Laslo Kraus

PROGRAMSKI JEZIK

Java

sa rešenim zadacima

AKADEMSKA MISAO
Beograd, 2013

Laslo Kraus

PROGRAMSKI JEZIK JAVA
SA REŠENIM ZADACIMA

Recenzenti
Dr Igor Tartalja
Mr Đorđe Đurđević

Lektor
Anđelka Kovačević

Korice
Zorica Marković, akademski slikar

Izdavač
AKADEMSKA MISAO
Bul. kralja Aleksandra 73, Beograd

Štampa
Planeta print, Beograd

Tiraž
300

ISBN 86-7466-???-?

Dušanki i Katarini

Predgovor

Ova knjiga predstavlja udžbenik za programski jezik *Java* za široki krug čitalaca. Knjigu mogu da koriste i početnici u programiranju, ali poznavanje osnovnih pojmova iz objektno orijentisanog programiranja i programskih jezika C/C++ ili C# znatno olakšava da se savlada materija iz ove knjige.

Programski jezik *Java* izložen je u obimu koji može da zadovoljava i naprednije neprofesionalne programere. Od ogromne standardne biblioteke klasa, koja prati jezik *Java*, objašnjeni su samo delovi koji su potrebni za efikasno programiranje pri rešavanju većine problema relativno visoke složenosti.

Jedini način da se nauči neki programski jezik jeste da se pišu programi na njemu. Ova knjiga je u potpunosti podređena tom osnovnom načelu.

Uvodno poglavlje 1 sadrži i informacije korisne pre svega za početnike: osnovne pojmove objektno orijentisanog programiranja, glavne osobine programskog jezika *Java* i načina obrade programa u tekstualnom i grafičkom okruženju.

U poglavlju 2 obrađeni su podaci koji su predmet obrade u programima. Objašnjen je pojam tipa podataka, načini definisanja i korišćenja skalarnih brojčanih, znakovnih i logičkih podataka. Da bi što pre mogli da se pišu potpuni programi u ovom poglavlju je obrađena i ulazna i izlazna konverzija brojčanih podataka. Naime, programi koji ne vrše čitanje početnih podataka i ispisivanje rezultata nemaju nikakvu upotrebnu vrednost.

Poglavlje 3 je posvećeno operatorima i načinima sastavljanja izraza pomoću njih. Tu su obrađeni operatori jezika *Java* koji čitaocu s prosečnim matematičkim obrazovanjem ne bi trebalo da budu strani. Prikazivanje nekih specifičnijih operatora je odloženo za kasnije. Kao proširenje skupa operatora dat je i pregled važnijih standardnih bibliotečkih funkcija koje postoje u jeziku *Java*. Odabrane su funkcije koje se odnose na opštepoznate pojmove iz matematike i na obradu već objašnjenih tipova podataka.

U poglavlju 4 prikazane su naredbe koje predstavljaju jedinične obrade u programima. Pored prostih naredbi obrađene su složene naredbe jezika *Java* koje omogućavaju sastavljanje složenih programa na efikasan i pregledan način: uslovno izvršavanje (selekcije), višestruko izvršavanje (ciklusi) delova programa i upravljačke naredbe (skokovi).

Iz matematike dobro poznati nizovni tipovi: vektori, matrice i višedimenzionalni nizovi obrađeni su u poglavlju 5.

Izučavanje objekto orijentisanog programiranja počinje u poglavlju 6 o klasama. Klase su složeni tipovi podataka koji programerima omogućuju modeliranje objekata iz realnog života. Sastoje se od polja čije vrednosti čine stanja primeraka klasa i metoda koje definišu

moguće načine korišćenja tih polja. Primerci klasa su podaci klasnih tipova i nazivaju se objektima. Jedan od važnih elemenata ispravnog rada programa je pravilna inicijalizacija objekata.

Programski jezik *Java* omogućava grupisanje srodnih klasa u pakete. U poglavlju 7 obrađeni su paketi: njihovo formiranje i korišćenje.

Poglavlje 8 posvećeno je sledećem fundamentalnom konceptu objektno orijentisanog programiranja, nasleđivanju. U grupama podataka koje su predstavljene klasama mogu da se uoče podgrupe sa nekim specifičnijim osobinama. Te podgrupe, predstavljene potklasama, nasleđuju osobine širih grupa, natklasa. Član podgrupe uvek može da se koristi i kao član opštije nadgrupe. U ovom poglavlju obrađeni su specijalni apstraktni tipovi, interfejsi, koji samo definišu određene osobine koje klase koje ih ostvaruju treba da poseduju, a ne i način kako se te osobine ostvaruju.

U poglavlju 9 obrađeni su ugnežđeni tipovi, klase i interfejsi koji su definisani unutar drugih klasa i interfejsa.

Jezik *Java* omogućava da rukovanje izuzecima (greškama) vremenski ne opterećuje program sve dok se izuzetak ne pojavi. Kada se izuzetak pojavi, kontrola se automatski predaje jednom od rukovalaca izuzecima koje je definisao programer. Način rukovanja izuzecima prikazan je u poglavlju 10.

U poglavlju 11 objašnjeni su generički tipovi i metode. Generičke klase i metode omogućavaju pisanje klasa i metoda za obradu podataka, čiji tipovi nisu poznati u vreme njihovog pisanja, na bezbedan način sa stanovišta provera ispravnog korišćenja tipova podataka. Generički interfejsi, na sličan način, opisuju očekivane osobine budućih klasa u uslovima kada tipovi pojedinih korišćenih podataka nisu poznati.

U poglavlu 12 izloženi su osnovni koncepti konkurentne obrade, obrade koja se sastoji od istovremenog odvijanja više tokova kontrole, više istovremenih obrada. Ti tokovi kontrola nazivaju se niti, čija je korektna sinhronizacija osnova dobrog programa s konkurentnom obradom.

Rukovodeći se već istaknutim načelom o učenju jezika, ova knjiga, pored manjih primera u toku izlaganja, na kraju svakog poglavlja sadrži nekoliko rešenih zadataka. Više zadataka može da se nađe u autorovoj zbirci *Rešeni zadaci iz programskog jezika Java* (videti [7]). Kroz zadatke u ovoj knjizi i u zbirci skreće se pažnja na neke specifičnosti jezika *Java* i daju ideje za elegantno i efikasno rešavanje nekih problema.

Ova knjiga je više nego udžbenik za programski jezik *Java*. Kroz rešene zadatke prikazane su i primene osnovnih elemenata objektno orijentisanog programiranja, a na primerima koji se uopšteno sreću u računarskoj tehnici: obradi redova, stekova, listi, tekstova, datoteka itd. Posebna pažnja posvećena je i inženjerskim aspektima programiranja. Nije dovoljno da program rešava zadati problem, već treba da bude i „lep”, razumljiv, da se lako održava, da troši što manje memorije i da bude što efikasniji.

Svi programi koji se nalaze u ovoj knjizi potpuni su u smislu da mogu da se izvršavaju na računaru. Provereni su na nekoliko računara korišćenjem nekoliko različitih prevodilaca za jezik *Java*.

Izvorni tekstovi svih programa iz ove knjige mogu da se preuzmu sa Interneta, sa adrese `home.etf.rs/~kraus/knjige/`. Na istoj adresi objaviće se ispravke eventualnih, naknadno otkrivenih grešaka u knjizi. Svoja zapažanja čitaoci mogu da upute autoru elektronskom poštom na adresu `kraus@etf.rs`.

Laslo Kraus

Beograd, *februara 2013.*

Sadržaj

1 Uvod

1.1 O programskom jeziku *Java*

Programski jezik *Java* je viši programski jezik opšte namene koji u potpunosti podržava paradigmu objektno orijentisanog programiranja.

Java je više od programskog jezika koji ima određenu sintaksu i semantiku. Ona je pravo programsko okruženje koje pomoću ogromne biblioteke standardnih programskih modula podržava obrade bez kojih danas ne može da se zamisli nijedan ozbiljan program. Tu spadaju komunikacija sa korisnikom preko prozora, višenitna obrada, rad preko *Interneta*, troslojna arhitektura rada s bazama podataka itd. Standardi programskih jezika do pojave jezika *Java* te elemente nisu obuhvatali. Svaki proizvođač je uvodio nestandardna rešenja za njih, ograničavajući time prenosivost programa s jedne na drugu platformu.

Programski jezik *Java* je srodnik jezika *C++* koji je direktni potomak jezika *C*.

Pogramski jezik *C* pojavio se početkom 1970-tih godina i u osnovama je promenio dotadašnji, nesistematizovani, stil programiranja. Kroz podršku za *strukturirano programiranje* obezbedio je da mogu da se napišu, razumeju i održavaju znatno veći programski sistemi nego pre. Pošto se u centru pažnje nalaze postupci koji se primenjuju na podatke, ta tehnika programiranja naziva se i *proceduralno programiranje*.

Pošto su jezik *C* projektovali programeri koji su i sami pisali programe, drugi programeri su ga rado prihvatili i uskoro je postao jedan od najrasprostranjenijih programskih jezika, koji se koristi i danas. Za autora jezika *C* smatra se Denis Riči (*Dennis Ritchie*) iz Belovih laboratorija (*Bell Laboratories*).

Početkom 1980-tih godina računarski programi narasli su do takvih veličina da su se tehnike proceduralnog programiranja pokazale neefikasnim. Rešenje se našlo u paradigmi *objektno orijentisanog programiranja*, kod koje se u centru pažnje nalaze podaci (objekti) na koje se primenjuju neki postupci. Od programskih jezika koji su projektovani da podržavaju nove tehnike programiranja najveći uspeh je imala nadogradnja jezika *C*, koja je danas poznata pod imenom *C++* (++ je u jeziku *C* operator povećavanja!). Preko 95% jezika *C* usvojeno je bez izmena i u jeziku *C++*.

C++ je hibridni jezik u smislu što, pored novijeg objektno orijentisanog programiranja, podžava i starije proceduralno programiranje.

Za autora jezika *C++* smatra se Bjarn Strostrup (*Bjarne Stroustrup*).

Početkom 1990-tih godina pojavila se potreba za programima koji su u prevedenom obliku nezavisni od platforme, tj. mogu da se izvršavaju na bilo kom računaru bilo kog proizvođača. Radilo se prvenstveno o programima za potrebe industrijske elektronike za upravljanje kućnim aparatima.

Postojeći viši programski jezici bili su, bar teorijski, prenosivi na nivou izvornog teksta programa. Taj tekst morao zasebno je da se prevede u izvodljivi oblik za svaku vrstu računara. To bi u uslovima naraslog broja različitih platformi tražilo izradu velikog broja prevodilaca sa višeg programskog jezika na mašinski jezik računara. Trebalo je projektovati programski jezik koji omogućava da prevedeni oblik programa bude nezavisan od platforme na kojoj se izvršava.

Zbog toga je grupa programera u firmi *Sun Microsystems, Inc.* s Džemsom Gazlingom (*James Gosling*) na čelu započela projektovanje novog programskog jezika koji je trebalo da bude lak za korišćenje, pouzdan i nezavisan od platforme na nivou prevedenog oblika. Taj jezik se u početku zvao *Oak* (hrast), a ime *Java* dobio je 1995. godine.

Nekako baš u vreme razvijanja jezika *Java* počelo je intenzivno širenje *Interneta* i posebno njegove usluge *World Wide Web* za pružanje najraznovrsnijih informacija, organizovanih u takozvane "veb stranice", sa specijalnih servera do proizvoljnih klijenata širom sveta. Veb stranice su u početku bile statične: sadržavale su samo tekstove i slike (u početku nepokretne, kasnije i pokretne). Za dinamični sadržaj, koji podrazumeva i interakciju s klijentima, trebalo je u veb stranice ugraditi male programe. Ti programi, pošto bi se izvršavali na računaru klijenta, morali su biti nezavisni od platforme i bezbedni da ne naprave štetu na klijentskom računaru.

Programski jezik *Java* imao je upravo te osobine, pa je današnju popularnost stekao kroz takozvane aplete (*applets*), male programe koji se ugrađuju u veb stranice. U međuvremenu apleti kao dinamični sadržaj veb stranica prevaziđeni su, ali i nove tehnologije programiranja za *Internet* u velikoj meri se oslanjaju na jezik *Java*.

Kao programski jezik opšte namene, *Java* se koristi i za izradu samostalnih programa koji se nazivaju *aplikacije*.

Programski jezik *Java* nema zvaničan standard. Kao *de facto* standard uzima se specifikacija koju je u početku publikovala firma *Sun Microsystems, Inc*., a u novije vreme firma *Oracle*, koja je kupila firmu *Sun*.

Jezik je još u intenzivnom razvoju i publikovan je veći broj izdanja. Prvo izdanje je bila verzija 1.0, ali je ubrzo zamenjena verzijom 1.1 koja nije donela bitne izmene. Verzija 1.2 donela je značajne izmene i od tada se jezik skraćeno zove *Java* 2, a punim imenom *Java J2SE* (*Java 2 Platform Standard Edition*). Sledile su verzije 1.3 i 1.4 koje su donele manje izmene. Verzija 1.5 donela je ogromne promene, pa da bi se to naglasilo, ta verzija je proglašena verzijom 5 (*Java J2SE 5*). Aktuelna verzija u vreme pisanja ove knjige je 7.

1.2 Uvod u objektno orijentisano programiranje

U svakom računarskom programu mogu da se uoče dve grupe elemenata: naredbe i podaci. Naredbe određuju šta se radi, a podaci čime se radi. Organizacija programa može da bude orijentisana ka jednoj od ove dve grupe elemenata.

Klasičan stil programiranja okrenut je prema postupcima i naziva se **proceduralno programiranje** (*procedural programming*). Po tom modelu program se sastoji od niza uzastopnih koraka. Logičke celine koraka mogu da se ostvaruju u obliku modula koji se

nazivaju potprogrami, procedure ili funkcije. Složena obrada ostvaruje se kombinovanjem takvih modula.

Mana ovakvog pristupa programiranju je veliki stepen povezanosti delova složenog sistema. To otežava održavanje i unapređivanje sistema. Da bi se unele nove mogućnosti često je potrebno preraditi vrlo veliki deo već gotovih delova programa.

Dva programska jezika koji podržavaju proceduralno programiranje i koji su u današnje vreme vrlo rašireni jesu *Pascal* i *C*.

Savremen stil programiranja okrenut je prema podacima i naziva se **objektno orijentisano programiranje** (*OOP – object-oriented programming*). Po tom modelu program se sastoji od **objekata** (*objects*) koji imaju neka moguća stanja i ponašanja. Stanja predstavljaju vrednosti objekata, koje vremenom mogu da se menjaju. Ponašanja predstavljaju pravila menjanja stanja, reakcije na uticaje okoline i načine uticanja na okolinu. Celokupna obrada ostvaruje se u obliku međusobnih delovanja objekata u programu.

Svi objekti u programu grupišu se po svojim osobinama. Grupe objekata sa sličnim osobinama čine **klase** (*classes*). Objekti iste klase imaju ista moguća stanja i ista ponašanja. Objekti date klase predstavljaju pojedinačne primerke svojih klasa.

Pojam klase i objekta najlakše se shvata kroz primere iz svakodnevnog života. *Pas* (klasa) je životinja sa svima poznatim opštim osobinama i ponašanjem. *Lesi* (objekat) je jedan tačno određeni pas s konkretnim osobinama (boja, težina, starost) i ponašanjem koje se uklapa u ponašanje svih pasa. Drugi primer: *automobil* je opšte ime (klasa) za sva prevozna sredstva te vrste, dok je *automobil BG0720AX* jedan tačno određeni automobil (objekat).

Klase su tipovi podataka prema definiciji tipova podataka u programiranju, jer određuju moguće vrednosti svojih primeraka i moguće radnje nad tim primercima. Po tome, objekti su podaci klasnih tipova. Za razliku od prostih tipova podataka, kao što su celi brojevi ili realni brojevi, klase su složeni tipovi podataka.

Stanja objekata predstavljaju se podacima unutar objekata i koji se nazivaju **polja** (*fields*). Polja mogu da budu podaci prostih i složenih tipova. U proste tipove podataka spadaju celi brojevi, realni brojevi itd., a u složene tipove nizovi i klase.

Ponašanje objekata ostvaruje se postupcima unutar klasa koji se nazivaju **metode** (*methods*). Metode odgovaraju funkcijama i procedurama u klasičnim programskim jezicima.

Polja i metode klasa zajedničkim imenom nazivaju se **članovi** (*members*) klasa.

Prednosti objektno orijentisanog programiranja ogledaju se u tome što je pri menjanju mogućnosti programskog sistema potrebno prerađivati samo mali deo već gotovog programa. Ako se prošire mogućnosti neke klase nije potrebno promeniti i deo programa koji koristi tu klasu. Naravno, pod uslovom da se ništa ne promeni u načinu korišćenja mogućnosti klase koje su postojale i pre promene. Takođe, ako se u programski sistem uvode nove klase, deo programa koji koristi samo stare klase ne treba promeniti.

Najviše zastupljeni jezici za objektno orijentisano programiranje su *C++*, *Java* i, u novije vreme, *C#*.

Objektno orijentisano programiranje zasniva se na pet osnovnih principa: apstrakciji, učaurivanju, nasleđivanju, polimorfizmu i ponovnom korišćenju koda.

2 Podaci

Podaci su predmet obrade u programima. U vom poglavlju su uvedeni osnovni pojmovi o podacima u programiranju i prikazani su prosti tipovi podataka koji se koriste u jeziku *Java*.

2.1 Elementi jezika *Java*

Skup znakova (*character set)* koji se koristi u jeziku *Java* čine mala i velika slova engleskog alfabeta, mala i velika slova većine alfabeta sveta, deset decimalnih cifara i veći broj znakova interpunkcije. Pravi se razlika između malih i velikih slova kako u službenim delovima programa tako i unutar običnih tekstova.

U nastavku teksta pod pojmom *slova* podrazumevaju se mala i velika slova bilo kog alfabeta, a pod pojmom *cifre* decimalne cifre.

Mada jezik *Java* predviđa upotrebu slova svih alfabeta, u službenom delu jezika koriste se samo slova engleskog alfabeta. Takođe, postoje radna okruženja koja ne podržavaju upotrebu slova bilo kog alfabeta. Ona obično podržavaju upotrebu slova engleskog alfabeta i, eventualno, još jedne grupe alfabeta. Na primer: latinična slovazapadno evropskih jezika, latinična slova istočnoevropskih jezika ili ćirilična slova.

Leksički simboli (*tokens*) su nedeljivi nizovi znakova. U jeziku *Java* dele se na identifikatore, konstante, ključne reči, operatore i separatore. Leksički simboli mogu da se pišu, uz nekoliko izuzetaka, spojeno ili međusobno razdvojeno proizvoljnim brojem „belih" znakova. Beo znak između leksičkih simbola je neophodan, ako bi se njegovim izostavljanjem dobio drugi postojeći leksički simbol.

U **bele znakove** (*white spaces*) spadaju znak za razmak, tabulacija, vertikalna tabulacija, prelazak u novi red i prelazak na novi list.

U širem smislu, u bele znakove se ubrajaju i komentari. **Komentari** (*comments*) su proizvoljni tekstovi koji su namenjeni čitaocu teksta programa radi lakšeg razumevanja namene i funkcionisanja programa. Te tekstove prevodilac zanemaruje.

U jeziku *Java* postoje tri vrste komentara:

- Komentari započeti sa `//` obavezno traju do kraja reda i tu se završavaju. Zgodni su za kratke komentare iza službenih elemenata s početka reda.
- Komentari stavljeni između `/*` i `*/` mogu da se stave između bilo koja dva leksička simbola (kao i beli znakovi) i mogu da se protežu i kroz više redova. Mogu da posluže kao kratki umeci unutar redova ili da sadrže duža objašnjenja.
- Komentari stavljeni između `/**` i `*/` su takozvani *dokumentacioni komentari*. Sadržaj takvih komentara program `javadoc` iz kompleta *JDK* koristi za sastavljanje dokumentacije programa. Mada i dokumentacioni komentari mogu da se umeću unutar redova, oni se obično protežu na više redova.

Naredbe (*statements*) su nizovi leksičkih simbola. Dele se na deklarativne i izvršne naredbe. ***Deklarativnim naredbama*** (*declarative statements*) definišu se neki elementi programa (podaci, funkcije, klase itd.), ***izvršnim naredbama*** (*executive statements*) izvode se elementarne obrade.

Programi su nizovi naredbi pomoću kojih se ostvaruju složene obrade podataka. Jezik *Java* ne postavlja nikakve uslove za raspoređivanje naredbi po redovima teksta programa. Jedna naredba može da se proteže u više redova i u jednom redu može da bude više naredbi. Osnovno načelo treba da bude što veća preglednost teksta celokupnog programa.

2.2 Identifikatori

Identifikatori (*identifiers*) služe za označavanje svih vrsta elemenata programa: podataka, simboličkih konstanti, tipova podataka koje definiše programer, potprograma i oznaka koje služe kao odredišta za naredbe skokova.

Identifikatori mogu da se sastoje od slova, cifara, znaka podvučeno (_) i znaka dolar ($). Prvi znak u identifikatoru ne sme da bude cifra. Upotrebu znaka `$` treba izbegavati. On je rezervisan za identifikatore koje razni alati za generisanje programskih tekstova generišu automatski.

Pravi se razlika između malih i velikih slova. To znači da su `q` i `Q` dva različita identifikatora.

Identifikatori mogu da budu proizvoljno dugački.

Ključne reči (*keywords*) jezika *Java* su rezervisane reči (*reserved words*) i ne mogu da se koriste kao identifikatori. Te ključne reči su:

abstract	**assert**	**boolean**	**break**	**byte**
case	**catch**	**char**	**class**	**continue**
const	**default**	**do**	**double**	**else**
enum	**extends**	**false**	**final**	**finally**
float	**for**	**goto**	**if**	**implements**
import	**instanceof**	**int**	**interface**	**long**
native	**new**	**null**	**package**	**private**
protected	**public**	**return**	**short**	**static**
strictfp	**super**	**switch**	**synchronized**	**this**
throw	**throws**	**transient**	**true**	**try**
void	**volatile**	**while**		

Reči **`const`** i **`goto`** su uvrštene u ključne reči, mada se ne koriste u jeziku *Java*, jer njihova pojava u programu u bilo koje neslužbene svrhe mogla bi da zbuni programere koji su ranije koristili jezike *C* i *C++*.

Primer 2.1 *– Ispravni identifikatori*

```
alfa                AnaVoliMilovana
ALFA                vrloDugačkoImeAMožeDaBudeJošIDuže
Alfa                $nePreporučujeSe$
x55_123             ime_i_prezime
                    datumRođenja
```

■

Uobičajeno je da se u identifikatorima koji su sastavljeni od više reči početna slova pojedinih reči pišu velikim slovima, a ostala malim slovima.

Primer 2.2 *– Neispravni identifikatori*

```
13a55                // Prvi znak ne sme da bude cifra.
alfa,beta            // Znakovi interpunkcije nisu dozvoljeni.
x-y                  // Operatori nisu dozvoljeni.
int                  // Rezervisane reči nisu dozvoljene.
```

■

2.3 Tipovi podataka

Podaci (*data*) su predmet obrade u programima. Svaki podatak ima određene osobine koje čine tip podatka. Tip podatka (*data type*) određen je skupom mogućih vrednosti koje može da uzme podatak i skupom mogućih operacija koje mogu da se izvode nad podatkom.

Podaci u programu mogu da se predstavljaju pomoću vrednosti ili pomoću identifikatora.

Podaci predstavljeni pomoću vrednosti ne mogu da promene svoje vrednosti u toku izvršavanja programa i nazivaju se **konstante** (*constants*). Na primer, `55` je celobrojna konstanta, a `-128.76` realna konstanta (u programiranju se koristi decimalna tačka umesto decimalnog zareza).

Konstantne vrednosti mogu da se predstave i pomoću identifikatora i tada se nazivaju **imenovane konstante** (*named constants*). Na primer, `PI` može da bude imenovana konstanta koja predstavlja iz matematike dobro poznatu vrednost 3,14159. Uobičajeno je da se identifikatori imenovanih konstanti pišu samo velikim slovima. Ako se sastoje od više reči koristi se znak podvučeno (_) za razdvajanje reči (na primer: `SVETLO_SIVA`).

Smatra se da konstante ne zauzimaju prostor u memoriji računara već da se ugrađuju neposredno u naredbe programa.

Podaci predstavljeni pomoću identifikatora (osim imenovanih konstanti) smeštaju se u memoriju računara gde zauzimaju određeni prostor. Vrednost podatka se smešta u taj prostor. U nastavku pod pojmom „podatak" podrazumeva se samo ova vrsta podataka. Konstante su izuzete iz ovog pojma jer one ne mogu da se obrađuju, već samo da se njihove vrednosti koriste u izračunavanjima. Na primer, `x`, `y`, `obim`, `alfa` i `početnaBrzina` mogu biti identifikatori podataka.

Podacima koji su smešteni u memoriji računara, obično, mogu da se promene vrednosti u toku izvršavanja programa. Takvi podaci nazivaju se **promenljivi podaci** ili kraće **promenljive** (*variables*).

Postoje, doduše ređe, i podaci čije vrednosti ne mogu da se promene u toku izvršavanja programa. Oni se nazivaju **nepromenljivi podaci**. Ne nazivaju se konstante jer se smeštaju u memoriju kao i promenljivi podaci i ne mogu u programu da se koriste na isti način kao i gore definisane „prave" konstante.

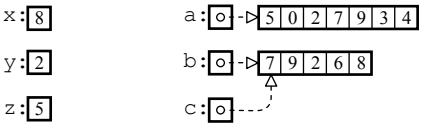
Podaci mogu da budu prosti ili složeni.

Podaci prostih tipova (*primitive types*) ili **prosti podaci** (*primitive data)* ne mogu da se rastave na manje elemente koji bi mogli nezavisno da se obrađuju. Zato se kaže da oni nemaju strukturu. Nazivaju se i *nestrukturirani podaci* ili *skalarni podaci*. U jeziku *Java*, prosti podaci su brojčani, znakovni i logički podaci.

Podaci složenih tipova (*compound types*) ili **složeni podaci** (*compund data)* sastoje se od nekoliko elemenata koji mogu da se nezavisno obrađuju. Elementi složenih podataka mogu da budu prosti, ali i sami mogu da budu složeni. Na taj način je broj različitih složenih tipova podataka, koji mogu da se izvode, polazeći od prostih tipova podataka, neograničen. Pošto složeni podaci imaju određenu strukturu nazivaju se i *strukturirani podaci*. U jeziku *Java*, složeni podaci su nizovi i klase.

Prema načinu uskladištavanja i pristupanja, podaci se dele na vrednosne i pokazane.

Promenljiva ***vrednosnog tipa*** (*value type*) sadrži vrednost podatka koga predstavlja (slika 2.1.a). Pod vrednošću promenljive podrazumeva se vrednost sadržanog podatka. Operacije nad promenljivom obrađuju taj, neposredno, sadržani podatak. Promena vrednosti jedne promenljive nikad ne utiče na vrednost druge.



a) *vrednosni* ***b)*** *pokazani*

Slika 2.1 *– Vrednosni i pokazani podaci*

Promenljiva ***pokazanog tipa*** (*reference type*) sadrži samo adresu mesta (pokazivač – *pointer*, *reference*) u memoriji gde se nalazi vrednost predstavljenog podatka (slika 2.1.b). Pod vrednošću promenljive podrazumeva se vrednost pokazanog podatka, a ne vrednost neposredno sadržanog pokazivača. Operacije nad promenljivom obrađuju pokazani podatak, a ne neposredno sadržani pokazivač. Više promenljivih mogu da pokazuju na fizički isti podatak. U tom slučaju promena vrednosti jedne promenljive promeniće i vrednost ostalih promenljivih koje predstavljaju (pokazuju na) isti podatak. Na primer, na slici 2.1.b promenljive `b` i `c` predstavljaju isti niz brojeva. Ako se promeni vrednost broja b_2, promeniće se i vrednost broja c_2.

U jeziku *Java* prosti tipovi su vrednosni, a složeni tipovi pokazani tipovi podataka.

U nastavku ovog poglavlja obrađeni su prosti tipovi podataka u jeziku *Java*.

2.4 Brojčani tipovi

Brojčani tipovi (*numeric types)* podataka jesu prosti tipovi za koje su definisane aritmetičke operacije. Među njima posebno se razlikuju celobrojni i realni tipovi.

3 Operatori

Operatori predstavljaju radnje koje se izvršavaju nad operandima dajući pri tome određeni rezultat. ***Izrazi*** (*expressions*) su proizvoljno složeni sastavi operanada i operatora. Opšti oblik izraza u jeziku *Java* je:

```
konstanta                                                    ili
promenljiva                                                  ili
operator izraz                                               ili
izraz operator                                               ili
izraz operator izraz                                         ili
izraz ? izraz : izraz                                        ili
( izraz )
```

Najjednostavniji izraz se sastoji od jedne `konstante` ili od jedne `promenljive`.

Složeniji izrazi se dobijaju primenom `operatora` na operande. Pošto operandi mogu da budu rezultati ranijih izračunavanja u gornjem opštem obliku su obeleženi sa `izraz`.

Operatori mogu da se primenjuju na jedan operand (unarni operatori) ili na dva operanda (binarni operatori). U jeziku *Java* postoji i jedan operator s tri operanda (ternaran operator).

Unarni operatori mogu da stoje ispred operanda (prefiksni operatori) ili iza operanda (postfiksni operatori).

Binarni operatori uvek stoje između svoja dva operanda (infiksni operatori).

Upotreba **ternarnog operatora** `?:` označava se stavljanjem znaka pitanja između prvog i drugog operanda i dve tačke između drugog i trećeg operanda.

Na kraju, izraz po potrebi i po želji može da se stavlja unutar para oblih zagrada `()`. Njihova uloga je da izdvoje deo složenog izraza kao neku manju celinu i da time utiču na redosled izračunavanja operatora.

Rezultat izračunavanja može biti:

- promenljiva – nešto što je u memoriji (*lvrednost* u jezicima *C/C++*),
- vrednost,
- ništa – označava se sa **`void`** (može biti samo rezultat metoda koje su tako obeležene – §6.4.1.1, 🗎103).

Postoje operatori za koje neki od operanada mora da bude promenljiva. Ako pri objašnjavanju pojedinih operatora nije posebno naglašeno, operandi ne moraju da budu promenljive.

Bočni efekat (*side effect*) predstavlja promenu vrednosti nekog od operanada operatora u toku izvršavanja operatora. Rezultat izračunavanja je *vrednost izraza* i može da se koristi kao operand u daljim izračunavanjima. Ako se to ne radi, vrednost izraza se odbacuje. Trajni rezultati mogu da se dobiju samo kao bočni efekti operatora. Za operatore koji proizvode bočne efekte posebno je to naglašeno prilikom objašnjavanja tih operatora.

Redosled izvršavanja operatora prvenstveno se određuje oblim zagradama (i). Unutar para oblih zagrada mogu da budu ugnežđeni drugi parovi zagrada do proizvoljne dubine. Operatori unutar para zagrada izvršavaju se pre operatora izvan tih zagrada.

Operatori unutar zagrada izvršavaju se po redosledu ***prioriteta*** (*priority*) ili ***prvenstva*** (*precedence*) operatora. Od dva susedna operatora, operator s višim prioritetom izvršava se pre operatora s nižim prioritetom. U jeziku *Java* postoje ukupno 16 nivoa prioriteta. Najviši prioritet ima nivo 16, a najniži nivo 1.

U slučaju niza operatora međusobno jednakih prioriteta, operatori će biti izvršavani sleva nadesno ili zdesna nalevo, u zavisnosti od njihovog ***smera grupisanja***. Na datom nivou prioriteta svi operatori se grupišu u istom smeru.

Kod binarnih operatora uvek se prvo izračunava levi i tek posle toga desni operand, bez obzira na smer grupisanja. Ovo pravilo je važno u slučajevima kada je jedan od operanada izraz koji pravi i bočne efekte.

U jeziku *Java* postoje sledeće vrste operatora (po opadajućem redosledu prioriteta):

- aritmetički operatori,
- operatori sa niskama,
- relacijski operatori,
- operatori za rad s bitovima,
- logički operatori,
- operatori za dodelu vrednosti.

3.1 Aritmetički operatori

Aritmetički operatori imaju brojčane operande i daju brojčane rezultate. Služe za obavljanje uobičajenih aritmetičkih operacija.

Binarni operatori +, −, *, / i % izračunavaju zbir, razliku, proizvod, količnik i ostatak posle deljenja dva broja, respektivno.

U slučaju deljenja dva cela broja razlomljeni deo se prosto odbacuje, tj. ne vrši se zaokruživanje. Ovakva operacija naziva se i *celobrojno deljenje*.

Tipovi operanada operatora % moraju da budu jedan od celobrojnih tipova, a predznak rezultata je jednak predznaku prvog operanda.

Operatori + i − imaju prioritet 11, a operatori *, / i % prioritet 12. Svi ovi operatori se grupišu sleva nadesno.

Primer 3.1 – *Izrazi s binarnim aritmetičkim operatorima*

```
a + b - c              // (a + b) - c
a + b * c              // a + (b * c)
a / b * c              // (a / b) * c
7 / 4 * 3              // 3                                   ali …
7 * 3 / 4              // 5                          za razliku od …
7. / 4. * 3.           // 5.25                              i od …
7. * 3. / 4.           // 5.25
7 % 3                  // 1
9 % 3                  // 0
```

*Prvi izraz prikazuje izraz s dva operatora istog prioriteta. Pošto se ovi operatori grupišu sleva nadesno, prvo se izvršava operator +. Rezultat tog izračunavanja predstavlja prvi operand operatora −. U drugom izrazu operator * ima viši prioritet, pa se prvo izvršava taj operator. U trećem izrazu, opet, oba operatora su istog prioriteta, pa se oni izvršavaju sleva nadesno.*

Četvrti i peti izraz pokazuju da u slučaju celobrojnih operanada nije sve jedno po kom se redosledu pišu operatori i operandi ako među njima ima i operatora za deljenje. Dva izraza, koja bi matematički gledano trebalo da daju isti rezultat, daju različite vrednosti. To nije slučaj kod računanja s realnim podacima, kao što to pokazuju šesti i sedmi izraz.

Osmi i deveti izraz prikazuju nalaženje ostatka deljenja prvog celobrojnog operanda s drugim celobrojnim operandom. Prvi operand je deljiv s drugim ako je ostatak nula. ■

Unarni operatori su +, −, ++ i −−.

Prefiksan unaran operator + daje kao rezultat vrednost svog operanda bez izmene, a prefiksan unaran operator − vrednost svog operanda s izmenjenim predznakom. Ovi operatori imaju prioritet 13 i grupišu se zdesna nalevo.

Operator ++ povećava, a −− smanjuje vrednost svog operanda za 1. Oni su jedini unarni operatori koji mogu da budu prefiksni (`++k`, `--k`) ili postfiksni (`k++`, `k--`).

Vrednost izraza `++k` i `--k` je nova vrednost promenljive `k`. To znači da u slučaju kada se koristi kao operand u nekom izrazu, prvo se izmeni vrednost promenljive `k`, i dalje se računa s tom novom vrednošću.

Vrednost izraza `k++` i `k--` je stara vrednost promenljive `k`. To znači da u slučaju kada se koristi kao operand u nekom izrazu, prvo se uzima vrednost promenljive `k` za dalje računanje a tek posle se menja vrednost promenljive `k`.

Operandi ovih operatora mogu da budu samo promenljive, a ne i rezultati drugih operatora. Osim u retkim slučajevima kada je rezultat tog drugog operatora promenljiva.

Prefiksne varijante ovih operatora su prioriteta 13 i grupišu se sleva nadesno. Postfiksne varijante su prioriteta 14 i grupišu se zdesna nalevo. Smerovi grupisanja nemaju upotrebnu vrednost, jer ne mogu dva ovakva operatora da se napišu jedan za drugim. Naime, rezultat prvog operatora nije promenljiva, pa ne može biti operand drugog operatora.

Primer 3.2 – *Izrazi s operatorima ++ i --*

```
++k                    // k = k + 1
k++                    // k = k + 1
j = --k                // k = k - 1, j = k
j = k--                // j = k, k = k - 1
j = ++k * k            // k = k + 1, j = k * k                    ali …
j = k++ * k            // u = k, k = k + 1, j = u * k
k++ ++                 // (k++)++ →  GREŠKA!
```

Prva dva izraza pokazuju da su u slučaju samostalnog izraza prefiksna i postfiksna varijanta operatora ++ istovetne. Isto važi, naravno, i za operator --.

Pod pretpostavkom da je vrednost promenljive `k` jednaka 8, nova vrednost promenljive `j` u trećem izrazu je 7, a u četvrtom 8. Nova vrednost promenljive `k` u oba slučaja je 7. Ovi izrazi pokazuju i jednu zanimljivu osobinu jezika Java*, mogućnost dobijanja više rezultata pomoću jednog izraza.*

Zbog pravila da se prvo izračuna levi operand binarnog operatora, pod pretpostavkom da je vrednost promenljive `k` jednaka 8, nova vrednost promenljive `j` u petom izrazu je 81, a u šestom 72. Nova vrednost promenljive `k` u oba slučaja je 9.

Poslednji izraz je neispravan jer rezultat izraza `k++` nije promenljiva, pa ne može na to još jednom primeniti operator ++ ■

4 Naredbe

Naredba (*statement*) je osnovna jedinica obrade u programima. Postoje proste, složene i upravljačke naredbe.

Proste naredbe (*simple statements*) predstavljaju elementarne obrade koje ne mogu da se podele na manje delove koji bi i sami bili naredbe.

Složene naredbe (*compound statements*) predstavljaju strukture naredbi kojima se određuje redosled izvršavanja naredbi sadržanih u strukturi. Te strukture naredbi nazivaju se *upravljačke strukture*. One mogu da se podele u sledeće tri grupe:

- sekvenca,
- selekcije, i
- ciklusi ili petlje.

Upravljačke naredbe (*control statements*) ne vrše nikakvu obradu već samo prenose tok upravljanja na neko mesto u programu. Ovoj grupi pripadaju razne naredbe skokova.

Naredbe unutar složenih naredbi i same mogu da budu proste ili složene. Na taj način mogu da se ostvaruju vrlo složene obrade. U nastavku knjige, ako se drugačije ne kaže, pod pojmom *naredba* podrazumevaće se bilo koja vrsta naredbi: prosta, složena ili upravljačka. Ove naredbe spadaju u grupu izvršnih naredbi. Ponekad, u širem smislu obuhvataće se i deklarativne naredbe kao što su naredbe za definisanje podataka (§2.8, 🗎20), tipova podataka (poglavlje 6 o klasama) itd.

4.1 Prosta naredba

Prosta naredba predstavlja elementarnu obradu u programu. Njen opšti oblik je:

```
izraz ;
```

Izvršavanje proste naredbe sastoji se u izračunavanju izraza. Dobijena vrednost izraza ne koristi se ni za šta. Trajni rezultati naredbe dobijaju se kao bočni efekti pojedinih operatora unutar izraza. Operatori s bočnim efektima jesu operatori za dodelu vrednosti (§3.2, 🗎36) i operatori ++ i −− (§3.1, 🗎35).

Specijalnu, i ređe korišćenu, naredbu čini takozvana ***prazna naredba*** (*empty statement*). Ona se sastoji samo od tačke-zareza (;). Prazna naredba nema nikakvog efekta. Koristi se u slučajevima kada sintaksa jezika *Java* zahteva naredbu, a desi se da na tom mestu nema šta da se radi.

4.2 Sekvenca ili blok: { }

Sekvenca ili blok je niz naredbi koje se izvršavaju jedna za drugom.

Sekvenca u jeziku *Java* piše se u obliku niza naredbi unutar vitičastih zagrada ({}).:

```
{ naredba1 naredba2 … naredbaN }
```

Pojedine *naredbe* mogu biti bilo deklarativne za definisanje podataka (§2.8, 🗎20), bilo izvršne iz ovog poglavlja. U jeziku *Java* nema ograničenja u redosledu korišćenja deklarativnih i izvršnih naredbi. Zbog toga podatke treba definisati što bliže mestu prvog korišćenja i tada, obično, nema razloga da se ne navedu i inicijalizatori za postavljanje početnih vrednosti.

Kao što postoji prazna naredba, tako postoji i prazna sekvenca, koja ne sadrži nijednu naredbu.

Programski jezik *Java* ne postavlja nikakve uslove u vezi podele naredbi u redove. U jednom redu može da bude više naredbi i jedna naredba može da se piše u proizvoljnom broju redova. Osnovni kriterijum treba da je preglednost teksta programa.

Ukoliko treba da se koristi više redova, preporučuje se da se vitičaste zagrade ({}) izdvoje u zasebne redove, a sadržaj sekvence da se uvlači za nekoliko mesta u odnosu na vitičaste zagrade. Na taj način se jasno ističe gde su početak i kraj sekvence:

```
{
  naredba1
  naredba2
  …
  naredbaN
}
```

Primer 4.1 – *Sekvence*

```
{ p = a; a = b; b = p; }
```

```
{
  System.out.prin("Re,Im= ");
  int a = Citaj.Double(); int b = Citaj.Double();
  int c = Math.sqrt(a*a + b*b); int d = Math.atan2(b, a);
  System.out.println("Abs=" + c + "\tArg=%" + d);
}
```

Prvi primer je jedna jednostavna sekvenca kojom se međusobno zamene vrednosti promenljivih `a` *i* `b` *korišćenjem pomoćne promenljive* `p`*. Treba posebno obratiti pažnju na to da je znak* `;` *ispred zagrade* `}` *neophodan. Naime,* `b=p` *je samo izraz i tek je* `b=p;` *naredba koja može da bude deo sekvence.*

Drugi primer predstavlja jednu malo složeniju sekvencu. Ona, posle poruke korisniku, čita dva realna broja dvostruke tačnosti, koji predstavljaju realni i imaginarni deo kompleksnog broja. Zatim izračunava apsolutnu vrednost i argument tog kompleksnog broja i ispisuje rezultat. Potrebne promenljive se definišu deklarativnim naredbama između dve izvršne proste naredbe za pisanje na glavnom izlazu. ■

4.3 Doseg identifikatora

Doseg (oblast važenja – *scope*) identifikatora u programiranju označava deo programa u kome identifikator može da se koristi.

Doseg identifikatora u jeziku *Java* proteže se od mesta deklarativne naredbe kojom je uveden do kraja bloka u kome se nalazi ta naredba. Zbog toga se ovakav doseg naziva **blokovski doseg** (*block scope*). Kasnije će se videti da postoje i druge vrste dosega identifikatora.

Treba obratiti pažnju na to da se doseg identifikatora ne proteže na ceo blok, već počinje tek od mesta definisanja identifikatora.

Primer 4.2 – *Definisanje identifikatora u bloku*

```
{
  int x = 5;
  double y = x / 2.;
  System.out.println("x=" + x + ", y=" + y);
}
```

Na početku bloka definišu se dve promenljive i posle se ispisuju njihove vrednosti. Vrednosti promenljivih zadate su kao inicijalizatori prilikom definisanja. Treba obratiti pažnju na to da se početna vrednost promenljive `y` *izračunava na osnovu vrednosti promenljive* `x`*, koja je definisana u redu neposredno pre naredbe kojom je definisana promenljiva* `y`*.* ■

Pojedine naredbe unutar bloka (sekvence) i same mogu biti blokovi. Time se u doseg koji je određen spoljašnjim blokom ugnežđuje doseg unutrašnjeg bloka. Pošto se ugnežđeni blok (*nested block*) nalazi unutar dosega spoljašnjeg bloka, unutar ugnežđenog dosega mogu da se koriste i identifikatori koji su u spoljašnjem bloku definisani pre ugnežđenog bloka. Kaže se da su identifikatori spoljašnjeg bloka ***globalni*** za unutrašnji blok.

Obrnuto, identifikatori iz ugnežđenog bloka ne mogu da se koriste unutar spoljašnjeg bloka, ni ispred ni iza unutrašnjeg bloka. Ti identifikatori postoje samo unutar ugnežđenog bloka. Zato se kaže da su identifikatori ugnežđenog bloka ***lokalni*** za taj blok.

U slučaju višestrukog ugnežđivanja, unutar najdubljeg bloka mogu da se koriste identifikatori iz svih okružujućih blokova. Unutar bloka na srednjem nivou ugnežđivanja mogu da se koriste identifikatori iz svih okružujućih blokova ali ne i identifikatori iz u njega ugnežđenih blokova. Naravno, mogu da se koriste samo oni identifikatori koji su definisani pre ulaska u određeni ugnežđeni blok.

Pošto je doseg unežđenog bloka sastavni deo okružujućeg dosega, u ugnežđenom bloku ne sme da se definiše identifikator koji postoji u spoljašnjem bloku, tj. koji već postoji u dosegu. Po napuštanju unutrašnjeg bloka dozvoljeno je definisanje identifikatora koji postoji u unutrašnjem bloku, pošto taj identifikator više nije u dosegu.

Skreće se pažnja na to da je telo (sadržaj) metode `main`, u stvari, blok i unutar nje mogu da postoje ugnežđeni blokovi.

Primer 4.3 – *Ugnežđivanje blokova*

Na početku spoljašnjeg bloka na slici 4.1 definišu se dve promenljive (`a` *i* `b`*).*

U prvom ugnežđenom bloku mogu da se koriste promenljive koje su definisane u spoljašnjem bloku pre njega (`a` *i* `b`*), ali ne sme da se koriste promenljive koje su definisane iza njega (*`h`*,* `i` *i* `j`*). Ne sme da se definiše promenljiva s identifikatorom* `b`*, pošto je taj identifi-*

```
{ // Spoljašnji blok.
  int a = 1, b = 2;
  { // Prvi ugnežđeni blok.
    int c = a + b;
    int d = h + i * j;      // GREŠKA: h još ne postoji.
    int b = c / d;          // GREŠKA: b već postoji.
    int e = f;              // GREŠKA: f još ne postoji.
    { // Drugi nivo ugežđivanja.
      int f = a * b / (c - d + e);
    }
    int g = f;              // GREŠKA: f više ne postoji.
    int h = c + d;
  }
  int h = a / b;            // U redu: prethodno h više ne postoji.
  int i = c + f;            // GREŠKA: c i f više ne postoje.
  { // Drugi unežđeni blok.
    double c = 1.234;
    boolean d = true;
    String e = "Dobar dan!";
    byte f = 55;
    float g = 35.8F;
    char h = '?';           //GREŠKA: h već postoji.
  }
  int j = (a + b) * (h - i);
}
```

Slika 4.1 *– Ugnežđivanje blokova*

kator uveden doseg u spoljašnjem bloku. Korišćenje promenljive `f` *nije dozvoljeno ni pre ni posle bloka na drugom nivou ugnežđivanja. Pre još, a posle već ne postoji.*

U bloku na drugom nivou ugnežđivanja mogu da se koriste svi identifikatori (promenljive) kako iz prvog ugnježđenog bloka (`c`*,* `d` *i* `e`*), tako i iz spoljašnjeg bloka (*`a` *i* `b`*).*

Iza prvog ugnežđenog bloka, u spoljašnjem bloku, može da se definiše promenljiva `h`*, bez obzira što je taj identifikator korišćen u ugnežđenom bloku. Napuštanjem ugnežđenog bloka napušten je i doseg prve definicije identifikator* `h`*. S druge strane, iz istog razloga, ne mogu se koristiti kao operandi promenljive* `c` *i* `f`*.*

Drugi ugnežđeni blok je potpuno nezavisan od prvog. Zbog toga identifikatori `c`*,* `d`*,* `e` *i* `f`*, koji su korišćeni u prvom bloku, mogu ponovo da se koriste. Tipovi promenljivih kojima se dodeljuju su nezavisni od tipova istoimenih promenljivih iz prvog bloka. Jedino identifikator* `h` *iz prvog bloka ne sme da se ponovo definiše u drugom bloku, jer je između ta dva bloka taj identifikator definisan u spoljašnjem bloku. Definicija iz spoljašnjeg bloka je u dosegu i u drugom ugnežđenom bloku.*

Iza drugog ugnežđenog bloka, u spoljašnjem bloku, kao operandi mogu da se koriste samo promenljive definisane ranije u tom bloku (`a`*,* `b`*,* `h` *i* `i`*), a ne i promenljive definisane unutar bilo kog ugnežđenog bloka.* ■

5 Nizovi

Nizovi podataka predstavljaju najjednostavnije i najčešće korišćene složene tipove. Oni služe za predstavljanje iz matematike dobro poznatih vektora, matrica i višedimenzionalnih nizova.

Podatak nizovnog tipa predstavlja niz podataka međusobno jednakih tipova. Oni se nazivaju *elementi niza* i identifikuju se pomoću rednog broja unutar niza. Ti redni brojevi nazivaju se ***indeksi*** elemenata niza. Prvi element niza u jeziku *Java* obavezno ima indeks 0, drugi 1 itd.

Elementi nizova mogu biti prostog i složenog tipa, uključujući i nizove. U slučaju kada su elementi nizovi, govori se o ugnežđavanju nizova ili višedimenzionalnim nizovima.

Niz je jednodimenzionalan (vektor) ako elementi nisu nizovi. Dvodimenzionalan niz (matrica) je niz čiji su elementi jednodimenzionalni nizovi. Niz ima dimenziju k ako su njegovi elementi nizovi sa $k-1$ dimenzija.

Nizovi kao složeni tipovi podataka u jeziku *Java* su pokazani tipovi (§2.3, 🗎16). To znači da je promenljiva nizovnog tipa samo pokazivač na sâm niz. Elementi niza nalaze se negde drugde u memoriji.

5.1 Definisanje nizova

Nizovne promenljive definišu se naredbama za definisanje podataka iz §2.8 (🗎20), s tim da kao tip treba navesti:

```
tip [ ] [ ] … [ ]
```

Tip predstavlja tip elemenata niza. Broj praznih parova uglastih zagrada (`[]`) označava nivo ugnežđavanja, odnosno broj dimenzija niza.

Jednom naredbom za definisanje podataka, kao i kod prostih tipova u §2.8 (🗎20), može da se definiše i više nizovnih promenljivih.

Primer 5.1 – *Definisanje nizovnih promenljivih*

```
int[] vektor;
double[][] matrica;
boolean[][][] p, q, r;
```

■

Prostor za elemente niza u jeziku *Java* može da se dodeljuje isključivo dinamički u vreme izvršavanja programa. Takvi podaci nazivaju se *dinamički podaci*, a deo memorije gde se oni uskladištavaju, *dinamička zona memorije*.

Opšti oblik izraza za dodelu memorije nizu je:

```
new tip [ dužina ] [ dužina ] … [ dužina ]
```

Operator za dodelu memorije **new** je prefiksan unaran operator najvišeg prioriteta (16) čiji je „operand" opis niza kome se dodeljuje memorija. Formalno se grupiše sleva nadesno,ali to nema značaja pošto ne mogu da se napišu dva operatora **new** jedan za drugim.

Tip predstavlja tip elemenata niza.

Dužina predstavlja broj elemenata niza po jednoj dimenziji i može biti proizvoljan celobrojan izraz (tipa **int**). U slučaju ugnežđenih (višedimenzionalnih) nizova, *dužine* za svaki nivo se piše unutar odvojenog para uglastih zagrada.

Tip rezultata operatora **new** je (pokazivač na) niz čija je dimenzionalnost jednaka broju parova uglastih zagrada (`[]`). Vrednost rezultata može da se dodeljuje ranije definisanoj nizovnoj promenljivoj (§5.5, 🗎79) ili kao inicijalizator pri definisanju nove promenljive:

```
tip [ ] [ ] … [ ] niz = new tip [ dužina ] [ dužina ] … [ dužina ]
```

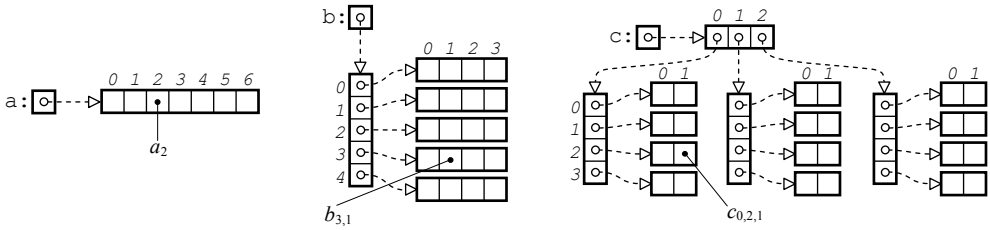
Broj parova uglastih zagrada u tipu promenljive i u izrazu za dodelu memorije mora biti isti. *Niz* predstavlja identifikator nizovne promenljive. Kao rezultat dodele memorije, izgradiće se složene stukture podataka, zavisno od nivoa ugnežđivanja (broja dimenzija) niza.

Ako dodela memorije ne uspe, greška se prijavljuje izuzetkom tipa `OutOfMemoryError`. Ako se taj izuzetak ne obrađuje od strane programa, program se prekida (izuzeci su objašnjeni u poglavlju 10).

Primer 5.2 *– Definisanje nizova*

```
int[] a = new int[7];
float[][] b = new float[5][4];
char[][][] c = new char[3][4][2];
```

Slika 5.1 prikazuje način uskladištavanja prethodnih nizova.



a) *vektora* ***b)*** *matrice* ***c)*** *trodimenzionalnog niza*

Slika 5.1 *– Usklasištavanje nizova*

Promenljiva `a` *pokazuje na niz od 7 elemenata tipa* **int** *koji se nalaze u dinamičkoj zoni memorije. Prvi element je* a_0*, a poslednji* a_7*.*

Pošto dvodimenzionalan niz, po definiciji iz §5 (🗎75), jeste niz jednodimenzionalnih nizova, promenljiva `b` *pokazuje na niz od 5 nizova, tj. na niz od 5 elemenata čiji svaki element pokazuje na niz od 10 elemenata tipa* **float***. Prvi element je* $b_{0,0}$*, a poslednji* $b_{4,3}$*.*

Na kraju, promenljiva c pokazuje na niz od 3 (pokazivača na) podmatrice dimenzije 4×2 elementa. Prvi element je $c_{0,0,0}$*, a poslednji* $c_{2,3,1}$. ■

U jeziku *Java* zanemaruje se (s nekoliko izuzetaka) pokazivačka priroda nizovnih promenljivih i govori se o definisanju nizova i da je definisana promenljiva niz određene dimenzije sastavljen od elemenata određenog tipa. Na primer, za nizove u primeru 5.2 kaže se da „`a` je jednodimenzionalan niz celih brojeva", odnosno „`b` je dvodimenzionalan niz realnih brojeva dvostruke tačnosti". Ili drugačije: „`a` je niz celih brojeva", odnosno „`b` je niz nizova realnih brojeva dvostruke tačnosti".

Jedna od situacija kada pokazivačka priroda nizovnih promenljivih dolazi do izražaja jeste mogućnost da nizovna promenljiva ne pokazuje ni na šta. To se predstavlja imenovanom konstantom **`null`**. Preporučuje se da se nizovna promenljiva nikad ne definiše bez inicijalizatora. Ako se u momentu definicije još ne zna broj elemenata niza, inicijalizator neka bude **`null`**.

Treba razlikovati **`null`** od praznog niza, tj. niza dužine nula.

Primer 5.3 *– Nula niz i prazan niz*

```
int[] c = null;
int[] d = new int[0];
```

Dok promenljiva `c` *ne pokazuje ni na šta, promenljiva* `d` *pokazuje na niz od nula elemenata. Prazan niz može da se shvati kao granični slučaj, analogno praznom skupu koji ne sadrži nijedan element.* ■

5.2 Inicijalizacija nizova

Svaki bajt memorijskog prostora koji se zauzme operatorom **`new`** popunjava se nulama. Zbog toga je početna vrednost elemenata nizova nula odgovarajućeg tipa (`0`, `0.0`, `'\u0000'`, **`false`**). Za složene tipove kao što su nizovi, niske (§2.7, 🗎20) i klase (§6, 🗎99), to je **`null`**.

Nenulte početne vrednosti (inicijalizator) mogu da se navedu u nastavku izraza za dodelu memorije ili prilikom definisanja nizovne promenljive. Opšti oblik inicijalizatora je:

```
{ vrednost , vrednost , … , vrednost }
```

Pojedinačne *`vrednosti`* mogu biti proizvoljni izrazi tipa elemenata niza i izračunavaju se redom sleva nadesno. Po potrebi, na pojedine vrednosti primenjuje se automatska, proširujuća, konverzija tipa (§3.5, 🗎40). Dužina niza biće jednaka broju navedenih *`vrednosti`*. U slučaju ugnežđavanja nizova pojedinačne *`vrednosti`* treba da su i same nizovi vrednosti odgovarajuće dubine ugnežđavanja.

Ako se u izrazu za dodelu memorije navodi inicijalizator, parovi uglastih zagrada moraju biti prazni (ne smeju da se navedu dužine).

Primer 5.4 *– Inicijalizacija nizova*

```
int[] e = new int[] {1, 2, 3, 4};                    // ili …
int[] f = {1, 2, 3, 4};
int[][] g = new int[][] {{1,2,3}, {4,5,6}};          // ili …
int[][] h = {{1,2,3}, {4,5,6}};
```

■

5.3 Pristupanje elementima niza: `[]`

Elementi niza se identifikuju pomoću rednog broja unutar niza koji se naziva *indeks*. Indeksi u jeziku *Java* su celi brojevi od 0 do $n-1$, gde je n broj elemenata (tj. dužina) niza.

Pristup elementima niza naziva se *indeksiranje*.

Indeksiranje u jeziku *Java* smatra se binarnim operatorom i obeležava se sa `[]`. Za razliku od ostalih binarnih operatora, ovaj operator se ne piše između svojih operanada već „oko" drugog operanda. Taj drugi operand je indeks traženog elementa, a prvi operand je niz unutar kojeg se indeksira:

```
niz [ indeks ]
```

Niz ne sme da bude **`null`**. Ako jeste, greška se prijavljuje izuzetkom tipa `NullPointerException`. Ako se izuzetak ne obrađuje od strane programa, program se prekida (izuzeci su objašnjeni u poglavlju 10).

Indeks može da bude proizvoljan celobrojan izraz čija vrednost treba da je >=0 i <*dužina*. Ako je vrednost indeksa izvan dozvoljenog opsega, greška se prijavljuje izuzetkom tipa `ArrayIndexOutOfBoundsException`. Ako se izuzetak ne obrađuje od strane programa, program se prekida (izuzeci su objašnjeni u poglavlju 10)

Operator za indeksiranje `[]` je prioriteta 15 i grupiše se sleva nadesno.

Rezultat indeksiranja je vrednost elementa niza sa zadatim indeksom. Tip rezultata je tip elemenata niza.

U slučaju ugnežđenih nizova rezultat indeksiranja je (pokazivač na) odgovarajući komponentni niz. Na tako dobijeni rezultat može ponovo da se primeni indeksiranje, sve dok se ne dođe do elementa koji nije nizovnog tipa. Odavde sledi da *niz* u gornjem opštem obliku može da bude identifikator niza ili rezultat operatora `[]` koji je niz.

Jednostavnije rečeno, nenizovnom elementu višedimenzionalnog niza može da se pristupa navođenjem po jednog indeksnog izraza za svaku dimenziju (nivo ugnežđavanja) unutar odvojenih parova zagrada:

```
niz [ indeks ] [ indeks ] … [ indeks ]
```

Samo na nenizovne elemente nizova mogu da se primene svi operatori iz poglavlja 3.

Primer 5.5 – *Pristup elementima nizova iz primera 5.2*

```
a[1] = 155;
a[i+(j-1)*n] = x + y * z;
b[i][j] = vektor[i+j];
```

■

5.4 Dohvatanje dužine niza: `length`

Broj elemenata (dužina) niza može da se dobija izrazom tipa **`int`** oblika:

```
niz . length
```

U slučaju ugnežđenih nizova rezultat je broj neposredno ugnežđenih komponentnih nizova. Za svaki od njih pojedinačno može da se traži njihov broj elemenata.

6 Klase

Klase (*classes*) u jeziku *Java* složeni su tipovi podataka koji se sastoje od elemenata koji mogu da budu međusobno različitih tipova. Ti elementi nazivaju se **članovi klasa** (*class members*).

Podaci klasnih tipova predstavljaju **primerke** date klase (*class instances*) i nazivaju se **objekti** (*objects*). Termin klasa često se koristi, skraćeno, za označavanje pojma primerak klase. Iz konteksta treba tada zaključiti, da li reč klasa označava tip ili objekat.

Klase kao složeni tipovi podataka u jeziku *Java* su pokazani tipovi (§2.3, 🗎16). To znači da je promenljiva klasnog tipa samo pokazivač na objekat. Sâm objekat nalazi se negde drugde u memoriji.

Članovi klasa mogu da budu polja ili metode. **Polja** (*fields*) klase, odnosno objekta su podaci koji čine **stanje** (*state*) objekta. **Metode** (*methods*) klase su funkcije koje mogu da izvode razne operacije nad poljima čime menjaju stanje (vrednost) objekta na koji se primenjuju.

Članovi klasa mogu da budu privatni ili javni. **Privatnim** (*private*) članovima može da se pristupa samo iz unutrašnjosti posmatrane klase. To znači da privatna polja mogu da koriste ili menjaju samo metode date klase, a privatne metode mogu da pozivaju samo druge metode iste klase. **Javnim** (*public*) članovima može da se pristupa bez ograničenja kako iz unutrašnjosti klase, tako i iz delova programa izvan posmatrane klase. Kaže se da ove osobine označavaju **prava pristupanja** (*access rights*) ili **dostupnost** članova.

U većini slučajeva polja su privatna, dok su od metoda neke privatne a neke javne. Javne metode su tada jedina mogućnost da se dođe do podataka sadržanih u klasnim objektima. Time se postiže učaurivanje podataka koje je pomenuto u §1.2.2 (🗎4).

Klase su pravi tipovi jer:

- određuju moguće vrednosti objekata,
- određuju moguće operacije nad objektima,
- mogu da spreče izvršavanje bilo koje druge operacije nad objektima,
- obezbeđuju inicijalizaciju (dodelu početnih vrednosti) stvaranih objekata,

6.1 Definisanje klasa

Definicija klase predstavlja navođenje svih članova klase. Na osnovu te definicije mora da se zna veličina potrebnog memorijskog prostora za smeštanje pojedinih objekata tipa te klase. Klasa se definiše opisom **class** čiji je opšti oblik:

```
class ImeKlase {
  modifikatori član
  modifikatori član
  …
}
```

Modifikatori određuju neke specifične osobine člana klase. U slučaju više modifikatora redosled nije bitan.

Modifikator **private** ili **public** označava da je član privatan, odnosno javan. Privatni članovi mogu da se koriste samo unutar date klase, a javni iz bilo kog dela programa.

ImeKlase je po formi identifikator i ima status identifikatora tipa. Može da se koristi u naredbama za definisanje podataka za podatke (objekte) tipa te klase. Uobičajeno je da se imena klasa pišu velikim početnim slovom.

Član u definiciji klase može da bude:

- definicija polja (§6.3, 🗎101),
- definicija metode (§6.4, 🗎102), i
- definicija druge klase (§9, 🗎209).

Pored toga u definiciji klase mogu biti:

- inicijalizacioni blokovi (§6.8, 🗎118), i
- definicije konstruktora (§6.5, 🗎111).

Oni se ne ubrajaju u članove klase zbog specifične namene i načina korišćenja.

Doseg svih identifikatora unutar klase je od mesta definisanja do kraja klase. Kaže se da članovi klase imaju **klasni doseg** (*class scope*).

6.2 Pristupanje članovima klasa

Članu klase pristupa se izrazom oblika:

```
objekat . član
```

Tačka (**.**) je binaran operator za pristup članu klase prioriteta 15 koji se grupiše sleva nadesno.

Objekat je najčešće identifikator objekta čijem članu se pristupa, ali može biti i rezultat složenijeg izraza čiji je rezultat (pokazivač na) objekat.

Član može da predstavlja polje ili metodu. Za polje treba navesti ime polja (§6.3, 🗎101). Pristupanje metodi predstavlja pozivanje te metode i treba navesti izraz za pozivanje (§6.4.2, 🗎106).

Unutar klase objekat kojem se pristupa može da se predstavlja ključnom reči **this**. Time se označava da se pristupa članu objekta koji se upravo obrađuje (tekućeg objekta):

```
this . član
```

Za pristup članu spostvene klase dovoljno je i navođenje samo člana:

```
člana
```

tj. **this.** se podrazumeva. Ponekad je, ipak, zgodno da se napiše to neobavezno **this.**, naročito ako se u metodi, pored tekućeg, koristi još i veći broj objekata.

6.3 Polja klasa

Polja klasa su podatkovni članovi čija definicija ima isti oblik kao i naredba za definisanje samostalnih podataka u §2.8 (🗎20):

```
imeTipa imePolja = početnaVredost , imePolja , ... ;
```

Odjednom mogu da se definišu više polja zajedničkog tipa. *ImeTipa* može biti bilo koji prost ili složen tip, uključujući i klasu koja se upravo definiše.

Uz *imenaPolja* mogu da se navedu i *početneVrednosti* (**inicijalizatori**). Za razliku od samostalnih promenljivih, polja imaju nulte početne vrednosti u odsustvu inicijalizatora.

U izrazima za *početneVrednosti* kao operandi mogu da se koriste i polja koja su već definisana po redosledu navođenja, a ne i polja koja će biti definisana tek kasnije u tekstu klase.

Polja mogu biti **konačna**, ako se na početku definicije navodi modifikator **final**. Vrednosti konačnih polja mogu da se postave samo u toku stvaranja objekata (§6.8.4, 🗎120), ne obavezno navođenjem *početneVrednosti*, već i u inicijalizacionom bloku (§6.8.3, 🗎120) ili u konstruktoru (§6.5, 🗎111).

Polja su skoro bez izuzetka privatna da bi se postiglo učaurivanje podataka (§1.2.2, 🗎4), tj. zaštita podataka od neprimerenog korišćenja.

Primer 6.1 – *Definisanje polja klasa*

```
class Alfa {
  private int a = 1, b;            // b == 0
  public double c = a + b;
  public double d = c * j;         // GREŠKA: k još ne postoji.
  private short[] e = {1, 2, 3, 4};
  private byte[] f = new byte[55];
  private char[][] g;              // g == null
  private String h = "Dobar dan.";
  private Alfa i;                  // i == null
  public final int j = 55;
  public final double k;
  …
}
class AlfaTest {
  public static void main(String[] varg) {
    Alfa a = new Alfa();       // Stvaranje objekta (§6.6.1).
    a.c = 35;
    a.b = 56;                  // GREŠKA: b nije dostupno.
    int x = a.j;
    a.k = 13;                  // GREŠKA: k je konačno polje.
  }
}
```

Na početku klase `Alfa` *definisana su privatna celobrojna polja* `a` *i* `b`*. Polje* `a` *se inicijalizatorom postavlja na vrednost 1, dok* `b` *poprima podrazumevanu vrednost 0.*

Javna realna polja `c` *i* `d` *se inicijalizuju vrednostima izraza u kojima se kao operandi koriste druga polja klase* `Alfa`*. Izraz* `a+b` *je prihvatljiv, pošto su oba operanda već definisana polja. Izraz* `c*j` *je neispravan jer je polje* `j` *definisano je tek kasnije.*

Javna polja se retko koriste u praksi, pa su sva preostala polja klase `Alfa` *privatna.*

Polja `e`*,* `f` *i* `g` *su nizovnog tipa. Vektor e se inicijalizuje na osnovu navedenih vrednosti. Vektor f se inicijalizuje, izrazom za dodelu memorije, nizom od 55 elemenata čije vrednosti su podrazumevano nule (§5.2, 77). Matrica g nema inicijalizator, pa to polje poprima podrazumevanu vrednost* **`null`**.

Polja `h` *i* `i` *su klasnog tipa. Niska* `h` *se inicijalizuje konstantnim tekstom. Polje* `i` *je tipa sopstvene klase* `Alfa` *i poprima podrazumevanu vrednost* **`null`** *(klase su pokazani tipovi i promenljive i polja klasnog tipa su, u stvari, pokazivači).*

Na kraju, polja `j` *i* `k` *su konačna. Polje* `j` *ima inicijalizator i time je postavljena njegova konačna vrednost. Polje* `k` *nema inicijalizator i zasad ima podrazumevanu vrednost 0. Ovo se, međutim, ne smatra postavljanjem vrednosti, tako da je dozvoljena jedna kasnija dodela vrednosti. Pošto konačna polja ne mogu da se promene izvan klase, ne smeta što su polja* `j` *i* `k` *javna. Izvan klase moći će samo da se dohvate, a ne i da se promene, pa nije narušen princip učaurivanja podataka.*

U metodi `main` *klase* `AlfaTest` *prikazane su mogućnosti korišćenja članova klase* `Alfa` *izvan te klase (unutar neke druge klase). Prvom naredbom se stvara objekat* `a` *tipa* `Alfa`*, što je detaljno objašnjeno u §6.5 (111). Javnom članu* `c` *dozvoljen je pristup sa* `a.c`*, ali* `a.b` *nije dozvoljeno, pošto je* `b` *privatan član klase* `Alfa`*.*

U poslednje dve naredbe pristupa se javnim konačnim članovima `j` *i* `k`*. Za njih je dozvoljeno samo dohvatanje vrednosti kao operand (*`x=a.j`*), a ne i promena vrednosti (*`a.k=13`*).*
■

6.4 Metode klasa

Metode klasa su funkcionalni članovi koji ostvaruju operacije nad poljima klasa menjajući time vrednosti (stanja) objekata.

Metode se obično definišu kao programski moduli koji na osnovu izvesnog broja argumenata daju jedan rezultat koji se naziva **povratna vrednost** (*return value*), ili skraćeno samo *vrednost* metode. Tip povratne vrednosti naziva se **tip metode**. Vrednost metode može da se koristi ugrađivanjem poziva metode u izraze. Poziv metode se, u stvari, smatra operatorom kao i svi ostali operatori.

Jezik *Java* dozvoljava da metode, pored vrednosti metode, daju i druge rezultate koji se nazivaju **bočni efekti** metoda. Štaviše, vrlo često vrednost metode uopšte se ne koristi, već samo njeni bočni efekti. Ukoliko nema potrebe da se koristi vrednost metode, poziv metode predstavlja jedini operand izraza u okviru kojeg se poziva metoda.

Jezik *Java* dozvoljava i da metoda uopšte ne stvara vrednost već samo bočne efekte. Nema smisla da se pozivi metoda, koje stvaraju samo bočne efekte, ugrađuju kao operandi u složenije izraze.

Metode su najčešće javne, a ređe privatne (§1.2.2, 4). Pomoću javnih metoda moguće je obrađivati objekte klase iz drugih klasa, dok se privatne metode koriste kao pomoćne metode unutar sopstvene klase.

7 Paketi

Paketi (*packages*) omogućuju grupisanje srodnih klasa u logičke celine. Svaki paket čini zaseban doseg koji se naziva **paketski doseg** (*package scope*). To znači da klasa s istim imenom može postojati u više paketa.

Klase u paketu čine članove paketa.

7.1 Definisanje paketa

Paket se definiše stavljanjem naredbe:

```
package imePaketa ;
```

kao prvu naredbu u jedinici prevođenja (datoteci). Time sve klase u datoteci pripašće tom paketu. Ne mogu u istoj datoteci da se nalaze klase koje pripiadaju različitim paketima. Sme da postoji najviše jedna naredba **`package`** u svakoj datoteci.

Sve klase istog paketa mogu da se nalaze u više datoteka. Ipak, sve te klase pripadaju zajedničkom dosegu. Ne treba (nije ni moguće) posebno naznačiti ako se u jednoj datoteci koristi klasa koja se nalazi u drugoj datoteci.

Klase koje se ne stavljaju u pakete pripadaju **bezimenom paketu** (*unnamed package*), koji se naziva i **podrazumevani paket** (*default package*). Sve dosadašnje klase u ovoj knjizi pripadaju bezimenom paketu.

Stavljanje klasa u bezimeni paket treba izbegavati. Njega treba koristiti samo za manje važne klase s ograničenim vekom upotrebe, i ponekad u početku razvijanja složenog sistema dok se ne uoče logičke celine klasa.

Klase unutar paketa su u izvesnoj meri međusobno prijateljske. Mogu da koriste sve neprivatne članove, uključujući i ugnežđene klase članove, drugih klasa istog paketa. To su, pored javnih članova, članovi koji nisu obeleženi ni kao javni ni kao privatni. Drugim rečima, član je dostupan iz drugih klasa istog, a ne i iz klasa drugih paketa se ako na početku definicije ne nalazi ni modifikator **`private`**, ni **`public`**. Nazivaju se i *polujavni* članovi: javni su unutar paketa, ali ne i izvan paketa. Ne postoji modifikator za označavanje ove vrste dostupnosti.

Da bi klase u paketu mogle da se koriste u drugim paketima treba navesti modifikator **`public`** na početku njihovih definicija. Drugim rečima, i za klase najvišeg nivoa (koje nisu

ugnežđene – §9, 🗎209) podrazumevana dostupnost jeste na nivou paketa. Klase najvišeg nivoa ne mogu biti privatne.

U svakoj jedinici prevođenja sme da postoji samo jedna javna klasa. Ime datoteke mora se poklapa s imenom sadržane javne klase.

Ako u jedinici prevođenja (datoteci) postoji više klasa najvišeg nivoa, klase čija se imena razlikuju od imena datoteke ne bi trebalo da se koriste izvan te jedinice prevođenja. U suprotnom mogu nastati problemi prilikom prevođenja druge jedinice prevođenja u kojoj se koristi neka klasa iz prve. Naime, prevodilac (program `javac` – §1.4.1, 🗎9) kad u toku prevođenja neke klase `T1` nailazi za potrebu za klasom `T2`, prvo će da traži datoteku `T2.java` da bi implicitno prevodio sadržaj te datoteke. Ako takve datoteke nema, tražiće datoteku `T2.class` i koristiće sadržaj te datoteke. Ako ne nađe ni tu datoteku, prijavljuje grešku da je tip `T2` nepoznat. Zato je u ovakvim slučajevima redosled prevođenja pojedinih jedinica prevođenja bitan: prvo treba eksplicitno tražiti prevođenje jedinice koja sadrži klasu `T2`, pa tek onda da se prevodi jedinica koja sadrži `T1`.

Kao konačan zaključak: izbegavati stavljati više klasa, čak i ako nisu javne, u istu jedinicu prevođenja (datoteku) i datoteke imenovati prema imenima sadržanih klasa. Eventualne dodatne klase u datoteci trebale bi da budu pomoćne klase "glavoj" klasi, klasi čije se ime poklapa s imenom datotete, i da se ne koriste izvan te datoteke.

Primer 7.1 *– Definisanje paketa*

```
// Beta.java
package prvi;
class Alfa {        // JAVNA KLASA.
  private int a;
          int b;
  public  int c;

  public void m() {
    Beta beta = new Beta();
    beta.a = 1;     // GREŠKA: polje a je privatno.
    beta.b = 2;
    beta.c = 3;
  }
}
public class Beta { // PAKETSKA KLASA.
  private int a;
          int b;
  public  int c;

  public void m() {
    Alfa alfa = new Alfa();
    alfa.a = 1;     // GREŠKA: polje a je privatno.
    alfa.b = 2;
    alfa.c = 3;
  }
}
```

Jedinica prevođenja u datoteci `Beta.java` pripada paketu `prvi`. Sadrži jednu paketsku klasu `Alfa` i javnu klasu `Beta`. Ime javne klase poklapa se s imenom datoteke.

U obe klase postoje po jedno privatno (`a`), paketsko (`b`) i javno (`c`) polje kao i javna metoda `m`. U metodama `m` pravi se jedan objekat tipa druge klase, posle čega se pristupa

poljima napravljenih objekata. Iz obe klase može da se pristupi paketskom i javnom polju druge klase, a ni iz jedne privatnom polju.

```
// Gama.java
package prvi;
class Gama {          // PAKETSKA KLASA.
  private int a;
          int b;
  public  int c;
  void m() {
    Alfa alfa = new Alfa();
    alfa.a = 1;       // GREŠKA: polje a je privatno.
    alfa.b = 2;
    alfa.c = 3;
    Beta beta = new Beta();
    beta.a = 1;       // GREŠKA: polje a je privatno.
    beta.b = 2;
    beta.c = 3;
  }
}
```

Paketska klasa `Gama` *nalazi se u datoteci* `Gama.java` *i, takođe, pripada paketu* `prvi`*. U metodi* `m` *pravi se po jedan objekat klase* `Alfa` *i klase* `Beta`*. Pošto te klase pripadaju istom paketu, bez obzira na to što se nalaze u drugoj jedinici prevođenja, uslovi za pristup članovima iz klase* `Gama` *su isti kao i između klasa* `Alfa` *i* `Beta`*.*

Pošto se klasa `Alfa` *ne nalazi u datoteci* `Alfa.java`*, klasa datoteka* `Gama.java` *ne može da se prevodi a da prethodno nije prevedena datoteka* `Beta.java`*. Naime, kad prevodilac u toku prevođenja datoteke* `Gama.java` *naiđe na ime klase* `Alfa`*, prijaviće grešku jer ne može da nađe ni datoteku* `Alfa.java`*, ni* `Alfa.class`*. Ako bi se prvo prevodila datoteka* `Beta.java`*, kao rezultat dobili bi datoteke* `Alfa.class` *i* `Beta.class`*, posle čega bi prevođenje datoteke* `Gama.java` *bilo uspešno.*

Može još da se primeti da kad bi se u klasi `Gama` *prvo pravio objekat klase* `Beta` *i tek posle toga objekat klase* `Alfa`*, prevođenje datoteke* `Gama.java` *bi svakako uspelo. Naime, kad bi prevodilac u toku prevođenja datoteke* `Gama.java` *naišao na ime klase* `Beta`*, potražio bi datoteku* `Beta.java`*, preveo bi je i time napravio datoteke* `Alfa.class` *i* `Beta.class`*. Posle kad naiđe na ime klase* `Alfa` *već će imati datoteku* `Alfa.class`*.* ■

7.2 Korišćenje članova paketa

Javna klasa iz drugog paketa može da se koristi navođenjem imena paketa ispred imena klase izrazom oblika:

```
imePaketa . ImeKlase
```

Ovo se naziva **potpuno ime** (*fully qualified name*) klase.

Ime klase može da se uveze iz dosega nekog paketa u doseg datoteke u kojoj se koristi naredbom oblika:

```
import imePaketa . ImeKlase ;
```

8 Nasleđivanje

Objekti u svakodnevnom životu mogu da se grupišu na osnovu njihovih zajedničkih osobina. Unutar grupa moguća su dalja grupisanja na osnovu nekih specifičnijih zajedničkih osobina. Svi objekti date podgrupe poseduju sve osobine svoje grupe kao i još neke specifične osobine. Na primer:

- geometrijske figure u ravni mogu da budu okarakterisane položajem, orijentacijom, površinom i obimom,
  - krugovi su geometrijske figure u ravni koje su dodatno okarakterisane poluprečnikom,
  - kvadrati su geometrijske figure u ravni koje su dodatno okarakterisane dužinom ivice,
  - trouglovi su geometrijske figure u ravni koje su dodatno okarakterisane dužinama stranica,
  - …;
- vozila prevoze stvari i ljude,
  - letelice su vozila koja lete,
    - avioni su letelice,
    - helikopteri su letelice,
    - rakete su letelice,
    - vasionski brodovi su letelice,
  - plovila su vozila koja plove po vodi,
    - čamci su plovila,
    - jedrenjaci su plovila,
    - brodovi su plovila,
  - suvozemna vozila su vozila koja se kreću po zemlji,
    - bicikli su suvozemna vozila s dva točka,
    - …,
  - …;
- …

Kaže se da podgrupa **nasleđuje** (*inherits*) sve osobine svoje opštije nadgrupe i da je nadgrupa **predak** podgrupe, odnosno podgrupa **potomak** nadgrupe.

Podgrupe se mogu formirati i u više koraka. Nadgrupa za neku podgrupu može i sama da bude podgrupa druge klase. Na primer, za avione je rečeno da su letelice, a same letelice su podgrupa vozila. S druge stane, i avioni bi mogli da se podele dalje na putničke i teretne.

Drugim rečima, osobine neke opšte grupe mogu da se nasleđuju u više koraka – kroz više generacija.

8.1 Potklase

U jeziku *Java* grupe objekata opisuju se pomoću klasa. Klase koje opisuju podgrupu objekata s nekim dodatnim, specifičnim osobinama u odnosu na opštije grupe nazivaju se **potklase** (*subclasses*). Polazne klase, koje opisuju opštiju grupu objekata, nazivaju se **natklase** (*superclasses*) za date potklase.

8.1.1 Definisanje potklasa

Potklase se definišu opisom **`class`** čiji je opšti oblik:

```
modifikatori class ImePotklase extends ImeNatklase {
  modifikatori član
  modifikatori član
  …
}
```

Klasa je potklasa ako u definiciji iza njenog imena postoji ključna reč **`extends`** i ime natklase. Inače, dobija se već dobro poznata „obična" klasa (§6.1, 100). Kao što se se vidi, u jeziku *Java* se kaže da potklasa **proširuje** (*extends*) natklasu u smislu da potklasa dodaje neke nove mogućnosti mogućnostima natklase.

Ako se na početku definicije klase stavlja modifikator **`final`**, klasa je **konačna** (*final class*) i ne može dalje da se proširuje. Drugim rečima, ne može biti natklasa drugim klasama.

Članove potklase, pored sopstvenih *`članova`* (navedenih unutar para vitičastih zagrada `{}`) čine i članovi (polja i metode) njene natklase. Kaže se da potklasa **nasleđuje** (*inherits*) sve članove svoje natklase i da je natklasa **predak** potklase, odnosno potklasa **potomak** natklase.

Nenasleđeni članovi potklase nazivaju se i sopstveni ili specifični članovi.

Kontrola korišćenja članova klase ostvaruje se dodavanjem jednog ili nijednog od modifikatora **`private`**, **`protected`**, **`public`**. Time se članovi dele na privatne, paketske, zaštićene i javne. Privatni članovi (**`private`**). mogu da se koriste samo u matičnoj klasi. Paketski članovi (bez modifikatora) mogu da se koriste u matičnoj klasi i u svim klasama koje pripadaju istom paketu kao i matična klasa. Zaštićeni članovi (**`protected`**) mogu da se koriste u matičnoj klasi, u svim klasama koje pripadaju istom paketu kao i matična klasa i u svim klasama drugih paketa koje su potklase matične klase. Na kraju, javni članovi (**`public`**) mogu da se koriste unutar svih klasa, bez ograničenja.

Kaže se da se članovi natklase koji ne mogu da se koriste u potklasi ne nasleđuju. To su privatni članovi uvek, a paketski članovi samo ako se potklasa ne nalazi u istom paketu s natklasom. Ovo „nenasleđivanje" treba shvatiti logički: imena tih članova se ne nalaze u dosegu potklase. Zbog toga, pored toga što ne mogu da se koriste, ne mogu ni dalje da se nasleđuju u eventualnim potklasama posmatrane potklase (u potpotklasama prvobitne kla-

se). Fizički, sva „nenasleđena" polja su prisutna u objektima potklase i ulaze u veličinu objekata potklase. Nasleđene metode koje mogu da se pozivaju iz potklase, i te kako mogu da koriste i ta nedostupna polja. Slično je i sa nedostupnim, i time „nenasleđenim", metodama: njih ipak mogu da pozivaju metode natklase koje su dostupne potklasi.

Klasa može biti samo javna ili paketska. To znači da za njih, od modifikatora iz prethodnog pasusa, sme da se koristi samo modifikator **`public`**. Može biti paketska ili javna, a ne i privatna ili zaštićena klasa.

Glavni smisao nasleđivanja manifestuje se kod nestatičkih članova. Jedan primerak nasleđenog nestatičkog polja postoji u svakom objektu potklase, pored toga što postoji i u svakom objektu natklase.

Nasleđivanje statičkog polja ogleda se samo u tome da u potklasi mogu da se koriste i samo prostim imenom. Potklasa ne dobija svoj primerak „nasleđenog" statičkog polja, već potklasa koristi isti primerak koji koristi i natklasa. Može da se kaže da se statički članovi nasleđuju samo logički, a ne i fizički.

Članovi potklase koji su nasleđeni od natklase imaju istu dostupnost kao i u natklasi. Posebno je važno to što su javni članovi natklase javni i u potklasi. Zbog toga s objektima potklase mogu da se izvode sve radnje koje mogu s objektima natklase i, pored njih, još neke druge radnje koje postoje samo u potklasi.

Kaže se da su potklasa i natklasa u odnosu *jeste* ili *je*: potklasa *je* natklasa s dodatnim mogućnostima. Tamo gde se očekuje objekat natklase, može ravnopravno da se navede i objekat potklase.

Konstruktori se ne smatraju članovima klasa (§6.5, 🗎111) i zbog toga se ne nasleđuju. Kao i bilo koja klasa, i potklasa može da ima proizvoljan broj konstruktora. Naredbom oblika:

```
super ( argumenti ) ;
```

kao prvom naredbom u telu konstruktora potklase, može da se pozove konstruktor natklase radi inicijalizacije polja koja su nasleđena od natklase. Pozivaće se konstruktor natklase koji odgovara navedenim *`argumentima`*. U nedostatku ove naredbe, pre prve naredne u telu konstruktora potklase, pozivaće se podrazumevani konstruktor natklase (**`super`**`()`).

Ako se u potklasi ne definiše nijedan konstruktor, generiše se podrazumevani konstruktor koji samo poziva podrazumevani konstruktor natklase.

U telu konstruktora eksplicitno može da se poziva najviše jedan konstruktor: konstruktor svoje klase (**`this`**`(…)` – §6.5, 🗎112) ili konstruktor natklase (**`super`**`(…)`).

Primer 8.1 *– Definisanje potklasa s konstruktorima*

```
// A.java
package pak1;
public class A {                          // Natklasa.
  …
  public A( … ) { … }
  …
}
```

Klasa `A` *je u paketu* `pak1`*. Pored nekoliko članova ima bar jedan konstruktor.*

```
// B.java
package pak1;
public class B extends A {                  // Potklasa u istom paketu.
  …
  public B( … ) { super( … ); … }
  …
}
```

*Klasa `B`, takođe u paketu `pak1`, je potklasa klase `A`. Prvom naredbom u telu konstruktora poziva se konstruktor natklase (**`super`**`(…)`) radi inicijalizacije nestatičkih polja nasleđenih iz klase `A`. Argumenti ovog poziva određuju koji će od konstruktora klase `A` biti pozvan.*

```
// C.java
package pak2;
public final class C extends pak1.A {    // Potklasa u drugom paketu.
  …
  public C( … ) { … }
  …
}
```

*Klasa `C` u paketu `pak2` je konačna. Neće moći da se definišu klase koje bi bile njene potklase. Da bi mogla biti potklasa klase `A`, koja nije u istom paketu, potrebno je navesti potpuno ime klase `A` (`pak1.A`). Druga mogućnost je da se ime klase `A` prethodno uveze naredbom **`import`** u doseg jedinice prevođenja u datoteci `C.java`. U konstruktoru klase `C` nema eksplicitnog poziva konstruktora natklase, pa će se, pre prve naredbe u telu, pozivati podrazumevani konstruktor klase `A` (**`super`**`()`).*

```
// D.java
package pak2;
public class D extends C {                  // GREŠKA: Klasa C je konačna.
  …
  public D(  …  ) { …  }
  …
}
```

Pošto je klasa `C` konačna, ne mogu da se definišu klase koje su njene potklase, bez obzira na to u kom paketu se one nalaze. ■

8.1.2 Korišćenje članova potklasa

Svi članovi potklase, kako sopstveni tako i nasleđeni, mogu da se koriste na jedinstven način koji je opisan za „obične" klase u §6.2 (100), vodeći računa o njihovoj dostupnosti.

Unutar potklase prosto ime člana je dovoljno (`član`) za sve članove: sopstvene – nasleđene, nestatičke – statičke. Po potrebi, može da se koristi i skriveni parametar metode (**`this.`**`član`).

Za nasleđene članove može da se koristi i izraz **`super.`**`član`, gde je **`super`** simboličko ime za podobjekat neposredne natklase. Ovo se preporučuje za nestatičke nasleđene članove ako samo `član` nije dovoljan. Treba razlikovati ovu upotrebu ključne reči **`super`** od poziva konstruktora natklase (**`super`**`(…)`) iz prethodnog odeljka. Izraz **`super.`**`član` može da se koristi u bilo kojoj metodi (ne samo u konstruktoru) i u bilo kojoj naredbi metode (ne samo u prvoj).

Za statičke članove može da se koristi i izraz `Natklasa.član`, pri čemu može da se navede ime natklase i na višem nivou hijerarhije nasleđivanja, a ne samo ime neposredne natklase. Ovo se preporučuje za nasleđene statičke članove ako samo `član` nije dovoljan.

9 Ugnežđeni tipovi

Ugnežđen tip (*nested type*) je tip (klasa ili interfejs) koji je definisan unutar drugog, **okružujućeg tipa** (*surrounding type*). Tip koji nije unutar drugog tipa naziva se **tip najvišeg nivoa** (*top level type*). Jedinice prevođenja (datoteke izvornog teksta) mogu da sadrže jedan ili više tipova najvišeg nivoa, od kojih najviše jedan može biti javan (§7.1, 🗎140).

Ime ugnežđenog tipa ne sme biti jednako imenu okružujućeg tipa, klase ili interfejsa. Njegovo ime pripada dosegu unutar kojeg je definisano. Unutar tog dosega može da se koristi prostim imenom. Izvan okružujućeg tipa treba da se navede i ime tog tipa:

```
ImeOkružujućegTipa . ImeUgnežđenogTipa
```

Doseg ugnežđenog tipa obuhvata i sve članove okružujućeg tipa, bez obzira na to da li se oni nalaze pre ili posle definicije ugnežđenog tipa.

Ugnežđeni tip može da se definiše i unutar bloka. Tada je njegovo ime u dosegu tog bloka i izvan tog bloka ne može da se koristi.

Ugnežđeni i okružujući tip su prijateljski jedan drugom. To znači da ugnežđeni tip može da pristupa i privatnim članovima okružujućeg tipa i obrnuto.

Članovi okružujućeg tipa iz ugnežđenog tipa mogu da se koriste prostim imenom, pošto za svaki ugnežđeni tip postoji samo jedan okružujući tip.

Pristup članovima ugnežđenog tipa moguć je samo navođenjem imena ugnežđenog tipa (za statičke članove) ili promenljive ugnežđenog tipa (za nestatičke članove), jer unutar okružujućeg tipa mogu da postoje više ugnežđenih tipova, a i za jedan ugnežđeni tip mogu postojati više objekata.

Član ugnežđenog tipa sakriva polje istog imena, odnosno metodu istog potpisa u okružujućem tipu. Metoda s istim imenom, ali drugačijim potpisom samo preopterećuje ime metode (§6.4.5, 🗎109).

Sakrivenim članovima može da se pristupa navođenjem imena okružujuće klase ispred imena člana:

```
ImeOkružujućegTipa . ImePolja                                    odnosno ...
ImeOkružujućegTipa . ImeMetode ( argumenti )
```

Inače, pristupa se članu ugnežđene klase. Eventualni izuzeci od ovog pravila navedeni su u narednim odeljcima.

Mada ugnežđeni tipovi donose određene pogodnosti, ne treba preterivati s njihovom upotrebom, jer čine definiciju tipa najvišeg nivoa kabasatim. Treba izbegavati složene i

dugačke ugnežđene tipove, a naročito sisteme ugnežđenih tipova. Definiciju klase od nekoliko desetina stranica vrlo je teško sagledati i održavati!

Primer 9.1 *– Ugnežđeni tipovi*

Ugnežđena klasa `Unutra1` *(Slika 9.1) članove okružujuće klase* `Spolja` *može da koristi navođenjem samo njihovih imena (*`b=a`*). Polje* `b` *te ugnežđene klase sakriva istoimeno polje okružujuće klase* `Spolja`*, zato samo* `b` *u izrazu* `c=b` *označava polje ugnežđene klase. Za pristup sakrivenom polju treba pisati* `Spolja.b`*.*

```
class Spolja {                  // Okružujuća klasa.
  private        int a = 1;
  private static int b = 2;
  public  class Unutra1 {       // Javna ugnežđena klasa.
    private int b = a;          // Sakriva Alfa.b
    private int c = b;          // Koristi se this.b
    private int d = Spolja.b;   // Koristi se Spolja.b
  }
  int e = c;                    // GREŠKA: Unutra1.c nije u dosegu.
  int f = new Unutra1().c;      // Ovako može.
  void m() {
    class Unutra2 {}            // Klasa unutar bloka (metode).
  }
}
```

Slika 9.1 *– Ugnežđene klase*

Okružujuća klasa `Spolja` *ne može da koristi članove ugnežđenih klasa samo imenom (*`e=c`*), već mora da se navede i neki objekat (*`f=…`*).*

Dostupnost članova nije važna bilo da se pristupa članu okružujuće klase iz ugnežđene klase ili obrnuto.

Klasa `Unutra2` *je primer ugnežđene klase definisane unutar metode* `m`*.*

```
Unutra1 p = null;             // GREŠKA: Unutra1 nije u dosegu.
Spolja.Unutra1 q = null;      // Ovako može.
```

Izvan okružuje klase `Spolja` *ime ugnežđene klase* `Unutra1` *nije u dosegu, već mora da se piše* `Spolja.Unutra1`*.* ■

9.1 Klase članovi

Klasa član (*member type*) je ugnežđena klasa koja je definisana neposredno u klasi, a ne u bloku neke metode.

Kao i svim članovima, dostupnost klase člana može da bude privatna, paketska, zaštićena ili javna što se pri definisanju označava odgovarajućim modifikatorom ispred imena tipa (redom: **`private`**, *bez modifikatora*, **`protected`** i **`public`**). Dostupan tip član izvan okružujuće klase može da se koristi navođenjem imena okružujućeg tipa:

```
ImeOkružujućegTipa . ImeTipaČlana
```

Klasa član može biti statička ili nestatička.

9.1.1 Statičke ugnežđene klase

Statička ugnežđena klasa (*static nested class*) je klasa član koja se definiše navođenjem modifikatora **`static`** ispred imena klase. Ponekad, skraćeno, pod *ugnežđena klasa* podrazumeva se statička ugnežđena klasa, dok ostale vrste ugnežđenih klasa imaju svoja zasebna imena.

Statička ugnežđena klasa može da ima statičke i nestatičke članove.

U statičkoj ugnežđenoj klasi neposredno, navođenjem samo imena, mogu da se koriste samo statički članovi okružujuće klase. Nestatičkim članovima može da se pristupa samo za konkretne objekte.

Članovi statičke ugnežđene klase mogu da sakriju samo statičke članove okružujuće klase. Pojam sakrivanja nestatičkih članova okružujuće klase ne postoji, pošto za pristup do njih uvek mora da se navodi objekat okružujuće klase.

Primer 9.2 – Statička ugnežđena klasa

Na početku okružujuće klase `Spolja` na slici 9.2 nalaze se četiri privatna polja (`a`, `b`, `c` i `d`), i dve javne metode `met1` i `met2`. Njih koristi statička ugnežđena klasa `Unutra` u metodi `met3`. Ta metoda ima parametar `s1` tipa `Spolja` za slučajeve da u telu metode treba navesti neki objekat tipa `Spolja`. Naglašava se da su okružujuća i ugnežđena klasa prijateljske jedna drugoj, i mogu da koriste i privatne članove druge klase.

Prva dva polja `c` i `d` statičke ugnežđene klase `Unutra` imaju ista imena kao i neka polja okružujuće klase. Tip i statičnost ovih polja nisu bitni. Smatra se da samo polje `c` sakriva istoimeno statičko polje klase `Spolja`, jer poklapanje imena polja `d` ne utiče na način korišćenja nestatičkog polja klase `Spolja`. Nesakriveno statičko polje može da se koristi samo imenom (`f=a` u metodi `met3`), dok za sakriveno polje treba navesti i ime okružujuće klase (`j=Spolja.c`). Za nestatička polja uvek treba navesti i neki objekat okružujuće klase (`h=s1.b` i `l=s1.d`). U slučaju jednakosti imena, samo ime uvek predstavlja polje ugnežđene klase (`i=c` i `k =d`).

Slično je i sa metodama `met1` i `met2` s istim potpisima (ne računajući tipove povratnih vrednosti) u poslednja četiri reda metode `met3`.

Metoda `met4` pripada okružujućoj klasi `Spolja` i pokazuje mogućnosti korišćenja članova statičke ugnežđene klase `Unutra` unutar svoje okružujuće klase. Na početku metode `met4` napravljen je objekat `u1` tipa `Unutra` za slučajeve potrebe u nastavku. Za pristup članovima ugnežđene klase uvek treba navesti neki objekat te klase za nestatičke članove (`u1.c` i `u1.met1()`), odnosno ime te klase za statičke članove (`Unutra.e` i `Unutra.met2()`).

*Klasa `Test` na slici 9.2 je nezavisna klasa od prethodne dve i pokazuje kako može statička ugnežđena klasa da se koristi izvan njene okružujuće klase. Uz ime ugnežđene klase uvek treba navesti i ime okružujuće klase (`Spolja.Unutra`), bez obzira da li se radi o definisanju promenljive (`Spolja.Unutra u3`), stvaranju objekta (**`new`** `Spolja.Unutra()`) ili pristupu statičkom članu (`Spolja.Unutra.met2()`) ugnežđene klase. Za nestatičke članove, kao i uvek, potrebno je samo navesti ime objekta ugnežđene klase (`u3.met1()`). Naravno, sve ovo pod uslovom da su kako sama ugnežđena klasa, tako i njeni članovi dostupni na mestu korišćenja.* ■

```
class Spolja {                          // Okružujuća klasa.
  private static int a = 1;
  private        int b = 2;
  private static int c = 3;
  private        int d = 4;
  public  static int met1() { return 0; }
  public         int met2() { return 0; }

  public static class Unutra {          // Statička ugnežđena klasa.
    private              byte c = 5; // Sakriva Spolja.c
    private static final int  d = 6;
    private static       int  e = 7;
    public         void met1() {}     // Sakriva Spolja.met1()
    public  static void met2() {}

    public         void met3(Spolja s1) {
      int f = a;                    // Koristi se Spolja.a
      int g = b;                    // GREŠKA: b nije statičko polje.
      int h = s1.b;                 // Koristi se Spolja.b
      int i = c;                    // Koristi se Unutra.c
      int j = Spolja.c;             // Koristi se Spolja.c
      int k = d;                    // Koristi se Unutra.d
      int l = s1.d;                 // Koristi se Spolja.d
      met1();                       // Poziva  se Unutra.met1()
      Spolja.met1();                // Poziva  se Spolja.met1()
      met2();                       // Poziva  se Unutra.met2()
      s1.met2();                    // Poziva  se Spolja.met2()
    }
  }

  public void met4() {
    Unutra u1 = new Unutra();
    int m = c;                  // Koristi se Spolja.c
    int n = Unutra.c;           // GREŠKA: Unutra.c nije statičko.
    int o = u1.c;               // Koristi se Unutra.c
    int p = e;                  // GREŠKA: Unutra.e nije u dosegu.
    int q = Unutra.e;           // Koristi se Unutra.e
    met1();                     // Poziva  se Spolja.met1()
    Unutra.met1();              // GREŠKA: Unutra.met1() nije statička.
    u1.met1();                  // Poziva  se Unutra.met1()
    met2();                     // Poziva  se Spolja.met2()
    Unutra.met2();              // Poziva  se Unutra.met2()
  }
}
class Test {                    // Neka druga klasa.
  void test() {
    Unutra u2 = new Unutra(); // GREŠKA: Unutra nije u dosegu.
    Spolja.Unutra u3 = new Spolja.Unutra();
    u3.met1();
    Spolja.Unutra.met2();
  }
}
```

***Slika 9.2** – Staička ugnežđena klasa*

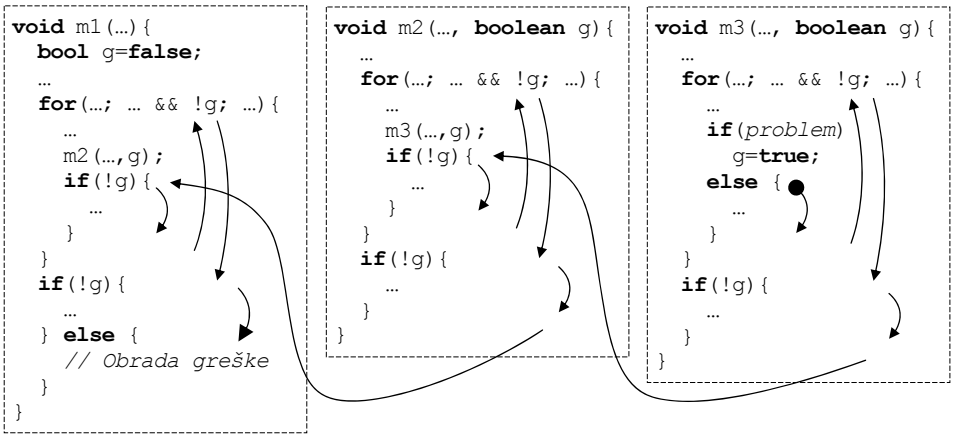
10 Izuzeci

Bez posebne podrške od strane programskog jezika konfliktne situacije (greške) u programima obično se obrađuju na sledeći način:

- Ako se unutar neke metode otkrije greška, metoda se završi vraćajući neku informaciju o tome pozivajućoj metodi. Ta informacija obično bude neka celobrojna šifra, ili logički indikator, koja se često vraća kao vrednost metode.
- Ako ni metoda koja dobija informaciju o grešci nije u stanju da razreši nastalu situaciju, prosleđuje šifru ili indikator greške metodi koja je nju pozvala. Ako se na taj način stiže do glavne metode prekida se izvršavanje celog programa.
- Kada se stigne do metode koja ume da razreši nastali problem, ona preduzme odgovarajuće korektivne akcije i izvršavanje programa se nastavi dalje.

Glavni problem je što su sve metode složenog programskog sistema opterećene brigom *Da li je negde otkrivena greška?*, a ne samo metoda u kojoj ta greška može da nastane.

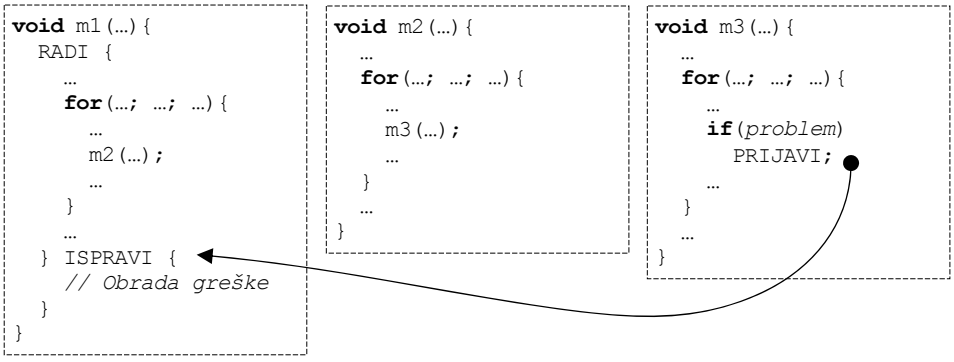
Na slici 10.1 prikazane su tri metode od kojih `m1` poziva `m2` i `m2` poziva `m3`. Greška koju može da obrađuje metoda `m1` otkrije se u metodi `m3`. Prenos upravljanja od mesta otkrivanja greške do mesta obrade greške omogućen je indikatorom `g` čija vrednost **`true`** označava pojavu greške. Vrednost tog indikatora mora svaki čas da se proverava. Strelice na slici 10.1 označavaju putanju od mesta otkrivanja greške do mesta njene obrade.



Slika 10.1 *– Obrada greške pomoću indikatora*

Jezik *Java* nudi efikasan mehanizam izuzetaka za obradu grešaka koji rasterećuje programera od većine, ako ne i od svih, navedenih problema. Kad se na nekom mestu u programu otkrije greška, potrebno je samo da se ona prijavi odgovarajućim **izuzetkom** (*exception*). Bezbedno prekidanje započetih aktivnosti i predaja upravljanja nadležnom **rukovaocu izuzecima** (*exception handler*) dešava se automatski.

Na slici 10.2 prikazane su metode sa slike 10.1 za slučaj upotrebe mehanizma izuzetaka. Pojedine metode se programiraju za slučaj da je sve u redu i nigde se ne postavlja pitanje, da li je otkrivena greška. Jedino se metoda `m1`, koja je u stanju da obradi greške, deli na dva dela. Prvi deo (blok `RADI`) odnosi se na obradu kada je sve u redu, a drugi deo (blok `ISPRAVI`) sadrži korake za obradu otkrivene greške. Kada se u metodi `m3` otkrije greška, njenim prijavljivanjem (naredba `PRIJAVI`) upravljanje se prenosi neposredno na blok `ISPRAVI` u metodi `m1`, pri čemu se metoda `m2` u potpunosti zaobilazi. Treba uočiti da su time prekinute sve do tada započete aktivnosti (ciklusi i metode).

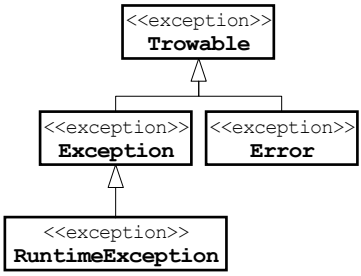


Slika 10.2 *– Mehanizam izuzetaka*

10.1 Klase za izuzetke

Izuzeci u jeziku *Java* predstavljaju se objektima tipa klase `Throwable` ili njenih potklasa u proizvoljnom broju nivoa nasleđivanja. Tip objekta koji se šalje rukovaocu naziva se *tip izuzetka*.

Na slici 10.3 prikazane su klase na vrhu hijerarhije klasa za izuzetke. Klase za izuzetke u dijagramima klasa označavaju se sa `<<exception>>` iznad imena klasa.



Slika 10.3 *– Glavne klase za izuzetke*

Klasa `Throwable` je neposredna potklasa klase `Object`, a predstavlja praklasu za sve klase kojima mogu da se prijavljuju izuzeci.

Klasa `Exception` predviđena je za konfliktne situacije iz kojih u većini programa želi da se oporavi i da se program nastavi dalje.

Klasa `RuntimeException` je predviđena za standardne izuzetke koji mogu da se javljaju vrlo često u toku izvršavanja programa. Svi izuzeci koji su spominjani dosada u ovoj knjizi su potklase ove klase.

Klasa `Error` predviđena je za konfliktne situacije iz kojih u većini programa ne želi da se oporavi, već da se program prekine.

Važnije metode ove četiri klase klasa su:

```
Throwable          ()
Throwable          (String por)
Exception          ()
Exception          (String por)
RuntimeException   ()
RuntimeException   (String por)
Error              ()
Error              (String por)
```

Konstruktori bez parametara stvaraju „prazne" objekte, dok konstruktori sa parametrom objekte koji sadrže poruku *por*.

`String` **`getMessage`**`()`

Ova metoda u sve četiri klase vraća poruku sadržanu u tekućem objektu, odnosno **`null`** ako objekat ne sadrži poruku.

`String` **`toString`**`()`

Ova metoda u sve četiri klase vraća tekstualni prikaz tekućeg objekta kao rezultat izraza:

```
getClass().getName() + ": " + getMessage()
```

tj. ime klase izuzetka i tekst poruke. Ako u tekućem objektu nema poruke, vraća se samo ime klase.

Klase za izuzetke koje definišu programeri trebalo bi da su potklase klase `Exception`, a mogu da budu i neposredne potklase klase `Throwable`. Nikako ne bi trebalo da budu potklase klasa `RuntimeException` i `Error`.

Klase za izuzetke mogu imati sopstvena polja i metode i da se na taj način u objektima prenose proizvoljne količine informacija sa mesta prijavljivanja izuzetka do rukovaoca izuzecima.

Primer 10.1 *– Definisanje klasa za izuzetke*

```
class G1 extends Exception { … }
class G2 extends Exception { … }
class G3 extends Exception { … }
class G4 extends G1 { … }
```

Klase `G1`, `G2` i `G3` su neposredne potklase klase `Exception`. Klasa `G4` je potklasa klase `G1` i time potklasa klase `Exception` u drugoj generaciji. ■

11 Generički tipovi i metode

Često je potrebno istu obradu primeniti na podatke različitih tipova. Na primer:

- naći veći od dva podatka,
- ostvariti rad sa stekom,
- urediti niz podataka po nekom kriterijumu,
- …

Sa stanovišta algoritma za sve navedene obrade potpuno je svejedno da li se radi o realnim brojevima, pravougaonicima ili pak o datumima.

Generički tipovi (klase i interfejsi) i metode omogućuju razumljivu i bezbednu obradu podataka različitih tipova zajedničkim klasama, odnosno metodama.

11.1 Tipovne promenljive

Generički tipovi i metode se parametrizuju nizom tipova koji se nazivaju **tipovni parametri** (*type parameters*). Nazivaju se i **tipovne promenljive** (*type variables*), jer u definicijama generičkih klasa, interfejsa i metoda mogu da se koriste analogno korišćenju parametara metoda u telima metoda.

Opšti oblik niza tipovnih parametara (promenljivih) je:

```
< Param , … , Param >
```

Parametar *Param*, u najjednostavnijem obliku, je ime parametra *Ime*. Složeniji oblici su prikazani u §11.4 (🗎268).

Parametri mogu da predstavljaju samo pokazane tipove (klase, interfejse i nizove), a ne i proste tipove (na primer, **`int`**). *Ime* može da se koristi na mestima na kojima se očekuje ime tipa. Doseg imena tipovnih parametara je od mesta uvođenja do kraja definicije klase, interfejsa ili metode za koju su parametri.

Obično se koriste jednoslovna imena. Najčešće se koristi slovo `T` (*type*) ili susedna slova `S` i `U` kao opšta imena tipovnih parametara. Za elemente zbirki (nizova, lista itd.) koristi se slovo `E` (*element*), a za parove ključ/vrednost slova `K` (*key*) i `V` (*value*).

Prilikom definisanja promenljivih generičkih tipova treba navesti niz konkretnih tipova koji se nazivaju **tipovni argumenti** (*type arguments*). Opšti oblik definisanja niza tipovnih argumenata je:

```
< Arg , … , Arg >
```

Broj argumenata *Arg* treba da je jednak broju parametara *Param* u definiciji generičkog tipa. Svaki argument može biti proizvoljan konkretan pokazan tip uključujući i nizove i generičke tipove. Prosti tipovi nisu dozvoljeni.

Pridruživanje tipovnih argumenata tipovnim parametrima (promenljivim) naziva se ***konkretizacija tipovnih parametara***. U definiciji generičkog tipa ili metode svaki argument će zameniti odgovarajući parametar. Ovo je analogno dodeljivanju vrednosti argumenta pri pozivanju metode odgovarajućem parametru. Na mestima parametra u telu metode koristi se dodeljena vrednost argumenta.

Primer 11.1 – *Tipovni parametri i argumenti*

Za niz tipovnih parametara

```
<T, S, U>
```

dve od sledećih konkretizacija su ispravne, a treća nije

```
<Byte, String, Object>
<Integer[], Float, int[]>
<Exception, byte, Random>          // GREŠKA: byte je vrednosni tip.
```

■

11.2 Generičke klase i interfejsi

Generički tipovi su tipovi parametrizovani tipovima. Određeni su imenom generičkog tipa i nizom tipovnih parametara iz prethodnog odeljka. Opšti oblik generičkog tipa je:

```
ImeGenTipa < Param , … , Param >
```

11.2.1 Definisanje generičkih klasa i interfejsa

Generičke klase i interfejsi definišu se tako da se na mestu imena navede generički tip. Opšti oblik definisanja generičke klase, onosno interfejsa je:

```
modifikatori class     GenTip { … }                                    odnosno …
modifikatori interface GenTip { … }
```

Tipovni parametri unutar generičkih tipova mogu da predstavljaju tipove polja, kao i parametara i povratnih vrednosti metoda. Ne mogu da se stvaraju objekti i nizovi objekata tipovnog parametra. Mogu da se definišu nizovne promenljive tipovnog parametra kojima mogu da se dodele nizovi elemenata tipa `Object` uz eksplicitnu konverziju u niz elemenata tipa tipovnog parametra. Prilikom te konverzije prevodilac za jezik *Java* davaće opomenu da nije u stanju da proveri korektnost te konverzije.

Primer 11.2 – *Definisanje generičkih klasa i interfejsa*

```
class GenC1<T>     { T t; }
class GenC2<T, S>  { S m(T t) { return null; } }
interface GenI1<T> { void m(T t, int i); }
```

Generička klasa `GenC1` *ima jedan tipovni parametar čije je ime* `T` *i sadrži jedno polje tipa* `T`.

Generička klasa `GenC2` *ima dva tipovna parametra čija su imena* `T` *i* `S` *i sadrži metodu* `m` *čiji je parametar tipa* `T`*, a povratna vrednost tipa* `S`.

Generički interfejs `GenI1` *ima jedan tipovni parametar čije je ime* `T` *i predviđa metodu* `m` *čiji je prvi parametar tipa* `T`*, a drugi* **`int`**.

```
class GenC3<T> {
  T a = new T();       // GREŠKA: ne može objekat tipa parametra.
  T[] b = new T [10];  // GREŠKA: ne može niz tipa parametra.
  T[] c = (T[]) new Object [10]; // Opomena: ne može da se proveri
                                 //          korektnost konverzije.
  { c[0] = a; }
}
```

`T` *je tipovni parametar generičke klase* `GenC3`. *Ne može da se stvara objekat tipa* `T` (**new** `T()`), *ni niz objekata tog tipa (***new** `T[10]`). *Može da se dodeli memorija nizu (pokazivača na) elemente tipa* `Object` (**new** `Object[10]`) *i da se rezultat, posle eksplicitne konverzije tipa* (`(T[])`) *dodeli promenljivoj* `c` *tipa* `T[]`. *Prevodilac za jezik* Java *će, pri tom, dati opomenu da ne može da garantuje korektnost konverzije.*

Pomoću promenljive `c`, *elementima niza mogu da se dodeljuju objekti tipa* `T` (`c[0]=a`). ■

11.2.2 Konkretizacija generičkih tipova

Prilikom definisanja promenljivih generičkog tipa (klase ili interfejsa) i stvaranja objekata generičke klase kao tip treba navesti ***konkretizaciju generičkog tipa*** koja se sastoji od imena generičkog tipa i niza tipovnih argumenata (konkretizaciju tipovnih parametara) iz §11.1 (🗎258):

```
ImeGenTipa < Arg , … , Arg >
```

Svaka konkretizacija generičkog tipa (kombinacija tipovnih argumenata) čini po jedan nezavisan tip. Ti tipovi nisu međusobno kompatibilni, čak i ako su nastali na osnovu kompatibilnih tipova. Drugačije rečeno, ako je `Gen<T>` generički tip i `P` je podtip nadtipu `N`, tip `Gen<P>` nije podtip tipu `Gen<N>`.

Opšti oblik definisanja promenljive generičkog tipa, odnosno stvaranja objekta tipa generičke klase je:

```
modifikatori GenTip imePromenljive ;                    odnosno …
new GenTip ( … )
```

Drugim rečima, pri definisanju promenljive ili stvaranju objekta potrebno je navesti konkretizovan generički tip.

Primer 11.3 *– Definisanje promenljivih i stvaranje objekata generičkih tipova iz primera 11.2*

```
GenC1<Byte>   a = new GenC1<Byte>();
GenC1<byte>   b = null;              // GREŠKA: ne može prost tip.
GenC1<byte[]> c = null;              // Niz prostog tipa može.
GenC2<String, Long> d = new GenC2<String, Long>();
GenC2<Object, GenC1<Float>> e = null;
```

Promenljiva `a` *u prvom redu je tipa* `GenC1<Byte>` *i inicijalizovana je objektom tog tipa. Sadržano polje* `t` *u objektu je tipa* `Byte`.

12 Niti

Program je pasivan niz naredbi u obliku izvornog teksta na nekom programskom jeziku ili u obliku niza bajtova koji je dobijen prevođenjem izvornog teksta na mašinski jezik računara, uskladišten u jednu ili više datoteka na disku.

Proces je program koji se izvšava u nekom okruženju. Stvara ga operativni sistem računara tako da mu dodeli određeni prostor u memoriji računara i potrebne spoljašnje resurse, napravi određene upravljačke strukture podataka za praćenje njegovog stanja i započinje njegovo izvršavanje. Proces je aktivna stvar koja se odvija u vremenu izvršavanjem njegovih naredbi od strane procesora računara. Današnji operativni sistemi omogućavaju da se istovremeno izvršavaju više programa, tj. da postoje istovremeno više procesa. Svaki ima svoj odvojen adresni prostor i međusobno mogu da izmenjuju podatke samo uz posredovanje operativnog sistema.

Neki operativni sistemi omogućavaju da se u okviru procesa više tokova kontrole (sekvenci naredbi) izvršavaju istovremeno. Ti tokovi kontrole nazivaju se **niti** (*threads*). Niti dele zajednički adresni prostor matičnog procesa i mogu međusobno da izmenjuju podatke bez posredovanja operativnog sistema. Za pristup zajedničkim, deljenim podacima potrebni su određeni mehanizmi **sinhronizacije** da bi komunikacija među nitima bila ispravna. Na primer, dok jedna nit menja skup deljenih podataka, druge niti ne smeju da pristupaju tim istim podacima. Nit koja samo dohvata te podatke, dobiće neusaglašeni skup podataka. Nit koja bi u isto vreme i menjala te podatke, pokvarila bi rezultate druge niti i na kraju taj skup podataka ne bi bio ispravan.

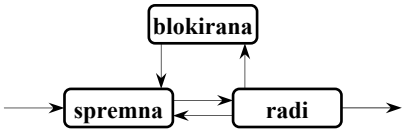
Deo niti u kojima se pristupa deljenim podacima naziva se **kritična sekcija**. Svakog momenta samo jedna nit sme da bude u kritičnoj sekciji za pristup istom skupu deljenih podataka. Kaže se da sadržaj kritične sekcija mora da se izvršava **atomično** (nedeljivo) u smislu da sve naredbe u kritičnoj sekciji moraju da se izvrše pre nego što neka druga nit može da pristupi podacima koji se obrađuju u toj sekciji.

Naredbe izvršava(ju) procesor(i) računara. Na jednoprocesorskom računaru taj jedini procesor treba da izvršava naredbe svih niti svih procesa. Naizgled istovremeno izvršavanje više niti se ostvaruje tako da procesor na smenu izvršava kraće sekvence pojedinih niti. Na dužoj stazi stiče se utisak istovremenog napredovanja svih niti. Ovakav način rada naziva se **konkurentan rad**, jer pojedini nite su konkurenti za korišćenje procesora. Na višeprocesorskom računaru operativni sistem može pojedine niti da rasporedi na različite procesore. Tada i stvarno postoji izvestan stepen istovremenosti, ali ne potpuna, jer obično ima više niti nego procesora.

Kaže se da dok procesor izvršava naredbe jedne niti, radi u kontekstu te niti. Prelazak procesora s jedne niti na drugu naziva se **smena konteksta** (*context switching*).

Iz dosad rečenog proizlazi da procesi su najmanje jednice za dodeljivanje memorije, dok su niti najmanje jedinice za dodeljivanje procesora.

Nit u toku svog života može da se nalazi u tri osnovna stanja (slika 12.1): **spremna** – kad može da koristi procesor ali čeka da dođe do njega, **radi** – kad je dobila proceror na korišćenje i izvršavaju se njene naredbe i **blokirana** – kada čeka na neki događaj da bi mogla da nastavi s radom.



Slika 12.1 *– Stanja niti*

Novostvorena nit se stavlja u stanje *spremna*. U tom stanju može da se nalazi više niti među kojima operativni sistem po nekom kriterijumu bira kojoj će da dodeli procesor kad se oslobodi. Jedan mogući kriterijum može biti po redosledu dolaženja u stanje *spremna*. Često se nitima dodeljuju prioriteti i tada prioritetnije niti biraju se pre manje prioritetnih.

Stanje *radi* nit može da napušta prinudno ili dobrovoljno. Prinudno, kad joj operativni sistem iz nekog razloga oduzme procesor i vraća je u stanje *spremna*. Razlog za oduzimanje može biti istek kvanta vremena koliko je niti dodeljeno da može bez prekida da koristi procesor. Drugi razlog može biti, ako se primenjuje dodeljivanje procesora sa preuzimanjem (*pre-emptive scheduling*), pojava niti višeg prioriteta u stanju *spremna*. Nit dobrovoljno napušta stanje *radi* kada prelazi u stanje *blokirana* da bi čekala na neki događaj ili kad završi s radom i napušta sistem.

Nit iz stanja *blokirana*, kad se desi očekivani događaj, uvek prelazi u stanje *spremna* da sačeka svoj red da dođe do procesora.

Jedan čest problem u višenitnom okruženju je **međusobno blokiranje** (*deadlock*) niti. To je situacija, u najjednostavnijem obliku, kada dve niti zatraže i dobiju ekskluzivno pravo korišćenja po jednog deljenog podatka i posle zatraže ekskluzivno pravo korišćenja deljenog podatka koji je zauzela suprotna nit. Tada će se obe niti blokirati i nikad neće moći da nastave s radom. U međusobnom blokiranju često može da učestvuje i veći broj niti. Tehnike sprečavanja da dođe do međusobnog blokiranja spadaju u domen operativnih sistema uopšte i ne razmatraju se u ovoj knjizi.

12.1 Niti u jeziku *Java*

Javina virtuelna mašina podržava programe s više niti, bez obzira na korišćeni operativni sistem. To je moguće zato što operativni sistem izvršava program `java.exe` (§1.4.1, 9) koji je, u stvari, *Javina virtuelna mašina* (*JVM*). *JVM* korisnički program „izvršava" interpretirajući bajtkod koji se nalazi u skupu datoteka tipa `.class` koje sadrže prevode klasa koje čine programski sistem.

Izvršavanje programa na jeziku *Java* počinje pokretanjem *JVM* koja stvara glavnu nit korisničkog pograma. U glavnoj niti izvršava se metoda `main` klase čije je ime navedeno u