoda je samo prvi korak u njegovom korišćenju. Naime, definicijom metoda se uspostavlja njegovo postojanje i ukazuje kako on radi. Ali metod se uopšte ne izvršava sve dok se ne pozove. (Ovo je čak tačno i za metod `main()` u nekoj klasi, iako se on ne poziva eksplicitno u programu — njega implicitno poziva JVM kada izvršava program.) Tek pozivom metoda se niz naredbi u telu metoda izvršava na uobičajeni način.

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

ime-metoda (*lista-argumenata*)

Prema tome, tamo gde se želi izvršavanje zadatka koji neki metod obavlja, navodi se samo njegovo ime i argumenti koji odgovaraju parametrima metoda u zagradi. Posmatrajmo primer definicije sledećeg metoda:

```
public void uradiNešto(int n, double x, boolean test) {
   // Naredbe koje rade nešto ...
}
```

Iz definicije metoda `uradiNešto` vidimo da on ne vraća nijednu vrednost i da ima tri parametra: `n` celobrojnog tipa, `x` realnog tipa i `test` logičkog tipa. Primeri ispravnih poziva ovog metoda u klasi u kojoj je definisan su naredbe poziva:

- `uradiNešto(17, 0.69, false);`
- `uradiNešto(i, a, brojProst);`
- `uradiNešto(23, Math.sqrt(y+1), z >= 10);`

Obratite posebnu pažnju u ovim primerima na to da parametar u definiciji metoda mora biti *ime*, odnosno prost identifikator, jer je parametar konceptualno jedna promenljiva. Ovim se objašnjava i zašto parametar mora imati deklarisan svoj tip podataka kome pripada. S druge strane, argument u pozivu metoda je konceptualno neka *vrednost* i zato argument može biti bilo koji izraz čije izračunavanje daje vrednost odgovarajućeg tipa.

Tako, u prvom primeru poziva metoda `uradiNešto`, argumenti su literalne vrednosti `17`, `0.69` i `false` čiji se tipovi slažu sa tipovima odgovarajućih parametara `n`, `x` i `test` u definiciji metoda `uradiNešto`. U drugom primeru poziva metoda `uradiNešto`, argumenti su aktuelne vrednosti promenljivih `i`, `a` i `brojProst`, pa se deklarisani tipovi ovih promenljvih moraju slagati sa tipovima odgovarajućih parametara `n`, `x` i `test`. Najzad, u trećem primeru poziva metoda `uradiNešto`, drugi i treći argument su vrednosti izraza `Math.sqrt(y+1)` i `z >= 10`, čiji se tipovi slažu sa tipovima odgovarajućih parametara `x` i `test`.

U svetlu ovih razmatranja, postupak izvršavanja naredbe poziva nekog metoda moramo bolje precizirati: pre izvršavanja naredbi u telu definicije metoda, izračunavaju se izrazi koji su navedeni kao argumenti u pozivu i njihove vrednosti se dodeljuju odgovarajućim parametrima. Ovaj način prenošenja vrednosti argumenata odgovarajućim parametrima u Javi se tehnički naziva ***prenošenje po vrednosti***.

Pozivom metoda se dakle izvršava telo metoda, ali sa inicijalnim vrednostima parametara dobijenim od odgovarajućih argumenata. To znači da, recimo, treća naredba poziva metoda `uradiNešto()`:

```
uradiNešto(23, Math.sqrt(y+1), z >= 10);
```

ima isti efekat kao ova blok naredba:

```
{
   // Alokacija i inicijalizacija parametara metoda uradiNešto
   int n = 23;
   double x = Math.sqrt(y+1);
   boolean test = (z >= 10);

   // Izvršavanje tela metoda uradiNešto
   // Naredbe koje rade nešto ...
}
```

Rekli smo da se definicija svakog metoda u Javi mora nalaziti unutar neke klase. S druge strane, naredba poziva metoda u najprostijem obliku se može navesti na bilo kom propisanom mestu u klasi u kojoj se nalazi definicija metoda. Na primer, pretpostavimo da je metod uradiNešto() definisan u klasi MojaKlasa, koja pored toga sadrži i definiciju metoda main(). Struktura klase MojaKlasa zato može imati ovaj oblik:

```
public class MojaKlasa {
   ⋮
   // Definicija metoda main
   public static void main(String[] args) {
      ⋮
      uradiNešto(17, 0.69, false); // poziv metoda
      ⋮
      uradiNešto(23, Math.sqrt(y+1), z >= 10); // poziv metoda
      ⋮
   }

   // Definicija metoda uradiNešto
   public void uradiNešto(int n, double x, boolean test) {
      // Naredbe koje rade nešto ...
   }
   ⋮
}
```

Obratite pažnju na to da se definicije metoda main() i uradiNešto() u klasi MojaKlasa nalaze jedna iza druge. To je posledica opšteg pravila u Javi po kojem nije ispravno pisati jedan metod ugnježđen u drugi. Zato metod main() može pozivati metod uradiNešto(), ali ga ne sme fizički obuhvatati. Pored toga, redosled navođenja metoda (opštije, svih članova) neke klase u Javi nije bitan.

Druga dva načina pozivanja metoda u Javi zavise od toga da li je metod statički ili objektni (tj. da li je definisan sa službenom rečju static ili bez nje). 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 sa imenom *ime-metoda* definisan.[1] Statički (klasni) metodi pripadaju klasi, pa upotreba imena klase u tačka-notaciji ukazuje u kojoj klasi treba naći definiciju metoda koji se poziva. Ovaj način smo već primenjivali za pozivanje statičkih metoda iz klase `Math`: na primer, `Math.sqrt(y+1)` ili `Math.pow(1+k, m)`. Pošto su `sqrt()` i `pow()` statički metodi definisani u klasi `Math`, van te klase u drugim, svojim klasama moramo koristiti tačka-notaciju za njihovo pozivanje.

Na kraju, objektni metodi pripadaju pojedinim objektima, a ne celoj klasi, pa se pozivaju uz ime odgovarajućeg objekta. Dakle, ako je metod objektni i poziva se izvan klase u kojoj je definisan, onda je opšti oblik njegovog poziva:

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

Ovde je *ime-objekta* promenljiva koja sadrži referencu na objekat klase u kojoj je metod sa imenom *ime-metoda* definisan. I ovaj način smo već primenjivali: na primer, kod učitavanja podataka preko tastature:

```
tastatura.nextInt()
```

U ovom pozivu je `tastatura` promenljiva koja ukazuje na (prethodno konstruisan) objekat klase `Scanner`. A u toj klasi je definisan, kao što znamo, objektni metod `nextInt()` kojim se učitava sledeća celobrojna vrednost preko tastature. Primetimo ovde generalno i da ako metod nema parametre (kao što je to slučaj sa `nextInt()`), onda se u njegovom pozivu moraju pisati zagrade koje „sadrže" praznu listu argumenata.

Drugi primer je u radu sa stringovima:

```
rečenica.CharAt(i)
```

Ovde se poziva objektni metod `CharAt()` klase `String`, sa vrednošću promenljive `i` kao argumentom, za string (objekat klase `String`) na koga ukazuje promenljiva `rečenica`. Rezultat ovog poziva je i-ti znak u nizu znakova koji predstavljaju vrednost tog stringa.

[1] U stvari, statički metod se može pozvati i preko nekog objekta klase u kojoj je metod definisan, odnosno na isti način kao što se objektni metod mora pozivati. Međutim, takav način pozivanja statičkog metoda se ne preporučuje.

5.3 Vraćanje rezultata

Kada smo govorili o definisanju metoda, naznačili smo da metod može vraćati jednu vrednost koja se dobija kao (dodatni) rezultat poziva metoda. Ako je to slučaj, onda ta vraćena vrednost 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.)

Poziv metoda koji vraća jednu vrednost se navodi u programu tamo gde je ta vrednost potrebna. To je obično na desnoj strani znaka jednakosti kod naredbe dodele, kao argument poziva drugog metoda ili unutar nekog složenijeg izraza. Pored toga, pozivi metoda koji vraćaju logičku vrednost mogu 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.)

Definicija metoda koji vraća jednu vrednost mora imati, pored tipa te vrednosti u zaglavlju metoda, bar jednu naredbu u telu metoda čijim izvršavanjem se ta vrednost zapravo vraća. Ta naredba se naziva ***naredba povratka*** i ima opšti oblik:

return *izraz*;

gde se tip vrednosti *izraza* mora slagati sa navedenim tipom rezultata metoda koji se definiše. (Drugim rečima, naredba dodele *izraza* nekoj promenljivoj čiji je tip jednak tipu navedenom kao tip rezultata metoda mora biti dozvoljena.)

U ovom obliku, return naredba ima dve funkcije: vraćanje vrednosti *izraza* i prekid izvršavanja pozvanog metoda. Preciznije, izvršavanje return naredbe u telu pozvanog metoda se odvija u dva koraka. Prvo se izračunava *izraz* na uobičajen način. Posle toga se završava izvršavanje pozvanog metoda i dobijena vrednost *izraza* i kontrola se vraćaju na mesto gde je taj metod pozvan.

Na primer, pretpostavimo da smo definisali metod koji izračunava hipotenuzu pravouglog trougla po Pitagorinoj formuli:

```
double hipotenuza(double a, double b) {
   // Hipotenuza pravouglog trougla sa katetama a i b
```

```
    return Math.sqrt(a*a + b*b);
}
```

Iz zaglavlja ovog metoda vidimo da metod hipotenuza ima dva parametra a i b tipa double koji predstavljaju vrednosti kateta pravouglog trougla, kao i da je double tip vrednosti koju taj metod vraća. Ovo je neophodno, jer ako su katete realne vrednosti, onda i hipotenuza u opštem slučaju ima realnu vrednost. Primetimo i da nismo naveli nijedan modifikator u zaglavlju ovog metoda, jer to ovde nije bitno i samo bi komplikovalo primer. (Podsetimo se ipak 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.)

Razmotrimo sada detaljno kako se izvršava sledeći programski fragment od dve naredbe:

```
double k1 = 3, k2 = 4;  // vrednosti kateta
double obim = k1 + k2 + hipotenuza(k1, k2);
```

Prvom naredbom se, naravno, rezerviše memorija za promenljive k1 i k2 i u tim lokacijama se redom upisuju vrednosti 3 i 4. U drugoj naredbi se na desnoj strani znaka jednakosti nalazi izraz čija se izračunata vrednost dodeljuje promenljivoj obim. U okviru ovog izraza se nalazi poziv metoda hipotenuza čija vraćena vrednost učestvuje u izračunavanju tog izraza.

Izvršavanje druge naredbe se zato sastoji od više faza, pa ćemo ih sada pažljivo redom ispratiti. Drugom naredbom se najpre rezerviše memorija za promenljivu obim i zatim se izvršava naredba dodele:

```
obim = k1 + k2 + hipotenuza(k1, k2);
```

Ova naredba dodele se izvršava na standardan način: izračunava se vrednost izraza na desnoj strani znaka jednakosti i tako dobijena vrednost se upisuje u memorijsku lokaciju koja je rezervisana za promenljivu na levoj strani znaka jednakosti. Sledeći korak je dakle izračunavanje izraza:

```
k1 + k2 + hipotenuza(k1, k2);
```

Ovaj izraz se isto tako uobičajeno izvršava kako smo objasnili u odeljku 3.5: nad trenutnim vrednostima promenljivih i vraćenim vrednostima poziva metoda u izrazu, primenjuju se navedeni operatori uz poštovanje njihovog prioriteta. U konkretnom slučaju dakle, nakon sabiranja trenutnih vrednosti promenljivih k1 i k2 (3 i 4), tom međurezultatu 7 se dodaje vraćena vrednost poziva metoda:

```
hipotenuza(k1, k2)
```

Ovaj poziv se dalje opet izvršava na standardan način: vrednosti argumenata u pozivu metoda se dodeljuju parametrima metoda u njegovoj definiciji i sa tim inicijalnim vrednostima parametara se izvršava telo metoda u njegovoj definiciji. U konkretnom slučaju dakle, argumenti poziva metoda hipotenuza 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 hipotenuza. Kontrola toka izvršavanja datog programskog segmenta se zato prenosi u telo metoda hipotenuza i redom se izvršavaju sve njegove naredbe dok se ne naiđe na naredbu povratka. Kako vidimo iz definicije, telo metoda hipotenuza se sastoji samo od return naredbe i njenim izvršavanjem se najpre izračunava navedeni izraz u produžetku, odnosno `Math.sqrt(a*a + b*b)`. Vrednost ovog izraza je dakako $\sqrt{9+16} = \sqrt{25} = 5$. Zatim se prekida izvršavanje metoda hipotenuza i izračunata vrednost 5 i kontrola se vraćaju tamo gde je taj metod pozvan.

Ako se podsetimo, metod hipotenuza je pozvan tokom izračunavanja vrednosti izraza:

```
k1 + k2 + hipotenuza(k1, k2);
```

radi sabiranja njegove vraćene vrednosti 5 sa međurezultatom 7 dobijenim nakon sabiranja trenutnih vrednosti promenljivih k1 i k2. Izračunata vrednost ovog izraza je dakle 12 i ona se dodeljuje promenljivoj obim da bi se najzad završilo izvršavanje naredbe dodele:

```
obim = k1 + k2 + hipotenuza(k1, k2);
```

Prema tome, nakon izvršavanja datog programskog fragmenta će se u memoriji programa nalaziti tri promenljive k1, k2 i obim sa vrednostima, redom, 3, 4 i 12.

Obratite pažnju na to da return naredba 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 return naredba izvršava negde u sredini 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. S obzirom na to da takav metod ima „prazan" tip (tj. void) kao

tip rezultata u svojoj definiciji, naredba povratka u ovom slučaju ne sme imati *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 se želi prekinuti izvršavanje takvog metoda negde u sredini. Ako naredba povratka nije izvršena prilikom izvršavanja tela takvog metoda, njegovo izvršavanje se normalno prekida (i kontrola se vraća na mesto gde je pozvan) kada se dođe do kraja tela metoda. S druge strane, kod metoda koji vraća jednu vrednost se u njegovom telu 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.)

5.4 Primeri metoda

Da bismo pokazali praktičnu primenu metoda, sada ćemo napisati nekoliko kompletnih programa koji koriste dodatne metode pored glavnog metoda `main()`. Ovi programi rešavaju iste probleme za koje smo već napisali programe u prethodnim odeljcima, ali bez upotrebe dodatnih metoda.

Svaki metod obavlja neki specifični zadatak koji je uvek deo glavnog zadatka celog programa. U jednostavnim programima u ovom odeljku, zadaci metoda su manje-više očigledni tako da tome nećemo posvetiti veću pažnju. Imajte ipak na umu da je za složeniji problem najteže pitanje to kako njegovo rešenje podeliti u manje zadatke koje obavljaju posebni metodi. Odgovor na to pitanje je predmet kreativne discipline ***dizajna programa*** koja od programera, pored iskustva, zahteva i umetničku crtu.

Prvi primer je redizajn programa za igru pogađanja broja iz odeljka 4.3. Pošto se pogađanje jednog broja može smatrati koherentnim zadatkom, to se nameće kao odgovarajući zadatak za jedan metod. Glavni metod `main()` poziva ovaj metod u petlji sve dok korisnik želi da pogađa novi broj.

LISTING 5.1: Igra pogađanja broja.

```
import java.util.*;

public class PogađanjeBroja {

   public static void main(String[] args) {

      String ponovo; // indikator nastavka igre
      Scanner tastatura = new Scanner(System.in);

      System.out.println("Ovo je igra pogađanja broja.");
      System.out.println("Ja ću zamisliti broj između 1 i 100,");
      System.out.println("a vi treba da ga pogodite.\n");

      do {
         pogodiBroj();  // poziv metoda za jedno pogađanje broja
         System.out.print("Želite li da ponovo igrate (d/n)? ");
         ponovo = tastatura.next();
      } while (ponovo.equals("d"));
      System.out.println("Hvala i do viđenja ...");
   }

   public static void pogodiBroj() {

      int zamišljenBroj; // broj koji je računar izabrao
      int pokušanBroj;   // broj koji je korisnik pokušao

      Scanner tastatura = new Scanner(System.in);
      zamišljenBroj = (int)(100*Math.random()) + 1;
      System.out.println("Zamislio sam ceo broj između 1 i 100.");
      System.out.println("Pokušajte da ga pogodite.\n");

      do {
         System.out.print("Pogodite broj> ");
         pokušanBroj = tastatura.nextInt();
         if (pokušanBroj < zamišljenBroj)
            System.out.println("Zamislio sam veći broj :-(");
         else if (pokušanBroj > zamišljenBroj)
            System.out.println("Zamislio sam manji broj :-(");
         else
            System.out.println("Bravo, pogodili ste broj :-)");
```

```
      } while (pokušanBroj != zamišljenBroj);
   }
}
```

U drugom primeru ćemo napisati program koji određuje da li je neki broj prost. Obratite pažnju na to da odgovarajući metod `jeProst()` ima jedan parametar koji predstavlja ulazni broj, a rezultat metoda je logička vrednost `true` ili `false` zavisno od toga da li je taj broj prost ili ne.

LISTING 5.2: Određivanje da li je broj prost.

```
import java.util.*;

public class ProstBroj {

   public static void main(String[] args) {

      int broj;  // ulazni broj

      // Učitavanje broja preko tastature
      Scanner tastatura = new Scanner(System.in);

      System.out.print("Ovo je program koji ");
      System.out.println("određuje da li broj prost.");

      System.out.print("Unesite broj veći od 1 (0 za kraj): ");
      broj = tastatura.nextInt();
      while (broj != 0) {
         System.out.print("Taj broj ");
         if ( jeProst(broj) )  // poziv metoda
            System.out.print("je ");
         else
            System.out.print("nije ");
         System.out.println("prost.");

         System.out.print("Unesite broj veći od 1 (0 za kraj): ");
         broj = tastatura.nextInt();
      }
   }

   public static boolean jeProst(int n) {
```

```
      int maxDelioc = (int)Math.sqrt(n); // maksimalni delioc

      // Potraga za deliocem između 2 i maxDelioc
      for (int k = 2; k < maxDelioc; k++) {
         if (n % k == 0)
            return false;
      }
      return true;
   }
}
```

U trećem primeru ćemo napisati program koji proverava palindrome, ali na malo drugačiji način. Zadatak programa je podeljen između dva dodatna metoda, pored glavnog. Prvi metod `sredi()` preuređuje ulaznu rečenicu, pretvaranjem svih slova u mala i zanemarivanjem nevažnih znakova, u oblik koji je lakši za proveravanje da li je palindrom. Drugi metod `obrni()` vraća sređenu rečenicu u obrnutom redosledu. Test da li je rečenica palindrom se svodi onda na proveru da li sređena rečenica ima isti oblik kao njena obrnuta verzija.

LISTING 5.3: Određivanje da li je rečenica palindrom.

```
import java.util.*;

public class Palindrom {

   public static void main(String[] args) {

      String red;       // ulazni red znakova rečenice
      String rečenica;  // „sređen" ulazni red znakova
      Scanner tastatura = new Scanner(System.in);

      do {
         System.out.print("Unesite rečenicu (<enter> za kraj): ");
         red = tastatura.nextLine();

         rečenica = sredi(red);              // poziv metoda sredi
         System.out.print("Ta rečenica ");
         if (rečenica.equals(obrni(rečenica)))// poziv metoda obrni
            System.out.print("je ");
         else
            System.out.print("nije ");
```

```
            System.out.println("palindrom.");
         } while (red.length() != 0);
      }

      public static String sredi(String niska) {

         String sređenaNiska = "";

         // Konvertovanje svih znakova ulazne niske u mala slova
         niska = niska.toLowerCase();

         // Zanemarivanje nevažećih znakova ulazne niske
         for (int i = 0; i < niska.length(); i++) {
            char znak = niska.charAt(i);
            if (Character.isLetterOrDigit(znak))
               sređenaNiska += znak;
         }
         return sređenaNiska;
      }

      public static String obrni(String niska) {

         String obrnutaNiska = "";

         for (int i = niska.length() - 1; i >= 0; i--) {
            obrnutaNiska += niska.charAt(i);
         }
         return obrnutaNiska;
      }
}
```

5.5 Preopterećeni metodi

Da bismo mogli pozvati neki metod, pored ostalih informacija o njemu moramo znati njegovo ime, ali i broj, redosled i tipove njegovih parametara. Ovi podaci se nazivaju ***potpis*** metoda. Na primer, potpis metoda

```
public void uradiNešto(int n, double x, boolean test) {
   // Naredbe koje rade nešto ...
}
```

koji smo koristili u odeljku 5.2 je:

```
uradiNešto(int, double, boolean)
```

Obratite pažnju na to da potpis metoda *ne* uključuje 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.

Poznato nam je da se definicija svakog metoda u Javi mora nalaziti unutar neke klase. Ali ono što je neobično je da u Javi neka 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 ali različitim potpisom se nazivaju ***preopterećeni*** (engl. *overloaded*) metodi.

Na primer, u klasi u kojoj je definisan prethodni metod `uradiNešto()` možemo definisati još jedan metod sa istim imenom, ali različitim potpisom:

```
public void uradiNešto(String s) {
   // Naredbe koje rade nešto drugo ...
}
```

Potpis ovog metoda je:

```
uradiNešto(String)
```

što je različito od potpisa prvog metoda `uradiNešto()`. Potpis dva metoda u klasi mora biti različit kako bi Java prevodilac mogao da odredi koji metod se poziva. A to se lako određuje na osnovu broja, redosleda i tipova argumenata navedenih u pozivu metoda. Na primer, naredbom:

```
uradiNešto("super program");
```

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

Na kraju, primetimo da potpis nekog metoda *ne* obuhvata tip rezultata tog metoda. To je razlog što, na primer, u klasi `Scanner` imamo metode sa različitim imenima bez parametara za čitanje vrednosti različitih tipova:

```
nextInt(), nextDouble(), nextBoolean(), ...
```

Kao što u suštini imamo jedan metod, print, za prikazivanje vrednosti različitih tipova,[2] bolje bi bilo kada bismo imali metod sa jednim imenom, recimo na engleskom nextValue, za čitanje vrednosti različitih tipova. Taj metod bi morao biti preopterećen na sledeći način:

```
int     nextValue() { ... }
double  nextValue() { ... }
boolean nextValue() { ... }
...
```

Ali potpisi svih ovih metoda bi bili isti, nextValue(), jer tipovi rezultata ne ulaze u njihove potpise. Pošto u jednoj klasi može postojati samo jedan metod sa imenom nextValue i bez parametara, ova varijanta nije moguća u Javi.

5.6 Lokalne i globalne promenljive

Definicije metoda, kao što nam je poznato, nalaze se unutar klasa. Ali određena klasa može pored metoda obuhvatati i deklaracije promenljivih. Ove promenljive, da bi se razlikovale od promenljivih koje su deklarisane u metodima, često se nazivaju ***polja*** ili ***promenljive-članice***. Na primer, klasa Zgrada iz odeljka 2.2 čija je definicija:

```
class Zgrada
{
   int brojSpratova;
   String boja;
   String adresa;
     ⋮
   public void okreči() { ... };
   public void dozidaj() { ... };
}
```

sadrži tri polja brojSpratova, boja i adresa koja su deklarisana izvan metoda okreči() i dozidaj().

Polja su u klasi deklarisana izvan metoda, pa ako ih posmatramo iz perspektive metoda, kažemo da su to ***globalne*** promenljive za sve metode

[2] Iako metod print nije preopterećen, nego ima promenljiv broj parametara.

klase. Isto tako, za promenljive koje su deklarisane unutar nekog metoda se kaže da su to ***lokalne*** promenljive za taj metod.[3]

Isto kao metodi, polja mogu biti statička (klasna) ili nestatička (objektna). Statičko polje pripada celoj klasi i memorijska lokacija za njega se rezerviše prilikom izvršavanja programa kada Java interpretator prvi put učita njegovu klasu u memoriju. Statičko polje zatim postoji u memoriji sve dok postoji i njegova klasa.

Objektna polja pripadaju pojedinačnim objektima klase, odnosno svaki objekat klase ima svoju kopiju objektnog polja. Memorijska lokacija za objektno polje se prilikom izvršavanja programa rezerviše kada se neki objekat klase konstruiše operatorom new. Objektno polje zatim postoji u memoriji sve dok postoji i njegov objekat.

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 kao što su `static`, `public` i `private`.

Neki primeri deklaracija polja su:

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

Modifikatorom `static` se polje deklariše 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 njegovom pristupu, 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 deklarisano. Pri tome, ako je polje statičko, ovaj zapis ima oblik *`ime-klase.ime-polja`*, dok se za objektna polja mora pisati *`ime-objekta.ime-polja`*. Ovde je *`ime-klase`*

[3] Možemo govoriti i o lokalnim promenljivima za blok, ukoliko su promenljive deklarisane unutar bloka ograničenog vitičastim zagradama.

ona klasa u kojoj je javno statičko polje deklarisano, a *ime-objekta* je neki konstruisani objekat klase u kojoj je objektno polje deklarisano.

Na primer, u poglavlju 3.4 smo pomenuli da je out jedno od javnih statičkih polja klase System. Zato smo u našim klasama morali da pišemo njegovo složeno ime System.out kada smo to polje koristili radi prikazivanja izlaznih podataka na ekranu.

Za drugi primer, pretpostavimo da je polje brojIgrača deklarisano kao javno objektno polje u klasi IgraSaKartama. Onda se ovo polje u svim metodima u klasi IgraSaKartama može koristiti pisanjem njegovog prostog imena brojIgrača. U metodima drugih klasa, ako bi se želelo koristiti objektno polje brojIgrača klase IgraSaKartama, morao bi se najpre konstruisati neki objekat klase IgraSaKartama, recimo poker. Tek posle toga bi se primerak objektnog polja brojIgrača koji pripada objektu poker mogao koristiti pisanjem njegovog složenog imena poker.brojIgrača.

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 memorijska lokacija odgovarajuće veličine za promenljivu i to se ponekad naziva ***alociranje*** promenljive. Postojanje promenljive u programu se završava kada se memorijska lokacija rezervisana za nju oslobodi, moguće da bi se iskoristila za neke druge potrebe. Ovaj postupak se naziva i ***dealociranje*** promenljive, a momenat njegovog dešavanja tokom izvršavanja programa zavisi od vrste promenljive. Tri vrste promenljivih koje možemo koristiti u telu nekog metoda su: lokalne promenljive deklarisane unutar metoda, parametri metoda i globalne promenljive koje su deklarisane izvan metoda, ali u istoj klasi kao i metod.

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 prosto 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. Ali kao što smo objasnili u odeljku 5.2, parametri imaju konceptualno istu ulogu kao lokalne promenljive. To znači da za parametre važe ista razmatranja kao i za lokalne promenljive.

Obratite ovde ipak pažnju na to da promene vrednosti parametara unutar tela metoda nemaju nikakav efekat na ostatak programa. (Bar što se tiče parametara primitivnih tipova; za parametre objektnih tipova je cela istina malo komplikovanija.) Na primer, izvršavanjem metoda `main()` u ovom programskom fragmentu:

```
public static void main(String[] args) {
   int x = 1;
   dodaj1(x);
   System.out.println("x = " + x),
}

public void dodaj1(int x) {
    x = x + 1;
}
```

na ekranu se prikazuje "`x = 1`", a ne "`x = 2`". Razlog za ovo je što promena vrednosti parametra `x` unutar metoda `dodaj1()` nema efekta van tog metoda, pa ni uticaja na promenljivu `x` unutar metoda `main()`. Preciznije, nakon alokacije i dodele broja 1 promenljivoj `x` u metodu `main`, izvršava se poziv `dodaj1(x)` čiji je argument jednak vrednosti promenljive `x`. Izvršavanjem metoda `dodaj1` se najpre alocira i inicijalizuje parametar `x` vrednošću argumenta, tj. brojem 1, a zatim se ovaj parametar uvećava za 1 i dobija vrednost 2. Nakon toga se dealocira parametar `x` i završava poziv metoda `dodaj1`. To znači da izvršavanje poziva metoda `dodaj1` nije uticalo na promenljivu `x` u metodu `main()`, pa se na kraju prikazuje njena neizmenjena vrednost 1.

Globalne promenljive metoda deklarisane unutar neke klase postoje sve dok postoji ta klasa ili objekat te klase u memoriji, zavisno od toga da li se radi o statičkim ili objektnim poljima. To znači da, grubo rečeno, globalna promenljiva postoji od momenta kada se njena klasa prvi put koristi ili objekat te klase konstruiše u programu, pa sve do momenta kada Java interpretator zaključi da njena klasa ili objekat nisu više potrebni. Tačan momenat dealociranja globalnih promenljivih nije bitan, jer konceptualno možemo pretpostaviti da one postoje sve do kraja

izvršavanja celog programa.

Ono što je važno međutim, jeste da primetimo da su globalne promenljive u nekoj klasi nezavisne od metoda te klase i uvek postoje dok se metodi te klase izvršavaju. Zato globalne promenljive mogu koristiti svi metodi u istoj klasi. To znači da promene vrednosti globalne promenljive u jednom metodu imaju prošireni efekat na druge metode. Na primer, ako se u jednom metodu menja vrednost nekoj globalnoj promenljivoj, u drugom metodu se uzima ta promenjena vrednost ukoliko se ova globalna promenljiva koristi 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.

Primer: igra pogađanja broja sa ograničenjem

Poslednja verzija programa za igru pogađanja broja vodi statistiku o tome koliko je puta korisnik pogodio zamišljene brojeve. Naravno, da bi ovo imalo smisla, broj pogađanja jednog broja je ograničen na 5 tako da se korisniku računa da nije pogodio zamišljen broj ukoliko ga ne pogodi u 5 dozvoljenih pokušaja.

Program je organizovan na isti način kao prethodna verzija ovog programa na strani 129: sve dok korisnik želi da pogađa novi zamišljen broj, glavni metod `main()` u petlji poziva metod `pogodiBroj()` koji realizuje pogađanje jednog zamišljenog broja. Pošto vodimo računa o tome koliko puta je korisnik pogodio zamišljene brojeve, uvodimo promenljivu `brojPogodaka` kojoj dodajemo 1 u metodu `pogodiBroj()` svaki put kada korisnik pogodi broj. Njenu vrednost prikazujemo na kraju metoda `main()` kada korisnik odustane od igranja.

Obratite pažnju na to da promenljiva `brojPogodaka` ne može biti lokalna promenljiva, jer nam je potreban pristup toj promenljivoj iz dva metoda. Ona ne može biti ni objektna promenljiva, jer je koriste statički metodi. Zbog toga smo je deklarisali da bude statička globalna promenljiva.

U programu smo deklarisali još dve statičke globalne promenljive, mada to nije obavezno kao za `brojPogodaka`, ali je bolje rešenje. Prva promenljiva je `tastatura` koja predstavlja objekat fizičkog uređaja tastature. Ovu promenljivu koriste oba metoda `main()` i `pogodiBroj()`, pa smo je izvukli da bude na globalnom nivou. Primetimo da smo u

prethodnoj verziji ovog programa koristili lokalne promenljive sa istim imenom tastatura. Stoga se prilikom izvršavanja te verzije alociraju zapravo dve različite promenljive, što je nepotrebno.

Druga promenljiva je, u stvari, konstanta MAX_BROJ_POKUŠAJA čija vrednost ukazuje na najveći dozvoljeni broj pokušaja pogađanja jednog zamišljenog broja. O konstantama u Javi će biti više reči u napomenama iza teksta kompletnog programa.

LISTING 5.4: Igra pogađanja broja sa ograničenjem.

```
import java.util.*;

public class PogadjanjeBroja {

   static int brojPogodaka = 0;        // brojač pogođenih brojeva
   final static int MAX_BROJ_POKUŠAJA = 5;// najveći broj pokušaja
   static Scanner tastatura = new Scanner(System.in);

   public static void main(String[] args) {

      String ponovo;

      System.out.println("Ovo je igra pogađanja broja.");
      System.out.println("Ja ću zamisliti broj između 1 i 100,");
      System.out.print("a vi treba da ga pogodite iz ");
      System.out.println(MAX_BROJ_POKUŠAJA + " puta.\n");

      do {
         pogodiBroj();
         System.out.print("Želite li da ponovo igrate (d/n)? ");
         ponovo = tastatura.next();
      } while (ponovo.equals("d"));

      System.out.print("Pogodili ste ");
      if (brojPogodaka == 1)
         System.out.println("jedanput.");
      else
         System.out.println(brojPogodaka + " puta.");
      System.out.println("Hvala i do viđenja ...");
   }

   public static void pogodiBroj() {
```

```
      int zamišljenBroj; // broj koji je računar izabrao
      int pokušanBroj;   // broj koji je korisnik pokušao
      int brojPokušaja;  // brojač pokušaja korisnika

      zamišljenBroj = (int)(100 * Math.random()) + 1;
      brojPokušaja = 0;

      System.out.println("Zamislio sam ceo broj između 1 i 100.");
      System.out.println("Pokušajte da ga pogodite.\n");

      for (;;) {  // beskonačna petlja!
         System.out.print("Pogodite broj> ");
         pokušanBroj = tastatura.nextInt();
         brojPokušaja++;

         if (pokušanBroj == zamišljenBroj) {
            System.out.println("Bravo, pogodili ste broj.");
            brojPogodaka++;
            break;
         }
         else if (brojPokušaja == MAX_BROJ_POKUŠAJA) {
            System.out.print("Niste pogodili broj iz ");
            System.out.println(MAX_BROJ_POKUŠAJA + " puta.");
            System.out.print("Zamislio sam broj ");
            System.out.println(zamišljenBroj);
            break;
         }
         else if (pokušanBroj < zamišljenBroj)
            System.out.println("Zamislio sam veći broj.");
         else if (pokušanBroj > zamišljenBroj)
            System.out.println("Zamislio sam manji broj.");
         else {
            System.out.println("Ovaj slučaj nije moguć.");
            break;
         }
      }
   }
}
```

Sledi lista posebnih i opštih napomena u vezi sa ovim programom:

- Petlja u metodu pogodiBroj() kojom se realizuje pogađanje zamišljenog broja je tipični slučaj kada je korisno koristiti beskonačnu petlju. Naime, uslov prestanka pogađanje broja je vrlo komplikovan i običnom petljom bi se dobilo prilično neelegantno rešenje. S druge strane, logika petlje se lako prepoznaje ukoliko se koristi beskonačna petlja i break naredbe. Obratite ipak pažnju na to da je bitan redosled logičkih uslova kod višestruke if naredbe, jer se oni ispituju redom odozgo na dole prilikom izvršavanja. Primetimo i da je ovde poslednji else slučaj suvišan i da se nikad ne može desiti. Bez obzira na to, programeri ga često navode iz dva razloga: prvo, da bi u programu jasno ukazali da je odgovarajući slučaj nemoguć; i drugo, ako se ipak desi (možda usled neke druge greške), da bi se to otkrilo na kontrolisan način.

- Lokalne promenljive za neki metod se u Javi ne inicijalizuju prilikom izvršavanja tog metoda. Stoga one moraju imati eksplicitno (nesumnjivo) dodeljenu vrednost u metodu pre nego što se mogu koristiti. S druge strane, globalne promenlive (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.) Zato, na primer, globalnu promenljivu brojPogodaka nismo morali da eksplicitno inicijalizujemo nulom, jer bi se to ionako automatski uradilo.

- Ponekad neka promenljiva u programu ne treba da se menja nakon početne dodele vrednosti. U prethodnom programu to je slučaj sa promenljivom MAX_BROJ_POKUŠAJA. U ovom slučaju, namera je pre bila da se bezličnom broju 5 dodeli smisleno ime kako bi se program lakše razumeo. U Javi se u naredbi deklaracije može dodati modifikator final kako bi se obezbedilo da se promenjiva ne može promeniti nakon što se inicijalizuje. Tako, iza naredbe deklaracije:

  ```
  final static int MAX_BROJ_POKUŠAJA = 5;
  ```

  svaki pokušaj promene vrednosti promenljive MAX_BROJ_POKUŠAJA izazvaće grešku.

  Ove „nepromenljive promenljive" se nazivaju ***imenovane konstante***, jer 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 Java standardnim klasama u kojima su definisane mnoge imenovane konstante. Tako na primer klasa `Math` sadrži konstantu `Math.PI` za označavanje broja π ili klasa `Integer` sadrži konstantu `MIN_VALUE` za označavanje minimalne vrednosti tipa `int`.

5.7 Oblast važenja imena

Do sada smo upoznali razne programske konstrukcije u Javi kojima se daju imena u programu: promenljive, metode, parametre metoda i tako dalje. U prethodnom odeljku smo videli da se promenljive dinamički alociraju i dealociraju tokom izvršavanja programa. Teorijski je zato moguće da se neke programske konstrukcije koriste u tekstu programa iako one fizički ne postoje u memoriji. Da bi se izbegle ove greške, 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 je cela klasa u kojoj je definisana. To znači da se globalna promenljiva može koristiti u tekstu cele klase, čak i eventualno ispred naredbe njene deklaracije.[4] S druge strane, oblast važenja imena lokalne promenljive je od mesta njene deklaracije do kraja bloka u kojem 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:

[4] Zato je moguće pisati deklaracije polja na kraju definicije klase, što je po nekima bolji način.

```
public 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 nije opet zaklonjeno istim imenom lokalne promenljive ili parametra u nekom drugom metodu. Ako ipak želimo da koristimo statičko polje `pobednik` unutar metoda `odigrajIgru()`, onda moramo pisati njegovo puno ime `IgraSaKartama.pobednik`, kao što to moramo raditi u nekoj drugoj klasi.

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` naredbe. Na primer:

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

U ovom primeru bismo mogli reći da je promenljiva `i` deklarisana lokalno u odnosu na metod koji sadrži `for` naredbu. Ipak, oblast važenja ovako deklarisane kontrolne promenljive je samo kontrolni deo i telo `for` naredbe, odnosno ne proteže se do kraja tela metoda koji sadrži `for` naredbu. 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 se njihova imena mogu ponovo davati. 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 govorimo o rekurzivnim metodima, o kojima će više reči biti u nastavku knjige.

Ovo opšte pravilo ima dodatni uslov s obzirom na to da metodi mogu biti statički ili objektni. Naime, objektni metodi i polja klase, koji pri-

padaju nekom objektu te 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 članovi klase (metodi i polja) ne mogu koristiti u statičkim metodima iste klase.

U Javi nije obavezno da promenljive i metodi imaju različita imena, jer se uvek može prepoznati da li se neko ime odnosi na promenljivu ili na metod: iza imena metoda uvek sledi leva zagrada. Čak ni imena klasa ne moraju da budu različita od imena promenljivih i metoda, jer sintaksna pravila jezika garantuju da Java prevodilac uvek može razlikovati ova imena. Ove mogućnosti mogu dovesti do apsurda, s obzirom na to da klasa definiše tip podataka u Javi koji se može iskoristiti za definisanje tipa promenljivih, kao i parametara i vraćene vrednosti metoda. To znači da je u Javi dozvoljeno definisati, na primer, klasu sa imenom `Ludost` u kojoj je definisan metod:

```
public Ludost Ludost( Ludost Ludost ) {
   // Telo metoda
}
```

Prva reč `Ludost` označava tip rezultata metoda, a druga reč `Ludost` je ime metoda. Treća reč `Ludost` označava tip parametra ovog metoda, dok je četvrta reč `Ludost` ime tog parametra. Ovo je dobar primer zašto treba negovati dobar stil programiranja, jer sve što je dozvoljeno nije često i ono što je dobro.

5.8 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 realni broj različit od nule i n ceo broj veći ili jednak nuli. Mada za ovaj problem možemo iskoristiti gotov metod `Math.pow(x,n)`, pošto stepen x^n predstavlja množenje broja x sa samim sobom n puta, to možemo i sami lako izračunati. Naime, primenjujući sličnu ideju kao kod izračunavanja

zbira većeg niza brojeva, n-ti stepen broja x možemo dobiti uzastopnim množenjem parcijalno izračunatog stepena x^i sa x za $i = 0, 1, \ldots, n-1$. Metod stepen1 koristi for petlju radi realizacije ovog iterativnog postupka:

```
double stepen1(double x, int n) {

   double y = 1;
   for(int i = 0; i < n; i++)
      y = y * x;
   return y;
}
```

Pored ovog klasičnog (iterativnog) načina, za izračunavanje stepena možemo primeniti drugačiji (rekurzivni) pristup ukoliko uočimo 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 za stepen $n > 0$ rezultat dobijamo ako x pomnožimo sa $(n-1)$-im stepenom broja x. Drugi metod stepen2 je definisan tako da koristi ovaj način za izračunavanje n-tog stepena broja:

```
double stepen2(double x, int n) {

   if (n == 0)
      return 1;
   else
      return x * stepen2(x, n-1);
}
```

Obratite pažnju na to da, za izračunavanje n-tog stepena broja x, metod stepen2 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 (kao stepen2) ako se u njegovoj definiciji nalazi poziv samog metoda koji se definiše. 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 rešavaju neki zadatak svođenjem polaznog problema na sličan prostiji problem (ili više njih). Na primer, metod stepen2 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 metoda `stepen2` 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 moramo 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 svakom rekurzivnom metodu mora nalaziti ***bazni slučaj*** za najprostiji zadatak čije je rešenje unapred poznato i koje se ne dobija rekurzivnim pozivom. To je kod metoda `stepen2` slučaj kada je stepen $n = 0$, jer se onda rezultat 1 neposredno dobija bez daljeg rekurzivnog rešavanja.

Prema tome, rekurzivni metod generiše 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.

Radi boljeg razumevanja ovih osnovnih elemenata rekurzivne tehnike programiranja, u nastavku odeljka ćemo pokazati još neke primere rekurzivnih metoda.

Primer: najveći zajednički delilac

Razmotrimo najpre problem izračunavanja najvećeg zajedničkog delioca dva pozitivna cela broja x i y. Ovaj naizgled jednostavan primer ima značajnu primenu u kriptografiji.

Najveći zajednički delilac za dva pozitivna cela broja x i y, u oznaci $\mathrm{nzd}(x, y)$, predstavlja najveći ceo broj koji deli bez ostatka oba broja x i y. Da li je ovo dobra definicija? Drugim rečima, da li svaki par pozitivnih celih brojeva ima najveći zajednički delilac? Za svaki par pozitivnih celih brojeva je jasno da je uvek broj 1 zajednički delilac, a najveći broj koji

istovremeno može da deli x i y bez ostataka je onaj broj koji je manji od x i y. Dakle, $\text{nzd}(x, y)$ uvek postoji i leži negde u zatvorenom intervalu između 1 i $\min\{x, y\}$.

Prethodna napomena o egzistenciji najvećeg zajedničkog delioca za x i y može se iskoristiti za iterativni način njegovog izračunavanja. Naime, ukoliko počnemo od manjeg od dva pozitivna cela broja x i y i zatim redom proveravamo sve brojeve za jedan manje da li su zajednički delioci za x i y, prvi nađeni zajednički delilac biće ujedno i najveći.

Drugi, rekurzivni način izračunavanja najvećeg zajedničkog delioca za x i y se zasniva na činjenici da je $\text{nzd}(x, y)$ jednak $\text{nzd}(y, x \,\%\, y)$, pod pretpostavkom $x \geq y$.[5] (Podsetimo se da % označava operator izračunavanja ostatka pri celobrojnom deljenju.) Preciznije rečeno,

$$\text{nzd}(x, y) = \begin{cases} y, & \text{ako je } x \,\%\, y = 0 \\ \text{nzd}(y, x \,\%\, y), & \text{ako je } x \,\%\, y \neq 0\,. \end{cases}$$

Na osnovu ove rekurzivne definicije sada je lako napisati rekurzivni metod za izračunavanje $\text{nzd}(x, y)$:

```
// Pretpostavlja se da je x ≥ y
int nzd(int x, int y) {

   if (x % y == 0)  // bazni slučaj
      return y;
   else
      return nzd(y, x % y);
}
```

U ovom metodu obratite pažnju na to da su u rekurzivnom pozivu $\text{nzd}(y, x \,\%\, y)$ oba argumenta manja od polaznih brojeva x i y. To znači da se tim pozivom zaista rešava prostiji zadatak izračunavanja najvećeg zajedničkog delioca za par manjih brojeva. Ostatak pri celobrojnom deljenju u lancu rekurzivnih poziva se dakle stalno smanjuje i zato u jednom trenutku mora biti nula. Tada će se lanac prekinuti, jer se onda rezultat neposredno dobija kao vrednost drugog argumenta poslednjeg poziva u lancu.

[5] Naime, pošto je $x = ky + x \,\%\, y$ za neki ceo broj k, svaki delilac za x i y je delilac i za brojeve y i $x \,\%\, y$. Obrnuto, svaki delilac za brojeve y i $x \,\%\, y$ je delilac i za x i y. Kako parovi brojeva x i y i y i $x \,\%\, y$ imaju isti skup zajedničkih delioca, to su i njihovi najveći zajednički delioci jednaki.

Primer: Fibonačijev niz brojeva

Fibonačijev niz brojeva je ime dobio po italijanskom srednjevekovnom matematičaru Leonardu Fibonačiju koji ga je prvi proučavao. U današnje vreme je ovaj niz brojeva našao brojne primene u umetnosti, matematici i računarstvu.

Fibonačijev niz brojeva počinje sa prva dva broja 1 i 1, a svaki sledeći broj u nizu se onda 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. Prema tome, početni deo Fibonačijevog niza brojeva je:

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

Ako brojeve Fibonačijevog niza označimo sa $f_1, f_2, f_3, \ldots$, onda je očevidno rekurzivna definicija za n-ti broj Fibonačijevog niza:

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

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

Za izračunavanje n-tog broja Fibonačijevog niza možemo lako napisati rukurzivni metod ukoliko iskoristimo prethodnu rekurzivnu definiciju tog broja:

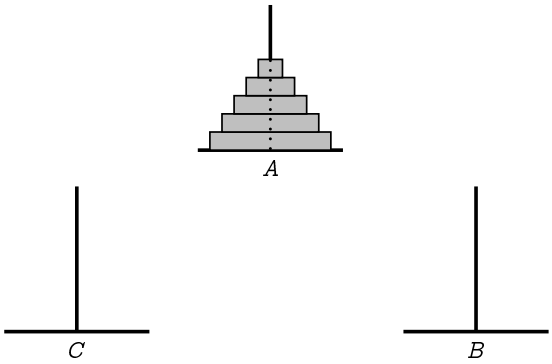
```
// Pretpostavlja se da je n ≥ 1
int fib(int n) {

   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.

Primer: Hanojske kule

Problem Hanojskih kula je jednostavna igra u kojoj su na početku n diskova različite veličine poređani na jednom od tri stuba u opadajućem redosledu svoje veličine od podnožja tog stuba ka vrhu. Svi diskovi treba da se premeste na drugi stub koristeći treći stub kao pomoćni. Pri tome, samo disk sa vrha jednog stuba se može premesti na vrh drugog stuba i nikad se veći disk ne sme postaviti iznad manjeg diska. Na slici 5.1 je prikazana početna konfiguracija ove igre za $n = 5$ diskova, ukoliko zamislimo da su tri stuba trougaono raspoređeni.



SLIKA 5.1: Početna konfiguracija igre Hanojskih kula.

Preciznije rečeno, igra Hanojske kule se sastoji od tri stuba A, B, C i n diskova različite veličine prečnika $d_1 < d_2 < \cdots < d_n$. Svi diskovi su početno poređani na stubu A u opadajućem redosledu svoje veličine od podnožja stuba A, odnosno disk najvećeg prečnika d_n se nalazi na dnu i disk najmanjeg prečnika d_1 se nalazi na vrhu stuba A. Cilj igre je premestiti sve diskove na stub C koristeći stub B kao pomoćni i poštujući dva pravila:

- Samo se disk sa vrha jednog stuba može premestiti na vrh drugog stuba.
- Nikad se veći disk ne može nalaziti iznad manjeg diska.

Primetimo da ova pravila impliciraju da se samo po jedan disk može premeštati, kao i da se svi diskovi u svakom trenutku moraju nalaziti samo na stubovima.

Bez mnogo objašnjenja, jer nam to nije cilj, navodimo rešenje igre koje se dobija iterativnim postupkom korak po korak: u svakom neparnom koraku, premestiti najmanji disk na naredni disk u smeru kretanja kazaljke na satu; u svakom parnom koraku, legalno premestiti jedini mogući disk koji nije najmanji. Ovaj postupak je ispravan (kao što se u to pažljivi čitaoci mogu uveriti), ali je teško razumeti zašto je to tako i još je teže doći do njega.

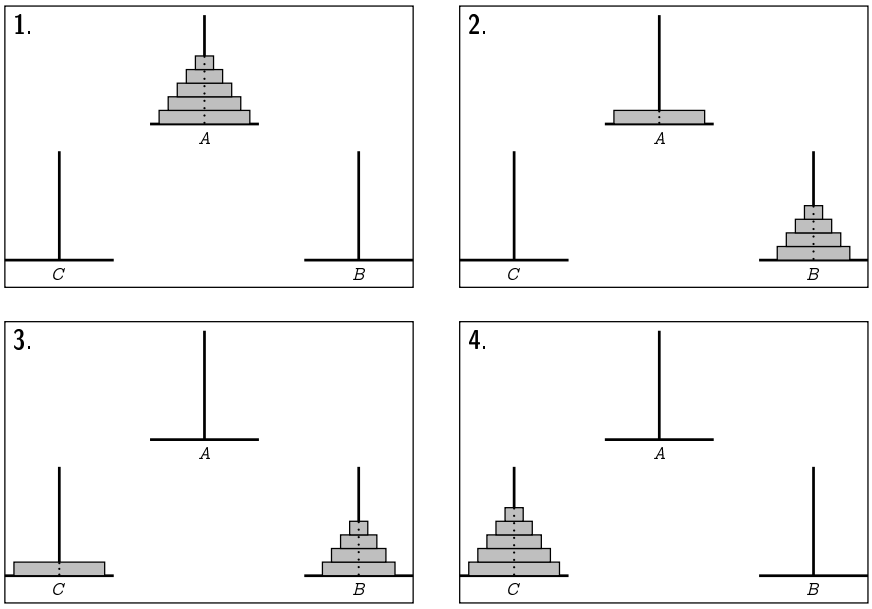
Igra Hanojskih kula je primer problema čije se rešenje mnogo lakše i prirodnije dobija rekurzivnim postupkom. Zadatak premeštanja n diskova sa stuba A na stub C koristeći B kao pomoćni stub možemo svesti na dva slična prostija zadatka premeštanja $(n-1)$-og diska. Prvi zadatak je premeštanje $n-1$ najmanjih diskova sa stuba A na stub B koristeći C kao pomoćni stub. Drugi zadatak je premeštanje $n-1$ najmanjih diskova sa stuba B na stub C koristeći A kao pomoćni disk. (Ovo su prostiji zadaci u smislu da se rešava sličan problem za manji broj diskova.) Polazni problem za n diskova onda možemo rešiti rekurzivnim postupkom koji se sastoji od tri koraka:

1. Primenjujući (rekurzivno) postupak za rešenje prvog zadatka, najpre premeštamo $n-1$ najmanjih diskova sa stuba A na stub B koristeći stub C kao pomoćni.
2. Na taj način nam ostaje samo najveći disk na stubu A. Zatim premeštamo taj najveći disk sa stuba A na slobodni stub C.
3. Na kraju, primenjujući (opet rekurzivno) postupak za rešenje drugog zadatka, svih $n-1$ najmanjih diskova koji se nalaze na stubu B premeštamo na stub C koristeći stub A kao pomoćni.

Ovaj rekurzivni postupak je ilustrovan na slici 5.2. Na toj slici su prikazani rezultati prethodna tri koraka od početne konfiguracije za $n = 5$ diskova.

Obratite pažnju na to koliko je rekurzivno rešenje problema Hanojskih kula konceptualno jednostavnije za razumevanje. Možda još važnije, verovatno su i sami čitaoci relativno lako došli do njega. Ova jednostavnost rekurzivnog postupka je prevashodno ono što čini rekurzivnu tehniku važnom, mada su u mnogim slučajevima dobijeni rekurzivni metodi i efikasniji od klasičnih.

Ako pretpostavimo da se jedan disk na vrhu stuba x premešta na vrh diskova na stubu y pomoću metoda `premestiDisk(x,y)`, onda sledeći



SLIKA 5.2: Rekurzivno rešenje problema Hanojskih kula.

jednostavni rekurzivni metod premestiDiskove(n,a,b,c) premešta n diskova sa stuba a na stub c koristeći b kao pomoćni stub.

```
void premestiDiskove(int n, char a, char b, char c) {

   if (n == 1)  // bazni slučaj
      premestiDisk(a, c);
   else {
      premestiDiskove(n-1, a, c, b);
      premestiDisk(a, c);
      premestiDiskove(n-1, b, a, c);
   }
}
```

Sledeći listing sadrži kompletan program u kojem se od korisnika najpre dobija željeni broj diskova za igru Hanojskih kula, a zatim se na ekranu prikazuju svi koraci rešenja za premeštanja tih diskova sa stuba A na stub C koristeći stub B kao pomoćni. Na primer, za četiri diska se na ekranu dobija rešenje u ovom obliku:

```
Disk na vrhu stuba A premestiti na vrh stuba B
Disk na vrhu stuba A premestiti na vrh stuba C
```

```
Disk na vrhu stuba B premestiti na vrh stuba C
Disk na vrhu stuba A premestiti na vrh stuba B
Disk na vrhu stuba C premestiti na vrh stuba A
Disk na vrhu stuba C premestiti na vrh stuba B
Disk na vrhu stuba A premestiti na vrh stuba B
Disk na vrhu stuba A premestiti na vrh stuba C
Disk na vrhu stuba B premestiti na vrh stuba C
Disk na vrhu stuba B premestiti na vrh stuba A
Disk na vrhu stuba C premestiti na vrh stuba A
Disk na vrhu stuba B premestiti na vrh stuba C
Disk na vrhu stuba A premestiti na vrh stuba B
Disk na vrhu stuba A premestiti na vrh stuba C
Disk na vrhu stuba B premestiti na vrh stuba C
```

LISTING 5.5: Hanojske kule.

```
import java.util.*;

public class HanojskeKule {

   public static void main(String[] args) {
      int n;  // broj diskova

      Scanner tastatura = new Scanner(System.in);

      System.out.print("Unesite broj diskova Hanojskih kula> ");
      n = tastatura.nextInt();

      System.out.println();
      System.out.println("Rešenje igre za taj broj diskova je: ");

      premestiDiskove(n, 'A', 'B', 'C');
   }

   static void premestiDiskove(int n, char a, char b, char c) {

      if (n == 1)  // bazni slučaj
         premestiDisk(a, c);
      else {
         premestiDiskove(n-1, a, c, b);
         premestiDisk(a, c);
         premestiDiskove(n-1, b, a, c);
      }
```

```
   }

   static void premestiDisk(char x, char y) {

      System.out.print("Disk na vrhu stuba " + x);
      System.out.println(" premestiti na vrh stuba " + y);
   }
}
```

Primetimo da smo metodu premestiDiskove() u kompletnom programu morali da dodamo kvalifikator static, jer se poziva iz statičkog metoda main(). Slično važi i za metod premestiDisk().

Glava

6

Klase i objekti

Metod u Javi predstavlja programsku konstrukciju za obavljanje jednog specifičnog zadatka u programu. Objekti su složeniji vid programske konstrukcije, jer mogu obuhvatati kako podatke u formi promenljivih, tako i više zadataka u vezi sa tim podacima u formi različitih metoda.

Objektno orijentisano programiranje (OOP) je jedan pristup za rešavanje problema na računaru kojim se apstrahuju specifičnosti računara i rešenje traži na prirodan način koji bi se primenio u svakodnevnom životu. U starijem, proceduralnom programiranju je programer morao da identifikuje računarski postupak za rešenje problema i da napiše niz programskih naredbi kojim se realizuje taj postupak. Naglasak u OOP nije na računarskim postupcima nego na objektima — softverskim elementima sa podacima i mogućnostima, koji uz to mogu da dejstvuju jedni na druge. Objektno orijentisano programiranje se zato svodi na dizajniranje skupa objekata kojima se na prirodan način modelira problem koji se rešava. Pri tome, softverski objekti u programu mogu predstavljati ne samo realne, nego i apstraktne entitete iz odgovarajućeg problemskog domena.

Objektno orijentisane tehnike se, u principu, mogu koristiti u svakom programskom jeziku. To je posledica činjenice što objekte iz standardnog ugla gledanja na stvari možemo 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 takav objektno orijentisan programski jezik koji podržava sve koncepte

objektno orijentisane paradigme.

U programima iz prethodnih poglavlja smo koristili neke standardne klase koje su unapred napisane kao deo jezika Java. Ipak skoro da uopšte nismo imali dodira sa objektno orijentisanim programiranjem, jer se svaki program sastojao od samo jedne klase koja je obuhvatala obavezni metod `main()` i eventualno neke dodatne metode. Ovu jedinu klasu smo morali da pišemo, jer se u Javi svi programski elementi moraju nalaziti unutar neke klase. Ali ako to zanemarimo i uz malo izmenjenu terminologiju, naši programi su isključivo primenjivali klasični, proceduralni pristup rešavanju problema.

Sledeći stadijum u učenju Jave je korišćenje njenih objektno orijentisanih mogućnosti i pisanje sopstvenih klasa za potrebe programa. Zbog toga ćemo u nastavku obratiti pažnju na detalje koncepta klase i njenog definisanja u Javi, kao i na potpuno razmevanje objekata i njihovog konstruisanja u programu.

6.1 Klasa i njeni članovi

Do sada smo naučili da klasa u Javi sadrži (globalne) promenljive i metode. Ovi članovi klase mogu biti statički (što se prepoznaje po modifikatoru `static` u definiciji člana) ili nestatički (objektni). S druge strane, ako i objekti sadrži promenljive i metode, po čemu se objekti razlikuju od klasa? Dobro razumevanje ove razlike, koju ćemo pokušati da razjasnimo u nastavku, jeste ključno za shvatanje suštine objektno orijentisanog programiranja.

U uvodnim napomenama o klasama smo generalno govorili da klasa opisuje objekte sa zajedničkim osobinama. Tačnije je reći da nestatički deo neke klase opisuje objekte koji pripadaju toj klasi. Ali najtačnije iz programerske perspektive je da klasa služi za konstruisanje objekata. Drugim rečima, neka klasa je šablon za konstruisanje objekata, pri čemu svaki konstruisan objekat te klase sadrži sve nestatičke članove klase.

Za objekat konstruisan na osnovu definicije neke klase se kaže da ***pripada*** toj klasi ili da je ***instanca*** te klase. Nestatički članovi klase (promenljive i metodi), koji ulaze u sastav konstruisanih objekata, nazivaju se i ***objektni*** ili ***instancni*** članovi (promenljive i metodi).

Glavna razlika između klase i objekata je dakle to što se klasa *definiše* u tekstu programa, dok se objekti te klase *konstruišu* operatorom new u tekstu programu. 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 izvršavanja operatora new kojim se konstruišu, moguće negde u sredini programa, do trenutka kada više nisu potrebni.

Posmatrajmo jednostavnu klasu koja može poslužiti za čuvanje osnovnih informacija o korisniku u dvema statičkim promenljivim:

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

Ova klasa sadrži statičke promenljive `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 imamo memorijski prostor za čuvanje podataka o samo jednom korisniku. Klasa `Korisnik` i njene dve statičke promenljive `Korisnik.ime` i `Korisnik.id` postoje sve vreme izvršavanja programa.

Posmatrajmo sada sličnu klasu koja sadrži nestatičke promenljive:

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

U ovom slučaju nemamo promenljive `Klijent.ime` i `Klijent.id`, jer su `ime` i `id` nestatički (objektni) članovi klase `Klijent`. Ova klasa dakle ne sadrži ništa konkretno, osim šablona za konstruisanje objekata te klase. Ali to je veliki potencijal, jer ova klasa može poslužiti za stvaranje velikog broja objekata. Svaki konstruisani objekat ove klase imaće *svoje* primerke promenljivih `ime` i `id`. Zato program koji koristi ovu klasu može modelirati više klijenata, jer po potrebi možemo konstruisati nove objekte za predstavljanje novih klijenata. Ako je to, recimo, neki bankarski program, kada klijent otvori račun u banci možemo konstruisati novi objekat ove klase za čuvanje podataka o tom klijentu. Kada neki klijent zatvori račun,

objekat koji ga predstavlja u programu može se ukloniti. Stoga u tom programu kolekcija objekata 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;
}
```

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

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

Mada smo u prethodnim primerima koristili samo promenljive klase, isto važi i za metode koji su definisani u klasi. Statički metodi pripadaju klasi i postoje sve vreme izvršavanja programa kad i klasa. Zato se statički metodi mogu pozivati u programu nezavisno od konstruisanih objekata klase, odnosno čak i ako nije konstruisan nijedan objekat.

Definicije objektnih metoda se nalaze unutar teksta klase, ali objektni metodi logički pripadaju konstruisanim objektima, a ne klasi. To znači da se objektni metodi mogu primenjivati samo na odgovarajući objekat koji je prethodno konstruisan. Na primer, klasa `ČlanPorodice` može imati objektni metod `uvećajUzrast()` kojim se broj godina člana porodice uvaćava za 1 nakon njegovog rođendana. Metodi `uvećajUzrast()` koji pripadaju različitim objektima klase `ČlanPorodice` obavljaju isti zadatak u smislu da povećavaju uzrast odgovarajućeg člana porodice. Ali njihov stvarni efekat je različit, jer uzrasti članova porodice mogu biti različiti. Prema tome, definicija objektnih metoda u klasi određuje zadatak koji obavljaju nad konstruisanim objektima, ali specifični efekat koji izazi-

vaju može varirati od objekta do objekta, zavisno od vrednosti njihovih objektnih promenljivih.

Kao što vidimo, statički i nestatički (objektni) članovi klase su vrlo različiti koncepti i služe za različite svrhe. Važno je praviti razliku između teksta definicije klase i samog pojma klase. Neka 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.

Opštu diskusiju o klasama završimo jednom terminološkom napomenom. Da bi se bolje razlikovale obične promenljive od promenljivih klasa i objekata, ove druge se često nazivaju ***polja***. Tako govorimo o poljima klase i objekata umesto o statičkim i objektnim promenljivim. Radi veće jasnoće u nastavku teksta, naše dalje izlaganje će se u velikoj meri bazirati na ovoj terminologiji.

Sada možemo dati opšti oblik definicije klase u Javi:

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

Deo *modifikatori* na početku definicije klase nije obavezan, ali se može sastojati od jedne ili više službenih reči kojima se određuju izvesne karakteristike klase. Na primer, specifikator pristupa `public` ukazuje da se klasa može koristiti izvan svog paketa. Inače, bez tog specifikatora, klasa se može koristiti samo unutar svog paketa. Na raspolaganju su još tri modifikatora koje ćemo upoznati u daljem tekstu.

Deo *ime-klase* je uobičajen identifikator u Javi, uz konvenciju da sve reči imena klase počinju velikim slovom. Najzad, deo *telo-klase* sadrži definicije (klasnih i objektnih) polja i metoda.

Do sada smo u knjizi već mnogo puta pisali definicije klasa, ali tek sada formalno znamo tačnu sintaksu za definisanje neke klase. Zato navodimo još jedan jednostavan primer radi kompletnosti:

```
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() {  // izračunati prosek ocena
       return (test1 + test2 + test3) / 3;
    }
}
```

Ova klasa može poslužiti u programu za predstavljanje studenata i njihovih rezultata pokazanih na testovima za jedan predmet. Nijedan od članova ove klase nema modifikator `static`, što znači da klasu 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()`. Ova polja u različitim objektima će generalno imati različite vrednosti. Metod `prosek()` primenjen na različite objekte će generalno davati različite rezultate, jer će se za svaki objekat koristiti vrednosti njegovih polja. (Ovaj efekat je zapravo tačno ono što se misli kada se kaže da objektni metod pripada pojedinačnim objektima, a ne klasi.)

6.2 Promenljive klasnog tipa i objekti

Definicijom neke klase u Javi se uvodi novi tip podataka, sličan primitivnim tipovima kao što su `int` ili `char`. Vrednosti tog klasnog tipa su objekti te klase. Iako ova činjenica zvuči pomalo tehnički, ona ima značajne posledice. To znači da ime definisane klase možemo koristiti za tip promenljive u naredbi deklaracije, kao i za tip formalnog parametra i za tip rezulatata u definiciji nekog metoda. Na primer, ako imamo u vidu prethodnu definiciju klase `Student`, u programu možemo deklarisati promenljivu `s` tipa `Student`:

```
Student s;
```

Kao što je to uobičajeno, ovom naredbom se u memoriji računara rezerviše prostor za promenljivu `s` radi čuvanja 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, već promenljiva može sadržati samo referencu na neki objekat. Da bismo ovo razjasnili, moramo malo 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 (*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 svodi se na adresu memorijske lokacije objekta u hip memoriji.

Promenljiva klasnog tipa u Javi 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 koga ukazuje aktuelna vrednost (referenca) te promenljive.

Već smo pominjali da se konstruisanje objekata u programu izvodi operatorom new. Sada možemo dodati da se kao rezultat ovog operatora dobija i referenca na novokonstruisani objekat. To je neophodno da bismo kasnije u programu mogli pristupiti tom objektu i s njim uradili nešto korisno. A u tom cilju, vraćenu referencu na novi objekat moramo sačuvati u promenljivoj klasnog tipa, inače novom objektu nikako ne možemo pristupiti. Na primer, ako je promenljiva s tipa Student deklarisana kao u prethodnom primeru, izvršavanjem naredbe dodele:

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

najpre bi se konstruisao novi objekat koji je instanca klase Student, a zatim bi se referenca na njega dodelila promenljivoj s. 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 terminologiju). Još manje je ispravno reći da promenljiva s sadrži taj objekat. Ispravnan način izražavanja je da promenljiva s ukazuje na novi objekat. (Ove tačnije terminologije ćemo se pridržavati u tekstu sem ako to nije zaista preterano, kao na primer u slučaju stringova).

Konstruisani objekat klase Student, na koga ukazuje promenljiva s, sadrži dakle polja ime, test1, test2 i test3. U programu se ova polja tog konkretnog objekta mogu koristiti pomoću standardne tačka-notacije: s.ime, s.test1, s.test2 i s.test3. Tako, na primer, možemo 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 koga ukazuje promenljiva s. Opštije, polja `s.ime` i, recimo, `s.test1` mogu se koristiti u programu svuda gde je dozvoljena upotreba promenivih tipa `String` odnosno `double`. Na primer, ako želimo broj znakova stringa u polju `s.ime`, možemo pisati `s.ime.length()`.

Slično, objektni metod `prosek()` se može pozvati za objekat na koga ukazuje s zapisom `s.prosek()`. Da bismo prikazali prosečnu ocenu studenta na koga ukazuje s možemo pisati, na primer:

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

U nekim situacijama je potrebno naznačiti da promenljiva klasnog tipa ne ukazuje ni na jedan objekat. U takvim slučajevima toj promenljivoj možemo dodeliti specijalnu referencu `null`. Na primer, možemo pisati naredbu dodele:

```
s = null;
```

ili naredbu grananja oblika:

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

Ako je vrednost promenljive klasnog tipa jednaka `null`, tada se ne mogu koristiti objektne promenljive i metodi preko te promenljive. To je logično jer nema objekta na koga promenljiva ukazuje, pa ni objektnih promenljivih i metoda koji se mogu koristiti. Tako, ako promenljiva s ima vrednost `null`, onda bi bila greška koristiti, recimo, `s.test1` ili `s.prosek()`.

Vezu između promenljivih klasnog tipa i objekata sada ćemo pokušati da razjasnimo na konkretnom primeru izvršavanja programskog fragmenta prikazanog u listingu 6.1.

LISTING 6.1: Veza između promenljivih klasnog tipa i objekata.

```
Student s1, s2, s3, s4;
s1 = new Student();
s2 = new Student();
```

```
s1.ime = "Pera Perić";
s2.ime = "Mira Mirić";
s3 = s1;
s4 = null;
```

Prvom naredbom u ovom programskom fragmentu deklarišu se četiri promenljive klasnog tipa Student. Drugom i trećom naredbom se konstruišu dva objekata klase Student i reference na njih se dodeljuju redom promenljivim s1 i s2. Narednim naredbama se odgovarajućim primercima polja ime ovih novih objekata dodeljuje vrednost koja je jednaka nekom stringu. (Podsetimo se da ostala polja dobijaju podrazumevane vrednosti prilikom konstruisanja objekata.) Na kraju, pretposlednjom naredbom se vrednost promenljive s1 dodeljuje promenljivoj s3, a poslednjom naredbom se referenca null dodeljuje promenljivoj s4.

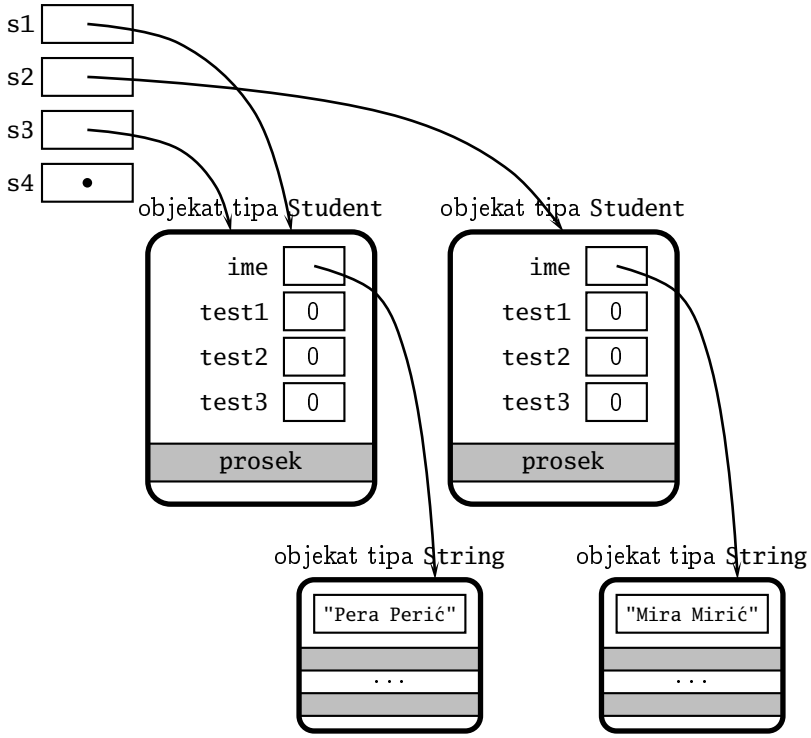
Stanje memorije posle izvršavanja svih ovih naredbi je prikazano na slici 6.1. Na toj slici su promenljive prikazana u obliku malih pravougaonika, a objekti u obliku većih pravougaonika sa zaobljenim uglovima. Ako promenljiva sadrži referencu na neki objekat, vrednost te promenljive je prikazana u obliku strelice koja pokazuje na taj objekat. Ako promenljiva sadrži referencu null, ona ne pokazuje ni na jedan objekat, pa je vrednost te promenljive prikazana kao podebljana tačka.

Sa slike 6.1 vidimo da promenljive s1 i s3 ukazuju na isti objekat, jer se naredbom dodele s3 = s1; prepisuje referenca iz s1 u s3. Obratite pažnju na to da ovo važi u opštem slučaju: kada se jedna promenljiva klasnog tipa dodeljuje drugoj, onda se kopira samo referenca, ali ne i objekat na koga ta referenca ukazuje. To znači da posle izvršavanja naredbi u listingu 6.1 imamo da je vrednost polja s1.ime jednaka "Pera Perić", ali i da je vrednost polja s3.ime jednaka "Pera Perić".

Jednakost ili nejednokost promenljivih klasnog tipa se može proveravati relacijskim operatorima == i != u Javi, ali pri tome treba biti obazriv. Ako napišemo, 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 samo ispituje da li 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 nam treba, ali ako želimo da ispitamo jednakost objekata na koje promenljive ukazuju,



SLIKA 6.1: Stanje memorija posle izvršavanja programskog fragmenta prikazanog u listingu 6.1.

moramo pojedninačno ispitati jednakost svih polja tih objekata. Drugim rečima, 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
```

U odeljku 3.4 smo govorili o tome da su stringovi zapravo objekti klase `String`, a ne primitivne vrednosti. Zato smo stringove `"Pera Perić"` i `"Mira Mirić"` prikazali na slici 6.1 kao objekte. Neka promenljiva tipa `String` može dakle sadržati samo referencu na string (kao i referencu `null`), a ne i sâm string. To objašnjava zašto su na slici 6.1 vrednosti dva primerka polja `ime` prikazane strelicama koje pokazuju na odgovarajuće string-objekte. 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".

Na kraju, spomenimo 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 samo ukazuju na njih.)

Prvo, pretpostavimo da je promenljiva klasnog tipa deklarisana sa modifikatorom final. To znači da se njena vrednost nakon inicijalizacije više ne može promeniti, kao što smo objasnili u odeljku 5.6. Ta početna, nepromenljiva vrednost final promenljive klasnog tipa je referenca na neki objekat, što znači da će ona ukazivati na njega za sve vreme izvršavanja programa. Međutim, ovo nas ne sprečava da promenimo vrednosti polja koje se nalaze u objektu. Drugim rečima, promenljiva je konstantna, a ne objekat na koga ona ukazuje. Ispravno je zato pisati, na primer:

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

Drugo, pretpostavimo da se promenljiva klasnog tipa x prenosi kao argument prilikom poziva nekog metoda. Onda se, kao što znamo, vrednost promenljive x dodeljuje odgovarajućem parametru metoda i metod se izvršava. Vrednost argumenta x se ne može promeniti u metodu, ali pošto dodeljena vrednost parametru predstavlja referencu na neki objekat, u metodu se mogu promeniti podaci u tom objektu. Posle završetka metoda, promenljiva x će i dalje ukazivati na isti objekat, ali podaci u tom objektu mogi biti promenjeni! Na primer, za metod:

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

izvršavanjem ovog programskog fragmenta:

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

na ekranu će se dobiti ime "Mika Mikić", jer je izvršavanje naredbe poziva promeni(x) ekvivalentno izvršavanju dve naredbe dodele:

```
s = x;
s.ime = "Mika Mikić";
```

odnosno polje ime objekta na koga ukazuje s, a time i x, će dobiti novu vrednost.

6.3 Konstrukcija i inicijalizacija objekata

Klasni tipovi u Javi su vrlo različiti od primitivnih tipova. Deklarisanjem neke promenljive klasnog tipa se alocira samo ta promenljiva, ali ne automatski i neki objekat odgovarajuće klase. Objekti se u programu moraju eksplicitno *konstruisati*. Postupak konstruisanja objekta se sastoji od nalaženja dovoljno mesta u hip memoriji u koju može da stane objekat i inicijalizacije njegovih polja podrazumevanim vrednostima.

Programer ne može da utiče na prvi korak pronalaženja dovoljno memorije 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 itd.), dodele vrednosti false logičkim poljima, dodele Unicode znaka '\u0000' znakovnim poljima i još, na kraju, dodele vrednosti null poljima klasnog tipa.

Ukoliko podrazumevana inicijalizacija nije odgovarajuća, poljima se mogu dodeliti početne vrednosti u njihovim deklaracijama, baš kao što se to može uraditi za bilo koje druge promenljive. Posmatrajmo klasu KockaZaIgru čiji objekat može predstavljati jednu kocku za bacanje u društvenim igrama. Ova klasa sadrži jedno objektno polje čiji sadržaj odgovara broju palom pri bacanju kocke i jedan objektni metod kojim se simulira slučajno bacanje kocke:

```
public class KockaZaIgru {

   public int broj = 6;  // broj koji je pao

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

U ovom slučaju polje broj dobija početnu vrednost 6 svaki put kada se konstruiše 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 kada želimo da broj koji je pao na početku bude slučajan:

```
public class KockaZaIgru {

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

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

U ovom slučaju se polje broj inicijalizuje slučajnom vrednošću. Kako se inicijalizacija obavlja za svaki novokonstruisani objekat, različiti objekti ove klase KockaZaIgru će možda imati različite početne vrednosti svojih primeraka polja broj.

Inicijalizacija (statičkih) polja neke klase je naravno drugačija priča, jer oni nisu vezani za pojedinačne objekte nego za klasu kao celinu. Kako postoji samo jedan primerak klasnog polja, ono se inicijalizuje samo jednom i to kada se klasa po prvi put učitava od strane JVM.

Posvetimo sada više pažnje detaljima konstrukcije objekata u Javi. Do sada smo više puta pominjali da se konstruisanje objekata obavlja operatorom new. Na primer, u programu koji koristi klasu KockaZaIgru možemo pisati:

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

ili čak kraće:

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

U ovim primerima se izrazom new KockaZaIgru() konstruiše novi objekat klase KockaZaIgru, inicijalizuju se njegova polja i vraća se referenca na njega 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.

Deo KockaZaIgru() iza operatora new podseća na poziv metoda, što zapravo to i jeste. Naime, to je poziv specijalnog metoda koji se naziva ***konstruktor***. 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 konstruktor. Taj podrazumevani konstruktor ima samo formalnu funkciju i praktično ne radi ništa. Ali kako se konstruktor poziva odmah nakon konstruisanja objekta i inicijalizuje njegovih polja podrazumevanim vrednostima, programer može definisati jedan ili više svojih konstruktora u klasi radi dodatne inicijalizacije objekta.

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

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

S druge strane, konstruktor sadrži uobičajeno telo metoda u formi blok naredbe unutar koje se mogu pisati bilo koje naredbe. Isto tako, konstruktor može imati parametre kojima se mogu preneti vrednosti za inicijalizaciju objekta prilikom njegove konstrukcije.

Sledi primer klase KockaZaIgru sa konstruktorom koji ima parametar za početnu vrednost broja koji pokazuje kocka:

```
public class KockaZaIgru {

   public int broj;              // broj koji je pao

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

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

U ovom primeru, 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(1);
```

Ovom deklaracijom se konstruiše novi objekat klase KockaZaIgru, poziva se konstruktor te klase kojim se polje broj novokonstruisanog objekta inicijalizuje vrednošću argumenta 1 i, na kraju, referenca na taj objekat se dodeljuje promenljivoj kocka.

Primetimo da se podrazumevani konstruktor dodaje klasi samo ukoliko nije eksplicitno definisan nijedan konstruktor u klasi od strane programera. Zato, na primer, ne bismo mogli više da konstruišemo objekat prethodne klase KockaZaIgru izrazom new KockaZaIgru(). Ali to i nije veliki problem, jer se konstruktori mogu preopterećivati kao obični metodi. To znači da klasa može imati više konstruktora pod uslovom, naravno, da su njihovi potpisi različiti. Na primer, klasi KockaZaIgru možemo dodati konstruktor bez parametara, koji polju broj konstruisanog objekta početno dodeljuje slučajan broj.

```
public class KockaZaIgru {

   public int broj;              // broj koji je pao

   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)(Math.random()*6) + 1;
   }
}
```

Sada sa ovom definicijom klase KockaZaIgru možemo njene objekte konstruisati na dva načina: new KockaZaIgru() ili new KockaZaIgru(x), gde je x izraz tipa int.

Na osnovu svega do sada rečenog, možemo 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*)

gde *lista-argumenata* može biti prazna. Rezultat tog izraza je referenca na konstruisani objekat, koja se najčešće 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 drugog većeg izraza. Zbog toga je važno razumeti tačan postupak izračunavanja tog izraza, jer je njegov efekat zapravo posledica izvršavanja redom ova četiri koraka:

1. Pronalazi se dovoljno veliki blok slobodne hip memorije za objekat navedene klase koji se konstruiše.
2. Inicijalizuju se 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 (ako početna vrednost polja nije zadata u njegovoj deklaraciji).
3. Poziva se konstruktor 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 izraza.

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

Primer: broj bacanja dve kocke dok se ne pokažu isti brojevi

Čitaoci u ovom trenutku možda mogu naslutiti jednu od prednosti objektno orijentisanog programiranja koja se naziva ***višekratna upotreba***.

Naime, prethodna klasa KockaZaIgru predstavlja fizičke objekte kocki za igranje i obuhvata sve njihove relevantne atribute i mogućnosti. Zato ovu klasu možemo ponovno koristi u svakom programu u kojem se radi nešto sa kockama za igranje. To je velika prednost, pogotovo za komplikovane klase, jer ne moramo uopšte brinuti o testiranju gotove klase ili o tome kako je ona realizovana.

Da bismo ovo ilustrovali, sada ćemo napisati program koji koristi klasu KockaZaIgru za brojanje koliko puta treba baciti dve kocke pre nego što se dobije isti broj na njima.

LISTING 6.2: Broj bacanja dve kocke dok se ne pokažu isti brojevi.

```
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.");
   }
}
```

Obratite pažnju na to da se ovaj program sastoji od dve datoteke:

- BacanjaDveKocke.java

- KockaZaIgru.java

U tim datotekama se redom nalaze klase BacanjaDveKocke i KockaZaIgru. Prema tome, za prevođenje i izvršavanje ovog programa treba primeniti postupak koji smo objasnili u odeljku 2.4.

6.4 Uklanjanje objekata

Novi objekat se konstruiše operatorom new i 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 (doduše tako nešto ne bismo napisali u pravom programu):

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

Prema tome, u programu se pozivom metoda novi() konstruiše objekat klase Student i referenca na njega se dodeljuje lokalnoj promenljivoj s. Nakon izvršavanja poziva metoda novi(), lokalna promenljiva s se dealocira tako da više ne postoji referenca na objekat konstruisan u metodu novi(). Prema tome, više nema načina da se tom objektu pristupi u programu, pa se objekat može ukloniti i memorija koju zauzima osloboditi za druge namene.

I sâm programer može ukazati da neki objekat nije više potreban u programu. Na primer:

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

U sredini ovog fragmenta se promenljivoj s eksplicitno dodeljuje referenca null, posle konstruisanja i korišćenja objekta klase Student preko te promenljive. Tada se gubi referenca na prethodno konstruisani objekat klase Student, 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 „đubre". U Javi se koristi posebna procedura, takozvano ***sakupljanje otpadaka*** (engl. *garbage collection*), koja s vremena na vreme „čisti đubre", odnosno oslobađa memoriju onih objekata za koje nađe da je broj referenci na njih u programu jednak nuli.

U prethodnim primerima je bilo vrlo lako zaključiti kada se objekat klase Student može ukloniti. U složenim programima je to obično mnogo teže. Ako je neki objekat bio korišćen u programu izvesno vreme, moguće je da više promenljvih sadrže referencu na njega. Taj objekat nije „đubre" 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 vezane za ovaj aspekt racionalnog upravljanja memorijom.

Kao što smo pomenuli, u nekim jezicima je odgovornost na programeru da briše nepotrebne objekte. Nažalost, voditi računa o tome u složenijim programima je 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 grešaka se javlja kada programer zanemari da ukloni nepotrebne objekte. Tada dolazi do greške ***curenja memorije*** koja se manifestuje time da program zauzima veliki deo memorije, iako je ona praktično neiskorišćena. To onda dovodi

do problema nemogućnosti izvršavanja istog ili drugih programa zbog nedostatka memorije.

6.5 Skrivanje podataka i enkapsulacija

Do sada smo malo pažnje posvećivali kontroli pristupa članovima neke klase. U primerima smo ih uglavnom deklarisali sa modifikatorom public kako bi bili dostupni iz bilo koje druge klase. Međutim, važno objektno orijentisano načelo pod nazivom ***enkapsulacija*** (ili ***učaurivanje***) nalaže da sva polja klase budu skrivena i deklarisana sa 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 svi podaci (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 ovu diskusiju potkrepili primerom, posmatrajmo klasu koja se može koristiti za obračun plata radnika:

```
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 smo sva objektna polja klase Radnik definisali da budu privatna i time postigli da se ona izvan te klase ne mogu direktno koristiti. Tako u drugoj klasi više ne možemo 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 moraju koristiti na neki način, pa smo zato obezbedili javne metode sa imenima koja počinju rečima *get* i *set*. Ovi metodi su deo javnog interfejsa klase i zato u drugoj klasi možemo 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 imamo bolje srpske izraze, konvencija je da se ime geter-metoda za neku promenljivu pravi dodavanjem reči „get" ispred kapitalizovanog imena te promenljive. 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*. Tako, da u klasi `Radnik` imamo polje `vredan` tipa `boolean`, njegov geter-metod mogli bismo zvati `isVredan()` umesto `getVredan()`. Konvencija za ime seter-metoda za neku promenljivu je da se ono pravi dodavanjam reči „set" ispred kapitalizovanog imena te promenljive. Zato za promenljivu `staž` imamo metod `setStaž`.

Ovo mešanje engleskih reči „get", „set" i „is" sa moguće srpskim imenima promenljivih dodatno doprinosi rogobatnosti tehnike skrivanja podataka. Međutim, neki aspekti naprednog Java programiranja potpuno se zasnivaju na prethodnoj konvenciji za imena getera i setera. Na primer, u koncepciji programskih komponenti JavaBeans se 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 prilagođavanje klase naprednim tehnikama.

Da li smo nešto dobili time što smo polja `ime`, `jmbg`, `staž` i `plata` učinili privatnim i morali da klasi dodamo nekoliko naizgled nepotrebnih metoda? Zar nije jednostavnije da smo sva ova polja samo deklarisali da budu javna i tako izbegli 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, tako da možemo biti sigurni da neki deo programa izvan klase `Radnik` neće (slučajno ili namerno) ovim poljima dodeliti neku nekonzistentnu 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

Drugu mogućnost koju možemo iskoristiti je to što geteri i seteri nisu ograničeni samo na čitanje i upisivanje vrednosti odgovarajućih polja. Na primer, u geter-metodu se može brojati koliko puta se pristupa 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 možemo promeniti internu implementaciju neke klase bez posledica na programe koji koriste tu klasu. Na primer, ako predstavljanje punog imena radnika moramo da razdvojimo u dva posebna dela za ime i prezime, tada je dovoljno klasi Radnik dodati još jedno polje:

```
private String prezime;
```

i promeniti geter-metod getIme():

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

Ove promene su potpuno nevidljive za programe koje 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 konvertovanja između stare i nove reprezentacije podataka. Ali poenta je da ostali programi koji koriste novu verziju klase ne moraju uopšte da se menjaju.

6.6 Službena reč `this`

Podsetimo se da se objektni metodi primenjuju na objekte i da u svom radu koriste konkretne vrednosti njihovih polja. Tako, recimo, objektni metod `povećajPlatu()` klase `Radnik`:

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

dodeljuje novu vrednost polju `plata` onog objekta za koji se ovaj metod pozove. Efekat poziva, na primer:

```
agent007.povećajPlatu(10);
```

sastoji se u povećanju vrednosti polja `plata` objekta na koga ukazuje promenljiva `agent007` za 10%. Drugim rečima, taj efekat je ekvivalnetan izvršavanju naredbe dodele:

```
agent007.plata += agent007.plata * 10 / 100;
```

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.

Kao što znamo, parametri koji odgovaraju eksplicitnim argumentima se 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. On se u Javi označava službenom rečju `this`. U ovom kontekstu 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, na primer, za metod `povećajPlatu()` možemo pisati:

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

Neki programeri čak stalno 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 je to što programeri ne moraju da smišljaju dobra imena za parametre konstruktora da bi se jasno videlo šta svaki od njih znači. Primetimo da se u telu konstruktora moraju pisati puna imena polja sa this, jer nema smisla pisati, recimo, ime = ime;. U stvari, u tom slučaju, ime se odnosi na parametar, pa bi se njegova vrednost opet dodelila njemu samom, a objektno polje ime bi ostalo netaknuto.

U prethodnim primerima nije bilo 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. Međutim, u nekim slučajevima je promenljiva this neophodna i bez nje se ne može dobiti željena funkcionalnost.

Pretpostavimo da je klasi Radnik potrebno dodati objektni metod kojim se upoređuju dva objekta radnika na osnovu njihovih plata. Tačnije, treba napisati metod većeOd() tako da se pozivom, na primer:

```
pera.većeOd(žika)
```

dobija objekat, na koga ukazuje promenljiva pera ili žika, koji ima veću platu. Ovaj metod mora koristiti promenljivu this, jer njegov rezultat može biti implicitni parametar metoda:

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

Obratite pažnju na to da u pozivu this.getPlata() možemo izostaviti promenljivu this, jer se bez nje poziv metoda getPlata() ionako odnosi na implicitni parametar metoda većeOd(). S druge strane, promenljiva

`this` u naredbi `return this;` jeste obavezna, jer je rezultat metoda `većeOd()` baš implicitni parametar ovog metoda.

Službena reč `this` ima još jedno, potpuno drugačije značenje od prethodnog. Kao što znamo, konstruktor klase je 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 želimo da klasi `Radnik` dodamo još jedan konstruktor kojim se konstruiše pripravnik bez radnog staža i sa fiksnom platom, to možemo 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 možemo kraće pisati:

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

Ovde je zapisom:

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

označen poziv prvobitnog konstruktora klase `Radnik` sa četiri parametra. Izvršavanje tog poziva sa navedenim argumentima je ekvivalentno baš onome što smo hteli da postignemo.

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

7

Osnovne strukture podataka

Osnovna jedinica za čuvanje podataka u programu je promenljiva. Međutim, kao što znamo, jedna promenljiva u svakom trenutku može sadržati samo jedan podatak. Ukoliko u programu želimo da više srodnih podataka imamo istovremeno na raspolaganju i da ih posmatramo kao jednu celinu, onda ih u Javi možemo organizovati u formi objekta bez metoda. Takva forma organizovanja podataka se u drugim jezicima naziva *slog*, ali to ima smisla samo za mali broj podataka. 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 organizovana na ovakav način, onda se govori o ***strukturi podataka***.

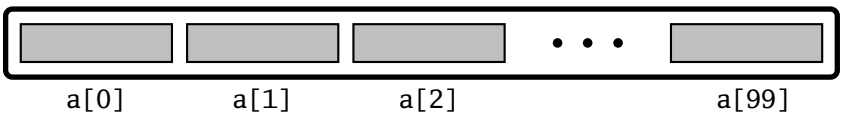
U ovom poglavlju ćemo govoriti o osnovnim strukturama podataka koje su raspoložive u Javi. Tu spadaju nizovi i dinamički nizovi.

7.1 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. U Javi, numeracija elemenata niza počinje od nule. To znači da indeks nekog elementa niza može biti između nule i dužine niza manje jedan. 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.

Konceptualno, niz elemenata (promenljivih) kao celina se u programu ukazuje promenljivom specijalnog, nizovnog tipa. Za označavanje svakog elementa niza se koristi zapis koji se sastoji od imena te promenljive i indeksa odgovarajućeg elementa u uglastim (srednjim) zagradama. Na primer, niz dužine 100 na koga ukazuje promenljiva a se sastoji od 100 promenljivih istog tipa čiji je izgled prikazan na slici 7.1.



SLIKA 7.1: 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 7.1 deklarisan da bude `int`, onda je svaka od promenljivih `a[0]`, `a[1]`, ..., `a[99]` obična celobrojna promenljiva koja se u programu može koristiti svuda gde su dozvoljene celobrojne promenljive.

Ova konceptualna slika nizova je u Javi realizovana na objektno orijentisan način: nizovi u Javi se 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 u tome što, mada kao objekti pripadaju nekoj klasi, njihova klasa ne mora da se definiše u programu. Naime, svakom postojećem tipu (klasi) `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 objekti su nizovi baznog tipa `int`. Ili, ako je u programu definisana klasa `Student`, onda je automatski raspoloživa i klasa `Student[]` kojoj pripadaju svi nizovi baznog tipa `Student`.

Za zapis `T[]` se koriste kraći termini „niz baznog tipa T" ili „niz tipa T". Primetimo da su strogo govoreći ti termini neprecizni, jer se nizom označava i klasa i objekat. Nadamo se ipak da, posle početnog upoznavanja, ova dvosmislenost neće izazvati konfuziju kod čitalaca.

Postojanje klasnog tipa, recimo, `int[]` omogućava nam da deklarišemo neku promenljivu tog klasnog tipa. Na primer:

```
int[] a;
```

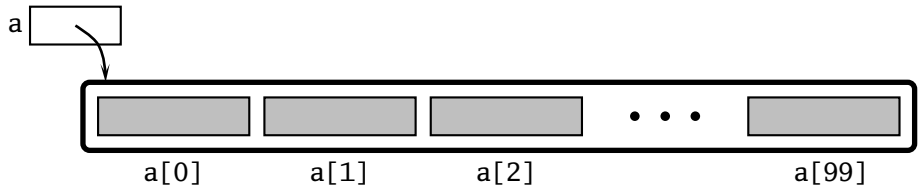
Kao i svaka promenljiva klasnog tipa, promenljiva a može sadržati referencu na neki objekat klase `int[]`. A kao što smo upravo objasnili, objekti klase `int[]` su nizovi baznog tipa `int`. Objekat niza tipa `int[]` se konstruiše operatorom new, doduše u posebnom obliku, a referenca na taj novi niz se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

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

gde vrednost 100 u uglastim zagradama određuje dužinu konstruisanog niza. Prethodna dva koraka se, kao što je to uobičajeno, mogu skratiti u jedan:

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

Ovom deklaracijom se dakle alocira promenljiva a klasnog tipa `int[]`, konstruiše objekat niza od 100 elemenata primitivnog tipa `int` i referenca na novokonstruisani objekat niza dodeljuje promenljivoj a. Svaki od 100 elemenata ovog niza je obična promenljiva tipa `int` čiji sadržaj može biti neka celobrojnu vrednost. Efekat prethodne deklaracije je slikovito prikazan na slici 7.2.



SLIKA 7.2: Konstruisanje niza a od 100 elemenata.

Obratite pažnju na to da se često za promenljivu koja ukazuje na neki niz kraće govori kao da sadrži sâm taj niz. Tako u poslednjem primeru kraće kažemo „niz a od 100 elemenata". Međutim, promenljiva a je obična promenljiva klasnog tipa i njena vrednost može biti samo referenca na neki objekat niza, kao i referenca `null`.

Deklaracija niza čiji bazni tip predstavlja neki klasni tip nije mnogo drugačija. Na primer:

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

Ovom deklaracijom se alocira promenljiva s klasnog tipa Student[], konstruiše niz od 15 elemenata klasnog tipa Student i referenca na novi niz dodeljuje promenljivoj s. Svaki od 15 elemenata ovog niza predstavlja promenljivu klasnog tipa Student čiji sadržaj može biti referenca na objekat klase Student ili null.

U opštem slučaju, ako je N celobrojni izraz, onda se izrazom:

new *bazni-tip*[N]

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

Nakon konstruisanja nekog niza, broj elementa tog niza se ne može promeniti. Prema tome, iako možemo pisati, na primer:

```
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.

Svaki objekat niza automatski sadrži javno celobrojno polje length u kojem se nalazi dužina niza. Zato, u prethodnom primeru, program može koristiti polje x.length čija je vrednost 20. Vrednost tog polja se naravno ne može menjati u programu. (To je obezbeđeno time što je polje length svakog niza deklarisano da bude final polje, odnosno ono koje se ne može menjati nakon inicijalizacije.)

Pošto elementi niza predstavljaju u suštini 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 deklaraciji niza, unutar vitičastih zagrada i međusobno razdvojene zapetama. Deklaracijom, na primer:

```
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, 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.

Kada se na ovaj način zadaju početne vrednosti, dužina niza se ne navodi u deklaraciji niza i implicitno se zaključuje 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)
            };
```

Opšti oblik operatora new u ovom kontekstu je:[1]

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

Rezultat ovog 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 jedini parametar jeste niz stringova, onda u programu možemo pisati:

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

Ovaj primer pokazuje da je konstruisanje „anonimnih" nizova ponekad vrlo korisno. Inače bismo u programu morali da za neke pomoćne nizove deklarišemo i imenujemo promenljive koje ukazuju na njih.

Kao što smo već pomenuli, elementi niza tipa *bazni-tip*[] su obične promenljive baznog tipa niza i mogu se koristiti u programu na svim mestima gde je dozvoljena upotreba promenljive baznog tipa. Takođe znamo da se elementi niza koriste u programu preko svojih indeksa i imena promenljive koja ukazuje na objekat niza. Na primer, deklaracijom:

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

[1] U stvari, deo **new** *bazni-tip*[] nije obavezan u naredbi deklaracije niza sa početnim vrednostima.

dobijamo zapravo 100 celobrojnih promenljivih a[0], a[1], ..., a[99].

Međutim, puna snaga nizova ne leži samo u mogućnosti dobijanja velikog broja promenljivih za čuvanje velikog broja podataka. Druga odlika nizova koja ih izdvaja među strukturama podataka je ta što se za indekse elemenata nizova u programu mogu koristiti proizvoljni celobrojni izrazi. Ako je i promenljiva tipa int, onda su, na primer, a[i] i a[3*i-7] takođe ispravni zapisi elemenata niza a iz prethodnog primera. Prilikom izvršavanja programa, zavisno od konkretne vrednosti promenljive i, element niza na koji se odnosi zapis, recimo, a[3*i-7] dobija je izračunavanjem vrednosti celobrojnog izraza u uglastim zagradama. Tako, 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 petlja kojom se na ekranu prikazuju vrednosti svih elemenata niza a:

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

U ovoj petlji, početna vrednost brojača i je 0, pa se u prvoj iteraciji prikazuje element a[0]. Zatim se izrazom i++ brojač i uvećava i dobija 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 99 i tada se prikazuje element a[99]. Nakon toga će i dobiti vrednost 100 i zato uslov nastavka petlje i < a.length neće biti tačan pošto a.length ima vrednost 100. Time se prekida izvršavanje petlje i završava prikazivanje vrednosti tačno svih elemenata niza a.

U radu sa elementima nekog niza u Javi su moguće dve vrste grešaka koje izazivaju prekid izvršavanja programa. Prvo, pretpostavimo da promenljiva a tipa niza sadrži vrednost null. Tada 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 izraza i u a[i] može biti van dozvoljenih granica indeksa niza. To će biti slučaj kada se izračunavanjem izraza i dobije i<0 ili i>=a.length.

Primer: prebrojavanje glasova na izborima

Pretpostavimo da na jednom biračkom mestu treba prebrojati glasove sa glasačkih listića posle završetka glasanja. Naravno, umesto ručnog brojanja koje je podložno greškama, želimo da napišemo Java program kojim će se glasovi unositi kako se redom čitaju listići. Pošto program pišemo mnogo pre raspisivanja izbora radi njegovog potpunog testiranja, ne znamo unapred broj partija koji je izašao na izbore, već taj podataka treba da bude deo ulaza programa.

Ako su partije u programu numerisane od 0, onda ćemo za sabiranje njihovih glasova koristiti niz čiji i-ti element sadrži broj glasova i-te partije. Prestanak unošenja glasova se programski realizuje unosom glasa za nepostojeću partiju. Nakon toga program prikazuje ukupan broj glasova svih partija.

LISTING 7.1: Prebrojavanje glasova na izborima.

```
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 partija za sabiranje glasova
      int[] partija = new int[brojPartija];

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

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

Primetimo da smo indekse niza `partija` u programu morali da prilagodimo za 1, pošto su partije na glasačkim listićima numerisane od 1 a nizovi u Javi od 0.

Primer: kopiranje nizova

Neka je deklarisan neki niz `x`, na primer:

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

i neka promenljiva `y` tipa istog tog niza, na primer:

```
double[] y;
```

Kao što znamo, kako su `x` i `y` obične promenljive klasnog tipa, to se naredbom dodele:

```
y = x;
```

u promenljivu `y` kopira samo referenca iz promenljive `x`, a ne i sâm objekat niza. To znači da na objekat niza sada ukazuju dve promenljive `x` i `y`, ali i dalje postoji samo po jedan primerak svakog elementa niza. Zbog toga se ti elementi sada mogu koristiti na dva načina: zapisi `x[i]` i `y[i]` se odnose na jedinstveni i-ti element niza. Ako je potrebno fizički kopirati svaki element niza, to se mora uraditi ručno. (Ili upotrebom posebnog metoda Javine klase `Arrays`). Da bismo ovo ilustrovali na malo interesantniji način, u nastavku ćemo napisati poseban metod kojim se ovaj problem rešava za realne nizove.

Bilo koji tip niza u Javi je tip kao i svaki drugi, pa se može koristiti na sve načine na koje se mogu koristiti drugi tipovi u Javi. Specifično, tip niza može biti tip parametra nekog metoda, kao i tip njegovog rezultata. Upravo ovu činjenicu možemo iskoristiti za pisanje metoda koji fizički kopira sve elemente 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 < original.length; i++)
      kopija[i] = original[i];
   return kopija;
}
```

Ako su x i y promenljive koje su definisane kao na početku ovog primera, onda se naredbom dodele:

```
y = kopirajNiz(x);
```

dobija novi niz na koga ukazuje y. Elementi novog niza su fizičke kopije elementa niza na koga ukazuje x tako da se sada zapisi x[i] i y[i] odnose na različite i-te elemente odgovarajućih nizova.

Primer: argumenti programa u komandnom redu

Konačno možemo potpuno razumeti metod main() koji smo koristili u svim programima do sada. Zaglavlje tog metoda ima oblik:

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

Ako je metod main() definisan u nekoj klasi A, onda Java interpretator (JVM) implicitno poziva taj javni metod klase A kada u komandnom redu navedemo:

```
java A
```

Sada možemo uočiti da metod main() ima jedan parametar args tipa String[].[2] To ukazuje da se metodu main() prilikom poziva može kao argument preneti niz stringova. Ali pošto se metod main() poziva implicitno, kako programer može navesti neki niz stringova koje treba preneti glavnom metodu? Ovo se postiže pisanjem željenog niza stringova u komandnom redu iza imena Java interpretatora i klase čiji se glavni metod izvršava. Metod main() dobija dakle svoj argument iz komandnog reda tako što JVM automatski inicijalizuje niz args stringovima koji su eventualno navedeni iza, recimo, java A u komandnom redu.

Razmotrimo na primer sledeći program:

[2] Ime parametra args metoda main() je proizvoljno i može se promeniti.

LISTING 7.2: Argumenti metoda main().

```
public class Poruka {

   public static void main(String[] args) {

      if (args[0].equals("-d"))
         System.out.print("Dobar dan");
      else if (args[0].equals("-z"))
         System.out.print("Zbogom");

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

Ako ovaj program izvršimo 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";
```

Prema tome, izvršavanjem prethodnog programa se na ekranu prikazuje ova poruka:

```
Zbogom okrutni svete!
```

Ako pak program izvršimo komandom

```
java Poruka -d tugo
```

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

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

pa se na ekranu prikazuje ova poruka:

```
Dobar dan tugo!
```

7.2 Naredba for-each

U verziji Java 5.0 je dodat novi oblik for petlje, takozvana for-each petlja koja je namenjena za rad sa *svim* elementima nekog niza. U stvari, for-each petlja se može koristiti opštije za bilo koju strukturu podataka (tj. kolekciju podataka, kako se struktura podataka zvanično naziva u Javi). Ako je *niz* tipa *bazni-tip*`[]`, onda for-each petlja za *niz* ima oblik:

```
for (bazni-tip element : niz) {
   .
   . // rad sa aktuelnim elementom niza
   .
}
```

Obratite pažnju na to da je znak dve tačke deo sintakse ove petlje. Pored toga, *element* je kontrolna promenljiva baznog tipa koja se mora deklarisati unutar petlje. Prilikom izvršavanja for-each petlje, kontrolnoj promenljivoj *element* se redom dodeljuje vrednost svakog elementa niza i izvršava se telo petlje za svaku vrednost. Prema tome, prethodni oblik for-each petlje je ekvivalentan sa sledećom običnom for petljom:

```
for (int i = 0; i < niz.length; i++) {
   bazni-tip element = niz[i];
   .
   . // rad sa aktuelnim elementom niza
   .
}
```

Na primer, ako je u programu konstruisan niz a tipa `int[]`, onda se svi elementi tog niza mogu for-each petljom lako prikazati na ekranu:

```
for (int e : a) {
   System.out.println( e );
}
```

Ili, ako želimo da saberemo sve pozitivne elemente niza a, to možemo uraditi na sledeći način:

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

Primetimo da je for-each petlja korisna u slučajevima kada je potrebno uraditi nešto sa svim elementima nekog niza, jer ne moramo da vodimo računa o indeksima i granici elemenata. 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 eventualno nešto radi sa njima), dok upisivanje vrednosti u elemente niza nije moguće. Na primer, ako bismo želeli da svim elementima prethodno konstruisanog celobrojnog niza a dodelimo vrednost recimo 17, bilo bi pogrešno napisati:

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

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

7.3 Metodi sa promenljivim brojem argumenata

Od verzije Java 5.0 je omogućeno pozivanje metoda sa promenljivim brojem argumenata. Metod `System.out.printf()` za formatizovano prikazivanje vrednosti na ekranu, o čemu smo govorili u odeljku 3.4, jeste primer metoda sa promenljivim brojem argumenata. Naime, prvi argument metoda `System.out.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. To je najbolje objasniti na jednom primeru, pa pretpostavimo da želimo da napišemo metod prosek koji izračunava i vraća prosek bilo kog broja vrednosti tipa double. Prema tome, poziv tog metoda može biti `prosek(1, 2, 3, 4)`, `prosek(1.41, Math.PI, 2.3)`, `prosek(Math.sqrt(3))` ili čak `prosek()`. U pozivu prvog metoda imamo dakle četiri argumenta, u drugom pozivu tri argumenta, u trećem poziva jedan argument, a u poslednjem pozivu nula argumenata.

Definicija metoda prosek mora pretrpeti male izmene u zaglavlju, na primer:

```
public static double prosek (double... broj)
```

Tri tačke iza imena tipa double parametra broj u zagradama ukazuju da se umesto tog parametra može navesti promenljiv broj argumenata u pozivu metoda. A kada se metod pozove, pridruživanje više argumenata parametru broj se razrešava tako što se implicitno najpre konstruiše niz broj tipa double[] čija je dužina jednaka broju datih argumenata, a zatim se elementi tog niza inicijalizuju datim argumentima. Telo metoda prosek moramo dakle pisati pod pretpostavkom da za aktuelni poziv tog metoda imamo konstruisan niz broj tipa double[], čija se dužina nalazi u polju broj.length i vrednosti stvarnih argumenata u elementima broj[0], broj[1] i tako dalje.

Imajući ovo u vidu, kompletna definicija metoda prosek je:

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

Još bolje, ako iskoristimo for-each petlju, dobijamo elegantniju verziju:

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

Ovaj postupak pridruživanja parametra nekog metoda promenljivom broju argumenata 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 kome 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 u pozivu, na mesto parametra kome odgovara promenljiv broj argumenata, možemo navesti stvarni niz, a ne listu pojedinačnih vrednosti. Tako, na primer, ako je u programu prethodno konstruisan niz ocene tipa `double[]`, onda je ispravno pozvati `prosek(ocene)` radi dobijanja proseka vrednosti ocena u nizu.

Klasa **Arrays**

U radu sa nizovima se može pokazati pogodnom 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. Sledi nepotpun spisak i opis ovih metoda.

- `String toString(`*`tip`*`[] a)` — vraća reprezentaciju niza `a` u formi stringa. Pri tome se svi elementi niza `a` nalaze unutar uglastih zagrada po redu njihovih pozicija u nizu i međusobno su razdvojeni zarezima.
- *`tip`*`[] copyOf(`*`tip`*`[] a, int d)` — vraća novu kopiju niza `a` dužine `d`. Ako je dužina `d` veća od `a.length`, višak elemenata kopije niza se inicijalizuje nulom ili vrednošću `false`. U suprotnom slučaju, kopira se samo početnih `d` elemenata niza `a`.
- *`tip`*`[] copyOfRange(`*`tip`*`[] a, int od, int do)` — kopira niz `a` od indeksa `od` do indeksa `do` u novi niz. Element sa indeksom `od` niza `a` se uključuje, dok se onaj sa indeksom `do` ne uključuje u novi niz. Ako je indeks `do` veći od `a.length`, višak elemenata kopije niza se inicijalizuje nulom ili vrednošću `false`.
- `void sort(`*`tip`*`[] a)` — sortira niz `a` u mestu u rastućem redosledu.
- `int binarnySearch(`*`tip`*`[] a, `*`tip`*` v)` — koristi binarnu pretragu za nalaženje vrednosti `v` u sortiranom nizu `a`. Ako je data 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 svim elementima niza `a` istu vrednost `v`.

- boolean equals(*tip*[] a, *tip*[] b) — vraća vrednost true ukoliko nizovi a i b imaju istu dužinu i jednake odgovarajuće elemente. U suprotnom slučaju, vraća vrednost false.

U svim prethodnim metodima je *tip* jedan od primitivnih tipova, osim u slučaju metoda sort i binarnySearch kod kojih jedino nije dozvoljeno da to bude boolean.

Primer: izvlačenje loto brojeva

Korišćenje klase Arrays ćemo ilustrovati programom koji simulira izvlačenje brojeva u igri na sreću loto. U toj igri se izvlači 7 slučajnih, međusobno različitih brojeva među celim brojevima od 1 do 39. Program će generisati takvih 7 brojeva koji predstavljaju jedno kolo igre loto. (Drugi ugao gledanja na rezultat programa je da generisane brojeve igrač može igrati u stvarnoj igri da bi osvojio glavnu nagradu.)

U programu se koriste konstante n i k za vrednosti dužina nizova svih mogućih brojeva i izvučene kombinacije. Celobrojni niz broj sadrži brojeve od 1 do n koji učestvuju u jednom kolu igre loto.

Ovaj niz je 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 broj.

Za izvlačenje slučajnih brojeva koristimo metod Math.random(), ali problem je to što time ne moramo obavezno dobiti različite izvučene brojeve lotoa. Zbog toga ćemo, u stvari, generisati slučajni indeks levog dela niza broj i uzeti broj koji se nalazi na tom indeksu. Zatim ćemo taj element međusobno zameniti sa poslednjim elementom levog dela niza broj i pomeriti ulevo granicu između dva dela niza smanjivanjem vrednosti m za jedan. Ponavljanjem ovog postupka k puta za svaki izvučeni broj, na kraju ćemo u desnom delu niza broj imati sve izvučene slučajne brojeve.

Najzad, da bismo izvučene brojeve prikazali u rastućem redosledu, desni deo niza broj, od indeksa n-k do kraja, kopiramo u novi niz kombinacija koji zatim sortiramo korišćenjem metoda klase Arrays.

LISTING 7.3: Izvlačenje loto brojeva.

```
import java.util.*;

public class Loto {

   public static void main (String[] args) {

      final int n = 39;  // dužina niza svih brojeva
      final int k = 7;   // dužina izvučene kombinacije

      // Inicijalizacija niza brojevima 1, 2, ..., n
      int[] broj = new int[n];
      for (int i = 0; i < n; i++)
         broj[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 b = broj[i];
         broj[i] = broj[m];
         broj[m] = b;
      }

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

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

      // Prikazivanje izvučenih brojeva u rastućem redosledu
      System.out.println("Dobitna kombinacija je: ");
      for (int b : kombinacija)
         System.out.print(b + "  ");
      System.out.println();
```

```
    }
}
```

7.4 Višedimenzionalni nizovi

Do sada smo posmatrali jednodimenzionalne nizove koji se mogu prostirati samo u dužinu. Međutim, svaki tip se može koristiti kao bazni tip nekog niza. Specifično, pošto je neki tip niza takođe uobičajen tip u Javi, ukoliko je bazni tip nekog niza baš takav tip, dobijamo niz nizova. Na primer, neki celobrojni niz ima tip `int[]`, a to znači da automatski postoji i tip `int[][]` koji predstavlja niz nizova celih brojeva. Takvi nizovi nizova se nazivaju ***dvodimenzionalni*** nizovi. Ništa nas ne sprečava da dalje formiramo i tip `int[][][]` koji predstavlja niz dvodimenzionalnih nizova, odnosno ***trodimenzionalni*** niz. Naravno, ova linija razmišljanja se može nastaviti i tako dobijamo ***višedimenzionalne*** nizove, mada je zaista retka upotreba nizova čija je dimenzija veća od tri. U daljem tekstu ćemo se skoncentrisati samo na dvodimenzionalne nizove, jer se svi koncepti u vezi sa njima lako proširuju na više dimenzija.

Deklarisanje promenljive tipa dvodimezionalnog niza je slično deklarisanju običnog, jednodimenzionalnog niza — razlika je samo u dodatnom paru uglastih zagrada. Na primer, deklaracijom:

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

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

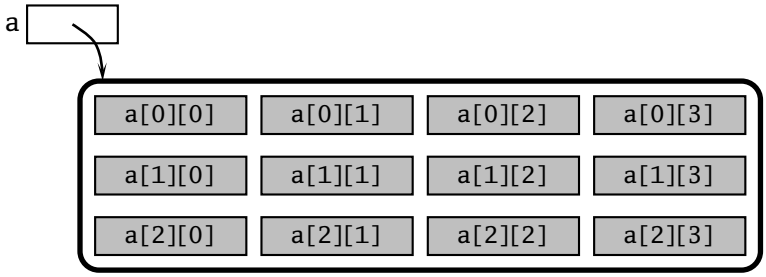
```
a = new int[3][4];
```

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

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

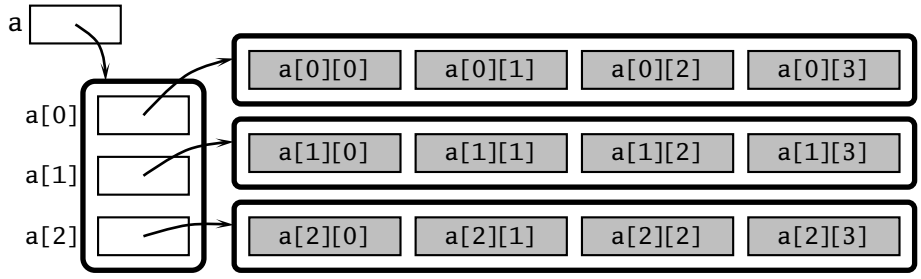
Konstruisani dvodimenzionalni niz na koga ukazuje a je najbolje zamisliti kao tabelu (ili matricu) koja ima tri reda i četiri kolone. Elementi tabele predstavljaju obične, u ovom slučaju, celobrojne promenljive koje

imaju dvostruke indekse `a[i][j]`: prvi indeks pokazuje red i drugi kolonu u kojem se element nalazi u tabeli. Pri tome treba imati u vidu da, kao što je to uobičajeno u Javi, numeracija redova i kolona tabele ide od nule, a ne od jedan. Na slici 7.3 je prikazan izgled relevantne memorije računara dobijen nakon izvršavanja prethodne naredbe dodele.



SLIKA 7.3: Dvodimenzionalni celobrojni niz a dimenzije 3×4.

U stvarnosti, konstruisani dvodimenzionalni niz na koga ukazuje promenljiva a je ipak niz celobrojnih nizova. Tako, izrazom `new int[3][4]` se zapravo konstruiše niz od tri celobrojna niza, od kojih svaki ima četiri elemenata. Prava slika je zato ona koja je prikazana na slici 7.4.



SLIKA 7.4: Prava slika dvodimenzionalnog celobrojnog niza a dimenzije 3×4.

Prava slika dvodimenzionalnog niza je komplikovanija za razumevanje i, srećom, može se u većini slučajeva zanemariti. Ponekad je ipak potrebno znati da svaki red tabele predstavlja zapravo jedan niz za sebe. Ovi nizovi u prethodnom primeru su tipa `int[]` i na njih ukazuju promenljive sa imenima `a[0]`, `a[1]` i `a[2]`. Te promenljive i nizovi se u programu mogu koristiti svuda gde su dozvoljeni obični celobrojni nizovi; na primer, kao 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 broju redova tabele a. Ako je potrebno dobiti broj kolona tabele, onda je to dužina jednodimenzionalnih nizova od kojih se sastoji svaki red tabele, odnosno `a[0].length`, `a[1].length` ili `a[2].length`. (U stvari, svi redovi tabele ne moraju biti jednake dužine i u nekim složenijim primenama se koriste tabele sa različitim dužinama redova.)

Elementi dvodimenzionalnih nizova se prilikom konstruisanja dvodimenzionalnih nizova inicijalizuju odgovarajućim podrazumevanim vrednostima. Ovo se može promeniti navođenjem 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 tabele daju po redovima, odnosno navode redom za jednodimenzionalne nizove redova tabele 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. Ovi indeksi mogu biti bilo koji celobrojni izrazi i njihovim izračunavanjem se dobijaju brojevi reda i kolone u kojima se nalazi odgovarajući element tabele.

U radu sa dvodimenzionalnim nizovima se često koriste ugnježđene for petlje tako da se u spoljašnjoj petlji prate redovi tabele, a u unutrašnjoj petlji se prate kolone tabele čime se postupa sa svim elementima aktuelnog reda. Na primer, za dvodimenzionalni niz b:

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

ukoliko želimo da elementima na glavnoj dijagonali dodelimo vrednost 1 i ostalim elementima vrednost 0, to možemo uraditi na sledeći način:

```
for (int i = 0; i < b.length; i++) {       // za svaki red
                                           //     u tabeli
   for (int j = 0; j < b[i].length; j++) { // i za svaku kolonu
                                           //  u aktuelnom redu
      if (i == j)  // da li je element na glavnoj dijagonali?
         b[i][j] = 1;
```

```
        else
            b[i][j] = 0;
    }
}
```

Na sličan način možemo sabrati sve elemente tabele b:

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

Sabiranje svih elemenata tabele b može se postići i ugnježđenim for–each petljama:

```
double zbir = 0;
for (double[] red : b)
    for (double e : red)
        zbir = zbir + e;
```

Primer: rad sa tabelarnim podacima

Dvodimenzionalni nizovi su naročito korisni za predstavljanje podataka koji su prirodno organizovani u obliku tabele po redovima i kolonama. Uzmimo primer firme ABC sa 10 prodavnica koja ima podatke o mesečnom profitu svake prodavnice tokom neke godine. Tako, ako su prodavnice numerisane od 0 do 9 i meseci godine od 0 do 11, onda se ovi podaci o profitu mogu predstaviti dvodimenzionalnim nizom `profit` na sledeći način:

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

Prema tome, redovima dvodimenzionalnog niza `profit` su obuhvaćene prodavnice firme, dok kolone tog niza ukazuju na mesece godine. Element, na primer, `profit[5][2]` označava vrednost profita koji je ostvarila prodavnica 5 u martu. Opštije, element `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-ti red dvodimenzionalnog niza `profit`, sadrži dakle vrednosti profita koji je ostvarila i-ta prodavnica tokom cele godine po mesecima.

Ako pretpostavimo da je niz profit popunjen podacima, na osnovu toga se mogu dobiti različiti analitički pokazatelji. Ukupni godišnji profit firme, na primer, može se dobiti na sledeći način:

```
public double godišnjiProfit() {

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

Često je potrebno obraditi samo pojedine redove ili pojedine kolone tabele podataka. Da bismo, recimo, dobili ukupni profit svih prodavnica u datom mesecu, treba sabrati vrednosti profita u koloni koja odgovara tom mesecu:

```
private double profitZaMesec(int m) {

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

Ovaj postupak možemo iskoristiti za prikazivanje ukupnog profita svih prodavnica po svim mesecima:

```
public void prikažiProfitPoMesecima() {

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

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

Korisno je imati i pokazatelj ukupnog profita pojedinih prodavnica za celu godinu. To prevedeno na dvodimenzionalne nizove znači da treba

formirati jednodimenzionalni niz čiji elementi predstavljaju zbir vrednosti pojedinih redova dvodimenzionalnog niza. Ovaj postupak i prikazivanje dobijenih vrednosti elemenata jednodimenzionalnog niza, uz prethodnu proveru da li je tabela profita zaista popunjena podacima, sadržani su u sledećem metodu:

```
public void prikažiProfitPoProdavnicama() {

   if (profit == null) {
      System.out.println("Greška: podaci nisu uneseni!");
      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();
   }
}
```

Sada ćemo napisati kompletan program koji korisniku nudi razne opcije za rad sa podacima o profitu imaginarne firme ABC o kojoj smo prethodno govorili. Program se sastoji od dve klase. Klasa `Firma` opisuje konkretnu firmu i sadrži polje `n` za broj njenih prodavnica, kao i polje `profit` koje ukazuje na njenu tabelu profita. Pored ovih atributa, klasa `Firma` sadrži i prethodne metode kojima se obrađuje tabela profita konkretne firme.

LISTING 7.4: Firma sa više prodavnica i tabelom profita.

```
import java.util.*;

public class Firma {

   private int n;              // broj prodavnica firme
   private double[][] profit;  // tabela 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 nisu uneseni!");
      return;
   }

   System.out.print("Tabela profita ");
   System.out.println("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();
   }
}

public double godišnjiProfit() {

   double ukupniProfit = 0;
   for (int i = 0; i < n; i++)
```

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

private double profitZaMesec(int m) {

   double mesečniProfit = 0;
   for (int i = 0; i < n; i++)
      mesečniProfit += profit[i][m];
   return mesečniProfit;
}

public void prikažiProfitPoMesecima() {

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

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

public void prikažiProfitPoProdavnicama() {

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

   double[] profitProdavnice = new double[n];
   for (int i = 0; i < n; 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 < n; i++) {
      System.out.print("Prodavnica " + i + ": ");
      System.out.printf("%7.2f", profitProdavnice[i]);
      System.out.println();
```

```
      }
   }
}
```

Druga klasa služi za izbor pojedinih opcija iz korisničkog menija radi dobijanja različitih pokazatelja o profitu firme ABC na ekranu.

LISTING 7.5: Rad sa tabelom profita firme preko menija.

```
import java.util.*;

public class ProfitFirme {

   public static void main(String[] args) {

      System.out.print("Program za rad sa tabelom profita ");
      System.out.print("koji je ostvarilo 10 prodavnica ");
      System.out.println("firme za 12 meseci.");
      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 nisu uneseni!");
               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: ");
  }
}
```

7.5 Dinamički nizovi

Dužina niza je fiksirana u trenutku konstruisanja niza i ne može se više menjati. Međutim, u mnogim primenama, broj podataka koji se čuvaju u elementima niza varira tokom izvršavanja programa. Jedan primer je

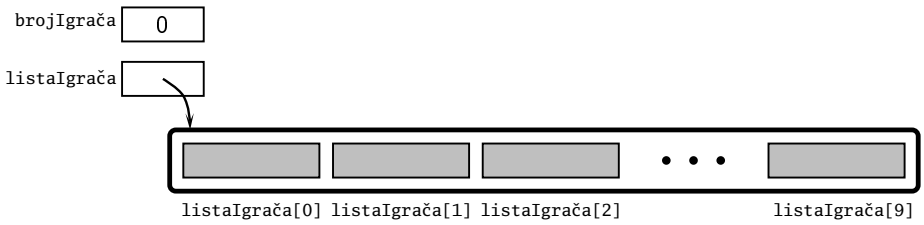
program za pisanje dokumenata u kojem možemo imati niz tipa `String[]` čiji elementi sadrže pojedinačne redove teksta dokumenta. Drugi primer je program za neku kompjutersku igru preko Interneta u kojem možemo imati niz čiji elementi sadrže igrače koji trenutno učestvuju u igri.

U ovim i sličnim primerima je karakteristično to što imamo niz (relativno velike) fiksne dužine, ali je niz popunjen samo delimično. Ovaj pristup ima nekoliko nedostatka. Manji problem je to što moramo voditi računa o tome koliko je zaista iskorišćen niz. Za taj zadatak možemo uvesti dodatnu promenljivu čija vrednost ukazuje na indeks prvog slobodnog elementa niza.

Uzmimo primer kompjuterske igru preko Interneta kojoj se igrači mogu priključiti ili je mogu napustiti u svakom trenutku. Ako imamo klasu `Igrač` koja opisuje pojedinačne igrače te igre, onda one igrače koji se trenutno igraju možemo predstaviti nizom `listaIgrača` tipa `Igrač[]`. Pošto 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[] listaIgrača = new Igrač[10];  // moguće do 10 igrača
int brojIgrača = 0;  // na početku, broj igrača je 0
```

Odgovarajući sadržaj memorije posle izvršavanja ovih deklaracija prikazan je na slici 7.5.



SLIKA 7.5: Početna slika niza `listaIgrača` i promenljive `brojIgrača` u memoriji.

Nakon što se nekoliko igrača priključi igri, njihov aktuelni broj će se nalaziti u promenljivoj `brojIgrača`. Pri tome, elementi niza:

`listaIgrača[0], listaIgrača[1], ... , listaIgrača[brojIgrača - 1]`

ukazuju na objekte konkretnih igrača koji učestvuju u igri. Obratite pažnju na to da promenljiva brojIgrača ima dvostruku ulogu — pored aktuelnog broja igrača, promenljiva brojIgrača sadrži indeks prvog „slobodnog" elementa niza listaIgrača koji je neiskorišćen. Registrovanje novog igrača noviIgrač u programu se vrši prosto dodavanjem igrača nizu listaIgrača na kraj niza:

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

Kod uklanjanja nekog igrača iz igre moramo paziti da ne ostavimo „rupu" u nizu. Pretpostavimo da želimo da uklonimo igrača čiji je indeks k u nizu listaIgrača. Ako redosled igrača nije bitan, jedan način za uklanjanje k-tog igrača je premeštanje poslednjeg igrača na k-to mesto u nizu i smanjivanje vrednosti promenljive brojIgrača za jedan:

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

Igrač koji se nalazio na k-tom mestu nije više u nizu listaIgrača. Poslednji igrač se nalazi dvaput u tom nizu, mada se zapravo računa samo njegovo novo k-to mesto u delu niza sa indeksima od 0 do brojIgrača − 1. Njegovo drugo, staro mesto u nizu je sada slobodno za upisivanje novih igrača, pošto je promenljiva brojIgrača umanjena za jedan.

Ako želimo da uklonimo k-tog igrača, ali je redosled aktuelnih igrača u nizu bitan, onda sve igrače od indeksa $k+1$ moramo 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++)
   listaIgrača[i - 1] = listaIgrača[i];
brojIgrača--;
```

Veći problem u radu sa nizovima fiksne dužine je to što moramo postaviti prilično proizvoljnu gornju granicu 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 listaIgrača. Očigledno rešenje je da taj niz konstruišemo sa mnogo većom dužinom. Ali i ta veća dužina nam 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, dužinu nizu možemo uvek deklarisati da bude vrlo velika i dovoljna za sve praktične slučajeve. Ovo rešenje ipak nije zadovoljavajuće, jer nam se može desiti da veliki deo niza bude neiskorišćen i da zato niz bespotrebno zauzima veliki memorijski prostor sprečavajući možda izvršavanje drugih programa na računaru.

Najbolje rešenje ovog problema bi bilo kada bismo mogli da niz produžimo kada nam to zatreba 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. Niz ovog objekta ne možemo produžiti, ali možemo konstruisati novi objekat dužeg niza i promeniti sadržaj promenljive koji ukazuje na objekat starog niza tako da ukazuje na objekat novog niza. Pri tome, naravno, elemente starog niza moramo 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. Pored toga, pošto nijedna promenljiva više ne ukazuje na objekat starog niza, on će biti automatski uklonjen od strane sakupljača otpadaka.

Primenimo ovaj pristup u primeru programa za kompjutersku igru. Kada se novi igrač priključuje igri i niz `listaIgrača` početne dužine 10 je popunjen, onda treba konstruisati novi, duži niz i primeniti prethodni postupak:

```
// Registrovanje novog igrača kada je niz listaIgrača popunjen
if (brojIgrača == listaIgrača.length) {
   int novaDužina = 2 * listaIgrača.length;
   Igrač[] noviNiz = Arrays.copyOf(listaIgrača, novaDužina);
   listaIgrača = noviNiz;
}

// Dodavanje novog igrača u stari ili novi niz
listaIgrača[brojIgrača] = noviIgrač;
brojIgrača++;
```

Primetimo da promenljiva `listaIgrača` ukazuje na stari, delimično popunjen niz ili na novi niz dvostruko duži od starog. Zbog toga, posle izvršavanja `if` naredbe u fragmentu, element `listaIgrača[brojIgrača]` je sigurno slobodan i možemo mu dodeliti novog igrača bez bojazni da ćemo premašiti granicu niza na koji ukazuje `listaIgrača`.

Nizovi čija se dužina može prilagoditi količini podataka koje treba da sadrže se nazivaju ***dinamički nizovi***. U radu sa dinamičkimm nizovima su dozvoljene iste dve operacije kao i sa običnim nizovima: upisivanje vrednosti u element na datoj poziciji i očitavanje vrednosti koja se nalazi u elementu na datoj poziciji. Ali pri tome ne postoji gornja granica za broj elemenata niza koji se mogu koristiti, osim naravno one koja zavisi od veličine memorije računara.

Umesto ručnog manipulisanja običnim nizom da bismo dobili efekat dinamičkog niza, odnosno umesto da svaki put pišemo programski kôd sličan onom koji smo pisali u prethodnim primerima za program kompjuterske igre, u Javi možemo koristiti standardnu klasu `ArrayList` iz paketa `java.util`. Klasa `ArrayList` opisuje objekte dinamičkih nizova koji u svakom trenutku imaju određenu dužinu, ali se ona automatski povećava kada je to potrebno. Slično kao kod običnih nizova, indeksi elemenata dinamičkog niza moraju biti u granicama od 0 do aktuelne dužine manje jedan.

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`. Pozivom podrazumevanog konstruktora u izrazu `new ArrayList()` se konstruiše 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 poziv ne može nalaziti na levoj strani znaka jednakosti u naredbi dodele.

- `set(n, elem)` — dodeljuje vrednost elementa `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 elementa se pomeraju jedno mesto ulevo i dužina niza se smanjuje za jedan.
- indexOf(elem) — pretražuje niz sleva na desno proveravajući da li se vrednost elementa elem nalazi 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.

Na primer, u programu za kompjutersku igru, niz listaIgrača možemo deklarisati da bude dinamički niz koji je početno prazan:

```
ArrayList listaIgrača = new ArrayList();
```

Sada ne moramo voditi računa o dužini niza, već za dodavanje novog igrača možemo jednostavno pisati:

```
listaIgrača.add(noviIgrač);
```

Slično, za uklanjanje k-tog igrača koristimo metod remove():

```
listaIgrača.remove(k);
```

ili, ako promenljiva isključenIgrač tipa Igrač ukazuje na objekat igrača kojeg treba ukloniti, onda možemo pisati:

```
listaIgrača.remove(isključenIgrač);
```

U ovim primerima je primena objektnih metoda klase ArrayList prirodna i očigledna. Međutim, za metod get(n) kojim se dobija vrednost elementa dinamičkog niza na poziciji n, moramo razumeti još neke detalje. Prvo, konstruktor ArrayList() konstruiše dinamički niz tipa Object[], odnosno elementi konstruisanog niza su tipa Object. Svaka klasa u Javi automatski nasleđuje klasu Object koja predstavlja objekte u najširem smislu. Klasa Object zato sadrži metode koji opisuju osnovne

mogućnosti svih objekata i o njoj će biti više reči u odeljku 8.4. Drugo, prema objektnom načelu koje se naziva *princip podtipa*, o čemu ćemo detaljnije govoriti u odeljku 8.2, referenca na objekat bilo kog tipa može biti dodeljena promenljivoj tipa `Object`. Zbog ovoga, kratko rečeno, u prethodnim primerima nismo morali da vodimo računa o tipu elemenata dinamičkog niza.

Međutim, deklarisan tip rezultata metoda `get()` je `Object`, dok primenom tog metoda na niz `listaIgrača` dobijamo zapravo objekat tipa `Igrač`. Zbog toga, da bismo mogli da uradimo nešto korisno sa rezultatom poziva metoda `get()`, moramo izvršiti eksplicitnu konverziju njegovog tipa:

```
Igrač pacer = (Igrač)listaIgrača.get(n);
```

Na primer, ako klasa `Igrač` sadrži objektni metod `odigrajPotez()` koji se poziva kada igrač treba da odigra svoj potez u igri, onda programski fragment za postupak kojim svaki aktivni igrač odigrava svoj potez može biti:

```
for (int i = 0; i < listaIgrača.size(); i++) {
   Igrač sledećiIgrač = (Igrač)listaIgrača.get(i);
   sledećiIgrač.odigrajPotez();
}
```

Dve naredbe u telu ove petlje se mogu spojiti u jednu:

```
((Igrač)listaIgrača.get(i)).odigrajPotez();
```

Ako ovu, u izvesnoj meri zamršenu naredbu raščlanimo na sastavne delove, vidimo da se najpre dobija i-ti element niza `listaIgrača`, zatim se vrši konverzija njegovog tipa u `Igrač` i, na kraju, poziva se metod `odigrajPotez()` za rezultujućeg igrača. Obratite pažnji i na to da se izraz `(Igrač)listaIgrača.get(i)` mora navesti u zagradama zbog unapred definisanih pravila prioriteta operatora u Javi.

U stvari, problem obavezne eksplicitne konverzije tipa prilikom očitavanja vrednosti nekog elementa dinamičkog niza više ne postoji od verzije Java 5.0. Taj problem je prevaziđen uvođenjem takozvanih ***parametrizovanih tipova***. Na primer, umesto običnog tipa `ArrayList` koji predstavlja dinamičke nizove čiji su elementi tipa `Object`, može se koristiti tip `ArrayList<T>`, gde je `T` bilo koji klasni tip (`T` ne sme biti primitivni tip). Tip `ArrayList<T>` predstavlja dinamičke nizove čiji su elementi tipa `T`. Naredbom deklaracije, na primer:

```
ArrayList<Igrač> listaIgrača = new ArrayList<Igrač>();
```

deklariše se promenljiva niza listaIgrača tipa ArrayList<Igrač>, konstruiše se prazan dinamički niz čiji su elementi tipa Igrač i referenca na njega se dodeljuje promenljivoj listaIgrača.

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, vrši se na jednostavan način kao i ranije. Na primer:

```
listaIgrača.add(noviIgrač);
```

ili

```
listaIgrača.remove(isključenIgrač);
```

Pored toga, Java prevodilac u fazi prevođenja programa sada može proveriti da li su noviIgrač i isključenIgrač zaista promenljive tipa Igrač i otkriti grešku ukoliko to nije slučaj. Ali ne samo to, pošto elementi niza listaIgrača moraju biti tipa Igrač, eksplicitna konverzija tipa nije neophodna prilikom očitavanja vrednosti nekog elementa tog niza:

```
Igrač pacer = listaIgrača.get(n);
```

Parametrizovan tip, kao što je to ArrayList<Igrač>, može se koristiti na potpuno isti način kao i običan tip: za deklarisanje promenljivih i za tip parametra ili rezultata nekog metoda. Jedini nedostatak parametrizovanih tipova je što parametar 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 o kojima smo govorili u odeljku 3.4. Sve klase koji su omotači primitivnih tipova mogu se koristiti za parametrizovane tipove. Objekat, recimo, tipa ArrayList<Double> predstavlja dinamički niz čiji elementi sadrže objekte tipa Double. Kako objekat tipa Double sadrži vrednost tipa double, to je skoro isto kao da ArrayList<Double> predstavlja dinamički niz čiji su elementi realni brojevi tipa double.

Na primer, ako su date deklaracije:

```
double x;
ArrayList<Double> nizBrojeva = new ArrayList<Double>();
```

onda se vrednost promenljive x može dodati u nizBrojeva metodom add():

```
nizBrojeva.add(new Double(x));
```

Pored toga, zbog autopakovanja i raspakovanja, možemo čak kraće pisati:

```
nizBrojeva.add(x);
```

jer se sve neophodne konverzije automatski primenjuju.

Primer: telefonski imenik

Telefonski imenik je niz stavki, pri čemu se svaka stavka sastoji od imena osobe i njenog telefonskog broja. Pojedine stavke telefonskog imenika se mogu, u najprostijem obliku, predstaviti objektima klase:

```
public class TelStavka {
   String ime;      // ime osobe
   String telBroj;  // tel. broj osobe
}
```

Telefonski imenik kao struktura podataka u programu je onda niz tipa ArrayList<TelStavka>. Međutim, za kompletan rad sa podacima u nekom imeniku, potrebno je napisati klasu TelImenik koja dodatno sadrži objektne metode kojima se realizuju tipične operacije nad konkretnim imenikom. Tu spadaju:

- Metod nađiBroj() koji za dato ime osobe nalazi njegov telefonski broj u imeniku.
- Metod dodajStavku() koji dodaje dati par ime/broj u imenik. Ovaj metod najpre proverava da li se dato ime nalazi u imeniku. Ako je to slučaj, stari broj se zamenjuje datim brojem. U suprotnom slučaju, nova stavka se dodaje u imenik.
- Metod ukloniStavku() koji uklanja stavku sa datim imenom osobe iz imenika. Ako se takva stavka ne pronađe u imeniku, ništa se posebno ne preduzima.

LISTING 7.6: Klasa za telefonski imenik.

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

```
class TelImenik {

   private ArrayList<TelStavka> imenik;  // niz stavki

   // Konstruktor klase konstruiše prazan imenik
   public TelImenik() {
      imenik = new ArrayList<TelStavka>();
   }

   private int nađiStavku(String imeOsobe) {

      for (int i = 0; i < imenik.size(); i++) {
         TelStavka s = imenik.get(i);
         // Da li i-ta osoba ima dato ime?
         if (s.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 = nađiStavku(imeOsobe);
      if (i >= 0)  // osoba se nalazi u imeniku?
         // Ako da, vratiti njegov tel. broj
         return imenik.get(i).telBroj;
      else
         // Ako ne, vratiti null referencu
         return null;
   }

   public void dodajStavku(String imeOsobe, String brojOsobe) {

      if (imeOsobe == null || brojOsobe == null) {
         System.out.println("Greška: prazno ime ili broj!");
         return;
      }
      int i = nađiStavku(imeOsobe);
      if (i >= 0)  // osoba se nalazi u imeniku?
         // Ako da, zameniti stari broj novim brojem
         imenik.get(i).telBroj = brojOsobe;
      else {
```

```
         // Ako ne, dodati novu stavku ime/broj u imenik
         TelStavka s = new TelStavka();
         s.ime = imeOsobe;
         s.telBroj = brojOsobe;
         imenik.add(s);
      }
   }

   public void ukloniStavku(String imeOsobe) {

      int i = nađiStavku(imeOsobe);
      if (i >= 0)  // osoba se nalazi u imeniku?
         // Ako da, ukloniti ga
         imenik.remove(i);
   }
}
```

Obratite pažnju na to da je u klasi definisan metod nađiStavku() u kojem se primenjuje linerana pretraga za nalaženje pozicije stavke sa datim imenom u imeniku. Metod nađiStavku() 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 stavke iz imenika i nalaženje telefonskog broja osobe na osnovu datog imena osobe. Jedan primer testiranja klase TelImenik je prikazan u sledećem glavnom metodu:

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

   TelImenik mojImenik = new TelImenik();
   mojImenik.dodajStavku("Pera", null);
   mojImenik.dodajStavku("Pera", "111-1111");
   mojImenik.dodajStavku("Žika", "222-2222");
   mojImenik.dodajStavku("Laza", "333-3333");
   mojImenik.dodajStavku("Mira", "444-4444");
   System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
   mojImenik.dodajStavku("Laza", "999-9999");
   System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
   System.out.println("Pera: " + mojImenik.nađiBroj("Pera"));
   mojImenik.ukloniStavku("Žika");
```

```
        System.out.println("Žika: " + mojImenik.nađiBroj("Žika"));
        System.out.println("Mira: " + mojImenik.nađiBroj("Mira"));
    }
```

Glava

8

Nasleđivanje klasa

Mogućnost nasleđivanja klasa je jedna od najvećih prednosti objektno orijentisanog programiranja u odnosu na proceduralno programiranje. Nasleđivanjem klasa možemo da napišemo novu klasu koja proširuje neku postojeću klasu. U ovom poglavlju ćemo najpre pokazati prednosti nasleđivanja klasa, a zatim ćemo se baviti pojedinostima tog koncepta u Javi.

8.1 Osnovni pojmovi

Klasa predstavlja skup objekata koji imaju zajedničku strukturu i mogućnosti. Neka klasa određuje strukturu objekata na osnovu objektnih polja, a mogućnosti objekata ta klasa određuje preko objektnih metoda. 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 da se klasama mogu izraziti sličnosti među objektima koji imaju neke, ali ne sve, zajedničke osobine.

U objektno orijentisanom programiranju, nova klasa se može napraviti na osnovu postojeće klase. To znači da nova klasa proširuje postojeću klasu i nasleđuje sva njena polja i metode. Prevedeno na objekte nove klase, ovo dalje znači da oni nasleđuje sve atribute i mogućnosti postojećih objekata. Ovaj koncept u OOP se naziva ***nasleđivanje klasa*** ili kraće

samo ***nasleđivanje***. Mogućnost proširivanja postojeće klase radi pravljenja nove klase ima mnoge prednosti od kojih su najvažnije polimorfizam i apstrakcija (o čemu detaljnije govorimo u odeljku 8.5 i odeljku 9.2), 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 polaznu i klasu-naslednicu: bazna i izvedena klasa, osnovna i proširena 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. Definisanja nove klase koja proširuje postojeću klasu se ne razlikuje mnogo od uobičajenog definisanja obične klase:

```
modifikatori class ime-nove-klase extends ime-stare-klase {
    .
    .  // Izmene ili dopune postojeće klase
    .
}
```

U ovom opštem obliku, *ime-nove-klase* je ime klase koja se definiše, a *ime-stare-klasa* je ime postojeće klase koja se proširuje. Telo nove klase između vitičastih zagrada ima istu strukturu kao telo neke uobičajene klase.

Kao što primećujemo iz opšteg oblika, jedina novost kod nasleđivanja je dakle 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 ili izmene
    .  // postojećih članova klase A
    .
}
```

Suštinu i prednosti nasleđivanja klasa je najbolje objasniti na nekom 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 može izgledati na primer:

```
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 " + ime + ", 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 koji obračunava platu 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 nam se odmah nameće, naročito ako dolazimo iz sveta proceduralnog programiranja, jeste da klasi Radnik dodamo indikator koji ukazuje na to da li se radi o jednoj ili drugoj vrsti radnika. Ako dodamo 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 koristi faktički 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 zaposli nova vrsta radnika u pogledu njihovog 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, možemo 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 " + ime + ", 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 " + ime + ", 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 trebamo dodati novu klasu, bar ne moramo menjati postojeće klase za stare vrste radnika. Ali, nedostatak ovog rešenja je što moramo ponavljati zajednička polja i metode 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 izbegavanje 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 " + ime + ", 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`.

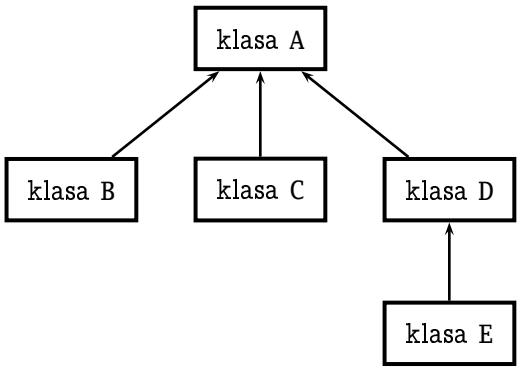
S druge strane, pretpostavimo da za svakog radnika želimo da čuvamo podatak o datumu zaposlenja. Pošto imamo standardnu klasu `Date` u Javi za predstavljanje datuma, da li je u ovom slučaju dobra ideja da klasu `Radnik` napišemo kao naslednicu klase `Date`? Odgovor je negativan, jer naravno ne važi odnos 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`.

8.2 Hijerarhija klasa

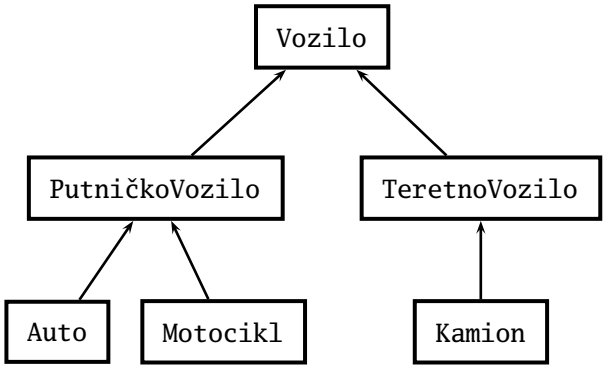
Proširena klasa kod nasleđivanja može dopuniti strukturu i mogućnosti koje nasleđuje. Ona može i zameniti ili modifikovati nasleđene mogućnosti, ali ne i nasleđenu strukturu. Prema opštem obliku definicije za klasu naslednicu, možemo zaključiti da u Javi klasa može direktno nasleđivati samo jednu klasu. Međutim, nekoliko klasa može direktno nasleđivati istu klasu. Ove klase-naslednice dele neke osobine koje nasleđuju od zajedničke klase-roditelja. Prema tome, nasleđivanjem se uspostavlja hijerarhijska relacija između srodnih klasa.

Relacija nasleđivanja između klasa se može prikazati ***klasnim dijagramom*** u kojem se klasa-dete nalazi ispod klase-roditelja i povezana je strelicom sa njom. Tako su na slici 8.1 klase `B`, `C` i `D` direktne naslednice zajedničke klase `A`, a klasa `E` je direktna naslednica klase `D`. Kao što možemo naslutiti sa slike 8.1, dubina nasleđivanja klasa u Javi nije ograničena i može se prostirati na nekoliko „generacija" klasa. Ovo je na slici 8.1 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 povezana relacijom nasleđivanja obrazuje na ovaj način ***hijerarhiju klasa***.



SLIKA 8.1: Hijerarhija klasa.

Razmotrimo jedan konkretniji primer. Pretpostavimo da pišemo program za evidenciju motornih vozila i da imamo klasu Osoba kojom se mogu predstaviti vlasnici motornih vozila. U programu se onda može definisati klasa 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 8.2.



SLIKA 8.2: Hijerarhija klasa motornih vozila.

Bazna klasa Vozilo treba da sadrži polja i metode koji su zajednički za sva vozila. To su, na primer, polja za vlasnika i brojeve motora i tablice, kao i metod za prenos vlasništva:

```
public class Vozilo {

   Osoba vlasnik;
   int brojMotora;
   String brojTablice;
   . . .
   public void promeniVlasnika(Osoba noviVlasnik) {
      . . .
   }
   . . .
}
```

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;
   Color boja;
   . . .
}

public class TeretnoVozilo extends Vozilo {

   int brojOsovina;
   . . .
}

public class Auto extends PutničkoVozilo {

   int brojSedišta;
   . . .
}

public class Motocikl extends PutničkoVozilo {

   boolean sedišteSaStrane;
   . . .
}

public class Kamion extends TeretnoVozilo {

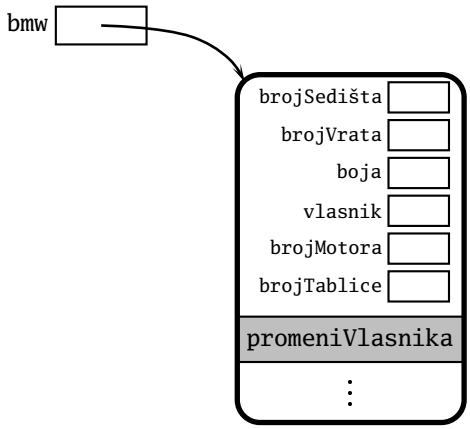
   int nosivost;
```

```
    . . .
}
```

Pretpostavimo sada da je bmw promenljiva klasnog tipa Auto koja je deklarisana i inicijalizovana naredbom:

```
Auto bmw = new Auto();
```

Efekat ove naredbe deklaracije u memoriji računara je prikazan na slici 8.3.



SLIKA 8.3: Objekat klase Auto na koga ukazuje bmw.

Pošto je brojSedišta objektno polje klase Auto, ono je naravno deo objekta na koga ukazuje promenljiva bmw. Zato u programu možemo pisati, na primer:

```
bmw.brojSedišta = 2;
```

Ali kako Auto nasleđuje PutničkoVozilo, objekat klase Auto na koga ukazuje promenljiva bmw sadrži i sva polja 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     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 koga 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` ima 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 da tretirajući objekat nekog podtipa kao da je objekat njegovog nadtipa, možemo iskoristiti samo njegove umanjene osobine, 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 principa podtipa na hijerarhiju klase `Vozilo` jeste da 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 fragmenata je da promenljiva mojeVozilo tipa Vozilo sadrži referencu na objekat tipa Auto koji je podtip tipa Vozilo.

Moguće je čak proveravati u programu da li dati objekat pripada određenoj klasi. Za tu svrhu se koristi logički operator instanceof koji se u opštem obliku piše na sledeći način:

ime-promenljive **instanceof** *ime-klase*

gde promenljiva na levoj strani mora biti klasnog tipa. Ako ova promenljiva sadrži referencu na objekat koji pripada klasi *ime-klase*, rezultat operatora instanceof je true; u suprotnom slučaju, rezultat je false. Na primer, naredbom:

```
if (mojeVozilo instanceof Auto) ...
```

određuje se da li objekat na koga ukazuje promenljiva mojeVozilo zaista pripada klasi Auto.

Radi izračunavanja rezultata operatora instanceof, što je samo jedan među ostalim razlozima, u svakom objektu se nalazi informacija o tome kojoj klasi objekat pripada. 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. Na primer, naredba dodele:

```
bmw = mojeVozilo;
```

nije ispravna, jer promenljiva mojeVozilo može sadržati reference objekata drugih tipova vozila koji nisu auta. Na primer, 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 tada bi upotreba polja bmw.brojSedišta, recimo, dovela do greške u programu zato što objekat na koga trenutno ukazuje bmw ne sadrži polje brojSedišta, prosto jer to polje nije definisano u klasi Kamion.

Ovo je nešto slično pravilu da nije dozvoljeno dodeliti neku vrednost tipa int promenljivoj tipa short, jer nije svaka vrednost tipa int isto-

vremeno 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 problema sa primitivnim tipovima, 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 napišemo u programu ne mora naravno odgovarati aktuelnom slučaju u toku izrvšavanja programa, pa se za klasne tipove prilikom izvršavanja programa proverava da li je zahtevana konverzija tipa ispravna. Tako, na primer, 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 koga ukazuje promenljiva klasnog tipa nije unapred poznat, ekplicitna 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 o enkapsulaciji 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 voditi računa da kod nasleđivanja dolazi u obzir i treći specifikator pristupa protected. Član klase koji je protected (zaštićen) može se koristiti

u (direktno ili indirektno) nasleđenoj klasi. 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 nasleđenim klasama 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 deklarišemo da bude zaštićeno kako bi dozvolili klasama-naslednicama, ali ne i svim klasama, da mogu slobodno menjati to polje. Još bolje rešenje bi bilo obezbediti `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.

S obzirom na specifikatore 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 ukoliko se želi omogućiti restriktivniji pristup privatnim poljima samo iz klasa-naslednica, onda geter i seter metodi za ta polja trebaju biti definisani kao zaštićeni.

8.3 Službena reč **super**

Klasa-naslednica u odnosu na svoju nasleđenu klasu može imati dodatne ili zamenjene članove (polja i metode). Za korišćenje dodatnih članova u klasi-naslednici nema nejasnoća, jer takvi članovi ne postoje u nasleđenoj klasi, pa se mogu nedvosmisleno prepoznati na osnovu svojih prostih imena. U slučaju zamenjenih članova koji su definisani u klasi-naslednici, a već postoje u nasleđenoj klasi, može se postaviti pitanje na šta se misli kada se u klasi-naslednici koriste njihova imena: da li na

primerak definisan u klasi-naslednici ili na onaj definisan u nasleđenoj klasi?

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. U klasi-naslednici isti član koji postoji i u nasleđenoj klasi, u stvari, ne zamenjuje taj član iz nasleđene klase, nego ga samo *zaklanja*. Taj zaklonjeni član iz nasleđene klase se i dalje može koristiti u klasi-naslednici 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.

Zaklanjanje polja u klasi-naslednici se retko koristi, ali ako je to slučaj, onda objekat klase-naslednice sadrži zapravo dva polja sa istim imenom: jedno koje je definisano u samoj klasi-naslednici i drugo koje je definisano u nasleđenoj klasi. Novo polje u klasi-naslednici ne zamenjuje staro polje sa istim imenom u nasleđenoj klasi, nego ga samo zaklanja time što se podrazumeva da se svaka prosta upotreba tog imena u klasi-naslednici odnosi na primerak polja definisanog u klasi-naslednici.

Slično, ako se u klasi-naslednici definiše metod koji ima isti potpis kao neki metod u nasleđenoj klasi, metod iz nasleđene klase je zaklonjen u klasi-naslednici na isti način. U ovom slučaju kažemo da metod u klasi-naslednici ***nadjačava*** metod iz nasleđene klase. Ali ako je nadjačan metod iz nasleđene klase potreban u klasi-naslednici, opet se može koristiti službena reč super uz njega.

Kao primer razmotrimo jednostavnu klasu Dete koju nasleđuje klasa Učenik: klasa Dete služi za registrovanje podataka pojedine dece, dok klasa Učenik služi za registrovanje podataka pojedinih učenika osnovnih škola. Klasa Učenik nasleđuje klasu Dete, jer je svaki učenik ujedno i 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 deklaracije:

```
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 smo u metodu prikaži() klase Učenik pozivali 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

učenika proširivao funkcionalnost metoda `prikaži()` za prikazivanje podataka deteta.

Neko bi mogao prigovoriti da u prethodnom primeru 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` smo mogli 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 smo u odeljku 5.6 koristili službenu reč `final` radi definisanja promenljivih (i polja) kojima se vrednost može dodeliti samo jedanput. U slučaju `final` metoda, onemogućava se nadjačavanje takvog metoda u bilo kojoj klasi koja nasleđuje onu u kojoj je metod definisan.

U stvari, modifikator `final` možemo 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 {...}
```

Prema tome, `final` klasa predstavlja 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 sa 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 jezika Java.

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 se nemaju 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 namena te reči je povezana sa njenom glavnom namenom 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 nadjačanog metoda. To u opštem slučaju izgleda kao da je reč super ime metoda, iako ona to nije a i konstruktori se ne pozivaju kao ostali metodi:

super(*lista-argumenata*)

Na primer, u konstruktoru klase `Učenik` prethodnog primera, pozvali smo 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 prvog konstruktora.
- Ako prva naredba pozvanog konstruktora predstavlja 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 prvog konstruktora.
- Ako prva naredba pozvanog konstruktora predstavlja poziv nadjačanog konstruktora pomoću službene reči this, poziva se odgovarajući nadjačani konstruktor aktuelne klase. Posle završetka tog poziva se odmah nastavlja sa izvršavanjem prvog konstruktora. Poziv konstruktora nasleđene klase se izvršava unutar nadjačanog konstruktora, bilo eksplicitno ili implicitno, tako da se i inicijalizacija objektnih polja tamo završava.

8.4 Klasa Object

Glavna odlika objektno orijentisanog programiranja je mogućnost nasleđivanja klasa i definisanja nove klase na osnovu stare klase. Nova klasa nasleđuje sve atribute i mogućnosti postojeće klase, ali ih može modifikovati ili dodati nove.

Objektno orijentisani programski jezik Java je karakterističan po tome što sadrži specijalnu klasu Object koja se nalazi na vrhu ogromne hijerarhije klasa koja uključuje sve druge klase u Javi. Klasa Object definiše osnovne mogućnosti koje moraju imati svi objekti. Tu spadaju mogućnosti proveravanja jednakosti 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 tipa `Object`, ove strukture podataka mogu zapravo sadržati objekte proizvoljnog tipa. Jedan primer ovih struktura su dinamički nizovi tipa `ArrayList` o kojima smo govorili u odeljku 7.5.

U klasi `Object` je dakle definisano nekoliko objektnih metoda koje nasleđuje svaka klasa. Ovi metodi se zato mogu bez izmena koristiti u svakoj klasi, ali se mogu i nadjačati ukoliko njihova osnovna funkcionalnost nije odgovarajuća. Ovde ćemo pomenuti samo one metode iz klase `Object` koji su korisni za jednostavnije primene — ostali se tiču složenijih programskih celina (tzv. višenitnih programa) 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 ovaj metod 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 klase obično nije odgovarajući, pa u toj klasi treba nadjačati ovaj metod radi realizacije pojma jednakosti njenih objekata.

Na primer, dva objekta-deteta klase Dete iz prethodnog odeljka možemo smatrati jednakim ukoliko imaju isto ime i uzrast. Zato u klasi Dete treba nadjačati osnovni metod equals() na sledeći način:

```
public boolean equals(Object obj) {
   if (obj == null || !(obj instanceof Dete))
      return false;
   Dete drugoDete = (Dete)obj; // Konverzija tipa obj u Dete
   if (this.ime == drugoDete.ime &&
       this.uzrast == drugoDete.uzrast)
         // Drugo dete ima isto ime i uzrast kao ovo dete,
         // pa se podrazumeva da su jednaki objekti-deca.
      return true;
   else
      return false;
}
```

Drugi primer je metod equals() u standardnoj klasi String kojim se dva stringa (objekta klase String) smatraju jednakim ako se sastoje od tačno istih nizova znakova (tj. istih 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 string objekta 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 nala-

ženje objekata. (Ove i slične strukture podataka su u Javi predstavljene klasama HashTable, HashMap i HashSet.)

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 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) čija je uloga da predstavlja objekat, za koji je metod pozvan, u obliku stringa. 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 mojeDete ukazuje na neki objekat klase Dete iz prethodnog odeljka, onda bi se kao rezultat poziva:

```
mojeDete.toString()
```

dobio string oblika "Dete@163b91" Pošto ovo nije mnogo korisno, u sopsttvenim klasama treba definisati novi metod toString() koji će nadjačati ovu nasleđenu verziju. Na primer, u klasi Dete iz prethodnog odeljka možemo dodati sledeći metod:

```
public String toString() {
   return ime + " (" + uzrast + ")";
}
```

Ako promenljiva mojeDete ukazuje na neki objekat klase Dete, onda bi se ovaj metod automatski pozivao za konvertovanje tog objekta u string ukoliko se mojeDete koristi u kontekstu u kojem se očekuje string. Na primer, izvršavanjem ovog programskog fragmenta:

```
Dete mojeDete = new Dete("Aca", 10);
System.out.print("Moj mali je " + mojeDete);
System.out.println(", ljubi ga majka.");
```

na ekranu se 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 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 da budu isto tako 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.

8.5 Polimorfizam

Tri stuba na koja se oslanja objektno orijentisano programiranje su enkapsulacija, nasleđivanje i polimorfizam. Do sada smo upoznali prva dva koncepta, a sada ćemo objasniti polimorfizam.

Reč polimorfizam ima korene u grčkom jeziku i znači otprilike „mnoštvo oblika" (na grčkom *poli* 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 u odeljku 5.5. 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 obe ove 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 8.3. 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 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 koga ljubimac ukazuje. A na osnovu tog tipa će se onda pozvati i odgovarajuća verzija metoda koSamJa().

Na primer, posle izvršavanja ovog programskog fragmenta:

```
KućniLjubimac ljubimac1 = new KućniLjubimac();
KućniLjubimac ljubimac2 = new Pas();
KućniLjubimac ljubimac3 = new Mačka();
ljubimac1.koSamJa();
ljubimac2.koSamJa();
ljubimac3.koSamJa();
```

na ekranu se dobija rezultat:

```
Ja sam kućni ljubimac.
Ja sam pas.
Ja sam mačka.
```

U ovom primeru su deklarisane tri promenljive ljubimac1, ljubimac2 i ljubimac3 istog tipa KućniLjubimac. Međutim, samo prva promenljiva ukazuje na objekat tipa KućniLjubimac, dok ostale dve ukazuju na objekte podtipova tog tipa. Na osnovu rezultata izvršavanja programskog fragmenta možemo zaključiti da je verzija metoda koSamJa() koja se poziva, stvarno ona koja odgovara tipu objekata na koje 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 koga ukazuje ta promenljiva, bez obzira na njen deklarisan tip. Zahvaljujući ovom aspektu polimorfizma možemo na efikasan način manipulisati velikim brojem objekata u istoj hijerarhiji klasa. Na primer, ako imamo deklarisan niz koji predstavlja populaciju svih kućnih ljubimaca koje drže žitelji nekog grada od 100000 stanovnika:

```
KućniLjubimac[] populacija = new KućniLjubimac[100000];
```

i ako pojedinačni elementi niza ukazuju na objekte tipova KućniLjubimac, Pas ili Mačka, onda pojedinačne vrste ljubimaca možemo prikazati u običnoj petlji:

```
for (KućniLjubimac ljubimac : populacija) {
   ljubimac.koSamJa();
}
```

Pored toga, ako hijerahiji klasa kućnih ljubimaca kasnije dodamo novu izvedenu klasu, na primer:

```
public class Kanarinac extends KućniLjubimac {
   public void koSamJa() {
      System.out.println("Ja sam kanarinac.");
   }
}
```

i ako neki elementi niza populacija mogu sada ukazivati i na objekte nove klase Kanarinac, onda prethodna petlja za prikazivanje vrste svih kućnih ljubimaca ne mora uopšte da se menja. Ova mogućnost u opštem slučaju, da možemo pisati programski kôd koji će uraditi nešto što nismo čak ni zamislili u vreme njegovog pisanja, jeste verovatno najznačanija osobina polimorfizma.

Glava

9

Posebne klase i interfejsi

Pored običnih klasa, o kojima smo do sada govorili, u Javi se mogu koristiti druge vrste klasa čija prava vrednost dolazi do izražaja tek u složenijim primenama objektno orijentisanog programiranja. U ovom poglavlju ćemo se upoznati sa takvim posebnim vrstama klasa.

U stvari, najpre ćemo govoriti o jednostavnom konceptu nabrojivih tipova koji su važni i za svakodnevno programiranje. Nabrojivim tipom se prosto predstavlja ograničen (mali) skup vrednosti. Mogućnost definisanja nabrojivih tipova je relativno skoro dodata programskom jeziku Java (od verzije 5.0), ali mnogi programeri smatraju da su nabrojivi tipovi trebali biti deo Jave od samog početka.

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.

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.

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 korišćenje mnogih standardnih tehnika u Javi. Prema tome, da bi Java programeri bili uspešni u svom poslu, oni moraju poznavati pomenute naprednije mogućnosti objektno orijentisanog programiranja u Javi bar na upotrebnom nivou.

9.1 Nabrojivi tipovi

Do sada smo naučili da 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, upoznali smo u poglavlju 3. Vrednosti klasnih tipova su objekti definišućih 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 naravno mogli da kodiramo 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 definisati novi nabrojivi tip:

```
public enum GodišnjeDoba {PROLEĆE, LETO, JESEN, ZIMA}
```

Ovim se definiše nabrojivi tip `GodišnjeDoba` koji se sastoji od četiri vrednosti čija su imena `PROLEĆE`, `LETO`, `JESEN` i `ZIMA`. Po konvenciji kao za obične konstante, vrednosti nabrojivog tipa se pišu velikim slovima i sa donjom crtom ukoliko se sastoje od više reči.

U pojednostavljenom 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, međusobno odvojene zapetama, navode se unutar vitičastih zagrada kao *lista-konstanti*. Pri tome, svaka konstanta u listi mora biti prost identifikator.

Definicija nabrojivog tipa ne može stajati unutar metoda, već mora biti član klase. Tehnički, nabrojivi tip je specijalna vrsta ugnježđene klase koja sadrži konačan broj instanci (nabrojive konstante) 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);
```

Iako se nabrojive konstante tehnički smatraju da su objekti, obratite ovde pažnju na to da se u if naredbi ne koristi metod equals() nego operator ==. To je zato što svaki nabrojivi tip ima fiksne, unapred poznate

[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.

instance i zato je za proveru njihove jednakosti dovoljno uporediti njihove reference. Primetimo isto tako da se drugom naredbom `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 odgovarajuće klase (iako se nabrojive konstante ne deklarišu 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 imenovane 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-zarez. 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 klasu `Enum` i zato se u radu sa 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);
```

Ovde se u naredbi println, kao za svaku promenljivu klasnog tipa, implicitno poziva metod modnaSezona.toString() i dobija kao rezultat string "LETO" (koji se na ekranu prikazuje bez navodnika).

Obrnutu funkciju metoda toString() u klasi Enum ima statički metod valueOf() čije je zaglavlje:

```
static Enum valueOf(Class nabrojiviTip, String ime)
```

Metod valueOf() kao rezultat vraća nabrojivu konstantu datog nabrojivog tipa sa datim imenom. 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 prilikom izvršavanja ovog programskog fragmenta preko tastature unesemo 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 int vrednost 0, GodišnjeDoba.LETO.ordinal() daje int vrednost 1 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 čije su vrednosti GodišnjeDoba.PROLEĆE, GodišnjeDoba.LETO, GodišnjeDoba.JESEN i GodišnjeDoba.ZIMA može se deklarisati 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
```

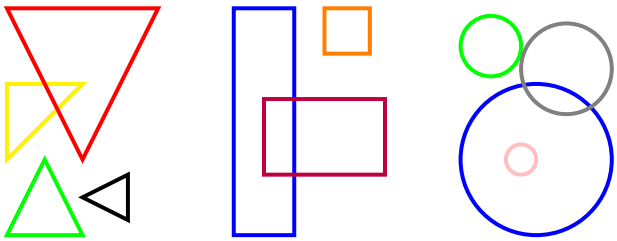
Nabrojivi tipovi se mogu koristiti i u paru sa switch naredbom — preciznije, izraz u switch naredbi može biti nabrojivog tipa. Konstante uz klauzule case onda moraju biti nabrojive konstante odgovarajućeg tipa, pri ćemu se moraju pisati njihova prosta imena a ne puna. Na primer:

```
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;
}
```

9.2 Apstraktne klase

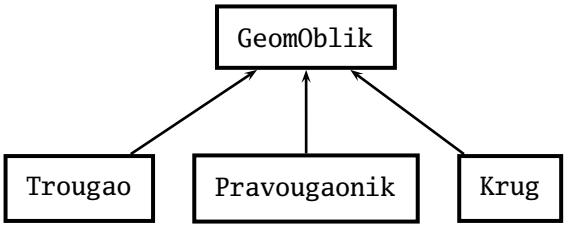
Da bismo objasnili svrhu primene apstraktnih klasa, uzmimo primer jednog programa koji crta geometrijske oblike na ekranu. Pretpostavimo da program treba da manipuliše geometrijskim oblicima među kojima

su mogući trouglovi, pravougaonici i krugovi u raznim bojama. Primer objekata ovog programa je prikazan na slici 9.1.



SLIKA 9.1: Geometrijski oblici programa za crtanje.

Nakon analize problema i imajući u vidu objektno orijentisani pristup, recimo da smo odlučili da definišemo tri klase `Trougao`, `Pravougaonik` i `Krug` koje će predstavljati tri tipa mogućih geometrijskih oblika u programu. Ove tri klase će nasleđivati zajedničku klasu `GeomOblik` koja će predstavljati zajednička svojstva koje poseduju sva tri oblika. Klasnim dijagramom na slici 9.2 je prikazano stablo nasleđivanja klase `GeomOblik`.



SLIKA 9.2: Hijerarhija klasa geometrijskih oblika.

Zajednička klasa `GeomOblik` može imati objektna polja za boju, poziciju na ekranu i veličinu geometrijskog oblika, kao i objektne metode za promenu boje, pozicije i veličine oblika. Postupak promene boje nekog oblika može se sastojati recimo najpre od promene vrednosti odgovarajućeg polja tog oblika i zatim ponovnog crtanja tog oblika u novoj boji na ekranu:[2]

```
public class GeomOblik {
```

[2] `Color` je standardna klasa Jave u paketu `java.awt` o kojoj će više reči biti u poglavlju 10.

```
    private Color boja;

    public void oboji(Color novaBoja) {
       boja = novaBoja;
       nacrtaj();
    }

    public void nacrtaj() {
       .
       . // Naredbe za crtanje geometrijskog oblika
       .
    }

    // Ostala polja i metodi
    . . .
}
```

Postavlja se pitanje kako napisati metod `nacrtaj()` u klasi `GeomOblik`? Problem je u tome što se svaki tip geometrijskih oblika crta na drugačiji način. Metod `oboji()` se može pozvati za svaki tip objekata, ali kako se u tom metodu može znati koji oblik treba nacrtati kada se izvršava metod `nacrtaj()`? Neformalan odgovor je da svaki oblik mora znati šta treba da uradi da bi sebe nacrtao. Praktično to znači da svaka klasa geometrijskih oblika ima sopstveni metod `nacrtaj()` koja nadjačava verziju tog metoda u klasi `GeomOblik`:

```
public class Trougao extends GeomOblik {

    public void nacrtaj() {
       .
       . // Naredbe za crtanje trougla
       .
    }

    // Ostala polja i metodi
    . . .
}

public class Pravougaonik extends GeomOblik {

    public void nacrtaj() {
       .
```

```
        . // Naredbe za crtanje pravougaonika
        .
    }

    // Ostala polja i metodi
    . . .
}

public class Krug extends GeomOblik {

    public void nacrtaj() {
        .
        . // Naredbe za crtanje kruga
        .
    }

    // Ostala polja i metodi
    . . .
}
```

Ako je promenljiva oblik klasnog tipa GeomOblik:

```
GeomOblik oblik;
```

onda ona u programu može ukazivati, prema principu podtipa, na objekat bilo kojeg od tipova 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 na osnovu principa polimorfizma, kada se izvršava naredba poziva:

```
oblik.nacrtaj();
```

onda se poziva metod nacrtaj() iz odgovarajuće klase kojoj pripada objekat na koga 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 koga 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 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` predstavlja apstraktni pojam geometrijskog oblika i zato ne postoji način da se tako nešto nacrta. 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 možemo prosto izostaviti iz klase `GeomOblik`, jer onda u prethodnom primeru za promenljivu `oblik` tipa `GeomOblik` naredba poziva:

```
oblik.nacrtaj();
```

ne bi bila ispravna. To je zato što bi prilikom prevođenja programa Java prevodilac proveravao da li klasa `GeomOblik`, koja je deklarisani tip promenljive `oblik`, sadrži metod `nacrtaj()`. Ako to nije slučaj, prevođenje ne bi uspelo i ne bismo dobili izvršnu verziju programa.

Sa druge strane, verzija metoda `nacrtaj()` iz klase `GeomOblik` se u programu neće nikad ni pozivati. Naime, ako bolje razmislimo, u programu ne postoji potreba da ikad konstruišemo stvarni objekat tipa `GeomOblik`. Ima rezona da deklarišemo neku promenljivu tipa `GeomOblik` (kao što je to `oblik`), ali objekat na koga ona ukazuje će uvek pripadati nekoj klasi koja nasleđuje klasu `GeomOblik`. Zato za klasu `GeomOblik` kažemo da je to apstraktna klasa. ***Apstraktna klasa*** je ona 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 možemo konstruisati samo objekte 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` kažemo 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 naznačiti dodavanjem modifikatora abstract u zaglavlju definicije klase ili metoda. Dodatno, u slučaju apstraktnog metoda, telo takvog metoda kojeg čini blok naredba između vitičastih zagrada se ne navodi, nego se zamenjuje tačkom-zapetom. Na primer, definicija klase GeomOblik kao apstraktne klase može imati sledeći oblik:

```
public abstract class GeomOblik {

   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 ili više apstraktnih metoda 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, imamo polje boja i konkretan metod oboji().

Klasa koja je u programu definisana da bude apstraktna ne može se instancirati, odnosno nije dozvoljeno operatorom new konstruisati objekte te klase. Na primer, ako u programu napišemo izraz:

```
new GeomOblik()
```

dobija se greška 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 opet da možemo deklarisati *objektne promenljive* čiji je tip neka apstraktna klasa. 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, ona mora sadržati definicije konkretnih nadjačanih verzija svih apstraktnih metoda nasleđene klase. To znači da ukoliko proširujemo neku apstraktnu klasu, onda imamo dve mogućnosti. Prva je da ostavimo neke ili sve apstraktne metode nedefinisane. Onda je i nova klasa apstraktna i zato se mora navesti službena reč abstract u zaglavlju njene definicije. Druga mogućnost je da definišemo sve nasleđene apstraktne metode i onda dobijamo novu konkretnu klasu.

9.3 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 kao što znamo neka klasa u Javi može direktno proširivati samo jednu klasu. Međutim, u Javi postoji koncept ***interfejsa*** koji obezbeđuje slične mogućnosti kao višestruko nasleđivanje, ali ne povećava značajno složenost programskog jezika.

U poglavlju 5 smo upoznali termin „interfejs" u vezi sa slikom nekog 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, doduše potpuno različito, koncepcijsko značenje za koje postoji službena reč `interface`. U tom značenju ćemo od sada u tekstu podrazumevati termin „interfejs", osim naravno ukoliko nije drugačije napomenuto. 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. Uzmimo jedan jednostavan primer interfejsa u Javi. Metod `sort()` klase `Arrays` može 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 vidimo, 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`, svaka klasa koja implementira 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 negova 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 u metodu compareTo() klase Radnik, radnici se 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 implementira 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[] spisakRadnika = new Radnik[500];
   .
   .
   .
Arrays.sort(spisakRadnika);
```

Napomenimo još da se u vezi sa interfejsima često koristi jedan kraći terminološki izraz. Naime, kaže se da neki objekat implementira interfejs, kada se zapravo misli da objekat pripada klasi koja implementira dati interfejs. Tako, u prethodnom primeru možemo reći da svaki 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 je 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()
```

izaziva se greška u programu.

Definicijom nekog interfejsa u Javi se tehnički uvodi novi klasni tip podataka. Vrednosti tog klasnog tipa su objekti klasa koje implementiraju definisani interfejs. To znači da ime interfejsa možemo 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 objektne 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 KlasičanRadio extends Radio {
   void izaberiStanicu();
}

public interface InternetRadio extends Radio {
   void izaberiSajt();
}
```

Za razliku od klase koja može direktno proširiti samo jednu klasu, interfejs 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 KlasičanRadio extends Radio {
   double B92 = 92.5;
   double STUDIO_B = 99.1;
   void izaberiStanicu();
}
```

Slično kao što se za metode interfejsa podrazumeva da su public, 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` mora sadržati definicije metoda `clone()` i `compareTo()`.

Interfejsi i apstraktne klase

Pažljiviji čitaoci se verovatno pitaju zašto su nam 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 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 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 ili bez nasleđivanja jedne klase):

```
public class Radnik extends Osoba implements Comparable  // OK
```

9.4 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 konstrukcije, mogu ugnježđavati i da li to ima smisla? Odogovor 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 (na kraći način) definisati baš na mestu konstruisanja njenog jedinstvenog objekta.
3. Metodi ugnježđene klase mogu koristiti sva polja, čak i privatna, obuhvatajuće klase.

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 razmevanje jednostavnijih

programskih tehnika.[3] Primena ugnježđenih klasa je naročito korisna kod grafičkog programiranja o čemu ćemo govoriti u poglavlju 10. 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 {
      . . .
   }
}
```

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, blok-naredbe) druge klase. Takve ugnježđene klase mogu biti ***lokalne*** ili ***anonimne***. Ove ugnježđene klase imaju koncepcijski slične osobine kao lokalne promenljive metoda.[4]

[3] 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.

[4] Nestatičke ugnježđene klase (tj. objektne, lokalne i anonimne) često se nazivaju *unutrašnje klase*.

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`.[5] 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:

```
public class KoordSistem {

   // Ostali članovi klase KoordSistem
   . . .

   public static class Duž {
         // Predstavlja duž u ravni od
         // tačke (x1,y1) do tačke (x2,y2)
      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

[5] Nabrojivi tip definisan unutar neke klase smatra se statičkom ugnježđenom klasom, iako se u njegovoj definiciji ne navodi modifikator `static`.

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 analogna 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.

Kao što je poznato, nestatički članovi regularne klase određuju šta će se nalaziti u svakom objektu koji se konstruiše na osnovu te klase. To važi i za objektne ugnježđene klase, bar logički, odnosno možemo 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*

pri čemu je *promenljiva* 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 koga ukazuje `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 kartašku igru tablića. 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 koga `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 šil 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 klasa koja je definisana unutar nekog metoda druge klase.[6] 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 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 oblasti 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 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

[6] Tačnije, lokalna klasa se može definisati unutar bloka Java koda.

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 će biti više reči u poglavlju 10 u kojem se govori o 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 nad njim.

U primeru programskog fragmenta koji sledi, 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 pristiskom tastera 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. Ovde je važno primetiti da nam treba samo jedan 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!
```

Pošto 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 posebne 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

10

Grafičko programiranje

Do sada smo pisali konzolne Java programe koji ulazne podatke prihvataju sa 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 onaj deo programa koji korisnik vizuelno vidi na ekranu i sa kojim može biti u interakciji tokom izvršavanja programa.

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.

10.1 Uvod

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 sa 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 jezika Java jeste da 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. Iz biblioteke Swing se dobijaju grafičke komponente sa većim mogućnostima, ali se iz 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.

Java grafički programi se danas 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 bazni 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 sa njima iz biblioteke AWT.

Primer: grafički ulaz/izlaz programa

Da bismo pokazali kako se u Java programima može jednostavno realizovati grafički ulaz/izlaz, sada ćemo ponoviti prvi Java program koji smo predstavili u odeljku 2.5, ali sa korišćenjem grafičkih dijaloga za ulazne i izlazne podatke. Radi jednostavnosti, ovaj program nema svoj glavni prozor, 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 smo koristili onu 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 10.1: Program *Zdravo* sa grafičkim dijalozima za ulaz/izlaz.

```
import javax.swing.*;

public class ZdravoGUI {

   public static void main(String[] args) {

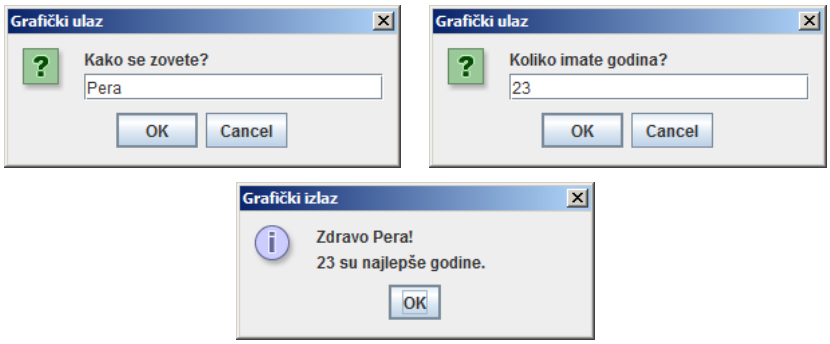
      String ime = JOptionPane.showInputDialog(null,
                        "Kako se zovete?",
                        "Grafički ulaz",
                        JOptionPane.QUESTION_MESSAGE);
      String godine = JOptionPane.showInputDialog(null,
                        "Koliko imate godina?",
                        "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()` je 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()` klase-omotača `Integer`. Izvršavanjem programa se dobijaju tri okvira za dijalog koja su prikazana na slici 10.1.



SLIKA 10.1: Tri okvira za dijalog grafičkog programa *Zdravo*.

10.2 Grafički elementi

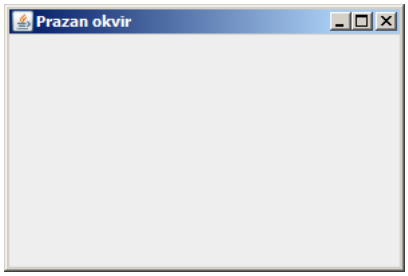
Grafički elementi u Javi koji se mogu koristiti za pravljenje korisničkog interfejsa programa pripadaju trima glavnim kategorijama:

- *Prozori*. To su jednostavne provaougaone oblasti koje se mogu nacrtati na ekranu. Prozori predstavljaju kontejnere najvišeg nivoa i dalje se dele u dve podvrste:
  - *Okviri*. To su permanenti prozori koji mogi imati meni sa opcijama. Okviri su predviđeni da ostaju na ekranu za sve vreme izvršavanja programa.

  - *Dijalozi*. To su privremeni prozori koji se mogu skloniti sa ekrana tokom izvršavanja programa.
- *Komponente*. To su vizuelni elementi koji imaju veličinu i poziciju na ekranu i koji mogu proizvoditi događaje.
- *Kontejneri*. To su komponente koje mogu obuhvatati druge komponente.

Prozor najvišeg nivoa, odnosno prozor koji se ne nalazi unutar drugog prozora, u Javi se naziva ***okvir*** (engl. *frame*). Okvir je dakle nezavisan prozor koji može poslužiti 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 i meniji. Zbog toga 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 je definisana klasa `JFrame` za konstruisanje i podešavanje okvira koji ima pomenute standardne mogućnosti. Ono što okvir tipa `JFrame` nema su druge komponente koje čine njegov sadržaj unutar okvira. One se moraju programski dodati nakon konstruisanja okvira. Primer jednostavnog programa koji prikazuje prazan okvir na ekranu sa slike 10.2 je naveden u listingu 10.2.



Slika 10.2: Prazan okvir na ekranu.

Listing 10.2: 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 podrazumevanu, prilično beskorisnu veličinu širine 0 i visine 0 piksela.[1] Zbog toga se u primeru 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 toga šta se događa 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.

[1] Sve grafičke dimenzije u Javi se navode u jedinicama koje označavaju pojedinačne tačke na ekranu i zovu se pikseli.

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 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 prosleđivanje 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 odgovarajućih klasa tako da `JFrame` nasleđuje mnoge 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` prepušta kontrolu operativnom sistemu da po svom nahođenju izabere poziciju (ali ne i veličinu) okvira na ekranu.
- Metod `setBounds()` služi za pozicioniranje okvira i istovremeno određivanje njegove veličine na ekranu. Na primer, dve naredbe:

  ```
  okvir.setSize(300, 200);
  okvir.setLocation(100, 150);
  ```

  možemo zameniti jednom:

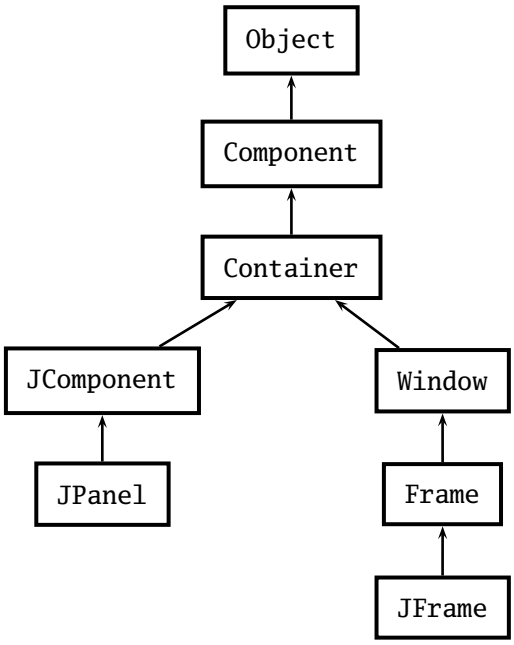
  ```
  okvir.setBounds(100, 150, 300, 200);
  ```

- 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 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 10.3 je prikazan mali deo ovog stabla nasleđivanja koji je relevantan za osnovne grafičke elemente.[2]



Slika 10.3: Hijerarhija klasa AWT/Swing grafičkih elemenata.

Klase `Component` i `Container` u biblioteci AWT su apstraktne klase

[2]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.

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.

Klase JComponent u biblioteci Swing je dete klase Container i predak svih Swing 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 smo već predočili, predviđen da bude prozor najvišeg nivoa. Njegova struktura je relativno složena i svi njeni detalji prevazilaze ciljeve ove knjige. Napomenimo samo da jedna od posledica specijalnog statusa okvira kao glavnog prozora jeste da 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 njihovo željeno razmeštanje unutar nekog kontejnera višeg nivoa. Ova primena panela kao potkontejnera je najrasprostranjenija, ali kako 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 klase JButton, JLabel, JTextField, JCheckBox, JRadioButton i JComboBox. (Ovo su samo neke od osnovnih komponenti — potpun spisak svih komonenti u biblioteci AWT/Swing je mnogo veći.) U svakoj od ovih klasa standardnih

komponenti je definisano nekoliko konstruktora koji se koriste za konstruisanje konkretnih komponenata-objekata. Neki primeri konstruisanja dugmadi, oznaka, tekstalnog polja, polja za potvrdu, radio dugmadi i kombinovanih polja su dati u sledećem programskom fragmentu:

```
// Dugme sa tekstom "OK"
JButton dugmeOK = new JButton("OK");

// Oznaka sa tekstom "Unesite svoje ime: "
JLabel oznakaIme = new JLabel("Unesite svoje ime: ");

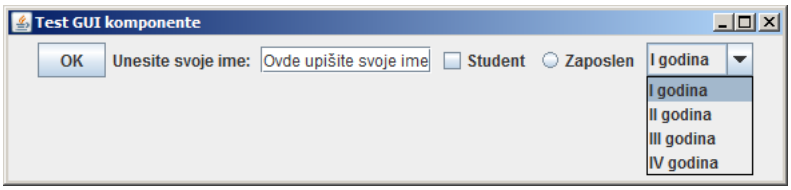
// Tekstualno polje sa tekstom "Ovde upišite svoje ime"
JTextField tekstPoljeIme =
               new JTextField("Ovde upišite svoje ime");

// 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"});
```

Na slici 10.4 je prikazan izgled ovih objekta unutar glavnog okvira programa.



Slika 10.4: Neke standardne GUI komponente iz biblioteke Swing.

Dodavanje komponente nekom kontejneru, kao i dodavanje jednog kontejnera drugom kontejneru, vrši se metodom add(). Ovaj metod ima više preopterećenih verzija, ali u najjednostavnijem obliku se kao njegov argument navodi komponenta ili kontejner koji se dodaje kontejneru. Na

primer, ako panel ukazuje na panel tipa JPanel i okvir ukazuje na okvir tipa JFrame, onda standardnu komponentu dugme možemo dodati panelu naredbom:

```
panel.add(new JButton("Test"));
```

Paneli se mogu nalaziti unutar okvira ili drugog panela, pa se prethodni panel sa dugmetom može dodati okviru na koga ukazuje 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 predstavljaju najmanje tačke za crtanje na ekranu. Popularno ime ove jedinice je ***piksel***, što je kovanica od engleskog termina *picture element*.

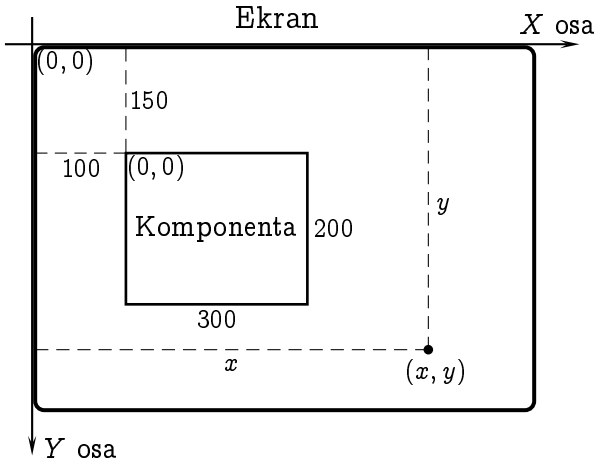
Ekran se fizički sastoji od dvodimenzionalne matrice tačaka sa određenim brojem redova i kolona koji zavise od rezolucije ekrana. Tako, na primer, rezolucija ekrana 1024×768 označava da je ekran podeljen u 1024 redova i 768 kolona u čijim presecima se nalazi jedna tačka za crtanje slike. Ovakva fizička 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 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 (videti sliku 10.5).

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



SLIKA 10.5: Koordinatni sistem ekrana i komponenti.

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");  // OK dugme
dugme.setBounds(20, 100, 60, 40);   // granice OK dugmeta
okvir.add(dugme);  // dodavanje OK dugmeta u okvir
```

Gornji levi ugao dugmeta se nalazi u tački (20, 100) relativno u odnosu na gornji levi ugao okvira kome se dodaje. (Ovde pretpostavljamo 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.

Razmeštanje komponenti unutar kontejnera

Prilikom dodavanja komponenti u kontejner, one se unutar kontejnera razmeštaju na određen način. To znači da se po unapred definisanom

postupku 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 implementirati. 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 biblioteci AWT/Swing:

- *FlowLayout*
- *GridLayout*
- *BorderLayout*
- *GridBagLayout*
- *CardLayout*
- *BoxLayout*
- *SpringLayout*
- *OverlayLayout*

U opštem slučaju dakle, komponente se smeštaju unutar kontejnera, a njegov pridruženi postupak razmeštanja (engl. *layout manager*) 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.

U narednom programu se panelu dodaju šest dugmadi koja se automatski razmeštaju po *FlowLayout* postupku, jer je to podrazumevani način za razmeštanje komponenti u svakom panelu.

LISTING 10.3: Program za ilustrovanje postupka *FlowLayout*.

```
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");

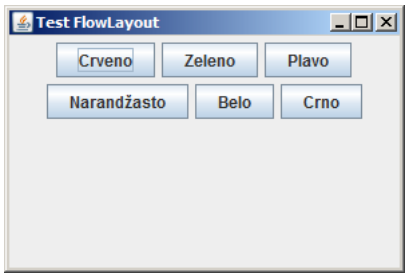
      // Konstruisanje panela za dugmad
      JPanel panel = new JPanel();

      // 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 10.6 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što možemo videti sa te slike, komponente su poravnate po sredini i prenose se u novi red kada nema više mesta u panelu.

Pored toga, ukoliko korisnik promeni veličinu okvira, komponente se automatski razmeštaju poštujući *FlowLayout* postupak. Ova činjenica



SLIKA 10.6: Panel sa šest dugmadi razmeštena po *FlowLayout* postupku.

je ilustrovana na slici 10.7.

SLIKA 10.7: Automatsko razmeštanje dugmadi zbog promene veličine okvira.

GridLayout. Ovim postupkom se kontejner deli u matricu polja po redovima i kolonama. Jedna komponenta se smešta u jedno od ovih polja 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 i tako dalje, onim redom kojim su dodate u kontejner.

U narednom programu se panelu dodaju šest dugmadi koja se u njemu razmeštaju po *GridLayout* postupku.

LISTING 10.4: Program za ilustrovanje postupka *GridLayout*.

```
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 10.8 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što možemo 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 se konstruiše (anonimni) 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 10.8: Panel sa šest dugmadi razmeštena po *GridLayout* postupku.

U slučaju postupka *GridLayout*, ukoliko se promeni veličinu okvira, matrični poredak komponenti se ne menja (tj. broj redova i kolona polja ostaje isti, kao i rastojanje između njih). Ova činjenica je ilustovana na slici 10.9.

Slika 10.9: 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.NORTH
- BorderLayout.CENTER
- BorderLayout.WEST
- BorderLayout.SOUTH

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.

U narednom programu se dodaje pet dugmadi u okvir koja se u njemu razmeštaju po *BorderLayout* postupku.

LISTING 10.5: Program za ilustrovanje postupka *BorderLayout*.

```
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 10.10 je prikazan okvir koji se dobija izvršavanjem ovog programa. Kao što možemo 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.

Slika 10.10: Panel sa pet dugmadi razmeštena po *BorderLayout* postupku.

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 izbaci 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, kao što je to ilustrovano na slici 10.11.

SLIKA 10.11: Promenom veličine okvira se komponente proširuju u dozvoljenom pravcu.

10.3 Definisanje komponenti

Svaka komponenta je odgovorna za sopstveno crtanje. To važi kako za standardne tako i za komponente koje programer definiše. Doduše, standardne komponente se samo dodaju odgovarajućem kontejneru, jer je za njih već definisano u biblioteci AWT/Swing kako se crtaju na ekranu.

Ukoliko je potrebno prikazati informacije na način koji se ne može izvesti unapred definisanim komponentama, moraju se definisati posebne komponente za sopstveno crtanje. To se radi definisanjem klase komponente koja je naslednica klase `JComponent`. Pri tome se mora nadjačati metod `paintComponent()` klase `JComponent` tako da ova nadjačana verzija obavlja potrebno crtanje u novoj komponenti:

```
public class MojaKomponenta extends JComponent {

   public void paintComponent(Graphics g) {

      // Naredbe za crtanje u komponenti
   }
}
```

Obratite pažnju na to da se prikazivanje informacija u nekoj komponenti izvodi jedino crtanjem u toj komponenti (čak i prikazivanje teksta se izvodi njegovim crtanjem). Ovo crtanje se obavlja u komponentinom metodu `paintComponent()` koji ima jedan parametar tipa `Graphics`. U stvari, kompletno crtanje u ovom metodu se mora izvoditi preko tog njegovog parametra tipa `Graphics`.

Klasa Graphics je apstraktna klasa koja u programu obezbeđuje grafičke mogućnosti nezavisno od fizičkog grafičkog uređaja. 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 bismo bolje razumeli svrhu klase Graphics, dobra analogija je ukoliko komponentu uporedimo sa listom papira i odgovarajući objekat tipa Graphics sa olovkom ili četkicom: metode klase Graphics (mogućnosti olovke ili četkice) možemo 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, recimo 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, sledeći program crta prigodan tekst i pravougaonik unutar okvira, što je prikazano na slici 10.12.

LISTING 10.6: Program za ilustrovanje 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);

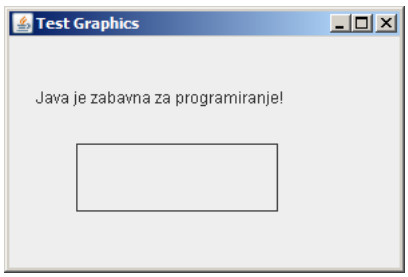
      // Konstruisanje komponente
      TestGraphicsKomponenta komp = new TestGraphicsKomponenta();
```

```
      // Smeštanje komponente u okvir
      okvir.add(komp);
      okvir.setVisible(true);
   }
}

class TestGraphicsKomponenta extends JComponent {

   public void paintComponent(Graphics g) {

      g.drawString("Java je zabavna za programiranje!", 20, 50);
      g.drawRect(50, 80, 150, 50);
   }
}
```



SLIKA 10.12: Rezultati nekih metoda klase Graphics.

Primetimo da se program sastoji od glavne klase TestGraphics i klase komponente MojaKomponenta. Definicija klase MojaKomponenta 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 MojaKomponenta. 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 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.

10.4 Klase za crtanje

Osnovna klasa za crtanje `Graphics` 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 ćemo govoriti samo o osnovama 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, od verzije Java 1.2 se metodu paintComponent() 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;

   // Metodi za crtanje primenjeni na objekat g2, a ne g
   . . .
}
```

Crtanje u biblioteci Java 2D se generalno zasniva na geometrijskim objektima u klasama iz paketa java.awt.geom koje implementiraju interfejs Shape. Među ovim klasama se nalaze 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 su definisani, između ostalog, 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 biblioteke 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 se koriste 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. Na primer, sledeće naredba dodele je pogrešna:

```
float x = 2.34;  // GREŠKA!
```

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:[3]

- Rectangle2D.Float
- Rectangle2D.Double

Kada se konstruiše pravougaonik tipa Rectangle2D.Float navode se koordinate njegovog gornjeg levog ugla i njegova širina i visina kao float

[3] To su zapravo statičke ugnježđene klase unutar apstraktne klase Rectangle2D da se ne bi koristila imena kao što su recimo FloatRectangle2D i DoubleRectangle2D.

vrednosti. Za pravougaonik tipa Rectangle2D.Double ove veličine se izražavaju kao double vrednosti:

```
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 o čemu smo govorili za pravougaonike važi potpuno isto i za ostale geometrijske oblike raspoložive 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. Dve njene klase-naslednice Point2D.Float i Point2D.Double služe za konstruisanje tačaka sa float i double (x, y) koordinatama:

```
Point2D t1 = new Point2D.Float(10F, 20F);
Point2D t2 = new Point2D.Double(10, 20);
```

Mnogi konstruktori i metodi imaju parametre tipa Point2D, jer je obično prirodnije u geometrijskim izračunavanjima koristiti tačke klase Point2D.

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 neka boja predstavlja pomoću tri broja iz intervala 0–255 koji određuju stepen crvene, zelene i plave boje u datoj boji.

Boja u Javi je objekat klase `Color` iz paketa `java.awt` i može se konstruisati navođenjem njene crvene, zelene i plave komponente:

```
Color mojaBoja = 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 neka 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:

```
Color mojaBoja = 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`) preko kojeg se crta u komponenti je 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);
```

gde je c objekat boje tipa Color. Ako želimo da crtamo, recimo, crvenom bojom u nekoj komponenti, onda bismo pisali:

```
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);
```

Svaka komponenta ima pridružene 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:

```
MojaKomponenta k = new MojaKomponenta();
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()) {  // obojiti pozadinu
      g2.setPaint(getBackground());
      g2.fillRect(0, 0, getWidth(), getHeight());
      // Za preciznije bojenje pravougaonika komponente:
      // g2.fill(new
      //    Rectangle2D.Double(0, 0, getWidth(), getHeight()));
      g2.setPaint(getForeground());
   }

   // Ostale naredbe za crtanje u komponenti
   . . .
}
```

Slično, boja lica komponente se može odrediti objektnim 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"
- "SansSerif"
- "Monospaced"
- "Dialog"
- "DialogInput"

"Serif" označava 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. "SansSerif" označava font u kojem

znakovi nemaju ove crtice. (Naslovi poglavlja i odeljaka ove knjige su na primer pisani ovakvim fontom.) "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.

Ova logička imena se uvek mapiraju na fontove koji zaista postoje na računaru. Na Windows sistemu recimo, fontu "SansSerif" odgovara stvarni font Arial.

Stil fonta se određuje koristeći statičke konstante koje su definisane u klasi Font:

- `Font.PLAIN`
- `Font.BOLD`
- `Font.ITALIC`
- `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. Veličina podrazumevanog 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);
```

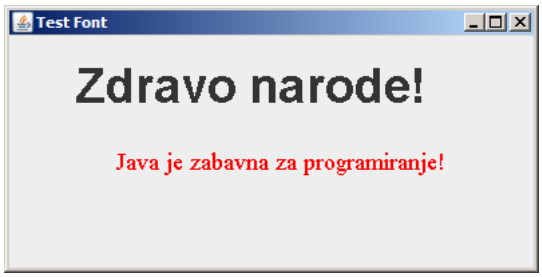
Grafički kontekst svake komponente ima pridružen font koji se koristi za prikazivanje (crtanje) teksta. Aktuelni font grafičkog konteksta se može menjati metodom `setFont()`. Obratno, vrednost aktuelnog fonta se može dobiti metodom `getFont()` koji vraća objekat tipa Font. Na primer, u sledećoj komponenti se prikazuje prigodan tekst čiji je izgled pokazan na slici 10.13:

```
class TestFontKomponenta extends JComponent {

   public void paintComponent(Graphics g) {

      Graphics2D g2 = (Graphics2D) g;

      Font sansBold36 = new Font("SansSerif", Font.BOLD, 36);
      g2.setFont(sansBold36);    // veliki font za tekst
```



SLIKA 10.13: Primer teksta u različitim fontovima.

```
        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

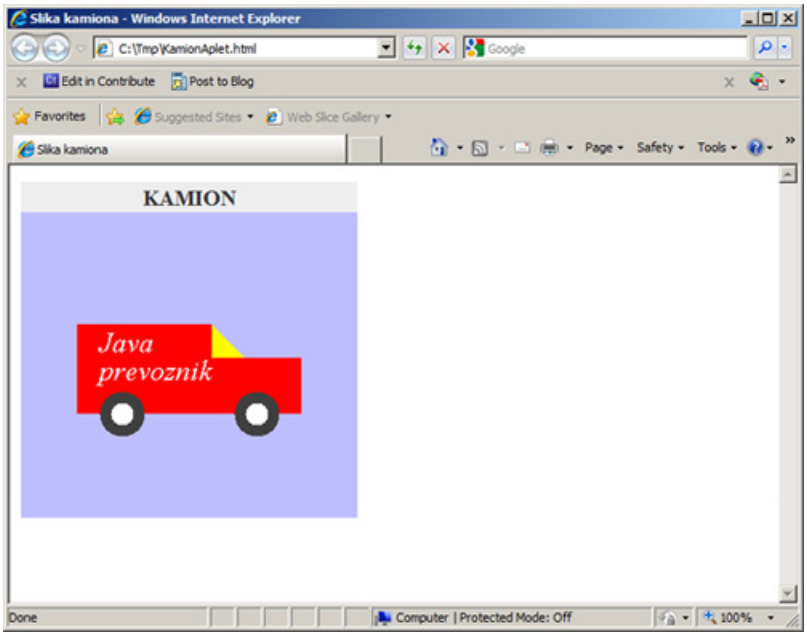
Aplet je grafički Java program koji izvršava brauzer (Internet Explorer, Mozilla, Safari, ...) prilikom prikazivanja neke veb strane. Naime, pored instrukcija za prikaz teksta, slika i drugog sadržaja, veb strane mogu imati instrukcije za izvršavanje specijalnog Java programa koji se popularno naziva aplet. Izvršavanje apleta unutar veb strane je omogućeno time što svi moderni brauzeri sadrže u sebi Java interpretator (JVM) pod čijom se kontrolom izvršava aplet.

Pisanje apleta nije mnogo drugačije od pisanja običnih grafičkih programa. 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()` služi dakle za inicijalizaciju vizuelne strukture apleta (kao i postupka rukovanja sa događajima) radi obezbeđivanja željene funkcionalnosti apleta. (Metod `init()` kod apleta odgovara u neku ruku metodu `main()` kod običnih programa.)

Da bismo ilustrovali način na koji se pišu apleti u Javi, u nastavku je predstavljen primer apleta za crtanje kamiona. Izgled veb strane sa ovim apletom je prikazan na slici 10.14.



SLIKA 10.14: Veb strana sa apletom za crtanje kamiona.

Slika kamiona u apletu se crta u klasi `Kamion` i predstavlja uobičajenu komponentu običnog grafičkog 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: naslov apleta kao instancu klase JLabel i sliku kamiona kao instancu klase Kamion.

LISTING 10.7: 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 od strane brauzera, pa pored pisanja samog apleta, u odgovarajućoj veb strani je potrebno navesti instrukcije za izvršavanje 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 tzv. <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 reći da se prethodnim primerom <applet> taga u prozoru za prikaz cele veb strane zauzima okvir veličine 300×300 piksela. Pritom se, kada brauzer prikazuje veb stranu, u tom okviru prikazuju rezultati izvršavanja Java bajtkoda koji se nalazi u datoteci `KamionAplet.class`.

10.5 Rukovanje događajima

Programiranje grafičkih programa se zasniva na događajima. Grafički program nema metod `main()` od kojeg počinje izvršavanje programa 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 može generisati korisnik svojim postupcima: pritiskom na taster tastature ili miša, pomeranjem miša i slično. Pored toga, grafičke komponente u prozoru programa mogu generisati 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 pritisne na standardnu komponentu dugme, generiše se *akcijski* događaj i konstruiše odgovarajući objekat klase pod imenom `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 tačno predviđenom metodu za određen tip događaja. Programer utiče na obradu događaja pisanjem ovih metoda koji definišu šta treba uraditi kada se događaj desi.

Objektno orijentisan model rukovanja događajima u Javi se zasniva na dva glavna koncepta:

1. **Izvor događaja.** Svaka grafička komponenta u kojoj se desio događaj je izvor događaja (engl. *event source*). 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()`.

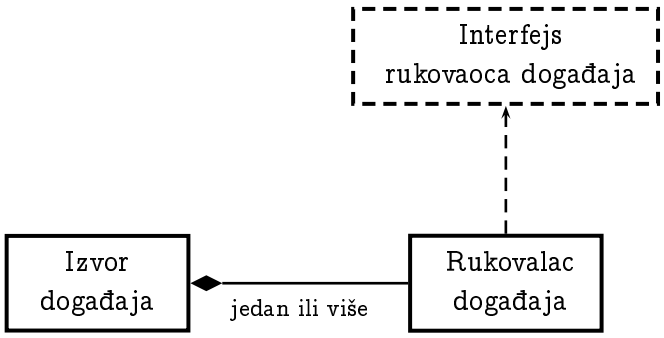
2. **Rukovalac događaja.** Rukovalac događaja (engl. *event listener*) je objekat koji sadrži poseban metod za obradu onog tipa događaja koji može da se desi u izvoru događaja. Da bi se taj metod u rukovaocu događaja zaista pozvao za obradu nastalog događaja, rukovalac događaja mora zadovoljiti dva uslova:
   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 događaja kako bi se znalo po nastanku događaja kome treba preneti konstruisani objekat događaja za obradu.

Opštu sliku rukovanja događajima u Javi možemo dakle opisati 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 stvarni događaj desi.
- Objekat rukovaoca događaja zatim koristi informacije iz dobijenog objekta događaja da bi odredio odgovarajuću reakciju na nastali događaj.

Na slici 10.15 je prikazan odnos između klasa i interfejsa za rukovanje događajima u Javi.

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



SLIKA 10.15: Odnos između klasa i interfejsa za rukovanje događajima.

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 ima samo jedan apstraktni metod pod imenom `ActionPerformed()`:

```
package java.awt.event;
public interface ActionListener extends java.util.EventListener {

   public void ActionPerformed(ActionEvent e);
}
```

Zbog toga rukovalac akcijskog događaja kojeg proizvodi dugme mora biti objekat klase koja implementira interfejs `ActionListener`:

```
public class MojRukovalac implements ActionListener {

   . . .
   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 MojRukovalac(...);
dugme.addActionListener(rukovalacDugmeta);
```

Nakon izvršavanja ovog programskog fragmenta, objekat na koga ukazuje rukovalacDugmeta se obaveštava svaki put kada se desi akcijski događaj u dugmetu, odnosno kada se tasterom miša pritisne na dugme. To znači da se konstruiše objekat događaja e tipa ActionEvent i poziva metod:

```
rukovalacDugmeta.ActionPerformed(e)
```

kome se kao argument prenosi novokonstruisani objekat događaja e.

Izvoru događaja može biti pridruženo više rukovalaca događaja. Za dugme na primer, tada bi se pozivao metod ActionPerformed() svih pridruženih rukovalaca akcijskog događaja koji nastaje kada se tasterom miša pritisne na dugme.

U Javi se koristi konvencija za imena klasa događaja i interfejsa rukovalaca događaja. Ime klase događaja je standardnog oblika <Ime>Event, a interfejsa rukovaoca odgovarajućeg događaja je oblika <Ime>Listener. Tako, akcijski događaj je predstavljen klasom ActionEvent i 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, imena metoda kojima se rukovaoc događaja pridružuje izvoru događaja radi obrade određenog tipa događaja, po konvenciji imaju standardni oblik add<Ime>Listener(). Zato, na primer, klasa Button sadrži metod:

```
public void addActionListener(ActionListener a);
```

koji smo koristitili u prethodnom primeru da bismo dugmetu pridružili rukovalac njegovog događaja.

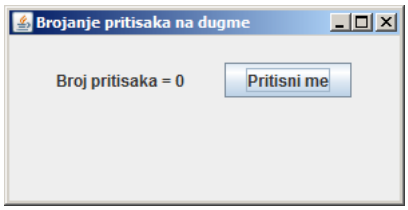
Bez sumnje, pisanje Java programa u kojima se reaguje na događaje obuhvata mnogo detalja. Navedimo zato osnovne postupke u glavnim crtama koji su neophodni za takav program:

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 (na primer, interfejs `ActionListener`).
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.
4. Objekat koji je instanca ove klase se mora pridružiti komponenti čiji generisani događaji se obrađuju. To se radi odgovarajućim metodom komponente kao što je `addActionListener()`.

Primer: brojanje pritisaka tasterom miša na dugme

Da bismo pokazali kompletan primer rukovanja događajima u Javi, sada ćemo napisati 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 10.16.

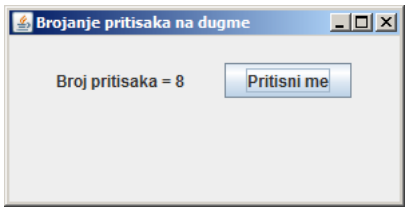


SLIKA 10.16: Početni prozor za brojanje pritisaka tasterom miša na dugme.

Uvek 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 izgleda kao što je to prikazano na slici 10.17.

Kada se tasterom miša pritisne na dugme, objekat koji obrađuje ovaj događaj treba da uveća polje `brojač` za jedan i da novi sadržaj tog polja prikaže promenom teksta komponente oznaka tipa `JLabel`:

```
class RukovalacDugmeta implements ActionListener {
```



SLIKA 10.17: Izgled prozora nakon 8 pritisaka tasterom miša na dugme.

```
    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 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; // 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);
      dugme.addActionListener(new RukovalacDugmeta());
   }
}
```

U konstruktoru klase DugmeBrojačOkvir se najpre određuje naslov, veličina i način razmeštanja komponenti glavnog okvira. Zatim se konstruišu komponente oznaka i dugme sa prigodnim tekstom i dodaju okviru. Na kraju, metodom addActionListener() se konstruisanom dugmetu pridružuje objekat rukovaoca njegovog akcijskog događaja koji je instanca 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 oznaka 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 u klasi DugmeBrojačOkvir. Postoji više rešenja ovog problema, od najgoreg da se ovo polje učini javnim, do najelegantnijeg koje je primenjeno u našem programu, da se RukovalacDugmeta definiše kao ugnježđena klasa u DugmeBrojačOkvir.

Ovo je inače česta primena ugnježđenih klasa o kojima smo govorili u odeljku 9.4. 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 sledećem listingu je dat kompletan program za brojanje pritisaka tasterom miša na dugme.

LISTING 10.8: Program za brojanje pritisaka na dugme.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

// Početna klasa
public class TestDugmeBrojač {

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

```
            DugmeBrojačOkvir okvir = new DugmeBrojačOkvir();
            okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            okvir.setVisible(true);
   }
}

// Glavni okvir sa dugmetom i oznakom
class DugmeBrojačOkvir extends JFrame {

   private JLabel oznaka; // oznaka u okviru

   // 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

generisanih događaja dugmeta. To znači da se RukovalacDugmeta može definisati da bude anonimna ugnježđena klasa. Prethodni program sa ovom izmenom postaje znatno kraći, i zato jasniji, kao što se može videti iz sledećeg listinga.

LISTING 10.9: Kraći program za brojanje pritisaka na dugme.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

// Početna klasa
public class TestDugmeBrojač {

   public static void main(String[] args) {

           DugmeBrojačOkvir okvir = new DugmeBrojačOkvir();
           okvir.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           okvir.setVisible(true);
   }
}

// 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
      dugme.addActionListener(new ActionListener() {
         // Anonimna klasa koja implementira 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:

```
package java.awt.event;
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 mouseEntered() i mouseExited() se pozivaju čim pokazivač miša uđe u pravougaonu oblast neke komponente i čim je napusti.

Obično se u programu ne mora voditi računa o svim događajima miša, mada se u klasi rukovalaca događaja koja implementira interfejs MouseListener moraju definisati svih pet metoda. Naravno, oni metodi koji nisu potrebni se definišu sa praznim telom metoda.

Drugi primer je komponenta okvir tipa JFrame koja je izvor događaja tipa WindowEvent. Klasa rukovalaca ovog događaja mora implementirati interfejs WindowListener koji sadrži sedam metoda:

```
package java.awt.event;
```

```
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 vraća iz minimizovanog stanja.

Ukoliko u programu hoćemo da korisniku omogućimo da se predomisli kada pokuša da zatvori glavni okvir programa, onda bismo trebali da definišemo klasu rukovalaca događaja tipa `WindowEvent` koja implementira interfejs `WindowListener`. U toj klasi bismo zapravo definisali samo odgovarajući metod `windowClosing()`, a ostale bismo trivijalno definisali sa praznim telom:

```
public class ZatvaračOkvira implements WindowListener {

   public void windowClosing(WindowEvent e) {

      if (korisnik se slaže)
         System.exit(0);
   }
   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 nasleđujemo adaptersku klasu WindowAdapter u prethodnom primeru klase ZatvaračOkvira, onda treba dakle nadjačati samo njen metod koji nas interesuje windowClosing():

```
public class ZatvaračOkvira extends WindowAdapter {

   public void windowClosing(WindowEvent e) {

      if (korisnik se slaže)
         System.exit(0);
   }
}
```

Objekat tipa ZatvaračOkvira se može 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 objekta tipa ZatvaračOkvira. Šest ovih 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.

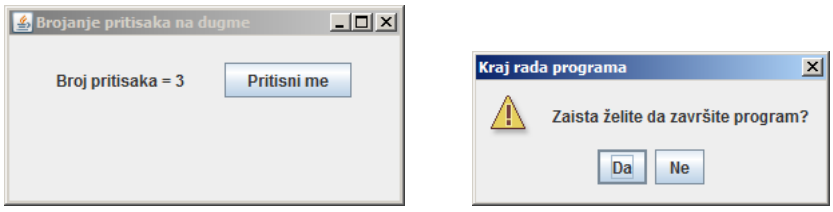
Ako se konstruiše samo jedan objekat klase ZatvaračOkvira u programu, ova klasa ne mora čak ni da se posebno definiše, nego može biti anonimna ugnježđena klasa:

```
okvir.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
```

```
        if (korisnik se slaže)
           System.exit(0);
     }
  });
```

Primer: uslovno zatvaranje glavnog okvira programa

Sada ćemo prethodni primer programa, koji broji pritiske tasterom miša na dugme, dopuniti mogućnošću 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 10.18, koji korisniku pruža šansu da odustane od zatvaranja glavnog okvira i da nastavi rad.



Slika 10.18: Okvir sa porukom za završetak rada programa.

Jedina izmena programa koji broji pritiske tasterom miša na dugme sastoji se u dodavanju rukovaoca događaja zatvaranja glavnog okvira. U programu se primenjuje postupak koji smo opisali u prethodnom odeljku za adaptersku klasu `WindowAdapter`. Glavnom okviru se pridružuje rukovalac događaja zatvaranja okvira predstavljen anonimnom ugnježđenom klasom. 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 operacija zatvaranja okvira mora izabrati da ne radi ništa:

```
okvir.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

To je zato što se ova operacija uvek izvodi nakon što neki rukovalac obradi događaj zatvaranja okvira, pa bi inače glavni okvir postao nevidljiv kada korisnik odustane od završetka programa. (Podrazumevana operacija zatvaranja svakog okvira je da se okvir učini nevidljivim.)

U sledećem listingu je dat kompletan program koji, pored brojanja pritisaka tasterom miša na dugme, omogućava korisniku da se predomisli kada pritisne na dugme X u gornjem desnom uglu glavnog okvira programa.

LISTING 10.10: Program sa uslovnim zatvaranjem glavnog okvira.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

// Početna klasa
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
      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)
               System.exit(0);
         }
      });
      okvir.setVisible(true);
   }
```

```
}

// 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);
      dugme.addActionListener(new ActionListener() {

         private int brojač; // brojač pritisaka na dugme

         public void actionPerformed(ActionEvent e) {
            brojač++;
            oznaka.setText("Broj pritisaka = " + brojač);
         }
      });
   }
}
```

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 Examples in a Nutshell*. O'Reilly, second edition, 2000.

[5] David Flanagan. *Java in a Nutshell*. O'Reilly, fifth edition, 2005.

[6] Mark Guzdial and Barbara Ericson. *Introduction to Computing and Programming in Java: A Multimedia Approach*. Prentice Hall, 2007.

[7] Cay S. Horstmann and Gary Cornell. *Core Java, Volume I–Fundamentals*. Prentice Hall PTR, eighth edition, 2007.

[8] Cay S. Horstmann and Gary Cornell. *Core Java, Volume II–Advanced Features*. Prentice Hall PTR, eighth edition, 2008.

[9] Ivor Horton. *Beginning Java 2*. Wiley Publishing, JDK 5 edition, 2005.

[10] Alison Huml, Kathy Walrath, and Mary Campione. *The Java Tutorial*. Addison-Wesley, third edition, 2000.

[11] Jonathan Knudsen and Patrick Niemeyer. *Learning Java*. O'Reilly, third edition, 2005.

[12] Daniel Liang. *Introduction to Java Programming, Comprehensive Version*. Prentice Hall, seventh edition, 2008.

[13] Richard F. Raposa. *Java in 60 Minutes a Day*. Wiley Publishing, 2003.

[14] Matthew Robinson and Pavel Vorobiev. *Java Swing*. Manning, second edition, 2003.

[15] Herbert Schildt. *Java Programming Cookbook*. McGraw-Hill, 2007.

Indeks