UNIVERZITET SINGIDUNUM



Dejan Živković

OSNOVE JAVA PROGRAMIRANJA

PRINCIPI I METODI PROGRAMIRANJA

Beograd, 2018.

UNIVERZITET SINGIDUNUM

Dejan Živković

OSNOVE JAVA PROGRAMIRANJA

Deveto izdanje

Beograd, 2018.

OSNOVE JAVA PROGRAMIRANJA

Autor:
dr Dejan Živković

Recenzent:
dr Dragan Cvetković

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

Za izdavača:
dr Milovan Stanišić

Tehnička obrada:
Dejan Živković

Dizajn korica:
Aleksandar Mihajlović

Godina izdanja:
2018.

Tiraž:
1000 primeraka

Štampa:
Mobid, Loznica

ISBN: 978-86-7912-558-3

Copyright:
© 2018. Univerzitet Singidunum
Izdavač zadržava sva prava.
Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Sadržaj

Spisak programa **iii**

Predgovor **v**

1 Uvod u Java programiranje **1**
1.1 Računari i programi 1
1.2 Java virtuelna mašina 5
1.3 Razvoj Java programa 8
Pitanja i zadaci 16

2 Uvod u Javu **19**
2.1 Kratak istorijski razvoj 19
2.2 Objekti i klase 20
2.3 Paketi i dokumentacija 24
2.4 Logička organizacija programa 27
2.5 Prvi Java program 29
Pitanja i zadaci 33

3 Osnovni elementi Jave **35**
3.1 Komentari 35
3.2 Imena 36
3.3 Tipovi podataka i literali 38
3.4 Promenljive 43
3.5 Konstante 48
3.6 Neke korisne klase 49
Pitanja i zadaci 63

4 Izrazi **69**
4.1 Prosti i složeni izrazi 69
4.2 Aritmetički operatori 70
4.3 Relacijski operatori 73
4.4 Logički operatori 75

4.5 Operator izbora 76
4.6 Operatori dodele 77
4.7 Prioritet operatora 78
Pitanja i zadaci 83

5 Složene naredbe **87**
5.1 Blok naredbi 87
5.2 Naredbe grananja 90
5.3 Naredbe ponavljanja 104
Pitanja i zadaci 123

6 Metodi **139**
6.1 Definisanje metoda 140
6.2 Pozivanje metoda 144
6.3 Vraćanje rezultata 149
6.4 Primeri metoda 151
6.5 Preopterećeni metodi 156
6.6 Lokalne i globalne promenljive 157
6.7 Rekurzivni metodi 165
Pitanja i zadaci 174

7 Klase **185**
7.1 Klase i objekti 186
7.2 Promenljive klasnog tipa 190
7.3 Konstrukcija i inicijalizacija objekata 196
7.4 Uklanjanje objekata 202
7.5 Skrivanje podataka (enkapsulacija) 204
7.6 Službena reč `this` 208
Pitanja i zadaci 211

8 Nizovi **223**
8.1 Jednodimenzionalni nizovi 223
8.2 Dvodimenzionalni nizovi 240
Pitanja i zadaci 251

Literatura **267**

Indeks **269**

Spisak programa

1.1 Program *Zdravo* 8
2.1 Program *Zdravo* 29
2.2 Program *Zdravo* 32
3.1 Pretvaranje Farenhajtovih stepena u Celzijusove 47
4.1 Vraćanje kusura 80
4.2 Izračunavanje rate za kredit 82
5.1 Sortiranje tri broja 96
5.2 Rad sa menijem 100
5.3 Određivanje sutrašnjeg datuma 101
5.4 Izračunavanje akumulirane štednje 107
5.5 Igra pogađanja broja 108
5.6 Prosek niza brojeva 110
5.7 Određivanje da li je broj prost 115
5.8 Prikazivanje tablice množenja 117
5.9 Određivanje da li je rečenica palindrom 122
6.1 Igra pogađanja broja 152
6.2 Određivanje da li je broj prost 153
6.3 Određivanje da li je rečenica palindrom 154
6.4 Igra pogađanja broja sa ograničenjem 159
6.5 Hanojske kule 173
7.1 Bacanja dve kocke 201
8.1 Prebrojavanje glasova na izborima 230
8.2 Argumenti programa u komandnom redu 233
8.3 Izvlačenje loto brojeva 236
8.4 Profit firme sa više prodavnica 246
8.5 Telefonski imenik 249

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

Predgovor

Programski jezik Java se koristi za rešavanje zadataka na računaru iz različitih oblasti ljudske delatnosti. Zato programi napisani u Javi mogu biti obični tekstualni ili grafički programi, ali i složenije aplikacije zasnovane na principima mrežnog programiranja, višenitnog programiranja, rada sa bazama podataka i mnogih drugih tehnologija. Obim i sadržaj ove knjige, međutim, prilagođeni su nastavnom planu i programu odgovarajućeg predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. 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 svima koji žele da se profesionalno bave Java programiranjem. Informacije o dodatnim nogućnostima Jave mogu se pronaći u literaturi koja je navedena na kraju knjige.

Java ima mnoge osobine svojstvene većini programskih jezika. Java će izgledati poznato C i C++ programerima, jer su mnoge programske konstrukcije preuzete iz tih jezika skoro bez izmena. Zbog toga je predznanje čitalaca iz ovih jezika velika olakšica za učenje Jave, ali to nije preduslov za uspešno praćenje ove knjige. Napomenimo i da knjiga ne predstavlja uvod u objektno orijentisano programiranje, iako je Java jedan objektno orijentisan programski jezik. U knjizi su ipak izloženi neki osnovni koncepti klasa i objekata kako bi se uspostavila neophodna terminologija.

Glavni cilj pisanja ove knjige je predstavljanje osnova Java programiranja na najjasniji način. U tekstu je stoga poštovano pedagoško načelo da izlaganje materijala iz svake oblasti treba da ide od poznatog ka nepoznatom i od konkretnog ka apstraktnom. Tekst je propraćen velikim brojem slika i urađenih primera kojima se konkretno ilustruju novouvedeni pojmovi. Primeri su birani tako da budu jednostavni, ali i da budu što realističniji i interesantniji. Njihova dodatna svrha je da posluže kao početna tačka za eksperimentisanje i dublje samostalno učenje. A kao što je to već uobičajeno za računarske knjige, za prikazivanje programskog teksta se koristi font sa znakovima `iste širine`.

Poglavlja u knjizi se završavaju odeljkom sa pitanjima i zadacima koji mogu poslužiti za proveru i utvrđivanje znanja iz odgovarajućeg poglavlja. Za svako pitanje su ponuđeni potencijalni odgovori od kojih treba izabrati one koji su tačni. (Neka pitanja imaju više tačnih odgovora.) Kod sastavljanja programskih zadataka je učinjen napor kako bi njihovo rešenje zahtevalo samo one elemente Jave koji su predstavljeni do odgovarajućeg poglavlja.

Zahvalnost. Moje veliko zadovoljstvo je što mogu da izrazim zahvalnost svima koji su mi pomogli u pripremi ove knjige. To se posebno odnosi na kolege koji su pročitali radne delove knjige i koji su svojim sugestijama pomogli da konačna verzija bude bolja. Zahvalnost dugujem i studentima za uočene greške u prethodnim izdanjima. Veoma bih bio zahvalan i drugim čitaocima na njihovom mišljenju o knjizi. Sve primedbe i pronađene greške (kao i pohvale) mogu se poslati na adresu `dzivkovic@singidunum.ac.rs`.

Dejan Živković
Beograd, Srbija
februar 2014.

UNIVERZITET PRIVATNI AKREDITOVANI SINGIDUNUM

1

Uvod u Java programiranje

Kada se laički govori o računarima, obično se podrazumeva da su to jednoobrazne mašine. Ali ako se posmatra njihova preciznija slika, mogu se prepoznati dve osnovne kategorije računara prema tome da li su računari namenjeni za opšte namene ili za specijalne primene. Pri tome, postoji više tipova računara opšte namene kao što su super računari, desktop računari, prenosivi računari i tako dalje. S druge strane, postoji i više tipova računara specijalne namene koji se ugrađuju u druge uređaje kao što su mobilni telefoni, automobili, proizvodne mašine i tako dalje. Ipak, bez obzira na namenu, zajedničko za sve tipove računara je to što svi računari rade na vrlo sličan način. U ovom poglavlju se uopšteno govori o tome kako rade računari i njihovi programi, a zatim se uvode osnovni elementi Java programiranja.

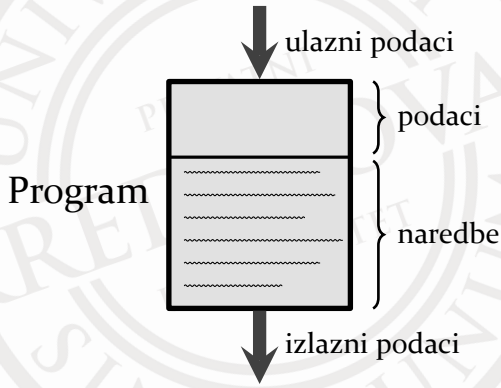
1.1 Računari i programi

Računar je elektronski uređaj koji se sastoji od *hardvera* i *softvera*. Pod hardverom se podrazumevaju fizički delovi računara u koje spadaju procesor, memorija, disk, tastatura, monitor i tako dalje. Softver obuhvata sve programe u računaru čijim izvršavanjem računar obavlja 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 mašina.

Osnovni princip rada računara je dakle izvršavanje *programa* čime se postiže odgovarajuća funkcionalnost računara. Program se sastoji od niza *naredbi* koje računar izvršava redom, jednu po jednu. Proces pisanja pro-

grama 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 (kao što se program sastoji od niza naredbi), a kuvar sprema jelo izvršavajući redom svaki korak recepta (kao što računar redom izvršava naredbe programa). Na kraju se kao rezultat rada kuvara dobija određeno jelo, recimo pica, što odgovara određenoj funkcionalnosti računara, 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"). Princip rada specijalnih računara ugrađenih u druge uređaje je sličan, osim što ti računari izvršavaju jedan isti program. (To bi odgovaralo priučenom kuvaru koji celog života pravi, na primer, sendviče po jednom receptu.) Grubi model računarskog programa je prikazan na slici 1.1.



Slika 1.1: Pojednostavljena slika računarskog programa.

Kao što se vidi sa slike 1.1, računarski program se ne sastoji samo od naredbi koje izvršava računar. Svaki program dodatno obuhvata memorijski prostor u kome 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 željenog sajta koji korisnik želi da poseti, zatim prostor u kome se nalaze razne poruke koje se prikazuju u toku rada, kao i prostor za tekst aktuelne veb strane koja se prikazuje.

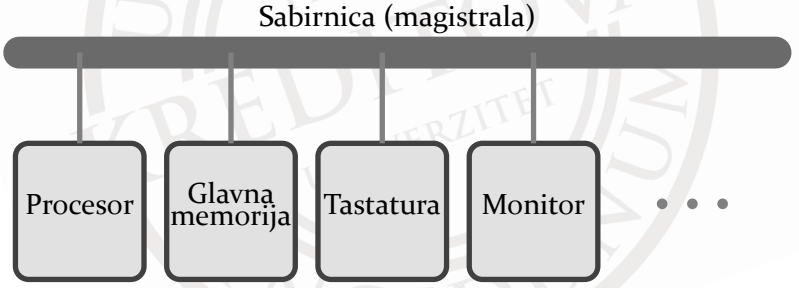
Podaci koje program mora dobiti od korisnika radi daljeg rada nazivaju se *ulazni podaci* programa. Ulazni podatak je, na primer, adresa željenog sajta u prethodnom primeru. Podaci koji predstavljaju rezultat rada programa dobijenog na osnovu ulaznih podataka nazivaju se *izlazni poda-*

ci programa. Izlazni podatak je, na primer, tekst prikazane veb strane u prethodnom primeru. Ostali podaci koji su neophodni za rad programa, a nisu ni ulazni ni izlazni, ponekad se nazivaju *radni podaci* programa. To su, na primer, razne poruke upozorenja programa za pregledanje Interneta u prethodnom primeru.

Za pisanje ispravnih programa treba dakle poznavati odgovore 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?

Pre nego što se može odgovoriti na ova pitanja, potrebno je imati opštu sliku o glavnim delovima hardvera računara i njihovim osnovnim funkcijama. Pojednostavljeni model hardvera svakog računara je prikazan na slici 1.2.



Slika 1.2: Funkcionalna šema računara.

Osnovni funkcionalni delovi svakog računara na slici 1.2 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 registara 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. Glavna 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 prenose u program iz spoljašnjeg okruženja računara, 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 računara 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 registara memorije u registre procesora, kao i obrnuto, zatim osnovne aritmetičke operacije nad registrima procesora, promena rednog 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 jedinice. Drugim rečima, operacije mašinskog jezika predstavljaju najprostije nizove nula i jedinica. Kako se i podaci kojima se manipuliše u programu predstavljaju binarnim brojevima, to znači da program koji procesor izvršava nije ništa drugo nego jedan vrlo dugačak niz nula i jedinica. Razlog zašto su naredbe i podaci u računaru predstavljeni u binarnom obliku je to što se nule i jedinice mogu lako elektronski predstaviti prekidačkim kolima: uključen prekidač je zamena za jedinicu i isključen prekidač je zamena za nulu.

Rad svakog računara je dakle zasnovan na izvršavanju programa koje se izvodi po vrlo jednostavnom modelu:

- Glavna memorija računara sadrži naredbe i podatke programa na mašinskom (binarnom) jeziku.
- Naredbe programa, od prve naredbe pa redom kako to logika programa zahteva, prenose se jedna za drugom iz glavne memorije u procesor, gde se izvršavaju odmah nakon prenošenja.

- Ovaj postupak se obavlja mehanički, odnosno naredbe se „slepo" izvršavaju tačno onako kako su napisane, bez ikakvog razumevanja šta naredbe znače ili da li imaju smisla. Ovo zapravo znači da su računari obične mašine koje ne poseduju nikakvu inteligenciju.

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 baš na tom jeziku. Iako je teoretski moguće napisati program koristeći mašinski jezik, to je praktično neizvodljivo i vrlo 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 još mnogi drugi.

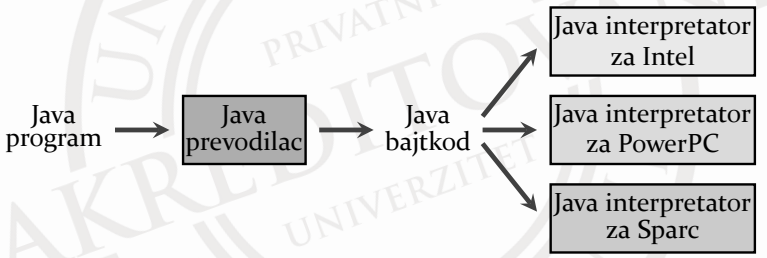
Programi napisani na nekom jeziku visokog nivoa ne mogu se direktno izvršiti na računaru, nego se moraju najpre prevesti na mašinski jezik određenog procesora. Srećom, to prevođenje ne obavljaju ljudi, već to rade specijalni računarski programi koji se zovu *prevodioci* (ili *kompajleri*). Prevodilac dakle na ulazu dobija program na jeziku visokog nivoa i prevodi ga u ekvivalentni izvršni program na mašinskom jeziku. Ovaj izvršni program na mašinskom jeziku može se zatim izvršiti proizvoljan broj puta, pod uslovom naravno da se to radi na računarima sa odgovarajućim tipom procesora. Da bi se program izvršio na računarima sa drugim tipom procesora, program se mora ponovo prevesti na odgovarajući mašinski jezik koristeći drugi prevodilac.

Alternativa prevođenju programa na jeziku visokog nivoa u mašinski jezik radi izvršavanja programa jeste 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, radi izvršavanja programa, interpretator redom uzima po jednu naredbu programa na visokom nivou, određuje neophodne mašinske operacije za njeno izvršavanje i konačno izvršava te mašinske operacije. Obratite pažnju na to da ako se po logici programa neka naredba ne izvršava, ona se nikad neće ni prevesti. S 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 ko-

ji je prethodno preveden na mašinski jezik.

U stvari, interpretator ne mora da „razume" baš neki jezik visokog nivoa, već može interpretirati samo mašinski jezik drugog tipa procesora. U tom slučaju se 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) može se izvršiti mašinski program za Macintosh računare (sa PowerPC procesorom) uz pomoć interpretatora na PC-ju koji interpretira mašinski jezik PowerPC procesora. Tada se obično govori 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. Ova šema je prikazana na slici 1.3.



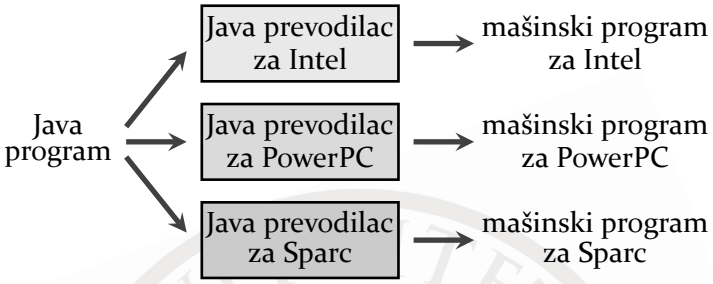
Slika 1.3: Prevođenje i izvršavanje Java programa.

Kao što se vidi sa slike 1.3, 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*. Java program se dakle prevodi u mašinski jezik JVM koji se ne može direktno izvršiti na stvarnom računaru. Za izvršavanje Java programa prevedenog u Java bajtkod potreban je interpretator Java bajtkoda na računaru na kome treba izvršiti Java program.

Naravno, 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 je uopšte potreban posredni Java bajt-

kod, odnosno mašinski jezik virtuelnog računara? Drugim rečima, zašto se obično ne koriste Java prevodioci za mašinski jezik različitih tipova stvarnih procesora, jer bi se onda prevedeni Java program mogao izvršiti na odgovarajućem računaru, čak i brže? Ova klasična šema je prikazana na slici 1.4.



Slika 1.4: Klasični način prevođenja i izvršavanja Java programa.

Postoji više razloga zašto nije izabran klasičan pristup za izvršavanje Java programa. Glavni razlog leži u značajno većoj kompleksnosti Java prevodioca od Java interpretatora. Naime, prevodilac za Javu je vrlo složen program, jer je i sama Java složen programski jezik visokog nivoa. Nasuprot tome, 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.

Drugi razlog je to što se mnogi Java programi automatski izvršavaju nakon što se preuzmu preko Interneta. 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 i to 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, kombinacijom Java jezika i Java bajtkoda omogućeno je 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 bi se ovo moglo primeniti za Javu, moraju se najpre upoznati neki tehnički detalji razvoja Java programa.

Postupak razvoja Java programa obuhvata četiri koraka: pisanje, unošenje, prevođenje i ispravljanje Java programa. Prvi korak pisanja (smišljanja) Java programa je najkreativniji i najteži u njegovom razvoju. Zbog toga je ovom najvažnijem koraku posvećen ostatak cele ove knjige, a u ovom odeljku se govori o preostala tri koraka koristeći kao radni primer jednostavan program *Zdravo* koji je prikazan u listingu 1.1 u nastavku. 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.");
   }
}
```

Tri poslednja koraka u razvoju Java programa — unošenje, prevođenje i ispravljanje programa — izvode se rutinski na računaru. Doduše, to je često mukotrpan postupak koji zahteva puno strpljenja. On 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 slične mogućnosti za druge operativne sisteme), programerima na raspolaganju stoje mnoga grafička okruženja, kako besplatna tako i komercijalna. Pošto se zbog brojnosti razvojnih okruženja ovde ne može dati potpun opis svih njih, u daljem izlaganju se ograničavamo na operativni sistem Windows. Pored toga, od grafičkih okruženja govori se samo o besplatnom razvojnom okruženju NetBeans. To ne znači da se time gubi mnogo na kompletnosti, jer sva okruženja rade na vrlo sličan način i osnovni koncepti su zajednički za sve. Zbog toga, ako treba nije teško preći na neko drugo okruženje ukoliko se poznaje jedno od njih.

Ali bez obzira na način na koji se razvijaju Java programi, na računaru se najpre mora instalirati programski jezik Java. To je srećom lako izvodljivo, jer se sve u vezi Jave može besplatno preuzeti sa zvaničnog sajta `www.java.com`. Programski jezik Java, njegov kompajler, interpretator i ostali pomoćni alati nalaze se u jednom paketu pod nazivom Java SE JDK (najnovija verzija je 7). Instalacija ovog neophodnog softvera je jednostavna — svodi se na prihvatanje ponuđenih opcija i prelaska na sledeće korake dok se ne završi instalacija.

Tekstualno razvojno okruženje

Za unošenje teksta Java programa u tekstualnom okruženju može se koristiti bilo koji tekstualni editor. To može biti, na primer, najprostiji 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 u tekst programa dodaju svoje specijalne znakove za formatiranje teksta.

Nakon što se tekst Java programa unese i sačuva u datoteci na disku, 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`.

Tačan postupak za radni program *Zdravo* (ili za bilo koji drugi Java program uz odgovarajuće izmene) sastoji se dakle od korišćenja dva uslužna programa, editora i DOS-a:

1. Koristeći bilo koji editor treba pažljivo upisati tekst programa *Zdravo* i sačuvati ga u datoteci pod nazivom `Zdravo.java`. Ime datoteke koja sadrži tekst nekog Java programa nije proizvoljno i mora biti sastavljeno od prefiksa i sufiksa razdvojenih tačkom. Prefiks imena datoteke (u ovom slučaju `Zdravo`) mora tačno odgovarati imenu klase

koja se nalazi u datoteci (to je ime koje se nalazi iza reči `class` u tekstu programa). Sufiks (ili ekstenzija) imena datoteke mora biti `java`. Obratite pažnju na to da je ovo opšte pravilo i da važi za svaki Java program.

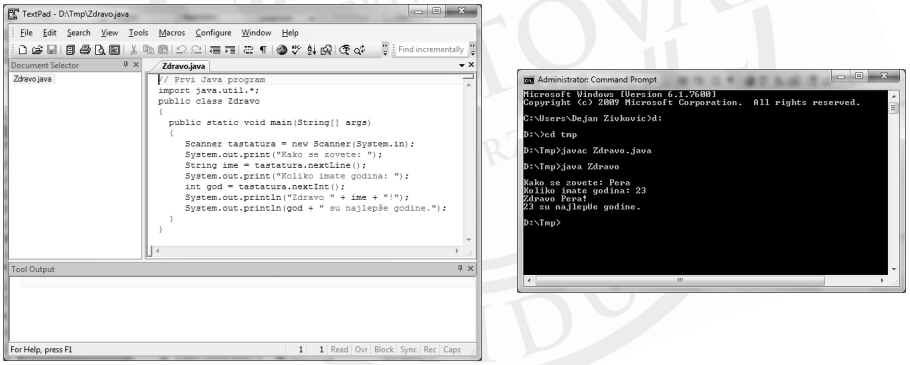
2. Koristeći DOS treba najpre promeniti radni direktorijum u prozoru DOS-a da bude onaj u kome se nalazi datoteka `Zdravo.java`. Zatim program *Zdravo* u datoteci `Zdravo.java` treba prevesti na mašinski jezik JVM komandom:

```
javac Zdravo.java
```

Rezultat prevođenja programa *Zdravo* je njegov bajtkod koji se dobija u datoteci `Zdravo.class`. Na kraju, dobijeni bajtkod programa *Zdravo* treba izvršiti (interpretirati) komandom:

```
java Zdravo
```

Ovaj postupak je ilustrovan na slici 1.5 na kojoj su prikazani prozori editora i DOS-a u slučaju programa *Zdravo*.



Slika 1.5: Prozori editora i DOS-a za program *Zdravo*.

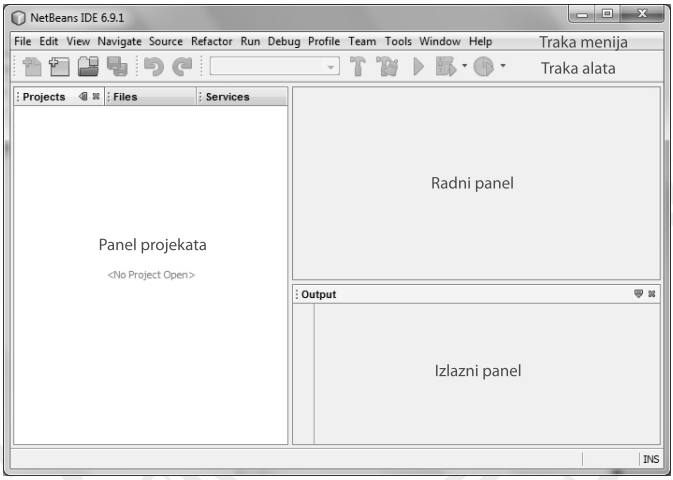
Ako prevođenje ili interpretiranje programa nije bilo uspešno, to znači da je pogrešno unet tekst programa *Zdravo*. Greške treba ispraviti u prozoru editora, sačuvati izmene u istoj datoteci `Zdravo.java` i ponoviti komande za prevođenje i izvršavanje programa u prozoru DOS-a.

Grafičko razvojno okruženje

NetBeans IDE je integrisano grafičko razvojno okruženje koje je namenjeno razvoju različitih vrsta programa. U nastavku se na primeru progra-

ma *Zdravo* pokazuju samo najosnovnije mogućnosti NetBeans-a za razvoj Java programa. To ipak ne treba da zavara čitaoce, jer NetBeans pruža sve što je potrebno za razvoj složenih i profesionalnih aplikacija. NetBeans je besplatan softver koji se može preuzeti sa sajta `www.netbeans.org` i instalirati na jednostavan način.

Osnovne mogućnosti NetBeans-a. Glavni prozor NetBeans-a je po svom izgledu vrlo sličan prozorima drugih grafičkih okruženja za razvoj programa na drugim programskim jezicima (slika 1.6).



Slika 1.6: Glavni prozor NetBeans-a.

Na vrhu glavnog prozora NetBeans-a se nalaze traka menija i traka alata, dok preostali deo prozora dele tri panela u kojima se prikazuju informacije relevantne za aktuelni program koji se razvija. Na levoj strani se nalazi panel projekata, a na desnoj strani se nalaze radni panel i ispod njega izlazni panel. Namena pojedinih delova glavnog prozora NetBeans-a su:

- Traka 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 datoteka, kao i za ostale standardne aktivnosti u vezi sa datotekama od kojih se sastoji Java program.

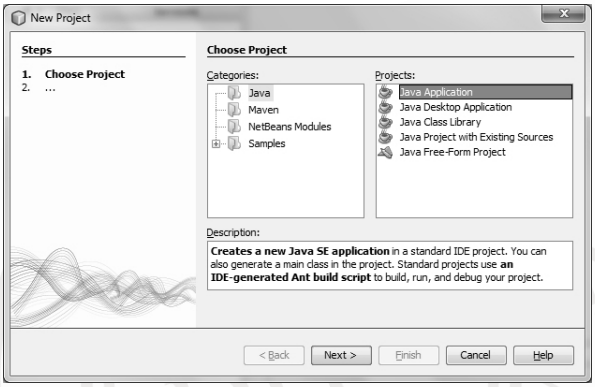
  - Meni *Edit* ima potpuno istu funkciju za rad sa tekstom programa kao istoimeni meniji u standardnim editorima (na primer: *undo*, *redo*, *copy*, *paste*, *find*, *replace*).
  - Meni *Build* je namenjen za prevođenje Java programa koji se sastoji od jedne datoteke ili više njih.
  - Meni *Run* je namenjen za izvršavanje prevedenog programa. Pored toga, iz ovog menija se može pokrenuti debager za otklanjanje grešaka u programu, kao i za testiranje i detaljno praćenje stanja promenjivih u toku rada programa.

- Traka alata sadrži ikone alata kojima je omogućen 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.
- U radnom panelu se prikazuje sadržaj datoteka koje su izabrane u jednoj od kartica panela projekata. Sadržaj ovih datoteka u radnom panelu se može menjati, jer se one automatski otvaraju u integrisanom editoru NetBeans-a.
- U izlaznom panelu se prikazuju sve važne poruke tokom prevođenja programa, kao i ulazno/izlazni podaci samog programa tokom njegovog izvršavanja.

Formiranje projekta. Kao i u drugim grafičkim okruženjima, osnovna jedinica rada u NetBeans-u 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 datoteka, dokumentaciji i slično. Svi ovi podaci se smeštaju u direktorijumu projekta koji se pravi na osnovu datog imena projekta. O svemu ovome vodi računa uglavnom sâm NetBeans, 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 odgovarajućeg šablona, projekat se otvara u glavnom prozoru NetBeans-a tako što se njegova logička struktura prikazuje u kartici *Projects*, a struktura datoteka u kartici *Files* panela projekata. Pored toga, u radnom panelu se otvara glavna datoteka projekta koja sadrži početni izvorni tekst Java programa.

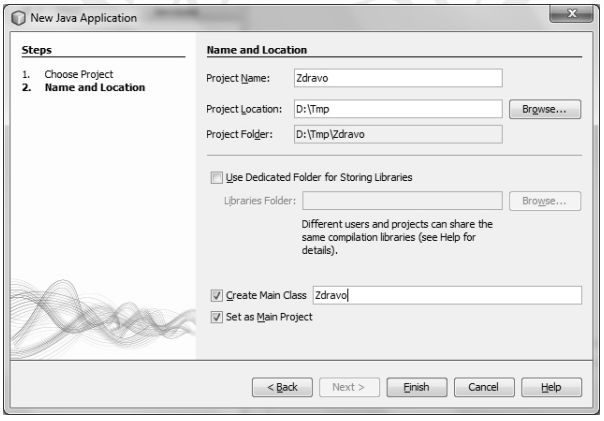
Tačan postupak formiranja novog projekta u NetBeans-u na primeru radnog programa *Zdravo* sastoji se od nekoliko koraka. Pri tome se pritiskom na dugme *Next* odgovarajućeg prozora u svakom koraku prelazi na sledeći korak:

1. U glavnom prozoru NetBeans-a iz trake menija treba izabrati opciju *File* > *New Project*.
2. U otvorenom prozoru *New Project* treba izabrati kategoriju *Java* i zatim opciju *Java Application* (slika 1.7).



Slika 1.7: Izbor kategorije novog projekta.

3. U otvorenom prozoru *New Java Application* treba upisati podatke o novom projektu (slika 1.8).

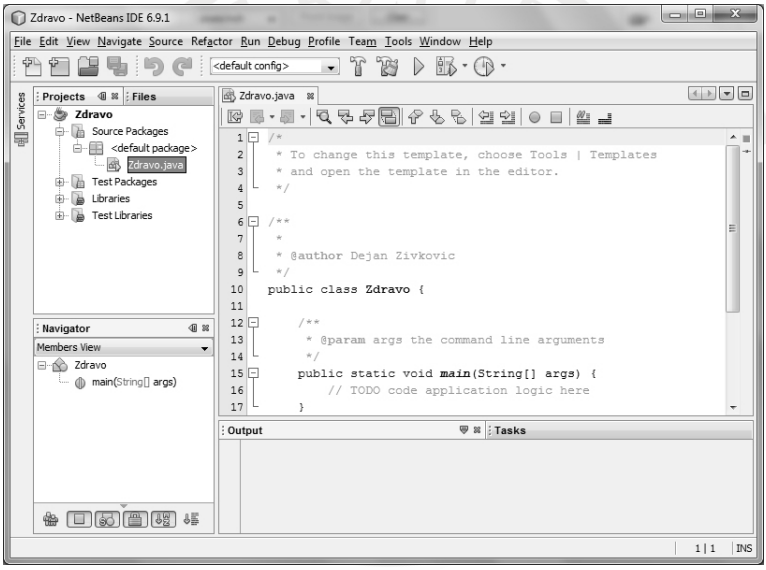


Slika 1.8: Upisivanje podataka za novi projekat.

Pri tome,

- u polju *Project Name* treba upisati ime novog projekta, recimo `Zdravo`;
- u polju *Project Location* treba izabrati direktorijum u kome će se nalaziti direktorijum formiranog novog projekta (u ovom primeru je to `D:\Tmp`);
- u polju *Create Main Class* treba upisati `Zdravo` za ime glavne klase programa.

Nakon pritiska na dugme *Finish*, završava se formiranje projekta Zdravo i on se otvara u glavnom prozoru NetBeans-a (slika 1.9).



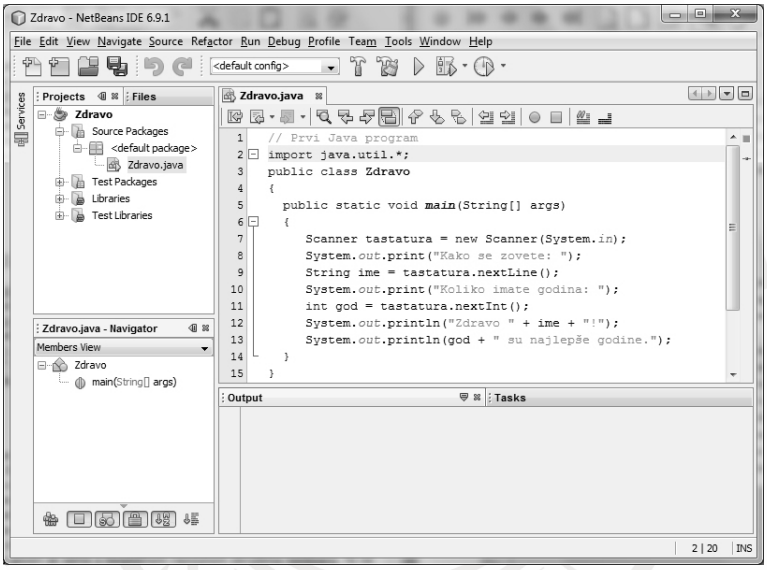
Slika 1.9: Izgled novog projekta za program *Zdravo*.

Pri tome,

- hijerarhijska struktura delova novog projekta (datoteke i paketi izvornog teksta, biblioteke koje su potrebne i tako dalje) prikazuje se u panelu projekata;
- kako je izabrano polje *Create Main Class*, formira se kostur glavne klase `Zdravo` u datoteci `Zdravo.java` i ova datoteka se otvara u radnom panelu pod kontrolom integrisanog editora;

- struktura članova klase iz radnog panela prikazuje se u panelu navigatora u kome se izborom pojedinih elemenata može lako prelaziti sa jednog elementa na drugi u izvornom tekstu klase radnog panela.

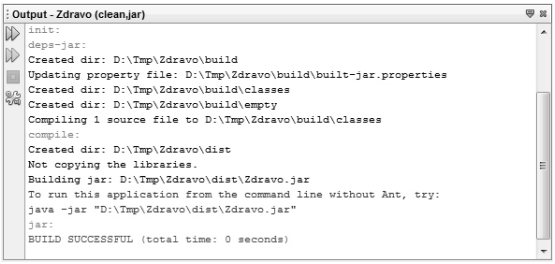
Unošenje izvornog teksta programa. Tekst koji je početno generisan u datoteci `Zdravo.java` u radnom panelu treba zameniti tekstom programa *Zdravo* (slika 1.10). Promene u datoteci `Zdravo.java` mogu se sačuvati biranjem opcije *File* > *Save*.



Slika 1.10: Unošenje teksta programa *Zdravo*.

Prevođenje programa. Za prevođenje unetog Java programa treba izabrati opciju *Build* > *Build Main Project* iz trake menija u glavnom prozoru NetBeans-a. (Drugi način za prevođenje Java programa jeste izbor odgovarajuće ikone iz trake alata.) Poruke o statusu postupka prevođenja mogu se videti u izlaznom panelu. U konkretnom slučaju prevođenja programa *Zdravo*, sadržaj izlaznog panela je prikazan na slici 1.11.

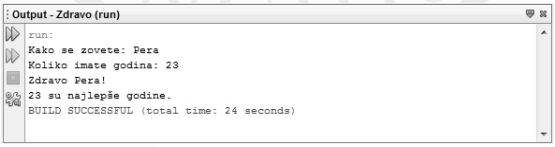
Ako je prevođenje programa uspešno završeno, što se prepoznaje po poslednjoj poruci BUILD SUCCESSFUL, program se dalje može izvršiti. U suprotnom, ako se rezultat prevođenja završava porukom BUILD FAILED, program sadrži greške koje se moraju ispraviti. Greške se u izlaznom panelu prikazuju u obliku hiperveza, a njihovim praćenjem se može preći u



Slika 1.11: Status prevođenja programa u izlaznom panelu.

izvorni tekst radnog panela na mestu odgovarajuće greške. Na ovaj ili drugi način se moraju ispraviti sve greške u programu i zatim se program mora ponovo prevesti. Ovaj ciklus — ispravljanje grešaka, prevođenje programa — mora se ponoviti sve dok prevođenje programa ne bude uspešno. U primeru programa *Zdravo*, rezultat njegovog uspešnog prevođenja je bajtkod tog programa u datoteci `Zdravo.class`.

Izvršavanje programa. Uspešno preveden program se izvršava izborom opcije *Run > Run Main Project* iz trake menija u glavnom prozoru NetBeans-a. (Izvršavanje prevedenog Java programa može se obaviti i izborom odgovarajuće ikone iz trake alata.) Program se naravno može izvršiti više puta, recimo radi testiranja sa promenjenim ulaznim podacima. Ulazno/izlazni podaci tokom izvršavanja programa se prikazuju u izlaznom panelu glavnog prozora. Jedan primer izvršavanja programa *Zdravo* je prikazan na slici 1.12.



Slika 1.12: Rezultat izvršavanja programa *Zdravo*.

Pitanja i zadaci

1. Kako se nazivaju fizički delovi od kojih se sastoji računar (procesor, memorija, disk, tastatura, monitor, ...)?

A. Softver B. Hardver C. Operativni sistem D. Windows

2. Kako se nazivaju svi programi u računaru čijim izvršavanjem računar obavlja korisne zadatke za ljude?

A. Softver B. Hardver C. Operativni sistem D. Windows

3. Šta je „mozak" računara?

A. Hardver B. Procesor (CPU) C. Memorija D. Disk

4. Od čega se sastoji računarski program, pojednostavljeno rečeno?

A. Bajtova B. Kontrola C. Naredbi i podataka D. Teksta

5. Koliko bitova ima jedan bajt?

A. 4 B. 8 C. 16 D. 32

6. Na kom jeziku moraju biti napisani programi da bi računar mogao da ih izvrši?

A. Mašinskom jeziku B. Engleskom jeziku C. Pseudo jeziku D. Jeziku visokog nivoa

7. Šta prevodi programe napisane na programskom jeziku visokog nivoa u programe na mašinskom jeziku?

A. Operativni sistemi B. Procesori C. Ljudi D. Prevodioci (kompajleri)

8. Kako se naziva program preveden na mašinski jezik Java virtuelne mašine?

A. Java bajtkod B. Java objektni kod C. Java aplet D. Java aplikacija

9. Kako se mora nazvati datoteka u kojoj se nalazi sledeći Java program koji se sastoji od jedne klase `Test`?

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

A. Test.txt B. Test.class C. Test.java D. Main.java E. Main.txt

10. Kojim sufiksom se završava ime datoteke u kojoj se nalazi preveden Java program (Java bajtkod)?

A. java B. obj C. class D. exe

11. Koja DOS komanda služi za prevođenje Java programa koji se nalazi u datoteci Test.java?

A. `javac Test.java` B. `compile Test.java` C. `prevedi Test.java` D. `javap Test.java`

12. Koja DOS komanda služi za izvršavanje (interpretiranje) prevedenog Java programa u datoteci Test.class?

A. `javac Test` B. `izvrsi Test` C. `java Test` D. `java Test.class`

13. Koja grafička razvojna okruženja služe za pisanje Java programa?

A. NetBeans B. DOS C. Windows D. DrJava E. Linux

2

Uvod u Javu

U ovom poglavlju se, nakon kratkog prikaza istorijskog razvoja programskog jezika Java, uvode osnovni koncepti objektno orijentisanog programiranja pošto se na njima zasnivaju skoro sve programske konstrukcije u Javi. Zatim se govori o opštoj organizaciji programa u Javi, a na kraju se kroz jedan mali primer programa ilustruju neki osnovni elementi jezika Java.

2.1 Kratak istorijski razvoj

Programski jezik Java je nastao 1990. godine u kompaniji Sun Microsystems kao pilot-projekat jednostavnog jezika za programiranje „pametnih" kućnih uređaja (mobilnih telefona, TV uređaja, raznih plejera i slično). Kao što je to često slučaj, od početne zamisli do prve realizacije jezika cirkulisalo je nekoliko probnih verzija i usput je dolazilo do promena početnih ciljeva. Java je zvanično objavljena 1995. godine i istovremeno je izdata njena besplatna implementacija. Prva verzija je dalje pretrpela mnogobrojne izmene, tako da je danas aktuelna sedma verzija jezika (Java 7). Međutim, ni to nije kraj jer se Java i dalje aktivno razvija i poboljšava.

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*) koje se koriste uz sam osnovni 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 personalnim računarima se najčešće koristi Java SE. Edicija Java ME je namenjena pisanju programa za uređaje sa smanjenim hardverskim mogućnostima, a Java EE je namenjene najzahtevnijim programima sa distribuiranim, servisno orijentisanim režimom rada.

Jedan od početnih ciljeva jezika Java je bio olakšano Internet programiranje. Zbog toga je ustanovljena posebna terminologija za kategorije Java programa koji koriste Internet tehnologije i obično se razlikuju tri vrste Java programa:

- **Aplikacija.** Pod tim se podrazumeva samostalni uobičajeni program. Takav program se izvršava (interpretira) pod kontrolom interpretatora Java virtuelne mašine na računaru.
- **Aplet.** To je program koji se ugrađuje u neku veb stranu (slično tekstu, slikama, zvuku i ostalom sadržaju) i izvršava ga veb čitač na računaru (na primer, Internet Explorer). Ono što izdvaja ove programe je njihova automatska distribucija prilikom otvaranja veb strane, kao i njihove ograničene mogućnosti zbog bezbednosti.
- **Servlet i JSP (Java Server Pages).** To je program za dinamičko generisanje delova veb strana nekog sajta, a izvršava ga veb server tog sajta.

Treba naglasiti da je sav softver u vezi sa Javom beplatan od samog početka. To je umnogome uticalo na popularnost samog jezika, a doprinelo je i 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 `www.java.com`.

2.2 Objekti i klase

Java je objektno orijentisan jezik, stoga pisanje Java programa zahteva poznavanje osnovnih pojmova objektno orijentisanog programiranja kao što su objekti i klase. U ovom odeljku se ukratko govori o ovim konceptima objektnog načina razmišljanja kojih nema u proceduralnom programiranju s kojim su čitaoci možda već upoznati.

Objektno orijentisan pristup programiranju se zasniva na manipulisanju *objektima*. To znači da se ostvarivanje programske logike postiže defi-

nisanjem klasa raznih objekata i međusobnim dejstvom tih objekata. Primeri stvarnih objekata su auto, zgrada, bankovni račun, student i slično, odnosno sve stvari koje poseduju neki identitet i nad kojima se može izvoditi neka aktivnost u određenoj situaciji. Ove maglovite ideje je najbolje potkrepiti jednim primerom iz svakodnevnog života, jer se u objektno orijentisanoj metodologiji teži ka rešavanju problema simuliranjem prirodnog načina koji se primenjuje u životnim situacijama za rešavanje problema.

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 dejstvom realizuje rad stvarnog restorana.

Iz ovog primera se vidi da postoje razni tipovi objekata. Njihova imena (gost, kelner, jelovnik, račun, ...) ljudima otprilike govore šta je uloga objekata, ali računari ne znaju šta se podrazumeva pod tim imenima. Da bi se to opisalo za računare, definišu se *klase* objekata. Klasa u Javi opisuje koje podatke svi objekti klase imaju i šta takvi objekti mogu da urade. Svaki objekat jedne klase ima iste atribute (podatke koji ga obeležavaju) i iste mogućnosti (procedure 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.

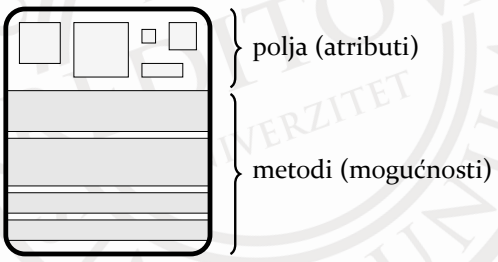
Obratite pažnju na to da može postojati više objekata iste klase. Restoran može imati dva kuvara, tri kelnera, dvadeset jela na meniju, četiri gosta u datom trenutku i tako dalje. Zašto restoran nema samo jednog zaposlenog, jer bi u principu sve poslove kuvara, šankera i kelnera mogao obavljati jedan 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 vrste poslova kako bi se oni izvodili efikasnije i bolje. Slično tome, objektno orijentisani programi distribuiraju zadatke na više klasa objekata kako bi programi bili efikasniji i bolji. Pored toga, na taj način se programi mogu lakše menjati i poboljšavati.

Šta je objekat?

Stvarni objekat u najopštijem smislu je entitet iz realnog sveta sa svojim jedinstvenim identitetom. Tako student, sto, stolica, taster na tastaturi,

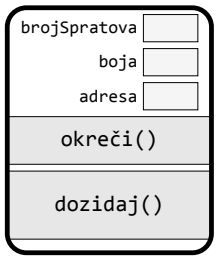
telefon i tako dalje mogu se smatrati objektima. Ali pored ovih „fizičkih" stvari, pojam objekta obuhvata i apstraktne stvari kao što su krug, kredit, ispit i tako dalje. Jedinstven identitet stvarnog objekta je određen obeležjima (atributima) i ponašanjem (mogućnostima) objekta. Jedan student posmatran kao objekat može se, na primer, identifikovati po imenu i prezimenu, fakultetu koji studira, broju indeksa i slično, a njegove mogućnosti su da polaže ispite iz određenih predmeta, da učestvuje u radu studentskih organizacija i slično.

Softverski objekti u računaru oponašaju stvarne objekte po tome što imaju pridružene atribute i mogućnosti. Atributi određuju šta su obeležja objekta, a mogućnosti određuju šta se može uraditi sa objektom. Atribute koje objekti sadrže se predstavljaju *poljima* (promenljivim), a njihove mogućnosti se predstavljaju *procedurama*. Procedure se u objektno orijentisanoj terminologiji nazivaju *metodima*, stoga se u nastavku knjige isključivo koristi ovaj termin. Najopštija struktura jednog softverskog objekta u programu je ilustrovana na slici 2.1.



Slika 2.1: Softverski objekat.

Objekti se u programu ne opisuju pojedinačno, nego se čitava kolekcija objekata istog tipa opisuje definisanjem *klase* tih objekata. Na primer, ako je reč o programu u kome se obrađuje nešto u vezi sa zgradama nekog naselja, onda se te zgrade mogu prirodno opisati jednom klasom, recimo pod nazivom `Zgrada`, čiji (softverski) objekti predstavljaju stvarne zgrade u naselju. Svaki objekat-zgrada klase `Zgrada` može imati polja, recimo, `brojSpratova`, `boja` i `adresa` čije vrednosti obeležavaju konkretnu zgradu. Pored toga, svaki takav objekat može imati metode, recimo, `okreči()` i `dozidaj()` s kojima se može nešto uraditi sa konkretnom zgradom. Softverski model jednog objekta-zgrade klase `Zgrada` je prikazan na slici 2.2.



Slika 2.2: Struktura objekta klase `Zgrada`.

Šta je klasa?

Klasa je opis nekih objekata sa zajedničkim svojstvima. Jednom klasom se određuje identitet njenih objekata pomoću polja (promenljivih) i metoda (procedura) koje poseduju svi objekti te klase. Klasa drugim rečima definiše šablon kako izgleda struktura pojedinih objekata klase.

Vrlo je važno razumeti da se u Java programu definišu (pišu) klase, a objekti se ne definišu nego se konstruišu na osnovu tih klasa. Dobra analogija pojmova klase i objekata u programiranju je ono što su tehnički nacrt zgrade i izgrađene zgrade u građevinarstvu:

- Tehnički nacrt zgrade za građevinare je ono što je definicija klase za programere.
- Izgrađena zgrada na osnovu tehničkog nacrta za građevinare je ono što je konstruisan objekat na osnovu klase za programere.

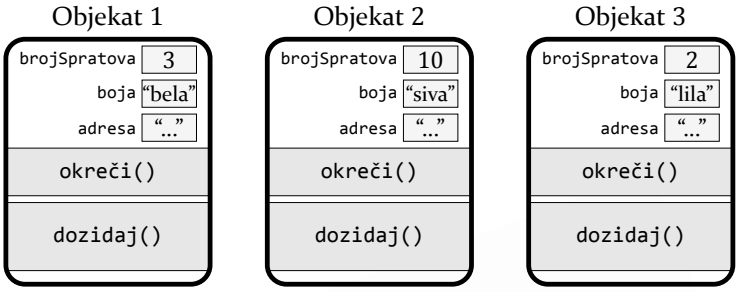
Pored toga, kao što se na osnovu jednog tehničkog nacrta zgrade može izgraditi više zgrada, tako se i na osnovu jedne klase može konstruisati više objekata te klase. Na primer, ako se posmatra primer klase `Zgrada` iz prethodnog odeljka, njena definicija u Javi ima otprilike sledeći oblik:

```
class Zgrada
{
   int brojSpratova;
   String boja;
   String adresa;

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

Detalji definicije ove klase za sada nisu važni, već je bitno to da se na osnovu nje može konstruisati više objekata klase `Zgrada`. Ako se konstrui-

šu, recimo, tri objekta te klase sa njihovim individualnim podacima, onda je njihov izgled u memoriji računara ilustrovan 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 polja i metoda te klase. To je glavna karakteristika objekata svake klase i jedna od konceptualnih odlika koja objektno orijentisano programiranje razlikuje od proceduralnog programiranja. (Prava slika objekata je ipak malo složenija, ali je za sada dovoljna njihova pojednostavljena interpretacija.)

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 olakšala njihova upotreba, sve klase Java platforme su dalje grupisane u *pakete*, analogno organizaciji datoteka i direktorijuma u okviru sistema datoteka na disku. Paket u Javi je dakle biblioteka klasa koje su namenjene jednoj oblasti primene i koje zato čine funkcionalnu celinu. Neki od osnovnih paketa su:

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

Ostale pakete ne vredi nabrajati, a pogotovo nije izvodljivo opisati namenu svih klasa u njima. (Ukupno ima preko 200 paketa i 4000 klasa.) 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 što pre naviknu da koriste zvanični opis svih klasa i paketa Java platforme. Ovaj opis u elektronskoj formi može se bes-

platno preuzeti sa sajta `www.java.com`. Dokumentacija je napisana u formatu pogodnom za čitanje u nekom veb čitaču, tako da se jednostavnim izborom hiperveza u tekstu može brzo pronaći ono što se traži.

Korišćenje paketa

Druge klase u definiciji nove klase u Javi mogu se koristi na jednostavan način samo ukoliko pripadaju istom paketu kao i nova klasa. Klasa iz nekog drugog paketa mora se koristiti pisanjem punog imena te klase koje se sastoji od dva dela, imena paketa i prostog imena klase, razdvojenih tačkom. Na primer, za rad sa datumima u Java programu je korisna klasa `Date` koja se nalazi u paketu `java.util`. Ta klasa se dakle u programu može bez ograničenja koristiti pod svojim punim imenom `java.util.Date`. Recimo, ako se definiše klasa `NovaKlasa` koja koristi objekat klase `Date`, onda se može pisati:

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

Upotreba punog, dugačkog imena klase zahteva više pisanja, povećava mogućnost slovnih grešaka i smanjuje čitljivost programa. Zbog toga u Javi postoji mogućnost „uvoženja" pojedinih klasa na početku programa da bi se iza toga u programu moglo koristiti prosto ime klase bez imena paketa u kome se nalazi. Ova mogućnost se koristi pisanjem deklaracije `import` na početku teksta klase. Tako, umesto punog imena `java.util.Date` u prethodnom primeru, može se kraće pisati:

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

Ako treba „uvesti" 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 može se zato ekvivalentno napisati na sledeći način:

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

U stvari, deklaracija `import` sa džoker-znakom je najčešći način njene primene u Java programima. Primetimo da se time ne povećava veličina programa, odnosno sve klase iz nekog paketa se fizički ne dodaju programu. Deklaracija `import` ukazuje samo u kojem paketu treba pronaći korišćene klase, što znači da se fizički dodaju samo one klase koje se zaista koriste u tekstu definicije nove klase.

U retkom slučaju kada dve različite klase iz dva različita paketa imaju isto ime, skraćivanje imena obeju ovih klasa nije moguće. Onda se ime jedne klase može pisati u skraćenom obliku „uvoženjem" te klase, bilo direktno ili putem džoker-znaka, dok se za drugu klasu mora koristiti puno ime.

Paket `java.lang` sadrži neke najosnovnije klase koje su potrebne svakom Java programu. Zbog toga se on ne mora direktno „uvoziti", već se to radi automatski time što se podrazumeva kao da se na početku teksta svake klase nalazi deklaracija `import java.lang.*`.

Definisanje paketa

Svaka klasa u Javi mora pripadati određenom paketu. Kada se definiše nova klasa, ako se drugačije ne navede, podrazumeva se da ta klasa pripada jednom paketu koji nema ime. To je takozvani *anonimni paket* kojem automatski pripadaju sve klase koje nisu eksplicitno dodate nekom imenovanom paketu.

Ukoliko novu klasu treba dodati imenovanom paketu, koristi se deklaracija `package` kojom se određuje paket kome pripada nova klasa. Ova deklaracija se navodi na samom početku teksta klase, čak pre deklaracije `import`. Na primer, ako klasa `NovaKlasa` treba da bude deo paketa `nekipaket`, to se definiše na sledeći način:

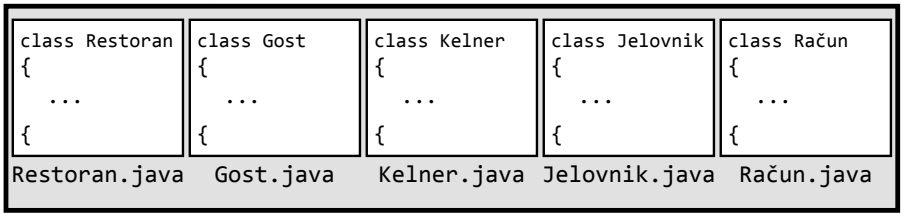
```
package nekipaket;
import java.util.*;
class NovaKlasa
{
   . . .
}
```

Klasa koja pripada imenovanom paketu zahteva posebnu organizaciju datoteke sa tekstom te klase, kao i drugačiji postupak njenog prevođenja i izvršavanja. Tako, u prethodnom primeru, datoteka `NovaKlasa.java` koja sadrži tekst klase `NovaKlasa` mora se nalaziti u posebnom direktorijumu čije ime mora biti `nekipaket`. U opštem slučaju, ime posebnog direktorijuma mora biti isto kao ime paketa kojem klasa pripada. Pored toga, prevođenje i izvršavanje klase `NovaKlasa` mora se obaviti iz direktorijuma na prvom prethodnom nivou iznad direktorijuma `nekipaket`. O svim ovim detaljima se u grafičkom razvojnom okruženju automatski brine odgovarajući softver, dok u tekstualnom razvojnom okruženju programer mora sâm voditi računa o ovome.

2.4 Logička organizacija programa

Da bi se moglo razumeti Java programiranje, moraju se poznavati još neka osnovna pravila 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 programa mogu postojati izvan definicije klase.

Na logičkom nivou, Java program se sastoji od jedne klase ili više njih. Razmotrimo konkretan primer Java programa koji simulira rad nekog restorana. Pretpostavimo da smo u tom programu predvideli pet klasa čiji objekti međusobnim dejstvom obezbeđuju programsko simuliranje rada restorana. To su klase, recimo, `Restoran`, `Gost`, `Kelner`, `Jelovnik` i `Račun`. Ako se podsetimo da se izvorni tekst svake klase piše u posebnoj datoteci čije ime mora biti isto kao ime klase koja se definiše, kao i da ekstenzija imena te datoteke mora biti `java`, onda logička struktura ovog programa izgleda kao na slici 2.4.



Slika 2.4: Java program od pet klasa za simuliranje rada restorana.

Ceo program se dakle sastoji od pet datoteka, jer definicije klasa koje učestvuju u programu treba da se nalaze u posebnim datotekama:

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

Postupak prevođenja i izvršavanja ovog programa zavisi od razvojnog okruženja u kome se program piše. U tekstualnom razvojnom okruženju, na primer, svih pet klasa se moraju prevesti u Java bajtkod pomoću Java prevodioca. Kao što je poznato iz odeljka 1.3, komanda za to je `javac`. U sledećoj tabeli su prikazane komande za prevođenje ovih klasa i datoteke sa odgovarajućim bajtkodom koje se time dobijaju.

Kompajliranje	Dobijeni bajtkod
`javac Restoran.java`	`Restoran.class`
`javac Gost.java`	`Gost.class`
`javac Kelner.java`	`Kelner.class`
`javac Jelovnik.java`	`Jelovnik.class`
`javac Račun.java`	`Račun.class`

Na kraju, izvršavanje celog programa se sastoji od interpretiranja bajtkoda logički početne klase programa. Ako u prethodnom primeru pretpostavimo da je to, recimo, klasa `Restoran`, onda se komandom Java interpretatora

```
java Restoran
```

započinje izvršavanje programa. Naravno, ovim se počinje interpretiranje bajtkoda klase `Restoran`, a prema logici programa se onda po potrebi u odgovarajućem trenutku interpretiraju bajtkodovi ostalih klasa programa. Primetimo da se kao argument Java prevodioca mora navesti puno ime datoteke sa ekstenzijom `java`, dok se kao argument Java interpretatora *ne* sme navesti ekstenzija `class` datoteke koja sadrži bajtkod odgovarajuće klase.

U slučaju kada se program piše u grafičkom razvojnom okruženju primenjuje se sličan postupak, ali na pojednostavljen način zavisno od konkretnog softvera. Obično je zapravo dovoljno samo pokrenuti izvršavanje projekta (izborom opcije *Run Main Project*), a prevođenje i interpretiranje svih potrebnih klasa se zatim automatski obavlja.

2.5 Prvi Java program

U ovom odeljku se posvećuje malo više pažnje programu *Zdravo* koji je korišćen kao radni primer u prethodnom poglavlju za opis razvojnih okruženja. U tom programu se mogu prepoznati elementi Jave o kojima smo govorili u prethodnom delu ovog poglavlja. Pored toga, kroz dodatna objašnjenja pokušaćemo i da bolje razumemo taj program. Mada dodatne uvodne napomene o jeziku Java doprinose „raščišćavanju magle" oko programa *Zdravo*, ipak i dalje ne treba očekivati da će baš sve biti jasno o tom programu — naizgled trivijalan, program koristi neke elemente o kojima se detaljno govori tek u narednim poglavljima ove knjige. Ali pre svega, da bi se u nastavku program *Zdravo* mogao lakše pratiti, u listingu 2.1 se ponavlja 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 rada pojedinih delova programa drugim programerima. Bez komentara je vrlo teško razumeti kako radi neki program, a to je naravno neophodno za one koji moraju da izvrše neke izmene u programu. Pisanje dobrih i kratkih komentara na pravom mestu u programu je odlika vrhunskih programera i nikako se ne sme olako shvatiti.

Iza reda sa komentarom u programu *Zdravo* uočava se deklaracija `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 pisanja. Klasa `System` služi za povezivanje ulaznih podataka programa sa standardnim ulazom (tastaturom) i izlaznih podataka programa sa standardnim izlazom (monitorom).

Sve iza deklaracije `import` u programu *Zdravo* je definicija jedine klase od koje se taj program sastoji. Podsetimo se da se u opštem slučaju Java program sastoji od više klasa, ali ovaj jednostavni program predstavlja najprostiji 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 polja i metodi koje sadrže svi objekti definisane klase.

U programu *Zdravo* je dakle 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 pomoću Java interpretatora. Da bi to bilo moguće, klasa mora da sadrži specijalni metod (proceduru) pod nazivom `main()` u obliku:

```
public static void main(String[] args)
{
   .
   . // Niz naredbi u telu metoda
   .
}
```

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()` izvršava. Izvršavanje metoda `main()` indirektno počinje kada se komandom

```
java Zdravo
```

pokrene izvršavanje programa *Zdravo*. Preciznije, ovom komandom počinje interpretiranje bajtkoda klase `Zdravo` automatskim pozivom njenog metoda `main()` za izvršavanje. A to znači da se onda redom izvršavaju naredbe

u telu tog metoda.

U telu metoda `main()` klase `Zdravo` 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 se detaljno govori u poglavlju 5.

Postupak koji se primenjuje u metodu `main()` klase `Zdravo` predstavlja tipični način za interaktivni ulaz/izlaz programa. Pošto se sličan pristup može iskoristiti i u drugim programima, u nastavku se ukratko objašnjavaju naredbe kojima je to realizovano u ovom primeru. U Javi se podaci u tekstualnom prozoru najčešće prikazuju dvema naredbama:

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

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

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

prikazuje se tekst `Kako se zovete:␣` na ekranu od trenutne pozicije kursora i kursor se postavlja neposredno iza znaka ␣. (Znakom ␣ je označen znak razmaka koji se ne vidi.) Slično, naredbom

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

prikazuje se odgovarajući tekst u zagradi, ali se i kursor pomera na početak sledećeg reda ekrana.

Učitavanje ulaznih podataka programa u Javi nije tako jednostavno i zahteva primenu objektno orijentisanih tehnika programiranja. Zato se prvom naredbom u metodu `main()` klase `Zdravo`, odnosno naredbom

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

operatorom `new` najpre konstruiše objekat klase `Scanner` povezan sa *standardnim ulazom* programa. Standardni ulaz programa u Javi je predstavljen objektom `System.in` koji obezbeđuje apstraktni model učitavanja podataka preko fizičkog uređaja tastature. Zbog toga je novo konstruisani objekat nazvan `tastatura` u programu. Primenom raznih metoda klase `Scanner` na ovaj objekat mogu se učitavati vrednosti odgovarajućeg tipa podataka. 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 istaći da je Java programski jezik *slobodnog formata*. To znači da ne postoje posebna ograničenja na izgled samog teksta programa, odnosno ne postoje posebni zahtevi u vezi sa načinom na koji se pišu pojedine sintaksne konstrukcije 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.

Na primer, isti program *Zdravo* na strani 29 prikazan je u nastavku u listingu 2.2 u nečitljivom obliku, ali je potpuno ispravan sa gledišta Java prevodioca.

Listing 2.2: 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.");}}
```

Naravno, ova „ispravna" verzija programa *Zdravo* je potpuno neprihvatljiva za ljude! Slobodan format jezika Java ne treba zloupotrebljavati, već ga treba iskoristiti radi pisanja programa čiji vizuelni oblik jasno odražava njegovu logičku strukturu. Pravila dobrog stila programiranja zato nalažu da se piše jedna naredba u jednom redu, da se uvlačenjem i poravnanjem redova označi blok naredbi koje čine jednu logičku celinu i, generalno, da se struktura programa učini što jasnijom.

Uz pridržavanje ovih preporuka i upotrebom komentara, razumljivost komplikovanog programa se može znatno poboljšati. Obratite pažnju na to da je ovo važno ne samo za one koji nisu pisali program, već i za autora programa, jer posle kratkog vremena će i sâm autor brzo zaboraviti način na koji je neki problem prvobitno rešen. Zato ukoliko je kasnije potrebno menjati stari program, treba se lako podsetiti i ponovo tačno razumeti kako on radi.

Pitanja i zadaci

1. Kako se u OOP zovu programske jedinice sa zajedničkim svojstvima koje opisuje jedna klasa?

A. Promenljive B. Procedure C. Izrazi D. Objekti

2. Kako se u OOP zovu programske jedinice kojima pripadaju objekti?

A. Promenljive B. Procedure C. Klase D. Nizovi

3. Koje su dve glavne karakteristike svakog objekta u OOP?

A. Obeležja (atributi) i mogućnosti (ponašanje) objekta.
B. Sastavni delovi i izgled objekta.
C. Broj i oblik objekta.
D. Veličina i način upotrebe objekta.

4. Kako se zovu programske jedinice u kojima su grupisane raspoložive klase Java platforme?

A. Datoteke B. Paketi C. Moduli D. Folderi

5. Kojom deklaracijom se nekom programu priključuju raspoložive klase Java platforme?

A. `import` B. `export` C. `module` D. `package`

6. Koje su od ovih rečenica o paketima tačne?

A. Po konvenciji, u Javi se imena paketa pišu svim malim slovima.
B. Deklaracija `package` nije obavezna.
C. Deklaracija `import` nije obavezna.
D. Klase u paketu `java.lang` se automatski priključuju svakom programu.

7. Zašto je važan dobar stil programiranja?

A. Program se neće prevesti zato što je napisan lošim stilom.
B. Program će se brže izvršavati zato što je napisan dobrim stilom.
C. Program će biti razumljiviji zato što je napisan dobrim stilom.
D. Program će imati verovatno manji broj grešaka zato što je napisan dobrim stilom.
E. Program će se lakše menjati zato što je napisan dobrim stilom.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

3

Osnovni elementi Jave

Od ovog poglavlja se sistematski predstavljaju sastavne konstrukcije programskog jezika Java i počinje se sa izučavanjem osnovnih programskih elemenata koji su neophodni za pisanje kompletnih programa. Pošto slični fundamentalni koncepti postoje u svakom programskom jeziku, čitaoci koji poznaju neki drugi programski jezik ovo poglavlje će lakše razumeti.

3.1 Komentari

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

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

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

Druga vrsta komentara sadrži tekst koji se prostire u više redova. Ovi komentari počinju simbolom `/*`, obuhvataju tekst u proizvoljnom broju

redova i završavaju se simbolom `*/`. Ceo tekst između simbola `/*` i `*/` potpuno se zanemaruje od strane Java prevodioca. Na primer:

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

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

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

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

3.2 Imena

Programiranje se u svojoj suštini svodi na davanje i korišćenje imena. Jedno ime u programu može označavati razne pojmove, na primer, memorijsku lokaciju ili 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 zvanični termin i često se nazivaju *identifikatori*.

Imena u programu 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:

- `tastatura`
- `brojStudenata`
- `PERICA`
- `x157`
- `Zdravo`
- `broj_studenata`
- `nnn`
- `jedno_VRLO_dugačko_ime`

Treba naglasiti da razmaci nisu dozvoljeni u imenima. Tako, recimo, ispravno ime u Javi je `broj_studenata`, ali `broj studenata` nije ispravno ime. Velika i mala slova se razlikuju u Javi tako da se `Zdravo`, `zdravo`, `ZDRAVO` i `zdRaVO` smatraju različitim imenima. Izvesne reči su predviđene za specijalnu svrhu u Javi i one se 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 Jave su u tekstu ove knjige obično naglašene na poseban način (drugom bojom ili tamnijim slovima).

Pored ovih formalnih pravila koje propisuje sâm jezik Java, postoje preporuke o tome kako treba birati dobra imena u programima. Pre svega, imena treba da budu smislena kako bi se na osnovu njih moglo lako zaključiti šta označavaju. U programu *Zdravo* na strani 29, na primer, objekat za učitavanje ulaznih podataka je nazvan `tastatura`, jer to zaista i predstavlja, mada se mogao zvati i `xy` ili `bzvz`. Davanje smislenih imena znatno doprinosi boljoj razumljivosti programa.

Radi bolje razumljivosti, Java programeri se za pisanje imena pridržavaju još nekih neformalnih konvencija. Tako, imena klasa uvek počinju velikim slovom, imena promenljivih i metoda počinju malim slovom, a imena konstanti se pišu sa svim velikim slovima. (O ovim programskim elementima se opširnije govori u nastavku knjige.) Ako se ime sastoji od nekoliko reči, recimo `brojStudenata`, onda se svaka reč od druge (a i prva reč ako se radi o klasi) piše s početnim velikim slovom. U pisanju imena ne treba koriste donju crtu, osim u nekim retkim slučajevima.

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. Jedan primer ove vrste imena korišćen je u programu *Zdravo*: `System.out.println`. Složena imena se ponekad nazivaju i *kvalifikovana imena*, a notacija kojom se pišu se naziva *tačka-notacija*.

[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.

Složena imena su potrebna zato što u Javi neki elementi mogu da obuhvataju druge elemente. Složeno ime onda određuje putokaz do nekog elementa kroz jedan ili više nivoa obuhvatanja. Tako ime `System.out.println` ukazuje da nešto pod nazivom `System` sadrži nešto pod nazivom `out`, koje dalje sadrži nešto pod nazivom `println`.

3.3 Tipovi podataka i literali

Naredbe programa manipulišu podacima koji se nalaze u delu programa predviđenom za njihovo smeštanje. U Java programima se mogu koristiti podaci koji se dele u dve glavne kategorije:

- Podaci poznatih tipova koji su unapred „ugrađeni" u jezik. Ova kategorija tipova podataka se naziva *primitivni tipovi podataka*.
- Podaci novih tipova koje programer mora sâm da definiše. Ova kategorija tipova podataka se naziva *klasni tipovi podataka*.

Primitivnim tipovima podataka su obuhvaćene osnovne vrste podataka 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. Zbog toga 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, odnosno brojevima bez decimala kao što su 17, -1234 ili 0. Celobrojni tipovi `byte`, `short`, `int` i `long` 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 -2^7 do $2^7 - 1$ (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^{15} do $2^{15} - 1$ (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^{31} do $2^{31} - 1$ (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^{63} do $2^{63}-1$ (ili od −9223372036854775808 do 9223372036854775807).

Ove detaljne činjenice se naravno ne moraju znati napamet, nego su navedene samo da se dobije predstava o magnitudi celih brojeva sa kojima se može raditi u programu. U najvećem broju slučajeva se koristi tip `int` kao srednje rešenje koje pokriva većinu namena. Ostali celobrojni tipovi se koriste u specijalnim primenama, na primer kada je potrebno štedeti na memoriji ili kada su mogući ogromni brojevi u programu.

Tipovi `float` i `double` služe za rad sa realnim brojevima, odnosno brojevima sa decimalama kao što su 1.69, −23.678 ili 5.0. Sem u posebnim slučajevima kada je važna ušteda memorijskog prostora ili je potrebna velika tačnost izračunavanja, u programima se obično koristi tip `double` za rad sa realnim brojevima. Tipovi `float` i `double` 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):

- U tipu `float` se koriste četiri bajta za binarni zapis realnih brojeva. Vrednosti ovog tipa su realni brojevi u opsegu približno $\pm 10^{38}$ i mogu imati oko 8 značajnih cifara. Posledica ovog ograničenja broja značajnih cifara je da bi ovim tipom brojevi, recimo, 32.3989231134 i 32.3989234399 bili predstavljeni istim brojem 32.398923.
- U tipu `double` se koristi osam bajtova za binarni zapis realnih brojeva. Vrednosti ovog tipa su realni brojevi u opsegu približno $\pm 10^{308}$ i mogu imati oko 15–16 značajnih cifara.

Tip `char` služi za rad sa alfabetskim znakovima i koristi dva bajta za njihov binarni zapis. Vrednosti ovog tipa su pojedinačni znakovi kao što su mala i velika slova (`A`, `a`, `B`, `b`, ...), cifre (`0`, `1`, ..., `9`), znakovi punktuacije (`%`, `&`, `?`, razmak, ...) i neki specijalni znakovi (za novi red, za tabulator, ...). Standard po kojem se ovi znakovi predstavljaju u binarnom obliku naziva se *Unicode*, a ovaj način je izabran u Javi zbog internacionalizacije. Naime, od samog nastanka, cilj jezika Java je bio pisanje programa za razne govorne jezike, a ne samo za engleski. Standard Unicode je idealan sa tog aspekta, jer se prema 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, a `*` 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 tehnički 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 pišu se 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 se pišu dva znaka između apostrofa u ovim specijalnim slučajevima, označena vrednost ovim literalima je samo jedan znak tipa `char`.

Numerički literali, odnosno načini za pisanje brojeva u Java programu su složeniji nego što bi se to moglo očekivati. Brojeve naravno možemo pisati na uobičajeni način na koji smo navikli, uz malu razliku da se koristi tačka umesto decimalne zapete: na primer, `107`, `23.55` ili `-0.5`. Ali postoje i druge mogućnosti koje su ponekad bolje od uobičajenog zapisa 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, odnosno označava realni broj, po automatizmu se smatra brojem tipa `double`.[2] Da bi se ovo promenilo 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 smatra se 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 znakom `\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`. Logičke vrednosti u programima se najčešće dobijaju kao rezultat izračunavanja relacijskih operacija.

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

Za operaciju deljenja celih brojeva treba naglasiti da se kao rezultat dobija isto ceo broj (tj. decimalni deo „tačnog" rezultata se odbacuje). Na primer, rezultat izraza `17/5` je ceo 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` dobija se 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 posebnu pažnju na to da se za operaciju određivanja jednakosti dve vrednosti pišu zapravo dva znaka jednakosti (`==`). 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 grade od sastavnih delova kojima se može nezavisno manipulisati. O klasnim tipovima podataka biće mnogo više reči u nastavku knjige, a ovde se radi ilustracije govori samo kratko o jednom važnom, unapred definisanom klasnom tipu `String`.

Objekti tipa `String` se nazivaju *stringovi* (ili *niske*) i predstavljaju nizove znakova. Neke vrednosti stringova već su korišćene u programu *Zdravo* na strani 29:

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

Dvostruki apostrofi na početku i na kraju niza znakova su neophodni kod pisanja vrednosti stringova, 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 stringa 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 `\"`. Tekst, na primer,

```
On reče: "Zdravo svima!"
```

sa znakom za novi red na kraju, piše se u programu u obliku stringa na ovaj način:

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

Nad stringovima je dozvoljena operacija *spajanja* (ili *konkatenacije*). Ta operacija se u Javi označava znakom `+`, jer podseća na „sabiranje” stringova. Na primer, kao rezultat izraza

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

dobija se string `Javaprogramiranje`. U opštem slučaju, rezultat operacije spajanja dva stringa je novi string koji se sastoji od znakova prvog stringa iza kojih se redom 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,

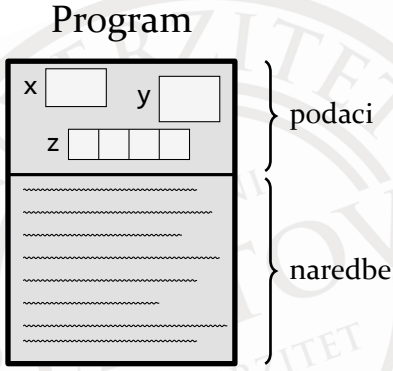
```
"Java" + " " + "programiranje" + " je " + "zabavno"
```

kao rezultat daje string `Java programiranje je zabavno`.

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

3.4 Promenljive

Podaci osnovnih tipova, koji su opisani u prethodnom odeljku, čuvaju se u memorijskim ćelijama u posebnom delu programa. Programer ne mora (a i ne može) da vodi računa o tačnim memorijskim lokacijama gde se nalaze podaci, već se za memorijske lokacije koriste simbolička imena preko kojih se u programu ukazuje na podatke u njima. Ova imena kojima su u programu pridruženi podaci nazivaju se *promenljive*. Dopunjena predstava računarskog programa, sa delom za podatke u kojem promenljive služe kao ćelije za čuvanje podataka, prikazana je na slici 3.1.



Slika 3.1: Računarski program sa tri promenljive `x`, `y` i `z`.

Promenljivu je najbolje zamisliti kao jednu kutiju u kojoj 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. 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.

Jedna kutija kao promenljiva može sadržati samo jedan podatak u svakom trenutku, ali se podaci u kutiji mogu menjati tokom izvršavanja programa. Prema tome, promenljiva može sadržati različite podatke u različitim trenucima izvršavanja programa, ali promenljiva uvek ukazuje na istu kutiju. Upravo ova činjenica da promenljiva može sadržati promenljive podatke tokom izvršavanja programa jeste razlog za njen termin *promenljiva*.

Najčešći način za dodelu neke vrednosti jednoj promenljivoj, odnosno za „stavljanje" neke vrednosti u kutiju koju označava promenljiva, jeste pomoću *naredbe dodele*. Zapis naredbe dodele u Javi ima ovaj opšti oblik:

```
promenljiva = izraz;
```

Ovde, na levoj strani znaka jednakosti, deo *promenljiva* označava ime odgovarajuće promenljive, dok na desnoj strani znaka jednakosti deo *izraz* predstavlja zapis kojim se navodi ili izračunava neka vrednost. Kada se neka naredba dodele izvršava tokom izvršavanja programa, najpre se izračunava *izraz* i zatim se dobijena vrednost dodeljuje *promenljivoj*.

Na primer, u konkretnoj naredbi dodele

```
kamatnaStopa = 0.08;
```

promenljiva na levoj strani je `kamatnaStopa`, a *izraz* na desnoj strani je literal `0.08`. To znači da se ova naredba dodele izvršava tako što se broj 0.08 na desnoj strani znaka jednakosti dodeljuje promenljivoj `kamatnaStopa` na levoj strani znaka jednakosti. Drugim rečima, uzimajući u obzir analogiju promenjive i kutije, broj 0.08 se „stavlja" u kutiju sa imenom `kamatnaStopa`, zamenjujući njen bilo koji stari sadržaj.

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

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

U izrazu `kamatnaStopa * kredit` na desnoj strani znaka jednakosti, imena `kamatnaStopa` i `kredit` označavaju promenljive, a znak * označava operaciju množenja. Uobičajeno, ova naredba dodele se 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`, a rezultat tog množenja se zatim upisuje u promenljivu `kamata` koja se nalazi na levoj strani znaka jednakosti.

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

Istaknimo još jednu važnu činjenicu u vezi sa naredbom dodele. Znak jednakosti koji se pojavljuje u naredbi dodele služi samo za razdvajanje dva sastavna dela ove naredbe: promenljive na levoj i izraza na desnoj strani znaka jednakosti. 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 ovu razliku, primetimo 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. S druge strane, sličan 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 *definisana* (ili *deklarisana*) u programu. Definicija promenljive se sastoji od pisanja njenog tipa i njenog imena, tim redom. Tip promenljive određuje zapravo tip podataka koje promenljiva može sadržati tokom izvršavanja programa. Ta informacija je potrebna radi rezervisanja memorijskog prostora čija je veličina dovoljna za smeštanje podataka odgovarajućeg tipa. Opšti zapis *naredbe definicije promenljive* je:

```
tip promenljiva;
```

Prilikom definicije promenljive se može dodatno dodeliti početna vrednost novoj promenljivoj. U tom slučaju, zapis naredbe definicije promenljive je:

```
tip promenljiva = izraz;
```

U prethodnim zapisima, *`tip`* je jedan od primitivnih ili klasnih tipova, *`promenljiva`* je ime promenljive koja se definiše, a *`izraz`* se izračunava da bi se dodelila početna vrednost promenljivoj koja se definiše. Na primer:

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

U ovom primeru se definišu 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 definicijama.

Obratite pažnju na to da promenljiva može imati različite vrednosti tokom izvršavanja programa, ali sve te vrednosti moraju biti istog tipa, tačno

onog navedenog u definiciji 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.

Naredbe definicija promenljivih mogu biti, u stvari, malo složenijeg oblika. Naime, iza tipa podataka se može pisati lista više promenljivih (sa ili bez početnih vrednosti) koje su odvojene zapetama. Time se prosto definiše (i inicijalizuje) više promenljivih istog tipa. Na primer:

```
int i, j, n = 32;
boolean indikator = 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 definišu. Pri tome od ovog pravila treba odstupiti samo ako je više promenljivih 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 definicije 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
```

Promenljive se mogu definisati unutar metoda, kao i unutar klase ali izvan svih metoda u klasi. Kada se definišu unutar klase van metoda radi opisa atributa objekata klase, promenljive se nazivaju *polja*. Za sada nećemo koristiti polja, već samo promenljive u užem smislu koje se definišu unutar metoda (i to najčešće unutar metoda `main()`). Promenljive definisane unutar nekog metoda se još nazivaju *lokalne* promenljive za taj metod. One postoje samo unutar metoda u kojem su definisane i potpuno su nedostupne izvan tog metoda.

Definicije promenljivih se mogu pisati bilo gde unutar metoda, pod uslovom da se poštuje pravilo kojim se propisuje da svaka promenljiva mora biti definisana pre nego što se koristi u nekoj naredbi. U vezi sa mestom definisanja promenljivih ne postoji konsenzus oko toga šta je dobar stil programiranja. Neki programeri vole da sve promenljive definišu na samom početku metoda. Drugi pak definišu 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 definišu na početku metoda

(uz komentar šta označavaju), a sve „pomoćne" promenljive da se definišu tek tamo gde se prvi put koriste.

Primer: pretvaranje Farenhajtovih stepena u Celzijusove

Da bismo primenili programske elemente Jave o kojima smo govorili do sada, u nastavku je prikazan kompletan program kojim se dati stepeni Farenhajta pretvaraju u stepene Celzijusa. Preciznije, ulaz za program je broj stepena Farenhajta koji se dobija od korisnika, a izlaz je odgovarajući broj stepena Celzijusa koji se prikazuje na ekranu.

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

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

Poznavajući ovu formulu nije teško napisati traženi program koji je prikazan u listingu 3.1. U programu se koriste dve celobrojne promenljive koje služe za čuvanje svih podataka programa: promenljiva `f` za broj stepena Farenhajta koji korisnik želi da pretvori u stepene Celzijusa i promenljiva `c` za izračunati broj stepena Celzijusa po prethodnoj formuli.

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

Listing 3.1: Pretvaranje Farenhajtovih stepena u Celzijusove

```
import java.util.*;

public class FarCel {

   public static void main(String[] args) {
      int f;  // broj stepena Farenhajta
      int c;  // broj stepena 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 u programu na to da je znak { naveden na kraju drugog i trećeg reda, a ne kao što je do sada bio slučaj da je taj znak pisan 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 bi se ovaj program izvršio 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 razvojnom okruženju je detaljno opisan 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.

3.5 Konstante

Promenljiva može sadržati promenljive vrednosti istog tipa tokom izvršavanja programa. U izvesnim slučajevima, međutim, vrednost neke promenljive u programu ne treba da se menja nakon dodele početne vrednosti

toj promenljivoj. U Javi se u naredbi deklaracije promenljive može dodati službena reč `final` kako bi se obezbedilo to da se promenjiva ne može promeniti nakon što se inicijalizuje. Na primer, iza deklaracije

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

svaki pokušaj promene vrednosti promenljive `MAX_POENA` proizvodi grešku.

Ove „nepromenjive promenljive" se nazivaju *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 standardnim klasama Jave u kojima su definisane mnoge konstante. Tako, na primer, u klasi `Math` se nalazi konstanta `PI` koja sadrži vrednost broja π, ili u klasi `Integer` se nalazi konstanta `MIN_VALUE` koja sadrži minimalnu vrednost tipa `int`.

3.6 Neke korisne klase

Metod u Javi je potprogram koji obavlja neki specifični zadatak. Drugačije rečeno, jedan metod se sastoji od niza naredbi i grupe promenljivih koji predstavljaju zaokruženu funkcionalnu celinu sa svojim posebnim imenom. Izvršavanje metoda, odnosno niza naredbi od kojih se metod sastoji, 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 je složeniji i o tome će biti više reči u poglavlju 6.

Java sadrži veliki broj standardnih klasa sa unapred napisanim metodima koji se mogu koristiti u programima. Ovi metodi se mogu koristiti (pozivati) bez ikakvog razumevanja kako su napisani ili kako rade, jer je dovoljno znati samo *šta* ti metodi rade, odnosno koja je njihova funkcija. To je, u stvari, prava suština metoda — metod je „crna kutija" koja se može koristiti bez poznavanja šta se nalazi unutar te kutije.

Klase u Javi imaju zapravo tri potencijalno 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 (klasni) članovi* klase. Na primer, u svakoj klasi od koje počinje izvršavanje programa mora se 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 koji imaju poseban identitet. U ovoj mnogo važnijoj ulozi, nekom klasom se definišu polja i metodi koje imaju svi objekti koji pripadaju toj klasi. Ova polja i metodi se nazivaju *nestatički (objektni, instancni) članovi* klase.

Treće, jednom klasom se u programu definiše i novi tip podataka u tehničkom smislu, a vrednosti tog tipa su objekti definisane klase. Na primer, standardnom klasom `String` u jeziku Java se definiše tip podataka sa istim imenom `String`, a vrednosti ovog tipa su objekti koji predstavljaju nizove znakova kao što su, recimo, `"Java je kul!"` ili `"Zdravo narode."`.

Ova višestruka uloga klasa u Javi može biti zbunjujuća za početnike, ali u praksi se te 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`. S druge strane, standardna klasa `Math` sadrži samo statičke metode kojima se izračunavaju uobičajene matematičke funkcije. Da bi se bolje razumeli ovi detalji, u nastavku su opisane neke „ugrađene" klase koje se često koriste u Java programima.

Klasa `System`

Klasa `System` je osnovna klasa koja obezbeđuje nezavisan model operativnog sistema računara u Java programima. Jedna od funkcija ove klase je, na primer, podrška za učitavanje ulaznih podataka i prikazivanje izlaznih podataka programa. Klasa `System` obuhvata kolekciju statičkih metoda i polja kojima se mogu realizovati najosnovnije funkcionalnosti potrebne u skoro svim programima.

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. Objekat standardnog izlaza na koji ukazuje polje `System.out` sadrži metode za prikazivanje vrednosti na ekranu. Jedan od njih je metod `print()`. Složeno ime `System.out.print` odnosi se dakle na metod `print()` u objektu na koji 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 koji ukazuje statičko polje `out` u klasi `System`.

U odeljku 2.5 na strani 31 pokazane su osnovne mogućnosti metoda `print()` i `println()` za prikazivanje vrednosti na ekranu. Podsetimo se da je njihova jedina funkcionalna razlika u tome što se metodom `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. (Primetimo da se zbog toga može lako razlikovati 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.) Na primer, ukoliko treba kursor samo pomeriti za jedan red, piše se:

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

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 se dobija ovaj rezultat:

```
Mesečna kamata: 1.6666666666666667
```

Pošto je promenljiva `kamatnaStopa` tipa `double`, njena vrednost se izračunava sa 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 dodat metod `printf()` koji ima slične mogućnosti kao ista funkcija u jeziku C. Broj opcija metoda `printf()` za formatizovanje podataka je vrlo velik, ali ovde ćemo pomenuti samo njih nekoliko. Dodatne informacije o metodu `printf()` čitaoci mogu naći u Java dokumentaciji.

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

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

na ekranu se dobija ovaj rezultat:

```
Mesečna kamata:  1.67
```

U opštem slučaju, 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. Na primer, izvršavanjem naredbi

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

na ekranu se dobija:

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

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

Pored delova koji počinju sa znakom `%` za opis formata podataka, prvi argument metoda `printf()` može sadržati i 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. Na primer, izvršavanjem naredbi

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

na ekranu se dobija ovaj tekst:

```
Kvadrat broja 23 je      529
```

U ovom primeru, format `%d` se odnosi na vrednost drugog argumenta metoda `printf()` (tj. promenljivu `x`), a format `%8d` se odnosi na vrednost trećeg argumenta (tj. izraza `x*x`). 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. Ova mogućnost se koristi u naredbi

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

u kojoj se vrednost promenljive `kamatnaStopa` prikazuje u formatu `%5.2f`. Deo između `%` i slova `f` se u opštem slučaju sastoji od ukupnog broja mesta

i broja decimala realnog broja razdvojenih tačkom. Tako `5.2` u primeru označava da realni broj koji je vrednost promenljive `kamatnaStopa` treba da se prikaže 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` označava da se neka vrednost treba prikazati u polju od (minimalno) deset mesta.

Klasa `System` sadrži i statički metod `exit()` koji se naravno u programu mora koristiti pod punim imenom `System.exit()`. Ovaj metod bezuslovno prekida izvršavanje programa i koristi se u slučajevima 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 recimo `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 programa. U praksi se vrednost argumenta obično ne uzima u obzir, iako se mora pisati.

Klasa `Math`

Svaki metod obavlja neki specifični zadatak. Rezultat toga kod nekih metoda je jedna vrednost određenog tipa koja se zatim može koristiti u programu. Ako metod daje neku vrednost kao rezultat svog izvršavanja, onda se govori 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 `x`.

Rezultat metoda `Math.sqrt(x)` predstavlja vrednost tipa `double` i može se koristiti na svakom mestu u programu gde je dozvoljeno koristiti vrednosti tog tipa. Na primer, rezultat kvadratnog korena vrednosti promenljive `x` može se na ekranu prikazati naredbom:

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

Isto tako, poziv metoda `Math.sqrt(x)` može biti deo naredbe dodele na desnoj strani znaka jednakosti radi dodele vrednosti izračunatog kvadratnog korena nekoj promenljivoj:

```
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)` kao rezultat daje realni broj 5.0, a rezultat poziva `Math.round(5.76)` je ceo broj 6.
- `Math.random()` kao rezultat daje (pseudo) slučajni realni broj iz intervala $[0, 1)$.

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

Klasa `Math` sadrži i nekoliko statičkih polja kojima su predstavljene poznate matematičke konstante. Na primer, polje `Math.PI` sadrži broj $\pi = 3.14159\cdots$, dok polje `Math.E` sadrži broj $e = 2.71828\cdots$. Ova statička polja su tipa `double` i naravno sadrže samo približnu vrednost odgovarajućih konstanti koje imaju beskonačno decimala.

Klasa `String`

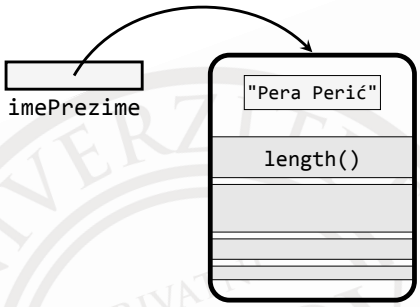
Vrednosti klase `String` su objekti koji se nazivaju stringovi (niske). Jedan objekat klase `String` sadrži podatke, tj. niz znakova koji čine string, ali

i metode kojima se može manipulisati nizovima znakova. Metod `length()`, na primer, vraća broj znakova u stringu.

Ako pretpostavimo da je promenljiva `imePrezime` definisana da bude tipa `String` i ako joj je dodeljena neka vrednost, recimo,

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

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



Slika 3.2: Veza promenljive i dodeljenog stringa u memoriji.

Zapis `imePrezime.length()` označava poziv metoda `length()` primenjenog za string na koji ukazuje promenljiva `imePrezime`. Rezultat ovog poziva je ceo broj 10 koji predstavlja dužinu (broj znakova) stringa `"Pera Perić"`. Opštije rečeno, za svaku promenljivu `s` tipa `String`, rezultat poziva `s.length()` je ceo broj tipa `int` jednak broju znakova stringa u objektu na koji ukazuje promenljiva `s`. Obratite pažnju na to da metod `length()` nema eksplicitni argument u zagradama 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 dosta nezgrapna. Naime, neka promenljiva tipa `String` ne sadrži niz znakova, nego samo ukazuje na objekat koji unutar sebe u jednom polju sadrži konkretni niz znakova. Radi kraćeg izražavanja, međutim, ovo se zanemaruje i govori se kao da je vrednost takve promenljive sâm niz znakova. Tako se u prethodnom primeru govori 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 svojstvena 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 još veliki broj dodatnih metoda. U nastavku su opisani najčešće korišćeni metodi klase `String`, uz pretpostavku da su `s1` i `s2` promenljive tipa `String`:

- `s1.equals(s2)` kao rezultat daje logičku vrednost `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, kao rezultat daje znak tipa `char` koji se nalazi u n-toj poziciji stringa `s1`. Pri tome, treba imati u vidu da su pozicije niza znakova koji čine string numerisane od 0. Stoga `s1.charAt(0)` daje prvi znak stringa `s1`, zatim `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, kao rezultat daje podniz tipa `String` koji se sastoji od znakova stringa `s1` u pozicijama n, $n+1$, ..., $m-1$. Obratite pažnju na to da se znak u poziciji m stringa `s1` ne uključuje.
- `s1.indexOf(s2)` kao rezultat daje 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, ako je `z` promenljiva tipa `char`, može koristiti i `s1.indexOf(z)` radi dobijanja pozicije prvog pojavljivanja znaka `z` u stringu `s1`.
- `s1.compareTo(s2)` kao rezultat daje ceo broj koji je 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()` kao rezultat daje 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 koristi se poziv `s1.toLowerCase()`.

- `s1.trim()` kao rezultat daje novi string koji je jednak stringu `s1`, osim što se svi nevidljivi znakovi na početku i kraju stringa `s1` odstranjuju. Na primer, ako string `s1` ima vrednost `"␣␣to␣je:␣"`, rezultat poziva `s1.trim()` jeste string `"to␣je:"`.

Primetimo da se u pozivima `s1.toUpperCase()`, `s1.toLowerCase()` i `s1.trim()` *ne* modifikuje implicitni argument string `s1`, 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();
```

Kao što je spomenuto u odeljku 3.3, nad stringovima se može primeniti operacija spajanja (konkatenacije) koja se označava znakom `+`. Na primer, izvršavanje naredbe

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

sastoji se od dva međukoraka. Najpre se argument metoda `println()` između zagrada izračunava tako što se navedena tri stringa (`"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 metodima `print()` i `println()` za prikazivanje vrednosti bilo kog tipa. Na primer, naredbom

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

prikazuje se realna 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. S druge strane, dodavanjem praznog stringa dobija se vrlo sličan izraz `"" + god + 1`, ali potpuno drugi rezultat: string `"231"`. Naime, u prvom izrazu, `god + 1`, znak `+` se interpretira kao operator sabiranja brojeva, jer su oba operanda brojevi. U drugom izrazu, `"" + god + 1`, prvi znak `+` se pak interpretira kao operator spajanja stringova, jer se kao jedan od operanada pojavljuje string. Zato se prazan string spaja sa stringom `"23"` i dobija se 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 kojom se olakšava učitavanje ulaznih podataka u 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 jedan objekat tipa Scanner.

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

```
new NekaKlasa( ... )
```

Ovde zapis NekaKlasa(...) iza reči new označava poziv specijalnog metoda koji se naziva *konstruktor* klase NekaKlasa čiji se objekat konstruiše. O konstruktorima će biti više reči u odeljku 7.3. Za sada je dovoljno znati da je to specijalni metod neke klase kojim se inicijalizuje konstruisani objekat.

Pored konstruisanog objekta, rezultat operatora new je i 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 zagrade 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, u ranije korišćenoj naredbe dodele

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

na desnoj strani znaka jednakosti iskorišćena je samo mogućnost jednostavnijeg zapisa konstruisanja stringa u punom obliku:

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

Ako se operator `new` primeni za konstruisanje objekta klase `Scanner`, može se 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 je 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 koji ukazuje statičko polje `in` u klasi `System`, baš kao što je standardni izlaz predstavljen objektom na koji ukazuje statičko polje `out` u klasi `System`.

Objekti klase `Scanner` predstavljaju dakle izvore podataka i sadrže metode koji se mogu primeniti na njih radi učitavanja podataka različitog tipa. Uz prethodnu definiciju promenljive `tastatura`, opis nekih od ovih metoda je:

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

Neki od ovih metoda su korišćeni u programu *Zdravo* na strani 29. Naredbom, na primer,

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

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

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

- `tastatura.hasNext()` — vraća se logička vrednost `true` (tačno) ukoliko je neki ulazni niz znakova, različit od znakova beline, raspoloživ

preko tastature; u suprotnom slučaju, vraća se logička vrednost `false` (netačno).

- `tastatura.hasNextInt()`, `tastatura.hasNextDouble()` itd. — vraća se logička vrednost `true` (tačno) ako je neka vrednost odgovarajućeg tipa `int`, `double` i tako dalje, raspoloživa preko tastature; u suprotnom slučaju, vraća se logička vrednost `false` (netačno).
- `tastatura.hasNextLine()` — vraća se logička vrednost `true` (tačno) ako je ulazni niz znakova raspoloživ preko tastature do kraja reda; u suprotnom slučaju, vraća se logička vrednost `false` (netačno).

Napomenimo na kraju da su ovde prikazane 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 pristup je ipak isti, a više detalja o dodatnim mogućnosti klase `Scanner` čitaoci mogu pronaći u Java dokumentaciji.

Klase omotači

Osam primitivnih tipova podataka u Javi nisu klase, a ni vrednosti primitivnih tipova nisu objekti. Ponekad je ipak potrebno primitivne vrednosti tretirati kao da su to objekti. To se ne može direktno uraditi, ali se vrednost primitivnog tipa može „umotati" u objekat odgovarajuće *omotačke klase*. Za svaki od osam primitivnih tipova postoji odgovarajuća klasa omotač: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` i `Boolean`.

Klase omotači 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. Glavno svojstvo omotačke klase ogleda se u tome što svaki njen objekat sadrži objektno polje odgovarajućeg primitivnog tipa. U tom smislu, takav objekat je „omotač" za vrednost odgovarajućeg primitivnog tipa. Na primer, objekat-omotač za realnu vrednost 5.220931 tipa `double` može se konstruisati operatorom `new`:

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

Ova promenljiva `d` klasnog tipa `Double` nosi iste informacije kao promenljiva primitivnog tipa `double` čija je vrednost 5.220931, ali u formi objekta. Ako treba koristiti vrednost tipa `double` koju sadrži objekat-omotač na koga ukazuje `d`, to se ne može uraditi direktno (bar ne u ranijim verzijama

Jave), nego se mora primeniti metod `doubleValue()` iz 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, neka vrednost tipa `int` u kontekstu u kojem je potreban objekat tipa `Integer`, ta vrednost tipa `int` će se automatski pretvoriti u objekat tipa `Integer`. Može se pisati, na primer,

```
Integer broj = 23;
```

i to će se automatski interpretirati kao da je napisano:

```
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`, onda se u najnovijoj verziji Jave može kraće pisati:

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

U ovom primeru, prilikom izračunavanja izraza na desnoj strani znaka jednakosti, vrednost tipa `double` koja se nalazi u objektu na koga ukazuje `d` biće automatski *raspakovana* i pomnožena sa 3.14.[3]

Autopakovanje i raspakivanje se u Javi može primeniti u svim slučajevima kada to ima smisla. Zbog toga se i koncepcijske razlike između vrednosti primitivnih tipova i objekata mogu skoro potpuno zanemariti.

✧ ✧ ✧

Druga dobra strana klasa omotača ogleda se 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 simbolička imena nego numeričke vrednosti. Iste konstante se nalaze i u klasama `Byte`, `Short`, `Long`, `Float`, `Double` i `Character`.

[3] Numeričke omotačke klase sadrže objektne metode kojima se može izvršiti i „ručno" raspakivanje. Primer toga je `d.doubleValue()` kojim se dobija realna vrednost tipa `double` koja se nalazi u objektu na koji ukazuje `d`.

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 vrednost `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, onda se u programu signalizira greška. Slično, klasa `Double` sadrži metod `parseDouble()` koji pretvara string u vrednost tipa `double`. Na primer, `Double.parseDouble("1.548e2")` kao rezultat daje realni broj 154.8 tipa `double`.

Statički metodi `valueOf()` u ovim klasama imaju slične funkcije, ali kao rezultat daju objekat odgovarajuće klase omotača. Tako na primer, `Integer.valueOf("17")` kao rezultat daje objekat klase `Integer` koji sadrži celobrojnu vrednost 17. Slično, `Double.valueOf("1.548e2")` kao rezultat daje objekat klase `Double` koji sadrži realnu vrednost 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 se znakovi u Javi predstavljaju prema Unicode šemi u kojoj se nalazi ogroman broj glifova.) Na primer, metodi `isLetter()`, `isLowerCase()` i tako dalje kao rezultat daju logičku vrednost tačno ili netačno prema tome da li je dati argument tipa `char` znak za slovo, malo slovo i tako dalje.

Pitanja i zadaci

1. U kojem od ovih slučajeva *nije* ispravno napisan komentar u Javi?

A. `/** tekst komentara */`
B. `// tekst komentara`
C. `-- tekst komentara`
D. `/* tekst komentara */`
E. `** tekst komentara **`

2. Koje su od ovih reči rezervisane (službene) reči u Javi?

A. `public` B. `static` C. `void` D. `class` E. `tačno`

3. U kojem su od ovih slučajeva ispravno napisana imena (identifikatori) u Javi?

A. `9x` B. `ekran` C. `brojStudenata` D. `znak+ili-`

4. U kojem su od ovih slučajeva ispravno napisana imena (identifikatori) u Javi?

A. `3praseta` B. `tri praseta` C. `prečnik` D. `bzvz`

5. U koje dve kategorije se dele svi mogući tipovi podataka u Javi?

A. Specifični i generalni B. Celobrojni i realni C. Numerički i tekstualni D. Primitivne i klasni

6. Koji od ovih celobrojnih tipova podataka zahteva najviše memorije za predstavljanje celih brojeva?

A. `long` B. `int` C. `short` D. `byte`

7. Koji od ovih tipova podataka u Javi služe za predstavljanje realnih brojeva?

A. `float` B. `int` C. `long` D. `double` E. `boolean`

8. Koliko bajtova u memoriji zauzima jedan znak u Javi tipa `char`?

A. Jedan B. Dva C. Tri D. Četiri

9. Koje su moguće logičke vrednosti tipa `boolean` u Javi?

A. `tačno` B. `true` C. `false` D. `0` E. `1` F. `netačno`

10. U kojem su od ovih slučajeva ispravno napisana imena promenljivih prema *konvenciji* u Javi za davanje imena promenljivim?

A. `kredit` B. `Kredit` C. `KREDIT` D. `kamatnaStopa`
E. `KamatnaStopa` F. `kamatna_stopa`

11. Kojim znakom se završava skoro svaka naredba u Javi?

A. tačkom B. tačkom-zapetom C. zapetom D. zvezdicom

12. U kojem su od ovih slučajeva ispravno napisane naredbe za definisanje promenljivih u Javi?

A. `int dužina; int širina;`
B. `int dužina, širina;`
C. `int dužina; širina;`
D. `int dužina, int širina;`

13. U kojem su od ovih slučajeva ispravno napisane naredbe za prikazivanje teksta `Java je kul!` na ekranu?

A. `System.out.println('Java je kul!');`
B. `System.println("Java je kul!");`
C. `System.out.writeln("Java je kul!");`
D. `System.out.println("Java je kul!");`
E. `System.out.print("Java je kul!");`
F. `System.out.printf("Java je kul!");`

14. U kojem su od ovih slučajeva ispravno napisana naredba dodele vrednosti 17 promenljivoj `x`?

A. `17 = x;` B. `x = 17;` C. `x := 17;` D. `x == 17;`

15. Kojom se od ovih naredbi ispravno definiše promenljiva `x` tipa `int` sa početnom vrednošću 17?

A. `int x = '17';`
B. `int x == 17;`
C. `int x = 17;`
D. `int x = 17.0;`

16. Koje od ovih naredbi dodele *nisu* ispravne?

A. `float f = -34;`

B. `int t = 23;`
C. `short s = 10;`
D. `int t = (int)false;`
E. `int t = 4.5;`

17. Kojim metodom se u Javi odmah prekida izvršavanje programa?

A. `System.halt(0)`
B. `System.exit(0)`
C. `System.stop(0)`
D. `System.terminate(0)`

18. Koji metod u Javi izračunava broj `x` na stepen `y`?

A. `Math.power(x, y)`
B. `Math.exp(x, y)`
C. `Math.pow(x, y)`
D. `Math.pow(y, x)`

19. Koja je od ovih naredbi definisanja promenljive ispravna u Javi?

A. `char c = 'A';`
B. `char c = '23';`
C. `char c = "A";`
D. `char c = "23";`

20. Koje su od ovih naredbi definisanja promenljive ispravne u Javi?

A. `string s = 'A';`
B. `String s = '23';`
C. `String s = "A";`
D. `String s = "23";`

21. Šta se dodeljuje promenljivoj `c` kao rezultat izvršavanja ovog programskog fragmenta?

```
String s = "Java";
char c = s.charAt(4);
```

A. `'a'` B. `'v'` C. `'"'` D. Ništa, zbog greške

22. Ako su `s1` i `s2` promenljive tipa `String`, koji su slučajevi ovih naredbi ili izraza pogrešni?

A. `String s = "neki string";`
B. `String s3 = s1 + s2;`
C. `s1 >= s2`
D. `int i = s1.length;`
E. `s1.charAt(0) = '?';`

23. Ako su `s1` i `s2` promenljive tipa `String`, koji su slučajevi ovih naredbi ili izraza pogrešni?

A. `String s3 = s1 - s2;`
B. `s1 == s2`
C. `boolean b = s1.compareTo(s2);`
D. `char c = s1[0];`
E. `char c = s1.charAt(s1.length());`
F. `char c = s1.charAt(s1.length() - 1);`

24. Ako su `s1` i `s2` promenljive tipa `String`, šta je rezultat ovog izraza?

```
s1.equals(s2) == s2.equals(s1)
```

A. 0 B. 1 C. `true` D. `false`

25. Šta je rezultat izraza `"java" + "program"` u Javi?

A. `javaprogram` B. `Java program` C. `Java_program`
D. `Javaprogram` E. Ništa, zbog greške

26. Kojim se od ovih naredbi ispravno pretvara string `s` u celobrojnu vrednost promenljive `i` tipa `int`?

A. `i = Integer.parseInt(s);`
B. `i = (new Integer(s)).intValue();`
C. `i = Integer.valueOf(s).intValue();`
D. `i = Integer.valueOf(s);`
E. `i = (int)(Double.parseDouble(s));`

27. Kojim se od ovih naredbi ispravno pretvara string `s` u realnu vrednost promenljive `d` tipa `double`?

A. `d = Double.parseDouble(s);`
B. `d = (new Double(s)).doubleValue();`
C. `d = Double.valueOf(s).doubleValue();`
D. `d = (double)(Integer.parseInt(s));`

28. Kojim se od ovih naredbi ispravno pretvara realna vrednost promenljive `d` tipa `double` u string `s`?

A. `s = d;`
B. `s = d.toString();`
C. `s = (new Double(d)).toString();`
D. `s = (Double.valueOf(d)).toString();`

✧ ✧ ✧

1. Napisati program koji u pravougaoniku na ekranu ispisuje vaše ime i prezime na otprilike sledeći način:

```
+----------------+
|                |
| Dejan Živković |
|                |
+----------------+
```

2. Napisati program koji na ekranu ispisuje vaše inicijale velikim slovima koja su „nacrtana" zvezdicama (znakom *) u matrici dimenzije 11×12. Na primer, na ekranu treba da se za inicijale DŽ dobije otprilike sledeća slika:

```
                      **  **
                        **
*******             ************
**     **                     **
**       **                 **
**        **              **
**        **             **
**        **            **
**       **           **
**     **           **
*******             ************
```

3. Napisati program koji format papira A4 (210×297 mm) prikazuje u inčima (1 inč = 2.54 cm).

4. Napisati program koji Celzijusove stepene pretvara u Farenhajtove po formuli $f = 9c/5 + 32$.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

4

Izrazi

Izrazi su važna gradivna komponenta svakog programskog jezika i zato im se u ovom poglavlju posvećuje posebna pažnja. Primer jednog izraza u Javi je `2*r*Math.PI`. Izrazi su, pojednostavljeno rečeno, formule na osnovu kojih se izračunava neka vrednost. Izračunata vrednost izraza se može dalje dodeliti nekoj promenljivoj, koristiti u drugim naredbama, koristiti kao argument u pozivima metoda, ili kombinovati sa drugim vrednostima u građenju složenijeg izraza. Način na koji se pišu i izračunavaju izrazi podleže određenim pravilima koja se obično podudaraju sa uobičajenim pravilima za izračunavanje formula, ali imaju i svoje specifičnosti zbog programske prirode izraza.

4.1 Prosti i složeni izrazi

Svaki izraz se izračunava tokom izvršavanja programa i kao rezultat se dobija tačno jedna vrednost koja se naziva *vrednost izraza*. Tri osnovna elementa od kojih se grade izrazi su:

- Literali (Podsetimo se, to su „bukvalne" vrednosti određenog tipa kao što su `17`, `0.23`, `true` ili `'A'`.)
- Promenljive
- Pozivi metoda

U stvari, pojedini literali, promenljive i pozivi metoda se smatraju *prostim* izrazima. Vrednost prostog izraza koja se dobija njegovim izračunavanjem je sama vrednost koju literal označava, trenutna vrednost promenljive ili rezultat poziva metoda.

Složeni izrazi se grade kombinovanjem prostih izraza sa dozvoljenim operatorima (aritmetičkim, relacijskim i drugim). Izračunavanje vrednosti složenog izraza odvija se primenom odgovarajuće operacija na vrednosti prostijih izraza koji učestvuju u složenom izrazu. Na primer, izraz `2*r*Math.PI` se izračunava tako što se vrednost literala `2` (koja je broj 2) množi sa aktuelnom vrednošću promenljive `r`, a zatim se ovaj međurezultat množi sa vrednošću konstante `Math.PI` (koja je 3.14⋯) da bi se dobila konačna vrednost polaznog izraza.

Ako se u složenom izrazu nalazi više operatora, onda se njihov redosled primene radi izračunavanja tog izraza određuje na osnovu prioriteta operatora. Izraz, na primer,

```
x + y * z
```

izračunava se tako što se najpre izračunava podizraz `y * z`, a zatim se njegov rezultat sabira sa `x`. Ovaj redosled je posledica toga što operacija množenja označena znakom `*` ima viši prioritet od operacije sabiranja koja je označena znakom `+`. Ako ovaj unapred definisan prioritet operatora ne odgovara potrebnom redosledu izračunavanja izraza u programu, redosled izračunavanja izraza može se promeniti upotrebom zagrada. Na primer, u izrazu `(x+y)*z` bi se najpre izračunao podizraz u zagradi `x+y`, a zatim bi se njegov rezultat množio sa `z`.

Operatori koji učestvuju u složenim izrazima označavaju uobičajene operacije nad odgovarajućim tipovima podataka.[1] Nepotpun spisak raspoloživih operatora u Javi može se podeliti u nekoliko grupa:

- Aritmetički operatori: `+`, `-`, `*`, `/`, `%`, `++`, `--`
- Relacijski operatori: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logički operatori: `&&`, `||`, `!`
- Operator izbora: `?:`
- Operatori dodele: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

4.2 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

[1] Operatori i njihove oznake su skoro potpuno preuzeti iz jezika C.

(`byte`, `short`, `int`, `long`, `float` i `double`). Ali kada se neka operacija stvarno primenjuje za dve vrednosti, obe vrednosti moraju biti istog tipa. Na primer, za izračunavanje `17.4 + 10` najpre se ceo broj `10` pretvara u ekvivalentni realni broj `10.0`, a zatim se izračunava `17.4 + 10.0`. Ovo pretvaranje vrednosti jednog tipa u ekvivalentnu vrednost drugog tipa naziva se *konverzija tipa* i obično se automatski izvršava ukoliko 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`, onda je i rezultat ceo broj tipa `int`; ako se sabiraju dva realna broja tipa `double`, onda je i rezultat realni broj tipa `double`. Ovo je prirodno pravilo, osim u slučaju operatora deljenja `/`. Rezultat deljenja dva cela broja je isto 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 tipa `int`. Tako, ako `n` ima vrednost `10`, rezultat izračunavanja `1 / n` je `0`.

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*). Taj operator u Javi se označava pisanjem imena ciljnog tipa podataka u zagradi koja se piše ispred vrednosti koju treba pretvoriti u navedeni tip. Na primer, ako su `n` i `m` celobrojne promenljive tipa `int`, onda se u izrazu

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

zahteva pretvaranje vrednosti promenljive `n` u tip `double` pre njenog deljenja sa vrednošću promenljive `m`. Na taj način 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()`, koji je opisan na strani 54, ne može se neposredno primeniti, jer on daje slučajni *realni* broj između 0 i 0.999 ⋯. Zbog toga, vraćenu vrednost tog metoda treba transformisati na odgovarajući način. Prvi korak je množenje te vrednosti sa 6, pisanjem `6 * Math.random()`, ka-

ko bi se dobio slučajni realni broj između 0 i 5.999 ···. Operator eksplicitne konverzije tipa `(int)` može se zatim iskoristiti za dobijanje celog broja ovog rezultata množenja, 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 je potreban slučajni ceo broj između 1 i 6, na kraju treba još dodati 1 da bi se dobilo konačno rešenje:

```
(int)(6 * Math.random()) + 1
```

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 zahtevanje suvišno, jer bi se konverzija ionako izvršila iz konteksta.)

Pošto celobrojno deljenje daje samo celobrojni rezultat, u Javi postoji 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. U Javi su na raspolaganju i nekoliko unarnih aritmetičkih operatora koji se primenjuju samo na jednu vrednost. Jedan takav operator je već spomenut — 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 se može pisati `+x`, mada to u stvari ne menja ništa.

Unarni operatori za inkrementiranje i dekrementiranje celobrojne 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 može se postići naredbom dodele `x=x+1`. Naime, na desnoj strani znaka jednakosti 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 jednakosti). 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 vrednosti celobrojne promenljive 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, onda 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` mogu se koristiti i kao samostalni izrazi ili biti deo složenijih izraza. Na primer, ispravno napisane 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` promenljivoj `y`, a zatim uvećavanje vrednosti promenljive `x` za 1. Na primer, ako `x` ima vrednost 10, onda nakon izvršavanja `y = x++;` promenljiva `y` će imati vrednost 10 (staru vrednost `x`), dok će nova vrednost `x` biti 11. S druge strane, efekat naredbe dodele `y = ++x;` je da se najpre uvećava vrednosti promenljive `x` za 1, a zatim se ta *nova* vrednosti promenljive `x` dodeljuje promenljivoj `y`. Na primer, ako `x` ima vrednost 10, onda nakon izvršavanja `y = ++x;` promenljive `x` i `y` će imati istu vrednost 11, jer se nova vrednost `x` dodeljuje `y`. U slučaju operatora dekrementiranja `--` važe slična pravila.

Efekat pisanja operatora inkrementiranja i dekrementiranja ispred ili iza promenljive je, očigledno, prilično komplikovan čak i u 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 jer u njima nema razlike da li se ti operatori nalaze ispred ili iza promenljive.

4.3 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 se govori u poglavlju 5. 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 ove naredbe 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` striktno manje od `y`; u suprotnom slučaju je rezultat `false`.
- `x > y` kao rezultat daje logičku vrednost `true` ako je `x` striktno 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` iz odeljka 3.6: `equals()`, `equalsIgnoreCase()` i `compareTo()`.

4.4 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 I 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 ILI 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 i ILI 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 s nulom, 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 ope-

randa (`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 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 izraza `! x` jednak `false`. Drugim rečima, rezultat logičke negacije je suprotna logička vrednost od vrednosti operanda.

4.5 Operator izbora

U Javi postoji i jedan ternarni operator (operator sa tri operanda) koji se pominje samo radi kompletnosti, jer je njegova razumljivost u programu diskutabilna. Pošto se ovaj operator može lako zameniti naredbom grananja o kojoj se govori u narednom poglavlju, dobar stil programiranja nalaže da operator izbora treba koristiti samo u izuzetnim slučajevima. Kako god, operacija izbora se u opštem obliku piše na sledeći način:

$$\textit{logički-izraz}\ ?\ \textit{izraz}_1\ :\ \textit{izraz}_2$$

Obratite pažnju na to da se znak pitanja piše iza *`logičkog-izraza`* i da znak dve-tačke razdvaja $\textit{izraz}_1$ i $\textit{izraz}_2$. Izvršavanje operacije izbora se odvija u dva koraka. Najpre se izračunava *`logički-izraz`*, a zatim se zavisno od njegove vrednosti dobija rezultat operacije izbora na sledeći način: ako *`logički-izraz`* daje tačno, onda je konačni rezultat jednak izračunatoj vrednosti za $\textit{izraz}_1$; u suprotnom slučaju, ako *`logički-izraz`* daje netačno, onda je konačni rezultat jednak izračunatoj vrednosti za $\textit{izraz}_2$.

Na primer, naredbom dodele

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

promenljivoj `n` skroz levo se dodeljuje vrednost izraza `2 * m` ako je vrednost promenljive `m` paran broj (tj. ako je izračunata vrednost logičkog izraza `m % 2 == 0` jednaka tačno). Ali, promenljivoj `n` se dodeljuje vrednost izraza `m - 1` ako je vrednost promenljive `m` neparan broj (tj. ako je izračunata vrednost logičkog izraza `m % 2 == 0` jednaka netačno). Primetimo da zagrade oko izraza u ovom primeru nisu neophodne, ali je s njima postignuta bolja razumljivost naredbe.

4.6 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 proizvodi rezultat, koji se eventualno može koristiti u sklopu nekog složenijeg izraza. Vrednost koja se dobija primenom operatora dodele u opštem obliku

promenljiva = izraz

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

Na primer, ako vrednost promenljive y treba dodeliti promenljivoj x i zatim testirati da li je ta vrednost jednaka 0, onda se može 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 dodele. Taj rezultat, odnosno vrednost promenljive y, 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 je to već pokazano kod 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 koristiti naredbu grananja o kojoj će više reči biti u odeljku 5.2.

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

promenljiva += izraz

ekvivalentno je sa:

promenljiva = promenljiva + 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
p &&= q;  // isto što i p = p && q
```

Kombinovani operator dodele `+=` se može primeniti i za stringove. Podsetimo se da operator `+` za stringove označava njihovo spajanje. Ako su `s` i `t` stringovi, budući da je `s += t` isto što i `s = s + t`, nova vrednost stringa `s` jednaka je njegovoj staroj vrednosti kojoj su dodati znakovi stringa `t`. Na primer, ako `s` ima vrednost `"Dragan"`, onda se izazom `s += "a"` menja njegova vrednost u `"Dragana"`.

4.7 Prioritet operatora

Ukoliko se u nekom izrazu pojavljuje više operatora i ako zagradama nije eksplicitno naznačen redosled njihovog izračunavanja, operatori se primenjuju po unapred definisanom prioritetu. U tabeli 4.1 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.

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

Tabela 4.1: Prioritet operatora u Javi od najvišeg do najnižeg.

Operatori u jednoj vrsti tabele 4.1 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. Zato se, na primer, izraz `x * y / z` izračunava postupno kao da stoji `(x * y) / z`, dok se izraz `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, svuda gde može doći do zabune treba navesti zagrade da bi se eksplicitno naznačio redosled izračunavanja izraza koji je potreban. Time se umnogome povećava razumljivost programa 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 se na raspolaganju nalazi neograničen broj svih novčića.

Postupak za rešavanje problema vraćanja kusura je onaj koji se primenjuje u svakodnevnom životu. Ako bismo trebali da usitnitimo neki iznos, 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 se usitnjava koristeći dva novčića vrednosti 25, jedan novčić vrednosti 10, jedan novčić vrednosti 5 i tri novčića vrednosti 1.

Da bismo ovaj postupak pretočili u Java program, primetimo da je broj najvećih novčića za neki 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 oduzima-

mo najveći novčić vrednosti 25, broj tih novčića za usitnjavanje je jednak `68 / 25` i preostali iznos 18 je jednak `68 % 25`.

Za usitnjavanje datog novčanog iznosa u Java programu u listingu 4.1 koriste se pet celobrojnih promenljivih za čuvanje potrebnih podataka. Pri tome, 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 se za ove preostale iznose mogu uvesti nove promenljive, izabrano je da se „reciklira" promenljiva `iznos` radi jednostavnosti 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 4.1: 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 kamata koja se zaračunava za neotplaćeni iznos (pri čemu $k = 0.08$ označava kamatnu stopu od 8%) i r iznos mesečne rate koja se plaća za kredit. Koliki je neotplaćeni iznos uzetog 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 se uvede oznaka $a = 1 + k/12$, onda je:

- početni iznos kredita posle „nultog" meseca:

$$p_0 = p$$

- iznos kredita posle prvog meseca:

$$p_1 = p_0 + \frac{k}{12}p_0 - r = \left(1 + \frac{k}{12}\right)p_0 - r = ap_0 - r$$

- iznos kredita posle drugog meseca:

$$p_2 = p_1 + \frac{k}{12}p_1 - r = \left(1 + \frac{k}{12}\right)p_1 - r = ap_1 - r$$

- i tako dalje ...
- iznos kredita posle m-tog meseca:

$$p_m = p_{m-1} + \frac{k}{12}p_{m-1} - r = \left(1 + \frac{k}{12}\right)p_{m-1} - r = ap_{m-1} - r$$

Ako se u poslednjem izraz za p_m na desnoj strani zameni formula za p_{m-1}, zatim u dobijeni izraz zameni formula za p_{m-2} i tako dalje obrnutim redom sve do formule za p_0, na kraju se dobija formula za iznos neotplaćenog kredita p_m nakon 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 je potrebno odrediti formulu za aktuelni iznos mesečne rate r, onda treba pretpostaviti da se iznos neotplaćenog kredita p_m nakon m meseci smanjio na 0. Zamenom $p_m = 0$ u prethodnoj formuli za p_m i rešavanjem dobijene jednačine po r dobija se:

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

Java program kojim se na osnovu ove formule izračunava mesečna rata za date vrednosti kredita, godišnje kamate i broja mesečnih rata prikazan je u listingu 4.2.

Listing 4.2: 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 iznosa 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 am = Math.pow(1 + mk, m); // a na m
      double r = (p * am * mk) / (am - 1);

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

Obratite pažnju u ovom programu na to da su, radi promene, promenljive definisane neposredno pre mesta gde su potrebne. Za izračunavanje vrednosti a^m u formuli za mesečnu ratu korišćen je metod `Math.pow()` za

izračunavanje stepena u Javi. Pri tome je rezultat poziva tog metoda sačuvan u promenljivoj `am` da bi program bio malo brži, jer bi se inače taj metod morao pozivati dva puta u narednoj naredbi dodele. Iz sličnog razloga je uvedena i promenljiva `mk` za vrednost mesečne kamate $k/12$, jer se ona dva puta koristi u daljem izračunavanju.

Pitanja i zadaci

1. Šta je rezultat izraza `45 / 4` u Javi?

A. 10 B. 11 C. 11.25 D. 12

2. Koji od ovih izraza kao rezultat daju 0.5 u Javi?

A. `1 / 2`
B. `1.0 / 2`
C. `1.0 / 2.0`
D. `(double) (1 / 2)`
E. `(double) 1 / 2`
F. `1 / 2.0`

3. Koji od ovih izraza kao rezultat daje 1 u Javi?

A. `2%1` B. `15%4` C. `25%5` D. `37%6`

4. Ako su `a` i `b` celobrojne promenljive tipa `int`, koji od ovih izraza u Javi kao rezultat daju tačan rezultat (realni broj) za matematički izraz $\frac{a}{b^2}$? (Na primer, ako su $a = 5$ i $b = 2$, onda je $\frac{a}{b^2} = \frac{5}{4} = 1.25$ tačan rezultat.)

A. `a / b * b`
B. `a / (b * b)`
C. `1.0 * a / b * b`
D. `1.0 * a / (b * b)`
E. `(double) a / (b * b)`

5. Koji se tekst dobija na ekranu izvršavanjem ovog programskog fragmenta?

```
double x = 5.5;
int y = (int)x;
System.out.println("x je " + x + " i y je " + y);
```

A. `x je 5 i y je 6`
B. `x je 6.0 i y je 6.0`
C. `x je 6 i y je 6`
D. `x je 5.5 i y je 5`
E. `x je 5.5 i y je 5.0`

6. Kojim se od ovih naredbi ispravno dodaje 1 celobrojnoj promenljivi `i` tipa `int`?

A. `i = i + (2 - 1);`
B. `i = i + 1;`
C. `i += 1;`
D. `i = 1 + i;`
E. `i++;`

7. Analizirajte sledeći program:

```
public class Test {
   public static void main(String[] args) {
      int mesec = 09;
      System.out.println("Mesec je " + mesec);
   }
}
```

A. Program na ekranu prikazuje `Mesec je 09`.
B. Program na ekranu prikazuje `Mesec je 9`.
C. Program na ekranu prikazuje `Mesec je 9.0`.
D. Program ima grešku, jer `09` nije ispravno napisana oktalna vrednost.

8. Kako se u Javi označava relacijski operator „manje ili jednako”?

A. `<` B. `=<` C. `>=` D. `<=` E. `<<` F. `!=`

9. Kako se u Javi označava relacijski operator „jednako”?

A. `<>` B. `!=` C. `==` D. `=`

10. Koji su od ovih logičkih izraza ispravno napisani u Javi?

A. `(true) && (3 => 4)`
B. `!(x > 0) && (x > 0)`
C. `(x > 0) || (x < 0)`

D. `(x != 0) || (x = 0)`
E. `(-10 < x < 0)`

11. Koji od ovih logičkih izraza ispravno daje vrednost `true` ako je broj `x` između 1 i 100 ili je broj `x` negativan?

A. `1 < x < 100 && x < 0`
B. `((x < 100) && (x > 1)) || (x < 0)`
C. `((x < 100) && (x > 1)) && (x < 0)`
D. `(x != 0) || (x = 0)`
E. `(1 > x > 100) || (x < 0)`
F. `(1 < x < 100) || (x < 0)`

12. Šta je vrednost promenljive `x` posle izračunavanja izraza

```
(y > 10) && (x++ > 10)
```

ako je pre izračunavanja tog izraza bilo $x = 10$ i $y = 10$?

A. 9 B. 10 C. 11 D. 12

13. Šta je vrednost promenljive `x` posle izračunavanja izraza

```
(y > 10) || (x++ > 10)
```

ako je pre izračunavanja tog izraza bilo $x = 10$ i $y = 10$?

A. 9 B. 10 C. 11 D. 12

14. Koju vrednost ima promenljiva `y` posle izvršavanja ovog programskog fragmenta? (*Savet:* operator izbora ima viši prioritet od operatora dodele.)

```
x = 0;
y = (x > 0) ? 10 : -10;
```

A. -10 B. 0 C. 10 D. 20 E. Ništa, zbog greške

✧ ✧ ✧

1. Napisati program koji izračunava iznos kamate na depozit i uvećano stanje depozita nakon jedne godine. Ulazne veličine programa su početni depozit i godišnja kamatna stopa, a izlazne veličine su novčani iznos kamate i uvećani depozit nakon jedne godine.

2. Napisati program koji izračunava hipotenuzu pravouglog trougla ako su date dve njegove katete. Pored toga, program treba da prikaže vreme u sekundama koje je utrošeno za obavljanje tog zadatka.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

5

Složene naredbe

Naredbe u Javi mogu biti proste ili složene. Proste naredbe predstavljaju osnovne sastavne elemente programa. U ove naredbe spadaju naredba definicije promenljivih, naredba dodele i naredba poziva metoda. Prve dve proste naredbe smo upoznali u prethodnom delu knjige, dok se o naredbi poziva metoda govori u sledećem poglavlju.

Složene naredbe se grade od ovih prostijih programskih elemenata na tri načina čime se dobijaju posebne kategorije upravljačkih naredbi: *blok naredbi*, *naredbe grananja* i *naredbe ponavljanja*. Svaka od ovih konstrukcija se smatra 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.

Od tri vrste složenih naredbi, jedino se blokom naredbi određuje sekvencijalni redosled izvršavanja naredbi od kojih se blok sastoji, dok se naredbama grananja i ponavljanja taj uobičajeni način izvršavanja programa može promeniti. U ovom poglavlju se posvećuje veća pažnja ovim i ostalim detaljima složenih naredbi. Pri tome se kroz razne primere opisuje njihova sintaksa (kako se ispravno pišu u programu) i semantika (kakav se efekat proizvodi prilikom njihovog izvršavanja u programu).

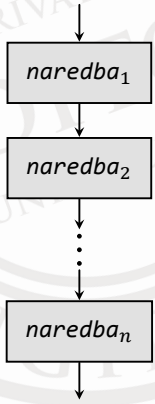
5.1 Blok naredbi

Blok naredbi (ili kraće samo blok) jeste najprostiji način kombinovanja naredbi kojim se samo niz naredbi grupiše u jednu naredbu. Opšti oblik zapisa bloka od n naredbi je:

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

Drugim rečima, blok se sastoji od niza naredbi između otvorene vitičaste zagrade `{` i zatvorene vitičaste zagrade `}`.[1] Blok se obično nalazi unutar drugih složenih naredbi kada je potrebno da se više naredbi grupiše u jednu (složenu) naredbu. U opštem slučaju, međutim, blok se može nalaziti na bilo kom mestu u programu gde je dozvoljeno pisati neku naredbu.

Izvršavanje bloka naredbi izvodi se tako što se sekvencijalno izvršavaju pojedine naredbe koje se nalaze u bloku. Naime, kada se naiđe na otvorenu vitičastu zagradu bloka, sve naredbe u nizu koje slede iza toga izvršavaju se redom jedna za drugom, dok se ne naiđe na zatvorenu vitičastu zagradu bloka. Efekat izvršavanja bloka od n naredbi u opštem obliku ilustrovan je blok-dijagramom na slici 5.1.



Slika 5.1: Izvršavanje bloka naredbi.

Blok naredbi je do sada korišćen, kao što su pažljivi čitaoci možda primetili, u metodu `main()` svakog programa. Naime, niz naredbi od kojih se sastoji telo ovog metoda, a i svakog drugog metoda, predstavlja zapravo blok naredbi, jer se taj niz naredbi nalazi unutar vitičastih zagrada. Tako, blok naredbi koje obrazuje telo metoda `main()` u primeru programa *Zdravo* na strani 29 je:

[1] 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 naredbi čini niz od 7 naredbi koje se redom izvršavaju jedna za drugom, od početka bloka označenog znakom { do kraja bloka označenog znakom }.

U narednom primeru bloka naredbi se zamenjuju vrednosti celobrojnih promenljivih x i y za koje se pretpostavlja da su definisane i inicijalizovane 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 naredbi čini niz od četiri naredbe koje se redom izvršavaju. Primetimo da je pomoćna promenljiva z definisana unutar samog bloka. To je potpuno dozvoljeno i, u stvari, dobar stil programiranja nalaže da promenljivu treba definisati lokalno unutar bloka u kome se koristi, a ne izvan tog bloka.

Promenljiva koja je definisana 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 definisanu unutar nekog bloka kaže se da je *lokalna* za taj blok ili da je taj blok njena *oblast važenja*. (Tačnije, oblast važenja lokalne promenljive je deo njenog bloka od mesta definicije promenljive do kraja tog bloka.)

Blok naredbi sam po sebi ne utiče na uobičajeni, sekvencijalni tok izvršavanja programa. Važnost bloka naredbi ogleda se u tome što određuje okvir u kome važe promenljive definisane u njemu, ali i što se može pisati u programu na svim mestima gde je dozvoljeno da se nalazi neka naredba. Ova druga mogućnost se često koristi u drugim složenim naredbama u

onim njihovim delovima koji se sastoje od dve ili više naredbi.

5.2 Naredbe grananja

Naredbe grananja obezbeđuju alternativne puteve za tok izvršavanja programa zavisno od vrednosti nekog logičkog izraza. Na primer, ako u nekoj tački programa korisnik treba da izabere neku od ponuđenih opcija, onda 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 naredbe grananja spadaju naredba `if`, naredba `if-else` i naredba `switch`.

Naredbe `if` i `if-else`

Naredbe `if` i `if-else` u programu služe za izbor jednog od dva alternativna niza naredbi za izvršavanje, zavisno od toga da li je vrednost datog logičkog izraza tačno ili netačno. Prva od ove dve naredbe, naredba `if`, u opštem obliku počinje službenom rečju `if` iza koje se navodi neki logički izraz u zagradama i, na kraju, neka naredba jezika Java:

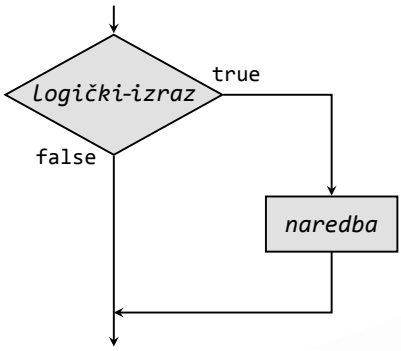
```
if (logički-izraz)
   naredba
```

Jedan konkretan primer naredbe `if` je:

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

U ovom primeru se naredbom `if` proverava da li je vrednost promenljive `r` (poluprečnik kruga) različita od nule. Ako je to slučaj, izvršava se naredba dodele u produžetku kojom se izračunava obim kruga i rezultat dodeljuje promenljivoj `obim`; u suprotnom slučaju, preskače se izvršavanje ove naredbe dodele.

U opštem slučaju, izvršavanje naredbe `if` se izvodi u dve faze: najpre se izračunava vrednost *`logičkog-izraza`* u zagradi, a zatim se, ako je ta vrednost `true`, izvršava *`naredba`* koja se nalazi u produžetku; ako je ta vrednost `false`, ništa se dodatno ne izvršava, odnosno ova *`naredba`* se preskače. Time se u oba slučaja izvršavanje naredbe `if` završava, a dalje izvršavanje programa se nastavlja od naredbe koja sledi iza naredbe `if`. Ovo izvršavanje naredbe `if` je ilustrovano blok-dijagramom na slici 5.2.



Slika 5.2: Izvršavanje naredbe `if`.

Kako blok naredbi unutar vitičastih zagrada predstavlja formalno jednu naredbu, *naredba* unutar naredbe `if` može biti i blok naredbi:[2]

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

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

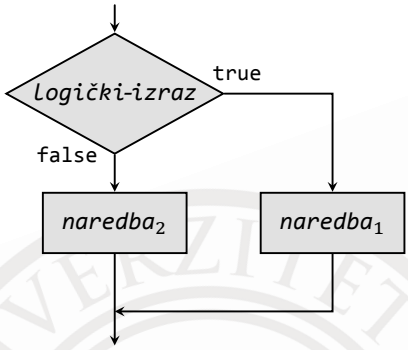
Ako je vrednost logičkog izraza u zagradi unutar naredbe `if` jednaka netačno, onda često programska logika nalaže da treba uraditi nešto drugo, a ne samo preskočiti naredbu navedenu iza logičkog izraza. Za takve slučajeve služi naredba `if-else` čiji je opšti oblik:

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

Izvršavanje naredbe `if-else` je slično izvršavanju naredbe `if`, osim što se izvršava $naredba_2$ i preskače $naredba_1$ u slučaju kada je vrednost logič-

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

kog izraza u zagradi jednaka netačno. U drugom slučaju, kada je vrednost logičkog izraza jednaka tačno, izvršava se *naredba*$_1$ i preskače *naredba*$_2$. Time se u oba slučaja izvršavanje naredbe `if-else` završava, a dalje izvršavanje programa se nastavlja od naredbe koja sledi iza naredbe `if-else`. Ovo je ilustrovano blok-dijagramom na slici 5.3.



Slika 5.3: Izvršavanje naredbe `if-else`.

Obratite pažnju na to da se prilikom izvršavanja naredbe `if-else` izvršava tačno jedna od dve naredbe — *naredba*$_1$ ili *naredba*$_2$. Ove dve naredbe predstavljaju alternativne puteve za tok izvršavanja programa koji se biraju zavisno od vrednosti logičkog izraza. Naravno, *naredba*$_1$ i *naredba*$_2$ unutar naredbe `if-else` mogu biti blokovi naredbi. U narednom primeru se računa i prikazuje obim kruga za dati poluprečnik:

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

U ovom primeru se koristi naredba `if-else` radi provere ispravnosti vrednosti poluprečnika kruga koju unosi korisnik. Pri tome, `if` deo te naredbe `if-else` sastoji se od jedne naredbe koja zato ne mora (ali može) da bude napisana unutar para vitičastih zagrada.[3] S druge strane, `else` deo naredbe

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

`if-else` u primeru sastoji se od dve naredbe koje se zato moraju nalaziti unutar para vitičastih zagrada u formi bloka.

✧ ✧ ✧

Važan detalj u vezi sa opštim oblikom složenih naredbi `if` i `if-else` jeste to što sve njihove sastavne naredbe, odnosno *naredba*, $naredba_1$ i $naredba_2$, mogu biti bilo koje naredbe jezika Java, a ne samo blokovi naredbi. Specifično, ovi delovi mogu biti neke druge naredbe `if` ili `if-else`. Jedna takva mogućnost *ugnježđavanja* naredbi je:

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

U ovom primeru se u `else` delu prve naredbe `if-else` nalazi druga naredba `if-else`. Ali kako je Java jezik slobodnog formata, ovo se skoro uvek piše u sledećem obliku:

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

U ovom obliku naredbe grananja bira se jedna od tri alternative za izvršavanje: $naredba_1$, $naredba_2$ ili $naredba_3$. Naime, na uobičajen način najpre se izračunava $logički\text{-}izraz_1$. Ako je njegova vrednost `true`, onda se izvršava $naredba_1$ i preskače sve ostalo; ako je njegova vrednost `false`, onda se preskače $naredba_1$ i izvršava se druga, ugnježđena naredba `if-else`. To znači da se izračunava $logički\text{-}izraz_2$ i prema tome da li je njegova vrednost `true` ili `false` izvršava se $naredba_2$ ili $naredba_3$.

Bolji je drugi od prethodna dva oblika naredbe grananja sa ugnježđavanjem, jer je razumljiviji, naročito u slučaju potrebe za većim brojem nivoa ugnježđavanja radi formiranja naredbe višestrukog grananja:

```
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 tačno. Zatim se izvršava njegova pridružena naredba i preskače se sve ostalo. Ukoliko vrednost nijednog logičkog izraza nije tačno, izvršava se naredba u poslednjem `else` delu. U stvari, ovaj `else` deo na kraju nije obavezan, pa ako nije naveden i nijedan logički izraz nema vrednost tačno, nijedna od n naredbi se ni ne izvršava.

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

Jedan primer upotrebe naredbe višestrukog grananja je naredni programski fragment u kome se određuje ocena studenta na ispitu na osnovu broja osvojenih poena od 0 do 100 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, posvetimo više pažnje jednom problemu u vezi sa ugnježđavanjem koji se popularno naziva „viseći" `else` deo. Razmotrimo sledeći primer:

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

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

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

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

Da bi se zaista dobio efekat koji je sugerisan prvobitnom interpretacijom, može se postupiti na jedan od dva načina. Prvo rešenje je da se ugnježđena naredba `if` piše unutar bloka:

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

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

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

Primer: sortiranje tri broja

Radi ilustracije naredbi `if` i `if-else`, u listingu 5.1 je prikazan jednostavan program kojim se sortiraju tri data broja. 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 unete tim redom za ulaz programa, onda izlaz programa treba da budu te iste vrednosti prikazane u rastućem redosledu: 17, 23, 69.

Ako pretpostavimo da su `x`, `y` i `z` celobrojne promenljive koje sadrže tri ulazne vrednosti, onda se za prikazivanje njihovih vrednosti u rastućem redosledu može postupiti na više načina. Jedan način, koji je primenjen u programu, jeste da se najpre odredi gde se nalazi, recimo, vrednost promenljive `x` u rastućem nizu. Ona dolazi na prvo mesto ako je manja i od `y` i od `z`; ona dolazi na poslednje mesto ako je veća i od `y` i od `z`; u preostalom slučaju, ona se nalazi u sredini. Nakon toga, u svakom od ova tri slučaja treba odrediti međusobni redosled vrednosti promenljivih `y` i `z`.

Listing 5.1: Sortiranje tri broja

```
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 ovom programu na to da su vitičaste zagrade nepotrebne u svakom od tri slučaja u kojem se određuje međusobni redosled promenljivih y i z. To je zato što se međusobni redosled y i z određuje jednom naredbom `if`, 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 je to ovde iskorišćeno radi bolje preglednosti logike programa.

Jedan isti problem se u programiranju često može rešiti na više načina. Tako, u ovom primeru za sortiranje tri broja, drugi način za rešenje tog problema je da se na početku obezbedi da je sigurno x manje od y. Zatim se upoređivanjem z sa x ili y može lako dobiti rastući redosled tri ulazna broja. Relevantni deo programa za ovakav pristup je:

```
      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, naredba `switch`, na neki način je specijalni slučaj naredbe višestrukog grananja iz prethodnog odeljka kada izbor jednog od više alternativnih blokova naredbi za izvršavanje zavisi od vrednosti datog celobrojnog ili znakovnog izraza. U tom slučaju je neefikasno da se izračunavanje datog izraza ponavlja u višestruko ugnježđenim naredbama `if`, a i naredbom `switch` se onda prirodnije odražava logika višestrukog grananja. Najčešći oblik naredbe `switch` je:

```
switch (izraz) {
   case konstanta1:  // izvrši odavde ako izraz == konstanta1
      niz-naredbi1
      break;         // kraj izvršavanja prvog slučaja
   case konstanta2:  // izvrši odavde ako izraz == konstanta2
      niz-naredbi2
      break;         // kraj izvršavanja drugog slučaja
   ⋮                 ⋮
   case konstantan:  // izvrši odavde ako izraz == konstantan
      niz-naredbin
      break;         // kraj izvršavanja n-tog slučaja
   default:          // inače, izvrši odavde u krajnjem slučaju
      niz-naredbin+1
}                    // kraj izvršavanja krajnjeg slučaja
```

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

Na kraju svakog slučaja naredbe `switch` obično se, ali ne uvek, navodi naredba `break`. (O naredbi `break` se više govori u nastavku ovog poglavlja na strani 118.) Rezultat naredbe `break` je prekid izvršavanja odgovarajućeg slučaja, a time i cele naredbe `switch`. Ako se na kraju nekog slučaja koji se izvršava ne nalazi naredba `break`, prelazi se na izvršavanje narednog slučaja unutar naredbe `switch`. Ovaj prelazak na naredni slučaj se nastavlja sve dok se ne naiđe na prvu naredbu `break` ili dok se ne dođe do kraja naredbe `switch`.

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

U narednom primeru su pokazane neke od prethodno pomenutih mogućnosti naredbe `switch`. Obratite pažnju i na to da se konstante uz klauzule `case` mogu pisati u proizvoljnom redosledu, pod uslovom da su sve različite.

```
// n je pozitivna celobrojna promenljiva
switch (2 * n) {
   case 1:
      System.out.println("Ovo se nikad neće 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;
   default:
      System.out.println("n ima vrednost veću od 4.");
}
```

Primer: rad sa menijem

Jedna od čestih primena naredbe `switch` je u programima čiji se rad zasniva na menijima. Program u listingu 5.2 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 5.2: 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 se pokazuje malo složeniji 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, tri broja `16 4 2008` predstavljaju ulazni datum 16. april 2008. godine. Izlaz programa treba da budu tri broja koja predstavljaju sutrašnji datum za dati datum, odnosno `17 4 2008`.

Sutrašnji datum je lako odrediti ako je dan datog datuma manji od broja dana u mesecu datog datuma — onda je samo dan sutrašnjeg datuma za jedan veći. Ali, ako je dan datog datuma jednak broju dana u mesecu datog datuma, recimo `30 4 2008`, onda je dan sutrašnjeg datuma `1`, ali je mesec sutrašnjeg datuma za jedan veći: `1 5 2008`. 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: `1 1 2009`.

Da bi se otkrilo o kojem se od ovih slučajeva radi, u programu u listingu 5.3 najpre se određuje ukupan broj dana u mesecu datog datuma. U tu svrhu se koristi promenljiva `maxDanaUMesecu` koja dobija odgovarajuću vrednost u naredbi `switch` na osnovu meseca datog datuma. Nakon što se odredi 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 5.3: 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 mes = tastatura.nextInt();  // mesec datog datuma
      int god = tastatura.nextInt();  // godina datog datuma

      // Određivanje broja dana u datom mesecu
      int maxBrojDana = 0;  // broj dana u datom mesecu
      switch (mes) {
         case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            maxBrojDana = 31;
            break;
         case 4: case 6: case 9: case 11:
            maxBrojDana = 30;
            break;
         case 2:
            if ((god%4 == 0) && (god%100 != 0) || (god%400 == 0))
               maxBrojDana = 29;
            else
               maxBrojDana = 28;
            break;
         default:
            System.out.println("Pogrešan mesec!\n");
            System.exit(0);
      }

      // Izračunavanje sutrašnjeg datuma
      if (dan < maxBrojDana)
         dan++;
      else {
         dan = 1;
         if (mes < 12)
            mes++;
         else {
            mes = 1;
            god++;
         }
      }
      // Prikazivanje sutrašnjeg datuma na ekranu
```

```
        System.out.print("Sutrašnji datum je: ");
        System.out.println(dan + " " + mes + " " + god);
    }
}
```

Program u listingu 5.3 zaslužuje nekoliko napomena:

- Celobrojne promenljive `dan`, `mes` i `god` 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 se određuje 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 (tj. `god%4==0`). Međutim, to nije potpuno tačan uslov, jer je godina prestupna ukoliko je ona bilo 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 naredbe `switch` se koristi metod `System.exit()` za prekid rada programa, jer se program ne može dalje nastaviti zbog pogrešnog ulaznog podatka. (O metodu `System.exit()` je bilo reči na strani 53.)
- Najzad, pažljivi čitaoci su možda primetili naredbu

  ```
  int maxBrojDana = 0;
  ```

  na početku dela programa u kojem se određuje broj dana u aktuelnom mesecu. Naravno, definicija promenljive `maxBrojDana` 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 inicijalizacije promenljive `maxBrojDana` dobija poruka o grešci kod korišćenja te promenljive u naredbi `if` 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 konkretnom primeru, u `default` slučaju `switch` naredbe se ne dodeljuje nijedna vrednost promenljivoj `maxBrojDana`, 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 `maxBrojDana`. Najjednostavniji način da se izbegne ovaj problem u ovom primeru je da se promenljivoj `maxBrojDana` dodeli inicijalna vrednost, kako bi se Java prevodilac naterao da zaključi da ova promenljiva ima vrednost u svim slučajevima `switch` naredbe.

5.3 Naredbe ponavljanja

Naredbe ponavljanja obezbeđuju ciklično izvršavanje nekog bloka naredbi više puta i zato se naredbe ponavljanja često nazivaju i *petlje* ili *ciklusi*. Broj ponavljanja određenog bloka naredbi kod naredbi ponavljanja se kontroliše vrednošću jednog logičkog izraza. To znači da se blok naredbi ponavlja sve dok je vrednost tog logičkog izraza jednaka tačno, a da se ponavljanje prekida u trenutku kada vrednost tog logičkog izraza postane netačno. Ponovljeno izvršavanje bloka naredbi se može, u principu, izvoditi beskonačno, ali takva *beskonačna petlja* obično predstavlja grešku u programu. Naredbe ponavljanja u Javi su naredbe `while`, `do-while` i `for`.

Naredba `while`

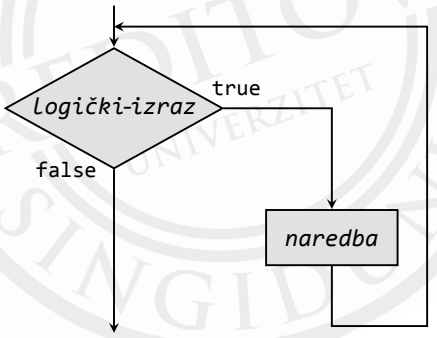
Naredba `while` je osnovna naredba za ponavljanje izvršavanja neke naredbe (ili bloka naredbi) više puta. Pošto ciklično izvršavanje bloka naredbi obrazuje petlju u sledu izvršavanja naredbi programa, naredba `while` se često naziva i `while` petlja. Blok naredbi unutar `while` petlje čije se izvršavanje ponavlja naziva se *telo petlje*, a jedno izvršavanje tela petlje se naziva jedna *iteracija* `while` petlje.

Broj ponavljanja tela petlje se kontroliše vrednošću jednog logičkog izraza. Ovaj logički izraz se naziva *uslov nastavka* (ili *uslov prekida*) petlje, jer se izvršavanje tela petlje ponavlja sve dok je vrednost tog logičkog izraza

jednaka tačno, a ovo ponavljanje se prekida u trenutku kada vrednost tog logičkog izraza postane netačno. Opšti oblik pisanja naredbe `while` je:

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

Izvršavanje naredbe `while` se izvodi tako što se najpre izračunava vrednost logičkog izraza u zagradi. U jednom slučaju, ako je dobijena vrednost jednaka netačno, odmah se prekida izvršavanje naredbe `while`. To znači da se preskače ostatak naredbe `while` i dalje izvršavanje programa se normalno nastavlja od naredbe koja sledi iza naredbe `while`. U drugom slučaju, ako izračunavanje logičkog izraza daje tačno, najpre se izvršava naredba unutar naredbe `while`, a zatim se ponovo izračunava logički izraz u zagradi i ponavlja isti ciklus. To jest, ako logički izraz daje netačno, prekida se izvršavanje naredbe `while`; ako logički izraz daje tačno, ponovo se izvršava naredba unutar naredbe `while`, opet se izračunava logički izraz u zagradi i zavisno od njegove vrednosti dalje se opet ili ponavlja ovaj postupak ili se prekida izvršavanje naredbe `while`. Ovaj način izvršavanja naredbe `while` je ilustrovan blok-dijagramom na slici 5.4.



Slika 5.4: Izvršavanje naredbe `while`.

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

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

Efekat naredbe `while` je, kratko rečeno, ponavljanje jedne naredbe ili niza naredbi u bloku sve dok je vrednost logičkog izraza jednaka tačno. U trenutku kada ta vrednost postane netačno, naredba `while` se završava i na-

stavlja se normalno izvršavanje programa od naredbe koja sledi iza naredbe `while`.

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

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

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

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

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

naredbe `while` može biti proizvoljan, uključujući nulu.

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

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

Primer: izračunavanje akumulirane štednje

U programu u listingu 5.4 prikazuje se iznos štednje koja se akumulira na kraju svake godine. Početni iznos štednje i godišnja kamata, kao i period izveštaja, dobijaju se od korisnika kao ulazne veličine programa.

Listing 5.4: 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.nextInt();

      // Prikazivanje akumulirane štednje nakon svake godine
      int godina = 1;  // godina za koju se prikazuje štednja
      while (godina <= period) {
         // Izračunavanje kamate za aktuelnu godinu
         double kamata = stanje * kamatnaStopa;
         // Dodavanje kamate na prethodno stanje
```

```
            stanje = stanje + kamata;
            // Prikazivanje novog stanja
            System.out.print("Stanje nakon " + godina + ". god.: €");
            System.out.printf("%1.2f\n", stanje);
            // Sledeća godina izveštaja
            godina++;
        }
    }
}
```

Primer: igra pogađanja broja

U ovom primeru je u listingu 5.5 prikazan program kojim se simulira igra pogađanja broja sa korisnikom. Preciznije, program „zamišlja" slučajni ceo broj između 1 i 100, a korisnik treba da ga pogodi. Ako korisnik nije pogodio zamišljen broj u jednom pokušaju, onda program treba da pomogne korisniku za sledeći pokušaj prikazujući odgovarajuću poruku da je zamišljen broj manji ili veći od pokušanog, kao i da omogući korisniku ponovno pogađanje zamišljenog broja. Ovaj dijalog između programa i korisnika treba da se odvija sve dok korisnik ne pogodi zamišljen broj.

Listing 5.5: Igra pogađanja broja

```
import java.util.*;

public class PogađanjeBroja {

   public static void main(String[] args) {

      int zamišljenBroj; // broj koji je program 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;
            }
         }
      }
}
```

U ovom programu treba obratiti pažnju na dva detalja. Prvo, slučajni broj između 1 i 100 se generiše slično načinu koji je opisan na strani 71 za generisanje slučajnog broja između 1 i 6. Drugo, logička promenljiva `pogodak` je potrebna da bi se prekinulo izvršavanje `while` petlje kada korisnik pogodi zamišljen broj. Pošto se uslov prekida petlje proverava na početku svake iteracije, pa i prve, neophodno je ovu promenljivu na početku inicijalizovati vrednošću `false` kako bi se prva iteracija sigurno izvršila.

Primer: prosek niza brojeva

Jedan od čestih problema u programiranju je sabiranje niza brojeva. Tipični način rešavanja tog problema je akumuliranjem brojeva datog niza primenom jedne petlje, pri čemu se van te petlje najpre inicijalizuje delimična suma niza brojeva nulom. Zatim se, u svakoj iteraciji te petlje, sledeći član niza brojeva dodaje delimičnoj sumi izračunatoj do tada za prethodne brojeve niza. Posle završetka petlje, kada poslednji član niza bude dodat delimičnoj sumi svih prethodnih članova niza brojeva, delimična suma biće jednaka traženoj konačnoj sumi.

Radi praktične ilustracije problema sabiranja niza brojeva i njegovog rešenja, u ovom primeru se izračunava prosek niza ulaznih brojeva. Preciznije, u programu u listingu 5.6 se redom učitavaju ulazni brojevi niza jedan po jedan i postupno se dodaju delimičnoj sumi svih prethodno učitanih brojeva. Istovremeno, pošto je za izračunavanje proseka potreban ukupan broj učitanih brojeva, njihovo brojanje se u programu izvodi uvećavanjem brojača za jedan čim se sledeći član niza brojeva učita u petlji. Na kraju, kada se odredi konačna suma ulaznog niza brojeva i njihov ukupan

broj, prosek se lako dobija kao količnik ove dve vrednosti.

Listing 5.6: Prosek 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: ");
      }

      // Prikazivanje rezultata
      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 u listingu 5.6 se za otkrivanje kraja niza učitanih brojeva koristi metod `hasNextDouble()` klase `Scanner`. Ovaj metod vraća logičku vrednost `true` ukoliko je preko tastature bio unesen neki broj tipa `double`.

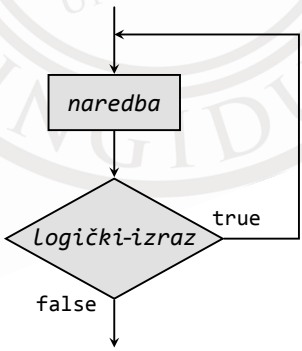
Ako je to slučaj, taj broj se učitava kao sledeći član niza ulaznih brojeva. Kraj ulaznog niza brojeva se preko tastature signalizira istovremenim pritiskom dva tastera <CTRL> i Z. U tom trenutku metod `hasNextDouble()` kao rezultat vraća logičku vrednost `false`, a time se i obezbeđuje ispunjenost uslova za prekid `while` petlje u programu.

Naredba `do-while`

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

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

Obratite pažnju na tačka-zapetu koja se nalazi na samom kraju opšteg oblika naredbe `do-while`. Tačka-zapeta je deo naredbe `do-while` i ne može se izostaviti. (Generalno, na kraju svake naredbe u Javi se mora nalaziti tačka-zapeta ili zatvorena vitičasta zagrada.) Efekat izvršavanja naredbe `do-while` u ovom opštem obliku ilustrovan je blok-dijagramom na slici 5.5.



Slika 5.5: Izvršavanje naredbe `do-while`.

Prema tome, izvršavanje naredbe `do-while` sastoji se najpre od izvršavanja tela petlje kojeg čini *naredba* između službenih reči `do` i `while`. Zatim se izračunava *logički-izraz* u zagradama. Ako je njegova izračunata vrednost jednaka `false`, onda se prekida izvršavanje naredbe `do-while` i nasta-

vlja se normalno izvršavanje programa od naredbe koja sledi iza naredbe `do-while`. U drugom slučaju, ako izračunavanje logičkog izraza daje `true`, izvršavanje se vraća na početak tela petlje i ponavlja se ciklus od početka.

Telo naredbe `do-while` može biti blok naredbi i onda je njen opšti oblik:

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

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

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

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

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

Da bismo pokazali da je ponekad prirodnije koristiti `do-while` petlju umesto `while` petlje, razmotrimo ponovo program za igru pogađanja broja na strani 108. U tom programu je bilo neophodno da se veštački uvede promenljiva `pogodak` kako bi se završila igra ili nastavilo sa pogađanjem zamišljenog broja. Ali ukoliko se koristi `do-while` petlja, ova veštačka promenljiva nije više potrebna, jer se svakako mora proći bar kroz jednu iteraciju pogađanja broja i na njenom kraju je poznato da li je zamišljen broj pogođen ili ne. Na primer, relevantni deo druge verzije ovog programa može biti:

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

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

Opšti oblik pisanja naredbe `for` je:

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

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

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

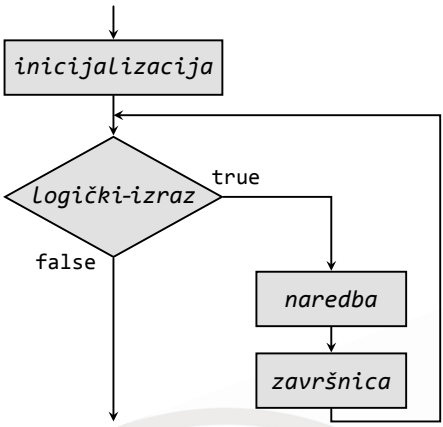
Kod naredbe `for` je kontrolnim delom na početku naredbe obuhvaćeno na jednom mestu sve što je bitno za upravljanje petljom, a to doprinosi čitljivosti i boljem razumevanju logike petlje. Ovaj kontrolni deo je dalje podeljen u tri dela koji se međusobno razdvajaju tačka-zapetom:

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

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

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

Blok-dijagramom na slici 5.6 je ilustrovan efekat izvršavanja naredbe `for` u ovom punom obliku.



Slika 5.6: Izvršavanje naredbe `for`.

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

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

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

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

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

Primetimo na kraju da se naredba `for` može uvek simulirati naredbom `while`, jer blok-dijagram na slici 5.6 odgovara sledećim programskom fragmentu:

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

Primer: određivanje da li je broj prost

Ceo broj veći od jedan je *prost* ako je deljiv bez ostatka samo sa samim sobom i jedinicom. Početak niza prostih brojeva je dakle 2, 3, 5, 7, 11, 13, 17, ... U ovom primeru je prikazan program u listingu 5.7 kojim se određuje da li je ulazni ceo broj prost.

Ako je n ulazni 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 bez ostatka. Ako se pronađe bar jedan takav delilac, n nije prost; u suprotnom slučaju, n jeste prost. Ovo proveravanje se može 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 5.7: Određivanje da li je broj prost

```
import java.util.*;

public class ProstBroj {

   public static void main(String[] args) {

      int n;  // dati broj

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

```
      System.out.print("Unesite ceo broj veći od 1: ");
      n = tastatura.nextInt();

      int maxDelilac = (int)Math.sqrt(n); // najveći delilac

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

Primetimo prvo da u programu u listingu 5.7 mora da se za izračunavanje maksimalnog delioca datog broja koristi eksplicitna konverzija tipa, jer metod `Math.sqrt()` kao rezultat vraća realni broj tipa `double`. Drugo, radi bolje preglednosti, telo `for` petlje je istaknuto tako što je uokvireno vitičastim zagradama. To ovde nije neophodno, jer se to telo sastoji samo od jedne naredbe `if`.

Primer: prikazivanje tablice množenja

Složene naredbe u Javi su naredbe koje sadrže druge naredbe. Specifično, složene naredbe mogu obuhvatati druge složene naredbe i tada se govori o ugnježđavanju ovih naredbi jedne u drugu. Do sada smo upoznali slučajeve pisanja naredbi grananja unutar drugih naredbi grananja i petlji, ali je generalno dozvoljena svaka kombinacija ugnježđavanja jedne složene naredbe u drugu, kao i neograničen broj nivoa njihovog ugnježđavanja.

U ovom primeru se ilustruje mogućnost ugnježđavanja jedne petlje u drugu u programu u listingu 5.8 kojim se prikazuje tablica 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 može se rečima opisati na sledeći način: idući redom vrsta po vrsta tablice, unutar svake vrste prikazati proizvod broja aktuelne vrste i broja aktuelne kolone. Preciznije, za svaki broj vrste od 1 do 9, prikazati proizvod broja aktuelne vrste i broja aktuelne 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 aktuelne vrste, dok unutrašnja petlja prikazuje odgovarajuće proizvode po kolonama u aktuelnoj vrsti.

Listing 5.8: Prikazivanje tablice množenja

```
import java.util.*;

public class TablicaMnoženja {

   public static void main(String[] args) {

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

U programu u listingu 5.8 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 `vrsta` ima početnu vrednost 1, brojač unutrašnje petlje `kolona` će promeniti sve vrednosti od 1 do 10; kada brojač spoljašnje petlje `vrsta` ima drugu vrednost 2, brojač unutrašnje petlje `kolona` će ponovo promeniti sve vrednosti od 1 do 10; i tako dalje, kada brojač spoljašnje petlje `vrsta` ima vrednost 9, brojač unutrašnje petlje `kolona` 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 se postiže upotrebom specifikatora formata `%4d` kako bi se

svaki broj u jednoj vrsti tablice množenja prikazao u polju dužine četiri mesta.

Naredbe `break` i `continue`

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

```
break;
```

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

U narednom primeru 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 uslov nastavka `while` petlje uvek tačan, izgleda kao da se radi o beskonačnoj petlji kojom se od korisnika stalno zahteva da unese pozitivan broj.[4] Naime, ukoliko uneti broj nije pozitivan, prikazuje se poruka o grešci i ponavlja se telo petlje od početka radi učitavanja novog broja. Ova `while` petlja se ipak ne izvršava beskonačno, jer se u njenom telu izvršava naredba `break` čim korisnik ispavno unese pozitivan broj, što dovodi do prekida izvršavanja petlje i normalnog nastavka programa.

Drugi primer korišćenja naredbe `break` je još jedan način za rešenje petlje u programu za igru pogađanja broja na strani 108:

```
while (true) {
   System.out.print("Pogodite broj> ");
```

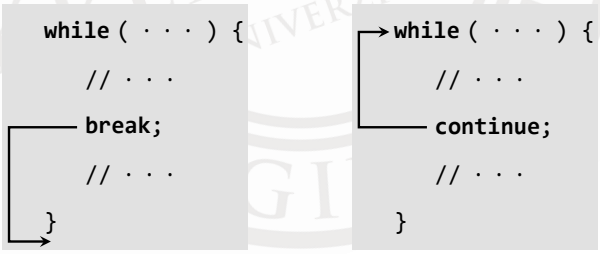
[4] Još jedan način za obrazovanje beskonačne petlje je pomoću naredbe `for` koja ima „prazan" kontrolni deo: `for (;;) { ... }`.

```
            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;
            }
         }
```

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

```
continue;
```

Međutim, izvršavanjem naredbe `continue` preskače se samo ostatak aktuelne iteracije petlje. To znači da se, umesto prekidanja izvršavanja cele petlje kao kod naredbe `break`, naredbom `continue` prekida izvršavanje samo aktuelne iteracije petlje i nastavlja sa izvršavanjem naredne iteracije petlje (uključujući i izračunavanje uslova prekida petlje radi provere da li treba uopšte nastaviti izvršavanje petlje). Razlika u dejstvu naredbi `break` i `continue` na primeru `while` petlje ilustrovana je na slici 5.7.



Slika 5.7: Efekat naredbi `break` i `continue` u petljama.

U narednom jednostavnom primeru naredbe `continue`, na ekranu se prikazuju samo parni brojevi iz intervala od 1 do 10 u jednom redu:

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

```
}
System.out.println();  // pomeriti kursor u novi red
```

✧ ✧ ✧

S obzirom na to da se petlje mogu ugnježđavati jedna u drugu, ukoliko se naredba `break` nalazi u unutrašnjoj petlji, postavlja se pitanje koju petlju prekida ova naredba `break` — unutrašnju ili spoljašnju? Pravilo je da naredba `break` uvek prekida izvršavanje samo prve obuhvatajuće petlje, ne i eventualnu spoljašnju petlju koja obuhvata petlju u kojoj se neposredno nalazi naredba `break`. Slično pravilo važi i za naredbu `continue`: ako se naredba `continue` nalazi u ugnježđenoj petlji, ona prekida aktuelnu iteraciju samo te petlje i nema nikakvog uticaja na eventualne obuhvatajuće petlje.

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

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

```
spolj_petlja: while (...) {
                  // Telo spoljašnje petlje unutar
                  // kojeg se nalazi unutrašnja petlja
                  // . . .
              }
```

Unutar tela ove spoljašnje `while` petlje mogu se zatim u nekoj unutrašnjoj petlji koristiti naredbe `break` i `continue` u obliku

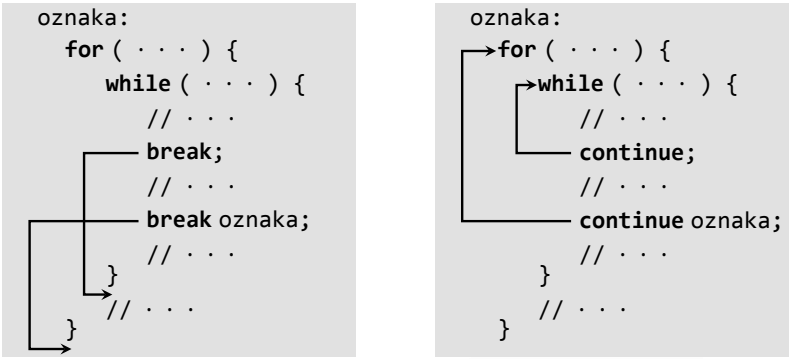
```
break spolj_petlja;
```

ili

```
continue spolj_petlja;
```

radi potpunog prekida petlje ili prekida samo aktuelne iteracije petlje sa oznakom `spolj_petlja`. Efekat naredbi `break` i `continue` u ugnježđenim označenim i neoznačenim petljama preciznije je ilustrovan na slici 5.8.

U narednom primeru se određuje da li dva stringa `s1` i `s2` sadrže neki isti znak. Postupak koji se primenjuje za to sastoji se od jedne unutrašnje `for` petlje koja je ugnježđena u spoljašnju `for` petlju sa oznakom `traži`. Pri



Slika 5.8: Efekat naredbi `break` i `continue` u ugnježđenim petljama.

tome se preko spoljašnje petlje za svaki znak stringa `s1`, u unutrašnjoj petlji proverava da li je aktuelni znak stringa `s1` jednak nekom znaku u stringu `s2`. Ako se zajednički znak pronađe, logička promenljiva `nađenIstiZnak` dobija vrednost `true` i naredbom `break` se prekidaju obe petlje i time se obustavlja dalje traženje zajedničkog znaka.

```
boolean nađenIstiZnak = false;   // s1 i s2 nemaju isti znak
traži: 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 traži;
             }
// Ovde se logika programa može nastaviti na osnovu
// vrednosti logičke promenljive nađenIstiZnak.
```

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

Primer: određivanje da li je rečenica palindrom

Rečenica se naziva *palindrom* ako se isto čita sleva na desno i zdesna na levo, ne računajući razmake i znakove interpunkcije, kao ni ne razlikujući

mala i velika slova. Primeri rečenica-palindroma su „Ana voli Milovana", „aroma sa mora", „radar" i tako dalje.

U programu u listingu 5.9 se određuje koje ulazne rečenice jesu palindrom i koje nisu. Korisnik može uneti više rečenica jednu po jednu, a one 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 5.9: 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();

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

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

         // Proveravanje znakova rečenice sa oba kraja, sve dok
         // se indeksi simetričnih znakova ne preklapaju
         int l = 0;                    // indeks levog znaka
```

```
        int d = rečenica.length() - 1; // indeks desnog znaka

        while (l < d) {
           if (rečenica.charAt(l) != rečenica.charAt(d)) {
              System.out.println("To nije palindrom.");
              continue novi_red;  // nastaviti sve ispočetka
           }
           l++;  // pomeriti levi indeks udesno
           d--;  // pomeriti desni indeks ulevo
        }
        System.out.println("To jeste palindrom.");
     } while (red.length() != 0);
  }
}
```

U programu u listingu 5.9 se koriste metodi `length()`, `toLowerCase()` i `charAt()` klase `String`, kao i metod `isLetter()` klase `Character`. Iako su funkcije ovih metoda očigledne iz njihovih imena, čitaoci se o tome mogu podsetiti u odeljku 3.6.

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

Pitanja i zadaci

1. Kojim se od ovih naredbi ispravno prikazuje površina kruga ako je njegov prečnik `r` pozitivan?

A. `if (r != 0) System.out.println(r*r*Math.PI);`
B. `if (r >= 0) System.out.println(r*r*Math.PI);`
C. `if (r > 0) System.out.println(r*r*Math.PI);`
D. `if (r > 0) {System.out.println(r*r*Math.PI);}`
E. `if {r > 0} System.out.println(r*r*PI);`
F. `if (r <= 0) System.out.println(r*r*Math.PI);`
G. `if (r > 0) System.out.println(Math.pow(r,2)*Math.PI);`

2. Ako je $x = 1$, $y = -1$ i $z = 1$, šta se prikazuje na ekranu izvršavanjem ovog programskog fragmenta? (*Savet:* najpre ispravno uparite `if` i `else` delove.)

```
if (x > 0)
   if (y > 0)
      System.out.println("x > 0 i y > 0");
else if (z > 0)
      System.out.println("x < 0 i z > 0");
```

A. `x > 0 i y > 0`

B. `x < 0 i z > 0`

C. `x < 0 i z < 0`

D. Ništa se ne prikazuje.

3. Analizirajte sledeći programski fragment:

```
boolean tačno = false;
if (tačno = true)
  System.out.println("To je tačno!");
```

A. Programski fragment ima grešku i ne može se izvršiti.

B. Programski fragment se normalno izvršava, ali se ništa ne prikazuje na ekranu.

C. Programski fragment se normalno izvršava i prikazuje se tekst `To je tačno!` na ekranu.

4. Ako je `broj` celobrojna promenljiva tipa `int`, analizirajte sledeća dva ekvivalentna programska fragmenta A i B:

Fragment A

```
boolean paranBroj;
if (broj % 2 == 0)
   paranBroj = true;
else
   paranBroj = false;
```

Fragment B

```
boolean paranBroj = (broj % 2 == 0);
```

A. Fragment A ima grešku.

B. Fragment B ima grešku.

C. Oba fragmenta imaju grešku.

D. Oba fragmenta su ispravna, ali je fragment B bolji.

5. Ako celobrojna promenljiva `plata` sadrži vrednost 4001, šta će biti prikazano na ekranu posle izvršavanja ovog programskog fragmenta?

```
if (plata > 3000)
   System.out.println("Plata je veća od 3000");
else if (plata > 4000)
   System.out.println("Plata je veća od 4000");
```

A. Ništa se neće prikazati.
B. `Plata je veća od 3000`
C. `Plata je veća od 4000`
D. `Plata je veća od 3000` u jednom redu i `Plata je veća od 4000` u sledećem redu.

6. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
double faktor = 1.5;
double cena = 0.0;
if ((faktor >= 0.0) && (faktor < 1.0))
    cena = 20 * faktor;
else if ((faktor >= 1.0) && (faktor < 1.5))
    cena = 15 * faktor;
else if (faktor >= 1.5)
    cena = 10 * faktor;
System.out.println("cena = " + cena);
```

A. `cena = 0.0`
B. Ništa se neće prikazati.
C. Programski fragment ima grešku i neće se izvršiti.
D. `cena = 15.0`

7. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {
   public static void main(String[] args) {
      String s1 = "Java";
      String s2 = s1;

      if (s1 == s2)
         System.out.println(
            "s1 i s2 ukazuju na isti string");
      else
         System.out.println(
            "s1 i s2 ukazuju na različite stringove");
   }
}
```

A. Ništa se ne prikazuje.
B. `s1 i s2 ukazuju na isti string`
C. `s1 i s2 ukazuju na različite stringove`

8. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {
   public static void main(String[] args) {
      String s1 = "Java";
      String s2 = new String("Java");

      if (s1 == s2)
         System.out.println(
            "s1 i s2 ukazuju na isti string");
      else
         System.out.println(
            "s1 i s2 ukazuju na različite stringove");
   }
}
```

A. Ništa se ne prikazuje.
B. `s1 i s2 ukazuju na isti string`
C. `s1 i s2 ukazuju na različite stringove`

9. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {
   public static void main(String[] args) {
      String s1 = "Java";
      String s2 = s1;

      if (s1.equals(s2))
         System.out.println(
            "s1 i s2 imaju isti sadržaj");
      else
         System.out.println(
            "s1 i s2 imaju različit sadržaj");
   }
}
```

A. Ništa se ne prikazuje.
B. `s1 i s2 imaju isti sadržaj`
C. `s1 i s2 imaju različit sadržaj`

10. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {
   public static void main(String[] args) {
      String s1 = "Java";
      String s2 = "Java";

      if (s1.equals(s2))
         System.out.println(
            "s1 i s2 imaju isti sadržaj");
      else
         System.out.println(
            "s1 i s2 imaju različit sadržaj");
   }
}
```

A. Ništa se ne prikazuje.
B. `s1 i s2 imaju isti sadržaj`
C. `s1 i s2 imaju različit sadržaj`

11. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {
   public static void main(String[] args) {
      String s1 = "Java";
      String s2 = s1.toUpperCase();

      if (s1 == s2)
         System.out.println("s1 i s2 ukazuju na isti string");
      else if (s1.equals(s2))
         System.out.println("s1 i s2 imaju isti sadržaj");
      else
         System.out.println("s1 i s2 imaju različit sadržaj");
   }
}
```

A. Ništa se ne prikazuje.
B. `s1 i s2 ukazuju na isti string`
C. `s1 i s2 imaju isti sadržaj`
D. `s1 i s2 imaju različit sadržaj`

12. Koju vrednost ima promenljiva y posle izvršavanja ovog programskog fragmenta?

```
int x = 3, y = 3;
switch (x + 3) {
   case 6:  y = 0;
```

```
        case 7:  y = 1;
        default: y = y + 1;
    }
```

A. 1 B. 2 C. 3 D. 4

13. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
char ch = 'a';
switch (ch) {
   case 'a':
   case 'A':
      System.out.print(ch); break;
   case 'b':
   case 'B':
      System.out.print(ch); break;
   case 'c':
   case 'C':
      System.out.print(ch); break;
   case 'd':
   case 'D':
      System.out.print(ch);
}
```

A. `abcd` B. `a` C. `aA` D. `A`

14. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
int ocena = 15;
switch (ocena) {
    case 0 :
        System.out.println("ocena je 0");
        break;
    case 15 :
        System.out.println("ocena je 15");
    case 30 :
        System.out.println("ocena je 15 ili 30");
        break;
    case 40 :
        System.out.println("ocena je 40");
    default :
        System.out.println("Pograšna ocena");
}
```

A. `ocena je 15`
B. `ocena je 15` u jednom redu, a zatim `ocena je 15 ili 30` u sledećem redu.
C. Ništa se neće prikazati.
D. `Pograšna ocena`

15. Analizirajte sledeći programski fragment:

```
int x;
double d = 1.5;
switch (d) {
   case 1.0: x = 1;
   case 1.5: x = 2;
   case 2.0: x = 3;
}
```

A. Programski fragment ima grešku, jer nedostaju naredbe `break` u svim slučajevima.
B. Programski fragment ima grešku, jer nedostaje slučaj `default` u naredbi `switch`.
C. Programski fragment ima grešku, jer kontrolna promenljiva `d` u naredbi `switch` ne može biti tipa `double`.
D. Programski fragment nema grešaka.

16. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
int k = 20;
while (k > 0)
   System.out.println(k);
```

A. Programski fragment ima grešku i neće se izvršiti.
B. `20`
C. Ništa se neće prikazati.
D. Stalno će se prikazivati `20` u beskonačnoj petlji.

17. Koliko puta se na ekranu prikazuje tekst `Kako ste?` kao rezultat izvršavanja ovog programskog fragmenta?

```
int brojač = 0;
while (brojač < 10) {
   System.out.println("Kako ste?");
   brojač++;
}
```

A. 9 B. 10 C. 11 D. 0

18. Analizirajte sledeći programski fragment:

```
int brojač = 0;
// Tačka 1
while (brojač < 10) {
   System.out.println("Kako ste?");
   brojač++;
// Tačka 2
}
// Tačka 3
```

A. Uslov `brojač < 10` je uvek tačan u tački 1.
B. Uslov `brojač < 10` je uvek netačan u tački 1.
C. Uslov `brojač < 10` je uvek tačan u tački 2.
D. Uslov `brojač < 10` je uvek netačan u tački 2.
E. Uslov `brojač < 10` je uvek tačan u tački 3.
F. Uslov `brojač < 10` je uvek netačan u tački 3.

19. Koliko puta se na ekranu prikazuje tekst `Kako ste?` kao rezultat izvršavanja ovog programskog fragmenta?

```
int brojač = 0;
do {
   System.out.println("Kako ste?");
   brojač++;
} while (brojač < 10);
```

A. 9 B. 10 C. 11 D. 0

20. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
int i = 1;
do {
   i++;
} while(i < 5);
System.out.println("i = " + i);
```

A. Programski fragment ima grešaku i neće se izvršiti.
B. `i = 5`
C. Ništa se neće prikazati.
D. Stalno će se prikazivati `i = 1` u beskonačnoj petlji.

21. Analizirajte sledeći programski fragment:

```
double suma = 0;
for (double d = 0; d < 10;) {
   d = d + 0.1;
   suma = suma + d;
}
```

A. Programski fragment ima grešku, jer nedostaje treći deo (završnica) u zagradama `for` petlje.

B. Programski fragment ima grešku, jer kontrolna promenljiva `d` u zagradama `for` petlje ne može biti tipa `double`.

C. Pošto je uslov `d < 10` uvek tačan, `for` petlja je beskonačna.

D. Programski fragment nema grešaka i normalno se izvršava.

22. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
int suma = 0;
for (int i = 0; i < 10; i++) {
   suma = suma + i;
}
System.out.println(suma);
```

A. `10` B. `11` C. `12` D. `13` E. `45`

23. Da li su ove dve `for` petlje ekvivalentne u smislu da daju istu vrednost promenljive `suma` nakon izvršavanja?

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

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

A. Da B. Ne C. Zavisi

24. Da li je ova `for` petlja sintaksno ispravna?

```
for ( ;  ;  ) ;
```

A. Da B. Ne C. Zavisi

25. Analizirajte sledeći program:

```
public class Test {
   public static void main (String args[]) {
      int i = 0;
      for (i = 0; i < 10; i++);
         System.out.println(i + 4);
   }
}
```

A. Program se neće izvršiti, jer se tačka-zapeta nalazi odmah iza zagrada `for` petlje.
B. Program će se bez problema izvršiti i prikazaće se `4` na ekranu.
C. Program će se bez problema izvršiti i prikazaće se `14` na ekranu.
D. U programu je `for` petlja ekvivalentna ovoj petlji:
`for (i = 0; i < 10; i++) { };`.

26. Da li će izvršavanje ovog programskog fragmenta biti beskonačno?

```
int stanje = 10;
while (true) {
   if (stanje < 9) break;
   stanje = stanje - 9;
}
```

A. Da B. Ne C. Zavisi

27. Koju vrednost sadrži promenljiva `suma` posle izvršavanja ovog programskog fragmenta?

```
int suma = 0;
int i = 0;
do {
   i++;
   suma = suma + i;
   if (suma > 4) break;
} while (i < 5);
```

A. `5` B. `6` C. `7` D. `8`

28. Da li će izvršavanje ovog programskog fragmenta biti beskonačno?

```
int stanje = 10;
while (true) {
   if (stanje < 9) continue;
   stanje = stanje - 9;
}
```

A. Da B. Ne C. Zavisi

29. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programskog fragmenta?

```
int x = 5, y = 20;
while (y > 1) {
   y--;
   if (y % x != 0) continue;
   System.out.print(y + " ");
}
System.out.println();
```

A. `20 19 18 17 16`
B. `20 15 10 5`
C. `15 10 5 0`
D. `15 10 5`
E. Ništa, zbog greške

30. Koja je sledeća naredba koja se izvršava nakon izvršavanja naredbe `break ponovo` u 6. redu u ovom programskom fragmentu?

```
1   ponovo:
2      for (int i = 1; i < 10; i++) {
3      dalje:
4         for (int j = 1; j < 10; j++) {
5            if (i * j > 50)
6               break ponovo;
7            System.out.println(i * j);
8         }
9      }
10  nastavak:
11     . . .
```

A. Naredba sa oznakom `ponovo` u 1. redu.
B. Naredba sa oznakom `dalje` u 3. redu.
C. Naredba sa oznakom `nastavak` u 10. redu.
D. Nijedna naredba, nego se program odmah završava.

31. Koja je sledeća naredba koja se izvršava nakon izvršavanja naredbe `continue ponovo` u 6. redu u ovom programskom fragmentu?

```
1   ponovo:
2      for (int i = 1; i < 10; i++) {
3      dalje:
```

```
4           for (int j = 1; j < 10; j++) {
5              if (i * j > 50)
6                 continue ponovo;
7              System.out.println(i * j);
8           // Tačka 1
9           }
10       // Tačka 2
11       }
12    nastavak:
13       . . .
```

A. Kontrola se prenosi u tačku 2 u u 10. redu radi izvršavanja sledeće iteracije spoljašnje petlje sa oznakom `ponovo`.

B. Kontrola se prenosi u tačku 1 u 8. redu radi izvršavanja sledeće iteracije unutrašnje petlje sa oznakom `dalje`.

C. Naredba sa oznakom `nastavak` u 12. redu.

D. Nijedna naredba, nego se program odmah završava.

✧ ✧ ✧

1. Uskrs je pokretni crkveni praznik koji uvek pada u nedelju. Tačan datum Uskrsa se određuje na osnovu prilično komplikovanih crkvenih i astronomskih pravila, a i razlikuje se postupak za izračunavanje pravoslavnog i katoličkog Uskrsa. Ipak, uz razumna ograničenja za vremenski period u kome se traži Uskrs i koristeći celobrojnu aritmetiku, izračunavanje datuma Uskrsa se može svesti na relativno jednostavan postupak. Tako, tačan datum pravoslavnog Uskrsa za datu godinu u periodu između 1900. i 2099. godine dobija se na osnovu sledećih vrednosti:

a) A je ostatak od deljenja godine sa 4

b) B je ostatak od deljenja godine sa 7

c) C je ostatak od deljenja godine sa 19

d) D je ostatak od deljenja $(19C + 15)$ sa 30

e) E je ostatak od deljenja $(2A + 4B - D + 34)$ sa 7

f) F je $D + E + 114$

Zatim, da bi se dobio datum pravoslavnog Uskrsa po starom julijanskom kalendaru, poslednju vrednosti F treba podeliti sa 31:

- mesec Uskrsa je rezultat celobrojnog deljenja F sa 31 (pri tome, 4 = april i 5 = maj)

- dan Uskrsa je za jedan veći od ostatka celobrojnog deljenja F sa 31

Na kraju, ovaj datum pravoslavnog Uskrsa po starom julijanskom kalendaru treba prevesti na novi gregorijanski kalendar dodavanjem 13 dana. Ovo prilagođavanje novom kalendaru zavisi od države do države, jer su različite države prešle na novi kalendar u različitim godinama. Srbija je, na primer, prešla na novi kalendar 1919. godine.

Napisati program koji prikazuje datum pravoslavnog Uskrsa u Srbiji za datu godinu u periodu između 1900. i 2099. godine.

2. Tačan datum katoličkog Uskrsa u periodu između 1982. i 2048. godine dobija se na osnovu sledećeg postupka (pri tome, za ostatak pri celobrojnom deljenju se koristi kraći matematički izraz „po modulu"):

a) A je godina po modulu 19
b) B je godina po modulu 4
c) C je godina po modulu 7
d) D je $(19A + 24)$ po modulu 30
e) E je $(2B + 4C + 6D + 5)$ po modulu 7
f) Uskrs je $(22 + D + E)$. marta ili, ako je $22 + D + E > 31$, Uskrs je $(22 + D + E - 31)$. aprila

Napisati program koji prikazuje datum katoličkog Uskrsa za datu godinu u periodu između 1982. i 2048. godine.

3. Postupak za izračunavanje datuma katoličkog Uskrsa u prethodnom zadatku može se lako proširiti za sve godine između 1900. i 2099. godine. Naime, u slučaju četiri godine, 1954., 1981., 2049. i 2076., prethodna formula daje datum koji je 7 dana kasniji nego što treba da bude. Modifikujte program iz prethodnog zadatka tako da se prikazuje datum katoličkog Uskrsa za datu godinu u periodu između 1900. i 2099. godine. (Napomena: u četiri posebne godine, oduzimanje 7 dana od datuma Uskrsa dobijenog po prvobitnoj formuli ne dovodi do promene meseca.)

4. Napisati program koji simulira bacanje novčića dati broj puta. Program treba da učita željeni broj bacanja novčića i da prikaže brojeve koliko puta je palo pismo i glava, kao i njihov količnik sa brojem bacanja. Program treba da ponavlja eksperiment sve dok se ne učita 0 za broj bacanja.

5. Napisati program koji simulira eksperiment bacanja dve kocke za igru i određuje koliko puta treba baciti dve kocke dok na obe kocke ne padne istovremeno broj 1. (Inače, kockarski termin za ovaj ishod je „zmijske oči".) Program treba da prikaže vrednosti kocki pri svakom bacanju i da na kraju prikaže ukupan broj bacanja kocki.

6. Niz brojeva za tzv. „$3n + 1$ problem" počinje od datog celog broja, a svaki sledeći broj niza se dobija sledeći način: ako je prethodni broj n paran, onda je sledeći broj niza jednak broju $n/2$; a ako je prethodni broj n neparan, onda je sledeći broj niza jednak broju $3n + 1$. Ovaj postupak za sledeći broj niza se ponavlja sve dok se na kraju ne dobije broj 1.

Na primer, ako je početak niza dati broj 6, sledeći broj niza je $6/2 = 3$ pošto je prethodni broj 6 paran. U sledećem koraku, budući da je 3 neparan broj, sledeći broj niza je $3 \cdot 3 + 1 = 10$. Zatim, kako je 10 paran broj, sledeći broj niza je $10/2 = 5$. Dalje, kako je 5 neparan broj, sledeći broj niza je $3 \cdot 5 + 1 = 16$. Nije teško proveriti da, nastavljajući ovaj postupak sve dok se ne dobije 1, kompletan niz za dati početni broj 6 jeste $6, 3, 10, 5, 16, 8, 4, 2, 1$.

Napisati program koji za dati početni broj n prikazuje niz brojeva za „$3n + 1$ problem".

7. Napisati program kojim se izračunava najveći zajednički delilac dva cela broja tako što se za traženje njihovog zajedničkog delioca počinje od manjeg broja od datih brojeva.

8. Napisati program koji određuje koji ceo broj između 1 i datog broja n ima najveći broj delioca i koliki je taj broj delioca. (Moguće je da više brojeva u ovom intervalu imaju isti, najveći broj delioca. Program treba da prikaže samo jedan od njih.)

9. Napisati program koji zvezdicama (znakom *) crta geometrijsku figuru romb. Broj redova romba je ulazni podatak programa i ukoliko je to paran broj, treba ga zaokružiti na prvi veći neparan broj. Na primer, ako je broj redova romba 11, na ekranu se dobija sledeća slika:

```
     *
    ***
   *****
  *******
 *********
***********
```

```
*********
 *******
  *****
   ***
    *
```

10. Napisati program koji učitava jedan red teksta i redom prikazuje njegove znakove brojeći pri tome broj slova i cifara koji se pojavljuju.

11. Napisati program koji učitava jedan red teksta i deli ga po rečima. Na primer, ako je ulazni red

```
Profesor reče: "Student je položio ispit".
```

onda se na ekranu kao rezultat dobija:

```
Profesor
reče
Student
je
položio
ispit
```

Jedna reč je niz susednih slova u ulaznom redu i treba da bude prikazana u posebnom redu. Svaki znak koji nije slovo u ulaznom redu se zanemaruje.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

6

Metodi

Metodi u Javi su potprogrami koje obavljaju neki specifični zadatak. Koncept potprograma postoji u svim programskim jezicima, ali se pojavljuje pod različitim imenima: metod, procedura, funkcija, rutina i slično. Osim što je deo programa i što obično obavlja jednostavniji zadatak, potprogram ima sličnu logičku strukturu kao svaki program. Potprogram se sastoji od niza naredbi i promenljivih koje predstavljaju neku funkcionalnu celinu. Potprogram mora imati jedinstveno ime koje se može koristiti na različitim mestima u programu kao zamena za ceo niz naredbi od kojih se potprogram sastoji.

Potprogrami se mogu čak koristiti unutar drugih potprograma. To znači da se mogu pisati 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. Potprogrami su dakle glavno sredstvo za lakše rešavanje složenih problema.

Baš kao i programi, potprogrami imaju ulazne i izlazne podatke preko kojih komuniciraju sa okolnim potprogramima. (Programi preko ulaznih i izlaznih podataka komuniciraju najčešće sa ljudima.) Ulazni i izlazni podaci potprograma su deo njegove veze sa drugim potprogramima koja se često naziva *interfejs* 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.

U Javi se za potprogram koristi termin „metod", u smislu postupak ili način za rešavanje nekog zadatka. *Metod* je dakle samostalan, imenovan

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

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

6.1 Definisanje metoda

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

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

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

Preciznije, opšti oblik pisanja definicije metoda u Javi je:

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

Prvi red ovog zapisa je zaglavlje metoda, a blok naredbi unutar vitičastih zagrada iza toga je telo metoda. Primetimo da iako je ovde, radi bolje istaknutosti, otvorena vitičasta zagrada bloka naredbi napisana u posebnom redu, ona se prema našem stilu pisanja uvek piše na kraju prethodnog reda. Na

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

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

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

Deo *`modifikatori`* na početku zaglavlja nekog metoda nije obavezan, ali se može sastojati od jedne ili više službenih reči koje su međusobno razdvojene razmacima. Modifikatorima se određuju izvesne karakteristike metoda. Tako, u primeru metoda `main()`, službena reč `public` ukazuje da se metod može slobodno koristiti u bilo kojoj drugoj klasi, a službena reč `static` ukazuje da je metod statički, a ne objektni. Za metode je u Javi na raspolaganju ukupno oko desetak modifikatora. O njima, kao i drugim elementima definicije metoda, govori se u daljem tekstu knjige u odgovarajućem kontekstu.

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

Formalna pravila za davanje imena metodu u delu *`ime-metoda`* jesu uobičajena pravila za imena u Javi o kojima smo govorili na strani 36. Dodajmo ovde samo to da je neformalna konvencija za imena metoda u Javi ista kao i za imena promenljivih: prva reč imena metoda počinje malim slovom, a ako se to ime sastoji od nekoliko reči, onda se svaka reč od druge piše sa početnim velikim slovom. Naravno, opšte nepisano pravilo je da imena metoda treba da budu smislena, odnosno da imena metoda treba da dobro opisuju šta je funkcija pojedinih metoda.

Na kraju zaglavlja metoda, deo *`lista-parametara`* u zagradama određuje sve parametre metoda. Parametri su promenljive koje se inicijalizuju vrednostima argumenata prilikom pozivanja metoda za izvršavanje. Parametri predstavljaju dakle ulazne podatke metoda koji se obrađuju naredbama u telu metoda. Na primer, pretpostavimo da je u klasi `Televizor` definisan metod `promeniKanal()` kojim se može promeniti kanal prikazi-

vanja određenog televizora, odnosno jednog objekta klase `Televizor`. Da bi izvršio ovu svoju funkciju, metod `promeniKanal()` mora imati broj kanala u koji televizor treba da promeni prikazivanje. Ali taj broj nije unapred poznat u vreme definisanja metoda `promeniKanal()`, jer televizor može prikazivati više kanala. Zato se metod `promeniKanal()` mora pisati sa parametrom koji će tek u vreme izvršavanja tog metoda dobiti vrednost kanala u koji televizor treba da promeni prikazivanje. Kako je ciljni kanal ceo broj, tip ovog parametra je `int`. Ako se još taj parametar označi sa `k`, onda nepotpuna definicija metoda `promeniKanal()` ima ovaj oblik:

```
public void promeniKanal(int k) {  // Zaglavlje metoda
    .
    . // Telo metoda
    .
}
```

U zagradama u zaglavlju ovog metoda `promeniKanal()` je navedena lista od jednog parametra pod nazivom `k` tipa `int`. Svaki parametar metoda predstavlja zapravo običnu promenljivu, stoga se kod definisanja parametra mora navesti tip i ime svakog parametra. U telu metoda se parametar koristi kao obična promenljiva čija se konkretna vrednost ne poznaje, a svaki parametar dobija specifičnu vrednost prilikom poziva metoda. Tako u prethodnom primeru parametar `k` recimo dobija vrednost 8 za poziv `promeniKanal(8)`.

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

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 se u listi parametara mora pisati `double x, double y`, a ne skraćeno `double x, y`.

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

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

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

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

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

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

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

Drugi metod `nacrtaj2DSliku()` u prethodnim primerima je objektni (nestatički) metod, jer njegovo zaglavlje ne sadrži službenu reč `static`. Generalno, metod se podrazumeva da je objektni ukoliko nije statički (što se određuje modifikatorom `static`). Modifikator `public` koji je korišćen u prvom primeru označava da je `odigrajPotez()` „javni" metod, odnosno da se može pozivati i izvan klase u kojoj je definisan, čak i u nekom drugom programu. Srodni modifikator je `private` kojim se označava da je metod „privatan", odnosno da se može pozivati samo unutar klase u kojoj je definisan. Modifikatori `public` i `private` su takozvani *specifikatori pristupa* kojima se određuju ograničenja pristupa metodu, odnosno na kojim mestima se metod može koristiti. Treći, manje korišćeni specifikator pristupa je `protected` kojim se njegovo pozivanje ograničava na klasu u kojoj je definisan i sve njene izvedene klase. Ukoliko u zaglavlju metoda nije naveden nijedan specifikator pristupa (kao u drugom i trećem od prethodnih primera), onda se metod može pozivati u svakoj klasi istog paketa kome pripada klasa u kojoj je metod definisan, ali ne i izvan tog paketa.

Modifikatori se u zaglavlju metoda moraju pisati ispred tipa rezultata metoda, ali njihov međusobni redosled nije bitan. Konvencija je ipak da se

najpre piše eventualni specifikator pristupa, zatim eventualno reč `static` i, na kraju, eventualno ostali modifikatori.

6.2 Pozivanje metoda

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

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

ime-metoda(*lista-argumenata*)

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

Da bismo ovo bolje razumeli, uzmimo primer definicije sledećeg metoda:

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

Iz zaglavlja ove definicije metoda `nekiMetod()` prepoznaje se da taj metod ne vraća nijednu vrednost i da ima tri parametra: `n` celobrojnog tipa, `x` realnog tipa i `test` logičkog tipa. Primeri ispravnih naredbi poziva metoda `nekiMetod()` u klasi u kojoj je taj metod definisan su:

- `nekiMetod(17, 0.69, false);`
- `nekiMetod(i, a, b >= 10);`
- `nekiMetod(23, Math.sqrt(y+1), s.equals(t));`

Obratite posebnu pažnju u ovim primerima na to da parametar u definiciji metoda mora biti neko *ime*, jer je parametar konceptualno jedna promenljiva. S druge strane, argument u pozivu metoda je konceptualno neka *vrednost* i zato argument može biti bilo koji izraz čije izračunavanje daje vrednost tipa odgovarajućeg parametra. Tako, u prvom primeru poziva metoda `nekiMetod()`, argumenti su konkretne vrednosti `17`, `0.69` i `false` čiji

se tipovi slažu sa tipovima odgovarajućih parametara `n`, `x` i `test` u definiciji metoda `nekiMetod()`. U drugom primeru poziva metoda `nekiMetod()`, argumenti su aktuelne vrednosti promenljivih `i` i `a`, kao i logička vrednost izraza `b >= 10`. Zbog toga, deklarisani tipovi argumenata `i` i `a` moraju se slagati sa tipovima odgovarajućih parametara `n` i `x`. Najzad, u trećem primeru poziva metoda `nekiMetod()`, drugi i treći argument su vrednosti izraza `Math.sqrt(y+1)` i `s.equals(t)`, čiji se tipovi slažu sa tipovima odgovarajućih parametara `x` i `test`.

Parametri u definiciji i argumenti u pozivu nekog metoda moraju se slagati ne samo po tipu, nego i po redosledu i broju. To znači da se argumenti u pozivu metoda moraju pisati istim onim redom kojim su navedeni odgovarajući parametri u definiciji metoda. Isto tako, broj argumenata u pozivu metoda mora biti tačno jednak broju parametara u definiciji metoda. Na primer, pogrešni pozivi metoda `nekiMetod()` su:

- `nekiMetod(0.69, 17, false);`
- `nekiMetod(i, a);`

U svetlu ovih novih detalja o argumentima metoda, izvršavanje naredbe poziva nekog metoda mora se bolje precizirati. Naime, pre izvršavanja naredbi u telu definicije metoda, izračunavaju se izrazi koji su navedeni kao argumenti u pozivu metoda i njihove vrednosti se dodeljuju odgovarajućim parametrima. Ovaj način prenošenja vrednosti argumenata odgovarajućim parametrima metoda u Javi tehnički se naziva *prenošenje po vrednosti*.

Pozivom metoda se dakle izvršava telo metoda, ali sa inicijalnim vrednostima parametara koje su prethodno dobijene izračunavanjem vrednosti odgovarajućih argumenata. To znači da, recimo, naredba poziva metoda `nekiMetod()` u ovom obliku

```
nekiMetod(23, Math.sqrt(y+1), s.equals(t));
```

ima isti efekat kao sledeći blok naredbi:

```
{
   // Alokacija i inicijalizacija parametara metoda nekiMetod()
   int n = 23;
   double x = Math.sqrt(y+1);
   boolean test = s.equals(t);

   // Izvršavanje naredbi u telu metoda nekiMetod()
   .
   . // Telo metoda
   .
}
```

Obratite pažnju na to da, zbog prenošenja argumenata po vrednosti, promene vrednosti parametara unutar tela metoda nemaju nikakav efekat na ostatak programa. (Ovo je zaista tako za parametre primitivnih tipova; za parametre klasnih tipova je cela istina malo komplikovanija.) Pretpostavimo, na primer, da su definisana dva metoda `prikaži()` i `dodaj1()` na sledeći način:

```
void prikaži() {
   int x = 1;
   dodaj1(x);
   System.out.println("x = " + x),
}

void dodaj1(int a) {
   a = a + 1;
}
```

U metodu `prikaži()` se definiše promenljiva `x` i dodeljuje joj se početna vrednost 1, a zatim se poziva metod `dodaj1()` radi uvećanja vrednosti ove promenljiva za 1. Međutim, to nije stvarni rezultat ovog metoda, jer se pozivom metoda `prikaži()` na ekranu prikazuje `x = 1`, a ne `x = 2`. Razlog za ovo je što promena vrednosti parametra `a` unutar metoda `dodaj1()` nema efekta van tog metoda, pa ni uticaja na promenljivu `x` u metodu `prikaži()`, iako je promenljiva `x` argument poziva metoda `dodaj1(x)`. Preciznije, nakon dodele broja 1 promenljivoj `x` u metodu `prikaži()`, izvršava se poziv `dodaj1(x)` sa argumentom koji je jednak vrednosti promenljive `x`. Izvršavanjem tog poziva se najpre inicijalizuje parametar `a` vrednošću argumenta `x`, tj. brojem 1 koji je trenutna vrednost promenljive `x`. Zatim se parametar `a` uvećava za 1 i njegova nova vrednost postaje 2. Nakon toga se završava metod `dodaj1()`, što znači da njegovo izvršavanje nije uticalo na promenljivu `x` u metodu `prikaži()`. Zbog toga se na kraju u metodu `prikaži()` prikazuje neizmenjena vrednost 1 promenljive `x`.

✧ ✧ ✧

Na početku ovog poglavlja je istaknuto da se definicija svakog metoda u Javi mora nalaziti unutar neke klase. S druge strane, naredba poziva jednog metoda u najprostijem obliku se može navesti na bilo kom propisanom mestu unutar definicije drugog metoda u klasi. (U stvari, poziv jednog metoda se može nalaziti i unutar definicije istog metoda. U tom slučaju je reč o rekurzivnom metodu, o čemu se opširnije govori u odeljku 6.7 ovog poglavlja.)

Pretpostavimo, na primer, da je metod `nekiMetod()` definisan u klasi

NekaKlasa, koja pored toga sadrži i definiciju metoda main(). Struktura klase NekaKlasa zato ima otprilike ovaj oblik:

```
public class NekaKlasa {

   // Definicija metoda main()
   public static void main(String[] args) {
      . . .
      nekiMetod(17, 0.69, false);                 // poziv metoda
      . . .
      nekiMetod(23, Math.sqrt(y+1), s.equals(t)); // poziv metoda
      . . .
   }

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

Obratite pažnju na to da se definicije metoda main() i nekiMetod() u klasi NekaKlasa nalaze jedna iza druge. To je posledica opšteg pravila u Javi po kojem nije dozvoljeno pisati jedan metod ugnježđen u drugi. Naravno, u metodu main() se može pozivati metod nekiMetod() za izvršavanje, ali se definicija metoda nekiMetod() ne može pisati unutar metoda main(). S druge strane, redosled pisanja metoda (opštije, svih članova) u nekoj klasi u Javi nije bitan.

Ovaj prost način pozivanja metoda navođenjem njegovog imena i eventualnih argumenata moguć je samo ukoliko se naredba poziva metoda i njegova definicija nalaze unutar iste klase. Ako se poziva metod koji je definisan u drugoj klasi, onda se mora koristiti notacija sa tačkom. Pri tome, način pozivanja takvog metoda u Javi zavisi od toga da li je metod u drugoj klasi definisan da bude statički (klasni) ili nestatički (objektni), odnosno da li je metod definisan sa modifikatorom static ili bez njega.

Ako je metod statički i poziva se izvan klase u kojoj je definisan, onda se mora koristiti tačka-notacija za njegov poziv u obliku:

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

Deo *ime-klase* u ovom obliku poziva metoda je ime one klase u kojoj je definisan metod koji se poziva. Statički metodi pripadaju celoj klasi, pa upotreba imena klase u tačka-notaciji ukazuje na to u kojoj klasi treba naći

definiciju metoda koji se poziva. (Statički metod se može pozvati i preko nekog objekta klase u kojoj je metod definisan, odnosno na isti način na koji se poziva nestatički metod. Međutim, takav način pozivanja statičkog metoda se ne preporučuje.)

U programima u prethodnom delu knjige već je korišćen ovaj način 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 klasama se mora koristiti tačka-notacija za njihovo pozivanje. Ali izvršavanje naredbe poziva, recimo `Math.sqrt(y+1)`, odvija se na standardni način: argument koji se prenosi metodu `sqrt()` je vrednost izraza `y+1`, a rezultat izvršavanja tela tog metoda je kvadratni koren prenesene vrednosti argumenta.

Drugi način pozivanja metoda se koristi za nestatičke (objektne) metode koji pripadaju pojedinim objektima, a ne celoj klasi. Zbog toga se oni moraju pozivati uz odgovarajući objekat koji predstavlja implicitni argument poziva metoda. Opšti oblik tačka-notacije poziva nestatičkog (objektnog) metoda izvan klase u kojoj je definisan je:

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

Deo *ime-objekta* u ovom obliku je zapravo jedna promenljiva koja sadrži referencu na onaj objekat za koji se poziva nestatički (objektni) metod. I ovaj način je korišćen u primerima programa u prethodnom delu knjige, na primer kod učitavanja podataka preko tastature:

```
tastatura.nextInt()
```

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

Drugi primer poziva nestatičkog (objektnog) metoda je naredba

```
rečenica.CharAt(i)
```

kojom se poziva objektni metod `CharAt()` iz klase `String` za objekat klase `String` na koji ukazuje promenljiva `rečenica`. Argument poziva ovog metoda je vrednost promenljive `i` koja je navedena u zagradama. Kako je funkcija metoda `CharAt()` vraćanje znaka na datoj poziciji u stringu, rezultat ovog poziva je *i*-ti znak u stringu koji je referenciran promenljivom `rečenica`.

6.3 Vraćanje rezultata

Metod u Javi 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.

Pozivi metoda koji vraćaju jednu vrednost se navodi tamo gde je ta vrednost potrebna u programu. To je obično na desnoj strani znaka jednakosti kod naredbe dodele, 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.)

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

```
return izraz;
```

Ovde se tip vrednosti izraza iza službene reči `return` mora slagati sa navedenim tipom rezultata metoda koji se definiše. To znači da naredba dodele tog izraza promenljivoj, čiji je tip jednak tipu rezultata metoda, mora biti dozvoljena uz eventualnu konverziju tipa.

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

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

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

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

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

Kao što je ranije napomenuto, definisanjem metoda se ništa ne izvršava, nego se tek pozivanjem metoda, u tački programa gde je potreban rezultat metoda, izvršavaju naredbe u telu metoda uz prethodni prenos argumenata. Zbog toga, pretpostavimo da se na nekom mestu u jednom drugom metodu u programu izvršavaju sledeće dve naredbe:

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

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

U konkretnom slučaju dakle, argumenti poziva metoda `obim()` su promenljive `k1` i `k2`, pa se njihove trenutne vrednosti 3 i 4 redom dodeljuju parametrima ovog metoda `a` i `b`. Zatim se sa ovim inicijalnim vrednostima parametara `a` i `b` izvršava telo metoda `obim()`. Kontrola toka izvršavanja se zato prenosi u telo metoda `obim()` i redom se izvršavaju sve njegove naredbe dok se ne naiđe na naredbu `return`. To znači da se najpre izračunava hipotenuza izrazom `Math.sqrt(a*a + b*b)` čija se vrednost $\sqrt{3*3+4*4} = \sqrt{9+16} = \sqrt{25} = 5$ dodeljuje promenljivoj `c`. Zatim se izvršava naredba `return` tako što se izračunava izraz u produžetku `a + b + c` i dobija njegova vrednost $3+4+5 = 12$. Nakon toga se prekida izvršavanje

metoda `obim()` i izračunata vrednost 12 i kontrola izvršavanja se vraćaju tamo gde je taj metod pozvan. To znači da se nastavlja izvršavanje naredbe dodele `O = obim(k1,k2)` koje je bilo privremeno prekinuto radi izvršavanja poziva metoda `obim()` na desnoj strani znaka jednakosti. Drugim rečima, vraćena vrednost 12 tog poziva se konačno dodeljuje promenljivoj `O`.

Iako u ovom primeru to nije slučaj, obratite pažnju na to da generalno naredba `return` ne mora biti poslednja naredba u telu metoda. Ona se može pisati u bilo kojoj tački u telu metoda u kojoj se želi vratiti vrednost i završiti izvršavanje metoda. Ako se naredba `return` izvršava negde u sredini tela metoda, izvršavanje metoda se odmah prekida. To znači da se sve eventualne druge naredbe iza naredbe `return` preskaču i kontrola se odmah vraća na mesto gde je metod pozvan.

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

```
return;
```

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

Upotreba naredbe povratka kod metoda koji ne vraća nijednu vrednost nije obavezna i koristi se kada izvršavanje takvog metoda treba prekinuti negde u sredini. Ako naredba povratka nije izvršena prilikom izvršavanja tela takvog metoda, njegovo izvršavanje se normalno prekida (i kontrola izvršavanja se vraća na mesto gde je metod pozvan) kada se dođe do kraja izvršavanja tela metoda.

S druge strane, ukoliko metod vraća neku vrednost, u njegovom telu se mora nalaziti bar jedna naredba povratka u punom obliku `return` *`izraz`*, makar to bila poslednja naredba u telu metoda. Ako ih ima više od jedne, sve one moraju imati ovaj pun oblik, jer metod uvek mora vratiti jednu vrednost.

6.4 Primeri metoda

Da bismo ilustrovali praktičnu primenu metoda, u ovom odeljku se prikazuje nekoliko kompletnih programa koji pored glavnog metoda `main()` koriste dodatne metode. Ovi programi rešavaju iste probleme za koje smo

već pokazali programe u prethodnom delu knjige 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 se tome ne posvećuje veća pažnja. Imajte ipak na umu da za složeniji problem nije lako odgovoriti na pitanje kako je najbolje podeliti njegovo rešenje u manje zadatke koje obavljaju posebni metodi.

✧ ✧ ✧

Prvi primer je redizajn programa za igru pogađanja broja iz odeljka 5.3. 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 6.1: 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 program 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 je prikazan program kojim se određuje da li je neki broj prost. Obratite pažnju na to da metod `jeProst()` u programu ima jedan parametar koji predstavlja ulazni broj, a rezultat tog metoda je logička vrednost `true` ili `false` zavisno od toga da li je taj broj prost ili ne.

Listing 6.2: Određivanje da li je broj prost

```
import java.util.*;

public class ProstBroj {

   public static void main(String[] args) {

      int n;  // dati broj

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

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

      while (true) {
         System.out.print("Unesite broj veći od 1 (0 za kraj): ");
```

```
            n = tastatura.nextInt();
            if (n == 0)
               break;
            System.out.print("Taj broj ");
            if (jeProst(n))  // da li je broj n prost?
               System.out.print("je ");
            else
               System.out.print("nije ");
            System.out.println("prost.");
         }
      }

      public static boolean jeProst(int n) {

         int maxDelilac = (int)Math.sqrt(n); // najveći delilac

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

✧ ✧ ✧

U trećem primeru je prikazan program kojim se proveravaju rečenice da li su palindromi. Zadatak programa je podeljen između dva dodatna metoda, pored glavnog metoda `main()`. Prvi metod `preuredi()`, pretvaranjem svih slova u mala slova i zanemarivanjem nevažnih znakova, preuređuje ulaznu rečenicu u oblik koji je lakši za proveravanje da li je palindrom. Drugi metod `obrni()` vraća neku rečenicu u obrnutom redosledu. Određivanje da li je neka rečenica palindrom svodi se onda na proveru da li je preuređena verzija te rečenice jednaka svom obrnutom obliku.

Listing 6.3: 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;  // preuređen ulazni red znakova
    Scanner tastatura = new Scanner(System.in);

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

       // Preuređivanje ulaznog reda znakova rečenice
       rečenica = preuredi(red);

       // Da li su iste rečenica i obrnuta rečenica?
       System.out.print("Ta rečenica ");
       if (rečenica.equals(obrni(rečenica)))
          System.out.print("je ");
       else
          System.out.print("nije ");
       System.out.println("palindrom.");

    } while (red.length() != 0);
}

public static String preuredi(String niska) {

    String preuređenaNiska = "";

    // 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.isLetter(znak))
          preuređenaNiska += znak;
    }
    return preuređenaNiska;
}

public static String obrni(String niska) {

    String obrnutaNiska = "";

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

```
        }
        return obrnutaNiska;
    }
}
```

6.5 Preopterećeni metodi

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

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

ima potpis:

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

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

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

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

Potpis ovog metoda je:

```
nekiMetod(String)
```

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

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

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

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

6.6 Lokalne i globalne promenljive

Definicije metoda moraju se nalaziti unutar klasa. Ali neka klasa može pored metoda obuhvatati i definicije promenljivih. Ove promenljive, kada se posmatraju iz perspektive klase, često se nazivaju *polja* klase. Na primer, klasa `Zgrada` koja je korišćena na strani 23,

```
class Zgrada
{
   int brojSpratova;
   String boja;
   String adresa;

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

sadrži tri polja `brojSpratova`, `boja` i `adresa` koja su definisana izvan metoda `okreči()` i `dozidaj()`.

Polja su u klasi definisana izvan svih metoda, pa kada se ove promenljive posmatraju iz perspektive metoda, za njih kažemo da su to *globalne* promenljive za sve metode klase. Analogno, za promenljive koje su definisane unutar nekog metoda se kaže da su to *lokalne* promenljive za taj metod.

Prva razlika između lokalnih i globalnih promenljivih u Javi je u načinu njihove inicijalizacije. Lokalne promenljive metoda se ne inicijalizuju prilikom izvršavanja tog metoda. Stoga one moraju imati nesumnjivo dodeljenu vrednost u metodu pre nego što se mogu koristiti. S druge strane, globalne promenljive (polja) se automatski inicijalizuju unapred definisanim

vrednostima. Za numeričke promenljive, ta vrednost je nula; za logičke promenljive, ta vrednost je `false`; i za znakovne promenljive, ta vrednost je Unicode znak `'\u0000'`. (Za klasne promenljive, ta inicijalna vrednost je specijalna vrednost `null`.)

Druga, važnija razlika između lokalnih i globalnih promenljivih je u dužini njihovog trajanja. Naime, svaka promenljiva ima svoj „životni vek" u memoriji tokom izvršavanja programa. Postojanje neke promenljive u programu počinje od trenutka izvršavanja naredbe definicije promenljive. Time se rezerviše memorijska lokacija odgovarajuće veličine za promenljivu. Ovaj postupak se 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 *dealociranje* promenljive, a momenat njegovog dešavanja tokom izvršavanja programa zavisi od vrste promenljive.

Tri vrste promenljivih koje se mogu koristiti u telu nekog metoda su: lokalne promenljive definisane unutar metoda, parametri metoda i globalne promenljive koje su definisane izvan metoda, ali u istoj klasi kao i metod. Lokalne promenljive se dealociraju čim se izvrši poslednja naredba metoda u kojem su definisane 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 su promenljive koje dobijaju početne vrednosti izračunavanjem argumenata prilikom izvršavanja naredbe poziva metoda. Parametri imaju konceptualno istu ulogu kao lokalne promenljive, pa za parametre važe ista pravila prethodno opisana za lokalne promenljive.

Globalne promenljive metoda definisane 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 to što su globalne promenljive u nekoj klasi nezavisne od metoda te klase i uvek postoje dok se metodi te klase izvršavaju. Zato globalne promenljive mogu koristiti svi metodi u istoj klasi. To znači da promene vrednosti globalne promenljive u jednom metodu ima-

ju 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 koji pripadaju samo jednom metodu.

Primer: igra pogađanja broja sa ograničenjem

U ovom primeru je prikazana još jedna verzija programa za igru pogađanja broja u kome se dodatno vodi statistika o tome koliko je puta korisnik pogodio zamišljene brojeve. Naravno, da bi ovo imalo smisla, broj pogađanja jednog broja mora biti ograničen tako da se korisniku računa da nije pogodio zamišljen broj ukoliko ga ne pogodi u ograničenom broju dozvoljenih pokušaja.

Program u listingu 6.4 je organizovan na isti način kao prethodna verzija ovog programa na strani 152: sve dok korisnik želi da pogađa novi broj, u glavnom metodu `main()` se u petlji poziva metod `pogodiBroj()` čiji je zadatak pogađanje jednog zamišljenog broja. Kako je potrebno voditi računa i o tome koliko puta je korisnik pogodio zamišljene brojeve, uvodi se poseban brojač `brojPogodaka` koji se uvećava za 1 u metodu `pogodiBroj()` svaki put kada korisnik pogodi zamišljen broj. Vrednost tog brojača se prikazuje na kraju metoda `main()` kada korisnik odustane od daljeg igranja. Obratite ovde pažnju na to da brojač `brojPogodaka` ne može biti lokalna promenljiva, jer se koristi unutar dva metoda, `main()` i `pogodiBroj()`. Kako brojač `brojPogodaka` ne može biti ni objektna promenljiva, jer ga koriste statički metodi, mora biti definisan da bude statička globalna promenljiva.

Listing 6.4: Igra pogađanja broja sa ograničenjem

```
import java.util.*;

public class PogađanjeBroja {

   static int brojPogodaka = 0;       // brojač pogođenih brojeva
   final static int MAX_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_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 program 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_POKUŠAJA) {
            System.out.print("Niste pogodili broj iz ");
            System.out.println(MAX_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;
         }
      }
   }
}
```

Program u listingu 6.4 zaslužuje nekoliko posebnih napomena:

- Da bi rešenje bilo bolje, u programu su pored brojača `brojPogodaka` definisane još dve statičke *globalne* promenljive, mada to nije neophodno kao u slučaju promenljive `brojPogodaka`. Prva promenljiva je `tastatura` koja predstavlja objekat fizičkog uređaja tastature. Ovu promenljivu koriste oba metoda `main()` i `pogodiBroj()`, pa je ta promenljiva izvučena 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_POKUŠAJA` čija vrednost 5 određuje najviše pet prilika za pogađanje jednog zamišljenog broja.
- Petlja u metodu `pogodiBroj()` kojom se realizuje pogađanje zamišljenog broja je tipični slučaj kada je dobro koristiti beskonačnu petlju. (Radi promene, ovde se koristi `for` petlja za to.) Naime, uslov prestanka pogađanja broja je vrlo komplikovan i običnom petljom bi se dobilo prilično neelegantno rešenje. S druge strane, logika programa se lako prepoznaje ukoliko se koristi beskonačna petlja i naredbe `break`. Obratite ipak pažnju na to da je bitan redosled logičkih uslova kod višestruke naredbe `if`, 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.

- Kao što je ranije spomenuto, globalne promenljive (polja) se automatski inicijalizuju unapred definisanim vrednostima. Za numeričke promenljive ta početna vrednost je nula, pa bi globalna promenljiva `brojPogodaka` automatski dobila vrednost nula na početku. Ipak, radi svake sigurnosti i bolje čitljivosti, ova globalna promenljiva je eksplicitno inicijalizovana nulom.

Oblast važenja imena

Do sada smo govorili o više programskih konstrukcija koje se označavaju imenima u tekstu programu. Tu spadaju promenljive, metodi, parametri metoda, klase i tako dalje. S druge strane, postojanje nekih od ovih elemenata se dinamički menja tokom izvršavanja programa. Da bi se izbegle greške usled neusklađenosti korišćenja ovih elemenata u statičkom tekstu programa i njihovog „životnog veka" u dinamičkom procesu izvršavanja programa, u Javi se moraju poštovati posebna 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*).

Promenljiva (globalna ili lokalna) dobija ime u naredbi definicije 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 definicije.[1] S druge strane, oblast važenja imena lokalne promenljive je od mesta njene definicije do kraja bloka u kome se ta definicija nalazi. Kako telo nekog metoda formalno predstavlja blok naredbi, to se lokalna promenljiva može koristiti samo u metodu u kojem je definisana.

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

```
public class IgraSaKartama {

   static String pobednik;  // globalna promenljiva (polje)
```

[1] Zato je moguće pisati definicije polja klase na kraju definicije klase, što je po nekima bolji način.

```
    . . .

    static void odigrajIgru() {

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

U telu metoda `odigrajIgru()`, ime `pobednik` se odnosi na lokalnu promenljivu. U ostatku klase `IgraSaKartama`, ime `pobednik` se odnosi na globalnu promenljivu (polje), osim ako ta globalna promenljiva takođe nije zaklonjeno istim imenom lokalne promenljive ili parametra u nekom drugom metodu. Ako je ipak potrebno koristiti globalnu promenljivu `pobednik` unutar metoda `odigrajIgru()` u kome je zaklonjena, onda se mora pisati njeno puno ime `IgraSaKartama.pobednik`. Ovo puno ime polja `pobednik` klase `IgraSaKartama` mora se pisati i u nekoj drugoj klasi ukoliko se koristi to polje.

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

Drugi specijalni slučaj oblasti važenja imena promenljivih pojavljuje se kod definisanja kontrolne promenljive (brojača) petlje unutar naredbe `for`. Na primer:

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

U ovom primeru izgleda da je kontrolna promenljiva `i` definisana lokalno u odnosu na metod koji sadrži `for` petlju. Ipak, oblast važenja ovako definisane kontrolne promenljive je samo kontrolni deo i telo naredbe `for`, odnosno ne proteže se do kraja tela metoda koji sadrži naredbu `for`. Zbog toga se u Javi vrednost kontrolne promenljive (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 brojača 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 metoda ili lokalne promenljive u oblasti njihovog važenja, čak ni u ugnježđenom bloku. Na primer, ni ovo nije dozvoljeno:

```
void nekiMetod(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 metoda ne mogu redefinisati. S druge strane, izvan oblasti važenja lokalne promenljive ili parametra metoda se njihova imena mogu ponovo koristiti. Na primer:

```
void nekiMetod(int n) {
    while (n > 0) {
        int x;
        . . .
        // Kraj oblasti važenja imena x
    }
    while (n < 0) {
        int x;  // OK: ime x se može ponovo 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 tog metoda. Takav metod se naziva *rekurzivni* metod, o čemu će više reči biti u narednom odeljku ovog poglavlja.

Ovo opšte pravilo ima dodatni uslov s obzirom na to da metodi mogu biti statički ili objektni. Naime, objektni metodi i polja klase, koji pripa-

daju 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 — naime, 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 koji se može iskoristiti za definisanje tipa promenljivih, kao i parametara i tipa rezultata metoda. To znači da je u Javi dozvoljeno definisati klasu, na primer, 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.

6.7 Rekurzivni metodi

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

Razmotrimo, na primer, izračunavanje stepena x^n, gde je x neki realan broj različit od nule i n ceo broj veći ili jednak nuli. Mada se za ovaj problem može iskoristiti gotov metod `Math.pow(x,n)`, pošto stepen x^n predstavlja množenje broja x sa samim sobom n puta, nije teško napisati i sopstveni metod za izračunavanje stepena x^n. Naime, n-ti stepen broja x možemo dobiti uzastopnim množenjem broja x sa delimično izračunatim stepenima

x^i za $i = 0, 1, \ldots, n-1$. Naredni metod `stepen()` koristi `for` petlju radi realizacije ovog iterativnog postupka:

```
double stepen(double x, int n) {
   double y = 1;  // rezultat

   for (int i = 0; i < n; i++)
      y = y * x;

   return y;
}
```

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

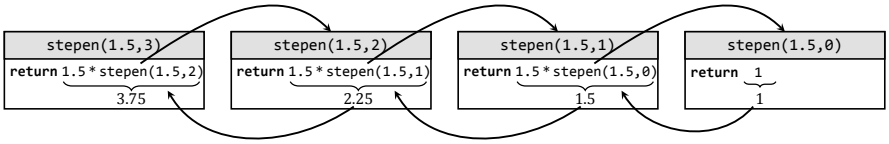
```
double stepen(double x, int n) {
   if (n == 0)
      return 1;
   else
      return x * stepen(x,n-1);
}
```

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

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

Mehanizam pozivanja i izvršavanja rekurzivnih metoda se ne razlikuje od uobičajenog postupka za obične, nerekurzivne metode: argumenti u pozivu se prenose parametrima metoda i s tim početnim vrednostima se izvršava telo metoda. Na primer, na slici 6.1 je ilustrovano izvršavanje poziva drugog, rekurzivnog metoda `stepen(1.5,3)` i dobijanje njegovog rezultata $1.5^3 = 3.75$.

Rekurzivni metodi u opštem slučaju rešavaju neki zadatak svođenjem polaznog problema na sličan prostiji problem (ili više njih). Tako, rekurzivni metod `stepen()` rešava problem izračunavanja n-tog stepena nekog



Slika 6.1: Rekurzivno izvršavanje poziva `stepen(1.5,3)`.

broja svođenjem na problem izračunavanja $(n-1)$-og stepena istog broja. Prostiji zadatak (ili više njih) se onda rešava rekurzivnim pozivom metoda koji se definiše.

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

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

Poziv rekurzivnog metoda dakle 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.

Rekurzivni način rešavanja problema u programiranju zahteva drugačiji pristup rešavanju problema od klasičnog, iterativnog. Najbolji način da se savlada rekurzivna tehnika programiranja je kroz praksu, te se u nastavku ovog odeljka posvećuje pažnja još nekim primerima rekurzivnih metoda.

Primer: najveći zajednički delilac

U ovom primeru se pokazuje rekurzivno rešenje za problem izračunavanja najvećeg zajedničkog delioca dva pozitivna cela broja x i y. Ovaj naizgled jednostavan primer ima značajnu primenu u kriptografiji.

Najveći zajednički delilac dva pozitivna cela broja x i y, koji se označava $\text{nzd}(x, y)$, predstavlja najveći ceo broj koji deli bez ostatka oba broja x i y. Da li svaka dva pozitivna cela broja imaju najveći zajednički delilac? Za svaki par pozitivnih celih brojeva je jasno da je uvek broj 1 najmanji zajednički delilac. Pored toga, 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 intervalu između brojeva 1 i $\min\{x, y\}$.

Ova napomena o egzistenciji najvećeg zajedničkog delioca za x i y može se iskoristiti za iterativni način njegovog izračunavanja. Naime, ukoliko se počne od manjeg od dva pozitivna cela broja x i y i zatim se $\min\{x, y\}$ i svi manji brojevi redom proveravaju da li su zajednički delioci za x i y, prvi nađeni zajednički delilac biće ujedno i najveći.

Pod pretpostavkom $x \geq y$, drugi način za izračunavanje najvećeg zajedničkog delioca za x i y se zasniva na činjenici da je $\text{nzd}(x, y)$ jednak najvećem zajedničkom deliocu broja y i ostatka pri deljenju x sa y. Ako znak % označava operator izračunavanja ostatka pri celobrojnom deljenju, onda se ovo može preciznije izraziti na sledeći način:[2]

$$\text{nzd}(x, y) = \begin{cases} y, & \text{ako je } x \,\%\, y = 0 \\ \text{nzd}(y, x \,\%\, y), & \text{ako je } x \,\%\, y \neq 0 \end{cases}$$

Primetimo da je ovo rekurzivna definicija za $\text{nzd}(x, y)$, jer se izračunavanje najvećeg zajedničkog delioca za dva broja x i y svodi na izračunavanje najvećeg zajedničkog delioca za dva manja broja y i $x \,\%\, y$. Na osnovu toga je onda 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             // opšti slučaj
```

[2]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. Dakle, budući da parovi brojeva (x, y) i $(y, x \,\%\, y)$ imaju isti konačni skup zajedničkih delioca, to su i njihovi najveći zajednički delioci jednaki.

```
        return nzd(y,x % y);
    }
```

U telu ovog metoda unutar `else` dela naredbe `if`, obratite pažnju na to da je u rekurzivnom pozivu `nzd(y,x%y)` drugi argument striktno manji od polaznog drugog parametra `y` u zaglavlju definicije metoda `nzd()`. (Naime, ostatak pri deljenju s nekim brojem uvek je striktno manji od tog broja.) To znači da se ostatak pri celobrojnom deljenju u lancu rekurzivnih poziva stalno smanjuje i zato taj ostatak u jednom trenutku mora biti nula. A tada će se lanac rekurzivnih poziva prekinuti, jer se onda rezultat neposredno dobija kao vrednost drugog argumenta poslednjeg poziva u lancu.

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 koja su oba jednaka 1, a svaki sledeći broj u nizu se zatim dobija kao zbir prethodna dva broja niza. Tako, treći broj niza je zbir drugog i prvog, odnosno $1 + 1 = 2$; četvrti broj niza je zbir trećeg i drugog, odnosno $2 + 1 = 3$; peti broj niza je zbir četvrtog i trećeg, odnosno $3 + 2 = 5$ i tako dalje. Prema tome, početni deo beskonačnog Fibonačijevog niza brojeva je:

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

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 Fibonačijevog niza za $n > 2$ jednak je zbiru prethodna dva, odnosno $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza.

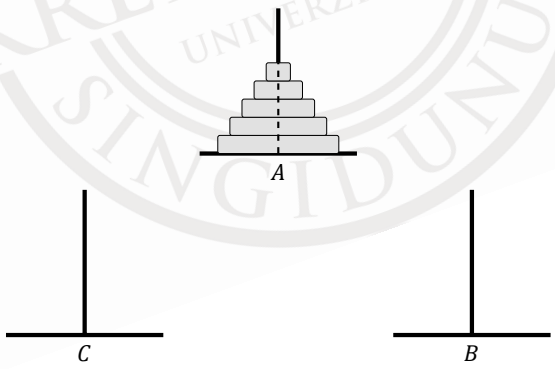
Ukoliko treba napisati metod za izračunavanje n-tog broja Fibonačijevog niza, prethodna rekurzivna definicija za izračunavanje tog broja može se neposredno iskoristiti za rekurzivno rešenje:

```
// Pretpostavlja se da je n >= 1
int fib(int n) {
   if (n <= 2)  // bazni slučaj
      return 1;
   else         // opšti slučaj
      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 te 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 određen broj diskova različite veličine poređani na jednom od tri stuba u opadajućem redosledu svoje veličine od podnožja tog stuba ka vrhu. Na slici 6.2 je prikazana početna konfiguracija ove igre za pet diskova ukoliko su tri stuba trougaono raspoređeni.



Slika 6.2: Početna konfiguracija sa pet diskova u igri Hanojskih kula.

Svi diskovi treba da se premeste na drugi stub koristeći treći stub kao pomoćni, ali se neki disk samo sa vrha jednog stuba sme premesti na vrh drugog stuba i nikad se veći disk ne sme staviti iznad manjeg diska. Preciznije, igra Hanojske kule se sastoji od tri stuba A, B, C i n diskova različitih 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 sa stuba A na stub C koristeći stub B kao pomoćni i poštujući dva pravila:

1. Samo se disk sa vrha jednog stuba može premestiti na vrh drugog stuba.
2. Nikad se veći disk ne može nalaziti iznad manjeg diska.

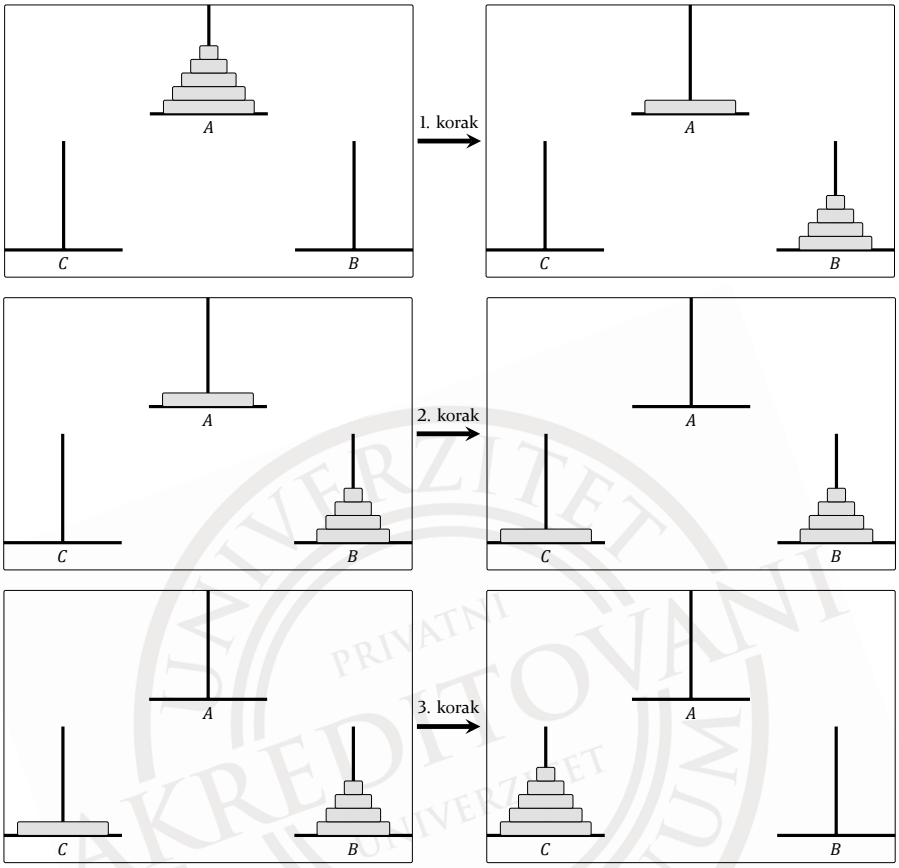
Primetimo da se ovim pravilima podrazumeva da se samo po jedan disk može premeštati u svakom potezu, kao i da se svi diskovi u svakom trenutku moraju nalaziti samo na stubovima.

Iterativno rešenje igre Hanojskih kula, koje navodimo bez mnogo objašnjenja samo radi poređenja sa rekurzivnim rešenjem, sastoji se niza koraka: 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, 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 može se 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 može se onda rešiti rekurzivno u tri koraka:

1. Primenjujući (rekurzivno) postupak za rešenje prvog zadatka, najpre se premešta $n-1$ najmanjih diskova sa stuba A na stub B koristeći stub C kao pomoćni.
2. Nakon prvog koraka ostaje samo najveći disk na stubu A. Zato se u drugom koraku premešta taj najveći disk sa stuba A na slobodni stub C.
3. U trećem koraku, primenjujući (opet rekurzivno) postupak za rešenje drugog zadatka, svih $n-1$ najmanjih diskova koji se nalaze na stubu B premeštaju se na stub C koristeći stub A kao pomoćni.

Ovaj rekurzivni postupak je ilustrovan na slici 6.3. Na toj slici su prikazana tri koraka tog postupka od početne do završne konfiguracije igre Hanojskih kula sa pet diskova.



Slika 6.3: Rekurzivno rešenje za igru Hanojskih kula.

Obratite pažnju na to koliko je rekurzivno rešenje problema Hanojskih kula konceptualno jednostavnije za razumevanje od iterativnog. Možda još važnije, da je trebalo, verovatno bi čitaoci uz relativno malo truda i sami došli do tog rešenja. Jednostavnost rekurzivnog postupka je prevashodno ono što čini rekurzivnu tehniku važnom, mada su u mnogim slučajevima rekurzivni metodi i efikasniji od iterativnih.

Za realizaciju rekurzivnog rešenja igre Hanojskih kula u Javi, radi bolje preglednosti pretpostavljamo da se jedan disk na vrhu stuba `x` premešta na vrh diskova na stubu `y` pomoću datog metoda `premestiDisk(x,y)`. Imajući to u vidu, rekurzivni metod `premestiDiskove(n,a,b,c)`, kojim se u igri Hanojskih kula premeštaju `n` diskova sa stuba `a` na stub `c` koristeći `b` kao pomoćni stub, ima jednostavan oblik:

```
void premestiDiskove(int n, char a, char b, char c) {

   if (n == 1)  // bazni slučaj
      premestiDisk(a,c);
   else {       // opšti slučaj
      premestiDiskove(n-1,a,c,b);
      premestiDisk(a,c);
      premestiDiskove(n-1,b,a,c);
   }
}
```

U listingu 6.5 je prikazan 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 premeštanje 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 6.5: 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("\nRešenje igre za " + n + " 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 {       // opšti slučaj
         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 u listingu 6.5 da je metodu `premestiDiskove()` dodat modifikator `static`, jer se poziva iz statičkog metoda `main()`. Slično važi i za metod `premestiDisk()`.

Pitanja i zadaci

1. Ako metod ne vraća nijednu vrednost, koja se službena reč koristi za njegov tip rezultata u definiciji tog metoda?

A. `void` B. `return` C. `public` D. `static`

2. Šta je potpis nekog metoda?

A. Ime metoda B. Ime metoda i lista parametara C. Tip rezultata, ime metoda i lista parametara D. Lista parametara

3. Analizirajte sledeći metod:

```
public double nađi(int x, int y, boolean b) {
   .
   . // Telo metoda
   .
}
```

A. Ime metoda je `public`.
B. Ime metoda je `nađi`.
C. Tip rezultata metoda je `int`.
D. Tip rezultata metoda je `boolean`.
E. Tip rezultata metoda je `double`.
F. Broj parametara metoda je tri.
G. Broj parametara metoda je šest.

4. Koji metod moraju imati svi Java programi?

A. `public static Main(String[] args)`
B. `public static Main(String args[])`
C. `public void main(String[] args)`
D. `public static void main(String[] args)`
E. `public static main(String[] args)`

5. Kako se navode argumenti nekog metoda u naredbi poziva tog metoda?

A. Unutar uglastih (srednjih) zagrada.
B. Unutar običnih (malih) zagrada.
C. Unutar vitičastih (velikih) zagrada.
D. Unutar dvostrukih apostrofa.
E. Unutar jednostrukih apostrofa.

6. Vrednosti argumenata u pozivu metoda se prenose odgovarajućim parametrima metoda. Kako se naziva ovaj način prenošenja argumenata?

A. Prenošenje po potrebi.
B. Prenošenje po vrednosti.
C. Prenošenje po referenci.
D. Prenošenje po imenu.

7. Koja se naredba koristi unutar tela nekog metoda za povratak iz tog metoda i vraćanje rezultata tog metoda?

A. `void` B. `return` C. `public` D. `static`

8. Da li naredba `return` u ovom metodu izaziva grešku?

```
public static void main(String[] args) {
   int max = 0;
   if (max != 0)
      System.out.println(max);
   else
      return;
}
```

A. Da B. Ne C. Zavisi

9. Da li poziv metoda `pow()` u ovom metodu izaziva grešku?

```
public static void main(String[] args) {
   Math.pow(2,4);
}
```

A. Da B. Ne C. Zavisi

10. Šta je rezultat poziva `nPrint("a",4)` ukoliko je metod `nPrint()` definisan na sledeći način?

```
void nPrint(String poruka, int n) {
   while (n > 0) {
      System.out.print(poruka);
      n--;
   }
}
```

A. Na ekranu se prikazuje `aaaaa`.
B. Na ekranu se prikazuje `aaaa`.
C. Na ekranu se prikazuje `aaa`.
D. Na ekranu se prikazuje `aa`.
E. Na ekranu se prikazuje `a`.

11. Neka je metod `nPrint()` definisan na sledeći način:

```
void nPrint(String poruka, int n) {
   while (n > 0) {
      System.out.print(poruka);
      n--;
   }
}
```

Koju vrednost ima promenljiva `k` posle izvršavanja ovog programskog fragmenta?

```
int k = 2;
nPrint("Java je kul!", k);
```

A. 0 B. 1 C. 2 D. 3

12. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      System.out.println(xMetod(5));
   }

   public double xMetod(int n, double t) {
      System.out.println("double");
      return t;
   }

   public int xMetod(int n) {
      System.out.println("int");
      return n;
   }
}
```

A. Program na ekranu prikazuje `int` i zatim `5`.
B. Program na ekranu prikazuje `double` i zatim `5`.
C. Program na ekranu prikazuje `double`.
D. Program na ekranu prikazuje `int`.
E. Program ima grešku, jer se ne može odrediti koju verziju preopterećenog metoda `xMetod()` treba pozvati.

13. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      System.out.println(m(17));
   }

   public int m(int n) {
      return n;
   }

   public void m(int n) {
```

```
        System.out.println(n);
    }
}
```

A. Program ima grešku, jer se ne može odrediti koju verziju preopterećenog metoda `m()` treba pozvati.
B. Program ima grešku, jer je druga verzija preopterećenog metoda `m()` definisana ali se nigde ne poziva.
C. Program se normalno izvršava i prikazuje `17` jedanput.
D. Program se normalno izvršava i prikazuje `17` dvaput.

14. Kako se naziva promenljiva koja je definisana unutar nekog metoda?

A. Globalna promenljiva B. Statička promenljiva C. Blokovska promenljiva D. Lokalna promenljiva

15. Koju vrednost ima promenljiva `k` posle izvršavanja ovog bloka?

```
{
   int k = 1;
   System.out.println(k + 1);
}
```

A. 0 B. 1 C. 2 D. `k` ne postoji van bloka

16. Koje su od ovih rečenica o rekurzivnim metodima tačne?

A. Rekurzivni metodi su oni koji pozivaju sami sebe, bilo direktno ili indirektno.
B. Rekurzivni metodi se pozivaju drugačije od nerekurzivnih metoda.
C. Rekurzivni metodi rešavaju neki zadatak svođenjem polaznog problema na sličan prostiji problem (ili više njih).
D. Svaki rekurzivni metod mora imati *bazni slučaj* za najprostiji zadatak čije se rešenje ne dobija rekurzivnim pozivom.

17. Analizirajte sledeći rekurzivni metod:

```
public long fakt(int n) {
   return n * fakt(n - 1);
}
```

A. Rezultat poziva `fakt(3)` je 2.
B. Rezultat poziva `fakt(3)` je 3.

C. Rezultat poziva `fakt(3)` je 6.

D. Poziv `fakt(3)` izaziva grešku jer proizvodi beskonačan lanac poziva istog metoda `fakt()`.

18. Analizirajte sledeći program:

```
public class Test {
   public static void main(String[] args) {
      rMetod(3);
   }

   public static void rMetod(int n) {
      if (n > 1) {
         System.out.print((n - 1) + " ");
         rMetod(n - 1);
      }
   }
}
```

A. Program proizvodi beskonačan lanac poziva metoda `rMetod()`.

B. Program na ekranu prikazuje `1 2 3`.

C. Program na ekranu prikazuje `3 2 1`.

D. Program na ekranu prikazuje `1 2`.

E. Program na ekranu prikazuje `2 1`.

19. Analizirajte sledeći program:

```
public class Test {
   public static void main(String[] args) {
      rMetod(2);
   }

   public static void rMetod(int n) {
      while (n > 1) {
         System.out.print((n - 1) + " ");
         rMetod(n - 1);
      }
   }
}
```

A. Program na ekranu ne prikazuje ništa.

B. Program na ekranu prikazuje `1 2`.

C. Program na ekranu prikazuje `2 1`.

D. Program na ekranu beskonačno prikazuje `1 1 1 1 1` ....

E. Program na ekranu beskonačno prikazuje `2 2 2 2 2 ...`.

20. Analizirajte sledeći rekurzivni metod:

```
public int rMetod(int n) {
   if (n == 1)
      return 1;
   else
      return n + rMetod(n - 1);
}
```

A. Pozivom `rMetod(5)` se isti metod `rMetod()` poziva još 3 puta.
B. Pozivom `rMetod(5)` se isti metod `rMetod()` poziva još 4 puta.
C. Pozivom `rMetod(5)` se isti metod `rMetod()` poziva još 5 puta.
D. Pozivom `rMetod(5)` se isti metod `rMetod()` poziva još 6 puta.

21. Analizirajte sledeći rekurzivni metod:

```
public int rMetod(int n) {
   if (n == 1)
      return 1;
   else
      return n + rMetod(n - 1);
}
```

A. Rezultat poziva `rMetod(5)` je 5.
B. Rezultat poziva `rMetod(5)` je 10.
C. Rezultat poziva `rMetod(5)` je 15.
D. Rezultat poziva `rMetod(5)` proizvodi beskonačan lanac poziva istog metoda `rMetod()`.

22. Šta je bazni slučaj u ovom rekurzivnom metodu?

```
public int rMetod(int n) {
   if (n == 1)
      return 1;
   else
      return n + rMetod(n - 1);
}
```

A. Slučaj kada je n jednako 1.
B. Slučaj kada je n manje od 1.
C. Slučaj kada je n veće od 1.
D. Nema baznog slučaja.

23. Dopunite 4. red sledećeg rekurzivnog metoda za određivanje da li je neki string palindrom:

```
1    public static boolean palindrom(String s) {
2       if (s.length() <= 1) // bazni slučaj
3          return true;
4       else if  ____________________
5          return false;
6       else
7          return palindrom(s.substring(1, s.length() - 1));
8    }
```

A. `(s.charAt(0) != s.charAt(s.length() - 1))`

B. `(s.charAt(0) != s.charAt(s.length()))`

C. `((s.charAt(1) != s.charAt(s.length() - 1))`

D. `(s.charAt(1) != s.charAt(s.length()))`

24. Dopunite 12. red sledećeg rekurzivnog metoda za određivanje da li je neki string palindrom:

```
1    public static boolean palindrom(String s) {
2       return palindrom(s, 0, s.length() - 1);
3    }
4
5    public static boolean palindrom(String s,
6                                    int leviKraj, int desniKraj) {
7       if (desniKraj <= leviKraj) // bazni slučaj
8          return true;
9       else if (s.charAt(leviKraj) != s.charAt(desniKraj))
10         return false;
11      else
12         return  ____________________;
13   }
```

A. `palindrom(s)`

B. `palindrom(s, leviKraj, desniKraj)`

C. `palindrom(s, leviKraj + 1, desniKraj)`

D. `palindrom(s, leviKraj, desniKraj - 1)`

E. `palindrom(s, leviKraj + 1, desniKraj - 1)`

25. Dopunite 2. red sledećeg rekurzivnog metoda za sortiranje niza realnih brojeva u rastućem redosledu:

```
public static void sortiraj(double[] niz) {
   ____________________;
}

public static void sortiraj(double[] niz, int n) {
   if (n > 1) {
      // Nalaženje najvećeg elementa niza,
      // kao i njegovog indeksa
      int iMax = 0;
      double max = niz[0];
      for (int i = 1; i <= n; i++)
         if (niz[i] > max) {
            max = niz[i];
            iMax = i;
         }

      // Zamena najvećeg i poslednjeg elementa niza
      niz[iMax] = niz[n];
      niz[n] = max;

      // Sortiranje prvog dela niza
      sortiraj(niz, n - 1);
   }
}
```

A. `sortiraj(niz)`
B. `sortiraj(niz, niz.length)`
C. `sortiraj(niz, niz.length - 1)`
D. `sortiraj(niz, niz.length + 1)`

26. Dopunite 15. red sledećeg rekurzivnog metoda za binarno pretraživanje sortiranog celobrojnog niza:

```
public static int nađiBroj(int[] niz, int broj) {
   return nađiBroj(niz, broj, 0, niz.length - 1);
}

public static int nađiBroj(int[] niz, int broj,
                           int leviKraj, int desniKraj) {
   if (leviKraj > desniKraj) // broj nije nađen u nizu
      return -1;

   // Pretraživanje prve ili druge polovine niza
   int sredina = (leviKraj + desniKraj) / 2;
```

```
12        if (broj < niz[sredina])
13           return nađiBroj(niz, broj, leviKraj, sredina - 1);
14        else if (broj > niz[sredina])
15           return  ____________________;
16        else
17           return sredina;
18     }
```

A. `nađiBroj(niz, broj, sredina + 1, leviKraj)`
B. `nađiBroj(niz, broj, sredina - 1, leviKraj)`
C. `nađiBroj(niz, broj, desniKraj, sredina + 1)`
D. `nađiBroj(niz, broj, sredina + 1, desniKraj)`

✧ ✧ ✧

1. Napisati metod `NZD()` kojim se izračunava najveći zajednički delilac dva cela broja koristeći Euklidov algoritam. Taj metod treba testirati pisanjem programa u kome se u metodu `main()` učitavaju dva cela broja i prikazuje njihov najveći zajednički delilac pozivom metoda `NZD()`.

2. Napisati metod `triN1()` kojim se prikazuje niz brojeva za „$3n + 1$ problem" (videti 6. zadatak u poglavlju 5). Taj metod treba testirati pisanjem programa u kome se u metodu `main()` učitava početni broj niza i prikazuje ostatak tog niza pozivom metoda `triN1()`.

3. Napisati metod `kapitalizuj()` kojim se početno slovo svake reči datog stringa pretvara u veliko slovo. Taj metod treba testirati pisanjem programa u kome se u metodu `main()` učitava jedan red teksta i prikazuje njegova „kapitalizovana" verzija pozivom metoda `kapitalizuj()`.

4. Heksadekadne „cifre" su dekadne cifre od 0 do 9 i slova A, B, C, D, E i F. U heksadekadnom sistemu ovi znakovi predstavljaju brojeve od 0 redom do 15. Heksadekadni broj je niz heksadekadnih cifara kao što su $34A7$, FF, $ABCD$ ili 172300.

a) Napisati metod `heksVrednost()` koji kao rezultat daje heksadekadnu vrednost datog znaka. Ako parametar ovog metoda nije jedan od dozvoljenih znakova, vraćena vrednost treba da bude -1.

b) Napisati program kojim se učitava heksadekadni broj i prikazuje dekadna vrednost tog broja pozivom metoda `heksVrednost()`. Ako svi znakovi heksadekadnog broja nisu dozvoljene heksadekadne cifre, program treba da prikaže odgovarajuću poruku o grešci.

5. Napisati program `ZbirDveKocke` kojim se simulira jedan eksperiment sa bacanjem dve kocke za igranje. Program treba da sadrži sledeća tri metoda:

a) Metod `baciZaZbir()` simulira bacanje dve kocke za igru dok njihov zbir ne bude jednak datom mogućem broju od 2 do 12. Rezultat ovog metoda treba da bude broj bacanja koji je izvršen dok se nije desio željeni ishod.

b) Metod `prosekZaZbir()` koristi prethodni metod `baciZaZbir()` za ponavljanje 1000 puta eksperimenta bacanja dve kocke dok se ne dobije dati zbir. Parametar metoda je željeni zbir svakog bacanja, a rezultat metoda je prosečan broj bacanja koji se dobija za taj zbir u 1000 pokušaja.

c) Metod `main()` poziva prethodni metod `prosekZaZbir()` za svaki mogući zbir dve kocke od 2 do 12 i rezultate prikazuje u tabeli sledećeg oblika:

```
Zbir dve kocke      Prosečan broj bacanja
--------------      ---------------------
      2                     35.87
      3                     18.08
      4                     11.95
      .                       .
      .                       .
      .                       .
```

7

Klase

Objektno orijentisano programiranje (OOP) je noviji pristup za rešavanje problema na računaru kojim se rešenje opisuje na prirodan način koji bi se primenio i u svakodnevnom životu. U starijem, proceduralnom programiranju je programer morao da identifikuje računarski postupak za rešenje problema i da napiše niz programskih naredbi kojima se realizuje taj postupak. Naglasak u OOP nije na računarskim postupcima nego na objektima — softverskim elementima sa podacima i metodima koji mogu da dejstvuju jedni na druge. Objektno orijentisano programiranje se zato svodi na dizajniranje različitih klasa 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.

U prethodnom delu knjige skoro uopšte nismo imali dodira sa objektno orijentisanim programiranjem, jer se sve spomenute osnovne programske konstrukcije mogu naći, uz malo izmenjenu terminologiju, i u klasičnim, proceduralnim programskim jezicima. Naime, objektno orijentisane tehnike mogu se, u principu, koristiti u svakom programskom jeziku. To je posledica činjenice što se objekti iz standardnog ugla gledanja na stvari mogu smatrati običnom kolekcijom promenljivih i procedura koje manipulišu tim promenljivim. Međutim, za efektivnu realizaciju objektno orijentisanog načina rešavanja problema je vrlo važno imati jezik koji to omogućava na neposredan i elegantan način.

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

njenog definisanja u Javi, kao i na potpuno razumevanje objekata i njihovog konstruisanja u programu.

7.1 Klase i objekti

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

Članovi klase

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

1. Definicija polja se mora nalaziti izvan svih metoda, ali naravno i dalje unutar neke klase.
2. Ispred tipa i imena polja se mogu nalaziti modifikatori kao što su `static`, `public` i `private`.

Neki primeri definicija polja su:

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

Modifikatorom `static` se polje definiše da bude statičko; ako se ovaj modifikator ne navede, podrazumeva se da je polje objektno. Modifikatori pristupa `public` i `private` određuju gde se polje može koristiti. Polje sa modifikatorom `private` (privatno polje) može se koristiti samo unutar

klase u kojoj je definisano. Za polje sa modifikatorom `public` (javno polje) nema nikakvih ograničenja u vezi sa njegovim korišćenjem, odnosno ono se može koristiti u bilo kojoj klasi.

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

✧ ✧ ✧

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

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

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

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

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

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

Posmatrajmo sada sličnu klasu čija je definicija skoro identična:

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

U klasi `Klijent` su definisana nestatička (objektna) polja `ime` i `id`, pa u ovom slučaju ne postoje promenljive `Klijent.ime` i `Klijent.id`. Ova klasa dakle ne sadrži ništa konkretno, osim šablona za konstruisanje objekata te klase. Ali to je veliki potencijal, jer se taj šablon može iskoristiti za stvaranje velikog broja objekata klase `Klijent`. Svaki konstruisani objekat ove klase imaće *svoje* primerke polja `ime` i `id`. Zato program koji koristi ovu klasu može modelirati više klijenata, jer se po potrebi mogu konstruisati novi objekti za predstavljanje novih klijenata. Ako je to, recimo, neki bankarski program, kada klijent otvori račun u banci može se konstruisati nov objekat klase `Klijent` za čuvanje podataka o tom klijentu. Kada neki klijent banke zatvori račun, objekat koji ga predstavlja u programu može se ukloniti. Stoga u tom programu kolekcija objekata klase `Klijent` na prirodan način modelira rad banke.

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

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

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

Ovom klasom se u programu mogu predstaviti članovi neke porodice. Pošto je prezime nepromenljivo za sve članove porodice, nema potrebe taj

podatak vezivati za svakog člana porodice i zato se može čuvati u jedinstvenom statičkom polju `ČlanPorodice.prezime`. S druge strane, ime i uzrast su karakteristični za svakog člana porodice, pa se ti podaci moraju čuvati u nestatičkim poljima `ime` i `uzrast`.

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

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

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

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

Definicija klase

Opšti oblik definicije obične klase u Javi je vrlo jednostavan:

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

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

Na primer, u programu se za predstavljanje studenata i njihovih rezultata na testovima za jedan predmet može definisati klasa na sledeći način:

```
public class Student {

   public String ime;                // ime studenta
   public double test1,test2,test3;  // ocene na tri testa

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

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

7.2 Promenljive klasnog tipa

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

u definiciji nekog metoda. Na primer, ako se ima u vidu prethodna definicija klase `Student`, u programu se može definisati promenljiva klasnog tipa `Student`:

```
Student s;
```

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

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

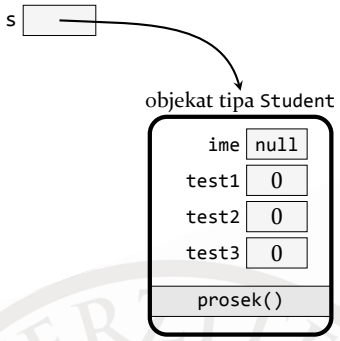
Promenljiva klasnog tipa dakle ne sadrži neki objekat kao svoju vrednost, već referencu na taj objekat. Zato kada se u programu koristi promenljiva klasnog tipa, ona služi za indirektan pristup objektu na koji ukazuje aktuelna vrednost (referenca) te promenljive.

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

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

najpre konstruiše novi objekat koji je instanca klase `Student`, a zatim se referenca na njega dodeljuje promenljivoj `s`. Ovaj efekat je ilustrovan na slici 7.1 u kojoj je prikazano stanje memorije nakon izvršavanja prethodne naredbe dodele. Na toj slici, referenca na novi objekat klase `Student` koja

je sadržaj promenljive s, prikazana je u obliku strelice koja pokazuje na taj objekat. Primetimo da je prikazan i automatski sadržaj dodeljen svim poljima novog objekta.



Slika 7.1: Efekat izvršavanja naredbe `Student s = new Student()`.

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

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

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

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

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

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

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

```
s = null;
```

ili naredba grananja u obliku:

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

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

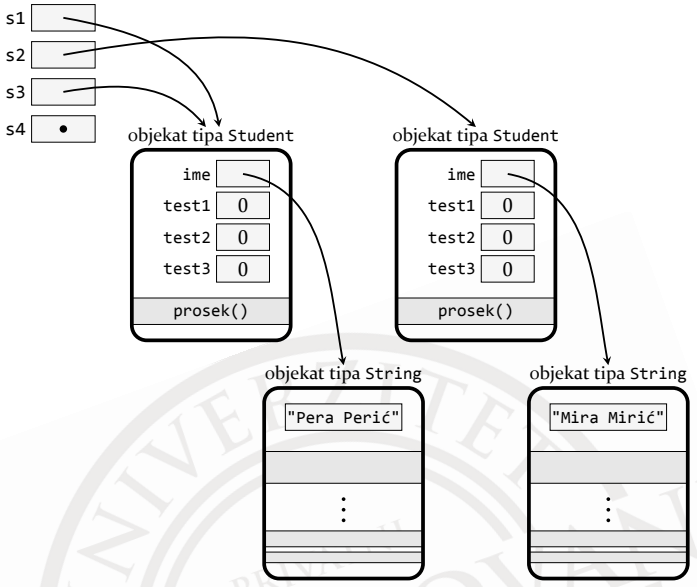
Da bismo bolje razumeli vezu između promenljivih klasnog tipa i objekata, razmotrimo detaljnije sledeći programski fragment:

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

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

Stanje memorije posle izvršavanja svih naredbi u ovom primeru prikazano je na slici 7.2. Na toj slici su reference na objekte prikazane na uobičajeni način kao strelice, dok je specijalna referenca `null` (koju sadrži

promenljiva s4) prikazana kao podebljana tačka.



Slika 7.2: Veza između promenljivih klasnih tipova i objekata.

Sa slike 7.2 se vidi da promenljive `s1` i `s3` ukazuju na isti objekat. Ta činjenica je posledica izvršavanja pretposlednje naredbe dodele `s3 = s1` kojom se kopira referenca iz `s1` u `s3`. Obratite pažnju na to da ovo važi i u opštem slučaju: kada se jedna promenljiva klasnog tipa dodeljuje drugoj, onda se kopira samo referenca, ali ne i objekat na koji ta referenca ukazuje. To znači da je vrednost polja `s1.ime` posle izvršavanja naredbi u prethodnom primeru jednaka stringu `"Pera Perić"`, ali i da je vrednost polja `s3.ime` jednaka `"Pera Perić"`.

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

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

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

gim rečima, za objekte na koje ukazuju promenljive `s1` i `s3` treba ispitati da li je tačan sledeći uslov:

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

Budući da su stringovi zapravo objekti klase `String`, a ne primitivne vrednosti, na slici 7.2 su stringovi `"Pera Perić"` i `"Mira Mirić"` prikazani kao objekti. Neka promenljiva klasnog tipa `String` može dakle sadržati samo referencu na objekat stringa (kao i referencu `null`), a ne i sâm niz znakova koji čine string. To objašnjava zašto su na slici 7.2 vrednosti dva primerka polja `ime` tipa `String` prikazane strelicama koje pokazuju na objekte odgovarajućih stringova. Primetimo da se u vezi sa stringovima često primenjuje neprecizna objektna terminologija, iako su stringovi pravi objekti kao i svaki drugi u Javi. Tako, na primer, kažemo da je vrednost polja `s1.ime` baš string `"Pera Perić"`, a ne pravilno ali rogobatnije da je vrednost polja `s1.ime` referenca na objekat stringa `"Pera Perić"`.

Spomenimo na kraju još dve logične posledice činjenice da promenljive klasnog tipa sadrže reference na objekte, a ne same objekte. (Istaknimo još jednom to važno pravilo: objekti se fizički nalaze negde u hip-memoriji, a promenljive klasnog tipa samo ukazuju na njih.) Prvo, pretpostavimo da je promenljiva klasnog tipa deklarisana s modifikatorom `final`. To znači da se njena vrednost nakon inicijalizacije više ne može promeniti. Ta početna, nepromenljiva vrednost ovakve promenljive klasnog tipa je referenca na neki objekat, što znači da će ona ukazivati na njega za sve vreme svog postojanja. Međutim, to nas ne sprečava da promenimo vrednosti polja koja se nalaze u objektu na koji ukazuje `final` promenljiva klasnog tipa. Drugim rečima, vrednost `final` promenljive je konstantna referenca, a nije konstantan objekat na koji ta promenljiva ukazuje. Ispravno je zato pisati, na primer:

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

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

x će i dalje ukazivati na isti objekat, ali polja u tom objektu mogu biti promenjena.

Konkretnije, pretpostavimo da je definisan metod

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

i da se izvršava programski fragment:

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

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

```
s = x;
s.ime = "Mika Mikić";
```

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

7.3 Konstrukcija i inicijalizacija objekata

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

Postupak konstruisanja jednog objekta se sastoji od dva koraka: rezervisanja dovoljno mesta u hip-memoriji za smeštanje objekta i inicijalizacije njegovih polja podrazumevanim vrednostima. Programer ne može da utiče na prvi korak pronalaženja mesta u memoriji za smeštanje objekta, ali za drugi korak u složenijem programu obično nije dovoljna samo podrazumevana inicijalizacija. Podsetimo se da se ova automatska inicijalizacija sastoji od dodele vrednosti nula poljima numeričkog tipa (`int`, `double`, ...), dodele vrednosti `false` logičkim poljima, dodele Unicode znaka `'\u0000'`

znakovnim poljima i dodele reference `null` poljima klasnog tipa. Ukoliko podrazumevana inicijalizacija nije dovoljna, poljima se mogu dodeliti početne vrednosti u njihovim deklaracijama, baš kao što se to može uraditi za bilo koje druge promenljive.

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

```
public class KockaZaIgru {

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

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

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

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

```
public class KockaZaIgru {

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

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

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

Inicijalizacija statičkih polja neke klase nije mnogo drugačija od inicijalizacije nestatičkih (objektnih) polja, osim što treba imati u vidu da statička polja nisu vezana za pojedinačne objekte nego za klasu kao celinu. Kako

postoji samo jedan primerak statičkog polja klase, ono se inicijalizuje samo jednom i to kada se klasa prvi put učitava od strane JVM prilikom izvršavanja programa.

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

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

Isti efekat se može postići kraćim zapisom:

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

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

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

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

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

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

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

```
public class KockaZaIgru {

   public int broj;              // broj na gornjoj strani kocke

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

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

U ovom primeru klase KockaZaIgru, konstruktor

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

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

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

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

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

```
public class KockaZaIgru {

   public int broj;              // broj na gornjoj strani kocke
```

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

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

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

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

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

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

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

1. Pronalazi se dovoljno veliki blok slobodne hip-memorije za objekat navedene klase koji se konstruiše.

2. Inicijalizuju se objektna polja tog objekta. Početna vrednost nekog polja objekta je ili ona izračunata iz deklaracije tog polja u klasi ili podrazumevana vrednost predviđena za njegov tip.

3. Poziva se konstruktor klase na uobičajen način: najpre se eventualni argumenti u pozivu konstruktora izračunavaju i dodeljuju odgovarajućim parametrima konstruktora, a zatim se izvršavaju naredbe u telu konstruktora.

4. Referenca na konstruisani objekat se vraća kao rezultat operatora `new`.

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

Primer: bacanje dve kocke dok se ne pokaže isti broj

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

Da bismo ilustrovali ovu mogućnost, u listingu 7.1 je prikazan program koji koristi klasu `KockaZaIgru` za određivanje broja puta koliko treba baciti dve kocke pre nego što se pokaže isti broj na njima.

Listing 7.1: Bacanja dve kocke

```
public class BacanjaDveKocke {

    public static void main(String[] args) {

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

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

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

          brojBacanja++;   // uračunati bacanje
```

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

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

class KockaZaIgru {

   public int broj;             // broj na gornjoj strani kocke

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

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

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

7.4 Uklanjanje objekata

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

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

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

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

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

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

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

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

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

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

oslobođeni odgovornosti od racionalnog upravljanja memorijom uklanjanjem nepotrebnih objekata u programu.

Iako procedura sakupljanja otpadaka usporava izvršavanje samog programa, razlog zašto se u Javi uklanjanje objekata obavlja automatski, a ne kao u nekim jezicima ručno od strane programera, jeste to što je vođenje računa o tome u složenijim programima vrlo teško i podložno greškama. Prva vrsta čestih grešaka se javlja kada se nenamerno obriše neki objekat, mada i dalje postoje reference na njega. Ove greške *visećih pokazivača* dovode do problema pristupa nedozvoljenim delovima memorije. Druga vrsta grešaka se javlja kada programer zanemari da ukloni nepotrebne objekte. Tada dolazi do greške *curenja memorije* koja se manifestuje time što program zauzima veliki deo memorije, iako je ona praktično neiskorišćena. To onda dovodi do problema nemogućnosti izvršavanja istog ili drugih programa zbog nedostatka memorije.

7.5 Skrivanje podataka (enkapsulacija)

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

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

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

```
public class Radnik {

   // Privatna polja
   private String ime;
   private long jmbg;
   private int staž;
   private double plata;

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

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

   public long getJmbg() {
      return jmbg;
   }

   public int getStaž() {
      return staž;
   }

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

   public double getPlata() {
      return plata;
   }

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

Obratite pažnju na to da su sva objektna polja klase `Radnik` definisana da budu privatna. Time je obezbeđeno da se ona izvan same klase `Radnik` ne mogu direktno koristiti. Tako, u nekoj drugoj klasi se više ne može pisati,

na primer:

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

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

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

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

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

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

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

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

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

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

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

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

```
private String prezime;
```

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

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

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

7.6 Službena reč `this`

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

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

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

```
pera.povećajPlatu(10);
```

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

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

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

```
laza.povećajPlatu(10);
```

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

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

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

Poznato je da se parametri koji odgovaraju eksplicitnim argumentima poziva metoda navode u definiciji metoda. S druge strane, parametar koji

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

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

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

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

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

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

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

Ipak, u nekim slučajevima je promenljiva `this` neophodna i bez nje se ne može dobiti željena funkcionalnost. Da bismo to pokazali, pretpostavimo da je klasi `Radnik` potrebno dodati objektni metod kojim se upoređuju dva radnika na osnovu njihovih plata. Tačnije, treba napisati objektni me-

tod `većeOd()` tako da se pozivom tog metoda u obliku, na primer,

```
pera.većeOd(laza)
```

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

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

Obratite ovde pažnju na to da se u zapisu `this.getPlata()` može izostaviti promenljiva `this`, jer se bez nje poziv metoda `getPlata()` ionako odnosi na implicitni parametar metoda `većeOd()` označen promenljivom `this`. S druge strane, promenljiva `this` jeste obavezna u naredbi `return this` u prvoj grani `if` naredbe, jer je rezultat metoda `većeOd()` u toj grani baš implicitni parametar ovog metoda.

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

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

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

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

Ali umesto toga, novi konstruktor se može kraće pisati:

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

U telu ovog konstruktora je zapisom

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

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

Prednost primene službene reči `this` u ovom kontekstu je dakle to što se zajedničke naredbe za konstruisanje objekata mogu pisati samo na jednom mestu u najopštijem konstruktoru. Onda se pozivom tog konstruktora pomoću `this` sa odgovarajućim argumentima mogu obezbediti drugi konstruktori za inicijalizovanje specifičnijih objekata. Pri tome treba imati u vidu i jedno ograničenje: poziv nekog konstruktora pomoću `this` mora biti prva naredba u drugom konstruktoru. Zato nije ispravno pisati

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

ali jeste ispravno:

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

Pitanja i zadaci

1. Koja od sledećih programskih jedinica predstavlja šablon za konstruisanje objekata istog tipa?

A. Paket B. Metod C. Promenljiva D. Klasa

2. Kakvi mogu biti članovi klase (polja i metodi)?

A. Statički (klasni) i nestatički (objektni).
B. Lokalni i globalni.
C. Proceduralni i neproceduralni.
D. Spoljašnji i unutrašnji.

3. Koja se službena reč koristi za definisanje klase?

A. `method` B. `class` C. `main` D. `object`

4. Kako se naziva specijalni metod neke klase koji se poziva po konstruisanju svakog objekta te klase?

A. Glavni metod B. Inicijalni metod C. Konstruktor klase
D. Rekurzivni metod

5. Koje su od ovih rečenica o konstruktorima tačne?

A. Podrazumevani konstruktor bez parametara se klasi automatski dodaje ukoliko u njoj nije eksplicitno definisan nijedan konstruktor.

B. U klasi se mora eksplicitno definisati bar jedan konstruktor.

C. Konstruktori nemaju tip rezultata, čak ni `void`.

D. Konstruktori moraju imati isto ime kao klasa u kojoj se definišu.

E. Konstruktori se pozivaju koristeći operator `new` kada se konstruiše objekat.

6. Analizirajte sledeći program koji se sastoji od dve klase u jednoj datoteci:

```
1   public class Test {
2      public static void main(String[] args) {
3         A a = new A();
4         a.prikaži();
5      }
6   }
7
8   class A {
9      String s;
10
11     public A(String s) {
12        this.s = s;
13     }
14     public void prikaži() {
15        System.out.println(s);
16     }
17  }
```

A. Program ima grešku, jer klasa `A` nije javna klasa.

B. Program ima grešku, jer klasa `A` nema podrazumevani konstruktor.

C. Program nema grešaka i normalno se izvršava ništa ne prikazujući na ekranu.

D. Program ima grešku koja se može ispraviti ukoliko se u trećem redu naredba `A a = new A();` zameni naredbom `A a = new A("poruka");`.

7. Analizirajte sledeći program koji se sastoji od dve klase u jednoj datoteci:

```
public class Test {
   public static void main(String[] args) {
      A a = new A(2);
   }
}

class A {
   int i;

   public void m(int j) {
      i = j;
   }
}
```

A. Program ima grešku, jer klasa `A` nije javna klasa.
B. Program ima grešku, jer klasa `A` nema podrazumevani konstruktor.
C. Program ima grešku, jer klasa `A` nema konstruktor sa parametrom tipa `int`.
D. Program nema grešaka i normalno se izvršava ništa ne prikazujući na ekranu.

8. Pod pretpostavkom da je data definicija `Krug k = new Krug()`, koja je od ovih rečenica najtačnija?

A. Promenljiva `k` sadrži celobrojnu vrednost.
B. Promenljivoj `k` se može dodeliti celobrojna vrednost.
C. Promenljiva `k` sadrži objekat klase `Krug`.
D. Promenljiva `k` sadrži referencu na objekat klase `Krug`.

9. Analizirajte sledeći program:

```
public class Test {

   int x;

   public Test(String s) {
```

```
        System.out.println("Test");
    }

    public static void main(String[] args) {
        Test t = null;
        System.out.println(t.x);
    }
}
```

A. Program ima grešku, jer promenljiva `t` nije inicijalizovana.
B. Program ima grešku, jer promenljiva `x` nije inicijalizovana.
C. Program ima grešku, jer klasa `Test` nema podrazumevani konstruktor.
D. Program ima grešku, jer se u nekoj klasi ne može deklarisati promenljiva tipa te iste klase kao što je to ovde slučaj sa promenljivom `t`.
E. Program ima grešku, jer promenljiva `t` ima vrednost `null` kada se prikazuje polje `t.x`.
F. Program nema grešaka i normalno se izvršava ništa ne prikazujući na ekranu.

10. Koje su automatske početne vrednosti za polja logičkog, numeričkog i klasnog tipa svakog objekta, tim redom ?

A. `true`, 1, `null`
B. `false`, 0, `null`
C. `true`, 0, `null`
D. `false`, 1, `null`
E. `false`, 0, `void`

11. Koje su od ovih rečenica o promenljivim tačne?

A. Lokalne promenljive metoda ne dobijaju automatski početne vrednosti.
B. Globalne promenljive (polja) objekata dobijaju automatski početne vrednosti.
C. Promenljiva nekog primitivnog tipa sadrži vrednost tog primitivnog tipa.
D. Promenljiva nekog klasnog tipa ukazuje na memorijsku adresu u kojoj se nalazi objekat tog klasnog tipa.
E. Promenljivoj klasnog tipa može se dodeliti ceo broj koji predstavlja važeću memorijsku adresu.

12. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      double prečnik;
      final double PI= 3.15169;
      double površina = prečnik * prečnik * PI;
      System.out.println("Površina je " + površina);
   }
}
```

A. Program ima grešku, jer promenljiva `prečnik` nije inicijalizovana.

B. Program ima grešku, jer je konstanta `PI` definisana unutar metoda.

C. Program ima grešku, jer konstanta `PI` ima previše decimala.

D. Program ima grešku, jer konstanta `PI` ima premalo decimala.

E. Program nema grešaka i normalno se izvršava.

13. Analizirajte sledeći program:

```
public class Test {

   int x;

   public Test(String s) {
      System.out.println("Test");
   }

   public static void main(String[] args) {
      Test t = new Test();
      System.out.println(t.x);
   }
}
```

A. Program ima grešku, jer se metod `System.out.println()` ne može koristiti u konstruktoru klase.

B. Program ima grešku, jer promenljiva `x` nije inicijalizovana.

C. Program ima grešku, jer klasa `Test` nema podrazumevani konstruktor.

D. Program ima grešku, jer se u nekoj klasi ne može konstruisati objekat te iste klase.

E. Program nema grešaka i normalno se izvršava prikazujući `0` na ekranu.

14. Koja je od ovih rečenica o objektima najtačnija?

A. Objektna promenljiva sadrži neki objekat.
B. Promenljiva klasnog tipa sadrži neki objekat.
C. Neki objekat može sadržati druge objekte.
D. Neki objekat može sadržati reference na druge objekte.

15. Koja polja su zajednička i jedinstvena za sve objekte neke klase?

A. Javna B. Privatna C. Objektna (instancna) D. Statička (klasna)

16. Da li se statičko polje neke klase može koristiti bez konstruisanja ijednog objekta te klase?

A. Da B. Ne C. Zavisi

17. U kojem od redova, 5 ili 9, u definiciji sledeće klase treba zameniti znakove `???` službenom rečju `static`?

```
1    public class Test {
2
3        private int broj;
4
5        public ??? int kvadrat(int n) {
6            return n * n;
7        }
8
9        public ??? int getBroj() {
10           return broj;
11       }
12   }
```

A. Samo u 5. redu. B. Samo u 9. redu. C. U oba reda 5 i 9.
D. U nijednom redu.

18. Kako se naziva metod koji se pridružuje svakom pojedinačnom objektu neke klase?

A. Statički metod B. Klasni metod C. Objektni metod
D. Glavni metod

19. Koji od sledećih načina je ispravan za definisanje konstante `MAX_CENA` kao članice neke klase?

A. `final static MAX_CENA = 99.98;`
B. `final static float MAX_CENA = 99.98;`
C. `static double MAX_CENA = 99.98;`
D. `final double MAX_CENA = 99.98;`
E. `final static double MAX_CENA = 99.98;`

20. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      int n = 2;
      xMetod(n);
      System.out.println("n je " + n);
   }

   void xMetod(int n) {
      n++;
   }
}
```

A. Program ima grešku, jer metod `xMetod()` ne vraća nijednu vrednost.
B. Program ima grešku, jer metod `xMetod()` nije definisan da bude statički.
C. Program prikazuje `n je 1` na ekranu.
D. Program prikazuje `n je 2` na ekranu.
E. Program prikazuje `n je 3` na ekranu.

21. Šta se prikazuje drugom naredbom `println` u metodu `main` prilikom izvršavanja ovog programa?

```
public class Test {

   int i;
   static int s;

   public static void main(String[] args) {
      Test t1 = new Test();
      System.out.println("t1.i je "+t1.i+", t1.s je "+t1.s);
      Test t2 = new Test();
      System.out.println("t2.i je "+t2.i+", t2.s je "+t2.s);
      Test t3 = new Test();
```

```
      System.out.println("t3.i je "+t3.i+", t3.s je "+t3.s);
   }

   public Test() {
      i++;
      s++;
   }
}
```

A. t2.i je 1, t2.s je 1
B. t2.i je 1, t2.s je 2
C. t2.i je 2, t2.s je 2
D. t2.i je 2, t2.s je 1

22. Šta se prikazuje trećom naredbom `println` u metodu `main` prilikom izvršavanja ovog programa?

```
public class Test {

   int i;
   static int s;

   public static void main(String[] args) {
      Test t1 = new Test();
      System.out.println("t1.i je "+t1.i+", t1.s je "+t1.s);
      Test t2 = new Test();
      System.out.println("t2.i je "+t2.i+", t2.s je "+t2.s);
      Test t3 = new Test();
      System.out.println("t3.i je "+t3.i+", t3.s je "+t3.s);
   }

   public Test() {
      i++;
      s++;
   }
}
```

A. t3.i je 1, t3.s je 1
B. t3.i je 1, t3.s je 2
C. t3.i je 1, t3.s je 3
D. t3.i je 3, t3.s je 1
E. t3.i je 3, t3.s je 3

23. Analizirajte sledeći program koji se sastoji od dve klase u jednoj datoteci:

```
public class Test {

   public static void main(String[] args) {
      A a = new A();
      a.n++;
   }
}

class A {
   int n;
   private A() {
   }
}
```

A. Program ima grešku, jer klasa `A` ima privatni podrazumevani konstruktor.
B. Program ima grešku, jer klasa `A` ima prazan podrazumevani konstruktor.
C. Program ima grešku, jer promenljiva `n` nije inicijalizovana.
D. Program nema grešaka i normalno se izvršava.

24. Koja se vrednost polja `b.n` prikazuje prvom naredbom `println` prilikom izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int k = 0;
      Brojač b = new Brojač();

      for (int i = 0; i < 100; i++)
         uvećaj(b, k);
      System.out.println("b.n = " + b.n);
      System.out.println("k = " + k);
   }

   public static void uvećaj(Brojač b, int k) {
      b.n++;
      k++;
   }
}
```

```
class Brojač {
   int n;

   public Brojač(int n) {
      this.n = n;
   }

   public Brojač() {
      this.n = 1;
   }
}
```

A. `b.n = 101` B. `b.n = 100` C. `b.n = 99` D. `b.n = 98`
E. `b.n = 0`

25. Koja se vrednost promenljive `k` prikazuje drugom naredbom `println` prilikom izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int k = 0;
      Brojač b = new Brojač();

      for (int i = 0; i < 100; i++)
         uvećaj(b, k);
      System.out.println("b.n = " + b.n);
      System.out.println("k = " + k);
   }

   public static void uvećaj(Brojač b, int k) {
      b.n++;
      k++;
   }
}

class Brojač {
   int n;

   public Brojač(int n) {
      this.n = n;
   }

   public Brojač() {
      this.n = 1;
```

```
        }
    }
```

A. `k = 101` B. `k = 100` C. `k = 99` D. `k = 98`
E. `k = 0`

26. Analizirajte sledeću klasu `Krug`:

```
public class Krug {

    private double prečnik;

    public Krug(double prečnik) {
        prečnik = prečnik;
    }
}
```

A. Klasa `Krug` ima grešku, jer nema metod `main()`.
B. Svaki konstruisani objekat klase `Krug` će imati prečnik 0. Na primer, naredbom `Krug k = new Krug(2.35)` dobija se krug `k` prečnika 0 iako se očekuje da njegov prečnik bude 2.35.
C. Klasa `Krug` ima grešku, jer se ne može pisati naredba dodele `prečnik = prečnik;` u konstruktoru.
D. Klasa `Krug` ima grešku, jer nema podrazumevani konstruktor.

27. Analizirajte sledeći program:

```
public class Test {

    private double x;

    public Test(double x) {
        this.t();
        this.x = x;
    }

    public Test() {
        System.out.println("Podrazumevani konstruktor");
        this(23);
    }

    public void t() {
        System.out.println("Poziv metoda t()");
    }
}
```

A. `this.t()` u konstruktoru `Test(double x)` može se pojednostaviti i zameniti samo sa `t()`.

B. `this.x` u konstruktoru `Test(double x)` može se pojednostaviti i zameniti samo sa `x`.

C. `this(23)` u konstruktoru `Test()` mora se pisati pre naredbe `System.out.println("Podrazumevani konstruktor");`.

D. `this(23)` u konstruktoru `Test()` mora se zameniti tačnijim izrazom `this(23.0)`.

✧ ✧ ✧

1. Napisati program koji prikazuje mesečni kalendar za dati mesec i datu godinu. (*Savet:* korisne klase za rad sa kalendarom su `Calendar` i `GregorianCalendar` iz paketa `java.util`.)

2. Napisati klase `Tačka`, `Kvadrat` i `Krug` koje predstavljaju tačku, kvadrat i krug u koordinatnom sistemu ravni. Klasa `Krug` treba da sadrži metod koji daje opisan kvadrat datog kruga. Napisati jednostavan program koji testira napisane klase.

3. Napisati klasu `KompleksniBroj` koja predstavlja kompleksni broj sa realnim i imaginarnim delom. Klasa `KompleksniBroj` treba da sadrži metode kojima se realizuju uobičajene operacije sa kompleksnim brojevima. Napisati jednostavan glavni metod `main()` u kojem se testiraju napisani metodi.

4. Napisati klasu `RimskiBroj` koja predstavlja rimski broj. (Pretpostavite da se rimski brojevi *ne* zapisuju u kondenzovanom obliku: na primer, broj 49 se piše XXXXVIIII umesto XLIX.) Klasa `RimskiBroj` treba da sadrži metode kojima se realizuju operacije sabiranja i množenja rimskih brojeva. Napisati jednostavan glavni metod `main()` u kojem se testiraju napisani metodi.

5. Ponovo uraditi 5. zadatak iz poglavlja 6, ali tako da program sadrži posebnu klasu `KockaZaIgru` koja predstavlja kocku za igranje.

6. Napisati program koji simulira bacanje novčića dati broj puta i prikazuje koliko puta su pali „pismo” i „glava”. Program treba da koristi posebnu klasu `Brojač` koja predstavlja opšti brojač za brojanje 0, 1, 2, 3, ...

8

Nizovi

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

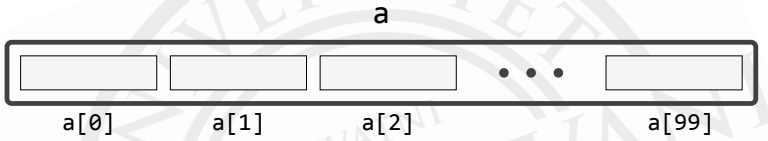
Istovremeno raspolaganje velikim (čak neograničenim) brojem podataka zahteva poseban način rada sa njima koji omogućava relativno lako dodavanje, uklanjanje, pretraživanje i slične operacije sa pojedinačnim podacima. Ako se ima u vidu kolekcija podataka organizovanih u ovom smislu, onda se govori o *strukturi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

8.1 Jednodimenzionalni nizovi

Niz je struktura podataka koja predstavlja numerisan niz promenljivih istog tipa. Pojedinačne promenljive u nizu se nazivaju *elementi* niza, a njihov redni broj u nizu se naziva *indeks*. Ukupan broj elemenata niza se naziva *dužina* niza. Dužina niza je jedina, „horizontalna" dimenzija niza elemenata (promenljivih), pa se za ovu vrstu nizova kaže da su jednodimenzionalni ukoliko ih je potrebno razlikovati od višedimenzionalnih koji pored dužine mogu imati i visinu, dubinu i tako dalje.

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

Niz elemenata se kao celina u programu predstavlja jednom promenljivom specijalnog, nizovnog tipa. Radi jednostavnijeg izražavanja, ova sama promenljiva se često naziva niz, mada je preciznije reći „promenljiva koja ukazuje na niz elemenata". Za označavanje bilo kog elementa niza se koristi zapis koji se sastoji od imena te promenljive i indeksa odgovarajućeg elementa u uglastim (srednjim) zagradama. Na primer, niz `a` od 100 elemenata se sastoji od 100 promenljivih istog tipa čiji je konceptualni izgled prikazan na slici 8.1.



Slika 8.1: 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 8.1 definisan da bude `int`, onda je svaka od 100 promenljivih `a[0]`, `a[1]`, `a[2]`, ..., `a[99]` obična celobrojna promenljiva koja se u programu može koristiti na svakom mestu gde su dozvoljene celobrojne promenljive.

Definisanje nizova

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

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

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

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

```
int[] a;
```

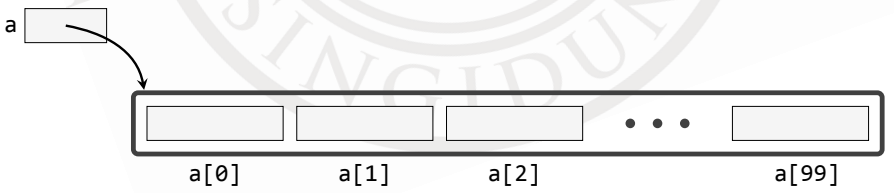
Kao i svaka promenljiva klasnog tipa, ova promenljiva `a` može sadržati referencu na neki objekat klase `int[]`. A kao što smo upravo objasnili, objekti klase `int[]` su nizovi baznog tipa `int`. Kao i svaki objekat, objekat niza tipa `int[]` se konstruiše operatorom `new`, doduše u posebnom obliku,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

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

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

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

Ovom naredbom se alocira promenljiva `a` klasnog tipa `int[]`, konstruiše se objekat niza od 100 elemenata koji su svi primitivnog tipa `int` i referenca na novokonstruisani objekat niza dodeljuje se promenljivoj `a`. Efekat prethodne naredbe je ilustrovan na slici 8.2.



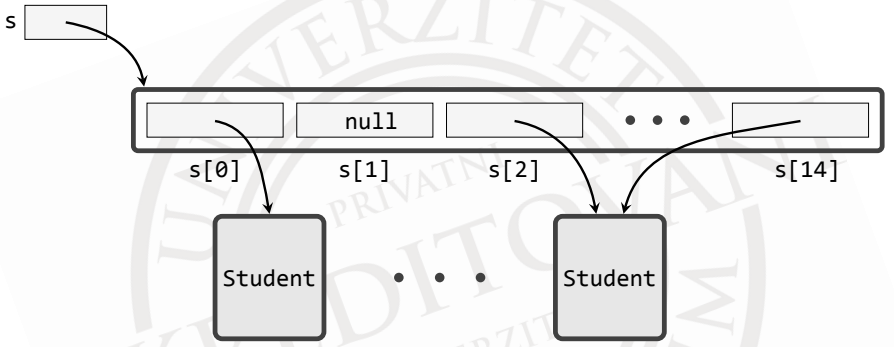
Slika 8.2: Objekat niza `a` od 100 elemenata celobrojnog tipa.

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

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

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

Ovom definicijom se alocira promenljiva `s` klasnog tipa `Student[]`, konstruiše se niz od 15 elemenata klasnog tipa `Student` i referenca na novi niz se dodeljuje promenljivoj `s`. Svaki od 15 elemenata ovog niza predstavlja promenljivu klasnog tipa `Student` čiji sadržaj može biti referenca na objekte klase `Student` ili `null`. Jedan primer sadržaja niza `s` prikazan je na slici 8.3.



Slika 8.3: Objekat niza `s` sa elementima klasnog tipa `Student`.

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

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

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

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

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

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

20 elemenata i taj njegov broj elemenata se kasnije ne može niti smanjiti niti povećati.

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

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

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

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

Opšti oblik operatora `new` u ovom kontekstu je:

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

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

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

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

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

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

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

Korišćenje nizova

Elementi niza su obične promenljive baznog tipa niza i mogu se koristiti u programu na svim mestima gde je dozvoljena upotreba promenljive baznog tipa niza. Ovi elementi niza se koriste preko svojih indeksa i imena promenljive koja ukazuje na objekat niza. Na primer, jedan niz `a` čija je definicija

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

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

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

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

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

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

Obratite pažnju na jednostavnost gornje petlje kojom je realizovan zadatak prikazivanja svih elemenata niza `a` — pomoću praktično dva reda se prikazuju vrednosti 1000 promenljivih! Ne samo to, i da niz `a` ima 100000 ili više elemenata, potpuno istom petljom bi se prikazale vrednosti mnogo većeg broja tih elemenata.

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

Primer: prebrojavanje glasova na izborima

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

U listingu 8.1 je prikazan jednostavan program za prebrojavanje glasova na izborima. U tom programu se koristi celobrojni niz čiji elementi sadrže broj glasova pojedinih partija. Drugim rečima, ako su partije u programu

numerisane od 0, onda se za sabiranje njihovih glasova koristi niz čiji *i*-ti element sadrži broj glasova *i*-te partije. Prestanak unošenja glasova se programski realizuje unosom glasa za nepostojeću partiju. Nakon toga se u programu prikazuje ukupan broj glasova svih partija.

Listing 8.1: 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 za sabiranje glasova partija
      int[] partije = new int[brojPartija];

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

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

Primetimo da su indeksi niza `partije` u ovom programu morali da budu prilagođeni za 1, pošto su partije na glasačkim listićima numerisane od 1 a nizovi u Javi su numerisani od 0.

Primer: kopiranje nizova

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

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

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

```
double[] a;
```

onda se naredbom dodele

```
a = b;
```

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

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

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

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

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

dobija novi niz na koji ukazuje a. Elementi novog niza su fizičke kopije

elementa niza na koji ukazuje `b` tako da se sada zapisi `a[i]` i `b[i]` odnose na različite *i*-te elemente odgovarajućih nizova.

Primer: argumenti programa u komandnom redu

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

```
java glavna-klasa
```

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

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

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

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

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

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

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

Metod `main()` dobija dakle vrednosti za parametar `args` iz komandnog reda tako što Java interpretator automatski inicijalizuje niz `args` stringovima koji su navedeni iza glavne klase u komandnom redu, a u polje `args.length` se upisuje broj tih stringova. Ako nije ništa navedeno, dužina niza `args` dobija vrednost 0. Kao praktičan primer, u listingu 8.2 je prikazan program kojim se samo prikazuje niz stringova naveden kao argument tog programa u komandnom redu.

Listing 8.2: Argumenti programa u komandnom redu

```
public class Poruka {

   public static void main(String[] args) {

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

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

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

Ako se program u listingu 8.2 izvrši komandom

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

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

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

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

```
Zbogom okrutni svete!
```

Ako se pak program izvrši komandom

```
java Poruka -d tugo
```

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

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

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

```
Dobar dan tugo!
```

Klasa `Arrays`

Radi lakšeg rada sa nizovima, u Javi se može koristiti standardna klasa `Arrays` iz paketa `java.util`. Ova pomoćna klasa sadrži nekoliko statičkih metoda koji obezbeđuju korisne operacije nad nizovima čiji je bazni tip jedan od primitivnih tipova. U nastavku je naveden nepotpun spisak i opis ovih metoda, pri čemu oznaka *`tip`* ukazuje na jedan od primitivnih tipova. Izuzetak su jedino metodi `sort()` i `binarySearch()` kod kojih nije dozvoljeno da to bude primitivni tip `boolean`.

- `String toString(`*`tip`*`[] a)` — vraća se reprezentacija niza `a` u obliku stringa. U tom obliku, vrednosti svih elemenata niza `a` se nalaze unutar uglastih zagrada po redu njihovih pozicija u nizu i međusobno su razdvojene zapetama.
- *`tip`*`[] copyOf(`*`tip`*`[] a, int n)` — vraća se nova kopija niza `a` koja se sastoji od prvih `n` njegovih elemenata. Ako je je vrednost `n` veća od vrednosti `a.length`, onda se višak elemenata kopije niza inicijalizuje nulom ili vrednošću `false`. U suprotnom slučaju, kopira se samo početnih `n` elemenata niza `a`.
- *`tip`*`[] copyOfRange(`*`tip`*`[] a, int i, int j)` — vraća se nova kopija niza `a` od indeksa `i` do indeksa `j`. Element sa indeksom `i` niza `a` se uključuje, dok se onaj sa indeksom `j` ne uključuje u novi niz. Ako je indeks `j` veći od `a.length`, višak elemenata kopije niza se inicijalizuje nulom ili vrednošću `false`.
- `void sort(`*`tip`*`[] a)` — sortira se niz `a` u mestu u rastućem redosledu.
- `int binarySearch(`*`tip`*`[] a, `*`tip`*` v)` — pronalazi se data vrednost `v` u sortiranom nizu `a` korišćenjem binarne pretrage. Ako je vrednost

v nađena u nizu, vraća se indeks odgovarajućeg elementa. U suprotnom slučaju, vraća se negativna vrednost k tako da $-k-1$ odgovara poziciji gde bi data vrednost trebalo da se nalazi u sortiranom nizu.

- `void fill(`*`tip`*`[] a, `*`tip`*` v)` — dodeljuje se svim elementima niza `a` vrednost `v`.
- `boolean equals(`*`tip`*`[] a, `*`tip`*`[] b)` — vraća se vrednost `true` ukoliko nizovi `a` i `b` imaju istu dužinu i jednake odgovarajuće elemente. U suprotnom slučaju, vraća se vrednost `false`.

Primer: izvlačenje loto brojeva

Radi ilustracije primene klase `Arrays` u radu sa nizovima, u listingu 8.3 je prikazan program koji simulira izvlačenje brojeva u igri na sreću loto. U igri loto se izvlači 7 slučajnih, međusobno različitih brojeva među celim brojevima od 1 do 39. U programu se generišu 7 takvih brojeva koji predstavljaju jedno kolo igre loto. (Drugi ugao gledanja na rezultat programa je da generisane brojeve čitaoci mogu igrati u stvarnoj igri loto da bi osvojili neku od nagrada.)

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

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

Za generisanje slučajnog broja između 1 i N generiše se, u stvari, slučajni indeks levog dela niza `bubanj` i uzima broj u elementu niza `bubanj` koji se nalazi na toj poziciji. Zatim se taj element međusobno zamenjuje sa poslednjim elementom levog dela niza `bubanj` i pomera ulevo granica između dva dela niza smanjivanjem vrednosti `m` za jedan. Ponavljanjem

ovog postupka K puta za svaki izvučeni broj, na kraju se u desnom delu niza `bubanj`, od indeksa $N - K$ do kraja, nalaze svi izvučeni slučajni brojevi.

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

Listing 8.3: Izvlačenje loto brojeva

```
import java.util.*;

public class Loto {

   public static void main (String[] args) {

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

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

      int m;  // granica levog i desnog dela niza

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

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

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

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

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

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

Naredba `for-each`

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

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

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

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

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

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

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

Ili, prikazivanje izvučenih loto brojeva u programu na strani 236 može se uraditi na elegantniji način:

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

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

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

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

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

Metodi sa promenljivim brojem argumenata

Od verzije Java 5 je omogućeno pozivanje metoda sa promenljivim brojem argumenata. Metod `printf()` za formatizovano prikazivanje vrednosti na ekranu predstavlja primer metoda sa promenljivim brojem argumenata. Naime, prvi argument metoda `printf()` mora biti tipa `String`, ali ovaj metod može imati proizvoljan broj dodatnih argumenata bilo kog tipa.

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Dvodimenzionalni nizovi

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

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

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

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

jednodimenzionalnog niza — razlika je samo u dodatnom paru uglastih zagrada. Naredbom, na primer,

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

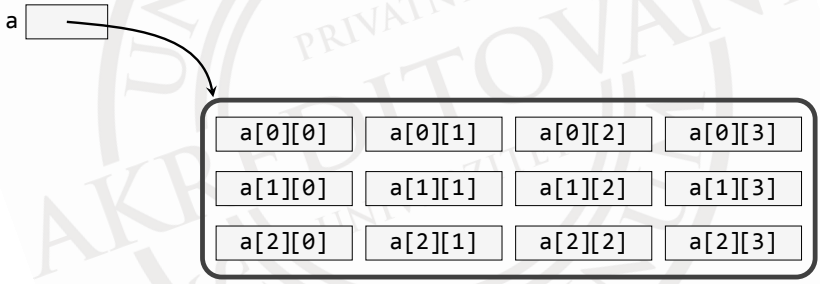
definiše se promenljiva `a` čiji sadržaj može biti referenca na objekat dvodimenzionalnog niza tipa `int[][]`. Konstruisanje aktuelnog dvodimenzionalnog niza se vrši operatorom `new` kao i u jednodimenzionalnom slučaju. Na primer, naredbom dodele

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

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

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

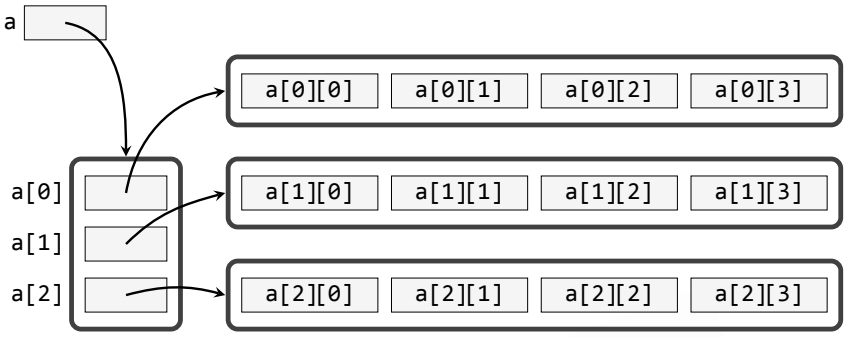
Na slici 8.4 je prikazan izgled relevantne memorije računara nakon izvršavanja ove naredbe dodele.



Slika 8.4: Dvodimenzionalni celobrojni niz `a` veličine 3×4.

Slika 8.4 pokazuje da se jedan dvodimenzionalni niz na koji ukazuje promenljiva `a` može najbolje zamisliti kao matrica (ili tabela) elemenata koja ima tri vrste i četiri kolone. Elementi matrice su obične, u ovom slučaju celobrojne, promenljive koje imaju dvostruke indekse `a[i][j]`: prvi indeks pokazuje vrstu i drugi indeks kolonu u kojima se element nalazi u matrici. Pri tome treba imati u vidu da, kao što je to uobičajeno u Javi, numeracija vrsta i kolona matrice ide od nule, a ne od jedan.

Tačnije govoreći, dvodimenzionalni niz `a` u prethodnom primeru nije matrica, nego je zaista niz celobrojnih nizova. Tako, izrazom `new int[3][4]` zapravo se konstruiše niz od tri celobrojna niza, od kojih svaki ima četiri elementa. Pravi izgled dvodimenzionalnog niza `a` je zato onaj koji je prikazan na slici 8.5.



Slika 8.5: Prava slika dvodimenzionalnog niza a dimenzije 3×4.

Prava slika dvodimenzionalnog niza je komplikovanija za razumevanje i, srećom, može se u većini slučajeva zanemariti i dvodimenzionalni niz smatrati matricom elemenata. Ponekad je ipak potrebno znati da svaka vrsta matrice predstavlja zapravo jedan niz za sebe. Ovi nizovi u prethodnom primeru su tipa `int[]` i na njih ukazuju promenljive čija su imena `a[0]`, `a[1]` i `a[2]`. Te promenljive i nizovi se u programu mogu koristiti na svim mestima gde su dozvoljeni obični celobrojni nizovi. Na primer, jedna vrsta matrice može biti argument u pozivu metoda čiji je parametar tipa `int[]`.

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

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

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

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

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

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

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

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

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

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

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

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

Primer: rad sa tabelarnim podacima

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

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

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

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

```
public double godišnjiProfit() {

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

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

```
private double mesečniProfit(int m) {

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

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

```
public void prikažiProfitPoMesecima() {

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

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

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

```
public void prikažiProfitPoProdavnicama() {

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

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

U listingu 8.4 je prikazan kompletan program kojim se korisniku nude razne opcije za rad sa podacima o profitu neke firme. Program se sastoji od dve klase. Prva, glavna klasa programa služi za izbor pojedinih opcija iz korisničkog menija radi prikazivanja različitih pokazatelja o profitu jedne firme na ekranu. Druga klasa `Firma` opisuje konkretnu firmu i sadrži polje `n`

za broj njenih prodavnica, kao i polje `profit` koje ukazuje na njenu matricu profita. Pored ovih atributa, klasa `Firma` sadrži i prethodne primere metoda kojima se manipuliše matricom profita konkretne firme.

Listing 8.4: Profit firme sa više prodavnica

```
import java.util.*;

public class ProfitFirme {

   public static void main(String[] args) {

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

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

      do {
         prikaziMeni();

         brojOpcije = tastatura.nextInt();

         switch (brojOpcije) {
            case 1:
               abc.unesiProfit();
               break;
            case 2:
               abc.prikaziProfit();
               break;
            case 3:
               if (abc.getProfit() == null)
                  System.out.println("Greška: podaci ne postoje!");
               else {
                  System.out.print("Ukupni godišnji profit firme:");
                  System.out.printf("%8.2f", abc.godišnjiProfit());
                  System.out.println();
               }
               break;
            case 4:
               abc.prikaziProfitPoMesecima();
               break;
```

```
            case 5:
               abc.prikažiProfitPoProdavnicama();
               break;
            case 0:
               System.out.println("Kraj programa ...");
               break;
            default:
               System.out.println("Greška: pogrešna opcija!");
         }
      } while (brojOpcije != 0);
   }

   private static void prikažiMeni() {

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

class Firma {

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

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

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

   public void unesiProfit() {

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

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

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

    public void prikaziProfit() {

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

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

    .
    . // Ostali metodi klase: godišnjiProfit(), mesečniProfit(),
    . // prikaziProfitPoMesecima(), prikaziProfitPoProdavnicama()
    .
}
```

Primer: telefonski imenik

U ovom primeru je ilustrovana struktura podataka koja predstavlja običan telefonski imenik u mobilnom telefonu ili sličnom uređaju. U najprostijem slučaju, telefonski imenik je niz kontakata koji se sastoje od dva dela: imena osobe i telefonskog broja te osobe. To se u programu na najjednostavniji način može predstaviti pomoću matrice koja ima dve kolone: prva kolona odgovara imenima osoba, a druga kolona njihovim telefonskim brojevima. Tip ove matrice je `String[][]`, jer su imena i telefonski brojevi osoba, u opštem slučaju, stringovi. To je jasno za imena osoba, dok za te-

lefonske brojeve treba uzeti u obzir da oni mogu sadržati znakove +, /, - i slično.

Imajući ova zapažanja u vidu, klasa `TelImenik` u listingu 8.5 predstavlja telefonski imenik kao strukturu podataka. Ta klasa sadrži polje `imenik` koja ukazuje na matricu podataka jednog imenika od najviše 100 kontakata, kao i polje `n` koje sadrži stvarni broj kontakata. Pored ovakve organizacije podataka u telefonskom imeniku, u klasi `TelImenik` se nalaze i metodi kojima se realizuju osnovne operacije nad podacima u telefonskom imeniku:

- Metod `nađiBroj()` za dato ime osobe pronalazi njegov telefonski broj u imeniku.
- Metod `dodajKontakt()` dodaje dati par ime/broj u imenik. U ovom metodu se najpre proverava da li se dato ime nalazi u imeniku. Ako je to slučaj, stari broj se zamenjuje datim brojem. U suprotnom slučaju, novi kontakt se dodaje u imenik.
- Metod `ukloniKontakt()` uklanja kontakt sa datim imenom osobe iz imenika. Ako se takav kontakt ne pronađe u imeniku, ništa se posebno ne preduzima.

Obratite pažnju i na to da je u klasi `TelImenik` definisan pomoćni metod `nađiKontakt()` u kojem se primenjuje linearna pretraga za nalaženje pozicije kontakta sa datim imenom u imeniku. Metod `nađiKontakt()` je privatni metod, jer je potreban samo za interno korišćenje od strane ostalih javnih metoda. Na kraju, radi testiranja, klasi `TelImenik` je dodat metod `main()` u kojem je simuliran tipični način rada sa telefonskim imenikom.

Listing 8.5: Telefonski imenik

```
public class TelImenik {

   private String[][] imenik;  // matrica kontakata ime/broj
   private int n = 0;          // broj kontakata (početno 0)
   private final int N = 100;  // maksimalni broj kontakata

   // Konstruktor klase za konstruisanje imenika
   // od najviše N kontakata
   public TelImenik() {
      imenik = new String[N][2];
   }

   private int nađiKontakt(String imeOsobe) {
```

```
    for (int i = 0; i < n; i++) {
        // Da li i-ta osoba ima dato ime?
        if (imenik[i][0].equals(imeOsobe))
            return i; // i-ta osoba ima dato ime
    }
    return -1;      // nema osobe sa datim imenom
}

public String nađiBroj(String imeOsobe) {

    int i = nađiKontakt(imeOsobe);
    if (i >= 0)  // osoba je u imeniku?
        // Ako jeste, vratiti njen tel. broj
        return imenik[i][1];
    else
        // Ako nije, vratiti referencu null
        return null;
}

public void dodajKontakt(String imeOsobe, String brojOsobe) {

    if (imeOsobe == null || brojOsobe == null) {
        System.out.println("Greška: prazno ime ili broj kontakta!");
        return;
    }
    int i = nađiKontakt(imeOsobe);
    if (i >= 0)       // osoba je u imeniku?
        // Ako jeste, zameniti stari broj novim brojem
        imenik[i][1] = brojOsobe;
    else if (n == N)  // imenik je pun?
        System.out.println("Greška: imenik je pun!");
    else {
        // Ako imenik nije pun, dodati novi par ime/broj u imenik
        imenik[n][0] = imeOsobe;
        imenik[n][1] = brojOsobe;
        n++;
    }
}

public void ukloniKontakt(String imeOsobe) {

    int i = nađiKontakt(imeOsobe);
    if (i >= 0) {  // osoba je u imeniku?
        // Ako jeste, ukloniti taj kontakt iz imenika
```

```
            imenik[i][0] = imenik[n-1][0];
            imenik[i][1] = imenik[n-1][1];
            n--;
         }
      }

      public static void main(String[] args) {

         TelImenik mojImenik = new TelImenik();
         mojImenik.dodajKontakt("Pera", null);
         mojImenik.dodajKontakt("Pera", "111-1111");
         mojImenik.dodajKontakt("Žika", "222-2222");
         mojImenik.dodajKontakt("Laza", "333-3333");
         mojImenik.dodajKontakt("Mira", "444-4444");
         System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
         mojImenik.dodajKontakt("Laza", "999-9999");
         System.out.println("Laza: " + mojImenik.nađiBroj("Laza"));
         System.out.println("Pera: " + mojImenik.nađiBroj("Pera"));
         mojImenik.ukloniKontakt("Žika");
         System.out.println("Žika: " + mojImenik.nađiBroj("Žika"));
         System.out.println("Mira: " + mojImenik.nađiBroj("Mira"));
      }
}
```

Pitanja i zadaci

1. Koje je ime trećeg elementa u nizu pod nazivom `a`?

A. `a[2]` B. `a(2)` C. `a[3]` D. `a(3)`

2. Koje su od ovih definicija niza `a` *pogrešne*?

A. `int[] a = new int[2];`
B. `int[] a = new int(2);`
C. `int a = new int[2];`
D. `int a() = new int[2];`

3. Ako je data deklaracija `int i = 5`, koji se od ovih izraza mogu koristiti za indekse elemenata niza `double[] d = new double[100]`?

A. `i`
B. `(int)(Math.random() * 100)`

C. `(int)(Math.random() * 100 + 1)`
D. `Math.random() * 100`
E. `i + 10`
F. `i + 6.5`

4. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      int[] a = new int[3];
      System.out.println("a[0] je " + a[0]);
   }
}
```

A. Program ima grešku, jer je dužina niza `a` premala.
B. Program ima grešku, jer elementi niza `a` nisu inicijalizovani.
C. Program ima grešku, jer element `a[0]` nije definisan.
D. Program nema grešaka i normalno se izvršava prikazujući `a[0] je 0` na ekranu.

5. Koje su od ovih deklaracija nizova u Javi ispravne?

A. `int i = new int(30);`
B. `double[] d = new double[30];`
C. `int[] i = {3, 4, 3, 2};`
D. `char[] c = new char();`
E. `char[] c = new char{'a', 'b', 'c', 'd'};`
F. `char[] c = {'a', 'b'};`

6. Ako je data deklaracija `int[] a = {1, 2, 3, 4}`, koju vrednost sadrži polje `a.length`?

A. 0 B. 3 C. 4 D. 5

7. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      int[] a = new int[5];
      int i;
      for (i = 0; i < a.length; i++)
```

```
            a[i] = i;
         System.out.print(a[i] + " ");
      }
   }
```

A. Program prikazuje `0 1 2 3 4` na ekranu.

B. Program prikazuje `4` na ekranu.

C. Program ima grešku, jer će se koristiti nepostojeći element `a[5]` u poslednjoj naredbi `print` u metodu `main`.

D. Program ima grešku, jer promenljiva `i` u poslednjoj naredbi `print` u metodu `main` neće imati nijednu vrednost.

8. Šta je rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] a = {120, 200, 016};
      for (int i = 0; i < a.length; i++)
         System.out.print(a[i] + " ");
   }
}
```

A. Program prikazuje `120 200 16` na ekranu.

B. Program prikazuje `120 200 14` na ekranu.

C. Program prikazuje `120 200 22` na ekranu.

D. Program ima grešku, jer umesto `016` treba pisati `16`.

9. Šta se prikazuje na ekranu za vrednosti niza `lista2` kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] lista1 = {1, 2, 3};
      int[] lista2 = {1, 2, 3};
      lista2 = lista1;
      lista1[0] = 0; lista1[1] = 1; lista2[2] = 2;

      for (int i = 0; i < lista2.length; i++)
         System.out.print(lista2[i] + " ");
   }
}
```

A. 1 2 3 B. 1 1 1 C. 0 1 2 D. 0 1 3

10. Šta se prikazuje na ekranu za vrednosti niza `lista1` kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] lista1 = {1, 2, 3};
      int[] lista2 = {1, 2, 3};
      lista2 = lista1;
      lista1[0] = 0; lista1[1] = 1; lista2[2] = 2;

      for (int i = 0; i < lista1.length; i++)
         System.out.print(lista1[i] + " ");
   }
}
```

A. 1 2 3 B. 1 1 1 C. 0 1 2 D. 0 1 3

11. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] x = {1, 2, 3, 4};
      int[] y = x;

      x = new int[2];

      for (int i = 0; i < y.length; i++)
         System.out.print(y[i] + " ");
   }
}
```

A. 1 2 3 4 B. 0 0 C. 0 0 3 4 D. 0 0 0 0

12. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] x = {1, 2, 3, 4};
      int[] y = x;

      x = new int[2];
```

```
        for (int i = 0; i < x.length; i++)
            System.out.print(x[i] + " ");
    }
}
```

A. `1 2 3 4` B. `0 0` C. `0 0 3 4` D. `0 0 0 0`

13. Analizirajte sledeći program:

```
public class Test {

    public static void main(String[] args) {
        final int[] x = {1, 2, 3, 4};
        int[] y = x;

        x = new int[2];

        for (int i = 0; i < y.length; i++)
            System.out.print(y[i] + " ");
    }
}
```

A. Program prikazuje `1 2 3 4` na ekranu.

B. Program prikazuje `0 0` na ekranu.

C. Program ima grešku kod naredbe `x = new int[2]`, jer je promenljiva `x` deklarisana da bude `final` i ne može se menjati.

D. Elementi niza `x` se ne mogu menjati, jer je promenljiva `x` deklarisana da bude `final`.

14. Analizirajte sledeći programski fragment:

```
int[] lista = new int[5];
lista = new int[10];
```

A. Programski fragment ima grešku, jer se promenljiva `lista` ne može menjati nakon što joj se dodeli vrednost.

B. Programski fragment nema grešaka i drugom naredbom se novi niz dodeljuje promenljivoj `lista`.

C. Programski fragment ima grešku, jer se promenljivoj `lista` dodeljuje novi niz.

D. Programski fragment ima grešku, jer se promenljivoj `lista` dodeljuje novi niz različite dužine od prvog.

15. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      int[] a = new int[4];
      a[1] = 1;

      a = new int[2];

      System.out.println("a[1] je " + a[1]);
   }
}
```

A. Program ima grešku kod naredbe `a = new int[2]`, jer se novi niz dodeljuje promenljivoj `a`.
B. Program ima grešku kod naredbe `println`, jer `a[1]` nije inicijalizovano.
C. Program na ekranu prikazuje `a[1] je 0`.
D. Program na ekranu prikazuje `a[1] je 1`.

16. Kojom naredbom se kopija niza `a` dodeljuje nizu `b`.

A. `b = Arrays.copyOf(a, a.length);`
B. `b = Arrays.copyOf(a);`
C. `Arrays.copyOf(b, a, a.length);`
D. `Arrays.copyOf(a, b);`

17. Kada se dati niz prenosi nekom metodu kao argument, šta se tačno prenosi tom metodu?

A. Kopija datog niza.
B. Kopija prvog elementa datog niza.
C. Dužina datog niza.
D. Referenca na dati niz.

18. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] x = {1, 2, 3, 4, 5};
      uvećaj(x);
```

```
        int[] y = {1, 2, 3, 4, 5};
        uvećaj(y[0]);

        System.out.println(x[0] + " " + y[0]);
    }

    public static void uvećaj(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i]++;
    }

    public static void uvećaj(int n) {
        n++;
    }
}
```

A. Poruka o grešci B. 1 1 C. 2 2 D. 2 1 E. 1 2

19. Šta se prikazuje na ekranu za vrednosti niza `lista` kao rezultat izvršavanja ovog programa?

```
public class Test {

    public static void main(String[] args) {
        int[] lista = {1, 2, 3, 4, 5};

        obrniNiz(lista);

        for (int i = 0; i < lista.length; i++)
            System.out.print(lista[i] + " ");
    }

    public void obrniNiz(int[] a) {
        int[] b = new int[a.length];

        for (int i = 0; i < a.length; i++)
            b[i] = a[a.length - 1 - i];

        a = b;
    }
}
```

A. 1 2 3 4 5 B. 5 4 3 2 1 C. 5 4 1 2 3 D. 1 2 5 4 3

20. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {
      xMetod(new double[]{3, 3});
      xMetod(new double[5]);
      xMetod(new double[3]{1, 2, 3});
   }

   public void xMetod(double[] a) {
      System.out.println(a.length);
   }
}
```

A. Program ima grešku, jer je u prvom pozivu metoda `xMetod()` nepravilno naveden argument `new double[]{3, 3}`.
B. Program ima grešku, jer je u drugom pozivu metoda `xMetod()` nepravilno naveden argument `new double[5]`.
C. Program ima grešku, jer je u trećem pozivu metoda `xMetod()` nepravilno naveden argument `new double[3]{1, 2, 3}`.
D. Program ima grešku, jer će sve vrednosti niza `a` imati vrednost `null` prilikom izvršavanja drugog poziva metoda `xMetod()`.

21. Kako se zove deo memorije u kojoj se smeštaju nizovi, kao i svi drugi objekti programa? U tom delu se, radi efikasnosti, ne vodi mnogo računa o redu po kojem se zauzima slobodna i oslobađa zauzeta memorija.

A. Stek memorija B. Hip memorija C. Keš memorija
D. Virtuelna memorija

22. Kada se niz vraća kao rezultat nekog metoda, šta se tačno prenosi kao rezultat?

A. Kopija tog niza.
B. Kopija prvog elementa tog niza.
C. Dužina tog niza.
D. Referenca na taj niz.

23. Ako je dato zaglavlje metoda `public static int[] xMetod()`, koja se od ovih naredbi `return` može koristiti u telu metoda `xMetod()`?

A. `return 1;`
B. `return {1, 2, 3};`
C. `return int[]{1, 2, 3};`

D. `return new int[]{1, 2, 3};`

24. Šta se prikazuje na ekranu za vrednosti niza `lista` kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] lista = {1, 2, 3, 4, 5};

      lista = obrniNiz(lista);

      for (int i = 0; i < lista.length; i++)
         System.out.print(lista[i] + " ");
   }

   public int[] obrniNiz(int[] a) {
      int[] b = new int[a.length];

      for (int i = 0; i < a.length; i++)
         b[i] = a[a.length - 1 - i];

      return b;
   }
}
```

A. 1 2 3 4 5 B. 5 4 3 2 1 C. 5 4 1 2 3 D. 1 2 5 4 3

25. Šta se prikazuje na ekranu za vrednosti niza `lista2` kao rezultat izvršavanja ovog programa?

```
public class Test {

   public static void main(String[] args) {
      int[] lista1 = {1, 2, 3, 4, 5};

      int[] lista2 = obrniNiz(lista1);

      for (int i = 0; i < lista2.length; i++)
         System.out.print(lista2[i] + " ");
   }

   public int[] obrniNiz(int[] a) {
      int[] b = new int[a.length];

      for (int i = 0; i < a.length; i++)
```

```
            b[i] = a[a.length - 1 - i];

        return b;
    }
}
```

A. 1 2 3 4 5 B. 5 4 3 2 1 C. 5 4 1 2 3 D. 1 2 5 4 3

26. Analizirajte sledeći program:

```
public class Test {
    public static void main(String[] args) {
        int[] x = {1, 2, 3, 4, 5};
        rMetod(x, 5);
    }

    public static void rMetod(int[] x, int n) {
        System.out.print(x[n - 1] + " ");
        rMetod(x, n - 1);
    }
}
```

A. Program na ekranu prikazuje `1 2 3 4 5`.

B. Program na ekranu prikazuje `1 2 3 4 5` i zatim grešku o prekoračenju granica indeksa niza `x`.

C. Program na ekranu prikazuje `5 4 3 2 1`.

D. Program na ekranu prikazuje `5 4 3 2 1` i zatim grešku o prekoračenju granica indeksa niza `x`.

27. Ako je data deklaracija `Krug[] k = new Krug[10]`, koja je od ovih rečenica najtačnija?

A. Promenljiva `k` sadrži niz od 10 celobrojnih vrednosti.

B. Promenljiva `k` sadrži niz od 10 objekata klase `Krug`.

C. Promenljiva `k` sadrži referencu na niz od 10 promenljivih klasnog tipa `Krug`.

D. Promenljiva `k` sadrži objekat klase `Krug` prečnika 10.

28. Ako je zaglavlje glavnog metoda klase `Test` dato u standardnom obliku `public static void main(String[] args)`, koji element niza `args` dobija vrednost stringa `"abc"` ukoliko je izvršavanje klase `Test` pokrenuto ovom DOS komandom?

```
java Test "+" 3 "abc" 2
```

A. `args[0]` B. `args[1]` C. `args[2]` D. `args[3]`

29. Koja su od ovih zaglavlja metoda `prikaži()` sa promenljivim brojem argumenata ispravne?

A. `public void prikaži(String... niska, double... broj)`
B. `public void prikaži(double... broj, String ime)`
C. `public void double... prikaži(double d1, double d2)`
D. `public void prikaži(double... broj)`
E. `public void prikaži(int n, double... broj)`

30. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {

      double[] d = {1.0, 2.0, 3.0};

      System.out.println(prosek(d));
      System.out.println(prosek(1, 2, 2, 1, 4));
      System.out.println(prosek(new double[]{1, 2, 3}));
      System.out.println(prosek(1.0, 2.0, 2.0, 1.0));
   }

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

A. Program ima grešku u prvoj naredbi `println`, jer je nepravilan poziv `prosek(d)`.
B. Program ima grešku u drugoj naredbi `println`, jer je nepravilan poziv `prosek(1, 2, 2, 1, 4)`.
C. Program ima grešku u trećoj naredbi `println`, jer je nepravilan poziv `prosek(new double[] {1, 2, 3})`.
D. Program ima grešku u četvrtoj naredbi `println`, jer je nepravilan poziv `prosek(1.0, 2.0, 2.0, 1.0)`.

E. Program se izvršava bez greške i prosek datih brojeva se tačno izračunava.
F. Program se izvršava bez greške, ali se prosek datih brojeva ne izračunava tačno.

31. Kojim od ovih poziva se sortira niz `lotoBrojevi` tipa `int[]`.

A. `Arrays(lotoBrojevi)`
B. `Arrays.sort(lotoBrojevi)`
C. `Arrays.sorts(lotoBrojevi)`
D. `Arrays.sortArray(lotoBrojevi)`

32. Ako je data deklaracija niza `int[] lotoBrojevi = {5, 8, 17, 23, 27, 33, 36}`, šta je rezultat poziva `Arrays.binarySearch(lotoBrojevi, 17)`?

A. 0 B. −1 C. 1 D. 2 E. −2

33. Koja je od ovih deklaracija ispravna?

A. `char[][] z = {'a', 'b'};`
B. `char[2][2] z = {{'a', 'b'}, {'c', 'd'}};`
C. `char[2][] z = {{'a', 'b'}, {'c', 'd'}};`
D. `char[][] z = {{'a', 'b'}, {'c', 'd'}};`

34. Ako je data deklaracija niza `double[][] d = new double[4][5]`, koje su vrednosti dužina `d.length` i `d[2].length`?

A. 4 i 4 B. 4 i 5 C. 5 i 4 D. 5 i 5

35. Analizirajte sledeći program:

```
public class Test {

   public static void main(String[] args) {

      boolean[][] x = new boolean[3][];

      x[0] = new boolean[1];
      x[1] = new boolean[2];
      x[2] = new boolean[3];

      System.out.println("x[2][2] je " + x[2][2]);
   }
}
```

A. Program ima grešku, jer je `new boolean[3][]` nepravilno.
B. Program ima grešku, jer će `x[2][2]` imati vrednost `null`.
C. Program se normalno izvršava i na ekranu se prikazuje `x[2][2] je null`.
D. Program se normalno izvršava i na ekranu se prikazuje `x[2][2] je false`.

✧ ✧ ✧

1. Napisati program kojim se prikazuje niz svih prostih brojeva manjih od dateg broja m koristeći postupak Eratostenovog sita: redom isključiti proizvode svih prostih brojeva manjih od $\sqrt{m}$, a oni brojevi koji preostanu su prosti. Granicu niza prostih brojeva m preuzeti iz komandnog reda.

2. Napisati program koji simulira „igru života". Ova igra se sastoji od kolonije organizama koji žive u sopstvenim ćelijama u jednoj dvodimenzionalnoj matrici ćelija. Konfiguracija organizama se menja u diskretnim vremenskim trenucima po generacijama, pri čemu je svaka ćelija matrice prazna ili zauzeta živim organizmom. Nova generacija organizama u ćelijama nastaje na osnovu stare generacije organizama zavisno od sadržaja osam susednih ćelija svake pojedine ćelije. (Ćelije na obodu matrice se podrazumevaju da na odgovarajućoj strani uvek imaju prazne susedne ćelije.) Pravila za formiranje nove generacije organizama su:

1. Živi organizam u ćeliji preživljava u sledećoj generaciji ukoliko je broj njegovih suseda dva ili tri.
2. Živi organizam u ćeliji umire u sledećoj generaciji ukoliko je broj njegovih suseda manji od dva (zbog usamljenosti) ili veći od tri (zbog prenaseljenosti).
3. U praznoj ćeliji se rađa novi organizam ukoliko se u tačno tri njene susedne ćelije nalazi živi organizam.

Drugim rečima, za ćelije u svakoj generaciji pravila prelaza su: puna ćelija ostaje puna ako ima dve ili tri pune susedne ćelije; puna ćelija postaje prazna ako ima manje od dve ili više od tri pune susedne ćelije; prazna ćelija postaje puna ako ima tačno tri pune susedne ćelije, a u suprotnom ostaje prazna.

Igra života počinje od zadate početne konfiguracije koja se učitava na ulazu. Zatim se u diskretnim trenucima redom formiraju sledeće

konfiguracije organizama *istovremenom* primenom gornjih pravila na sve ćelije prethodne konfiguracije (tj. nova generacija se formira isključivo na osnovu prethodne generacije).

3. Napisati program kojim se simulira igranje igre iks-oks sa računarom. Vrste i kolone table za igranje numerisane su brojevima 0, 1 i 2 radi lakšeg korišćenja polja table pomoću njihovih koordinata. Program treba da obaveštava korisnika o koordinatama svog izabranog polja kada je na potezu, a korisnik svoj potez zadaje koordinatama željenog polja. Posle svakog odigranog poteza programa (X) ili korisnika (O), na ekranu treba prikazati trenutni sadržaj table, na primer:

```
-------------
| X |   |   |
-------------
| O | X | O |
-------------
|   | O | X |
-------------
```

4. Napisati klasu kojom se realizuje keš-memorija. Keš-memorija (engl. *cache*) je struktura podataka koja se sastoji od niza elemenata fiksne dužine. U niz se mogu samo dodavati novi elementi i pritom se novi element uvek dodaje na prvo mesto niza. Ali, ukoliko se novi element već nalazi u nizu, elementi ispred njega se pomeraju za jedno mesto udesno. A ukoliko se novi element ne nalazi u nizu, postojeći elementi se takođe pomeraju za jedno mesto udesno, ali se poslednji element odbacuje ako je niz već popunjen.

5. Iskoristiti prethodni zadatak i napisati program u kojem se otkrivaju često ponavljane reči u tekstu. Jedna reč u nekom tekstu se često ponavlja ukoliko se pojavljuje više od jedanput unutar bilo koje sekvence susednih reči od, recimo, 30 jedinstvenih reči. Minimalna funkcionalnost koju program mora imati je:

a) Program treba svaku reč da konvertuje u mala slova kako se ne bi razlikovale iste reči sa različitom veličinom slova.
b) Ukoliko je neka reč skoro ponovljena u ulaznom tekstu, program treba da prikaže tu reč i broj puta koliko je ta reč bila ponovljena.

6. Napisati program kojim se upravlja čekaonicom jednog doma zdravlja. U domu zdravlja radi više doktora čiji se podaci unose na početku programa. Svaki doktor ima svoju specijalnost koja je predstavljena slov-

nim kodom: „o" za lekara opšte prakse, „k" za kardiologa, „g" za ginekologa i tako dalje. Kada pacijent dođe u dom zdravlja, prijavljuje se na recepciju i medicinska sestra na osnovu simptoma bolesti odlučuje o specijalnosti doktora koji treba da primi pacijenta. Kada jedan doktor odgovarajuće specijalnosti postane slobodan, pacijent koji najduže čeka za tu specijalnost ide kod takvog doktora na pregled.

U domu zdravlja radi više doktora iste specijalnosti i oni naizmenično primaju pacijente. Ako doktor završi pregled i nema pacijenata koji čekaju za njegovu specijalnost, doktor je slobodan i odmara se sve dok ne bude potreban. Kada je jedan doktor neke specijalnosti potreban, pacijenta prima doktor te specijalnosti koji se najduže odmarao.

Prema tome, pacijenti se primaju kod doktora na fer način (oni koji duže čekaju pre će biti primljeni) i doktori se odmaraju na fer način (oni koji su se duže odmarali pre će primiti nove pacijente). Ali, moguće je da pacijenti čekaju iako ima slobodnih doktora, pod uslovom da nijedan od slobodnih doktora nije potrebne specijalnosti.

Pored opcija za prijavljivanje novog pacijenta i registrovanje pregledanog pacijenta, program treba da sadrži još tri dodatne opcije: za prikazivanje pacijenata u čekaonici po redu po kome će biti primljeni kod doktora; za prikazivanje slobodnih doktora po redu po kome će primiti pacijente; i za prikazivanje aktuelnih pregleda, odnosno parova zauzetih doktora i njihovih trenutnih pacijenata.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

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] Herbert Schildt. *Java Programming Cookbook*. McGraw-Hill, 2007.

[15] Dejan Živković. *Java programiranje*. Univerzitet Singidunum, 2013.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

Indeks

A

argumenti metoda, 140
aritmetički operatori, 70
`Arrays`, 234
autopakovanje, 61

B

beskonačna petlja, 104
blok naredbi, 87
    prazan blok, 88
`break` naredba, 118

C

`continue` naredba, 118
curenje memorije, 204

D

definicija metoda, 140
    specifikatori pristupa, 143
`do-while` naredba, 111
dvodimenzionalni nizovi, 240

E

eksplicitna konverzija tipa, 71
enkapsulacija (učaurivanje), 204

F

Fibonačijev niz brojeva, 169
`final`
    konstante, 49
`for` naredba, 113

G

geteri, 206
glavna memorija, 3

H

Hanojske kule, 170
hardver, 1
hip-memorija, 191

I

imena (identifikatori), 36
interpretator, 5
izlazni podaci, 3
izrazi, 69

J

Java bajtkod, 6
Java programski jezik
    osnovni elementi, 35
Java virtuelna mašina, 6
jezici visokog nivoa, 5

K

klasa, 23
    polja, 157
    članovi, 187
        nestatički (objektni), 50, 187
        statički (klasni), 49, 187
klase omotači, 60
    `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`, 60
komentari, 35
    dokumentacioni, 36
kompajleri, 5
konstante, 48
    `final`, 48
konstruktor, 198

L

literali, 40
logički operatori, 75

M

Math, 53
  abs(), 54
  cos(), 54
  exp(), 54
  floor(), 54
  log(), 54
  pow(), 54
  random(), 54
  sin(), 54
  sqrt(), 53
  tan(), 54
MAX_VALUE, 61
mašinski jezik, 4
metod, 139
  argumenti, 140
  definicija, 140
  geter, 206
  konstruktor, 198
  parametri, 140
  potpis, 156
  pozivanje, 140
  preopterećeni, 156
  promenljiv broj argumenata, 238
  rekurzivni, 165
  rezultat, 140
  seter (mutator), 206
  telo, 140
  vraćanje rezultata, 149
  zaglavlje, 140
MIN_VALUE, 61

N

najveći zajednički delilac, 168
NaN, 62
naredba continue, 119
naredba definicije (deklaracije) promenljive, 45
naredba dodele, 43
naredba povratka, 149
naredbe grananja, 90
  if-else naredba, 91
  if naredba, 90
  switch naredba, 98
  ugnježđavanje, 93
naredbe ponavljanja, 104
  do-while naredba, 111
  for-each petlja, 237
  for naredba, 113
    kontrolna promenljiva (brojač), 114
  while naredba, 104
NEGATIVE_INFINITY, 62
NetBeans, 10
  projekat, 12
niz, 223
  bazni tip, 224
  dužina, 223
  dvodimenzionalni, 240
    matrica (tabela), 240
  elementi, 223
    indeks, 223
  inicijalizacija, 227
  jednodimenzionalni, 223
  length, 227
  višedimenzionalni, 240

O

objekat, 21
  instanca klase, 187
  konstrukcija i inicijalizacija, 196
objektno orijentisano programiranje, 185
oblast važenja imena, 162
operativni sistem, 1
operator new, 58, 191, 200
operator dodele, 77
operator izbora, 76
označena petlja, 120

P

paket, 24
  import, 25
  package, 26
  anonimni paket, 26
  korišćenje, 25
palindrom, 121
parametri metoda, 140
petlja, 104
  iteracija, 104
  označena, 120
  telo petlje, 104
  ugnježđavanje, 107
  uslov nastavka (ili prekida), 104
polja, 46, 157
POSITIVE_INFINITY, 62
potpis metoda, 156

pozivanje metoda, 140, 144
  prenos argumenata po vrednosti, 145
prazna naredba, 95
preopterećeni metodi, 156
prevodioci, 5
prioritet operatora, 78
procesor, 3
programiranje, 2
  objektno orijentisano, 185
programski jezik Java
  aplet, 20
  aplikacija, 20
  dokumentacija, 25
  istorijski razvoj, 19
  komentar, 29
  objektno orijentisani jezik, 20
  platforma, 19
  servlet i JSP, 20
  slobodni format, 32
promenljiva, 43
  alociranje, 158
  dealociranje, 158
  dužina trajanja, 158
  globalna, 157
    zaklonjena, 162
  lokalna, 46, 89, 157
  oblast važenja, 89

R

raspakovanje, 61
referenca (pokazivač), 191
rekurzivni metodi, 165
  bazni slučaj, 167
relacijski operatori, 73
`return` naredba, 149
rezultat metoda, 140

S

sabirnica (magistrala), 4
sakupljanje otpadaka, 203
`Scanner`, 58
  `hasNext()`, 59
  `hasNextDouble()`, 60
  `hasNextInt()`, 60
  `hasNextLine()`, 60
  `next()`, 59
  `nextDouble()`, 59
  `nextInt()`, 59
  `nextLine()`, 59
seteri (mutatori), 206
složena (kvalifikovana) imena, 37
službene (rezervisane) reči, 37
softver, 1
standardni izlaz, 50
standardni ulaz, 59
`String`, 54
  `charAt()`, 56
  `compareTo()`, 56
  `equals()`, 56
  `equalsIgnoreCase()`, 56
  `indexOf()`, 56
  `substring()`, 56
  `toLowerCase()`, 56
  `toUpperCase()`, 56
  `trim()`, 57
stringovi, 42
  spajanje (konkatenacije), 42
strukture podataka, 223
`System`, 50
  `System.in`, 59
  `System.out`, 50
  `System.out.print()`, 50
  `System.out.printf()`, 51
  `System.out.println()`, 50

T

tačka-notacija, 37
telo metoda, 140
telo petlje, 104
`this`, 208
  implicitni argument metoda, 209
  pozivanje preopterećenog konstruktora, 210
tipovi podataka
  klasni, 38
  primitivni, 38
    celobrojni, 38
    logički, 41
    realni, 39
    znakovni, 39

U

uklanjanje objekata, 202
ulazni podaci, 2

ulazno/izlazni uređaji, 4
Unicode, 39
uslov nastavka (ili prekida), 104

V

viseći pokazivači, 204
višedimenzionalni nizovi, 240
višekratna upotreba, 201

W

`while` naredba, 104

Z

zaglavlje metoda, 140

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд
004.438JAVA(075.8)
004.42:004.738.5(075.8)
ЖИВКОВИЋ, Дејан, 1956-
Osnove Java programiranja / Dejan Živković. - 9. izd. - Beograd : Univerzitet Singidunum, 2018 (Loznica : Mobid). - VI, 272 str. : ilustr. ; 24 cm
Tiraž 1.000. - Bibliografija: str. 267. - Registar.
ISBN 978-86-7912-558-3
a) Програмски језик «Java» b) Интернет - Програмирање
COBISS.SR-ID 265710092

© 2018.
Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.

www.singidunum.ac.rs

9 788679 125583



Dejan Živković

OSNOVE JAVA PROGRAMIRANJA

OBJEKTNO ORIJENTISANI KONCEPT PROGRAMIRANJA

Programski jezik Java se koristi za rešavanje zadataka na računaru iz različitih oblasti ljudske delatnosti. Zato programi napisani u Javi mogu biti obični tekstualni ili grafički programi, ali i složenije aplikacije zasnovane na principima mrežnog programiranja, višenitnog programiranja, rada sa bazama podataka i mnogih drugih tehnologija. Obim i sadržaj ove knjige, međutim, prilagođeni su nastavnom planu i programu odgovarajućeg predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. Knjiga pruža solidnu osnovu za dalje učenje svima koji žele da se profesionalno bave Java programiranjem. Informacije o dodatnim nogućnostima Jave mogu se pronaći u literaturi koja je navedena na kraju knjige.

Java ima mnoge osobine svojstvene većini programskih jezika. Java će izgledati poznato C I C++ programerima, jer su mnoge programske konstrukcije preuzete iz tih jezika skoro bez izmena. Zbog toga je predznanje čitalaca iz ovih jezika velika olakšica za učenje Jave, ali to nije preduslov za uspešno praćenje ove knjige. Napomenimo i da knjiga ne predstavlja uvod u objektno orijentisano programiranje, iako je Java jedan objektno orijentisan programski jezik. U knjizi su ipak izloženi neki osnovni koncepti klasa i objekata kako bi se uspostavila neophodna terminologija.

Glavni cilj pisanja ove knjige je predstavljanje osnova Java programiranja na najjasniji način. U tekstu je stoga poštovano pedagoško načelo da izlaganje materijala iz svake oblasti treba da ide od poznatog ka nepoznatom i od konkretnog ka apstraktnom. Tekst je propraćen velikim brojem slika i urađenih primera kojima se konkretno ilustruju novouvedeni pojmovi. Primeri su birani tako da budu jednostavni, ali i da budu što realističniji i interesantniji. Njihova dodatna svrha je da posluže kao početna tačka za eksperimentisanje i dublje samostalno učenje.

Univerzitet Singidunum