Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva

Marko Topolnik, Mario Kušek

Uvod u programski jezik *Java*

Skripta uz kolegij Informacija, logika i jezici

Zagreb, 2008.

Sadržaj

Popis ispisa

Predgovor

Svrha jednog dijela kolegija posvećena jezicima, točnije jezicima Java i UML (Unified Modeling Language), je ne samo uputiti studente u sintaksu, već prije svega podučiti tehnike objektno-orijentiranog programiranja. Pretpostavlja se da su studenti savladali programiranje u jeziku C i da su su upoznati s osnovama jezika UML.

Pri savladavanju nove, objektne paradigme student kao prvo treba dobro razumjeti osnovna svojstva i mogućnosti objekta. Isto tako za mnoge elementarne probleme postoje standardna rješenja koja je razvijateljska zajednica prihvatila kao najbolja. Sva takva rješenja treba usvojiti umjesto "otkrivati toplu vodu" vlastitim pristupima. Početnik treba biti svjestan da još nije došao u poziciju da ima puni pregled nad razlozima zašto je neko rješenje bolje od drugog.[1] Također, tendencija studenata je da se za rješavanje nekih od problema oslanjaju na već poznate tehnike iz jezika C i imaju velik otpor pri usvajanju tehnika uobičajenih za objektno–orijentirane jezike tj. Javu.

Učenje programskog jezika u mnogočemu podsjeća na učenje prirodnog jezika. Iz istog razloga zašto nitko još nije naučio npr. engleski jezik isključivo čitajući gramatiku i rječnik, nitko nije i neće naučiti Javu isključivo čitajući ovu skriptu i slušajući predavanja. Neizostavna aktivnost jest *trening*, a to u ovom kontekstu znači vlastoručno pisanje koda iz nule, uz nadzor asistenta koji zadaje zadatke, proučava vaša rješenja, korigira vas i usmjerava. Kriterij uspjeha nije krajnji proizvod sadržan u datotekama `.java`, nego vještina koju ste stekli. Zadatak je tek sredstvo, primjer na kojem se može najbolje utvrditi razina vaše vještine i, ponajprije, pratiti podizanje te razine tijekom semestra.

[1] Detaljnija argumentacija ovoga krije se iza fraze "shu-ha-ri" (usvajanje pravila-kršenje pravila-napuštanje pravila) o kojoj se može informirati na Webu, npr. `c2.com/cgi/wiki?ThreeLevelsOfAudience`

Iako nije dovoljna sama za sebe, ova skripta ima važnu ulogu u čitavom procesu učenja: da biste uopće mogli komunicirati s nekim poznavateljem Jave, morate već razumjeti određeni skup pojmova i stručnih izraza. Ako asistent kaže npr. "ovaj atribut ne bi trebao biti statički jer mu je vrijednost različita za svaku instancu", onaj tko nije ni pročitao skriptu neće imati koristi od takvog savjeta jer ga naprosto neće razumjeti. Jednako tako, prilikom ispitivanja dogodit će vam se da ne razumijete postavljeno pitanje, iako biste možda čak i znali odgovor.

Na kraju, želim se izjasniti i po pitanju jedne vrlo proširene predrasude: pisanje dobrog koda nije nikakvo mehanizirano "štrikanje", to je intelektualno zahtjevan i kreativan proces sa svojom vlastitom estetikom. Ako netko smatra drugačije, to je najvjerojatnije stoga što nije još razvio estetske kriterije da bi uvidio razliku između lijepog, kreativnog i ružnog, nabacanog koda. Ta činjenica predstavlja značajnu prepreku: učenik prvo treba skupiti motivaciju da "umoči noge" da bi uopće došao u poziciju da cijeni vlastiti rad, a tko ne cijeni ono što radi ne može ni imati dobro mišljenje o svojoj struci.

Uvod

Skripta se odnosi na Javinu verziju 5 (*Java 2 Platform Standard Edition 5.0*) i UML verziju 1.5. Sadržaj ove skripte bit će izlagan na predavanjima, tako da se studentima preporuča da unaprijed pročitaju (barem prolistaju) poglavlje čija će se materija izlagati na sljedećem predavanju.

U skripti je obrađen samo osnovni podskup jezika.

Također je iznimno važno imati na umu da je i materija koja se obrađuje u skripti obrađena samo sa svrhom uvoda u Javu i UML za početnika – nikako se ne smije shvatiti kao potpuna informacija. Gotovo sve ovdje rečeno predstavlja drastično pojednostavljenje u odnosu na preciznu i potpunu istinu, sve u svrhu lakše razumljivosti početniku. Ako je čitatelj već poznavatelj Jave i takva ga pojednostavljenja smetaju, neka slobodno odloži skriptu jer njemu ona i ne treba.

Studentu se preporuča da prva dva poglavlja pročita u cijelosti jer se tamo spominje velik broj najvažnijih pojmova, a tekst je pisan kao jedna priča u kojoj se pojmovi uvode gdje koji zatreba. Tekst daljnjih poglavlja nije toliko usko povezan pa čitanje po redu nije tako bitno.

Poglavlje 1

Uvod u UML

UML (Unified Modeling Language) je grafički jezik koji služi za specifikaciju, vizualizaciju, izradu i dokumentiranje objektno–orijentiranih programskih rješenja [10, 3]. Osnovne konstrukte jezika je moguće vrlo brzo naučiti i primijeniti te se kao takvi nalaze u mnogim knjigama koje se bave objektno orijentiranom analizom, dizajnom i programiranjem. UML nije ovisan o procesu ili programskom jeziku koji se koristi za izradu programskog rješenja.

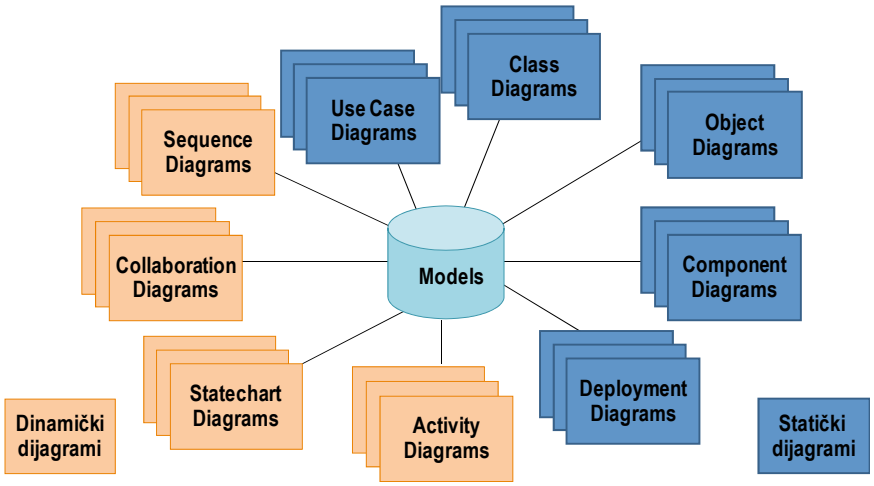
Nastanak UML-a može se promatrati još negdje od početka 70-ih godina dvadesetog stoljeća. Naime, od tada pa sve do kraja 80-ih godina dvadesetog stoljeća ukazivala se sve veća potreba za objektno–orijentiranim jezicima. Na razvoj UML-a imalo je utjecaj i nekoliko drugih jezika kao što su Entity Relationship modeling, SDL (Specification & Description Language), te drugi. Broj objektno–orijentiranih jezika narastao je s 10 na više od 50 u periodu od 1989. do 1994. godine.

Razvoj samog UML-a počeo je u listopadu 1994. godine kada su Grady Booch i Jim Rumbaugh, članovi Rational Software Corporationa, započeli posao na ujedinjavanju dotadašnjih metoda: Booch – metoda (Booch '93) i metoda OMT-a (Object Modeling Technique). Ove su se dvije metode već duži niz godina zasebno razvijale i imale su vodeću ulogu u dotadašnjem razvoju objektno – orijentiranih jezika. U listopadu 1995. godine izlazi radna verzija ovog projekta pod nazivom Unified Method 0.8. U jesen 1995. godine ovoj već spomenutoj dvojici pridružuje se Ivar Jacobson sa svojom tvrtkom Objectory Company. Njih trojica "tri amigosa" stvorili su jezik UML. Prve verzije su su bile 0.9 i 0.91 u lipnju odnosno listopadu 1996. godine. Nakon izlaska verzije 1.0 u 1997. godini velike informatičke tvrtke

podržale su UML te je standardizacija UML-a povjerena OMG konzorciju u kojem su sve velike informatičke tvrtke [9]. Od tada pa do danas svakih par godina izlazi nova dorađena verzija UML-a (1.1 do 1.5). Bitna prekretnica dogodila se izlaskom UML-a 2.0 koji danas omogućuje puno veću fleksibilnost i prilagodbu UML-a različitim domenama primjene. U ovom kolegiju koristi se UML verzije 1.x jer je najrašireniji i većina konstrukcija koji će se koristiti za vrijeme programiranja u Javi može se dobro opisati ovom verzijom UML-a.

1.1 Dijagrami

Osnovni elementi UML-a su dijagrami. Svaki dijagram opisuje sustavi ili njegov dio s jednog stajališta (slika 1.1). Dijagrami se dijele na statičke i dinamičke. Statički dijagrami su oni koji u dijagramu ne pokazuju vrijeme. Dinamički dijagrami pokazuju vrijeme u dijagramu odnosno pokazuju kako se "nešto" odvija u vremenu.



Slika 1.1: UML dijagrami

Dijagram slučaja uporabe (*Use Case Diagram*) prikazuje odnose između korisnika i sustava. Korisnik ne mora biti osobe, već može biti i drugi sustav ili program. Ovaj dijagram se izrađuje u početnoj fazi projekta, opisuje funkcionalno ponašanje sustava u interakciji sa sudionicima u sustavu. U njemu se na sustav gleda kao crna kutija i ne zanima nas što se unutra nalazi, već su bitne samo veze sustava s vanjskim svijetom. Treba još istaknuti da se gotovo ne može niti zamisliti situacija u kojoj se ne bi koristio

ovaj dijagram. Zbog svoje jednostavnosti i učinkovitosti on je nezamjenjiv dio razvojne faze izrade projekata i vrlo brzo ga shvate osobe koji nisu programeri (naručitelji).

Dijagram klasa (*Class Diagram*) opisuje klase u sustavu i različite vrste veza među njima. Dijagram klasa također prikazuje operacije (u Javi metode) i atribute klasa. Uobičajeno je da se prvo u dijagramu nacrta nekoliko klasa i veze među njima (konceptualni dijagram), a kako se paralelno rade drugi dijagrami tako se dijagram klasa nadopunjuje detaljima kao što su dodatne klase, veze, atributi i operacije. Ovaj dijagram je jedan od najkorisnijih za programera i često ga nalazimo u knjigama o programiranju kako opisuje različite objektno orijentirane koncepte.

Dijagrami interakcije opisuju interakciju između objekata u sustavu tako da prikazuju objekte i poruke koje razmjenjuju te njihov redoslijed. Oni obično predstavljaju ponašanje sustava u jednom slučaju uporabe. U dijagrame interakcije spadaju dva dijagrama: slijedni dijagram (*Sequence Diagram*) i dijagram međudjelovanja (*Collaboration Diagram*). Ako imamo jedan dijagram onda iz njega možemo konstruirati drugi dijagram jer pokazuju iste informacije na različit način.

Slijedni dijagram (*Sequence Diagram*) opisuje dinamičku suradnju između objekata. On vrlo zorno prikazuje vremenski slijed poruka koje objekti razmjenjuju, te vrijeme života pojedinih objekata (kreiranje i uništavanje). Nedostatak mu je to što se teško vide relacije između objekata. Ovaj dijagram se isto kao i dijagram klasa često koristi prilikom programiranja i u knjigama.

Dijagram međudjelovanja (*Collaboration Diagram*) isto tako opisuje dinamičku suradnju između objekata, ali vrlo zorno se vide objekti koji surađuju i veze među njima. Vremenski redoslijed je prikazan brojevima na porukama što je teško pronaći u dijagramu.

Dijagram stanja (*Statechart Diagram*) prikazuje dinamičko ponašanje u sustavu. Temelji se na konceptima automata tj. ima stanja i prijelaze te obično prikazuje stanja jednog objekata. U dijagramu se opisuju sva moguća stanja u koje neki objekt može dospjeti i kako se ta stanja mijenjaju kao posljedica vanjskih događaja. Nedostatak ovakvog pristupa je što se već kod malo složenijih objekata dolazi do eksplozije stanja i prijelaza pa zato treba dobro procijeniti isplati li se ovaj dijagram koristiti.

Dijagram aktivnosti (*Activity Diagram*) opisuje redoslijed aktivnosti kako slijednih, tako i paralelnih. Ovi dijagrami su posebno korisni prilikom

modeliranja ili opisa paralelnih aktivnosti. Isto tako su vrlo korisni u prikazu algoritama, analizi slučaja uporabe. Ovaj dijagram ima korijene u dijagramima događaja [8], jeziku SDL (*Specification and Description Language*) [7], modeliranju hodograma (*workflow*) [11] i Pertijevim mrežama [2].

Fizički dijagrami su dijagrami koji opisuju kako je programsko rješenje isporučeno i od koji h je komponenti složeno te kako je sve to povezano u jedan sustav. U fizičke dijagrame spadaju dijagram isporuke (*Deployment Diagram*) i dijagram komponenti (*Component Diagram*). U dijagramu isporuke se opisuju fizičke veze softvera i hardvera u isporučenom sustavu. Dobar je za prikaz kako komponente mogu biti distribuirane u sustavu. Dijagram komponenti opisuje komponente sustava i njihovu ovisnost. Oba dijagrama se najčešće koriste kao jedan tj. simboli jednog i drugog se koriste na jednom dijagramu.

U ovom poglavlju nećemo ulaziti u detalje svakog dijagrama, već ćemo kroz sljedeća poglavlja uvoditi dijagrame po potrebi.

Poglavlje 2

Osnovni elementi jezika Java

Početak razvoja na Javi je bio 1991. godine kada je Sun Microsystems pokrenuo projekt razvoja novog programskog jezika (Green Project) koji je namijenjen za male uređaje (npr. preklopnik za kablovsku televiziju). Cilj je bio stvoriti jezik koji će trošiti malo memorije, raditi na slabim procesorima i to različitim. Razvoj je započeo s modifikacijom UCSD Pascala [1]. Generirani koda (nakon prevođenja) je napravljen da bi se izvodio na virtualnom stroju koji ne postoji, a stvarni stroj bi imao ugrađen interpreter koji bi interpretirao takav kod (*byte code*). Takav interpreter bi bio jednostavan i brz jer bi se jednostavno preslikavao na strojne instrukcije stvarnog procesora, a s druge strane dobilo se na fleksibilnosti izvođenja jer bi svaki stroj koji ima interpreter mogao izvoditi takav program. Ova je revolucionarna ideja se danas koristi i u drugim programskim jezicima kao što su C#.

Nakon toga je jezik oblikovan tako da je sintaksa slična C++-u jer je puno programera naviknuto na C i C++ pa im takva sintaksa nije strana. Prve verzije jezika su se zavale Oak, a onda je preimenovan u Java. Prvi program u Javi se zvao *7, a to je bio program za inteligentni daljinski upravljač za televizor. Nakon toga su se nizali programi za različite usluge kabelske televizije. 1993. godine počinje intenzivna potraga za kupcima, a 1994. Sun razvija HotJava preglednik za pregledavanje web stranica. To je bio prvi preglednik koji je mogao izvršavati Javine applete (programe koji se izvršavaju u pregledniku unutar neke web stranice). HotJava je predstavljen na konferenciji SunWorld 1995. godine. Iste godine Netscape licencira Javu i 1996. izlazi Netscape 2.0 koji ima podršku za Javu. Nakon toga Ja-

[1] `http://en.wikipedia.org/wiki/UCSD_Pascal`

vu licenciraju i ostale važne tvrtke kao što su: IBM, Symantec, Borland (Inprise), Microsoft, itd. Početkom 1996. izlazi prva službena verzija Jave (v1.0) i u svibnju iste godine započinje serija konferencija JavaOne koja promovira sve što se tiče budućnosti Jave.

Prva bitna nadogradnja je bile Java 1.2. odnosno Java 2 kada je ugrađen Swing programski okvir koji je omogućio grafičko sučelje i elemente potpuno napisane u Javi tj. neovisne o operacijskom sustavu. Nakon toga Microsoft izdaje .Net platformu i jezik C# koji je vrlo sličan Javi. Sljedeću veliku nadogradnju Jave je potaknuo C# tako da su u Javi 5 odnosno Javi 1.5 (što su zapravo dva imena za istu stvar) ugrađeni mehanizmi automatskog pretvaranja primitivnih vrsta podataka, *generics*, enumeracije, itd., a što je već bilo ugrađeno u C#-u. Trenutna verzija Jave je 1.6 odnosno 6. 2007. godine je Sun otvorio kod Jave, a rezultat je puno stabilnija verzij za razliku od svojih prethodnika jer je "vojska" programera pronašla većinu grešaka prije službenog izlaska verzije 6. Trenutno se vode žestoke rasprave što treba, a što ne treba ugraditi u novu verziju Jave. Postoje dvije struje, jedna zagovara jednostavnost jezika, a druga ugradnju novih mogućnosti i novih konstukcija (npr. klozure odnosno *closure*).

2.1 Svojstva jezika Java

U počecima je jezik bio jednostavan jer su izbačene konstrukcije i mehanizmi koji su se koristili u C-u i C++-u, a koji su dovodili do grešaka. To su:

- `typedef` i `define` tj. predprocesor,
- strukture i unije koje se mogu realizirati punopravnim klasama,
- funkcije koje se mogu realizirati statičkim metodama na razini klase,
- višestruko nasljeđivanje gdje se dio konstrukcija može realizirati sučeljima,
- goto naredba,
- preopterećivanje operatora i
- pokazivači, a umjesto njih se koriste reference.

Robusnost Jave se temelji na mehanizmima rane i kasne dinamičke provjere koda za vrijeme izvršavanja te stroge sintakse (*strong typed language*) i njene provjere za vrijeme prevođenja. Kod Jave eliminirana svaku mogućnost da dvije varijable posjeduju istu memorijsku lokaciju što sprječava mnoge greške prilikom programiranja. U Javi je ugrađena provjera indeksa polja i niza znakova koja se u slučaju pogrešnog korištenja bacaju iznimke. Iznimke (*exceptions*) su mehanizam za obradu pogrešaka.

Java je dinamičan jezik što znači da za vrijeme rada (dinamički) može učitavati kod programa s diska ili preko mreže i izvršavati ga. Ovo svojstvo je iskorišteno u distribuiranim sustavima u Javi kao što su applet-i, Java RMI (*remote methos invocation*) i sl. Isto tako je bitno da je prevedeni Javin program spremeljen u više datoteka na disku tj. svaka klasa je smještena u svoju datoteku. Ovo svojstvo omogućuje da se prilikom prevođenja prevedu samo klase koje su u međuvremenu promijenjene. Za razliku od C++-a u kojem postoji problem stalnog prevođenja (*fragile superclass problem*) gdje se svaki put moraju sve klase u programu ponovno prevoditi.

Sigurnost u Javi je ugrađena pomoću različitih mehanizama koji jedan drugoga potpomažu. Izbacivanjem pokazivača i korištenjem referenci izbjegnut je neovlašten pristup memoriji. Prije izvršavanja oktetnih kodova (eng. *bytecode*) vrši se provjera poštuje li taj kod određena sigurnosna pravila. Ovisno o tome s kojeg mjesta (lokalni disk, mreža) je kod učitan i je li potpisan i od koga takav program onda ima ili nema određena prava pristupa.

2.2 Prvi primjer

Pogledajmo prvi program napisan u Javi (Ispis 2.1).

Ispis 2.1: Prvi program

```
1 public class FirstExample
2 {
3   public static void main(String[] args)
4   {
5     System.out.println("Ovo radi!!");
6   }
7 }
```

Program počinje definiranjem klase `FirstExample` (linija 1). Prvo se definira dostupnst klase koja je u ovom slučaju javna (`public`). Nakon toga

slijedi ključna riječ `class` koja označava da se radio o klasi. Iza toga se stavlja naziv klase (`FirstExample`). Nakon toga dolazi tijelo klase koje se stavlja u vitičaste zagrade (linije 2–7). Tijelo klase se može sastojati od atributa i metoda. U ovom slučaju imamo samo metode i to jednu (`main`). U lijini 3 je deklarirana metoda `main`. Ona je posebna metoda koja služi za pokretanje programa. Ima jedan parametar `args`. To je polje objekata `String` koje predstavlja parametre pokretanja. Svaka metoda započinje i završava vitičastim zagradama (linije 4 i 6). U tijelu metode je linija 5 koja ispisuje tekst "Ovo radi!!". Za sada je dovoljno zapamtiti da konstrukcija `System.out.println` ispisuje tekst koji je proslijeđen kao parametar.

2.2.1 Rad u naredbenom retku

Nekim editorom teksta treba stvoriti datoteku s izvornim kodom. Kod treba staviti u datoteku `FirstExample.java`. Opće pravilo je da ime datoteke mora biti identično imenu klase + sufiks `.java`. Nakon toga ga treba prevesti ("kompajlirati") u byte kod. Da bismo mogli prevoditi ili pokretati programe u Javi potrebno je skinuti i instalirati JDK (Java Development Kit). Prevođenje se može pokretati u naredbenom retku (Command Prompt u operacijskom sustavu Windows ili Terminal u Unixoidnim operacijskim sustavima). Datoteku se prevodi ("kompajlira") Javinim alatom `javac` (*Java compiler*):

```
> javac FirstExample.java
```

Rezultat ovog koraka je prevedeni kod u datoteci `FirstExample.class`. To *nije* izvršna datoteka koju se može izravno pokrenuti, već je *klasna datoteka* (eng. *class file*) koja sadrži binarnu definiciju klase u posebnom Javinom *međukodu* (eng. *bytecode*, vidjeti odjeljak 3.13). To je izvršni kod za tzv. *Javin virtualni stroj* (eng. *Java Virtual Machine* odnosno JVM) – posebnu hardversku platformu koja postoji samo "na papiru", a svaka realna hardverska platforma ju *emulira*. Sljedeći korak je pokretanje emulatora tj. JVM-a kojem treba proslijediti ime naše izvršne klase:

```
> java FirstExample
```

Važno! Kao argument ove naredbe ne navodi se ime klasne datoteke, već samo ime klase. To će pogotovo biti važno kasnije, kad uvedemo *pakete* (odjeljak 2.13).

Kada pokrenemo program on će ispisati sljedeće u komandnoj liniji:

```
Ovo radi!!
```

2.2.2 Rad u okružju *Eclipse*

Kako pokrenuti ovaj primjer u *Eclipse*-u (vrijedi za verziju 3.1):

1. File, New, Project... Java Project, Next
2. Upisati ime projekta, uvjeriti se da je uključeno "Create separate source and output folders", Finish
3. U okviru Package Explorer naći projekt, ekspandirati ga, namjestiti oznaku na direktorij src.
4. File, New, Class. Upisati ime klase. U editorskom okviru otvara se stvorena datoteka i u njoj već piše prvi redak koda. Upisati ostatak koda.
5. Pokretanje klase: desni klik na izvršnu klasu u Package Explorer-u, Run As, Java Application. Ili, dok je kursor u editoru dotične klase, upotrijebiti kombinaciju tipaka Alt-Shift-X, J.
6. U okviru Console vidi se ispis programa.
7. Ako bilo koji od okvira nije vidljiv, može ga se otvoriti pomoću Window, Show view

2.3 Parametri naredbenog retka

Ako želimo napraviti program koji nas "pozdravlja" tako da mu pošaljemo ime u naredbenom retku onda to izgleda kao u ispisu 2.2.

Ispis 2.2: Parametar komandne linije

```
public class ArgumentExample
{
  public static void main(String[] args)
  {
    System.out.println("Lijepi pozdrav " + args[0] + "!");
  }
}
```

Razlika je u nazivu klase koja se sada zove `ArgumentExample` i u liniji u kojoj ispisujemo tekst imamo sljedeće: `"Lijepi pozdrav " + args[0] + "!"`. Ovaj dio koda spaja tekstualni niz u jedan. Prvi dio niza je: Lijepi pozdrav. Na taj dio se spaja argument koji je poslan kao parametar prilikom pokretanja i na njega je spojen znak !. Dakle, kod `args[0]` predstavlja prvi element polja (indeksi počinju od 0).

Sada ga trebamo prevesti i onda pokrenuti sljedećom linijom:

```
> java ArgumentExample Tomislav
```

Ispis je:

```
Lijepi pozdrav Tomislav!
```

Sljedeći program (Ispis 2.3) ispisuje brojeve od 1 do 9.

Ispis 2.3: Petlja

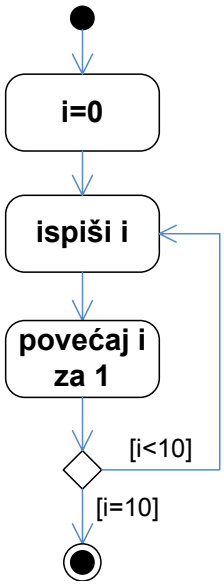
```
public class LoopExample
{
  public static void main(String[] args)
  {
    for(int i = 0; i<10; i++)
    {
      System.out.println(i);
    }
  }
}
```

Ovdje vidimo da se ispisivati mogu i brojevi. Ispis je:

```
1
2
3
4
5
6
7
8
9
```

Algoritmi, pa i petlje, se mogu lijepo prikazati dijagramom aktivnosti (slika 2.1). Dijagram aktivnosti se sastoji od aktivnosti i prijelaza između njih. Svaki dijagram mora imati simbol početka koji je prikazan crnom točkom (zapravo je mali ispunjeni krug). Početak mora biti povezan (prijelaz) s prvom aktivnošću. Aktivnost se prikazuje pravokutnikom koji ima zaobljene kutove. Prijelaz iz početnog stanja i aktivnosti se prikazuje strelicom.

Ukoliko imamo grananja ona se prikazuju simbolom romba. U taj simbol mora ulaziti barem jedna strelica i izlaziti barem dvije. Izlazne strelice moraju imati uvjete koji se zapisuju u uglatim zagradama. Ako algoritam ima kraj onda se kraj označava točkom oko koje je krug. Primjer algoritma s ispisa 2.3 prikazan je slikom 2.1.



Slika 2.1: Dijagram aktivnosti za metodu `main` iz klase `LoopExample`

2.4 Objektna paradigma, enkapsulacija

U objektno-orijentiranom programskom jeziku izrada programa može se u osnovi svesti na izgradnju *sustava objekata*. Objekti su međusobno povezani ("znaju" jedan za drugog) i međusobno komuniciraju razmjenom *poruka*, što se svodi na to da jedan objekt poziva *metode* drugoga. Dakle, pri kretanju u rješavanje zadanog problema prvo treba osmisliti koji objekti će biti potrebni i kako će biti povezani. Za to se koristi jezik UML. Ovaj proces nije linearan već zahtjeva sveobuhvatno promišljanje o sustavu i postepeno detaljiziranje. Često je moguće postići da programski objekti modeliraju "stvarne" objekte. Za svaki objekt najvažnije je utvrditi što će se od njega očekivati da radi, tj. njegovo *ponašanje*. U praksi to se svodi na popis metoda koje će on posjedovati. Za svaku metodu najbitnije je koje argumente prima i kakvu vrijednost vraća. Također je bitno na koji način poziv metode utječe na *stanje* objekta. O stanju ovisi kako će objekt odgovarati

na buduće pozive metoda. Dakle, identičan poziv metode u različitim trenucima može vratiti različitu vrijednost, ovisno o stanju u kojem je objekt zatečen. Stanje objekta u praksi se svodi na vrijednosti njegovih *atributa* (unutarnjih varijabli, na engleskom se u kontekstu Jave često koristi i izraz *field*). Osnovno načelo dizajna objekata – *enkapsulacija* ili *zatvaranje* – nalaže da stanje objekta mora ostati skriveno, a izvana trebaju biti vidljive samo one *posljedice* tog stanja koje su zaista potrebne – dakle utjecaj stanja na povratne vrijednosti metoda. Evo jednostavnog primjera opisa objekta u Javi:

Ispis 2.4: Klasa `SimpleCounter`

```
public class SimpleCounter
{
  private int counterState;

  public void increment()
  {
    this.counterState++;
  }

  public int getState()
  {
    return this.counterState;
  }
}
```

Naš objekt opisan klasom `SimpleCounter` raspolaže dvjema metodama, od kojih jedna mijenja njegovo stanje (`increment`), a druga to stanje dojavljuje "vanjskom svijetu" (`getState`). Dakle, rezultat metode `getState` je ovisan o trenutnom stanju objekta. Stanje je pohranjeno u atributu `counterState`. To je običan cijeli broj (vrste `int`, poznat iz jezika C). Pri kreiranju objekta svi brojevni atributi se implicitno inicijaliziraju na vrijednost nula.

Važno je da svi atributi imaju oznaku `private`, čime se ostvaruje načelo skrivenosti stanja odnosno *enkapsulacije*. U protivnom bi ih objektu "iza leđa" mogli mijenjati drugi objekti.

2.4.1 Dijagram klasa

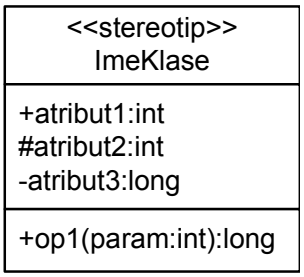
Kao što je već prije spomenuto dijagram klasa prikazuje statičke veze između klasa. Simbol za klasu u dijagramu klasa je pravokutnik koji je podijeljen na 3 dijela (slika 2.2). U prvom dijelu se prikazuje naziv klase i

stereotip. Stereotip se, između ostalog, koristi za označavanje sučelja (poglavlje 2.10). Srednji dio se koristi za prikaz atributa. Sintaksa označavanja atributa takva da je prvi dio razina dostupnosti iza koje dolazi naziv atributa, zatim dolazi znak ":" i iza njega vrsta atributa. Razine dostupnosti mogu imati četiri vrijednosti:

- + javna razina (*public*),
- **-** privatna razina (*private*),
- # zaštićena razina (*protected*) i
- **bez oznake** paketno privatna razina (*package–private*).

Treći dio prikazuje metode. U sintaksi prikaza metode, prvo je razina dostupnosti pa naziv metode iza koje dolaze zagrade i na kraju znak ":" te vrsta podataka koju vraća metoda. Unutar zagrada se nalazi lista parametara metode. Svaki parametar je odvojen zarezom, a sastoji se od imena parametra, znaka ":" i vrste parametra.

Dio s atributima ili s metodama se može izostaviti. Isto tako se mogu izostaviti neki njihovi dijelovi. Sve ovisi o tome što želimo prikazati tj. na čemu je naglasak.

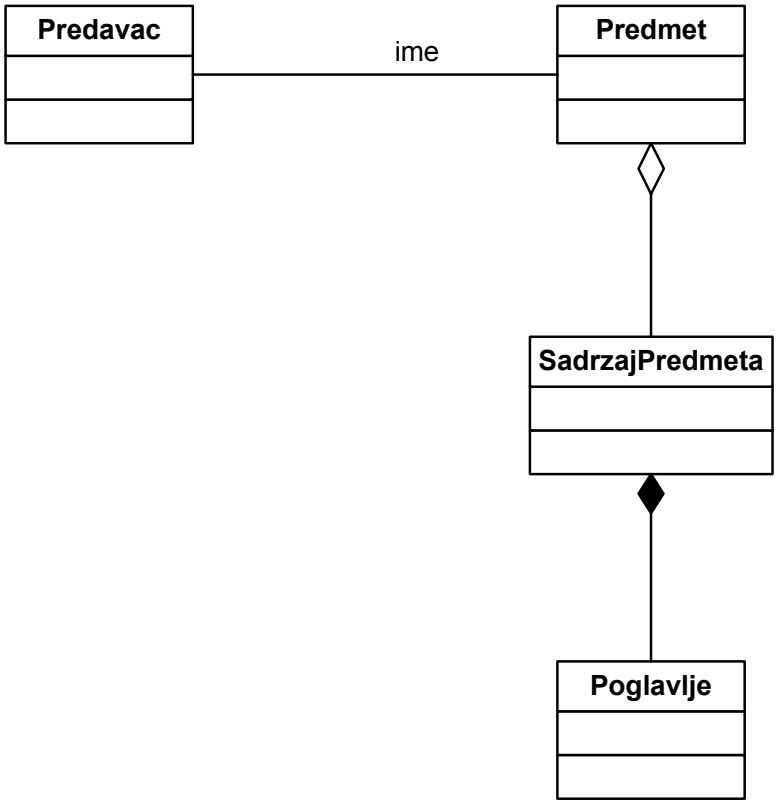


Slika 2.2: Dijagram klasa – simbol klase

Relacije u dijagramu klasa prikazuju različite veze između dvije klase. One se prikazuju linijama koje povezuju dva simbola za klasu. Linije mogu biti isprekidane ili pune, a na krajevima linija mogu se nalaziti različiti simboli. Ovisno o tome kakva je linija i kakvi su simboli, definirane su različite relacije.

Pogledajmo sada dijagram na slici 2.3. Ovdje su definirane tri klase: `Predavac`, `Predmet`, `SadrzajPredmeta` i `Poglavlje`. Klasa `Predavac` je povezana s klasom `Predmet` relacojim asocijacije. Asocijacija opisuje najopćenitiju vezu

između klasa. Asocijacija može biti jednosmjerna ili dvosmjerna. U ovom slučaju je dvosmjerna jer nema nikakvih oznaka na krajevima. Kada bi asocijacija bila jednosmjerna prema klasi `Predmet` onda bi na kraju linije prema klasi `Predmet` bila nacrtana strelica. Jednosmjernost ne znači da je komunikacija između klasa jednosmjerna, već da samo jedna klasa može započeti komunikaciju i ta komunikacija započinje u smjeru strelice. Svaka veza može imati ime koje se piše u sredini kao što je asocijaciji na slici.



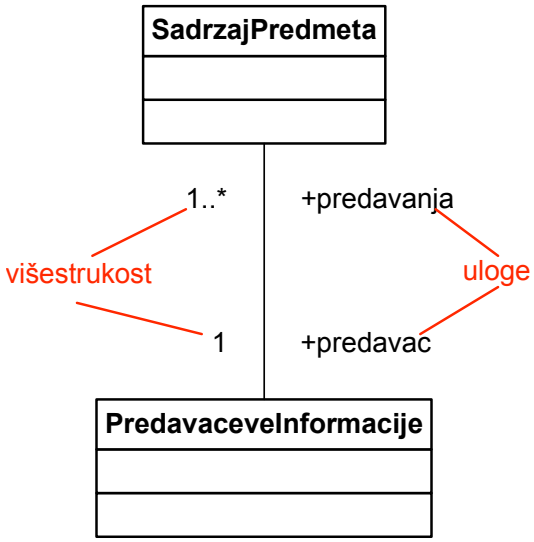
Slika 2.3: Dijagram klasa – osnovne relacije

Druga vrsta veze je agregacija. Ona za razliku od asocijacije ima na jednom kraju linije romb (neispunjeni). Kod agragacije jedna klasa sadrži drugu i bez nje ne može funkcionirati. U primjeru na slici klasa `Predmet` sadrži klasu `SadrzajPredmeta`.

Relacija kompozicije je slična agregaciji i po izgledu i po funkciji. Kompozicija se crta s ispunjenim rombom i isto kao i kod agregacije jedna klasa (uz romb) sadrži drugu i bez nje ne može funkcionirati. U primjeru na slici klasa `SadrzajPredmeta` sadrži klasu `Poglavlje`. Razlika je u tome što je kod kompozicije objekt napravljen iz klase `SadržajPredmeta` jedini koji ima refe-

rencu na objekt napravljen iz klase `Poglavlje`. To ne znači da drugi objekti iz drugih klasa ne mogu imati referencu na objekt iz klase `Poglavlje`, već znači da na objekt iz klase `Poglavlje`, koji je sadržan u objektu iz klase `SadrzajPredmeta`, niti jedan drugi objekt nema referencu.

Ove tri osnovne relacije mogu imati definirane i uloge (slika 2.4). Uloge se označavaju na kraju relacije koji se spajaju na simbole klasa. Uloga objašnjava svrhu relacije i ima definiranu višestrukost i vidljivost. Vidljivost se definira neposredno ispred imena uloge, a višestrukost oznakom pored imena (obično s druge strane relacije). Višestrukost se označava na nekoliko načina. Vrijednost višestrukosti može biti bilo koji broj ili znak *. Oni se mogu kombinirati kao pojedini podaci npr. `2,4` označava ili 2 ili 4, te se mogu kombinirati u raspon npr. `2..4` što označava vrijednosti od 2 do 4. Vrijednost `*` označava 0 ili više tako da kombinacija `2..*` označava od 2 na više, a samo oznaka `*` označava 0 ili više.

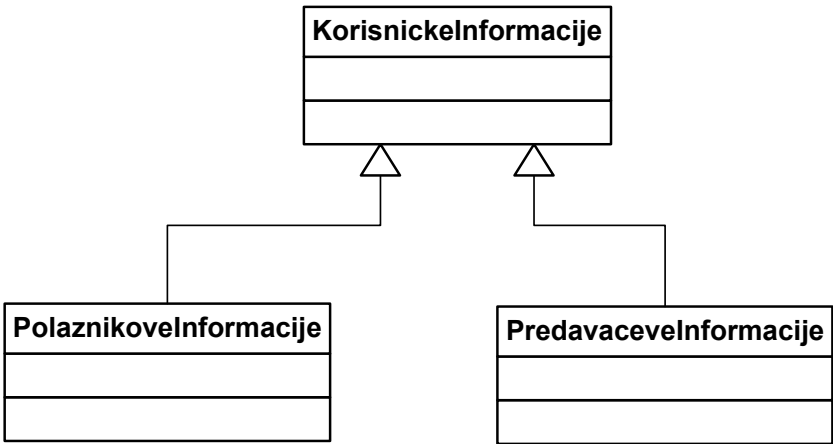


Slika 2.4: Dijagram klasa – uloge

U primjeru na slici 2.4 jedan ili više objekata klase `SadrzajPredmeta` je vezano na jedan objekt klase `PredavaceveInformacije`. U objektu klase `PredavaceveInformacije` je javni atribut `predavanja` koji sadrži jedan ili više objekata klase `SadrzajPredmata`. Ovaj atribut može primjerice biti realiziran poljem objekata klase `SadrzajPredmeta`.

Relacija nasljeđivanja na jednom kraju ima trokut (slika 2.5). Nasljeđivanje je relacija između dvije klase koje dijele strukturu i/ili ponašanje. U primjeru na slici klasa `PolaznikoveInformacije` nasljeđuje klasu

`KorisnickeInformacije`. U toj relaciji je klasa `KorisnickeInformacije` nadklasa. Umjesto pojma nadklasa se još koristi i pojam klasa roditelj. S druge strane klasa `PolaznikoveInformacije` je podklasa, a još se koriste i pojmovi: klasa dijete, derivirana klasa ili izvedena klasa. Različiti pojmovi se koriste u različitim programskim jezicima, a zapravo su sinonimi. Dakle, možemo reći da podklasa nasljeđuje nadklasu.

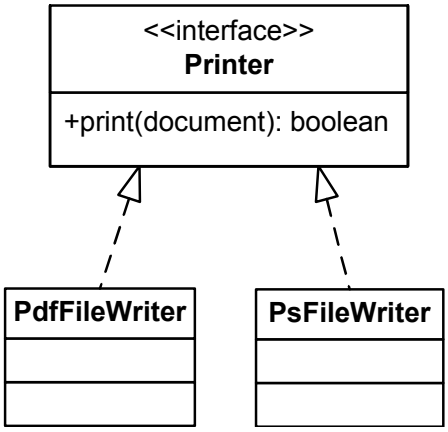


Slika 2.5: Dijagram klasa – nasljeđivanje

Umjesto pojma nasljeđivanje koristi se i pojam generalizacije ili pojam specijalizacije. Koji pojam se koristi ovisi samo o pogledu na klase. Primjerice ako pogledamo klasu `PolaznikoveInformacije` onda je klasa `KorisnickeInformacije` općenitija, odnosno generalnija, klasa pa je relacija između njih generalizacija. Ako pak pogledamo klasu `KorisnickeInformacije` onda je u odnosu na nju klasa `PolaznikoveInformacije` specifičnija klasa pa je relacija između njih specijalizacija. Ako pak idemo programirati te dvije klase onda nam je logično da klasa `PolaznikoveInformacije` nasljeđuje klasu `KosinickeInformacije` pa je relacija između njih nasljeđivanje.

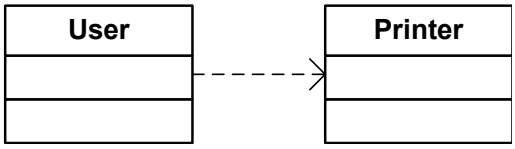
Sljedeća relacija je realizacija (slika 2.6). Realizacija je slična nasljeđivanju samo što se kod realizacije nasljeđuju samo metode s parametrima bez njihove implementacije dok kod se kod nasljeđivanja nasljeđuju i implementacije metoda. Realizirati se mogu samo sučelja (*interface*).

U primjeru na slici 2.6 prikazano je sučelje `Printer` koje je označeno simbolom klase sa stereotipom `interface`. Ovo sučelje je realizirano u dvije klase: `PdfFileWriter` i `PsFileWriter`. Realizacija je prikazana istim simbolom kao i nasljeđivanje, jedino je razlika što realizacija ima isprekidanu liniju, a nasljeđivanje ima punu liniju.



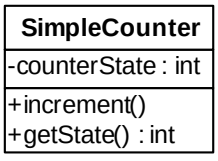
Slika 2.6: Dijagram klasa – realizacija

Zadnja relacija je relacija korištenja ili ovisnosti (slika 2.7). Ova relacija se koristi kada metoda u nekoj klasi koristi objekt iz druge klase. U primjeru na slici, metoda u klasi `User` koristi objekt iz klase `Printer` i samim tim klasa `User` ovisi o klasi `Printer`. Simbol korištenja je strelica s isprekidanom linijom.



Slika 2.7: Dijagram klasa – korištenje / ovisnost

Vratimo se na klasu `SimpleCounter`. UML-ov dijagram klase `SimpleCounter` prikazan je na slici 2.8. Iz njega se vidi da klasa ima privatni atribut `counterState` koji je vrste `int`. Isto tako vidimo da klasa ima dvije metode: `increment` i `getState`. Obje metode su javne i nemaju parametara. Metoda `getState` vraća podatak vrste `int`.



Slika 2.8: UML-ov dijagram klase `SimpleCounter`

Pogledajmo sada malo drugačiju klasu, `ModuloCounter`, čija metoda `increment` ustvari izgleda ovako:

Ispis 2.5: Klasa `ModuloCounter` s drugačijom metodom `increment`

```
public class ModuloCounter
{
  public void increment()
  {
    if ( this.counterState == 9 )
      this.counterState = 0;
    else
      this.counterState++;
  }

  ... atribut counterState i metoda getState isti kao u SimpleCounter
}
```

Dakle, brojač cirkularno broji samo do devet i onda se vraća na nulu. Kad atribut `counterState` ne bi bio privatan, što znači da se može koristiti samo unutar te klase, mogao bi se izvana promijeniti na proizvoljnu vrijednost, npr. 10. Time bi se narušila funkcionalnost objekta jer bi se, kao prvo, brojač našao u nedozvoljenom stanju i, kao drugo, daljnji pozivi metode `increment` nastavili bi brojati prema beskonačnosti.

Enkapsulacija, iako ju sam jezik Java ne nameće, uglavnom je neizostavan dio dizajna objekta jer sprečava velik broj programerskih pogrešaka uz najčešće malu ili nikakvu žrtvu u eleganciji koda. Također daje programeru veću slobodu u eventualnom redizajniranju objekta. Imena i podatkovne vrste (tipovi) atributa mogu se proizvoljno mijenjati dok god se zadržava isto *ponašanje* objekta kakvo se očituje u vrijednostima koje vraćaju metode.

2.5 Pokretanje programa pisanih u Javi

U jeziku Java program se pokreće tako da se prvo definira *izvršnu* klasu koja sadrži posebnu metodu `main`. Ona će biti implicitno pozvana kad "izvršimo" klasu, tj. pokrenemo svoj program. Metoda prima parametar – polje znakovnih nizova. U njemu su pobrojani argumenti zadani u naredbenom retku prilikom pokretanja programa.

Ispis 2.6: Izvršna klasa

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

```
    SimpleCounter cntr = new SimpleCounter();
    System.out.println( "Početno stanje brojača: " + cntr.getState() );
    cntr.increment();
    System.out.println( "Stanje nakon poziva metode increment: "
      + cntr.getState() );
  }
}
```

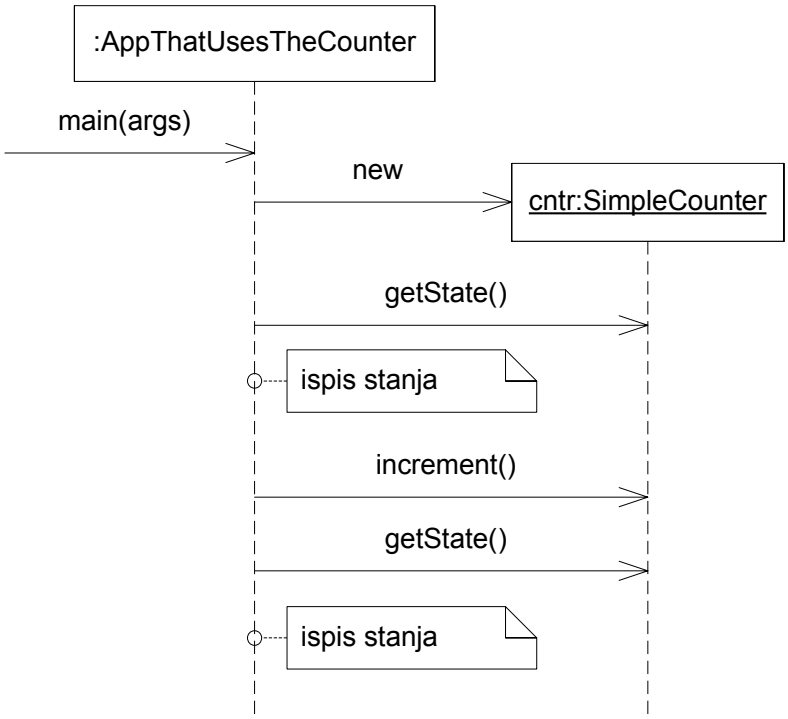
Prikazana metoda `main` stvara jedan objekt, koristi ga i o rezultatima izvješćuje korisnika ispisom teksta na zaslon. Izraz `new SimpleCounter()` kreira novi objekt klase `SimpleCounter`. *Referenca* na taj objekt zapisuje se u *lokalnu varijablu* `cntr` i nakon toga se preko te varijable mogu pozivati metode objekta. Ispis koji generira program bi trebao biti sljedeći:

```
Početno stanje brojača: 0
Stanje nakon poziva metode increment: 1
```

2.5.1 SlijedniDijagram

Slijedni dijagram opisuje dinamičku suradnju između objekta, a posebno je dobar za prikaz slijeda poruka. U njemu se može prikazati vrijeme života objekta (kreiranje i uništavanje). Slika 2.9 prikazuje slijedni dijagram prethodnog koda.

Pravokutnik predstavlja objekt. Osnovna sintaksa objekta je `naziv_objekta:naziv_klase`. Ako je ovakva oznaka podvučena onda predstavlja objekt, a ako nije onda klasu. Ime objekta se može izostaviti. Svaki objekt ima životnu liniju. To je linija koja je povezana sa simbolom objekta i okomito se spušta dolje. Ona zapravo predstavlja tijek vremena, dakle vrijeme teče od gornjeg dijela dijagrama prema dolje. Kako objekti komuniciraju porukama one se prikazuju strelicama između životnih linija. Iznad strelice se stavlja naziv metode i parametri s kojima se metoda poziva. Ako strelica završava na simbolu objekta onda to znači da se objekt kreira pomoću te poruke. Kada se neka poruka ponavlja u petlji onda ispred naziva poruke stavljamo znak * kao oznaku ponavljanja. Kada nešto ne možemo prikazati u dijagramu onda možemo staviti komentar. Komentar ima simbol pravokutnika kojem je gornji desni kut "presavinut" kao da smo kut papira presavinuli. Takav komentar se spaja s linijom poruke ili na životnu liniju, ovisno o tome što pojašnjava. UML verzije 2.0 ima i dodatne simbole koji se mogu koristiti u ovom dijagramu, ako nekoga zanima neka pogleda u specifikaciju [9].



Slika 2.9: Dijagram klasa – `AppThatUsesTheCounter`

2.6 Pojam klase

U ovom trenutku može se usvojiti i osnovna ideja *klase*. Možemo ju smatrati predloškom za objekte koji prvenstveno propisuje koje atribute će objekt imati za čuvanje stanja i sadrži definicije metoda koje će se nad njim moći pozivati. Objekt se još naziva i *instancom* svoje klase. Sve instance iste klase imaju isti popis atributa, ali naravno svaka ima svoje vlastite vrijednosti tih atributa. To se može ilustrirati ako izmijenimo tekst gornje metode `main` na sljedeći:

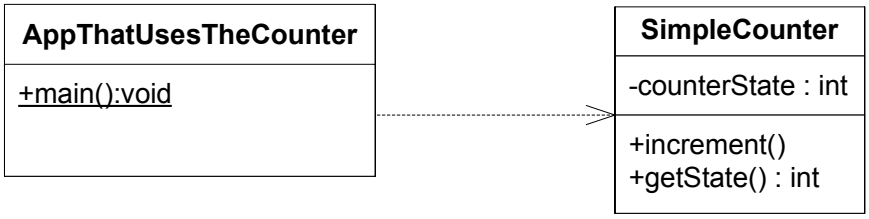
Ispis 2.7: Program koji koristi dvije instance iste klase

```
public static void main( String[] args )
{
  SimpleCounter cntr1 = new SimpleCounter();
  SimpleCounter cntr2 = new SimpleCounter();
  System.out.println( "Početna stanja brojača: " + cntr1.getState() +
      ", " + cntr2.getState() );
  cntr1.increment();
  System.out.println( "Stanja nakon poziva increment nad cntr1: "
      + cntr1.getState() + ", " + cntr2.getState() );
}
```

Očekivani ispis:

```
Početna stanja brojača: 0, 0
Stanja nakon poziva metode increment nad cntr1: 1, 0
```

Dijagram klasa ovog primjera prikazan je na slici 2.10.



Slika 2.10: Dijagram klasa – `AppThatUsesTheCounter`

Klasa `AppThatUsesTheCounter` s ispisa 2.6 je drugačija: ona ne opisuje nikakav smisleni objekt, već samo sadrži metodu `main` koja je poseban slučaj i nije normalna metoda objekta. O tome će se više raspravljati kasnije (u odjeljku 3.7).

2.7 Povezivanje objekata, izgradnja objektnog sustava

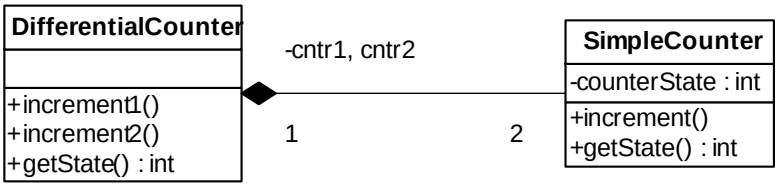
Dva objekta može se povezati tako da jedan objekt sadrži atribut koji pokazuje na drugi objekt, npr.

Ispis 2.8: Klasa `DifferentialCounter`

```
public class DifferentialCounter
{
  private SimpleCounter cntr1 = new SimpleCounter();
  private SimpleCounter cntr2 = new SimpleCounter();

  public void increment1()
  {
    this.cntr1.increment();
  }

  public void increment2()
  {
    this.cntr2.increment();
  }

  public int getDifference()
  {
    return this.cntr2.getState() - this.cntr1.getState();
```

```
  }

  public void reset()
  {
    this.cntr1 = new SimpleCounter();
    this.cntr2 = new SimpleCounter();
  }
}
```

UML-ov dijagram klasa prikazan je na slici 2.11. Ovdje se vidi da je relacija između dviju klasa kompozicija. Klasa `DifferentialCounter` sadrži klasu `SimpleCounter`. Kod kompozicije je karakteristično da na sadržanu klasu `SimpleCounter` nitko nema referencu. Iz koda se vidi da su atributi `cntr1` i `cntr2` privatni i da se objekti `SimpleCounter` stvaraju prilikom stvaranja objekta `DifferentialCounter`. Pošto nema metoda koje vraćaju reference na te sadržane objekte onda je to dokaz da je relacija između njih kompozicija.

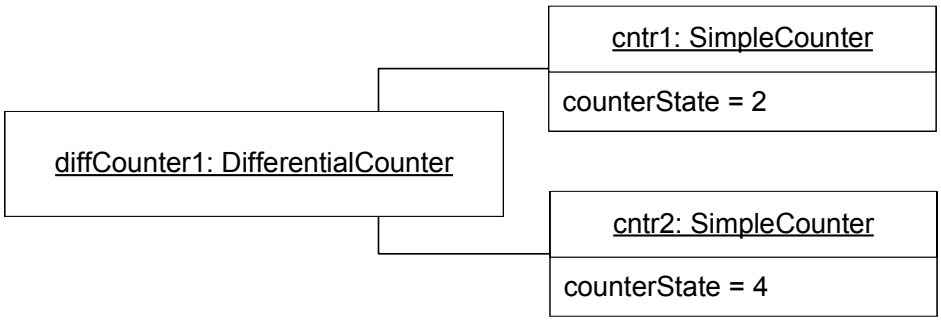


Slika 2.11: Klasni dijagram minimalnog objektnog sustava

2.7.1 Dijagram objekata

Ovaj dijagram opisuje sustav u jednom vremenskom trenutku tj. prokazuje objekte, asocijacije i vrijednosti atributa. Objekt se prikazuje istim simbolom kao i klasa u dijagramu klasa, ali bez dijela s metodama. Atributi se prikazuju sljedećom sintaksom: `nazivAtributa = vrijednost`. Primjer jednog mogućeg stanja objekta iz klase `DifferentialCounter` je prikazan na slici 2.12.

Objekt klase `DifferentialCounter` povezan je s dva objekta klase `SimpleCounter` i komunicira s njima. Na primjer, kad se pozove metoda `getDifference`, objekt će stupiti u kontakt sa svakim od brojača i zatražiti njihovo trenutno stanje. Time je izgrađen minimalan sustav objekata. Asocijativni odnos između `DifferentialCounter` i `SimpleCounter` naziva se *kompozicijom*. Ključna karakteristika ovog odnosa je da podređeni objekti postoje isključivo kao sastavni dio nadređenog i njihovo postojanje je uvjetovano tim



Slika 2.12: Objektni dijagram minimalnog objektnog sustava

objektom. U to se možemo uvjeriti ako uvidimo da je `DifferentialCounter` taj koji stvara podređene objekte i da nijedan vanjski objekt ne može doći do reference na njih (zapisane su u privatnim atributima, nijedna javna metoda ih ne vraća).

Prikazani slučaj u kojem se atributi `cntr1` i `cntr2` inicijaliziraju na mjestu gdje su i deklarirani samo je jedan od načina inicijalizacije atributa. Na primjer, metoda `reset` također ih (re)inicijalizira, stvarajući nove objekte `SimpleCounter` (time se stari objekti implicitno odbacuju i bit će automatski uklonjeni iz radne memorije postupkom zvanim *garbage collection*, "sakupljanje smeća"). Tipičan način inicijalizacije atributa je unutar *konstruktora*, o kojem će se pričati u sljedećem odjeljku.

2.8 Inicijalizacija objekta konstruktorom

Svaki put kad upotrijebimo ključnu riječ `new` kojom zatražimo kreiranje nove instance zadane klase, implicitno smo zatražili i pozivanje posebne inicijalizacijske procedure opisane *konstruktorom* objekta. Svaka klasa ima bar jedan konstruktor, a ako ih ima više, oni se razlikuju po broju i vrstama parametara. Pomoću parametara možemo prenijeti dodatnu informaciju inicijalizacijskom postupku, čime postižemo da se objekt inicijalizira u onakvo stanje kakvo je potrebno u danom trenutku. Primjerice, možemo proširiti našu klasu `SimpleCounter`:

Ispis 2.9: Proširena varijanta klase `SimpleCounter`

```
public class SimpleCounter
{
  private int counterState;
```

```
  public SimpleCounter( int initialState )
  {
    this.counterState = initialState;
  }

  public SimpleCounter()
  {
    //ne radi ništa dodatno; counterState se automatski
    //inicijalizira na nulu
  }

   ... deklaracije metoda increment i getState kao ranije ...
}
```

U kodu 2.9 definirana su dva konstruktora: prvi prima jedan cijeli broj – početnu vrijednost brojača. Drugi je takozvani *podrazumijevani* konstruktor – onaj koji ne prima nijedan parametar. U ranijem slučaju (ispis 2.4), iako nije bio deklariran nijedan, klasi je implicitno dodan upravo ovakav, prazan podrazumijevani konstruktor. Međutim, čim deklariramo bilo koji konstruktor eksplicitno, ništa se ne dodaje implicitno. Dakle, klasa *ne mora* imati podrazumijevani konstruktor, iako bi se to moglo zaključiti sudeći po njegovom imenu.

Primjer poziva konstruktora:

Ispis 2.10: Korištenje konstruktora

```
  public static void main( String[] args )
  {
    SimpleCounter cntr1 = new SimpleCounter();
    SimpleCounter cntr2 = new SimpleCounter( 3 );
    System.out.println( "Početna stanja brojača: " + cntr1.getState() +
        ", " + cntr2.getState() );
    cntr1.increment();
    System.out.println( "Stanja nakon poziva metode increment nad cntr1: "
        + cntr1.getState() + ", " + cntr2.getState() );
  }
```

U ovom primjeru `cntr1` se inicijalizira podrazumijevanim konstruktorom, a `cntr2` konstruktorom koji putem parametra zaprima željeno početno stanje. Očekivani ispis:

```
Početno stanje brojača: 0, 3
Stanje nakon poziva metode increment nad cntr1: 1, 3
```

2.9 Povezivanje objekata agregacijom

Uzmimo da imamo objekt koji služi za jednostavnu analizu tekstualne datoteke – ustanovljuje koliko redaka teksta sadrži. Datoteku predstavlja objekt vrste `TextFile`. Nad njim možemo pozvati metodu `readLine` koja čita jedan redak datoteke i metodu `endReached` koja nas izvještava ima li još redaka za pročitati. Za čuvanje rezultata brojanja koristi se objekt vrste `SimpleCounter`. Objekt-analizator teksta je instanca klase `TextAnalyzer`:

Ispis 2.11: Analizator teksta

```
public class TextAnalyzer
{
  private SimpleCounter cntr;
  private TextFile file;
  private boolean countDone;

  public TextAnalyzer( SimpleCounter cntr, TextFile file )
  {
    this.cntr = cntr;
    this.file = file;
  }

  public int countLines()
  {
    if ( !this.countDone )
    {
      while ( !file.endReached() )
      {
        file.readLine();
        this.cntr.increment();
      }
      this.countDone = true;
    }
    return cntr.getState();
  }
}
```

Najvažnije je uočiti da konstruktor klase prima kao parametre instance klasa `SimpleCounter` i `TextFile`. Možemo u zapisu konstruktora vidjeti da se reference na te objekte izravno prepisuju u atribute `cntr` odnosno `file`. Klasa `TextAnalyzer` stoga ima asocijativnu vezu prema `SimpleCounter`-u i `TextFile`-u, ali ovdje se ne radi o kompoziciji, koju smo sreli ranije. To se vidi po tome što `TextAnalyzer` prima izvana stvorene objekte, tako da oni nisu njegovi privatni. Ovdje to može biti sasvim korisno: recimo da imamo jedan veći dokument razbijen u više datoteka, npr. jedna datoteka

za svako poglavlje. Ono što nas zanima je ukupan broj redaka u čitavom dokumentu. To možemo postići korištenjem našeg analizatora teksta ovako (primjer pretpostavlja da imamo dokument razbijen u dvije datoteke):

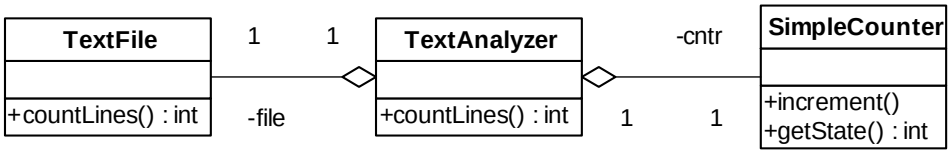
Ispis 2.12: Analiza višedijelnog dokumenta

```
public static void main( String[] args )
{
  TextFile file1 = new TextFile( 15 ); // datoteka od 15 redaka
  TextFile file2 = new TextFile( 25 ); // datoteka od 25 redaka
  SimpleCounter cntr = new SimpleCounter();

  TextAnalyzer ta1 = new TextAnalyzer( cntr, file1 );
  System.out.println(
    "Broj redaka u prvoj datoteci: " + ta1.countLines() );

  // Instanciramo analizator druge datoteke, ali s istim brojačem:
  TextAnalyzer ta2 = new TextAnalyzer( cntr, file2 );
  System.out.println( "Ukupan broj redaka: " + ta2.countLines() );
}
```

Očekivani ispis:

```
Broj redaka u prvoj datoteci: 15
Ukupan broj redaka: 40
```

Iz primjera je jasno da je životni vijek objekta brojača neovisan o objektu analizatora. Objekt analizatora, nakon što je obavio svoj posao, može se ukloniti iz memorije a da to ne utječe na objekt brojača. Za takav slučaj koristi se izraz *agregacija* – to je općenito odnos između cjeline i njenih sastavnih dijelova, bez obzira na to pripadaju li dijelovi isključivo toj cjelini ili ne. Uočimo da kompozicija također zadovoljava ovu definiciju, ali je njena definicija stroža. Dakle, svaka kompozicija je ujedno i agregacija, ali obrat ne vrijedi. UML-om se agregacija prikazuje kao na slici 2.13.



Slika 2.13: UML-ov dijagram klasa u odnosu agregacije

U kodu klase `TextAnalyzer` koristi se objekt vrste `TextFile`. Umjesto punog objekta koji radi s pravim datotekama radi isprobavanja koristimo ovaj jednostavan nadomjestak:

Ispis 2.13: Nadomjesna implementacija klase `TextFile`

```
public class TextFile
{
  private int linesTotal;
  private int linesRead;

  public TextFile( int lineCount )
  {
    this.linesTotal = lineCount;
  }

  public void readLine()
  {
    if ( this.linesRead < this.linesTotal )
      this.linesRead++;
  }

  public boolean endReached()
  {
    return this.linesRead == this.linesTotal;
  }
}
```

2.10 Polimorfizam i Javino sučelje

U Javi možemo razdvojeno opisati *sučelje* objekta prema okolini (dakle popis metoda s kojima raspolaže) i *implementaciju* tog sučelja (sam kod metoda, atributi potrebni za čuvanje stanja). Posebno je bitno da za jednu definiciju sučelja možemo imati mnogo različitih implementacija. Ovo je korisno iz mnogo razloga, a ovdje ćemo opisati jedan od njih.

Recimo da imamo gornji slučaj analizatora teksta. Želimo da, ovisno o konkretnoj primjeni, možemo dobiti ili ukupan broj redaka ili samo njegovu posljednju znamenku (brojanje po modulu 10, kao u ispisu 2.5). Treba uvesti samo sitnu izmjenu u kod s ispisa 2.11:

Ispis 2.14: Polimorfni analizator teksta

```
public class TextAnalyzer
{
  private Counter cntr;
  private TextFile file;
  private boolean countDone;

  public TextAnalyzer( Counter cntr, TextFile file )
  {
    this.cntr = cntr;
```

```
    this.file = file;
  }

  public int countLines()
  {
    if ( !this.countDone )
    {
      while ( !file.endReached() )
      {
        file.readLine();
        this.cntr.increment();
      }
      this.countDone = true;
    }
    return cntr.getState();
  }
}
```

Ovaj kod je identičan kao ranije, osim što se umjesto `SimpleCounter` koristi vrsta `Counter`, čiju definiciju još nismo vidjeli. Radi se o Javinom *sučelju*:

Ispis 2.15: Sučelje `Counter`

```
public interface Counter
{
  void increment();
  int getState();
}
```

U sučelju, kao što smo najavili, nema *definicije* metoda, već samo popis metoda koje objekt s tim sučeljem posjeduje, tj. *deklaracije* metoda. Konstruktor klase `TextAnalyzer` prihvatit će bilo koji objekt koji *implementira* to sučelje, a metodi `countLines` bit će svejedno o kakvoj se točno implementaciji radi. Međutim, različite implementacije rezultirat će različitim ponašanjem metode. Ova karakteristika objektno-orijentiranih jezika zove se *polimorfizam* ili *višeobličnost*. Metoda "misli" da uvijek radi s objektom iste vrste (`Counter`), koji se iz te perspektive doima kao da se mijenja od slučaja do slučaja – jednom se može ponašati kao `SimpleCounter`, a drugi put kao `ModuloCounter`. Uočimo i još jednu sitnicu: metode u sučelju nisu označene kao `public` jer se to podrazumijeva – nemoguće je u sučelju imati metodu koja ne bi bila javna. Klasu `TextAnalyzer` možemo koristiti npr. ovako:

Ispis 2.16: Polimorfizam na djelu

```
  public static void main( String[] args )
  {
    TextAnalyzer ta1 =
      new TextAnalyzer( new SimpleCounter(), new TextFile( 37 ) );
    TextAnalyzer ta2 =
      new TextAnalyzer( new ModuloCounter(), new TextFile( 37 ) );

    int count1 = ta1.countLines();
    int count2 = ta2.countLines();

    System.out.println( "SimpleCounter je izbrojao " + count1 );
    System.out.println( "ModuloCounter je izbrojao " + count2 );
  }
```

Očekivani ispis:

```
SimpleCounter je izbrojao 37
ModuloCounter je izbrojao 7
```

Prisutnost polimorfizma možemo uočiti već u prvom retku koda. Konstruktoru klase `TextAnalyzer`, koji je definiran da prima argumente vrste `Counter`, prosljeđuje se referenca na instancu klase `SimpleCounter`. Jezična pravila to dopuštaju jer objekt `SimpleCounter` implementira sučelje `Counter` (uz sitan dodatak, prikazan u ispisu 2.18) i nad njim je moguće pozvati sve metode deklarirane u sučelju. U drugom retku istom konstruktoru prosljeđuje se referenca na instancu druge klase – ovaj put `ModuloCounter`. U trećem i četvrtom retku najjasnije se vidi djelovanje polimorfizma. Nad `ta1` i `ta2` poziva se ista metoda s identičnim argumentom. Time se pokreće identičan kod metode `countLines` (ispis 2.14). Međutim, ta dva poziva rezultiraju različitim povratnim vrijednostima, što se manifestira u ispisu u posljednja dva retka. Razlika nastupa u trenutku izvršavanja retka `this.cntr.increment()`, gdje se donosi odluka koju točno metodu `increment()` treba pozvati. U prvom slučaju to će biti metoda definirana u klasi `SimpleCounter`, a u drugom slučaju u `ModuloCounter`. Dakle, taj redak propisuje samo *ime* metode koju će se pozvati (i parametre koje ona mora imati), ali ne veže se ni uz jednu konkretnu metodu. Ovaj postupak zove se *dinamičko povezivanje* – za razliku od statičkog, gdje se u trenutku prevođenja ("kompajliranja") donosi konačna odluka o tome koja metoda će biti pozvana.

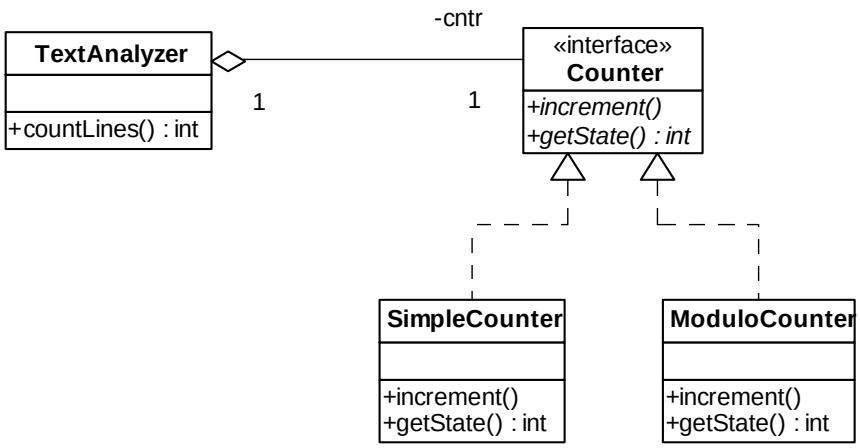
Novu situaciju sa sučeljem možemo vizualizirati UML-ovim dijagramom sa slike 2.14. Ovakav stil dizajna, u usporedbi s dizajnom sa slike 2.13, od presudne je važnosti u objektno-orijentiranim jezicima. Gotovo bi se moglo reći da je činjenica da omogućuju takav dizajn smisao njihovog postojanja.

Uz malu izmjenu na ispisu 2.16 možemo pokazati još jedan tipičan način korištenja polimorfizma:

Ispis 2.17: Još jedan primjer korištenja polimorfizma

```
public static void main( String[] args )
{
  Counter cntr1 = new SimpleCounter();
  Counter cntr2 = new ModuloCounter();

  TextAnalyzer ta1 = new TextAnalyzer( cntr1 );
  TextAnalyzer ta2 = new TextAnalyzer( cntr2 );

  ... ostatak identičan kao gore ...
}
```

Ovdje se u prva dva retka varijablama vrste `Counter` pridjeljuju reference na instance klasa `SimpleCounter` odnosno `ModuloCounter`. Zatim se s tim varijablama poziva konstruktor klase `TextAnalyzer`. Možemo uočiti da se prednosti polimorfizma ne koriste samo u asocijativnim vezama (korištenjem sučelja za vrstu atributa), već u pravilu gdje god je to moguće. Ako objekte `cntr1` i `cntr2` u kodu koristimo kao općenite brojače, ne oslanjajući se na nikakve detalje specifične za konkretne klase, varijable trebaju biti općenite vrste `Counter`. Sučelje se redovito koristi i, kao što smo vidjeli u primjeru, za vrste parametara – a također i za vrste povratnih vrijednosti metoda. Načelno, uvijek treba koristiti najopćenitiju vrstu koja dolazi u obzir , a to je u našim primjerima sučelje `Counter`.



Slika 2.14: UML-ov dijagram klasa za polimorfni analizator teksta (klasa `TextFile` je ovdje izostavljena)

U Javi svaka klasa mora se unaprijed obvezati koja sve sučelja (može ih biti više) će implementirati. U našem slučaju bit će potrebno dodati redak "`implements Counter`" u deklaracije obiju klasa. Puni kod tako izmijenjenih klasa prikazan je na ispisu 2.18. Primijetimo i mali dodatak u konstruktoru klase `ModuloCounter` kojim se osigurava od slučaja zadavanja prevelikog početnog broja uzimanjem ostatka dijeljenja s 10.

Ispis 2.18: Obvezivanje na implementaciju sučelja

```
public class SimpleCounter
implements Counter
{
  private int counterState;

  public SimpleCounter( int initialState )
  {
    this.counterState = initialState;
  }

  public SimpleCounter()
  {
  }

  public void increment()
  {
    this.counterState++;
  }

  public int getState()
  {
    return this.counterState;
  }
}


public class ModuloCounter
implements Counter
{
  private int counterState;

  public ModuloCounter( int initialState )
  {
    this.counterState = initialState % 10;
  }

  public ModuloCounter()
  {
  }

  public void increment()
  {
```

```
    if ( this.counterState == 9 )
      this.counterState = 0;
    else
      this.counterState++;
  }

  public int getState()
  {
    return this.counterState;
  }
}
```

2.11 Izbjegavanje ponavljanja koda nasljeđivanjem

Ono što nas u ispisu 2.18 može zasmetati, a u kompleksnijem kodu može prerasti i u puno veći problem, je ponavljanje koda u dvije očito srodne klase. Obje klase koriste isti atribut, a i metoda `getState` im je identična. Java omogućuje tehniku kojom se takvo ponavljanje može izbjeći – *nasljeđivanje* (eng. *inheritance*). U našem slučaju mogli bismo postupiti tako da propišemo da klasa `ModuloCounter` ne nastaje "od nule", već da je *izvedena* (eng. *derived*) iz klase `SimpleCounter`. U tom slučaju ona će *naslijediti* sve njene metode i atribute. Nakon toga možemo predefinirati (*nadjačati*, eng. *override*) one metode čija implementacija više ne odgovara (metodu `increment`). Izvodom se između `ModuloCounter` i `SimpleCounter` uspostavlja odnos *podklasa-nadklasa*.

Konstruktori se ne nasljeđuju i moramo ih uvijek pisati iznova. Međutim, u prvom retku konstruktora možemo pozvati odgovarajući konstruktor iz nadklase i time prepustiti njemu odrađivanje zajedničkog dijela posla.

Ispis 2.19: Nasljeđivanje – osnovna klasa

```
public class SimpleCounter
implements Counter
{
  int counterState;

  public SimpleCounter( int initialState )
  {
    this.counterState = initialState;
  }

  public SimpleCounter()
  {
```

```
  }

  public void increment()
  {
    this.counterState++;
  }

  public int getState()
  {
    return this.counterState;
  }
}
```

Ispis 2.20: Nasljeđivanje – izvedena klasa

```
public class ModuloCounter
extends SimpleCounter
{
  public ModuloCounter( int initialState )
  {
    super( initialState % 10 );
  }

  public ModuloCounter()
  {
  }

  @Override
  public void increment()
  {
    super.increment();
    if ( this.counterState == 10 )
      this.counterState = 0;
  }
}
```

U ovom slučaju bili smo prisiljeni proširiti dostupnost atributu `counterState` uklanjanjem modifikatora `private`. Time je njegova dostupnost proširena na *paketno-privatnu* (eng. *package-private*) razinu: dostupan je iz svih klasa unutar istog *paketa* (o tome kasnije, u odjeljku 2.13), pa tako i iz podklase `ModuloCounter`. Proučimo novu implementaciju metode `increment` u podklasi. Implementacija je izvedena tako da se prvo pozove izvorna implementacija iz nadklase (`super.increment`), a zatim se korigira slučaj kad brojač dođe do 10 i treba ga poništiti. Ova tehnika je vrlo prikladna u učestaloj situaciji iz prakse gdje je posao koji obavlja izvorna metoda većinom ispravan, ali ga treba još malo korigirati ili proširiti. Općenito, nadjačavajuća metoda nema obvezu pozivati metodu iz nadklase, a ako ju poziva,

nije nužno da to čini u prvom retku koda. Poziv pomoću `super` koristi se i u zapisu konstruktora – više o ovome u odjeljku 3.12.3.

Iznad definicije metode `increment` u `ModuloCounter`-u pojavljuje se i `@Override`. To je tzv. Javin *annotation*, "primjedba" uz metodu. Postoje razne vrste *annotation*-a, a ovdje korišten *Override* naznačuje da ta metoda nadjačava metodu iz nadklase. Njegovo korištenje je opcionalno, ali korisno. Time prevoditelju dajemo do znanja da predviđamo da ta metoda nadjačava metodu iz nadklase pa će nas on upozoriti ako pogriješimo i npr. navedemo krivo ime metode, recimo `inkrement`. Takve pogreške ponekad je teško uočiti jer ćemo time umjesto nadjačavanja definirati novu metodu a da toga nismo svjesni.

Uočimo još i sljedeće: klasa `ModuloCounter` ne ponavlja izjavu `implements Counter` jer se to podrazumijeva – podklasa automatski, samom prirodom nasljeđivanja, implementira sva sučelja koja implementira njena nadklasa.

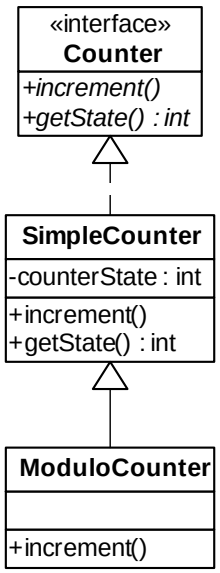
2.12 Klasna hijerarhija, apstraktna klasa

UML-ov dijagram slučaja iz ispisa 2.19 prikazan je na slici 2.15. Ova organizacija je riješila problem ponavljanja koda. Međutim, sad se pojavio novi problem. Dosad je bilo naglašeno da polimorfizam omogućuje instancama različitih klasa da se ponašaju kao da su instance sučelja koje implementiraju. Isto se odnosi i na slučaj instance izvedene klase – ona se može koristiti i kao instanca svoje nadklase. U ovom slučaju to znači da sve instance klase `ModuloCounter` možemo tretirati kao da je njihova vrsta `SimpleCounter`. Ako npr. negdje imamo deklariranu metodu

```
void setSimpleCounter( SimpleCounter cntr )
```

ona će uredno prihvatiti i argument vrste `ModuloCounter`. Međutim, najvjerojatnije je da metoda očekuje objekt s ponašanjem definiranim u `SimpleCounter` i neće ispravno raditi s drugačijim objektima. U protivnom je mogla npr. zahtijevati argument općenite vrste `Counter`. Ovu situaciju možemo izbjeći uz uporabu više koda, ali još uvijek zadržavajući tu prednost da se ne ponavlja deklaracija atributa ni implementacija metode `getState`. Možemo u hijerarhiju ubaciti *apstraktnu* klasu:

Ispis 2.21: Apstraktna klasa



Slika 2.15: UML-ov dijagram klasa iz ispisa 2.19

```
abstract class AbstractCounter
implements Counter
{
  int counterState;

  public AbstractCounter( int initialState )
  {
    this.counterState = initialState;
  }

  public AbstractCounter()
  {
  }

  public int getState()
  {
    return this.counterState;
  }

  public abstract void increment();
}
```

Glavna novost ovdje je metoda `increment`, koja je sad postala *apstraktna*. Najavljeno je njeno postojanje, ali nema implementacije. Logično slijedi da se ovakvu klasu ne smije moći instancirati jer je opis njenog ponašanja nepotpun – ne zna se što bi se trebalo dogoditi prilikom poziva metode `increment`. Upravo tu zabranu uvodi modifikator `abstract` nad klasom. Općenito, svaku klasu smije se proglasiti apstraktnom i time zabraniti njeno

instanciranje – bez obzira ima li ona apstraktnih metoda ili ne. S druge strane, postojanje ijedne apstraktne metode prisiljava nas da takvu klasu proglasimo apstraktnom.

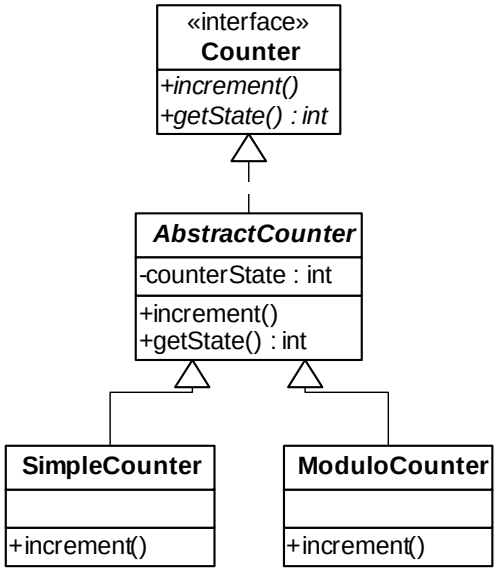
Koji je smisao ovog zahvata? Pogledajmo preuređeni kod dviju otprije poznatih klasa:

Ispis 2.22: Konkretne klase izvedene iz apstraktne

```
public class SimpleCounter
extends AbstractCounter
{
  public SimpleCounter( int initialState )
  {
    super( initialState );
  }

  public SimpleCounter()
  {
  }

  @Override
  public void increment()
  {
    this.counterState++;
  }
}

public class ModuloCounter
extends AbstractCounter
{
  public ModuloCounter( int initialState )
  {
    super( initialState % 10 );
  }

  public ModuloCounter()
  {
  }

  @Override
  public void increment()
  {
    if ( this.counterState == 9 )
      this.counterState = 0;
    else
      this.counterState++;
  }
}
```

Ovu situaciju prikazuje UML-ov dijagram na slici 2.16. Sad su obje konkretne klase izvedene iz `AbstractCounter`. Sav zajednički kod nalazi se u

apstraktnoj klasi, a metoda `increment` implementirana je u svakoj konkretnoj klasi posebno. Takva situacija bolje odgovara ulozi jedne i druge klase, koje su po svojoj prirodi dvije paralelne implementacije sučelja `Counter`. Uočimo još na ispisu 2.21 da je za klasu `AbstractCounter` izostavljen modifikator `public`. Ona ima pomoćni karakter i ne bi trebala služiti da se deklariraju varijable ili parametri čija vrsta je `AbstractCounter`. Umjesto toga treba uvijek koristiti sučelje `Counter`. Stoga je njena vidljivost smanjena na samo unutar svog paketa.



Slika 2.16: UML-ov dijagram klasa iz ispisa 2.21 i 2.22

2.13 Paketi, imenovanje klasa i sučelja

2.13.1 Koncepcija paketa

Dosad se već više puta spominjala koncepcija paketa. Vrijeme je da se razjasni taj pojam, a i da ga se stavi u potpunu uporabu u kodu. Jezik Java omogućava grupiranje više klasa u jednu cjelinu koja se naziva *paketom*. Pripadnost klase paketu objavljuje se u prvom retku datoteke u kojoj je definirana, npr. ovako:

```
package hr.fer.tel.ilj;

public interface Counter
{
```

```
  void increment();
  int getState();
}
```

Time se sučelje `Counter` našlo u paketu `hr.fer.tel.ilj` i sada je njegovo puno ime `hr.fer.tel.ilj.Counter`. U kodu svih klasa unutar tog istog paketa ne mora se navoditi puno ime, ali u bilo kojem drugom paketu to postaje potrebno. Da kod ipak ne bi postao previše "zagušen" ovako dugačkim imenima, Java podržava i *uvoz* pojedinih imena klasa/sučelja tako da se na njih može pozivati samo kratkim imenom. Za to služi ključna riječ `import` koja se pojavljuje odmah ispod deklaracije paketa:

```
package hr.fer.tel.ilj;

import java.util.List;
import java.util.ArrayList;
import javax.swing.*;

public class Xyz
{
...
}
```

Korišenjem zvjezdice, kao u `javax.swing.*`, uvozi se kompletan paket. Postoji i poseban paket `java.lang` koji je automatski uvezen, kao da je uvijek navedeno `import java.lang.*`. U njemu su najvažnije "ugrađene" klase kao što su `String` i `Object`.

2.13.2 Pravila imenovanja paketa

U praksi se paketi redovito koriste, tako da sav naš dosadašnji kod u tom smislu odudara od prakse i svaku datoteku treba dopuniti tom deklaracijom.

Već je vjerojatno očito da su na snazi i posebna pravila imenovanja paketa: on se imenuje prema imenu internetske domene organizacije pod čijom "kapom" se izrađuje kod, i to tako da se kreće od segmenta najviše hijerarhijske razine. Naš Zavod ima domenu `tel.fer.hr` i stoga sva imena paketa izrađenih na Zavodu trebaju počinjati na `hr.fer.tel`. Za sasvim male projekte bit će dovoljan jedan paket, ali već za samo malo veće trebat će ih obično više. U tom slučaju opet ćemo imati glavno ime, kao u našem primjeru `hr.fer.tel.ilj`, na koje će se dodavati još segmenata po potrebi.

Paketi se primjenjuju prije svega zato da ne dođe do "sudara" između imena klasa koje kreiraju različiti autori i imaju različite namjene. Upravo zato su definirana gornja pravila imenovanja paketa koja osiguravaju jedinstvenost njegovog imena. Iz istog razloga treba dobro paziti koja imena se uvoze da se ne bi uvezlo pogrešno. Na primjer, u standardnu biblioteku klasa koja dolazi s Javom ugrađeni su i `java.awt.List` i `java.util.List`, dvije klase vrlo različitih namjena. Uvozom pogrešnog imena doći će do vrlo čudnih pogrešaka gdje se prevoditelj ("kompajler") buni za svaku metodu koju pozivamo, iako sve izgleda ispravno.

2.13.3 Pravila za raspored datoteka po kazalima

Ime paketa u kojem je klasa povlači za sobom i propis u kojem kazalu ona mora biti smještena u sustavu datoteka. Počevši od nekog proizvoljnog kazala koje se proglašava korijenskim potrebno je izgraditi hijerarhiju kazala identičnu hijerarhiji segmenata u imenu paketa. Konkretno, datoteka `Counter.java` mora se naći u kazalu `<korijensko>/hr/fer/tel/ilj`. Ime same datoteke također je propisano i mora biti jednako imenu klase deklarirane u njoj. Bez obzira na ovakav hijerarhijski način organiziranja paketa po kazalima, formalno između dva paketa različitog imena ne postoji nikakva povezanost. Dakle, paket `hr.fer` nije u ništa bližem odnosu s paketom `hr.fer.tel` nego s npr. paketom `com.sun`.

Svaki razvojni alat automatski se brine za poštivanje svih ovih pravila i stoga je bitno koristiti tu njegovu podršku. Na primjer, u ovom trenutku trebalo bi sve datoteke, koje su dosad bile u neimenovanom, tzv. *podrazumijevanom* ili *korijenskom* paketu, preseliti u paket `hr.fer.tel.ilj`. Jedan zgodan način u okružju *Eclipse* je da se u Package Explorer-u označi sve klase i pritisne kombinacija tipaka Alt-Shift-V, čime se zahtijeva preseljenje klasa u drugi paket. Dotični još ne postoji pa ga se u prozoru koji se pojavio stvori pomoću gumba New.

2.13.4 Uloga paketno-privatne razine dostupnosti

Sad se može bolje razumjeti ono što se ranije govorilo o razini dostupnosti atributa, metoda i klasa. Svaki razvojni projekt u Javi oslanja se na velik broj klasa i sučelja koja već postoje – mnogo njih dolazi već kao dio samog jezika, odnosno dostupan je odmah nakon instaliranja Jave na računalo.

Većina samih softverskih proizvoda također su takve prirode da trebaju služiti drugim razvijateljima kao gradivni elementi. Korisniku svih tih paketa neće biti dostupni oni dijelovi koji nisu označeni kao javni (`public`). Time razvijatelj paketa može imati bolju kontrolu nad time kako će se njegov kod koristiti: olakšava korisniku ispravno korištenje, a i zadržava slobodu kasnijih izmjena nad svim dijelovima koji nisu javni. Na primjer, uklanjanje oznake `public` s klase `AbstractCounter` spriječit će korisnika našeg paketa da radi izravno s tom klasom. Kasnije ju možemo npr. preimenovati ili možda potpuno ukinuti bez utjecaja na korisnika našeg paketa.

2.14 Refaktorizacija

Kako smo napredovali u izgradnji koda našeg primjera, u više navrata radili smo prepravke u kodu s ciljem njegove bolje organizacije (npr. izbjegavanje ponavljanja), a bez utjecaja na ponašanje objekata. Takve izmjene učestale su u razvoju softvera i nazivaju se *refaktorizacijom*. Gledajući trenutno stanje našeg koda (ispisi 2.21 i 2.22), opet možemo uočiti jednu sitnicu koja bi se mogla popraviti. Klasa `ModuloCounter` na dva neovisna mjesta nameće ciklus brojanja od nula do devet: posebno u konstruktoru i u metodi `increment`. Takva situacija otvara mogućnost za nekonzistentnost – može se greškom dogoditi da se na jednom mjestu nameće npr. ciklus od nula do 7, a na drugom do 9. Poželjno je provesti još jednu refaktorizaciju koja će ovo otkloniti. Više je načina za to, ali ovdje ćemo pokazati jedan koji sadrži i neke novosti u dizajnu objekata.

Dosad smo se susreli s enkapsulacijom atributa kao jednom od najosnovnijih tehnika dizajna. Međutim, često je korisno enkapsulirati i metode. To ćemo ovdje iskoristiti: uvest ćemo novu, paketno-privatnu metodu `setState` koja će se pozivati prilikom svake promjene stanja i u kojoj će se provoditi kontrola nailaska na gornju granicu ciklusa. Dakle, svaka od dviju naših konkretnih klasa imat će svoju implementaciju `setState`, a time će nestati potreba da metoda `increment` ima različite implementacije – ona će se oslanjati na promjenjivo ponašanje metode `setState`. Za takvu metodu kaže se da je *polimorfna*. Konstruktor će također pozivati ovu metodu. Metoda `setState` nije dio javnog sučelja objekta i stoga se definicija `Counter`-a ne mijenja. Iz perspektive koda koji barata objektima vrste `Counter` nije se dogodila nikakva promjena.

2.14.1 Modifikator `final` – kontrola polimorfizma

U novi kod apstraktne klase dodan je još jedan detalj: modifikatorom `final` označene su metode koje je *zabranjeno nadjačavati*. Uvijek je dobro svaku metodu za koju je predviđeno da bude ista za sve podklase označiti kao `final`. Polimorfizam, osim što daje fleksibilnost, može i otežati čitanje koda upravo zbog toga što pravila odabira metode koja će biti pozvana postaju znatno složenija. Oznaka `final` na metodi garantira da nije potrebno pregledavati sve podklase da se ustanovi nije li neka od njih nadjačala tu metodu. To je praktičnije i za programera koji dodaje novu podklasu jer taj modifikator ističe ulogu metode u čitavoj organizaciji klasa i ukazuje na to kako ispravno uvesti novu klasu.

Ispis 2.23: Nova verzija apstraktne klase

```
package hr.fer.tel.ilj;

abstract class AbstractCounter
implements Counter
{
  int counterState;

  public AbstractCounter( int initialState )
  {
    setState( initialState );
  }

  public AbstractCounter()
  {
  }

  public final int getState()
  {
    return this.counterState;
  }

  public final void increment()
  {
    setState( getState() + 1 );
  }

  abstract void setState( int newState );
}
```

U ispisima koji slijede modifikatorom `final` označene su klase. Kad djeluje na klasu, ovaj modifikator zabranjuje deklaraciju klase izvedene iz nje. Time se ponovo osigurava od polimorfizma tamo gdje je on nepoželjan –

ako npr. parametar neke metode ima vrstu `SimpleCounter`, garantirano je da će proslijeđeni objekt biti instanca upravo te klase, a ne neke podklase s izmijenjenim ponašanjem. Takav nepoželjan slučaj upravo smo imali ranije, prije uvođenja apstraktne klase.

Ispis 2.24: Nove konkretne klase

```
package hr.fer.tel.ilj;

public final class SimpleCounter
extends AbstractCounter
{
  public SimpleCounter( int initialState )
  {
    super( initialState );
  }

  public SimpleCounter()
  {
  }

  @Override
  void setState( int newState )
  {
    this.counterState = newState;
  }
}


package hr.fer.tel.ilj;

public final class ModuloCounter
extends AbstractCounter
{
  public ModuloCounter( int initialState )
  {
    super( initialState );
  }

  public ModuloCounter()
  {
  }

  @Override
  void setState( int newState )
  {
    this.counterState = newState % 10;
  }
}
```

2.15 Uvođenje više varijanti iste metode – preopterećenje

Ovisno o tome za što nam sve treba brojač, mogao bi se pojaviti npr. zahtjev da treba omogućiti i veće korake brojanja umjesto samo po jedan. U tom slučaju mogli bismo pojačati naše objekte novom varijantom metode `increment` koja prima i jedan brojevni parametar. Tada ćemo imati dvije metode istog imena, ali različitih parametara – to se zove *preopterećenjem metode* (eng. *overloading*). Što se tiče jezika Java, te dvije metode jednako su različite kao da imaju i različita imena jer se potpuni identitet metode sastoji od njenog imena i popisa vrsta parametara koje prima. Ta cjelina naziva se *potpisom metode* (eng. *signature*). Konkretno, ovdje ćemo imati metode `increment()` i `increment( int )`. Uočimo da s druge strane povratna vrijednost *ne ulazi* u potpis metode i ne smije se definirati dvije metode koje su iste po svemu osim po povratnoj vrijednosti.

Ispis 2.25: Prošireno sučelje `Counter`

```
public interface Counter
{
  void increment();
  void increment( int step );
  int getState();
}
```

Ispis 2.26: Proširenje apstraktne klase

```
package hr.fer.tel.ilj;

abstract class AbstractCounter
implements Counter
{
  .. početak koda identičan kao ranije ...

  public final void increment()
  {
    increment( 1 );
  }

  public final void increment( int step )
  {
    setState( getState() + step );
  }

  abstract void setState( int newState );
}
```

Možemo uočiti tipičan pristup koji se koristi u izgradnji sustava preopterećenih metoda: metoda koja prima cijeli broj je općenitija, a metoda bez parametara je u biti njen poseban slučaj u kojem se podrazumijeva jedinični korak. Stoga se poziv metode bez parametara *delegira* na općenitiju metodu, koju se poziva s podrazumijevanom vrijednosti.

Primjer korištenja preopterećenih metoda:

```
public static void main( String[] args )
{
  Counter cntr = new SimpleCounter();
  cntr.increment();
  System.out.println( "Stanje nakon poziva increment(): "
    + cntr.getState() );
  cntr.increment( 2 );
  System.out.println( "Stanje nakon poziva metode increment( 2 ): "
    + cntr.getState() );
}
```

VAŽNO! Prilikom korištenja sustava preopterećenih metoda može na prilično suptilan, čak i neintuitivan način doći do izražaja razlika između statičkog i dinamičkog povezivanja. Obvezno vidjeti raspravu u odjeljku 4.2.1.

2.16 Prijelaz s agregacije na kompoziciju – kopiranje objekata

U ispisu 2.14 definirana je klasa koja primjenjuje agregaciju, što je ilustrirano slikom 2.13. Konstruktor klase `TextAnalyzer` zaprima gotovu instancu brojača i ugrađuje ju u svoju instancu. U većini situacija iz prakse ovdje će ustvari isto biti potrebna prava kompozicija da bi se analizator teksta zaštitio od neželjenih intervencija izvana. Ako želimo imati oba svojstva – i da se objekt dobiva izvana, i da se primjenjuje čista kompozicija – bit će potrebno omogućiti *kopiranje* objekata. Analizator teksta zaprimit će objekt izvana i od njega napraviti *obrambenu kopiju* (eng. *defensive copy*) – identičan objekt, ali privatan. Daljnje promjene nad objektom dobivenim izvana neće utjecati na njegov rad. U Javi postoji poseban mehanizam za ovo – metoda `clone` – međutim, on se zbog određenih tehničkih problema počeo izbjegavati (vidjeti odjeljak 4.2.4). Ovdje ćemo stoga prikazati preporučenu alternativu – *kopirajući konstruktor* (eng. *copy constructor*) u

kombinaciji s posebnom metodom `getCopy`. Sučelje brojača moramo proširiti ovom metodom:

Ispis 2.27: Dodatak metode za kopiranje u sučelje `Counter`

```
public interface Counter
{
  void increment();
  void increment( int step );
  int getState();
  Counter getCopy();
}
```

Implementacije ove metode koristit će kopirajuće konstruktore. Ovdje se donosi primjer za `SimpleCounter`; za `ModuloCounter` postupa se analogno.

Ispis 2.28: Klasa `SimpleCounter` s dodanom podrškom za kopiranje

```
package hr.fer.tel.ilj;

public final class SimpleCounter
extends AbstractCounter
{
  public SimpleCounter( int initialState )
  {
    super( initialState );
  }

  public SimpleCounter()
  {
  }

  // Kopirajući konstruktor -- brojač se postavlja na
  // istu vrijednost kao u izvorniku.
  // Privatan! Služi samo kao podrška metodi getCopy
  private SimpleCounter( SimpleCounter cntr )
  {
    this.counterState = cntr.counterState;
  }

  public Counter getCopy()
  {
    return new SimpleCounter( this );
  }

  void setState( int newState )
  {
    this.counterState = newState;
  }
}
```

Metodu `getCopy`, nažalost, ne možemo riješiti općenito i staviti u apstraktnu klasu jer u svakoj klasi treba stvarati instancu upravo te klase.

Klasu `TextAnalyzer` jednostavno je prepraviti da koristi mehanizam kopiranja. Potrebna je samo sitna izmjena u konstruktoru[2]:

Ispis 2.29: Analizator teksta koji koristi kompoziciju

```
public class TextAnalyzer
{
  private TextFile file;
  private Counter cntr;

  public TextAnalyzer( TextFile file, Counter cntr )
  {
    this.file = file.getCopy();
    this.cntr = cntr.getCopy();
  }

  public int countLines()
  {
    while ( !file.endReached() )
    {
      file.readLine();
      this.cntr.increment();
    }
    return cntr.getState();
  }
}
```

[2]U konstruktoru se koristi i metoda `getCopy` iz klase `TextFile`. Njena implementacija nije prikazana u skripti.

Poglavlje 3

Razrada detalja jezičnih svojstava

3.1 Varijabla, referenca, objekt

Jedan od najbitnijih elemenata potrebnih za precizno razumijevanje jezičnih konstrukcija u Javi jest pojam varijable. Ono što će ovdje biti rečeno za varijablu vrijedi i za parametre metode, uhvaćene iznimke i atribute objekta. Jedna od razlika između lokalne varijable i atributa koju treba imati na umu je da se atribut inicijalizira automatski, a lokalna varijabla ne. Prevoditelj ("kompajler") će analizirati izvorni kod i prijaviti pogrešku ako kod omogućava čitanje varijable prije inicijalizacije.

Varijabla je spremnik u radnoj memoriji kojem je dodijeljeno ime (identifikator) i vrsta podatka koji se u nju sprema. Jezik Java razlikuje varijable *primitivne* i *referentne* vrste.

3.1.1 Varijabla primitivne vrste

Varijabla primitivne vrste je jednostavniji od dva slučaja i poznat je iz jezika C. Vrijednost takve varijable je podatak primitivne vrste, npr. `boolean`, `int` ili `double`. Sam podatak je izravno spremljen u varijablu i vrijednost varijable može se izmijeniti jednostavnom dodjelom:

```
int i; //deklaracija varijable
i = 2; //prva dodjela vrijednosti (inicijalizacija)
i = 3; //druga dodjela vrijednosti ...
```

Atributi primitivne vrste inicijaliziraju se na vrijednost nula odnosno, za logičku vrstu, na `false`. Varijable se uspoređuju po vrijednosti na uobičajen

način:

```
int i = 0;
int j = 0;
if ( i == j )
  System.out.println( "Varijable imaju istu vrijednost" );
else
  System.out.println( "Varijable imaju različite vrijednosti" );
```

(ispisat će se "Varijable imaju istu vrijednost"). Ako varijablu proslijedimo kao argument metodi, metoda će dobiti vlastitu kopiju vrijednosti varijable, tako da promjene na toj kopiji ne utječu na original:

```
private static void useInt( int i )
{
  i++;
}

public static void main( String[] args )
{
  int i = 0;
  useInt( i );
  System.out.println( i ); // ispisuje "0"
}
```

3.1.2 Varijabla referentne vrste

U varijablu referentne vrste sprema se *referenca* na objekt. Vrijednost varijable nikad nije sam objekt, već uvijek referenca na njega. To je važno imati na umu, pogotovo u sljedećim situacijama:

- Dvije varijable mogu imati referencu na isti objekt. Manipulacija objektom putem jedne varijable manifestirat će se i kad se objektu pristupa putem druge varijable:

  ```
  Counter cntr1 = new SimpleCounter();
  Counter cntr2 = cntr1;
  System.out.println( cntr1.getState() ); // ispisuje "0"
  cntr2.increment();
  System.out.println( cntr1.getState() ); // ispisuje "1"
  ```

- Ako se metodi prosljeđuje objekt putem parametra referentne vrste, akcije nad objektom obavljene unutar metode ostat će vidljive na objektu i nakon što metoda završi:

```
private static void useCntr( Counter cntr )
{
  cntr.increment();
}

public static void main( String[] args )
{
  Counter cntr1 = new SimpleCounter();
  System.out.println( cntr1.getState() ); // ispisuje "0"
  useCntr( cntr1 );
  System.out.println( cntr1.getState() ); // ispisuje "1"
}
```

- Vrijednost dviju varijabli je jednaka samo ako sadrže istu referencu, tj. pokazuju na *jedan te isti* objekt. Ako pokazuju svaka na svoj objekt, njihove vrijednosti neće biti jednake bez obzira na to jesu li sami objekti međusobno jednaki:

```
public static void main( String[] args )
{
  Counter cntr1 = new SimpleCounter();
  Counter cntr2 = new SimpleCounter();
  if ( cntr1 == cntr2 )
    System.out.println( "Varijable imaju istu vrijednost" );
  else
    System.out.println( "Varijable imaju različite vrijednosti" );
}
```

  (ispisat će se "Varijable imaju različite vrijednosti"). Za usporedbu samih objekata mora se koristiti posebnu metodu `equals` (odjeljak 4.2.1).

Varijabla referentne vrste može imati i posebnu vrijednost `null`, što znači da ne sadrži nikakvu referencu. Atributi referentne vrste se automatski inicijaliziraju na ovu vrijednost.

3.1.3 Opseg vidljivosti (*scope*) lokalne varijable

Lokalna varijabla uobičajeno se deklarira unutar glavnog bloka metode. Međutim, općenito njena vidljivost proteže se uvijek od mjesta deklaracije do kraja bloka u kojem je deklarirana:

```
public static void method()
{
  int i = 0;
```

```
  if ( i >= 0 )
  {
    int j = i + 1;
  }
  j++; // prevoditelj prijavljuje pogrešku da nije deklarirana
}
```

Nijedna deklaracija lokalne varijable ne smije "prekriti" drugu lokalnu varijablu koja je na tom mjestu vidljiva:

```
public static void method()
{
  int i = 0;
  if ( i >= 0 )
  {
    int i = 3; // prevoditelj se tuži da je varijabla 'i' već deklarirana
  }
}
```

Sintaksa petlje `for` omogućuje da se već prije otvorene vitice deklarira varijabla koja će vrijediti unutar bloka petlje:

```
public static void method()
{
  for ( int i = 0; i < 33; i++ )
  {
    System.out.println( i % 7 );
  }
  i++; // prevoditelj prijavljuje pogrešku da nije deklarirana
}
```

3.2 Primitivne vrste podataka

Jezik Java raspolaže ograničenim skupom primitivnih vrsta podataka. Ne raspolaže npr. brojevima bez predznaka. Definirane su sljedeće vrste:

- `boolean`: logička vrijednost, poprima jednu od dvije konstante: `true` ili `false`.
- `byte`: 8-bitni cijeli broj (zapisan u jednom oktetu).
- `short`: 16-bitni cijeli broj.
- `char`: 16-bitni Unicode-ov znak. Može se tretirati i kao broj. Podržane su doslovne vrijednosti, npr. `'a'`, `'\n'` za znak prelaska u novi redak, `'\t'` za znak tabulatora.

- `int`: 32-bitni cijeli broj, koristi se za sve cijele brojeve "opće namjene", kad nema posebno jakih razloga za korištenje nekog drugog.
- `long`: 64-bitni cijeli broj.
- `float`: 32-bitni broj s plivajućim zarezom. Preciznost je oko 6 decimalnih mjesta.
- `double`: 64-bitni broj s plivajućim zarezom. Preciznost je oko 15 decimalnih mjesta. Najbolje je rutinski koristiti `double` za decimalne brojeve.

3.3 Numerički i logički izrazi

Jezik Java raspolaže uobičajenim skupom unarnih i binarnih numeričkih i logičkih operatora. Ovdje se donosi njihov kratak pregled.

3.3.1 Numerički operatori

Unarni operatori:

- negacija broja, npr `-i`
- komplement broja bit-po-bit, npr. `~i`
- *pre-increment*, npr. `++i`
- *pre-decrement*, npr. `--i`
- *post-increment*, npr. `i++`
- *post-decrement*, npr. `i--`

Binarni operatori:

- zbroj, npr. `i + j`; isti znak koristi se i za operator ulančavanja (vidjeti 4.3)
- razlika, npr. `i - j`
- umnožak, npr. `i * j`

- količnik, npr. `i / j`. Ako su `i` i `j` cijeli brojevi, rezultat će biti cijeli broj puta koliko `j` stane u `i`. Drugim riječima, rezultat se zaokružuje prema nuli odnosno na manju apsolutnu vrijednost.
- ostatak dijeljenja, npr. `i % j`. Pravilo za rezultat operacije: `(i/j)*j + (i%j) = i`

Binarni operatori koji djeluju na razini bitova:

- pomak (*shift*) ulijevo, npr. `i << j`. Na ispražnjena desna mjesta stavljaju se nule.
- pomak udesno uz *sign-extension*, npr. `i >> j`. Na ispražnjena lijeva mjesta stavlja se vrijednost koju je imao najljeviji bit prije operacije. Koristi se kad se nad cijelim brojem, bio pozitivan ili negativan, želi postići efekt dijeljenja s 2^j.
- pomak udesno uz *zero-extension*, npr. `i >>> j`. Na ispražnjena lijeva mjesta stavljaju se nule. Za negativne brojeve narušava efekt dijeljenja potencijom broja dva.
- operacija *i*, npr. `i & j`
- operacija *ili*, npr. `i | j`
- operacija *isključivo ili*, npr. `i ^ j`

Za usporedbu brojeva koriste se operatori `==` (jednakost), `!=` (nejednakost), `<`, `>` `<=` i `<=`.

3.3.2 Logički operatori

Java striktno razlikuje numeričke od logičkih vrijednosti i izraza. Brojevnu vrijednost nemoguće je tretirati kao logičku i obratno. Dostupni su sljedeći operatori nad logičkim vrijednostima:

- negacija, npr. `~a`
- *i*, npr. `a & b`
- *ili*, npr. `a | b`

- *isključivo ili*, npr. `a ^ b`
- *uvjetno i*, npr. `a && b`. Izraz `b` će se izračunavati jedino ako je izraz `a` bio istinit.
- *uvjetno ili*, npr. `a || b`. Izraz `b` će se izračunavati jedino ako je izraz `a` bio lažan.

Osim za posebne namjene treba uvijek koristiti uvjetne verzije operatora jer je to računski ekonomičnije. Jedino u slučaju kad je nužno da oba izraza uvijek budu izračunata (zbog nekih dodatnih posljedica koje ti izrazi ostavljaju) treba koristiti bezuvjetne operatore.

3.3.3 Uvjetni (ternarni) operator

Ovaj operator služi za odabir između dva izraza od kojih će jedan biti rezultat čitave operacije ovisno o zadanom uvjetu. Tipičan primjer uporabe:

```
public static void showErrorMessage( String msg )
{
  String fullMsg = "Error: " + ( msg != null? msg : "" );
  System.out.println( fullMsg );
}
```

Ternarni operator koristi se u izrazu `msg != null? msg : ""`. Prvo se provjerava uvjet `msg != null`. Ako je uvjet zadovoljen, rezultat čitavog izraza je `msg`, a u protivnom je rezultat prazan niz, `""`.

Ovaj operator **nije primjereno koristiti** kao općenitu zamjenu za konstrukciju *if-then-else*, već samo u slučajevima kao gore, gdje se radi o sasvim kratkim izrazima i gdje se time postiže bolja čitljivost koda.

3.3.4 Operatori složene dodjele

Za sve binarne operatore za koje je to smisleno postoje i odgovarajući operatori složene dodjele, npr. `i += 3` postojećoj vrijednosti varijable `i` dodaje 3, `a &= b` varijabli `a` pridjeljuje rezultat operacije `a & b`, itd.

3.3.5 Pretvorbe između numeričkih vrsta

Ako se u numeričkom izrazu koriste podaci različitih vrsta – npr. `int` i `long`, svi podaci niže preciznosti bit će implicitno pretvoreni u podatke više preciznosti. Razmotrimo ovaj primjer korištenja cijelih brojeva:

Ispis 3.1: Implicitna pretvorba vrste cjelobrojnih podataka

```
int int1 = 1;
long long1 = 1;

long long2 = int1;
long2 = int1 + long1;

int int2 = long1; // pogreška!
int2 = int1 + long1; // pogreška!
```

U trećem i četvrtom retku vrijednost vrste `int` se pridjeljuje varijabli vrste `long` i to je ispravno. Međutim, u petom i šestom retku pokušava se vrijednost vrste `long` pridijeliti varijabli vrste `int`. Ovdje prevoditelj prijavljuje pogrešku "`Type mismatch: cannot convert from long to int`".

Java općenito poštuje sljedeće pravilo: ako zatreba pretvorba na veću preciznost, odvija se automatski; ako zatreba pretvorba na manju preciznost, potrebno je eksplicitno zatražiti konverziju. Pogreške iz gornjeg primjera ispravit ćemo ovako:

Ispis 3.2: Eksplicitna pretvorba vrste cjelobrojnih podataka

```
int int2 = (int) long1;
int2 = (int) ( int1 + long1 );
```

Sad prevoditelj neće prijaviti pogrešku, ali naravno treba imati na umu opasnost od prekoračenja maksimalnog opsega `int`-a. Sve ovdje rečeno vrijedi i za odnose između `float` i `double`, kao i između `float` i `double` s jedne strane i cjelobrojnih vrsta s druge strane.

3.4 Enumeracija

U softverskom razvoju vrlo čest slučaj je da neki podatak može poprimiti svega nekoliko različitih vrijednosti. Na primjer, ako razvijamo neku kartašku igru, trebat će podatak s četiri moguće vrijednosti (*pik*, *karo*, *herc*, *tref*). Za takve slučajeve treba definirati posebnu klasu, tzv. *enumeraciju*. Najosnovniji slučaj je vrlo jednostavan:

Ispis 3.3: Jednostavna enumeracija

```
package hr.fer.tel.ilj;

public enum Suit
```

```
{
  clubs, diamonds, hearts, spades;
}
```

Enumeracija je ustvari obična klasa s javnim konstantama (vidjeti odjeljak 3.8) čija vrsta je upravo ta klasa. To možemo vidjeti na sljedećem primjeru korištenja enumeracije:

Ispis 3.4: Osnovno korištenje enumeracije

```
public static void main( String[] args )
{
  Suit suit1, suit2;
  suit1 = Suit.clubs;
  suit2 = Suit.hearts;
  System.out.println( "Prva boja: " + suit1 + "; druga boja: " + suit2 );
}
```

Odmah možemo uočiti da članovi enumeracije imaju ispravno implementiranu metodu `toString` koja vraća ime člana.

Konstrukcija `switch` općenito se ne može koristiti za referentne vrste, ali posebno je omogućena za enumeraciju, što znatno olakšava zapis koda kao u sljedećem primjeru:

Ispis 3.5: Korištenje enumeracije u bloku `switch`

```
  static void printSuitInCroatian( Suit suit )
  {
    String suitName;
    switch ( suit )
    {
      case clubs:    suitName = "tref";   break;
      case diamonds: suitName = "karo";   break;
      case hearts:   suitName = "herc";   break;
      case spades:   suitName = "pik";    break;
      default:       suitName = "";
    }
    System.out.println( "Boja karte je " + suitName );
  }

  public static void main( String[] args )
  {
    printSuitInCroatian( Suit.clubs );
    printSuitInCroatian( Suit.hearts );
  }
```

Da nema mogućnosti korištenja bloka `switch`, morali bismo pisati ovako:

```
if ( suit == Suit.clubs )
  suitName = "tref";
else if ( suit == Suit.diamonds )
  suitName = "karo";
else if ( suit == Suit.hearts )
  suitName = "herc";
else if ( suit == Suit.spades )
  suitName = "pik";
else
  suitName = "";
```

Primijetimo da je članove enumeracije uvijek sigurno uspoređivati po referenci, operatorom ==, jer je njihova priroda takva da su svaka dva distinktna člana međusobno različiti. Dodatna prednost konstrukcije `switch` je u tome što omogućuje prevoditelju da nas upozori ako smo slučajno izostavili neki član enumeracije.

Sve enumeracije implementiraju i sučelje `Comparable` (vidjeti odjeljak 6.8.1). Između članova enumeracije utvrđen je poredak koji odgovara poretku kojim su navedeni u definiciji enumeracije. To nam omogućuje da napišemo npr. ovakav kod:

```
public static void main( String[] args )
{
  Suit suit1, suit2;
  suit1 = Suit.clubs;
  suit2 = Suit.hearts;
  int difference = suit1.compareTo( suit2 );
  if ( difference == 0 )
    System.out.println( "Boje su iste" );
  else if ( difference > 0 )
    System.out.println( "Prva boja pobjeđuje!" );
  else
    System.out.println( "Druga boja pobjeđuje!" );
}
```

Kao i ostale klase, i enumeraciju je moguće proširivati dodatnim atributima i metodama. U našem slučaju mogli bismo npr. dodati hrvatsko ime boje:

Ispis 3.6: Obogaćena enumeracija

```
package hr.fer.tel.ilj;

public enum Suit
{
  clubs( "tref" ),
  diamonds( "karo" ),
```

```
  hearts(  "herc" ),
  spades( "pik" );

  private String croName;

  private Suit( String name )
  {
    this.croName = name;
  }

  public String getCroName()
  {
    return this.croName;
  }
}
```

Ovdje smo dodali instancin atribut `croName`, odgovarajući (privatni!) konstruktor i metodu `getCroName`. Uz ovako definiranu enumeraciju izvedba metode `printSuitInCroatian` postaje trivijalna:

```
  static void printSuitInCroatian( Suit suit )
  {
    System.out.println( "Boja karte je " + suit.getCroName() );
  }
```

3.5 Konstrukcije za upravljanje slijedom izvršavanja

Jezik Java raspolaže upravljačkim konstrukcijama poznatim iz jezika C: `if`, `while`, `do-while`, `for` i `switch`. Ovdje će one biti prikazane samo u najkraćim crtama, kroz minimalistički primjer.

Ispis 3.7: Kratki primjeri uporabe upravljačkih konstrukcija

```
  static void controlStructures()
  {
    int j = 0, k = 1;

    if ( j == k )
    {
      k++;
    }
    else if ( j < k )
    {
      j++;
    }
    else
```

```
      k = 0;
//-------------------------------------
    for ( int i = 0; i < 5; i++ )
    {
      if ( i == j - 2 )
        continue;
      j += i;
      if ( j > 3 )
        break;
    }
//-------------------------------------
    while ( j < 13 )
      j++;
//-------------------------------------
    do
    {
      k++;
    } while ( k < 13 );
//-------------------------------------
    switch ( j )
    {
    case 0:
      k++;
      break;
    case 1:
      j++;
      break;
    default:
      j = k - j;
    }
  }
```

Posebnu pozornost zahtijevaju jedino naredbe za izvanredan prekid petlje (`break`) ili izravan pomak na sljedeći korak petlje (`continue`), pogotovo u kombinaciji s ugniježđenim petljama. Korištenjem *oznaka* (eng. *label*) moguće je npr. u unutarnjoj petlji narediti izlazak iz vanjske petlje. To je demonstrirano ovim primjerom:

Ispis 3.8: Izjave `break` i `continue` koje koriste oznake

```
  static void breakContinue()
  {
    outerLoop:
    for ( int i = 1; i < 4; i++ )
    {
      for ( int j = 1; j < 4; j++ )
      {
        if ( i - j == -1 )
          continue outerLoop;
        if ( i % j == 1 )
          break outerLoop;
```

```
        System.out.println( "i = " + i + ", j = " + j );
      }
    }
```

Očekivani ispis je:

```
i = 1, j = 1
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
```

Ove konstrukcije uvedene su u jezik Javu kao zamjena za najčešći slučaj korištenja naredbe `goto`, koja ne postoji u jeziku.

3.6 Sve razine dostupnosti

Dosad smo se susreli s tri razine dostupnosti:

- *privatna* – član je dostupan samo u klasi gdje je definiran;
- *paketno-privatna* – član je dostupan iz svih klasa u istom paketu;
- *javna* – član je dostupan bez ograničenja.

Java poznaje još jednu razinu, širu od paketno-privatne – koja dodatno dozvoljava dostup iz *svake podklase*, bez obzira u kojem paketu se nalazila. Pristup je još uvijek zabranjen klasama iz drugih paketa ako nisu podklase. Ova razina dostupnosti naznačuje se ključnom riječi `protected`. Koristi se prvenstveno u klasi koja je posebno namijenjena da korisnik paketa iz nje izvodi svoje klase. Sveukupno, razine dostupnosti mogu se sistematizirati tablicom 3.1.

Tablica 3.1: Pregled razina dostupnosti

razina (eng.)	klasa	paket	podklase	ostali
private	o			
package-private	o	o		
protected	o	o	o	
public	o	o	o	o

3.6.1 Model klasne enkapsulacije

Java koristi model enkapsulacije koji je zasnovan ne na instanci, već na klasi. Ako je npr. neki atribut označen kao `private`, to znači da je on privatan za sve instance te klase i nedostupan instancama drugih klasa. Drugim riječima, sve instance iste klase međusobno imaju neograničen dostup atributima i metodama. Na primjer, ovaj kod neće proizvesti prevoditeljsku pogrešku:

Ispis 3.9: Model klasne enkapsulacije

```
class Abc {
  private int i;

  public void manipulateAnotherObject( Abc anotherObject )
  {
    anotherObject.i++;
  }
}
```

Ovakav model, iako nije striktno u duhu paradigme objekata s nedodirljivom unutrašnjosti, u praksi je opravdan jer je sav kod koji može manipulirati privatnim dijelovima drugog objekta lociran na istom mjestu pa se ne mogu pojaviti problemi zbog kojih je i uvedena enkapsulacija – da kod koji koristi neki objekt prestane funkcionirati zbog izmjena na njegovim unutarnjim detaljima.

3.7 Statička metoda – izuzeta od polimorfizma

Modifikator `static` već je usputno spomenut i korišten u ispisima koda uz posebnu metodu `main`. Njime je moguće označiti i bilo koju drugu metodu. Time se postiže da se takva metoda ne poziva nad nekom instancom, već samostalno. Dakle, ne prosljeđuje joj se implicitni argument `this` (vidjeti odjeljak 3.11). Argument `this` ključan je za funkcioniranje polimorfizma i bez njega ne može doći do dinamičkog povezivanja.

Takva metoda odgovara pojmu *funkcije* iz proceduralne paradigme (kao npr. u jezicima C ili Pascal). Za statičke metode koristi se statičko povezivanje – tijekom izvršavanja ne donosi se odluka koju metodu pozvati, to je fiksirano u trenutku prevođenja. Za statičke metode ponekad se koristi i

izraz *metoda klase* odnosno *klasina metoda* jer je vezana uz klasu, a ne uz njene instance.

U ovoj skripti ponekad će se koristiti statičke metode, u primjerima za koje dinamičko povezivanje nije bitno. Područja primjene u praksi:

- Kad su parametri metode instance "tuđih" klasa – koje samo koristimo, a ne razvijamo pa nemamo dostup njihovom izvornom kodu i ne možemo dodati svoju metodu. Takva primjena značajno je otežana gubitkom dinamičkog povezivanja.
- Kad su svi parametri primitivne vrste. U tom slučaju dinamičko povezivanje ionako ne dolazi u obzir. Ovakva primjena prikazana je u primjeru koji slijedi malo niže.
- Kad metoda ne ovisi ni o jednom parametru. Tada jednostavno nema razloga zašto ne bi bila statička, a statičko povezivanje je efikasnije od dinamičkog.

Studenti ponekad, ugledajući prevoditeljevu pogrešku "`Instance method used in a static context`", problem "otklanjaju" proglašavanjem metode statičkom i time zapravo samo stvaraju dodatne probleme. Pogrešku treba otkloniti tako da se jasno ustanovi nad kojom instancom je metoda trebala biti pozvana i da se kod ispravi u skladu s time.

Ako se statička metoda poziva iz neke druge klase, poziv se zapisuje u obliku `Klasa.statičkaMetoda`:

Ispis 3.10: Poziv statičke metode

```
public class BooleanFunctions
{
  public static boolean implies( boolean left, boolean right )
  {
    return !( left && !right ) ;
  }
}

public class Executable
{
  public static void main( String[] args )
  {
    boolean a = true, b = false;
    boolean result = BooleanFunctions.implies( a, b );
    System.out.println( "" + a + " implies " + b + "? " + result );
  }
}
```

Ispis:

```
true implies false? false
```

Specifikacija jezika Java [5] dozvoljava i poziv statičke metode istom sintaksom kao da je instancina metoda, npr:

Ispis 3.11: Poziv statičke metode sintaksom kao da je instancina metoda

```
public class Executable
{
  public static void main( String[] args )
  {
    boolean a = true, b = false;
    BooleanFunctions bf = null;
    boolean result = bf.implies( a, b );
    System.out.println( "" + a + " implies " + b + "? " + result );
  }
}
```

Prevoditelj utvrđuje vrstu izraza lijevo od točke i ta vrsta mora biti klasa u kojoj je definirana dotična metoda. Ovaj način zapisa **ne treba nikada koristiti** jer samo zavarava, a ne donosi nikakve dodatne prednosti (nema polimorfizma). Primijetimo da je u gornjem primjeru metoda pozvana putem varijable čija vrijednost je `null`, čime se jasno demonstrira da se potpuno ignorira vrijednost varijable, bitna je samo njena vrsta.

3.8 Statički atribut, konstanta

Slično kao metoda, atribut isto može biti statički i time nevezan uz instancu. Postoji samo jedna, globalna vrijednost takvog atributa. Jedna od najčešćih uporaba statičkih atributa je za definiranje *konstanti*, npr. mogli bismo u konstanti definirati podrazumijevanu duljinu ciklusa brojača `ModuloCounter`:

Ispis 3.12: Statički atribut, javna konstanta

```
public class ModuloCounter
{
  public static final int DEFAULT_MODULUS = 10;

  private int modulus;

  public ModuloCounter()
  {
    this.modulus = DEFAULT_MODULUS;
```

```
  }

  ...
}
```

3.8.1 Modifikator `final` – rekapitulacija

Pri definiranju konstante pojavljuje se i modifikator `final`. U odjeljku 2.14 objašnjeno je kako se on primjenjuje na metode i klase radi kontrole polimorfizma. Na atribute i varijable djeluje tako da zabranjuje promjenu njihove vrijednosti nakon inicijalizacije. Njime se, ako zatreba, može označiti i parametar metode, npr.

```
public void increment( final int step )
{ ... }
```

Korištenjem modifikatora `final` na svim varijablama gdje je to smisleno povećava se čitljivost koda jer se time jasno naglašava da se ustvari radi o konstanti. U praksi velik broj, a možda i većina varijabli imaju ulogu konstante.

3.9 Pravila oblikovanja identifikatora

U jeziku Java na snazi su neformalna, ali vrlo snažna pravila oblikovanja identifikatora, od kojih se ne smije odudarati. Najčvršća pravila tiču se rasporeda velikih i malih slova:

- U imenu *paketa* **zabranjeno je koristiti velika slova**. Npr. ime paketa korištenog u primjerima je `hr.fer.tel.ilj`.
- Ime *referentne vrste* podatka (klasa i sučelje) **obavezno počinje velikim slovom**. Svaka daljnja riječ u imenu ponovo počinje velikim slovom. Npr. `SimpleCounter`.
- Ime *lokalne varijable, parametra, uhvaćene iznimke, atributa i metode* **obavezno počinje malim slovom**. Svaka daljnja riječ u imenu počinje velikim slovom. Npr. `counterState`.
- Poseban slučaj su *javne konstante*, koje se pišu **velikim slovima**, a riječi se odvajaju podvlakom. Npr. `DEFAULT_MODULUS`.

Za sve identifikatore koristi se **isključivo engleski jezik**. Za sučelje se uvijek odabire najkraće, najčišće ime – bez posebnih prefiksa ili sufiksa – kao u našem primjeru `Counter`. To je stoga što se upravo ime sučelja najviše koristi u kodu. Ime apstraktne klase u vrhu hijerarhije uobičajeno je napraviti od imena sučelja s prefiksom `Abstract` – npr. `AbstractCounter`. Konkretne klase uobičajeno zadržavaju ime svog sučelja na zadnjem mjestu, a naprijed se dodaju riječi koje pobliže opisuju o kakvoj implementaciji se radi – npr. `ModuloCounter`.

3.10 *Upcast* i *downcast*

Općenito, pojam *type cast* odnosi se na tretiranje podatka jedne vrste kao da je neke druge. U nekim drugim jezicima, prije svega C-u, moguće je *cast*-ati između vrsta na različite načine – na primjer, cijeli broj moguće je *cast*-ati u pokazivač. Za referentne vrste jedini dozvoljeni oblik *cast*-anja je unutar klasne hijerarhije – ili prema gore (*upcast*), ili prema dolje (*downcast*).

Slučaj *upcast*-a je jednostavniji i događa se automatski. Tako se naziva situacija s kojom smo se već sreli – na primjer, kad se instanca klase `SimpleCounter` tretira kao instanca sučelja `Counter` (u ispisu 2.16). Isto se događa i kad imamo dvije klase u odnosu nadklasa-podklasa. Dakle, *upcast* je tehničko sredstvo koje omogućava transparentno korištenje polimorfizma. Njega je uvijek sigurno provesti – ne može se dogoditi da instanca podklase nema neku metodu definiranu za nadklasu.

Java dozvoljava i *cast*-anje u suprotnom smjeru, prema dolje po hijerarhiji – *downcast*. Njega se ne koristi da bismo instancu nadklase tretirali kao instancu podklase – dapače, to bi prouzročilo pogrešku jer podklasa može imati dodatne metode koje nadklasa nema. *Downcast* služi da bi se npr. u metodi koja prima općeniti parametar (npr. `Counter`) ipak mogla koristiti puna funkcionalnost instance konkretne klase (npr. `ModuloCounter`). Uzmimo konkretan primjer – recimo da klasa `ModuloCounter` (ispis 2.24) ima dodatnu metodu `setModulus` koja nije definirana u sučelju `Counter` jer je specifična za slučaj cikličkog brojača:[1]

[1]Konstruktor je ovdje izmijenjen u odnosu na raniji ispis. To je bilo nužno jer se konstruktor `AbstractCounter( int )` oslanja na metodu `setState`, a ovdje njena implementacija čita atribut `modulus`, koji u tom trenutku još nije inicijaliziran.

Ispis 3.13: `ModuloCounter` s dodatnom metodom `setModulus`

```
package hr.fer.tel.ilj;

public class ModuloCounter
extends AbstractCounter
{
  private int modulus;

  public ModuloCounter( int initialState )
  {
    super();
    this.modulus = 10;
    setState( initialState );
  }

  public ModuloCounter()
  {
  }

  public void setModulus( int newModulus )
  {
    this.modulus = newModulus;
  }

  void setState( int newState )
  {
    this.counterState = newState % this.modulus;
  }
}
```

Nadalje, recimo da negdje imamo metodu koja prima parametar općenite vrste `Counter`, ali u slučaju da se radi o cikličkom brojaču, želi postaviti duljinu ciklusa na potrebnu vrijednost:

Ispis 3.14: Metoda koja upravlja `ModuloCounter`-om

```
  public static void useCounter( Counter cntr )
  {
    ModuloCounter modCntr = (ModuloCounter) cntr;
    modCntr.setModulus( 7 );
  }
```

Varijabli vrste `ModuloCounter` pridjeljuje se vrijednost parametra vrste `Counter` i stoga je potrebno eksplicitno provesti *downcast* izrazom

```
(ModuloCounter) cntr
```

3.10.1 Operator `instanceof`

U slučaju da obavimo *downcast* nad objektom koji nije instanca dotične klase (ili eventualno njena podklasa), tijekom izvršavanja programa bit će prijavljena pogreška. U trenutku kompilacije općenito nije moguće ustanoviti hoće li se to dogoditi. Na primjer, ako imamo ovakav poziv metode `useCounter`:

```
useCounter( new SimpleCounter() );
```

jasno je da mora doći do pogreške jer klasa `SimpleCounter` ne definira metodu `setModulus`. Java neće dozvoliti ni da kod stigne do tog poziva jer će prijaviti pogrešku već prilikom pokušaja *downcast*-anja. Prekid izvršavanja zbog pogreške može se izbjeći obavljanjem prethodne provjere operatorom `instanceof`:

Ispis 3.15: Dorađena metoda `useCounter`

```
public static void useCounter( Counter cntr )
{
  if ( cntr instanceof ModuloCounter )
  {
    ModuloCounter modCntr = (ModuloCounter) cntr;
    modCntr.setModulus( 7 );
  }
}
```

Prije nego što se obavi *downcast*, provjerava se je li proslijeđen objekt zaista instanca klase `ModuloCounter`. Operator `instanceof` s lijeve strane općenito ima neki izraz čija vrijednost je referentne vrste, a s desne strane doslovno navedeno ime neke referentne vrste (klase ili sučelja). Operator vraća `true` ako je lijevi operand instanca desnog operanda ili njegove podklase. Odnosno, ako je desni operand sučelje, provjerava je li lijevi operand objekt koji implementira to sučelje. U praksi, u većini slučajeva gdje se radi *downcast* potrebna je ovakva provjera.

3.11 Implicitni parametar metode – `this`

Već smo se na samom početku, u ispisu 2.4, susreli s ključnom riječi `this` pomoću koje smo u kodu metode pristupali atributima objekta nad kojim je ona pozvana. Formalno, svaka instancina metoda (metoda objekta, tj. svaka koja nije statička), osim eksplicitno navedenih parametara, prima

implicitno i dodatni parametar čije ime je `this`, vrsta mu je klasa u kojoj se metoda nalazi, a vrijednost referenca na objekt nad kojim je metoda pozvana. Na primjer, metodu `increment` iz ispisa 2.19 (deklaracija klase `SimpleCounter`) mogli smo zapisati i ovako, zadržavajući identično ponašanje:

Ispis 3.16: Implicitni parametar `this`

```
public void increment()
{
  SimpleCounter cntr = this;
  cntr.counterState++;
}
```

Prilikom poziva metode, npr. `cntr1.increment()` u ispisu 2.7, referenca iz varijable `cntr1` prepisujse se u parametar `this`.

3.11.1 Parametar `this` i polimorfizam

Ono što je najvažnije za parametar `this` je da upravo on omogućuje funkcioniranje polimorfizma jer se samo za njega provodi dinamičko povezivanje. Tijekom izvršavanja programa prije svakog poziva metode prvo se provjerava točna vrsta objekta na koji će pokazivati `this`. Na osnovu te vrste donosi se odluka koju metodu pozvati. Za sve ostale, "obične" parametre vrsta se *ne utvrđuje* prilikom izvršavanja, već samo prilikom prevođenja i to na osnovu *deklarirane* vrste izraza koji se pojavljuju kao argumenti metode. Drugim riječima, statički se ustanovljuje *potpis* pozvane metode (vidjeti odjeljak 2.15).

Bez ovog saznanja mogli bismo pomisliti da su npr. ova dva pristupa definiranju metode `increment` ekvivalentni (modifikator `static` naređuje da se ne proslijedi implicitni parametar `this`, vidjeti odjeljak 3.7):

Ispis 3.17: Neuspio pokušaj uvođenja eksplicitnog parametra `this`

```
public interface Counter
{
  void increment();
  void incrementWithExplicitThis( Counter explicitThis );
  ... ostale metode ...
}


public class SimpleCounter
implements Counter
{
```

```
  private int counterState;

  public void increment()
  {
    this.counterState++;
  }

  public static void incrementWithExplicitThis( Counter explicitThis )
  {
    explicitThis.counterState++;
  }

  ... ostale metode ...
}
```

Metode bismo mogli pokušati koristiti ovako:

Ispis 3.18: Neuspio pokušaj korištenja metode s eksplicitnim `this`

```
  public static void main( String[] args )
  {
    Counter cntr = new SimpleCounter();
    cntr.increment();
    incrementWithExplicitThis( cntr );
  }
```

Naoko, situacija je ekvivalentna – prvi put se argument metode navodi lijevo od točke, a drugi put kao normalni argument u zagradama. Međutim, uvođenje metode `incrementWithExplicitThis` ustvari generira čitav niz prevoditeljevih pogrešaka, sve zbog toga što Java ne provodi dinamičko povezivanje nad normalnim argumentima.

Prilikom prevođenja poziva `incrementwithExplicitThis( cntr )` utvrđuje se da je deklarirana (statička) vrsta argumenta `cntr` jednaka `Counter`. Tu se nigdje ne pojavljuje prava vrsta objekta, `SimpleCounter`, i Java uopće neće uzeti u obzir tu klasu. Dakle, nemoguće je da će se pozvati metoda iz klase `SimpleCounter`. Vrsta `Counter` je ustvari *sučelje* u kojem uopće nema definicija metoda. Jasno je dakle da je ovakva situacija nemoguća i da mora uzrokovati prevoditeljevu pogrešku. U praksi, prevoditeljeva pogreška pojavit će se već čim metodu `incrementwithExplicitThis` označimo kao `static` jer je to metoda sučelja, a metode sučelja nikad nisu statičke – iz upravo objašnjenog razloga.

Nadalje, čak i da se svi ovi problemi na neki neobjašnjeni način riješe i da zaista bude pozvana metoda `incrementwithExplicitThis` u klasi

`SimpleCounter`, tamo stoji izjava `explicitThis.counterState++`. Parametar `explicitThis` je vrste `Counter`, a u toj vrsti ne spominje se nikakav atribut `counterState` i stoga će se i za to prijaviti prevoditeljeva pogreška.

Još jedan primjer u kojem se očituje razlika između parametra `this` i ostalih može se vidjeti u raspravi u odjeljku 4.2.1.

Pojam koji se u struci koristi za proces dinamičkog povezivanja je *method dispatch*. Za Javu se stoga kaže da podržava samo *single dispatch* jer obavlja *dispatch* po samo jednom parametru metode. Suprotan pojam je *multiple dispatch*.

Stručni pojam za jezično svojstvo da se prilikom prevođenja, tj. statički utvrđuju vrste svih izraza je *static typing*. Java je, dakle, *statically typed* jezik. Suprotan pojam je *dynamic typing*.

3.12 Detalji o hijerarhiji klasa

3.12.1 Korijenska klasa `Object`

U 2. poglavlju već se govorilo o izvedenim klasama. U Javi je svaka klasa izvedena iz točno jedne klase, osim posebne ugrađene klase `java.lang.Object`, koja nema nadklasu. Ako nacrtamo detaljan UML-ov dijagram klasa prikazujući sve veze "izvedena iz", dobit ćemo *stablo* čiji korijen je klasa `java.lang.Object`. Ako deklaracija klase ne navodi eksplicitno iz koje je klase izvedena, podrazumijeva se "`extends java.lang.Object`". Klasa `Object` definira i neke metode koje stoga imaju svi objekti (vidjeti 4.2).

Graf odnosa "implementira sučelje" s druge strane, jest hijerarhija, ali nije stablo. Jedna klasa može implementirati proizvoljan broj sučelja, a i sučelje može biti izvedeno iz proizvoljnog broja drugih sučelja.

3.12.2 Definicije izraza vezanih uz hijerarhiju

Kad se priča o hijerarhijskim odnosima između klasa, treba imati na umu točna značenja izraza kao što su *nadklasa* i *podklasa*. Vrijede sljedeće definicije:

- *Nadklasa* klase `C` je klasa `B` iz koje je ona izvedena, ali također i klasa `A` iz koje je izvedena `B`, itd. Koristeći izraze iz genealogije, *nadklasa* odgovara pojmu *pretka*. Klasa `Object` je, dakle, nadklasa svim ostalim Javinim klasama.

- *Podklasa* klase `A` je, analogno, svaka klasa-*potomak*, tj. svaka klasa `B` kojoj je `A` nadklasa.
- Klasa-*roditelj* klase `C` je njena *izravna nadklasa* – ona iz koje je izvedena.
- Klasa-*dijete* klase `C` je svaka klasa koja je iz nje izvedena, tj. kojoj je klasa `C` roditelj.

3.12.3 Ulančavanje poziva konstruktora

Svaki konstruktor, prije nego što se izvrši njegov kod, dužan je prvo pozvati neki konstruktor svoje nadklase. To se događa ili eksplicitno, korištenjem ključne riječi `super`, ili, u odsustvu toga, implicitno se dodaje poziv podrazumijevanog konstruktora nadklase (`super()`). Posljedica toga je da se konstruktori izvršavaju *ulančano*, tako da se prvo uvijek izvrši konstruktor klase `Object`, zatim kontruktor neke njene izravne podklase, itd. sve do konkretne klase koja se instancira. Zbog toga čak i za apstraktne klase, koje se ne smiju instancirati, vrijede ista pravila o posjedovanju bar jednog konstruktora. Bez toga bi bilo nemoguće instancirati njihove podklase.

Imajući rečeno u vidu, možemo uvidjeti da je puni oblik dvaju konstruktora u ispisu 2.23 ustvari ovakav:

Ispis 3.19: Eksplicitan poziv konstruktora nadklase

```
public AbstractCounter( int initialState )
{
  super();
  setState( initialState );
}

public AbstractCounter()
{
  super();
}
```

Implicitne pozive treba uvijek imati na umu, posebice stoga što nadklasa *ne mora* posjedovati podrazumijevani konstruktor. Ako dakle imamo konstruktor s izostavljenim pozivom konstruktora nadklase, a nadklasa nema podrazumijevani konstruktor, doći će do pogreške prilikom prevođenja koja može biti prilično zbunjujuća jer se dogodila u "praznom prostoru" u koji je implicitno dodan izostavljeni poziv `super()`.

Konstruktor može umjesto iz nadklase pozvati i neki drugi konstruktor iz iste klase. Time je omogućena višestruka iskoristivost koda u više konstruktora. U konačnici, nakon što se konstruktori klase međusobno "dogovore" koji će se prvi izvršiti, taj konstruktor opet poziva konstruktor iz nadklase. Poziv "susjednog" konstruktora postiže se korištenjem ključne riječi `this` umjesto `super`. Na primjer, konstruktore u ispisu 2.23 mogli smo definirati i ovako:

Ispis 3.20: Poziv konstruktora iste klase

```
public AbstractCounter( int initialState )
{
  super();
  setState( initialState );
}

public AbstractCounter()
{
  this( 0 );
}
```

Korištenjem podrazumijevanog konstruktora `AbstractCounter()` prvo će se izvršiti podrazumijevani konstruktor nadklase (`super()`), zatim ostatak konstruktora s jednim parametrom, a zatim ostatak podrazumijevanog konstruktora (u ovom slučaju nema daljnjih akcija u njemu).

Napomenimo na kraju da korištenje ključnih riječi `super` i `this` za pozive konstruktora nema nikakve veze s korištenjem istih ključnih riječi u smislu reference na trenutni objekt i pozivanja metode nadklase. Radi se samo o štednji na ključnim riječima njihovim višestrukim korištenjem u jeziku. Svaka dodatna ključna riječ znači još jedan zabranjeni identifikator za varijable, metode itd.

3.13 Java kao softverska platforma

Pod pojmom "Java" ne podrazumijeva se samo programski jezik i njegova biblioteka klasa, nego i cjelokupna softverska platforma, čiji temelj je Javin virtualni stroj.

Programi napisani u Javi ne prevode se, kao što je inače uobičajeno, u izvršni kod koji se izravno može izvršavati na procesoru računala, već u posebnu vrstu izvršnog koda za Javin virtualni stroj (JVM), koji onda konkretno računalo emulira. Taj poseban binarni kod naziva se *međukodom*

ili *byte*-kodom (eng. *bytecode*). Cilj toga je postizanje neovisnosti o računalskoj platformi: kod se kompilira jednom, a izvršava na bilo kojem stroju koji ima implementaciju JVM-a, bez obzira na vrstu procesora. Također se tehnički omogućuje i bolja kontrola nad interakcijom između Javinog programa i računala na kojem se izvršava. Lakše je moguće blokirati svaki pokušaj pristupa osjetljivim resursima i tako onemogućiti zlonamjerni kod. Više o ovome može se doznati na [6].

Sve ovo potječe od prvotnog cilja Jave da služi za tzv. *embedded applications*, aplikacije koje se učitavaju unutar webskih stranica. Čim se nešto implicitno učitava putem Weba, otvara se prostor za napade na računalo pisanjem zlonamjernog koda. U međuvremenu se Java počela masovno koristiti i za pisanje običnih, samostojnih aplikacija, gdje je pitanje sigurnosti manje bitno. Na primjer, čitavo razvojno okružje *Eclipse*, kao i Borlandov *JBuilder*, napisani su u Javi.

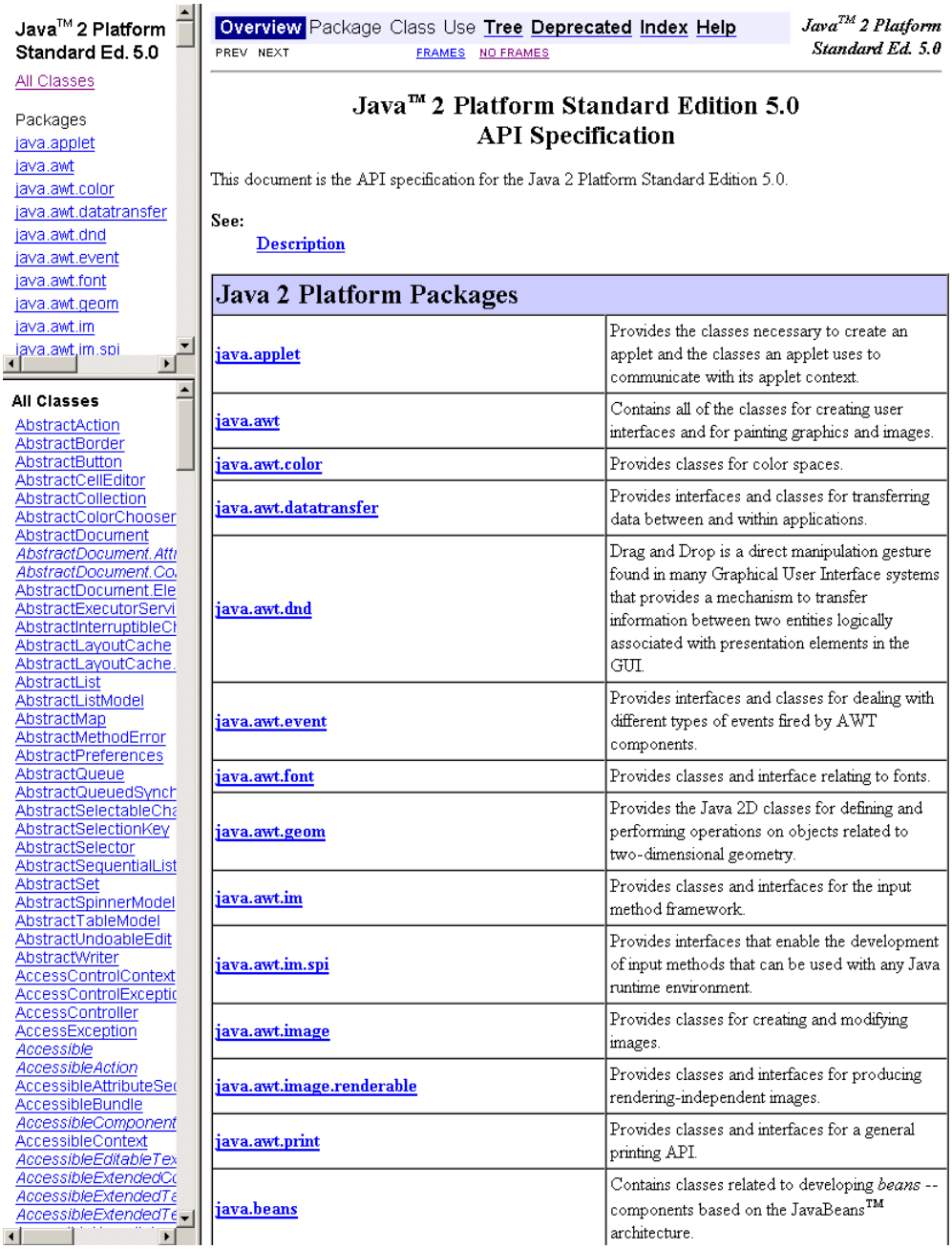
Poglavlje 4

Najvažnije ugrađene klase

Već je rečeno da jezik Java sadrži i veći broj ugrađenih klasa i sučelja. Također je istaknut i jedan od ugrađenih paketa koji je posebno usko povezan s jezikom – paket `java.lang`. Ovdje će biti izloženi detalji nekoliko najvažnijih klasa iz tog paketa. Za daljnje detalje treba se obratiti dokumentaciji Javine biblioteke klasa dostupne na Webu [1].

4.1 Uvodne upute za snalaženje u Javinoj dokumentaciji API-ja

Javin sustav za HTML-ovsku dokumentaciju zove se *Javadoc*. Na primjer, na upravo spomenutoj internetskoj lokaciji, [1], nalazi se *Javadoc*-ova dokumentacija cjelokupne Javine biblioteke klasa. Kad otvorimo tu lokaciju u pregledniku Weba, dobit ćemo prikaz kao na slici 4.1. Pregled treba započeti u okviru gore lijevo. Tamo je popis svih paketa. Treba kliknuti na paket od interesa, u ovom slučaju `java.lang`. Time se u okviru dolje lijevo sužava izbor i prikazuju se samo klase iz dotičnog paketa. U tom okviru zatim možemo naći klasu koja nas zanima, npr. `Object`. U glavnom okviru (desno) pojavljuje se dokumentacija tražene klase. Na vrhu je općeniti opis klase, a zatim slijedi popis članova – konstruktora, atributa i metoda. Prvo su u tablicama pobrojani svi članovi s kratkom napomenom. Klikom na ime člana odlazi se dublje u stranicu, gdje je njegov detaljan opis. Na slici 4.2 prikazana je detaljna dokumentacija za metodu `equals`.



Slika 4.1: Početna stranica Javadoc-a

Slika 4.2: Dokumentacija za metodu `equals`

4.2 Klasa `java.lang.Object`

Klasa `Object` je u korijenu Javine hijerarhije klasa, kao što je napomenuto u odjeljku 3.12.1. Ovdje će se detaljnije govoriti o toj klasi. To nije – kao što bi se moglo pomisliti za najopćenitiju klasu koja predstavlja bilo kakav objekt – samo prazna, apstraktna klasa bez svojih funkcionalnosti. Radi se o konkretnoj klasi koju je moguće normalno instancirati. Osim toga je i prilično kompleksna i njena implementacija je vezana uz čitav niz programskih tehnika. Ovdje će se od svega spominjati samo njene najosnovnije karakteristike.

4.2.1 Metoda `equals`

U odjeljku 3.1.2 spomenuto je da za provjeru jednakosti objekata treba koristiti metodu `equals`. Metodu se koristi tako da ju se pozove nad prvim objektom i kao argument se proslijedi drugi objekt: `o1.equals( o2 )`. Metoda `equals` definirana je u klasi `Object`, ali na trivijalan način – uspoređuju se reference (`o1 == o2`). Ako je potrebna smislena usporedba objekata neke klase, potrebno je nadjačati ovu metodu odgovarajućom implementacijom. Potpis metode je `equals( Object )` i upravo takvu metodu moramo dodati u svoju klasu. Budući da je parametar vrste `Object`, prvo će biti potrebno ustanoviti je li proslijeđeni objekt odgovarajuće vrste – jer ako nije, onda je jasno da nije jednak trenutnom objektu. Stoga kod ove metode rutinski započinje korištenjem operatora `instanceof`, iza čega slijedi odgovarajući *downcast*. Na primjer, mogli bismo imati ovakav `equals` u klasi `SimpleCounter` (ispis 2.24):

Ispis 4.1: Implementacija metode `equals`

```
package hr.fer.tel.ilj;

public final class SimpleCounter
extends AbstractCounter
{
  ... sve isto kao ranije ...

  public boolean equals( Object o )
  {
    if ( !( o instanceof SimpleCounter ) )
      return false;

    SimpleCounter that = (SimpleCounter) o;
```

```
    return this.counterState == that.counterState;
  }
}
```

Za metodu `equals` tipično je da se okorištava klasnom enkapsulacijom i da izravno pristupa atributima drugog objekta.

Jedna od tipičnih pogrešaka kod studenata je deklaracija metode s parametrom čija vrsta je trenutna klasa (npr. `equals( SimpleCounter )`). Parametar **mora biti vrste** `Object`, u protivnom se neće raditi o nadjačavanju, već o definiranju nove metode koja samo preopterećuje ime `equals`. U jednostavnim primjerima kod će čak ispravno raditi jer će se pozivati ta metoda umjesto prave metode `equals`. Sljedeći primjer pokazuje u kojem slučaju će raditi, a u kojima ne:

Ispis 4.2: Tipična pogreška s preopterećenjem umjesto nadjačavanjem

```
public class EqualsTester
{
  public boolean equals( EqualsTester et )
  {
    return true;
  }

  public static void main( String[] args )
  {
    EqualsTester et1 = new EqualsTester();
    EqualsTester et2 = new EqualsTester();
    boolean b;

    b = et1.equals( et2 );
    System.out.println( b ); // true

    // upcast argumenta na vrstu Object:
    Object o2 = et2;
    b = et1.equals( o2 );
    System.out.println( b ); // false!

    // upcast objekta nad kojim se poziva metoda:
    Object o1 = et1;
    b = o1.equals( et2 );
    System.out.println( b ); // false!
  }
}
```

Kad god se ispiše `false`, znači da je pozvana originalna metoda `equals` iz klase `Object`, koja je ostala nenadjačana. Razlog za ovakvo ponašanje, koje se nekima može činiti nelogičnim s obzirom na postojanje dinamičkog

povezivanja, je u tome što se dinamičko povezivanje provodi samo nad parametrom `this`, tj. nad objektom lijevo od točke u pozivu metode (vidjeti odjeljak 3.11). Vrste argumenata, tj. *potpis* pozvane metode utvrđuje se statički, prilikom prevođenja. U prvom pozivu, `et1.equals( et2 )` utvrđeni potpis metode je `equals( EqualsTester )` pa se poziva naša metoda. U drugom slučaju ispada `equals( Object )` pa se poziva metoda naslijeđena od klase `Object`.

Treći slučaj je najsuptilniji. Argument je ponovo vrste `EqualsTester` – kao u prvom slučaju, koji je radio ispravno. Međutim, sada je izraz lijevo od točke vrste `Object`. Prevoditelj u toj klasi traži sve metode imena `equals` i nalazi samo jednu, s parametrom vrste `Object`. Na osnovu toga odabire se potpis metode `equals( Object )`. Tek tijekom izvršavanja uočit će se da je stvarna vrsta objekta lijevo od točke `EqualsTester`, međutim tada je već fiksiran potpis metode koju treba pozvati, i to je `equals( Object )` – metoda naslijeđena iz klase `Object`.

Problem kao u drugom slučaju događa se uvijek kad se metoda preopterećuje tako da su vrste parametara u odnosu podklasa-nadklasa. Takvo preopterećenje **redovito treba izbjegavati**. Ako postoji potreba za takve dvije metode, trebaju imati različita imena. Za manifestaciju problema kao u trećem slučaju dodatno je bilo potrebno da metoda iz jedne klase bude preopterećena u njenoj podklasi.

4.2.2 Metoda `hashCode`

Ova metoda vezana je uz vrlo značajnu programsku tehniku – heš-tablicu (eng. *hash table* ili *hashtable*). To je struktura koja služi za memorijski efikasno spremanje veće količine podataka uz sposobnost brzog dohvaćanja podatka prema njegovom ključu – *heš-kodu*. U Javu ugrađene *kolekcije* (vidjeti 6. poglavlje) interno koriste heš-tablice (npr. često korištena klasa `HashSet`). One se oslanjaju na ispravnu implementaciju metode `hashCode` u svakom objektu koji se sprema u kolekciju. Ova metoda usko je povezana s metodom `equals` i njih dvije se uvijek nadjačava u paru.

Metoda `hashCode` zadužena je da za svaki objekt vrati njegov heš-kod u obliku cijelog broja (`int`-a). Presudno je da dva jednaka objekta uvijek prijavljuju i jednak heš-kod. Bitno je i da dva različita objekta što češće vraćaju različit heš-kod – dakle, da heš-kodovi budu što bolje raspršeni po objektima. U protivnom će se degradirati efikasnost heš-tablice. Jasno je

da objekt koji ima više od 2^{32} mogućih stanja ne može za svako stanje prijaviti različit heš-kod i stoga je potrebno promišljeno implementirati ovu metodu da se postigne dobro raspršenje. Postoji standardni predložak za implementaciju ove metode i najbolje ga se uvijek držati. U ispisu 4.3 prikazana je klasa koja sadrži sve moguće slučajeve vrsta podataka i način izračuna njihovog heš-koda. Ukratko, postupak je sljedeći:

1. Deklariramo cjelobrojnu varijablu (u primjeru: `hc`) i inicijaliziramo ju na 17.
2. Za svaki atribut koji sudjeluje u metodi `equals` izračunamo njegov heš-kod kao u primjeru.
3. U varijablu `hc` ugrađujemo pojedini heš-kod atributa prema formuli `hc = 37*hc +` *`heš-kod`*
4. Konačnu vrijednost varijable `hc` vratimo kao ukupan heš-kod objekta.

Ispis 4.3: Predložak za generiranje heš-koda

```
public class HashCodeExample
{
  private boolean boolVal;
  private byte byteVal;
  private char charVal;
  private short shortVal;
  private int intVal;
  private long longVal;
  private float floatVal;
  private double doubleVal;
  private Object objVal;
  private int[] intArray = { 1, 2, 3 };

  public int hashCode()
  {
    int hc = 17;
    hc = 37*hc + ( boolVal? 1 : 0 );
    hc = 37*hc + byteVal;
    hc = 37*hc + charVal;
    hc = 37*hc + shortVal;
    hc = 37*hc + intVal;
    hc = 37*hc + (int) ( longVal ^ ( longVal >>> 32 ) );
    hc = 37*hc + Float.floatToIntBits( floatVal );

    long dblToLong = Double.doubleToLongBits( doubleVal );
    hc = 37*hc + (int) ( dblToLong ^ ( dblToLong >>> 32 ) );
```

```
    hc = 37*hc + objVal.hashCode();

    for ( int i = 0; i < intArray.length; i++ )
      hc = 37*hc + intArray[i];

    return hc;
  }
}
```

4.2.3 Metoda `toString`

Pri razvoju programa, posebice tijekom traženja i uklanjanja pogrešaka, program mora ispisivati detalje tijeka izvršavanja. Na primjer, često je potreban ispis trenutnog stanja ključnih objekata prije i nakon neke operacije. Također, u trenutku kad dođe do pogreške iznimno je korisno dobiti izvještaj o stanju objekta koji je izazvao pogrešku. Za ovakve i druge namjene postoji metoda `toString` koju imaju svi objekti jer je definirana u klasi `Object`. Metoda je zadužena da vrati znakovni niz koji opisuje objekt.

Izvorna implementacija u klasi `Object` prilično je beskorisna (vraća ime klase čija je instanca u pitanju i njen heš-kod u heksadecimalnom zapisu), stoga ju je redovito potrebno nadjačati smislenom implementacijom. Studenti ponekad krivo shvate namjenu metode pa ju izvedu tako da sama ispisuje na zaslon. Pazite da izbjegnete takvu pogrešku. Primjer implementacije u klasi `SimpleCounter` (ispis 2.24):

Ispis 4.4: Implementacija metode `toString`

```
package hr.fer.tel.ilj;

public final class SimpleCounter
extends AbstractCounter
{
  ... sve isto kao ranije ...

  public String toString()
  {
    return "SimpleCounter:" + this.counterState;
  }
}
```

4.2.4 Metoda `clone`

Već je u prvom poglavlju (odjeljak 2.16) objašnjeno da je kopiranje objekata česta potreba u Javi. Tamo je objašnjeno i kako je najbolje izvesti kopiranje,

a metoda *clone* je samo spomenuta uz napomenu da ju je bolje izbjegavati. Ovdje se nju ipak opisuje prvenstveno zato da se demonstrira zašto ju treba izbjegavati.

Metoda *clone* ima dosta neobično ponašanje i način izvedbe. Slijede pravila kojih se **mora bezuvjetno pridržavati**, iako ih kompilator ne nameće:

1. Klasa mora implementirati sučelje *java.lang.Cloneable*. U protivnom će se baciti iznimka *CloneNotSupportedException*

2. Metoda u klasi *Object* je *protected*, ali sve nadjačavajuće metode moraju biti *public*.

3. Nadjačavajuća metoda **ne smije koristiti konstruktor** za kreiranje kopije objekta (osim ako je dotična klasa *final*). Mora dobiti kopiju od *Object*-ove metode *clone*, pozivom `super.clone()`. Dobivena kopija je tzv. *plitki klon* jer ima prepisane sve vrijednosti atributa. Plitka je jer to znači da su prepisane i reference na već postojeće objekte, tako da novi objekt koristi iste unutarnje objekte kao i stari. Da bismo dobili duboko kloniranje (skoro uvijek nam treba baš to), moramo eksplicitno napraviti kopije unutarnjih objekata.

Razlozi postojanja ovih pravila su složene naravi; koga zanimaju detalji, neka čita knjigu Effective Java **??**. Poglavlje o ovoj temi dostupno je i na Webu.

Primjer pravilne izvedbe metode *clone*:

```
public class DifferentialCounter
{
  private Counter cntr1 = new SimpleCounter();
  private Counter cntr2 = new SimpleCounter();

  public Object clone()
  throws CloneNotSupportedException
  {
    DifferentialCounter copy = (DifferentialCounter) super.clone();

    // kreiranje vlastitih podobjekata:
    copy.cntr1 = (Counter) this.cntr1.clone();
    copy.cntr2 = (Counter) this.cntr2.clone();

    return copy;
  }
```

```
}

Da ponovimo još jednom: \textbf{izbjegavajte korištenje metode
\emph{clone}!} Koristite mehanizam opisan u odjeljku
\ref{kopiranje-objekata}.
```

4.3 Klasa `java.lang.String`

U Javi, znakovni nizovi predstavljeni su instancama ugrađene klase `String`. Ova klasa integrirana je u sam jezik i postoje jezične konstrukcije posebno za nju:

- Doslovno navođenje niza unutar dvostrukih navodnika, npr. `"znakovni niz"`. Rezultat ovog izraza je instanca klase `String` s navedenim tekstom kao sadržajem. Znakovne nizove, dakle, **ne kreira se uobičajenim konstruktorom**: `new String("znakovni niz")`. Ovo doduše radi, ali se sasvim nepotrebno stvara objekt viška – sam niz pod navodnicima je jedan objekt, a dodatni se stvara operatorom `new`.

- Operator ulančavanja, npr. `"znakovni" + " " + "niz"`. Rezultat je novi `String` sadržaja `"znakovni niz"`. U skladu s ovime definiran je i složeni operator dodjele `+=`. Znak `+` koristi se dakle i za zbrajanje i za ulančavanje. Odluka o tome koji je smisao danog znaka donosi se prema vrsti lijevog operanda: ako je to `String`, operacija je ulančavanje. U složenijim izrazima ne mora biti sasvim očito što je lijevi operand pa treba biti na oprezu (npr. što je lijevi operand drugog plusa u izrazu `"a" + 2 + 3`? —to je izraz `"a" + 2`, vrste `String`). Ako se kao operand ulančavanja pojavi nešto drugo osim `String`-a, automatski se pretvara u odgovarajući `String`. Ako se radi o objektu, poziva se njegova metoda `toString`, ako se radi o nul-referenci, pretvara se u niz `"null"`, a i primitivni podaci se pretvaraju na smisleni način. Primjerice, rezultat izraza `"a" + 2 + 3` je `"a23"`. S druge strane, "rezultat" izraza `2 + 3 + "a"` je prevoditeljeva pogreška jer se u ovom slučaju radi o zbrajanju, a nizove se ne može pribrajati brojevima. Iz ovakve situacije možemo se izvući malim trikom: `"" + 2 + 3 + "a"` će dati željeni rezultat `"23a"`.

4.3.1 Provjera jednakosti znakovnih nizova

Uspoređivanje znakovnih nizova *nije* na sličan način elegantno integrirano u jezik i mora ih se uspoređivati kao i ostale objekte, pozivom metode `equals`. Usporedba pomoću `==` samo će usporediti reference, kao i inače. Korištenje pogrešnog načina usporedbe može izazvati vrlo neobično ponašanje – ponekad ispravno, a ponekad neispravno. To je stoga što Java automatski optimizira korištenje instanci `String`-a pa može višestruko iskoristiti istu instancu na više mjesta gdje se pojavljuje isti niz. Ako se to dogodi, provjera pomoću `==` će proći:

Ispis 4.5: Posljedice pogrešnog korištenja `==` za usporedbu `String`-ova

```
public static void main( String[] args )
{
  String a = "a";
  String b = "b";
  String ab1 = "ab";
  String ab2 = "ab";
  String ab3 = a + b;

  System.out.println( ab1 == ab2 ); // true
  System.out.println( ab1 == ab3 ); // false

  System.out.println( ab1.equals( ab2 ) ); // true
  System.out.println( ab1.equals( ab3 ) ); // true
}
```

U primjeru je konstanta `"ab"` uspješno iskorištena dvaput – i za `ab1` i za `ab2`. Međutim, kad taj isti niz sastavimo ulančavanjem iz druga dva, kreira se novi objekt pa referenca na njega više nije ista.

4.3.2 Nepromjenjivost instanci `String`-a

Klasa `String` dizajnirana je da bude *nepromjenjiva* (eng. *immutable*) – njene instance nikad ne mijenjaju stanje. Ovo svojstvo uvijek je izrazito poželjno za svaku klasu gdje je to moguće. Takvi objekti imaju neka korisna svojstva koja inače imaju podaci primitivne vrste:

- Ako se takav objekt proslijedi metodi, nema opasnosti da će ga ona izmijeniti i tako neizravno i nepredviđeno utjecati na funkciju pozivajuće metode.

- Ako je neki od atributa objekta nepromjenjiv objekt, može ga se dati na raspolaganje vanjskom svijetu bez opasnosti da ga se izvana izmijeni i tako neizravno utječe na stanje objekta koji ga sadrži.
- U višenitnom programu nema potrebe za sinkronizacijom pristupa nepromjenjivom objektu iz više niti.

Budući da su `String`-ovi nepromjenjivi, jasno je da se pri svakoj promjeni vrijednosti neke `String`-ovske varijable stvara novi objekt. Npr. izraz `s += "tri"` neće izmijeniti postojeći objekt pridijeljen varijabli `s`, već će stvoriti novi i referencu na njega zapisati u varijablu. Ako u kodu imamo situaciju gdje se događaju brojne izmjene znakovnih nizova, možemo radi štednje radne memorije pribjeći korištenju druge klase, `StringBuffer`, koja modelira promjenjivi niz. U tom slučaju, međutim, nećemo se moći koristiti elegantnim jezičnim konstrukcijama kao kad radimo sa `String`-om.

4.4 Klasa `System`

Ova klasa ne služi za instanciranje, već sadrži korisne statičke metode i atribute koji ostvaruju vezu s uslugama računalske platforme na kojoj se izvršava Javin program. Pregled najvažnijih članova klase:

- atributi `in`, `out` i `err`: veza na standardni ulaz, izlaz i izlaz za pogreške. Vidjeti 8.3
- metoda `currentTimeMillis`: trenutno sistemsko vrijeme
- metoda `exit( int )`: prekid izvršavanja programa. Proslijeđeni broj je statusni kod s kojim program završava. Kod `0` znači da program završava na redovan način; ostali kodovi signaliziraju neku vrstu pogreške. Uobičajeno je npr. koristiti kod `-1` ako se prekida zbog pogreške.
- metoda `getenv`: dohvaćanje sistemskih varijabli (*environment variables*)
- metoda `arrayCopy`: efikasno kopiranje sadržaja iz jednog polja u drugo

Detalje o ovim i drugim članovima klase treba potražiti u dokumentaciji [1].

4.5 Klase-omotači primitivnih vrsta, *autoboxing*

Java razlikuje primitivne od referentnih vrsta podataka i stoga brojevi, logičke vrijednosti i znakovi nisu objekti. Međutim, mnoga programska rješenja, kao npr. kolekcije podataka (vidjeti poglavlje 6), rade isključivo s objektima. Da bi se ipak u takvim slučajevima moglo baratati i njima, u Javinu biblioteku klasa ugrađene su posebne klase čije instance predstavljaju podatke primitivnih vrsta. Radi se o sljedećim klasama u paketu `java.lang`:

1. `Boolean`
2. `Byte`
3. `Character`
4. `Short`
5. `Integer`
6. `Long`
7. `Float`
8. `Double`

Jedna instanca predstavlja jednu vrijednost. Instance su *nepromjenjive*, što dodatno olakšava njihovo tretiranje kao da su podaci primitivne vrste (vidjeti odjeljak 4.3.2). Na primjer, ako negdje moramo koristiti objekte, a podatak kojim trebamo baratati je vrste `int`, možemo koristiti omotač `Integer` na sljedeći način:

```
int i = 1;
Integer intObj = i; // omotavanje u objekt
... radimo nešto s omotanom vrijednosti ...
int j = intObj;  // razmotavanje natrag u primitivnu vrijednost

Integer intObj2 = i + 1;
int k = intObj + intObj2;
```

Java dozvoljava implicitnu konverziju između primitivnih vrijednosti i njihovih omotača u izrazu dodjele vrijednosti. To jezično svojstvo naziva se *autoboxing/unboxing*. Dozvoljeno je i izvoditi računske operacije nad omotačima, kao što se vidi u posljednjem retku primjera.

4.6 Konverzija između podatka i njegovog znakovnog zapisa

Klase omotača primitivnih vrsta sadrže i statičke metode koje parsiraju znakovne zapise vrijednosti. Primjeri:

```
int i = Integer.parseInt( "-2" );
double d = Double.parseDouble( "2.021e-2" );
boolean b = Boolean.parseBoolean( "true" );
```

Očekivani ispis:

```
i = -2, d = 0.02021, b = true
```

Konverzija u suprotnom smjeru, iz primitivne vrijednosti u znakovni niz, može se obaviti na dva načina. Jedna je memorijski ekonomičnija, ali druga ima elegantniji zapis:

```
double d = 2.02;
String s1 = String.valueOf( d );
String s2 = "" + d;
```

U drugom retku koristi se statička metoda `String.valueOf`. Metoda je preopterećena i ima verziju za svaku primitivnu vrstu podataka. U trećem retku koristi se već prikazano svojstvo operacije ulančavanja (u odjeljku 4.3) da automatski pretvara sve primitivne podatke u nizove.

Poglavlje 5

Iznimke

U praktički svakom programu dobar dio koda posvećen je provjeri ispravnosti neke situacije i prijavljivanju pogrešaka ako se uoči nepravilnost. Na primjer, tipično je provjeriti je li proslijeđeni cijeli broj u potrebnom opsegu. U našem slučaju brojača po modulu (ispis 3.13; klasa `AbstractCounter` je u ispisu 2.23, a sučelje `Counter` u ispisu 2.15) bit će potrebno npr. provjeriti da je ciklus brojanja proslijeđen metodi `setModulus` pozitivan broj:

Ispis 5.1: `ModuloCounter` s provjerom parametra

```
package hr.fer.tel.ilj;

public class ModuloCounter
extends AbstractCounter
{
  private int modulus;

  public ModuloCounter( int initialState )
  {
    super();
    this.modulus = 10;
    setState( initialState );
  }

  public ModuloCounter()
  {
  }

  public void setModulus( int newModulus )
  {
    if ( newModulus < 0 )
      System.out.println( "Negativan ciklus brojanja!" );
    else
      this.modulus = newModulus;
  }
```

```
  void setState( int newState )
  {
    this.counterState = newState % this.modulus;
  }
}
```

Na prvi pogled ovo može izgledati kao savršeno rješenje – ako je broj krivi, ispisuje se pogreška, inače se obavlja zatražena akcija. Međutim, u stvarnosti ovo je prilično neupotrebljivo. Pozivajuća metoda uopće neće imati informaciju o tome je li naredba za postavljanje ciklusa uspješno izvršena. Program će nastaviti dalje i, naravno, raditi neispravno. Programer, gledajući ispis, moći će samo ustanoviti da je u jednom trenutku metoda `setModulus` pozvana s pogrešnom vrijednošću. Tijekom izvršavanja ta metoda se pozvala tko zna koliko puta i s kojih sve lokacija.

Malo bolji pristup, koji se npr. koristi u jeziku C, je da metoda ne bude `void`, nego `int` i da broj koji vraća ovisi o uspješnosti izvršenja. Svaka pozivajuća metoda tada može provjeriti je li sve dobro završilo. Ovaj način opet ima niz mana – npr. to znači da za svaki pojedini poziv bilo koje metode koja bi se mogla neuspješno izvršiti treba imati po jedan blok `if` za provjeru. Dodatni je problem što se sad povratna vrijednost ne može koristiti za regularne namjene – mnoge metode nisu `void` pa da ih možemo lako preurediti da vraćaju `int`.

5.1 Bacanje iznimke – prijava iznimne situacije

U jeziku Java za rješavanje ovih problema postoji poseban mehanizam – *iznimke*. Gornja metoda `setModulus` u Javi bi se izvela ovako:

```
public class ModuloCounter
{
  ...

  public void setModulus( int newModulus )
  {
    if ( newModulus < 0 )
      throw new IllegalArgumentException( "Negativan ciklus brojanja" );

    this.modulus = newModulus;
  }
```

```
  ...
}
```

Primjećujemo ključnu riječ `throw`: njome se *baca* iznimku. Ovo je sasvim nov i neobičan pojam i trebat će malo više vremena da "legne" početniku. Bacanjem iznimke prekida se izvršavanje metode i ona, ako je i trebala vratiti neku povratnu vrijednost, u ovom slučaju neće ju vratiti, nego će se u pozivajućoj metodi "pojaviti" ta bačena iznimka kao rezultat metode. Ako tamo nema nikakvog koda specifičnog za baratanje iznimkama, dogodit će se nešto prilično neočekivano: izvršavanje te pozivajuće metode *također* će odmah biti prekinuto i ona će ponovo baciti tu istu iznimku svojoj pozivajućoj metodi. Ako dakle imamo ovakav slučaj:

Ispis 5.2: Bacanje i ponovno bacanje iznimke

```
public class ModuloCounter
{
  ...

  public void setModulus( int newModulus )
  {
    if ( newModulus < 0 )
      throw new IllegalArgumentException( "Negativan ciklus brojanja" );

    this.modulus = newModulus;
  }

  ...
}

public class AppThatUsesTheCounter
{
  private static boolean intermediateMethod( int i )
  {
    ModuloCounter mc = new ModuloCounter();
    mc.setModulus( i );
    System.out.println( "intermediateMethod uspješno završava" );
    return true;
  }

  public static void main( String[] args )
  {
    boolean b = intermediateMethod( -1 );
    System.out.println( "intermediateMethod je vratila " + b );
  }
}
```

pokretanje programa rezultirat će sljedećim ispisom:

Ispis 5.3: Izvještaj o bačenoj iznimci

```
Exception in thread "main" java.lang.IllegalArgumentException: Negativan ciklus brojanja
  at hr.fer.tel.ilj.AppThatUsesTheCounter.setModulus(AppThatUsesTheCounter.java:8)
  at hr.fer.tel.ilj.AppThatUsesTheCounter.intermediateMethod(AppThatUsesTheCounter.java:13)
  at hr.fer.tel.ilj.AppThatUsesTheCounter.main(AppThatUsesTheCounter.java:18)
```

Primijetimo da se poruke "intermediateMethod je vratila..." i "intermediateMethod uspješno završava" *nisu* ispisale. U `intermediateMethod` iznimka se pojavila u prvom retku i, u nedostatku koda koji ju obrađuje, tu je metoda završila ponovnim bacanjem iste iznimke (eng. *rethrow*). Iznimka je tako stigla do metode `main`, također u njenom prvom retku. Budući da od nje nema kamo dalje (to je metoda od koje je program započeo), čitav program je prekinut uz prethodni ispis detaljnog izvještaja o tome gdje je točno došlo do iznimke (cjelokupan redoslijed pozivanja metoda i točno mjesto u kodu gdje se svaki poziv događa), koja je vrsta iznimke (`IllegalArgumentException`) i njen pobliži opis ("Negativan ciklus brojanja"). Sve se to dogodilo automatski, bez ijednog retka koda vezanog uz obradu iznimke.

Varijabli `b` u metodi `main` očito nije mogla biti dodijeljena nijedna vrijednost jer `intermediateMethod` nije završila vraćanjem povratne vrijednosti. Iz toga je već jasno da se izvršavanje ne može nastaviti normalnim putem jer je varijabla ostala neinicijalizirana, a već u sljedećem retku se njena vrijednost treba ispisati.

5.2 Hvatanje iznimke – obrada iznimne situacije

Prikazani način korištenja iznimke još uvijek nije previše svrsishodan jer će uslijed svakog bacanja iznimke program "pucati". U većini realnih situacija trebat će se na odgovarajući način pobrinuti za iznimnu situaciju i zatim nastaviti s izvršavanjem programa. Ključna je prednost što možemo odabrati u kojoj metodi ćemo napraviti provjeru – to ne treba biti niti metoda gdje je došlo do pogreške, niti metoda koja ju je izravno pozvala. Možemo npr. sve iznimke obrađivati u metodi `main`:

Ispis 5.4: Hvatanje iznimke

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

```
    try
    {
      b = intermediateMethod( -1 );
      System.out.println( "intermediateMethod je vratila " + b );
    }
    catch ( IllegalArgumentException e )
    {
      System.err.println( "Iznimka!");
      e.printStackTrace();
    }
    b = intermediateMethod( 1 );
    System.out.println(
      "Drugi pokušaj: intermediateMethod je vratila " + b );
  }
```

Tu se pojavljuje nova konstrukcija, koja koristi i dvije nove ključne riječi: `try` i `catch`. Pomoću `try` najavljuje se blok koda koji će se *pokušati* izvršiti, ali u kojem bi moglo doći i do iznimke.

U ovom trenutku potrebno je napomenuti par detalja u vezi s karakterom iznimke. U Javi to je jednostavno objekt kao i svaki drugi – instanca klase `java.lang.Exception` ili bilo koje njene podklase. Iznimku se kreira na uobičajen način, operatorom `new`. Da je iznimka normalan objekt najjasnije se vidi po izjavi

```
e.printStackTrace();
```

Tu smo pozvali metodu nad objektom iznimke. Ovo je vrlo korisna metoda jer ona ispisuje izvještaj o iznimci, kao što je pokazano u ispisu 5.3. Ranije je prikazan primjer u kojem se izvještaj ispisuje automatski, prije prekida programa. Ovom metodom možemo eksplicitno zatražiti takav izvještaj, a bez prekidanja programa. Izvještaj se šalje na standardni izlaz za pogreške (`System.err`).

Iznimku se *hvata* ključnom riječju `catch`. Čitava konstrukcija je `catch ( VrstaIznimke imeParametra )` i vrlo je srodna konstrukciji za deklaraciju parametra metode. Sama iznimka koja će biti uhvaćena može biti instanca navedene klase ili bilo koje podklase – isto vrijedi i za parametar metode. Nakon ove konstrukcije slijedi blok koda u kojem se `imeParametra` može koristiti kao i svaka druga varijabla. U gornjem primjeru nad njom se poziva metoda `getMessage`, kojom raspolaže svaka iznimka i vraća poruku koja joj je dodijeljena prilikom bacanja.

Ako je iznimka uspješno uhvaćena u odgovarajućem bloku `catch`, neće se dalje bacati. Nakon izvršenja ovog bloka program dalje nastavlja normalno, izvršavanjem koda koji slijedi nakon čitave konstrukcije `try...catch`. Osta-

tak koda nakon pojave iznimke pa do kraja bloka `try` neće se izvršiti – a to je i bio cilj. U protivnom bismo se ponovo susreli s problemom što učiniti npr. s neinicijaliziranom varijablom `b`.

S druge strane, ako bačena iznimka nije one vrste koja se hvata `catch`-om, ponašanje će biti kao da tog bloka i nema – iznimka će se proslijediti dalje, pozivajućoj metodi.

5.3 Završne radnje prije bacanja iznimke – blok `finally`

Blok `try` možemo popratiti i posebnim blokom koda koji će se sigurno izvršiti, bez obzira je li došlo do iznimke. To je prvenstveno korisno za slučaj kad dođe do iznimke jer u tom bloku imamo šansu obaviti neku završnu radnju – npr. tipičan slučaj je zatvaranje datoteka, internetskih konekcija i sl. – prije nego što se metoda prekine bacanjem iznimke. Za to služi ključna riječ `finally`:

Ispis 5.5: Blok `finally`

```
public class ModuloCounter
{
  ...

  private void setModulus( int newModulus )
  {
    if ( newModulus < 0 )
      throw new IllegalArgumentException( "Negativan ciklus brojanja" );
  }

  ...
}

public class AppThatUsesTheCounter
{
  private static boolean intermediateMethod( int i )
  {
    try
    {
      ModuloCounter mc = new ModuloCounter();
      mc.setModulus( i );
    }
    finally
    {
      System.out.println( "intermediateMethod završava" );
    }
```

```
    return true;
  }

  public static void main( String[] args )
  {
    boolean b;
    try
    {
      b = intermediateMethod( -1 );
      System.out.println( "intermediateMethod je vratila " + b );
    }
    catch ( IllegalArgumentException e )
    {
      System.out.println( "Iznimka: " + e.getMessage() );
    }
    b = intermediateMethod( 1 );
    System.out.println(
      "Drugi pokušaj: intermediateMethod je vratila " + b );
  }
}
```

Očekivan ispis za ovaj, već dosta složen slučaj je:

```
intermediateMethod završava
Iznimka: Negativan ciklus brojanja
intermediateMethod završava
Drugi pokušaj: intermediateMethod je vratila true
```

Vidimo kako se sada tekst "intermediateMethod završava" ispisuje u oba slučaja – i kad je došlo do iznimke, i kad nije. U gornjem primjeru blok `finally` korišten je samostalno, kao i blok `catch` u metodi `main`, ali općenito oni se mogu i kombinirati. Jedino treba poštivati pravilo da `finally` dođe na kraj, iza `catch`.

U primjeru je korištena još jedna metoda definirana za sve iznimke – `getMessage`. Ona vraća znakovni niz koji je proslijeđen konstruktoru prilikom bacanja iznimke.

5.4 Višestruki blokovi `catch`

Još jedna prednost mehanizma iznimki je u tome što možemo imati podulji blok koda u kojem se mogu pojavljivati raznorazne iznimke bez potrebe da se svaka obrađuje tamo gdje se pojavila. Da bi ta prednost bila u potpunosti iskoristiva, moguće je nakon bloka `try` imati ne jedan, nego proizvoljno mnogo blokova `catch`, svaki za svoju vrstu iznimke. Od svih njih izvršit će

najviše jedan, i to prvi čija vrsta parametra odgovara vrsti iznimke. Realističan primjer u kojem je potrebno hvatati razne iznimke bio bi prilično kompleksan pa ćemo ovdje dati samo shematski primjer:

Ispis 5.6: Višestruki blokovi `catch`

```
public int someMethod()
{
  try
  {
    method1();
    method2();
    method3();
    return 0;
  }
  catch ( IllegalArgumentException e )
  {
    System.err.println( "Pogrešan argument" );
  }
  catch ( NullPointerException e )
  {
    System.err.println( "Nul-referenca" );
  }
  catch ( FileNotFoundException e )
  {
    System.err.println( "Datoteka ne postoji" );
  }
  catch ( Exception e )
  {
    System.err.println( "Nepredviđena iznimka" );
  }
  finally
  {
    System.err.println( "Blok try završava" );
  }
  return -1;
}
```

Uočimo da se u posljednjem bloku `catch` hvata vrsta `Exception`. Taj blok je tu da uhvati bilo koju iznimku koju nije uhvatio nijedan prethodni blok. Važno je, naravno, da taj blok bude posljednji, u protivnom će uvijek on hvatati iznimku.

Potpuni redoslijed izvršavanja je sljedeći: prvo se normalno izvršava kod bloka `try`. Ako ne dođe do iznimke, u posljednjem retku bloka `try` metoda treba uredno završiti s povratnom vrijednosti `0`. Međutim, prije toga izvršava se blok `finally`. Ako dođe do iznimke, izvršava se odgovarajući blok `catch` i nakon njega `finally`. U tom slučaju nakon izvršenja `finally` izvršavanje nastavlja redovitim putem i nailazi se na `return -1` pa metoda

vraća vrijednost `-1`. Ako se pojavi iznimka koju se ne hvata, izvršava se blok `finally` i nakon toga metoda se prekida ponovnim bacanjem iste iznimke. Vidimo dakle da se blok `finally` izvršava *apsolutno uvijek*, bez obzira završava li blok `try` na redovan ili izniman način i bez obzira je li iznimka uhvaćena ili nije.

5.5 Najava iznimke; provjeravana i neprovjeravana iznimka

Iznimke se u Javi dijele na dvije podvrste: *provjeravane* (eng. *checked*) i *neprovjeravane* (eng. *unchecked*). Sve neprovjeravane iznimke izvedene su iz `RuntimeException`; ostale su provjeravane. Sama `RuntimeException` je izravna podklasa spomenute klase `Exception`.

Ako se može dogoditi da metoda baci provjeravanu iznimku, tada ju je potrebno *najaviti* ključnom riječju `throws`:

Ispis 5.7: Najava iznimke

```
import java.io.*;

...

  public static void fileMethod()
  throws FileNotFoundException
  {
    File f = new File( "/hr/fer/tel/file1.txt" );
    InputStream fis = new FileInputStream( f );
    ...
  }
```

U primjeru se koriste Javine ugrađene klase za baratanje datotekama (iz paketa `java.io`). U konstruktoru klase `FileInputStream` najavljeno je da bi mogao baciti provjeravanu iznimku `FileNotFoundException` (ako zatražena datoteka ne postoji). Budući da tu iznimka ne hvata, naša metoda također mora najaviti da bi mogla baciti ovu iznimku, u protivnom će prevoditelj prijaviti pogrešku da metoda može baciti dotičnu iznimku, ali to ne najavljuje.

Iz navedenoga je jasno značenje pridjeva "provjeravana": za takvu iznimku kompilator *provjerava* može li doći do njenog bacanja i inzistira da to bude najavljeno. Naš primjer koda ne sadrži eksplicitnu izjavu `throw new`

`FileNotFoundException()`, ali kompilatoru je dovoljna činjenica da se koristi konstruktor koji tu iznimku najavljuje. Općenito, spomenuti konstruktor može biti izveden i tako da nikad ne baci iznimku, ali kompilatoru je dovoljna sama najava. Isto tako, svaka metoda koja poziva našu metodu `fileMethod` morat će se pobrinuti za ovu iznimku – čak i ako izbacimo redak u kojem se koristi sporni konstruktor.

Najavljena iznimka korisna je za slučajeve kad do nje može doći tijekom normalnog, ispravnog izvršavanja programa. Na primjer, u našem slučaju programer ne može unaprijed znati hoće li potrebna datoteka postojati svaki put kad se program izvršava. To općenito i ne mora značiti da je došlo do pogreške – radi se jednostavno o izvanrednoj situaciji koju se zahvaljujući mehanizmu iznimaka može obrađivati na mjestu gdje je to najpraktičnije. Korištenjem provjeravane iznimke postići ćemo to da programer ne može zabunom zaboraviti zapisati kod koji obrađuje tu uvijek prisutnu mogućnost.

S druge strane, *neprovjeravana* iznimka koristi se za slučajeve u kojima se prijavljuje *neispravnost* – situacija do koje u ispravnom programu jednostavno ne smije doći. Takav je naš raniji primjer s metodom `setModulus` – ona je dizajnirana da prima pozitivan broj i nijedan smislen poziv ne može biti s negativnim brojem. Dakle, mora da se radi o programerskoj pogrešci. Mogućnosti za ovakve pogreške vrebaju na svakom koraku i kod bi bio neprihvatljivo nepregledan kad bi se inzistiralo na najavi ili obradi svake moguće takve iznimke.

U praksi uvijek postoje i rubni slučajevi, gdje je nejasno koju vrstu iznimke koristiti. Java dozvoljava da se i neprovjeravane iznimke najavljuju – kao pomoć programeru koji koristi takvu metodu, da bude spreman na mogućnost pojave dotične iznimke, iako nije dužan išta poduzeti. Najava neprovjeravane iznimke nema nikakvog utjecaja na kompilator – iznimka i dalje ostaje neprovjeravana.

5.5.1 Najava više iznimki

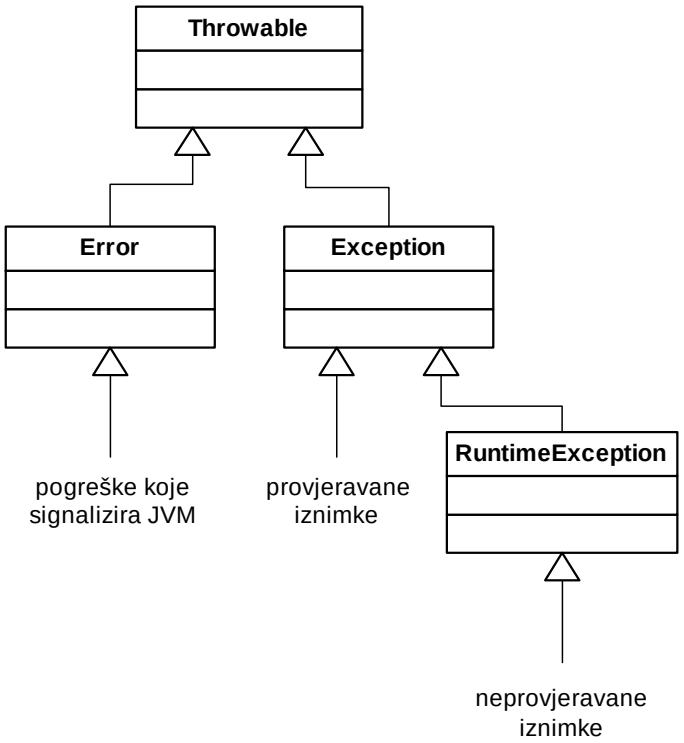
Metoda može najaviti proizvoljan broj iznimki. Navodi ih se redom, odvojene zarezom, npr.

```
public void throwingMethod()
throws FileNotFoundException, ClassNotFoundException, NamingException
{
  ...
```

```
}
```

5.6 Klasna hijerarhija iznimki

Dosad je spomenuto nekoliko detalja o hijerarhiji klasa iznimki. Spomenuta je klasa `Exception` kao glavna klasa iz koje su izvedene druge iznimke. Međutim, potpuna slika uključuje još nekoliko klasa. Prije svega tu je klasa `Throwable`, iz koje su izvedene točno dvije klase: `Error` i `Exception`. Dakle, `Throwable` je u korijenu čitave klasne hijerarhije iznimki. Klasa `Error` i njene podklase rezervirane su za uporabu od strane Javinog virtualnog stroja i nisu namijenjene da ih se eksplicitno baca izjavom `throw`. Cjelokupna hijerarhija klasa čije instance se mogu baciti prikazana je na slici 5.1. Dodatne klase iznimki koje uvode razvijatelji neizostavno moraju biti podklase od `Exception`.



Slika 5.1: UML-ov dijagram klasa čije instance se mogu baciti

5.7 Korištenje ugrađenih i vlastitih iznimki

U Javinoj biblioteci postoji već mnoštvo iznimki, provjeravanih i neprovjeravanih. Nekoliko njih, neprovjeravanih, ima općenit karakter i u kodu ih treba koristiti gdje god značenje odgovara karakteru pogreške:

- `NullPointerException` baca se kad god se problem sastoji od toga da je neka referenca jednaka `null` umjesto da pokazuje na neki objekt.
- `IllegalArgumentException` baca se kad se ustanovi da je metoda pozvana s pogrešnom vrijednosti nekog argumenta – vrijednosti koja je protivna pravilima pozivanja dotične metode. U ovom poglavlju imali smo primjer pozivanja `setModulus` s negativnim brojem.
- `IllegalStateException` baca se kad se neku metodu pozove dok je objekt u takvom stanju koje ne dopušta njezino pozivanje. Recimo da imamo klasu koja predstavlja brojač koji odbroji do neke konačne vrijednosti i nakon toga ne može dalje, već se mora poništiti. Ako u trenutku kad je brojač već došao do gornje granice pozovemo `increment`, to bi bila situacija za bacanje ove iznimke.
- `ClassCastException` baca se ako je metoda pozvana s instancom krive klase. To se događa tipično kad se radi o metodi koja je dio nekog sučelja pa formalno prima parametar općenitije vrste od one s kojom stvarno može raditi.
- `UnsupportedOperationException` baca se ako je metoda pozvana nad objektom koji ju ne implementira. To se tipično događa kad objekt najavljuje da implementira neko sučelje, ali neka od metoda iz sučelja nije primjerena za takav objekt pa ju nema smisla implementirati.

Za situacije specifične za softverski proizvod koji razvijamo treba definirati vlastitu klasu iznimke. Pravilo je da za čitav proizvod postoji samo jedna ili eventualno, za veće projekte, nekoliko iznimki. Ako ih je više, preporuča se da sve budu izvedene iz jedne, općenite. Definiranje vlastite iznimke vrlo je jednostavno – dovoljno je kreirati klasu izvedenu iz `Exception` za provjeravane odnosno `RuntimeException` za neprovjeravane iznimke. U klasi treba samo definirati konstruktore koji primaju iste parametre kao već postojeći konstruktori u nadklasi i samo prosljeđuju poziv na njih. Primjer:

Ispis 5.8: Definiranje vlastite iznimke

```
public class CounterException
extends RuntimeException
{
  public CounterException()
  {
    super();
  }

  public CounterException( String message )
  {
    super( message );
  }

  public CounterException( String message, Throwable cause )
  {
    super( message, cause );
  }

  public CounterException( Throwable cause )
  {
    super( cause );
  }
}
```

Ovime je definirana neprovjeravana iznimka `CounterException`. Definirana su četiri standardna konstruktora i sve što rade je da pozivaju isti takav konstruktor u nadklasi.

Poglavlje 6

Kolekcije objekata

Jedna vrlo značajna stvar u svakom programskom jeziku su *kolekcije* podataka. Kolekcija je jednostavno skupina objekata koju se može tretirati kao cjelinu. Osnovni primjeri su *lista*, *skup* i *rep*. Struktura srodna kolekciji je i *mapa* (skup parova ključ-vrijednost).

Jezik Java pruža dobru podršku za kolekcije objekata. Kolekcije primitivnih podataka podržane su neizravno – putem posebnih objekata-*omotača* primitivnih podataka. U biblioteci klasa, u paketu `java.util`, nalazi se poveći broj sučelja i klasa koje sačinjavaju sustav za manipulaciju kolekcija, tzv. *Collections Framework*.

6.1 Sučelje `Collection`

Sučelje `Collection` jedno je od temeljnih sučelja u *Collections Framework*-u. Apstrahira mogućnosti zajedničke za sve kolekcije – liste, skupove i repove. To sučelje je definirano ponajprije iz razloga koji je iznesen u odjeljku 2.10 – da putem varijable (parametra) ove vrste možemo baratati bilo kojom kolekcijom u opsegu funkcionalnosti koji je zajednički za sve njih. Uz kolekcije vežu se i dosad neprikazana jezična svojstva Jave – tzv. *generics*. Pogledajmo metodu u kojoj se demonstrira nekoliko osnovnih tehnika baratanja kolekcijom:

Ispis 6.1: Osnovno baratanje kolekcijom

```
static void manipulateCollection( Collection<String> col )
{
  String msg = "Poruka";
  boolean b = col.add( msg );
```

```
  if ( b )
    System.out.println( "Poruka dodana u kolekciju" );

  System.out.println( "Sadrži li kolekcija poruku? " +
      col.contains( msg ) );

  b = col.add( msg );
  if ( b )
    System.out.println( "Poruka dodana u kolekciju drugi put" );

  System.out.println( "Broj elemenata u kolekciji: " + col.size() );

  b = col.remove( msg );
  if ( b )
    System.out.println( "Poruka uklonjena iz kolekcije" );

  System.out.println( "Sadrži li kolekcija poruku? " +
      col.contains( msg ) );

  System.out.println( "Je li kolekcija prazna? " + col.isEmpty() );
}
```

Sintaksu vezanu uz *generics* vidimo u prvom retku, u deklaraciji parametra metode: `Collection<String> col`. Time naznačujemo da kolekcija sadrži instance klase `String`. Vrsta `Collection<String>` je tzv. *parametrizirana vrsta* – parametar je vrsta objekata koji se u njoj nalaze. Kôd demonstrira korištenje metoda sučelja `Collection`: `add`, `contains`, `isEmpty`, `remove`, `size`. Ispis će se razlikovati ovisno o tome koja konkretna implementacija kolekcije se proslijedi metodi. Razlika će prije svega biti između lista i skupova.

6.2 Sučelja `List` i `Set`

Napravimo ovakvu metodu `main` da ispitamo razlike između liste i skupa:

Ispis 6.2: Lista i skup

```
public static void main( String[] args )
{
  List<String> list1 = new ArrayList<String>();
  List<String> list2 = new LinkedList<String>();

  Set<String> set1 = new HashSet<String>();
  Set<String> set2 = new TreeSet<String>();

  System.out.println( "Poziv s ArrayList:" );
  manipulateCollection( list1 );
  System.out.println( "\nPoziv s LinkedList:" );
```

```
  manipulateCollection( list2 );
  System.out.println( "\nPoziv s HashSet:" );
  manipulateCollection( set1 );
  System.out.println( "\nPoziv s TreeSet:" );
  manipulateCollection( set2 );
}
```

Ovdje je uveden čitav niz novih vrsta podataka. `List` i `Set` su sučelja izvedena iz sučelja `Collection`. Svako od njih dodaje metode specifične za listu odnosno skup. `ArrayList` i `LinkedList` su konkretne klase koje implementiraju listu. U praksi se rutinski koristi `ArrayList` koja elemente liste drži u polju (eng. *array*), a `LinkedList` samo iznimno, kad neko od svojstava `ArrayList` ne zadovoljava (npr. ako će postojati velike oscilacije u broju pospremljenih elemenata). `HashSet` i `TreeSet` su konkretne klase koje implementiraju skup. Rutinski se koristi `HashSet`, na bazi heš-tablice. Ovo je jedan primjer klase koja neće ispravno raditi ako ju koristimo s objektima koji nemaju ispravno izvedene metode `equals` i `hashCode` (vidjeti odjeljak 4.2). `TreeSet` je specifičan po tome što sortira svoje elemente (implementira i posebno sučelje za sortirane skupove – `SortedSet`). Da bi sortiranje bilo moguće, objekti trebaju imati definiran *prirodni poredak* (vidjeti odjeljak 6.8.1). Svi hijerarhijski odnosi spominjanih referentnih vrsta prikazani su na slici 6.1.

Za `ArrayList` i `LinkedList` dobit ćemo sljedeći ispis:

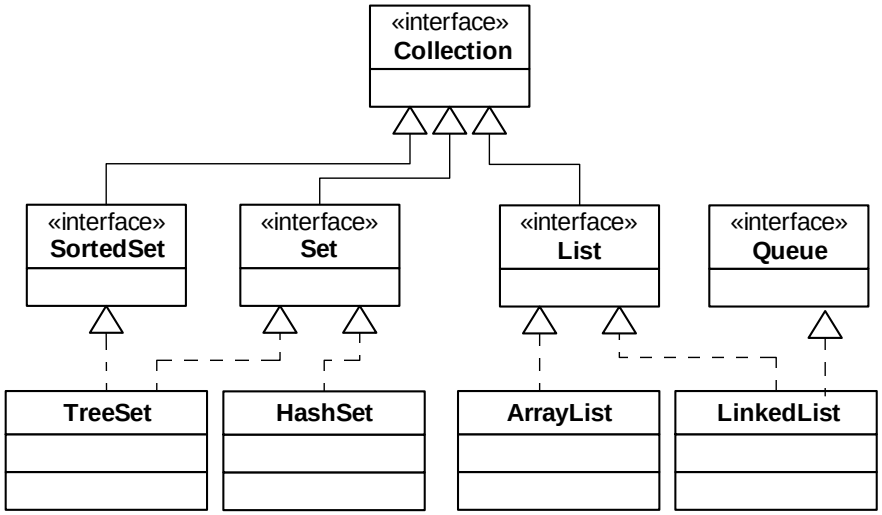
```
Poruka dodana u kolekciju
Sadrži li kolekcija poruku? true
Poruka dodana u kolekciju drugi put
Broj elemenata u kolekciji: 2
Poruka uklonjena iz kolekcije
Sadrži li kolekcija poruku? true
Poruka uklonjena iz kolekcije drugi put
Je li kolekcija prazna? true
```

Za `HashSet` i `TreeSet` ispis će izgledati drugačije:

```
Poruka dodana u kolekciju
Sadrži li kolekcija poruku? true
Broj elemenata u kolekciji: 1
Poruka uklonjena iz kolekcije
Sadrži li kolekcija poruku? false
Je li kolekcija prazna? true
```

Primijetimo da poruka nije dodana/uklonjena po drugi put jer skup po definiciji sadrži elemente bez ponavljanja. Dodatna osobina skupa je da ne zadržava poredak kojim su elementi dodavani u njega (za razliku od liste).

U metodi `main` korišten je udomaćeni pristup da se za vrstu varijable **uvijek koristi sučelje** – tipično `List` ili `Set`, ovisno o namjeni – a nikada konkretne vrste `ArrayList` ili `HashSet`. Same klase ne definiraju gotovo nijednu dodatnu metodu i gotovo nikad nema razloga koristiti ih za vrstu varijable (ili parametra, atributa itd.). Njihovim korištenjem samo onemogućujemo kasniji jednostavan prelazak na drugačiju implementaciju.



Slika 6.1: UML-ov dijagram klasa za *Collections Framework*

6.3 Sučelje `Queue`

Još jedna značajna vrsta kolekcije je *rep*. Dok su liste i skupovi po mnogočemu srodni i za mnogo namjena može poslužiti bilo koji od njih, repovi imaju sasvim drugačije namjene. Rep je najbolje zamisliti kao crnu kutiju s dva otvora – ulaznim i izlaznim. Na ulaz se ubacuju objekti, a s izlaza se vade. Način na koji objekti putuju od ulaza do izlaza je interna stvar repa. Najčešći način je tzv. FIFO (first in-first out), tj. da se na izlazu objekti pojavljuju istim redom kako su ubacivani na ulaz. Lista i skup se, s druge strane, koriste kao "prozirne" kutije i uglavnom se pristupa svim članovima redom. Interno, rep može biti organiziran kao lista, skup, ili na neki treći način. Primjerice, tipična izvedba FIFO-repa je pomoću vezane liste.

Najčešća primjena repa je, u skladu s njegovim imenom, za privremeno spremište objekata koji čekaju na svoj red za obradu. Uz primjenu složenijeg modeliranja repa može se upravljati raspoređivanjem računalskih

resursa po zahtjevima. Može se koristiti npr. prioritetni rep koji nekim vrstama posla dodjeljuje viši prioritet pa oni prije dolaze na red za obradu i stoga, statistički gledano, dobivaju veći dio raspoloživih resursa. Repu se može zadati maksimalna duljina, čime se može upravljati maksimalnim vremenom čekanja na ispunjenje zahtjeva i odbijati one zahtjeve koji bi predugo čekali.

Javino sučelje `Queue` izvedeno je iz sučelja `Collection`, što znači da sadrži sve metode definirane za njega, a dodaje još pet metoda. Jedna služi za dodavanje, a ostalih četiri su varijante metoda za dohvaćanje, koje pokrivaju sve četiri moguće kombinacije sljedećih dvaju izbora:

- objekt koji je na redu se dostavlja sa ili bez vađenja iz repa;
- ako je rep prazan, to se signalizira bacanjem iznimke ili samo vraćanjem `null`.

Metode su sljedeće:

- `offer` zahtijeva dodavanje elementa u rep. Ako ga rep ne može prihvatiti, metoda vraća `false`.
- `poll` dostavlja sljedeći element i uklanja ga iz repa. Ako je rep prazan, vraća `null`.
- `remove` dostavlja sljedeći element i uklanja ga iz repa. Ako je rep prazan, baca iznimku `NoSuchElementException`.
- `peek` dostavlja sljedeći element bez vađenja iz repa. Ako je rep prazan, vraća `null`.
- `element` dostavlja sljedeći element bez vađenja iz repa. Ako je rep prazan, baca iznimku `NoSuchElementException`.

Umjesto metode `offer` može se koristiti i standardna metoda za kolekcije, `add`. Razlika je u tome što `add` baca iznimku ako se element ne može dodati pa ju treba koristiti samo u slučaju da slučaj odbijanja elementa nije prihvatljiv i predstavlja pogrešku u radu.

Metode iz sučelja `Collection` omogućuju i pristup repu po modelu "prozirne kutije". Moguće je npr. prelaziti po svim članovima te dodavati i vaditi elemente na proizvoljnim mjestima u repu. Međutim, u kontekstu repova

te operacije su od sporednog značaja, a njihovim korištenjem lako se može i narušiti funkcija repa.

U Javinoj standardnoj biblioteci postoji prilično velik broj izvedbi repova koje se razlikuju ne samo po pravilima za redoslijed prolazaka elemenata kroz rep, nego i po detaljima oko usklađivanja pristupa repu iz više niti. Repovi se naime redovito koriste za komunikaciju između niti. Kao primjer najjednostavnije izvedbe možemo spomenuti klasu `LinkedList`, koja ujedno implementira i sučelje `List`. Ona je jedina prikazana na slici 6.1.

6.4 Prelazak preko svih članova kolekcije

Najčešći način korištenja kolekcije je da se prijeđe po svakom njenom članu, jedan po jedan, i obavi neki posao. Za to postoji posebna jezična konstrukcija, tzv. petlja *enhanced for*[1]:

Ispis 6.3: Prolaz po kolekciji petljom *enhanced for*

```
static void iterateOverCollection( Collection<String> col )
{
  for ( String s : col )
    System.out.println( "Član kolekcije: " + s );
}

public static void main( String[] args )
{
  List<String> list = new ArrayList<String>();
  list.add( "a" );
  list.add( "b" );
  list.add( "c" );
  iterateOverCollection( list );
}
```

U metodi `iterateOverCollection` demonstrirana je konstrukcija *enhanced for*. Izraz u zagradama prije dvotočke je deklaracija varijable koja će u svakom koraku petlje sadržati po jedan element kolekcije. Član iza dvotočke je kolekcija po kojoj treba obaviti prelazak.

Ako metodu `main` prepravimo da koristi `HashSet`, moći ćemo uočiti da se po elementima ne prolazi u istom poretku u kojem smo ih dodavali:

```
public static void main( String[] args )
{
  Set<String> set = new HashSet<String>();
```

[1] U praksi se još koriste i izrazi *for each* i *for/in*

```
    set.add( "a" );
    set.add( "b" );
    set.add( "c" );
    iterateOverCollection( set );
  }
```

Ispis:

```
Član kolekcije: a
Član kolekcije: c
Član kolekcije: b
```

6.5 Kolekcije i primitivne vrste podataka

Podaci primitivne vrste ne mogu se spremati u kolekcije pa se koriste objekti-omotači podataka primitivnih vrsta. Jezično svojstvo *auto-boxing/unboxing* dozvoljava sintaksu po kojoj izgleda kao da se spremaju sami podaci primitivne vrste. Primjer:

Ispis 6.4: Kolekcija i primitivne vrste podataka

```
public static void main( String[] args )
{
  List<Integer> intList = new ArrayList<Integer>();
  intList.add( 2 );
  intList.add( 3 );
  intList.add( 4 );
  if ( intList.contains( 3 ) )
    System.out.println( "Lista sadrži broj 3" );
  for ( int i : intList )
    System.out.println( "Član liste: " + i );

  List<Boolean> boolList = new ArrayList<Boolean>();
  boolList.add( true );
  boolList.add( 2 == 3 );

  List<Character> charList = new ArrayList<Character>();
  charList.add( 'a' );
  charList.add( 'b' );
  char c = charList.get( 0 ); // dohvaća prvi element liste
}
```

6.6 Napredni prelazak po članovima kolekcije

Konstrukcija *enhanced for* prikazana u odjeljku 6.4 kraći je zapis za ovakav kod, koji se implicitno generira prilikom prevođenja te konstrukcije i koristi

poznatu sintaksu petlje `for`:

Ispis 6.5: Prolaz po kolekciji eksplicitnim korištenjem iteratora

```
static void iterateOverCollection( Collection<String> col )
{
  for ( Iterator<String> iter = col.iterator(); iter.hasNext(); )
  {
    String s = iter.next();
    System.out.println( "Član kolekcije: " + s );
  }
}
```

(u realnom slučaju ne koristi se ime varijable `iter`, koje bi tada bilo rezervirano, nego neko ime koje se ne može normalno pojaviti u kodu.)

Za neke specifične namjene potrebno je koristiti upravo ovakvu sintaksu jer nam je tako eksplicitno dostupan poseban objekt – *iterator* po kolekciji. Njega možemo zamisliti kao pokazivač na trenutni objekt u kolekciji. Preusmjerujemo ga na sljedeći element pozivom metode `next`, čija povratna vrijednost je taj sljedeći element. Prije prvog poziva te metode iterator još ne pokazuje ni na jedan element, tako da se prvim pozivom dobiva prvi element.

Jedna od dodatnih mogućnosti je metoda `remove` kojom se trenutni objekt uklanja iz kolekcije:

Ispis 6.6: Korištenje iteratora za mijenjanje liste

```
public static void main( String[] args )
{
  List<Integer> intList = new ArrayList<Integer>();
  intList.add( 2 );
  intList.add( 3 );
  intList.add( 4 );

  for ( Iterator<Integer> iter = intList.iterator(); iter.hasNext(); )
  {
    int i = iter.next();
    if ( i == 3 )
      iter.remove();
  }
  for ( int i : intList )
    System.out.println( "Član kolekcije: " + i );
}
```

Postoje i druge mogućnosti, o kojima se može informirati u dokumentaciji [1]. Za prolazak po listama postoji i specijalizirani `ListIterator` koji nudi još više mogućnosti, specifičnih za liste.

6.7 Mapa

Pojam *mapa* odnosi se na skup preslikavanja između nekih podataka (*vrijednosti*) i s njim povezanih identifikacijskih podataka (njihovih *ključeva*). Na primjer, većina baza podataka koje sadrže podatke o državljanima RH kao ključ koriste JMBG jer je to kratak podatak jedinstven za svakog građana. Ključ je dakle što kraći podatak na osnovu kojega se može nedvosmisleno identificirati neki veći slog podataka.

Mapa je korisna struktura podataka za velik broj namjena, npr. kad god imamo zbirku objekata koje je potrebno dohvaćati po imenu ili nekom drugom ID-ju. U Javi je za mape definirano sučelje `Map`. Ovaj primjer demonstrira njegovu uporabu:

Ispis 6.7: Klasa `Student` koja demonstrira mapu

```
public class Student
{
  private String id;
  private String firstName;
  private String lastName;

  public Student( String id, String firstName, String lastName )
  {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
  }

  ... metode getId, getFirstName, getLastName ...

  public String toString()
  {
    return this.firstName + " " + this.lastName;
  }

  public static void main( String[] args )
  {
    Student stud1 = new Student( "id1", "Bill", "McGill" );
    Student stud2 = new Student( "id2", "John", "Pepper" );

    Map<String, Student> students = new HashMap<String, Student>();
    students.put( stud1.id, stud1 );
    students.put( stud2.id, stud2 );

    Student fetchedStudent = students.get( "id2" );
    System.out.println( "Dohvaćeni student: " + fetchedStudent );
  }
}
```

Definirana je mala klasa koja sadrži podatke o studentu. Instance te klase spremaju se u mapu tako da se kao ključ koristi atribut `id`. Općenito nije nužno da ključ bude dio objekta, ali je uobičajeno. I za ključ i za vrijednost može se koristiti bilo koji objekt, bilo koje vrste. Vrsta `Map` je parametrizirana s dva parametra – prvi je vrsta ključa, a drugi vrsta vrijednosti. Isto se, naravno, odnosi i na `HashMap`. Ovo je, inače, još jedna klasa koja se oslanja na ispravno funkcioniranje metoda `equals` i `hashCode` u objektima koji se koriste kao ključ. Klasa `String` ima uredno implementirane ove metode pa neće biti problema.

Iz primjera mogu se iščitati dvije najvažnije metode sučelja `Map`: `put(` *`key, value`* `)` i `get(` *`key`* `)`. Za ostale raspoložive metode treba konzultirati dokumentaciju [1].

6.8 Često korišteni algoritmi nad kolekcijama

U paketu `java.util` postoji klasa `Collections` koja ne služi za instanciranje, već sadrži poveći broj statičkih metoda koje provode često korištene algoritme za obradu kolekcija. Postoje npr. `shuffle`, `reverse`, `sort`, `min`, `max` itd. Za ove algoritme kaže se da su *polimorfni* jer, služeći se Javinim polimorfizmom, mogu raditi s velikim brojem različitih vrsta kolekcija. Za mnoge algoritme (prije svega `sort`) potrebno je da elementi kolekcije budu *usporedivi*, što tehnički znači da moraju ispravno implementirati sučelje `Comparable`.

6.8.1 Implementiranje sučelja `java.lang.Comparable<T>`

Sučelje `Comparable<T>` je *parametrizirano* – parametar je neka vrsta `T`. U načelu se kao parametar navodi upravo klasa u kojoj implementiramo to sučelje. Time kažemo kompilatoru da se objekte naše klase može uspoređivati samo s drugim objektima iste klase. Općenito, tu bi se mogla staviti bilo koja klasa, ali druge vrijednosti neće imati smisla – osim eventualno neke nadklase trenutne klase, u slučaju da imamo više sličnih klasa čije instance se mogu međusobno uspoređivati.

Ovo sučelje sadrži samo jednu metodu: `compareTo( T )`. Povratna vrijed-

nost je `int` koji mora biti nula za jednake objekte, negativan ako je trenutni objekt "manji" od proslijeđenog i pozitivan ako je trenutni objekt "veći" od proslijeđenog. Značenje pojmova "manji" odnosno "veći" ovisi o vrsti objekta. Poredak objekata u kojem se nijedan objekt ne pojavljuje prije nekog drugog koji je od njega "manji" naziva se *prirodnim poretkom* tih objekata. Metoda `compareTo`, dakle, određuje prirodni poredak instanci klase u kojoj je implementirana. Na primjer, u klasi `Student` iz ispisa 6.7 mogli bismo tu metodu implementirati ovako:

Ispis 6.8: Metoda `compareTo`

```
public class Student
implements Comparable<Student>
{
  ... isti kod kao ranije ...

  public int compareTo( Student that )
  {
    return this.lastName.compareTo( that.lastName );
  }
}
```

Klasa sad najavljuje da implementira sučelje `Comparable`. Metoda `compareTo`, oslanjajući se na `String`-ovu implementaciju iste metode, uspoređuje studente po prezimenu.

6.8.2 Poredak različit od prirodnog – sučelje `java.util.Comparator<T>`

Sve metode u klasi `Collections` koje se oslanjaju na prirodni poredak objekata imaju i verzije koje se oslanjaju na poseban objekt-*komparator*, koji može nametnuti proizvoljan poredak nad objektima. Takav objekt mora implementirati sučelje `java.util.Comparator<T>`. Evo primjera komparatora za studente koji nameće poredak po ID-ju:

Ispis 6.9: Komparator za studente koji nameće poredak po ID-ju

```
public class StudentComparatorById
implements java.util.Comparator<Student>
{
  public int compare( Student left, Student right )
  {
    return left.getId().compareTo( right.getId() );
  }
}
```

6.8.3 Primjer korištenja metode `Collections.sort`

```
public static void main( String[] args )
{
  List<Student> students = Arrays.asList( new Student[]
  {
    new Student( "005", "Joshua", "Bloch" ),
    new Student( "003", "Paul", "Graham" ),
    new Student( "004", "Richard", "Dawkins" ),
    new Student( "002", "Rick", "Martinoff" ),
  }
  );

  System.out.println( "Početni popis:" );
  for ( Student student : students )
    System.out.println( student );

  Collections.sort( students );
  System.out.println( "\nStudenti sortirani po prezimenu:" );
  for ( Student student : students )
    System.out.println( student );

  Collections.sort( students, new StudentComparatorById() );
  System.out.println( "\nStudenti sortirani po ID-ju:" );
  for ( Student student : students )
    System.out.println( student );
}
```

Očekivani ispis:

```
Početni popis:
005 Joshua Bloch
003 Paul Graham
004 Richard Dawkins
002 Rick Martinoff

Studenti sortirani po prezimenu:
005 Joshua Bloch
004 Richard Dawkins
003 Paul Graham
002 Rick Martinoff

Studenti sortirani po ID-ju:
002 Rick Martinoff
003 Paul Graham
004 Richard Dawkins
005 Joshua Bloch
```

Poglavlje 7

Polja

Polje je blok u memoriji u koji je spremljen niz podataka iste vrste. U Javi polje je izvedeno kao objekt, ali posebnog karaktera. Polja su objekti posebne *referentne vrste polja* i njima se barata putem referenci, kao i ostalim objektima. Ovo treba uvijek imati na umu jer vrijedi sve rečeno za referentne varijable u odjeljku 3.1.2.

Sva polja u osnovi su jednodimenzionalna. Dvodimenzionalno polje izvedeno je kao jednodimenzionalno polje referenci na druga jednodimenzionalna polja.

7.1 Osnovni elementi sintakse

Jednodimenzionalna polja:

Ispis 7.1: Rad s jednodimenzionalnim poljem

```
int[] intArray; // deklaracija varijable polja
intArray = new int[3]; // kreiranje praznog polja (popunjeno nulama)
int i = intArray[0]; // pristup članu polja
intArray = { 1, 2, 3 }; // kreiranje uz popunu
                        // navedenim vrijednostima
```

Uočimo sintaksu u posljednjem retku: na desnoj strani ne treba koristiti operator `new` i točnu vrstu podataka, dovoljan je samo doslovan popis članova. Java će automatski stvoriti polje one vrste koja je deklarirana na lijevoj strani.

Članovi polja inicijaliziraju se po istim pravilima kao i atributi objekta – brojevi na nulu, logičke vrijednosti na `false`, a reference na `null`.

Dvodimenzionalna polja (analogno i za višedimenzionalna):

Ispis 7.2: Rad s dvodimenzionalnim poljem

```
int[][] intArray;
intArray = new int[2][3];
int i = intArray[0][0];
intArray = { {1, 2}, {3, 4}, {5, 6} };
```

Petlja *enhanced for* za prolaz po svim članovima polja:

Ispis 7.3: Prolaz po svim članovima polja petljom *enhanced for*

```
public static void main( String[] args )
{
  int[] intArray = { 1, 2, 3, 4, 5 };
  for ( int i : intArray )
    System.out.print( " " + i );

  System.out.println();
}
```

Dvodimenzionalno polje ne mora nužno biti pravokutno, što slijedi iz rečenog da je to ustvari polje polja. Sljedeći primjer demonstrira ovo:

Ispis 7.4: Trokutasto dvodimenzionalno polje

```
public static void main( String[] args )
{
  int[][] int2Array = { {11}, {21, 22}, {31, 32, 33} };

  for ( int[] intArray : int2Array )
  {
    for ( int i : intArray )
      System.out.print( " " + i );

    System.out.println();
  }
}
```

U prvom retku koda kreira se polje polja tako da je njegov prvi član polje duljine jedan, drugi član polje duljine dva i treći član polje duljine tri. Zatim slijedi dvostruka petlja u kojoj se ispisuje cjelokupan sadržaj stvorene strukture. Time se vizualizira trokutasti izgled strukture. Ispis je ovakav:

```
11
21 22
31 32 33
```

Polja mogu sadržavati i reference na objekte. U deklaraciji se navodi referentna vrsta, npr. `String[] stringArray`.

7.2 Polje kao objekt

Polje, iako nije u pravom smislu instanca neke klase, svejedno je objekt koji je `instanceof java.lang.Object`, pa raspolaže i svim odgovarajućim metodama. Svako polje ima i cjelobrojni atribut `length`, jednak njegovom kapacitetu. Atribut je `final` jer se kapacitet polja ne može naknadno mijenjati. Koristi ga se npr. za složenije primjere prolaska po svim članovima polja:

Ispis 7.5: Prolaz po polju korištenjem eksplicitnog indeksa

```
int[] intArray = new int[3];

for ( int i = 0; i < intArray.length; i++ )
{
  int j = intArray[i];
  ...
}
```

7.3 Interakcija s kolekcijama

Kolekcije su fleksibilnije i općenito korisnije strukture od polja. Međutim, polja su "siroviji" oblik strukture i stoga računski i memorijski efikasnija. Prednost je i što se za rad s poljima može koristiti elegantnija sintaksa. Javina biblioteka podržava laku transformaciju između kolekcija i polja:

Ispis 7.6: Pretvorbe između polja i liste

```
String[] stringArray = { "Harry", "Moe", "Joe" };

// konverzija polja u listu:
List<String> stringList = Arrays.asList( stringArray );

// konverzija liste u polje:
String[] stringArray2 = stringList.toArray( new String[0] );
```

Metodi `toArray` mora se proslijediti makar prazno polje jer je potrebna informacija o tome koju vrstu polja treba stvoriti. Ovdje, nažalost ne pomažu *generics* i to se ne događa automatski. Ako je proslijeđeno polje dovoljno veliko, iskoristit će se; u protivnom se stvara novo.

Metoda `asList` može se iskoristiti i za jednostavno stvaranje liste popunjene doslovno navedenim vrijednostima:

```
List<String> stringList = Arrays.asList( "Harry", "Moe", "Joe" );
```

Dobivena lista ima fiksiran sadržaj – zabranjeno je pozivati metode `add`, `remove` i ostale koje bi ga mijenjale.

Poglavlje 8

Komunikacija s okolinom

Svaka aplikacija treba komunicirati s okolinom. To se prije svega odnosi na korisničko sučelje (tipkovnica i zaslon), sustav datoteka i računalsku mrežu. Ovdje se prikazuje na koji način se u jeziku Java komunicira s okolinom.

8.1 Tokovi podataka

Osnovna paradigma koja se u Javi koristi za razmjenu podataka s okolinom je *tok* podataka (eng. *data stream*). Postoje ulazni i izlazni tokovi i predstavljeni su apstraktnim klasama `InputStream` odnosno `OutputStream`. Nalaze se u paketu posvećenom komunikaciji, `java.io`. Za svaku vrstu resursa s kojim se može komunicirati (zaslon, datoteka, udaljeno računalo itd.) postoje konkretne klase izvedene iz njih. Apstraktne klase deklariraju osnovne metode za rad s tokom – `available`, `read`, `reset` i `close` za `InputStream` odnosno `write`, `flush` i `close` za `OutputStream`. Sljedeći primjer demonstrira korištenje izlaznog toka:

Ispis 8.1: Rad s izlaznim tokom

```
import java.io.OutputStream;
import java.io.IOException;
...
  static void sendToOutput( OutputStream os )
  throws IOException
  {
    for ( int i = 0; i < 256; i++ )
      os.write( i );

    os.flush();
```

```
    ... daljnje operacije nad tokom ...

    os.close();
  }
```

Ova metoda može raditi s bilo kakvim izlaznim tokom. U nju će zapisati niz okteta od nule do 255 i nakon toga ju zatvoriti. Metoda `flush` poziva se kad se želi osigurati da podaci budu odmah isporučeni na odredište. U protivnom će se najvjerojatnije gomilati u međuspremniku u radnoj memoriji dok ih se ne skupi dovoljno za masovno slanje. Korištenje međuspremnika općenito značajno podiže efikasnost komunikacije jer se šalju veći blokovi podataka, tako da `flush` **ne treba koristiti** ako to nije zaista potrebno.

Ovaj primjer demonstrira korištenje ulaznog toka:

Ispis 8.2: Rad s ulaznim tokom

```
import java.io.InputStream;
import java.io.IOException;
...
  static void receiveInput( InputStream is )
  throws IOException
  {
    while ( true )
    {
      int i = is.read();
      if ( i == -1 )
        break;
      System.out.println( i );
    }
    is.close();
  }
```

Metoda `read` vraća pročitani oktet. Ako vrati vrijednost `-1`, to znači da je tok došao do kraja. Općenito se ne može samo ispitati je li tok došao do kraja bez pokušaja čitanja pa ne postoji posebna metoda za to koju bi se moglo koristiti u uvjetu petlje `while`. Metoda `available` vraća broj okteta raspoloživih odmah, bez čekanja. Ako vrati nulu, to ne mora značiti da je tok došao do kraja, već samo da trenutno nema novih podataka.

8.2 Pisači i čitači

Za komunikaciju tekstualnim podacima postoje posebni omotači "sirovih" tokova okteta koji ih pretvaraju u tokove znakova. U Javinoj terminologiji

to su *pisači* (osnovna klasa `Writer`) i *čitači* (osnovna klasa `Reader`). Koriste se slično kao sirove tokove, ali barataju podacima vrste `char`. Još je praktičnije koristiti klase s ugrađenim međuspremnikom (`BufferedReader` i `PrintWriter`) koje omogućuju komunikaciju redak po redak umjesto znak po znak. Ovo je prikazano u primjerima:

Ispis 8.3: Pisač i čitač

```
import java.io.Writer;
import java.io.PrintWriter;
import java.io.Reader;
import java.io.BufferedReader;
import java.io.IOException;
...
  static void writeOutput( Writer writer )
  throws IOException
  {
    PrintWriter pw = new PrintWriter( writer );
    pw.println( "Prvi redak" );
    pw.println( "Drugi redak" );
    pw.flush();
    pw.close();
  }

  static void readInput( Reader reader )
  throws IOException
  {
    BufferedReader br = new BufferedReader( reader );
    while ( true )
    {
      String line = br.readLine();
      if ( line == null || line.length() == 0 )
        break;
      System.out.println( line );
    }
    br.close();
  }
```

Metoda `readInput` završit će ako naiđe na kraj ulaznog toka (`line == null`) ili ako pročita prazan redak (`line.length() == 0`).

Pisač i čitač rade tako da se povezuju sa sirovim tokom okteta i obavljaju potrebne konverzije. Povezuje ih se pri instanciranju prosljeđivanjem objekta toka konstruktoru. Sljedeći primjer prikazuje metode koje pozivaju metode iz ispisa 8.3 s odgovarajuće omotanim sirovim tokovima:

Ispis 8.4: Omotači podatkovnih tokova

```
import java.io.OutputStream;
```

```
import java.io.OutputStreamWriter;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
...
  static void writeOutput( OutputStream os )
  throws IOException
  {
    writeOutput( new OutputStreamWriter( os ) );
  }

  static void readInput( InputStream is )
  throws IOException
  {
    readInput( new InputStreamReader( is ) );
  }
```

8.3 Komunikacija s korisnikom

Osnovna tekstualna komunikacija s korisnikom ostvaruje se putem standardnog ulaz/izlaza koji pruža svaka računalska platforma. U Javi se s njima povezuje putem statičkih atributa u klasi `System`. Ispis na standardni izlaz sreli smo već mnogo puta u kodu, svaki put kad se koristio izraz `System.out.println( ... )`. Statički atribut `out` sadrži referencu na objekt vrste `PrintStream`, koja je vrlo srodna prikazanoj vrsti `PrintWriter` i definira mnoge identične metode. Za praktično korištenje standardnog ulaza potrebno je eksplicitno dodati omotače, kao u ispisu 8.4. Kod iz ispisa 8.3 i 8.4 možemo isprobati dodavanjem sljedeće metode `main`:

```
import java.io.IOException;
...
  public static void main( String[] args )
  throws IOException
  {
    System.out.println( "Utipkaj nešto!" );
    readInput( System.in );
  }
```

U konzolu treba utipkati tekst i pritisnuti Enter. Program će ponoviti utipkano. Program dovršavamo utipkavanjem praznog retka. Atribut `System.in` je vrste `InputStream` pa će se pozvati metoda `readInput( InputStream )` iz ispisa 8.4, koja će onda proslijediti poziv metodi `readInput( BufferedReader )` iz ispisa 8.3.

8.4 Rad s datotekama

Datotekama se u Javi barata putem objekata vrste `File`. To su tzv. *handle*-objekti jer postojanje objekta ne znači da postoji i datoteka koju on predstavlja. Objektu se zada putanja do datoteke i zatim ga se može pitati postoji li datoteka, je li u pitanju prava datoteka ili kazalo itd. Ako datoteka ne postoji, može ju se stvoriti. Sljedeći kod demonstrira rad s datotekama. I on se oslanja na metode iz ispisa 8.3 i 8.4.

```
import java.io.IOException;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.File;
...
  public static void main( String[] args )
  throws IOException
  {
    File file = new File( "Text.txt" );
    file.createNewFile();
    System.out.println( "Zapisujem u datoteku..." );
    writeOutput( new FileOutputStream( file ) );
    System.out.println( "Čitam iz datoteke..." );
    readInput( new FileInputStream( file ) );
  }
```

Možemo vidjeti da se datoteku otvara za pisanje instanciranjem klase `FileOutputStream` kojoj se prosljeđuje instanca klase `File`. Čim se taj objekt kreira, datoteka je već otvorena i spremna za zapisivanje. Analogno se i čita iz datoteke.

8.5 Serijalizacija podataka

Serializacija podataka je postupak pretvaranja objekata u serijski niz primitivnih podataka (brojeva). Obrnuti postupak se zove deserijalizacija. Takav serijaliziran niz podataka se onda može spremiti u trajnu memoriju (npr. disk) ili ga možemo poslati preko mreže na drugo računalo.

Da bi se neki objekt mogao automatski serijalizirati potrebno je da taj objekt implementra sučelje `java.io.Serializable`. To sučelje nema niti jedne metode, već služi JVM-u da prepozna objekte koji se mogu serijalizirati. Ako objekt sadrži reference na druge objekte onda i oni moraju imati implementirano sučelje `java.io.Serializable` u protivnom će doći do pogreške. Svi objekti u Javinoj okolini za koje ima smisla (koji sadrže podatke) već

imaju implementirano to sučelje (npr. kolekcije). Primjer klase `PhoneNumber` pripremljene za serijalizaciju vidimo u ispisu 8.5.

Ispis 8.5: Klasa `PhoneNumber` priremljena za serijalizaciju

```
package hr.fer.tel.ilj;

import java.io.Serializable;

public class PhoneNumber implements Serializable {
  private int number, area, country;


  public PhoneNumber(int country, int area, int number) {
    this.country = country;
    this.area = area;
    this.number = number;
  }

  public int getNumber() {
    return number;
  }

  public int getArea() {
    return area;
  }

  public int getCountry() {
    return country;
  }
...
}
```

Za samu serijalizaciju i deserijalizaciju se koriste posebne klase toka: `ObjectInputStream` za deserijalizaciju i `ObjectOutputStream` za serijalizaciju. Ove klase su zapravo omotači koji dalje koriste ostale tokove podataka. Primjer serijalizacije u datoteku je prikazan na ispisu 8.6.

Ispis 8.6: Primjer serijalizacije u datoteku

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializeToFile {

  public static void main(String[] args) throws IOException {
    FileOutputStream fos = new FileOutputStream("name.dat");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
    oos.writeObject("Mario Kusek");
    oos.writeObject(new PhoneNumber(385, 1, 6129801));

    oos.close();
  }
}
```

Prvo se treba napraviti izlazni tok u datoteku (`FileOutputStream`). Nakon toga se napravi serijalizacijski omotač (`ObjectOutputStream`). Kada je to napravljeno onda pozivom metode `writeObject` serijaliziramo objekt koji je poslan kao parametar i serijalizirani podaci se šalju toku datoteke koju je omotala tok za serijalizaciju. Na kraju trebamo zatvoriti tok podataka. U serijalizacijskom omotaču toka postoji više metoda `writeX`, a svaka služi za serijalizaciju druge vrste podataka (npr. `writeInt(int val)`.

Na sličan način se koristi i deserijalizacija. Ispis 8.7 prikazuje upravo to.

Ispis 8.7: Primjer deserijalizacije iz datoteke

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializeFromFile {

  public static void main(String[] args)
      throws IOException, ClassNotFoundException {
    FileInputStream fis = new FileInputStream("name.dat");
    ObjectInputStream ois = new ObjectInputStream(fis);

    String name = (String) ois.readObject();
    System.out.println("name=" + name);
    PhoneNumber phone = (PhoneNumber) ois.readObject();
    System.out.println("phone" + phone);

    ois.close();
  }
}
```

Prvi dio postupka je isti tj. treba napraviti tok čitanja iz datoteke (`FileInputStream`) i nju omotati tokom deserijalizacije (`ObjectInputStream`). Za čitanje koristimo metode `readX`. U ovom slučaju koristimo `readObject`. Posebnost ove metode je da ona može baciti iznimku `ClassNotFoundException` jer je moguće da pokušamo pročitati objekt iz datoteke za koji nemamo izvršni kod. Ova metoda vraća vrstu `Object`, a pošto mi trebamo `String` onda taj objekt trebamo *castati* u `String`. Ovdje treba paziti jer se može dogoditi iznimka `ClassCastException`. U našem slučaju se to neće dogoditi jer znamo

da smo u datoteku prvo snimili objekt klase `String`, a nakon toga objekt klase `PhoneNumber`.

Ako prvo snimimo neku klasu i u međuvremenu ju promijenimo, dodamo/maknemo/promijenimo neki atribut ili metodu, onda čitanje iz datoteke neće raditi i prilikom čitanja će se dogoditi iznimka `java.io.InvalidClassException`. To je tipičan problem za vrijeme programiranja, ali se može javiti kod isporuke novije verzije programa korisniku. Da bi se to izbjeglo potrebno je u svaku klasu koja se serijalizira dodati konstantu `serialVersionUID`. Evo isječka koda:

```
...
public class PhoneNumber implements Serializable {
  private static final long serialVersionUID = 120331323592941832L;
...
}
```

Vrijednost konstante je proizvoljan i ne smije se mijenjati ako želimo i dalje učitavati podatke koji su snimljeni s tom konstantom.

8.6 Rad s internetskim konekcijama

Za dohvat sadržaja s nekog URL-a dovoljna su samo dva retka koda (uz već postojeći kod iz ispisa (8.3 i 8.4):

```
java.net.URL url = new java.net.URL( "http://www.google.com" );
readInput( url.openStream() );
```

Klasa `URL` predstavlja lokaciju na Internetu identificiranu URL-om (*uniform resource locator*). Pozivom metode `openStream` automatski se obavljaju sve radnje potrebne za pristup resursu pod tim URL-om – kontaktira se DNS, otvara mrežna konekcija na potrebnom portu, šalje HTTP-ov zahtjev za dostavom resursa i kreira objekt toka podataka koji će isporučivati podatke koje poslužitelj šalje kao odgovor.

Poglavlje 9

Uzorci u dizajnu koda

U praksi, pri izradi programskih rješenja mnoge vrste problema koje treba rješavati opetovano se pojavljuju bez obzira na konkretni zadatak. S vremenom su sazrjela za svaki takav tipičan problem najbolja rješenja, koja pokrivaju sve suptilnije nijanse problema[1]. Razvijatelj si može drastično olakšati život ako prouči ta rješenja tako da ih može primijeniti kad zatreba. Također, baratanje vokabularom vezanim uz uzorke (prije svega prepoznavanje uzoraka po imenu) bitno olakšava komunikaciju između razvijatelja, koji tako mogu puno efikasnije raspravljati o svojim programskim rješenjima.

Ideja uzoraka u dizajnu koda doživjela je najveći proboj 1995. godine, kad je objavljena kultna knjiga "Design Patterns: Elements of Reusable Object-Oriented Software" [4] četvorice autora popularno zvanih "The Gang of Four" ili GoF, kako se najčešće oslovljava i samu knjigu. U ovom poglavlju bit će kao primjer ove koncepcije prikazano nekoliko najjednostavnijih uzoraka: *Template method*, *Observer*, *Adapter* i *Decorator*.

9.1 *Template method*

Ovaj uzorak jednostavan je za razumjeti i ima vrlo široku primjenu. Ustvari, primijenili smo ga još u prvom poglavlju, prilikom refaktorizacije u odjeljku 2.14. Drugo ime za *Template method* je *polimorfna metoda*. Sličan izraz, *polimorfni algoritam*, spominjao se već u poglavlju o kolekcijama, u odjeljku 6.8. Dakle, radi se naprosto o metodi koja u svojem kodu sadrži

[1] U ovoj skripti se, međutim, neće ulaziti u te nijanse. Ovdje se daje samo najosnovniji uvod u koncepciju uzoraka.

pozive metoda koje podliježu polimorfizmu odnosno dinamičkom povezivanju. Metoda na apstraktnoj razini implementira neki algoritam, gdje se pojedini koraci definiraju samo na razini "što" a ne i "kako". U prvom poglavlju imali smo sljedeći trivijalan primjer polimorfne metode:

```
abstract class AbstractCounter
implements Counter
{
  ...

  public final void increment()
  {
    setState( getState() + 1 );
  }

  abstract void setState( int newState );
}
```

Metoda `increment` je polimorfna jer se oslanja na apstraktnu metodu `setState`. Dakle, određuje da se mora "postaviti novo stanje", ali ne propisuje točno kako se postavlja novo stanje. To će ovisiti o konkretnoj implementaciji metode `setState`. U našem primjeru imali smo dvije varijante te metode: u klasi `SimpleCounter` jednostavno se proslijeđeni `int` zapisao u atribut, dok se u klasi `ModuloCounter` zapisao ostatak dijeljenja s modulom.

9.2 *Observer*

Recimo da razvijamo softver za simulaciju rada mreže TCP/IP. Svaki komunikacijski čvor može primiti neki paket namijenjen bilo njemu, bilo nekom daljnjem čvoru i treba ga u skladu s time proslijediti dalje ili "konzumirati". Želimo da našem objektu koji predstavlja komunikacijski čvor možemo dodavati procedure za obradu događaja "paket stigao u čvor X". Na primjer, u najjednostavnijem slučaju trebat će ispisati tekstualnu poruku na standardni izlaz. U malo naprednijem slučaju trebat će prikazati poruku u prozoru grafičkog sučelja. Za očekivati je i da ćemo imati grafički prikaz mreže na kojem treba vizualizirati putovanje paketa od čvora do čvora. U tom slučaju bit će potrebno kad paket stigne u čvor promijeniti neki detalj u prikazu, npr. boju čvora. Sve te mogućnosti želimo imati na raspolaganju kao *korisnik* klase koja predstavlja mrežu TCP/IP, što znači bez ikakvih intervencija u izvorni kod te klase – dapače, bez posjedovanja

njenog izvornog koda. Očito je da će klasa čvora morati imati ugrađenu podršku za to. Ta podrška ugrađuje se korištenjem uzorka *Observer*.

Prvo što treba uvesti je objekt-*osluškivač* (to je *observer*, ali u kontekstu Jave uobičajen je izraz *listener*). On ima metodu koju će objekt čvora pozvati i time signalizirati "stigao je paket". Osluškivača kreiramo i prijavimo čvoru – pozivanjem čvorove metode npr. `addReceiveListener`. Svaki put kad primi paket, čvor mora proći po listi svih prijavljenih osluškivača i pozvati njihovu metodu npr. `packetReceived`. U toj metodi nalazi se kod koji obrađuje događaj primitka paketa.

Za osluškivača prvo treba definirati sučelje, `ReceiveListener`. Njega onda mogu implementirati različite verzije osluškivača.

Ispis 9.8: Sučelje osluškivača dolaznih paketa

```
package hr.fer.tel.ilj.observer;

public interface ReceiveListener
{
  void packetReceived( Packet packet, Node receivingNode );
}
```

Pokažimo odmah i jednu jednostavnu implementaciju osluškivača:

Ispis 9.9: Jednostavna implementacija osluškivača dolaznih paketa

```
package hr.fer.tel.ilj.observer;

public class SysoutReceiveListenerHr
implements ReceiveListener
{
  public void packetReceived( Packet packet, Node receivingNode  )
  {
    System.out.println(
        "Čvor " + receivingNode.getName() + " je primio paket: "
        + packet.getData() );
  }
}
```

Klasa komunikacijskog čvora s podrškom za osluškivače:

Ispis 9.10: Komunikacijski čvor s podrškom za osluškivače

```
package hr.fer.tel.ilj.observer;

import java.util.HashSet;
import java.util.Set;
```

```
public class Node
{
  private final Set<ReceiveListener> recListeners =
    new HashSet<ReceiveListener>();

  private final String name;

  public Node( String name )
  {
    this.name = name;
  }

  public void addReceiveListener( ReceiveListener listener )
  {
    this.recListeners.add( listener );
  }

  public void receivePacket( Packet packet )
  {
    for ( ReceiveListener listener : this.recListeners )
      listener.packetReceived( packet, this );
  }

  public String getName()
  {
    return this.name;
  }
}
```

Napokon, pogledajmo primjer kako se sve ovo koristi. Gore je prikazana klasa osluškivača koja ispisuje poruku na hrvatskom; u primjeru koji slijedi pojavit će se i klasa koja ispisuje poruku na engleskom. Cilj je prikazati situaciju s više od jednog osluškivača.

Ispis 9.11: Korištenje sustava

```
  public static void main( String[] args )
  {
    Node node1 = new Node( "Node1" );
    node1.addReceiveListener( new SysoutReceiveListenerEn() );
    node1.addReceiveListener( new SysoutReceiveListenerHr() );

    System.out.println( "Šaljemo paket čvoru..." );
    node1.receivePacket( new Packet( "Testing Testing 1-2-3" ) );
  }
```

Ispis:

```
Šaljemo paket čvoru...
Node Node1 received packet: Testing Testing 1-2-3
Čvor Node1 je primio paket: Testing Testing 1-2-3
```

9.3 *Adapter*

S primjerima uzoraka *Adapter* i *Decorator* susreli smo se u poglavlju o komunikaciji s okolinom, točnije u odjeljku 8.2 o pisačima i čitačima. Podsjetimo se situacije koju smo imali. U ispisu 8.4 imamo metodu `writeOutput( OutputStream )`. Njen parametar, objekt vrste `OutputStream`, predstavlja odlazni tok podataka. To je je tzv. "sirovi" tok podataka jer se preko njega šalju čisti okteti, a ne npr. neki apstraktniji podaci kao što su znakovi. Međutim, nama u implementaciji te metode upravo treba tok podataka u koji se mogu slati znakovi. To je zato što moramo pozvati metodu iz ispisa 8.3, `writeOutput( Writer )`, koja očekuje da ju se pozove s *pisačem*, što je Javin izraz za odlazni *znakovni* tok podataka.

U praksi ćemo tipično imati već gotove komponente koje rade sa znakovnim tokovima i ne znaju raditi drugačije. Dakle, zamislimo da je metoda `writeOutput( Writer )` ugrađena u neku klasu za koju nemamo izvorni kod jer je ona dio biblioteke klasa koju samo koristimo, a ne razvijamo. Dovedeni smo u situaciju koja se često u praksi, u analogiji s elektrotehnikom, naziva *impedance mismatch* ("neprilagođenost impedancija"). S jedne strane imamo metodu koja očekuje znakovni tok, a s druge strane imamo sirovi tok u koji ta metoda treba slati podatke. Problem ćemo riješiti primjenom uzorka *Adapter* – on će nam *adaptirati*, tj. *prilagoditi* sučelje objekta koji imamo (sirovi tok) na sučelje objekta koji očekuje metoda iz biblioteke (znakovni tok). Moramo napraviti klasu adaptera sa sirovog na znakovni tok. U primjeru s ispisa 8.4 koristi se Javin adapter, klasa `OutputStreamWriter`. Pogledajmo kako bi ona mogla biti realizirana:

Ispis 9.12: Pokazna izvedba adaptera na znakovni tok

```
package hr.fer.tel.ilj;

import java.io.IOException;
import java.io.OutputStream;
import java.io.Writer;

public class OutputStreamWriter
extends Writer
{
  private OutputStream os;

  public OutputStreamWriter( OutputStream os )
  {
    this.os = os;
```

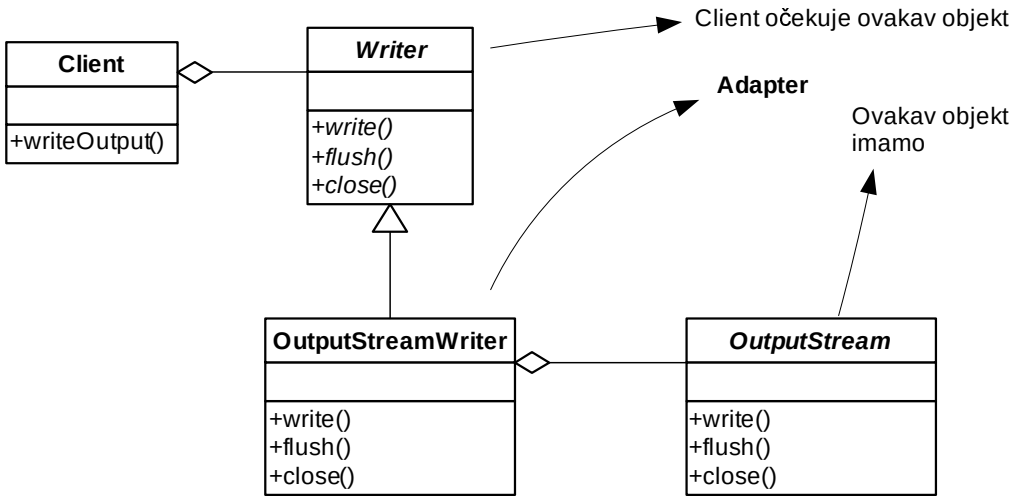
```
  }

  @Override
  public void write( char[] cbuf, int off, int len )
  throws IOException
  {
    for ( int i = off; i < off+len; i++ )
    {
      char c = cbuf[i];
      if ( c < 0x80 )
        os.write( c );
      else
        os.write( '#' );
    }
  }

  @Override
  public void flush()
  throws IOException
  {
    os.flush();
  }

  @Override
  public void close()
  throws IOException
  {
    os.close();
  }
}
```

Ova jednostavna izvedba u sirovi tok propušta samo znakove iz podskupa *Unicode*-a zvanog *Basic Latin*, koji odgovara ASCII-ju, a umjesto svih ostalih znakova šalje znak `#`. Primijetimo da su nadjačane samo tri metode, dok klasa `Writer` posjeduje znatno veći broj njih. To je zahvaljujući tome što su sve ostale metode u toj klasi pažljivo izvedene tako da se oslanjaju na osnovnu funkcionalnost koju pružaju samo ove tri. Naš adapter možemo isprobati na primjerima iz 8. poglavlja tako da u naš paket dodamo klasu adaptera, a izbacimo izjavu `import java.io.OutputStreamWriter`. Alternativno, možemo koristiti ovakvu metodu `main`, koju možemo ubaciti u klasu `OutputStreamWriter`:

```
  public static void main( String[] args )
  throws IOException
  {
    Writer osw = new OutputStreamWriter( System.out );
    osw.write( "Probni ispis\n" );
    osw.close();
  }
```

U drugom retku poziva se metoda `write( String )`, koju nismo eksplicitno nadjačali, ali koja se oslanja na našu metodu `write( char[], int, int )`[2].

Uzorak *Adapter* možemo dosta pregledno vizualizirati UML-om. Naša situacija prikazana je na slici 9.1.



Slika 9.1: UML-ov dijagram uzorka *Adapter*

9.4 *Decorator*

Osim što često imamo situaciju koju rješavamo `Adapter`-om, dakle nesuglasje između sučelja koje nam treba i sučelja koje imamo na raspolaganju, često se pojavljuje i slična situacija u kojoj imamo objekt čije sučelje odgovara, ali mu nedostaje još neka dodatna funkcionalnost. Dakle, nisu nam potrebne drugačije metode (kao u slučaju uzorka *Adapter*), već samo izmijenjeno ponašanje postojećih metoda. Za takve situacije primjenjujemo uzorak *Decorator*. Iz rečenog već možemo naslutiti da je ovaj uzorak dosta blizak uzorku *Adapter* – dapače, njegova shema prikazana UML-om je identična.

U poglavlju o komunikaciji imamo situaciju na kojoj možemo prikazati ovaj uzorak. Tamo se pojavljuje klasa `Reader` koja predstavlja neki dolazni znakovni tok. Problem koji sad želimo riješiti je sljedeći: već raspolažemo

[2]Metoda `write( String )` je stoga još jedan primjer primjene uzorka *Template method*.

nekim objektom čitača, a želimo mu dodati funkciju korisnu prilikom traženja pogrešaka u programu – da sve što pročita iz dolazne struje ispiše i na zaslon radi kontrole. Naglasimo: ne kaže se da raspolažemo *klasom* čitača, čiji kod bismo mogli izmijeniti izravno, nego njenom *instancom*, dok samim izvornim kodom klase *ne rapolažemo*.

Kako ćemo riješiti ovaj problem? Definirat ćemo novu klasu izvedenu iz klase `Reader`. Klasa, kao i kod *Adapter*-a, ima karakter *omotača*. U ovom konkretnom slučaju instanca naše nove klase omotač je za neki objekt vrste `Reader`. Dakle, u uzorku *Decorator* (hrv. *ukrašivač*) klasa koju "ukrašavamo" pojavljuje se u dva konteksta odjednom: i kao vrsta ugrađenog objekta (koji biva "ukrašen"), i kao nadklasa "ukrašivača". Najbolje je da proučimo implementaciju našeg ukrašivača:

Ispis 9.13: Primjer primjene uzorka *Decorator*

```
package hr.fer.tel.ilj;

import java.io.IOException;
import java.io.Reader;

public class DebugReader
extends Reader
{
  private Reader rdr;

  public DebugReader( Reader rdr )
  {
    this.rdr = rdr;
  }

  @Override
  public int read( char[] cbuf, int off, int len )
  throws IOException
  {
    // prvo proslijeđujemo poziv na omotani objekt:
    int charsRead = this.rdr.read( cbuf, off, len );
    // zatim obavljamo dodatne radnje:
    if ( charsRead != -1 )
      System.out.println( "Reading " +
        String.valueOf( cbuf, off, charsRead ) );
    return charsRead;
  }

  @Override
  public void close()
  throws IOException
  {
    // samo delegiramo na omotani objekt:
```

```
    this.rdr.close();
  }
}
```

Odmah možemo uočiti veliku sličnost s primjerom za uzorak *Adapter*. Kod *Adapter*-a u igri su dvije različite klase jer jednu prilagođavamo drugoj, a ovdje samo dodajemo funkcionalnost jednoj klasi, koja se stoga pojavljuje u dvostrukoj ulozi.

Našu klasu `DebugReader` možemo isprobati na primjerima iz 6. poglavlja, tako da prepravimo ovu metodu iz ispisa 8.4:

```
static void readInput( InputStream is )
throws IOException
{
  readInput( new InputStreamReader( is ) );
}
```

Treba samo ubaciti još jedan omotač, naš ukrašivač:

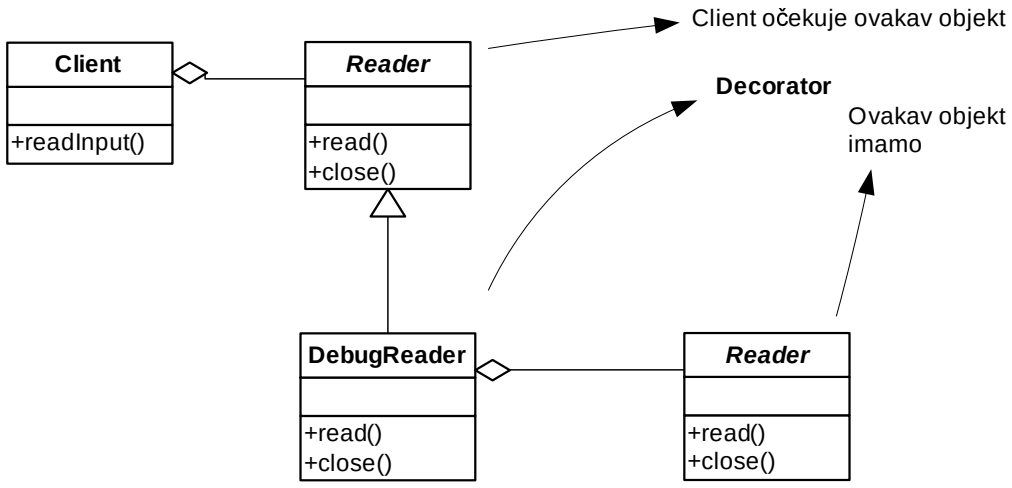
```
static void readInput( InputStream is )
throws IOException
{
  readInput( new DebugReader( new InputStreamReader( is ) ) );
}
```

Nakon toga možemo isprobati primjere iz odjeljaka 8.3–8.6.

Uzorak *Decorator*, kao što je najavljeno, u UML-ovom dijagramu izgleda vrlo slično kao *Adapter*. To se očituje na slici 9.2, na kojoj je jasno istaknuta dvojaka uloga ukrašavane klase[3].

Kao primjer korištenja uzorka *Decorator* u Javinoj biblioteci možemo izdvojiti klase koje se pojavljuju u poglavlju o komunikaciji s okolinom: `PrintWriter` i `BufferedReader`. Te klase su i posebno zanimljive jer one istovremeno implementiraju i *Adapter* i *Decorator*: npr. klasa `BufferedReader`, osim što dodaje funkcionalnost međuspremnika svim postojećim metodama (ukrašavanje), ima i dodatnu, vrlo značajnu metodu `readLine`, koje nema u običnom `Reader`-u. Iz perspektive nekog koda koji očekuje da će čitači imati upravo tu metodu, ona obavlja *prilagodbu* s običnog čitača na čitač s metodom `readLine`.

[3]Dijagram sa slike 9.2 nije sasvim korektan jer se ista klasa ne bi smjela crtati dvaput; međutim, ispravan crtež ne bi bio tako jasan.



Slika 9.2: UML-ov dijagram uzorka *Decorator*

Poglavlje 10

Savjeti za rad u *Eclipse*-u

10.1 Automatizacija rada na kodu

Razvojno okružje *Eclipse* ima dosta bogatu zbirku alata koji automatiziraju mnoge aktivnosti na razvoju koda. Njihovim korištenjem kod se kreira brže (puno manje tipkanja) i pouzdanije (*Eclipse* ne radi tipfelere). Preporuča se detaljno proučavanje menija Edit, Source i Refactor. Ovdje će se spomenuti nekoliko najvažnijih alata.

Quick fix aktivira se kombinacijom tipaka Ctrl-1 ili iz menija Edit. Pojavit će se mali prozorčić u blizini kursora i ponudit će različite mogućnosti prepravljanja koda, i dodatno, ako se kursor nalazi na mjestu gdje je otkrivena pogreška, standardne načine njenog otklanjanja. Treba imati na umu ovo: ***Quick fix* ne zna bolje od vas!** Student mora već znati kako treba ukloniti pogrešku, *Quick fix* je tu samo da taj postupak automatizira. Ako nasumice slušate "savjete" *Quick fix*-a, najvjerojatnije je da ćete napraviti kaos u svom kodu.

Rename in file je jedna od opcija koje nudi *Quick fix* kad smo s kursorom unutar nekog identifikatora. Opcija je dostupna i izravno pomoću Ctrl-2, R. Opciju koristimo uvijek kad želimo promijeniti ime nekoj lokalnoj varijabli, privatnom atributu/metodi i općenito bilo kojem identifikatoru koji se koristi samo u trenutnoj datoteci.

Content assist aktivira se pomoću Ctrl-Space ili iz menija Edit. Koristi se npr. na mjestu gdje želimo otipkati ime metode koju pozivamo. Ponudit će se sve metode definirane za objekt lijevo od točke. Npr. imamo varijablu `String s` i započinjemo novi redak koda tipkajući

```
s.
```

U ovom trenutku koristimo Ctrl-Space i uz kursor pojavljuje se popis svih metoda definiranih za `String`. Odaberemo pravu i njeno ime se ubacuje u kod. Ako je *Eclipse* odgovarajuće podešen, izbornik *Content Assist*-a pojavit će se i sam od sebe ako otipkamo točku i pričekomo koju sekundu (ovisno o brzini računala).

Content assist koristi se i za ekspanziju *kodnih predložaka*: npr. možemo natipkati `eq`, pritisnuti Ctrl-space i odabrati "equals". Time se ubacuje deklaracija metode `equals`. Isto radi i za `toString`, `hashCode`, `main`. Posebno koristan je predložak `sysout` koji ubacuje često korištenu a dugačku konstrukciju `System.out.println();` Također su vrlo korisni predlošci for – standardne petlje za prolazak po elementima polja ili kolekcije. Popis svih predložaka može se pregledati i editirati pod Window, Preferences, Java, Editor, Templates.

Organize imports aktivira se pomoću Ctrl-Shift-O ili iz menija Source. Kad god se pojavi pogreška da je neka vrsta podatka nepoznata, to je najvjerojatnije stoga što dotična vrsta nije uvezena. Ovom naredbom automatski se pronalaze i uvoze sve potrebne vrste. Ako postoji više vrsta istog imena, pojavit će se prozor u kojemu treba odabrati pravu. Ova komanda također i uklanja nepotrebne uvoze, koji se trenutno ne koriste u kodu.

Pomoću *Rename* možemo promijeniti ime klasi, paketu, javnoj metodi i općenito bilo kojem identifikatoru koji se koristi u više datoteka. Dostupno iz menija Refactor ili pomoću Alt-Shift-R.

Pomoću *Move* možemo klasu preseliti u drugi paket i metodu/atribut u drugu klasu. *Eclipse* će se pobrinuti da sva pozivanja na ono što se seli budu izmijenjena na odgovarajući način, odnosno upozorit će nas ako postoji prepreka preseljenju. Dostupno iz menija Refactor ili pomoću Alt-Shift-V.

10.2 Povijest datoteke

Ako shvatimo da smo krenuli s razvojem klase u pogrešnom smjeru, možemo se vratiti na ranije stanje datoteke. U Package Explorer-u kliknemo desnim gumbom na našu datoteku i odaberemo Replace with, Local history. Pojavit će se popis svih ranijih verzija datoteke koje su zapamćene.

10.3 Seljenje projekta s računala na računalo

10.3.1 Tipičan problem prilikom selidbe

Studenti se redovito susreću s potrebom da svoj projekt razvijaju na više računala – u labosu, na svom laptopu, na kućnom računalu, itd. Projekt je prilično jednostavno preseliti, ali još je jednostavnije to izvesti pogrešno, pogotovo stoga što je na raspolaganju čitav niz postupaka. Problemi nastaju zbog toga što postupak *izvoza* projekta nije usklađen s postupkom *uvoza*. Posebice, uočena je sljedeća **tipična pogreška**:

1. Projekt se ispravno izveze tako da glavno kazalo projekta sadrži kazalo `src` koje je korijensko za datoteke s izvornim kodom. Ono će tipično biti bez datoteka, samo s potkazalom `hr`.

2. Na odredišnom računalu projekt se uveze tako da se kazalo `src` ne proglašava korijenskim, već glavno kazalo projekta. Tako ispada da je `src` ustvari prvi segment imena paketa. *Eclipse* se sada žali da deklaracija paketa u datoteci (npr. `hr.fer.tel` – što je ispravno) ne odgovara kazalu u kojem se nalazi (`src/hr/fer/tel` – što je **neispravno!**).

3. Student odlazi na *Quick fix* koji mu nudi izmjenu imena paketa u `src.hr.fer.tel` i time malu pogrešku pretvara u puno veću. To ponavlja za svaku datoteku.

Ono što je zaista trebalo napraviti je popraviti korijenski direktorij izvornog koda. To se obavlja u Project, Properties, Java Build Path, Source. Tamo se nalazi okvir naslovljen "Source folders on build path" i unutra bi trebao biti navedeno kazalo `src`. Ako nije, dodajemo ga pomoću tipke Add folder...

10.3.2 Savjeti za uspješnu selidbu

Sve datoteke projekta nalaze se na jednom mjestu. Potrebno je samo znati koje kazalo *Eclipse* koristi kao svoj *Workspace*. To se može doznati putem File, Switch Workspace. U tom kazalu za svaki projekt postoji po jedno potkazalo istog imena kao projekt. Općenito je dovoljno kazalo projekta

preseliti na drugo računalo, ponovo u kazalo tamošnjeg *Workspace*-a, pokrenuti *Eclipse* i pomoću File, New, Project, Java Project zatražiti stvaranje projekta istog imena. *Eclipse* će vas obavijestiti (u dnu dijaloškog prozora) da kazalo tog imena već postoji i da će sve postavke projekta biti preuzete.

Eclipse nudi i izravnu podršku za selidbu projekta, korištenjem naredbi Export/Import. Na izvorišnom računalu radimo ovo:

1. U Package Explorer-u koristimo desni klik nad našim projektom, zatim Export, Archive file

2. Kad pritisnemo Next, nudi nam se mogućnost detaljnog odabira datoteka za izvoz. Datoteke `.classpath` i `.project` obvezno zadržimo (tu su postavke projekta). Ako je bitno da arhiva bude što manja (mejl i sl.), možemo izbaciti sva potkazala osim `src`.

3. Niže u dijaloškom prozoru pomoću Browse odaberemo točnu lokaciju i ime arhive koja će se stvoriti. Provjerimo da su odabrane opcije Save in zip format i Create directory structure for files.

4. Pritiskom na Finish izdajemo naredbu za stvaranje arhive.

Stvorenu arhivu dopremamo do odredišnog računala na kojem pokrećemo *Eclipse* i zatim:

1. File, Import, Existing projects into Workspace. U dijaloškom prozoru odabiremo opciju Select archive file i pomoću Browse nalazimo svoju arhivu.

2. U dijaloškom prozoru pojavljuje se popis svih projekata nađenih u arhivi (trebao bi biti samo jedan). Provjerimo da je ime projekta ispravno i označeno kvačicom.

3. Pritiskom na Finish izdajemo naredbu za uvoz projekta.

Literatura

[1] Java 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api.

[2] Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, 2001.

[3] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, 2005.

[6] James Gosling and Henry McGilton. The Java Language Environment, chapter 6: Security in Java. http://java.sun.com/docs/white/langenv/Security.doc.html.

[7] Andreas Mitschele-Thiel. *Systems Engineering with SDL: Developing Performance-Critical Communication*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[8] James Odell. Events and their specification. *JOOP*, 7(4):10–15, 1994.

[9] OMG. Unified modeling language. http://www.uml.org/.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[11] Alec Sharp and Patrick Mcdermott. *Workflow Modeling: Tools for Process Improvement and Application Development*. Artech House Publishers, 2001.