Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek

Priručnik programiranja u Javi

Tomislav Galba, Alfonzo Baumgartner, Mirko Köhler

Osijek, 2026.

Izdavač
Sveučilišste Josipa Jurja Strossmayera u Osijeku
Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek

Za izdavača
prof. dr. sc. Tomislav Matić

Autori
doc. dr. sc. Tomislav Galba
izv. prof. dr. sc. Alfonzo Baumgartner
izv. prof. dr. sc. Mirko Köhler

Recenzenti
izv. prof. dr. sc. Marko Čupić
Sveučilišste u Zagrebu
Fakultet elektrotehnike i računarstva
prof. dr. sc. Krešimir Nenadić
Sveučilišste Josipa Jurja Strossmayera u Osijeku
Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek

Lektor
doc. dr. sc. Dragana Božić Lenard
Sveučilišste Josipa Jurja Strossmayera u Osijeku
Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek

Veljača, 2026. godine

ISBN: 978-953-8184-05-5

Sveučilišste Josipa Jurja Strossmayera u Osijeku

Suglasnost za izdavanje ovog sveučilišnog udžbenika donio je Senat Sveučilište Josipa Jurja Strossmayera u Osijeku na 5. sjednici u akademskoj godini 2025./2026. održanoj 25. veljače 2026. godine pod brojem 4/26.

Sadržaj

Poglavlje 1

O Javi

Zamislite jedan programski jezik s kojim će se programirati svi elektronički uređaji koji u sebi sadrže mikroprocesore. Nešto kao:

"One Ring to rule them all, One Ring to find them, One Ring to bring them all, and in the darkness bind them."

Sada zamijenite riječ *Ring* s *Programming languages* i `darkness` s *light* i to je opis Jave.

1.1 Novi programski jezik

Što se dogodi kada jedan tim programera toliko isfrustrira programiranje u programskom jeziku C? Ako se radi o zaposlenicima tvrtke Sun Microsystems, oni odluče napraviti novi programski jezik. U prosincu 1990. skupina nezadovoljnih zaposlenika dobila je zadatak napraviti novi programski jezik, a njihov projekt nazvan je *Stealth Project*. No, kako su se i drugi inženjeri priključivali timu, shvatili su da bi promjena imena bila poželjna i preimenovali su ga u *Green Project*. Tim koji je radio na projektu (*Green Team*) brojao je 13 zaposlenika. Još je jedna ideja za ime bila C++ ++ −−. Da, dobro je napisano. Nastavak ++ i −− bilo je u smislu dodavanja dijelova koje nedostaju i uklanjanja dijelova koji su problematični u do tada korištenom programskom jeziku C++. No, na kraju su programski jezik nazvali *Oak*, po drvetu hrasta ispred njihova ureda. Sada možemo zaključiti da se programeri, koliko god bili stručni i kreativni u rješavanju problema stvaranja novog programskog jezika, ne mogu pohvaliti kreativnošću kada je u pitanju nazivlje. Kako je ime Oak već bilo registriran i zaštićen simbol, Gosling i njegov tim morali su odabrati drugo ime. Nakon jednog sastanka bombardiranja idejama (engl. *brainstorming*) došli su do imena Java. Ime Java odabrano je zbog svoje jedinstvenosti, a potječe od naziva otoka na kojem raste kava koja se pila u njihovom uredu.

1.2 Zahtjevi

Java je stvorena na principima poput objektno-orijentirana, sigurna, robusna, prijenosna, neovisna o platformi, visokih performansi, s ugrađenom podrškom za višedretvenost, itd.

Java je jednostavan jezik za naučiti. Broj stvari koje treba upamtiti vrlo je malen, nekih pedesetak ključnih riječi i sintaksu pisanja kôda koja je već viđena u drugim programskim jezicima. Autorima je bio cilj učiniti ju što sličnijom programskim jezicima koje većina programera već dobro poznaje (tj. C-u). O objektno-orijentiranom pristupu svakako ćemo reći puno više u sljedećem poglavlju. Programeri opisuju objekte i od njih stvaraju nacrte koji se nazivaju klase. Te su klase utemeljene na trima konceptima – enkapsulaciju, polimorfizmu i nasljeđivanju – u kojima leže znatne prednosti nad klasičnim pristupima poput proceduralnog ili funkcijskog programiranja.

Robusnost se u Javi očituje kako prevoditelj pronalazi programske greške, a memorijski sustav sam vodi brigu o oslobađanju i zauzimanju memorije. U Javi ne postoje pokazivači (pokazivači postoje samo u obliku referenci. Reference su pokazivači kojima ne možemo proizvoljno dodjeljivati vrijednosti niti vršiti s njima bilo kakvu aritmetiku) te je nemoguće da programer sam piše po nedozvoljenoj memorijskoj lokaciji. Na taj je način uklonjena mogućnost uništavanja nekih podataka. Java također ima sustav automatskog prikupljanja memorijskog smeća (engl. *garbage collection*) koji otklanja pojavljivanje memorijskih rupa.

Prenosivost aplikacija pisanih u Javi jedna je od najvažnijih prednosti Jave. Princip WORA (engl. *write once, run anywhere*) je moguć jer se prevedene aplikacije (odnosno njihov oktetni kôd) izvršavaju unutar javinog virtualnog stroja (engl. *Java Virtual Machine – JVM*). To možemo zamisliti kao prividni stroj ili računalo u računalu. To je okolina unutar koje se izvršavaju Java-aplikacije. Pomoću ovog sustava, Java kôd može se napisati na bilo kojem računalu. Taj kôd može se prevesti u oktetni kôd na istom tom računalu, a onda se taj oktetni kôd može pokrenuti na bilo kojem drugom računalu koje ima ispravno instaliran JVM. Najvažnije je da se pri tome Java-aplikacija ne mora niti najmanje prilagođavati. Nakon što je prevedena u oktetni kôd, ona se može distribuirati mrežom, a izvršni je sustav tijekom izvršavanja prevodi u specifične naredbe računala za koje je izvršni sustav napisan.

Kada programer napiše izvorni kôd u Javi, on ga može spremiti kao datoteku s nastavkom .java. Taj se izvorni kôd prevodi pomoću prevoditelja koji se naziva `javac` (čita se java-see). Kao rezultat dobijemo datoteku s nastavkom .class u kojoj je zapisan oktetni kôd. Taj se oktetni kôd može izvršiti na bilo kojem sustavu, bilo kojoj arhitekturi koja na sebi ima izvršno okruženje Jave (engl. *Java Runtime Environment – JRE*[1]). JRE predstavlja osnovni podskup Javine platforme koji korisnicima nudi mogućnost pokretanja prevedenih programa. JRE se sastoji od implementacije Javinog virtualnog stroja te obaveznih biblioteka čije se postojanje jamči programima. Ovo je minimum koji će svima omogućiti pokretanje izvršnih Java programa.

Sigurnost u nekom programskom jeziku odnosi se na štetu koju može izazvati program napisan u njemu i mogućnost izmjene oktetnog kôda nakon prevođenja. Java nudi neka od tih rješenja. JVM provjerava ispravnost oktetnog kôda tijekom izvršavanja. JVM provjerava oktetni kôd (iako ga je prevoditelj stvorio) kako bi zaista bio siguran u konzistentnost oktetnog kôda. Svaki sumnjivi oktetni kôd ili oktetni kôd koji bi sadržavao nedozvoljene radnje (direktan pristup memoriji, pogrešni parametri uz naredbe oktetnog kôda, nevažeće klase, ...) JVM mora zaustaviti prije izvršavanja. Dok razina zaštite leži u tome da JVM određuje memorijski raspored klasa tek nakon što se uvjeri u ispravnost oktetnog kôda. Na taj način ako netko želi izmijeniti prevedeni program u oktetnom kôdu, to ne može učiniti jer ne zna raspored klasa u memoriji.

Java ima biblioteke koje omogućavaju izradu programa koji će komunicirati uporabom protokola TCP, IP, FTP i HTTP. S njima imamo osnovne funkcije za rukovanje mrežnim protokolima prijenosnog ili aplikacijskog sloja, pa tako programi pisani u Javi mogu pristupati raznim bazama podataka i drugim poslužiteljima na Internetu.

Java je jedan od prvih programskih jezika koji podržava višenitno izvršavanje programa. Višenitnost (engl. *multithreading*) je svojstvo računalnog sustava u izvođenju programa ili njegovih dijelova (niti) istovremeno, a Java omogućava pisanje takvih programa.

Za neki programski jezik kažemo da ima svojstvo dinamičnosti ako se bez problema mogu stvarati novi objekti. Svaki program pisan u Javi, kada pokuša pristupiti nekom novom tipu podatka za koji ne zna kako ga obraditi, zatražit će od poslužitelja da mu pošalje klasu koja rukuje tim tipom podataka. Znači programi pisani u Javi sami će se dinamički nadograditi dobivenom klasom i obaviti željeni posao.

1.3 Verzije

Java je od svog predstavljanja pa do danas prošla više verzija. U početku su se verzije pojavljivale svakih 3 do 5 godina, ali od verzije 9, nova se verzija pojavljuje svakih 6 mjeseci. Tvrtka Oracle kupila je 20. 4. 2009. godine Sun Microsystems i s njom sva prava na programski jezik Java.

Prva verzija Jave imala je oznaku JDK 1.0 i objavljena je u studenom 1995. godine. Kratica JDK označava *Java Development Kit*. JDK predstavlja nadskup platforme JRE. JDK sadrži sve što je potrebno kako bi programer mogao pokrenuti prevođenje izvornog kôda Java programa u oktetni kôd te kako bi mogao pokrenuti izvođenje Java programa. To znači da JDK uključuje JRE te donosi još i implementaciju prevoditelja i drugih pomoćnih alata.

JDK pri tome ne sadrži okolinu za razvoj Java programa. Na stranicama tvrtke Oracle moguće je pronaći i paket koji sadrži, uz JDK, i okolinu za razvoj programa pod nazivom NetBeans. Osim navedene, danas je dostupno još puno drugih okolina, kako besplatnih tako i komercijalnih, od čega svakako treba naglasiti okolinu otvorenog kôda pod nazivom Eclipse.

Tablica 1.1: Verzije Jave

JDK 1.1	Izdvojene nove karakteristike

[1] Danas je teško doći do "čistog" JRE-a. Java se danas najčešće može dobiti otvorenom licencom kao kompletan JDK

Datum objave: 19. 2. 1997.	The idea of Inner Class; JavaBeans; JDBC; RMI; Reshaped AWT event model; JIT (Just in Time) compiler: Used on Microsoft Windows stages, developed for JavaSoft by Symantec; Internationalization and Unicode support beginning from Taligent
J2SE Version 1.2	
Datum objave: 8. 12. 1998.	Collections structure; Java String memory map for constants; JIT (Just in Time) compiler; Jar Signer for marking Java Archive (JAR) records; Policy Tool for allowing access to framework assets; Java Foundation Classes (JFC) which comprises Swing 1.0, Drag and Drop, and Java 2D class libraries; Java Plug-in; Scrollable result sets, BLOB, CLOB, user characterized types in JDBC; Audio help in Applets
J2SE Version 1.3	
Datum objave: 8. 5. 2000.	Java Sound; Jar Indexing; Huge list of advancements for improving the Java area.
J2SE Version 1.4	
Datum objave: 6. 2. 2002.	XML Processing; Java Print Service; Logging API; Java Web Start; JDBC 3.0 API; Assertions; API preferences; IPv6 Support; Regular Expressions; Image I/O API
J2SE Version 5.0	
Datum objave: 30. 9. 2004.	Generics; Enhanced for Loop; Autoboxing/Unboxing; Typesafe Enums; Static Import; Metadata (Annotations); Instrumentation
Java Version SE 6	
Datum objave: 11. 12. 2006.	Scripting Language Support; JDBC 4.0 API; Java Compiler API; Pluggable Annotations; Java GSS, Kerberos and LDAP support; Incorporated Web Services; Many more improvements
Java Version SE 7	
Datum objave: 7. 6. 2011.	Strings in switch Statement; Type Inference for Generic Instance Creation; Different Exception Handling; Backing for Dynamic Languages; Attempt with Resources; Java NIO Package; Binary Literals, underscore in literals; Null Handling
Java Version SE 8	
Datum objave: 18. 3. 2014.	Lambda Expressions; Pipelines and Streams; Date and Time API; Default Methods; Type Annotations; Nashhorn JavaScript Engine; Concurrent Accumulators; Parallel operations; TLS SNI
Java SE 9	
Datum objave: 21. 9. 2017.	Modularization of the JDK under Project Jigsaw; Given Money and Currency API; Reconciliation with JavaFX; Java usage of reactive streams; More Concurrency Updates; Provided Java Linker; Programmed scaling and measuring
Java SE 10	
Datum objave: 20. 3. 2018.	Local Variable Type Inference; Exploratory Java-Based JIT Compiler: This is the incorporation of the Graal dynamic compiler for the Linux x64 stage; Time-sensitive Release Versioning; Parallel Full GC for G1; Garbage collector Interface; Extra Unicode Language-Tag Extensions; Root Certificates; String Local Handshakes; Remove the Native-Header Generation Tool – java; Combine the JDK Forest into a Single Repository
Java SE 11	
Datum objave: 25. 9. 2018.	Dynamic class-file constants; Epsilon: a no-op garbage collector; Local-variable syntax for lambda parameters; Low-overhead heap profiling; HTTP client (standard); Transport Layer Security (TLS) 1.3; Flight recorder; ZGC: a scalable low-latency garbage collector (Experimental); JavaFX, Java EE and CORBA modules have been removed from JDK; Unicode 10.0.0 support
Java SE 12	

Datum objave: 12. 3. 2019.	Unicode 11.0.0 support; Shenandoah: A Low-Pause-Time Garbage Collector (Experimental); Microbenchmark Suite; Switch Expressions (Preview); JVM Constants API; One AArch64 Port, Not Two; Default CDS Archives; Abortable Mixed Collections for G1; Promptly Return Unused Committed Memory from G1
Java SE 13	
Datum objave: 17. 9. 2019.	Dynamic CDS Archives; ZGC: Uncommit Unused Memory; Reimplement the Legacy Socket API; Switch Expressions (Preview); Text Blocks (Preview)
Java SE 14	
Datum objave: 17. 3. 2020.	Pattern Matching for instanceof (Preview); Packaging Tool (Incubator); NUMA-Aware Memory Allocation for G1; JFR Event Streaming; Non-Volatile Mapped Byte Buffers; Helpful NullPointerExceptions; Records (Preview); Switch Expressions (Standard); Deprecate the Solaris and SPARC Ports; Remove the Concurrent Mark Sweep (CMS) Garbage Collector; ZGC on macOS; ZGC on Windows; Deprecate the ParallelScavenge + SerialOld GC Combination; Remove the Pack200 Tools and API; Text Blocks (Second Preview); Foreign-Memory Access API (Incubator)

* Paketi, dodaci i ostale karakteristike verzija nisu prevođene na hrvatski jezik.

Osim navedenih mogućnosti svaka je nova verzija uklanjala probleme iz prethodne i povećavala sigurnost razvijenih aplikacija.

1.4 Java danas

Koliko se često Java koristi? Ako pogledamo različite izvore na Internetu, možemo dobiti poprilično zbunjujuće rezultate o broju ljudi koji koristi Javu. No ako pogledamo koliko je ljudi zaposleno da "programira u Javi", možemo vidjeti da je Java sigurno jedan od najčešće korištenih programskih jezika.

Na stranicama tvrtke Oracle o Javi pišu ovako:

"Java is the #1 programming language and development platform. It reduces costs, shortens development timeframes, drives innovation, and improves application services. With millions of developers running more than 45 billion Java Virtual Machines worldwide, Java continues to be the development platform of choice for enterprises and developers.
Java is the #1 language for DevOps, AI, VR, Big Data, Continuous Integration, Analytics, Mobile, Chatbots, and Social."

Poglavlje 2

Uvod u objektno-orijentirano programiranje u jeziku Java

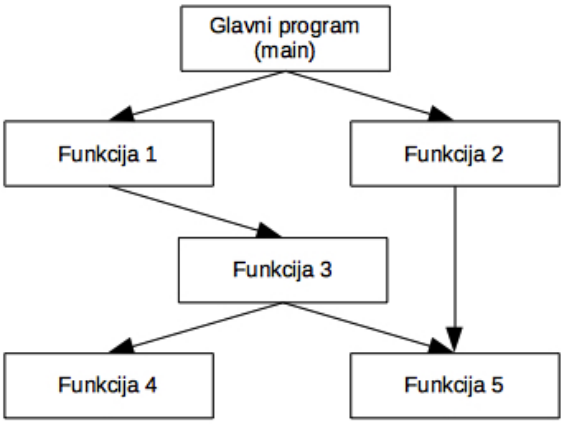
2.1 Pristupi programiranju

Dva su najpoznatija pristupa programiranju:

- proceduralno programiranje;
- objektno-orijentirano programiranje;

2.2 Proceduralno programiranje

Proceduralno programiranje predstavlja stil programiranja koji se zasniva na izvršavanju skupa poredanih operacija jednu iza druge. Proceduralni program definira varijable nad kojima se pozivaju procedure ili funkcije (skupine pojedinačnih operacija grupiranih u logičke cjeline) za unos, izlaz te manipulaciju vrijednosti koje su spremljene na tim mjestima. Proceduralni program često se sastoji od nekoliko stotina definiranih varijabli i procedura.

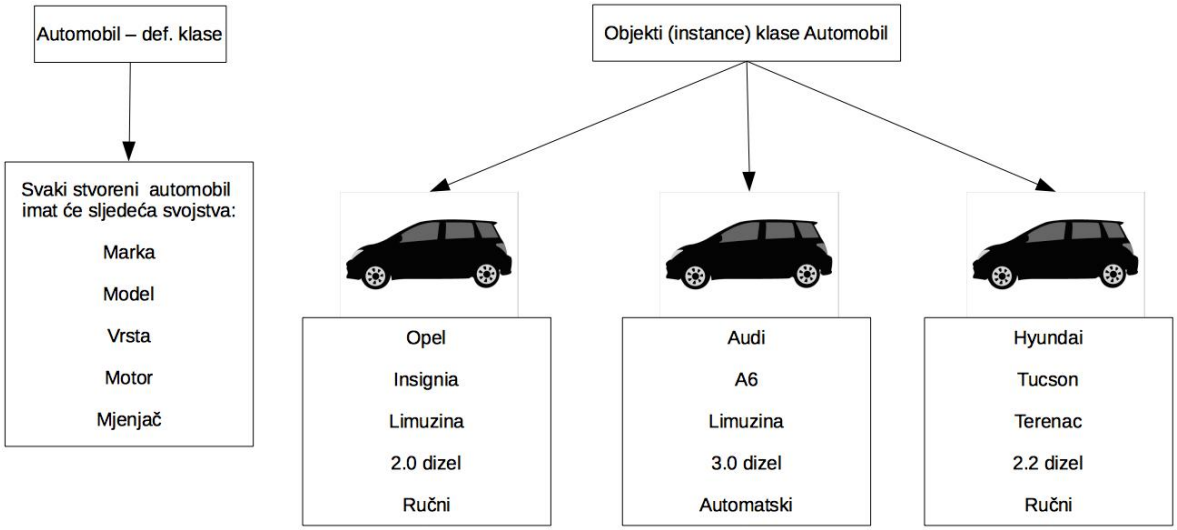


Slika 2.1: Proceduralni pristup.

2.3 Objektno-orijentirano programiranje

Objektno-orijentirano programiranje predstavlja nastavak na proceduralno programiranje uvođenjem drugačijeg pristupa pisanja programskog kôda. Ovaj način programiranja predstavlja pokušaj približavanja programa ljudskom načinu razmišljanja. Evolucija objektne paradigme kroz povijest dogodila

se preko starijih pristupa programiranja, a to su proceduralno i modularno[1] programiranje. Pisanje objektno-orijentiranog programa podrazumijeva definiranje klasa (razreda) te stvaranje objekata na temelju tih klasa u svrhu razvijanja okvira koji će omogućiti brži, točniji i ekonomski isplativ razvoj svakog budućeg programa. Termin ***klasa*** (ili razred) opisuje skupinu objekata koji imaju zajednička svojstva. ***Definicija klase*** opisuje atribute unutar objekata te metode koje opisuju što ti objekti mogu napraviti. ***Objekt*** u sebi sadrži vrijednosti atributa, a skup tih vrijednosti zovemo i stanje objekta. ***Atributi*** definiraju objekt, odnosno vrijednosti sadržane u atributima razlikuju jedan objekt od drugog. ***Metoda*** predstavlja blok programskog kôda (slično proceduri kod proceduralnog programiranja) koja omogućava izvršavanje nekog zadatka.

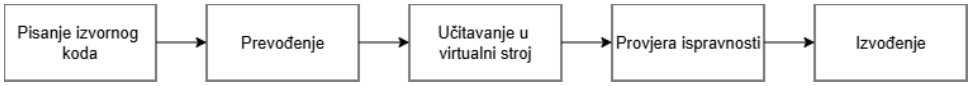


Slika 2.2: Primjer objektno-orijentiranog pristupa iz stvarnog svijeta.

Kako bi shvatili razlike objektno-orijentiranog programiranja u odnosu na proceduralno programiranje potrebno je razumjeti sljedeća tri koncepta:

- Enkapsulacija - mogućnost ograničavanja izravnog pristupa elementima neke klase. Drugim riječima, objekti mogu međusobno komunicirati dok implementacijski detalji pojedinog objekta ostaju sakriveni.
- Nasljeđivanje - mogućnost iskorištavanja elemenata (metoda, atributa) neke klase u nasljeđenoj klasi (podklasa).
- Polimorfizam - mogućnost poprimanja više oblika ili nadjačavanja metoda preuzetih iz nadklase. Drugim riječima, omogućava se stvaranje metoda istog imena unutar jedne klase pri čemu su ulazni argumenti definirani drugačijim tipovima podataka.

Može se reći da postoji pet koraka u stvaranju i izvođenju Java-aplikacije.

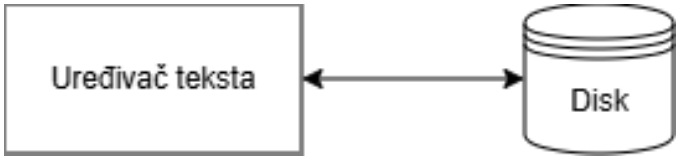


Slika 2.3: Koraci u razvoju Java-aplikacije.

2.4 Stvaranje programa

Prva faza odnosi se na stvaranje programskog kôda i odvija se u nekom od dostupnih okruženja. Kôd napisan u nekom uređivaču teksta naziva se izvorni kod (engl. *source code*) i sprema se na tvrdi disk u datoteku s nastavkom ".java".

[1]Način programiranja gdje se pojedini programski dijelovi razdvajaju u smislene cjeline - module

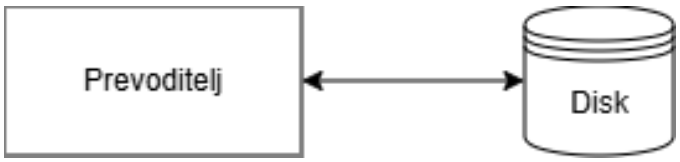


Slika 2.4: Korak 1 – pisanje programskog kôda.

Kao uređivač teksta programskog kôda moguće je koristiti vrlo jednostavne alate kao što su Notepad na operacijskom sustavu Microsoft Windows, TextEdit na operacijskom sustavu MacOS ili VIM na operacijskom sustavu Linux ili naprednija programska okruženja (engl. *Integrated developement environments – IDE*) kao što su Eclipse, Netbeans ili IntelliJ IDEA koji su dostupni za sve gore navedene operacijske sustave.

2.4.1 Prevođenje izvornog kôda u oktetni kôd

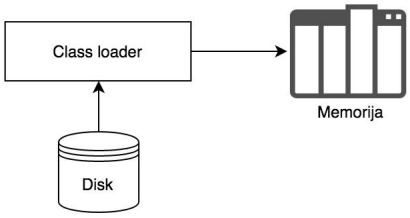
Java prevoditelj ima zadaću prevesti Java izvorni kôd u binarni program (oktetni kôd ili byte-code) koji predstavlja skup instrukcija koje će se izvršavati na JVM-u u zadnjem koraku. Oktetni kôd sprema se na disku u obliku datoteke s nastavkom (ekstenzijom) .class.



Slika 2.5: Korak 2 – prevođenje programskog kôda.

2.4.2 Učitavanje programa u memoriju

U trećem koraku podsustav virtualnog stroja zadužen za učitavanje klasa (engl. *class loader*) uzima datoteku s nastavkom .class koja u sebi sadrži oktetni kôd te ju prenosi u primarnu memoriju.



Slika 2.6: Korak 3 – prijenos oktetnog kôda.

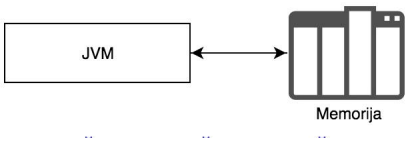
2.4.3 Provjera ispravnosti oktetnog kôda

U četvrtom koraku provjerava se ispravnost oktetnog kôda kako bi se osigurala ispravnost kao i da kôd prati sigurnosne smjernice.

Slika 2.7: Korak 4 – verifikacija.

2.4.4 Izvršavanje

U petom koraku JVM izvršava oktetni kôd koristeći kombinaciju interpretacije i prevođenja na zahtjev (engl. *Just-in-Time, JIT*). U tom procesu, JVM analizira oktetni kôd na način da traži najčešće korištene dijelove oktetnog kôda gdje onda JIT, kao što je Oracle HotSpot prevoditelj, prevodi oktetni kôd u strojni jezik, što omogućava brže izvršavanje kôda. Drugim riječima, može se reći da Java programi prolaze kroz dvije faze prevođenja. U prvoj se izvorni kôd prevodi u oktetni kôd kako bi se osigurala portabilnost dok se u drugoj, tijekom izvođenja, oktetni kôd prevodi u strojni jezik za trenutno računalo na kojem se program izvodi.



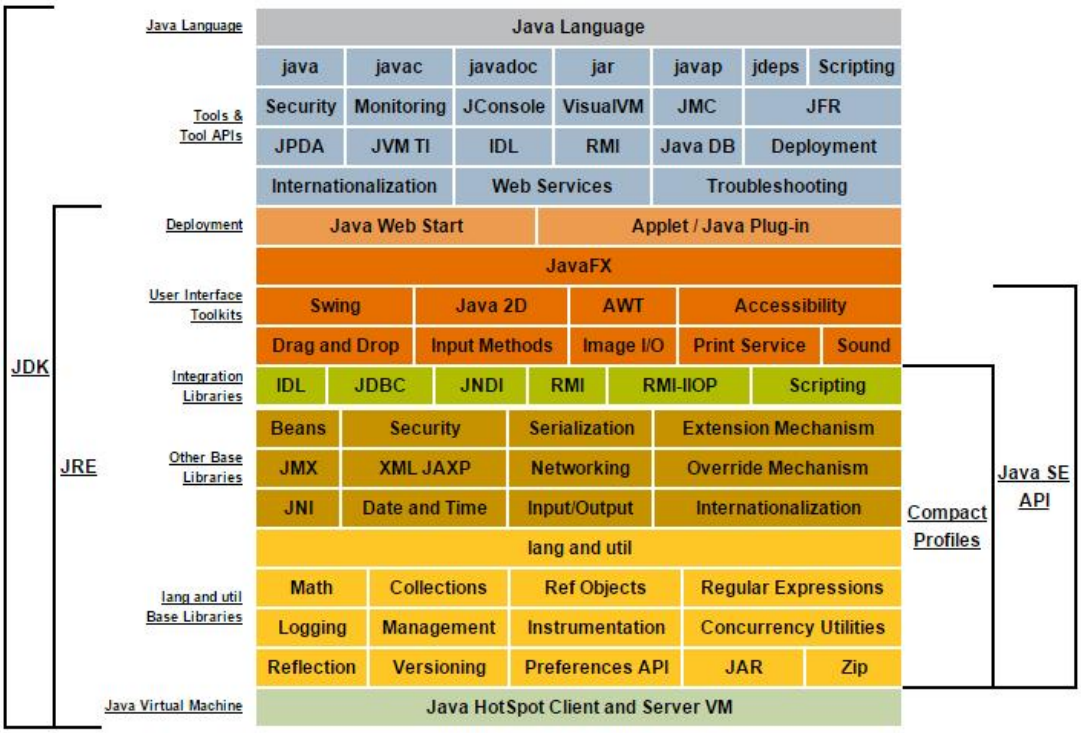
Slika 2.8: Korak 5 – izvođenje programskog kôda.

2.5 Zahtjevi za instalacijom Jave

Kako bi uspješno pisali aplikacije u programskom jeziku Java (neovisno o razvojnom okruženju), potrebno je instalirati JDK.

JDK predstavlja programski paket (engl. *developement kit*) koji u sebi sadrži i JRE uz prevoditelj i alate kao što su JavaDoc i Java Debugger. Detalji pojedinih paketa vidljivi su na slici 2.9, a preuzeti se mogu sa sljedeće poveznice:
`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Preporuča se uvijek koristiti najnoviju verziju JDK-a i obrisati staru prije toga.



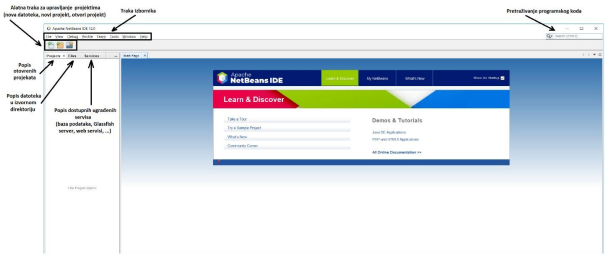
Slika 2.9: Opis Javinog konceptualnog dijagrama. Izvor: Oracle, "Jave SE Platform at a Glance", `https://www.oracle.com/java/technologies/platform-glance.html`, pristupljeno 3. 4. 2022.

2.5.1 Programsko okruženje za razvoj Java-aplikacija

Za potrebe izvođenja laboratorijskih vježbi koristit će se NetBeans. NetBeans predstavlja programsko okruženje razvijeno u Javi, a primarno je i namijenjeno za razvoj Java-aplikacija, dok podržava i razvoj u drugim jezicima kao što su PHP, C/C++, HTML5, JavaScript i drugi. Razvijeno je tako da bude platformski neovisno, što znači da ga je moguće koristiti na operacijskim sustavima Microsoft Windows, MacOS i Linux uz dodatak i nekih drugih operacijskih sustava koji podržavaju JVM.

Razvojno okruženje NetBeans može se preuzeti sa sljedeće poveznice:
`https://netbeans.apache.org/download/index.html`

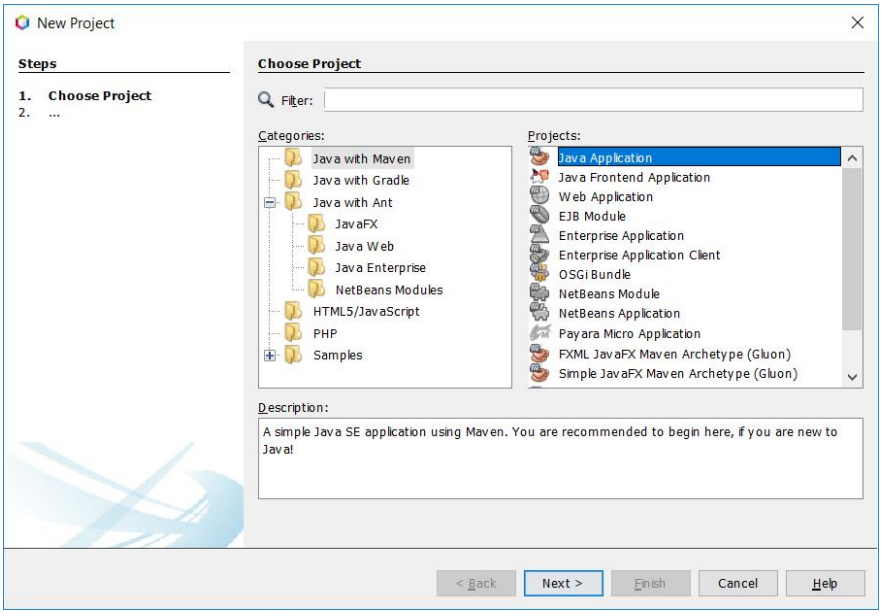
Pri prvom otvaranju sučelje će izgledati kao na slici 2.10.



Slika 2.10: Izgled programskog okruženja NetBeans.

Objasnit ćemo kako se stvara novi projekt u programskom jeziku Javi u okruženju NetBeans. Možete odabrati u traci izbornika `File >> New Project` ili odabrati drugu ikonu u alatnoj traci `New Project`.

Nakon toga otvori se prozor kao na slici 2.11.

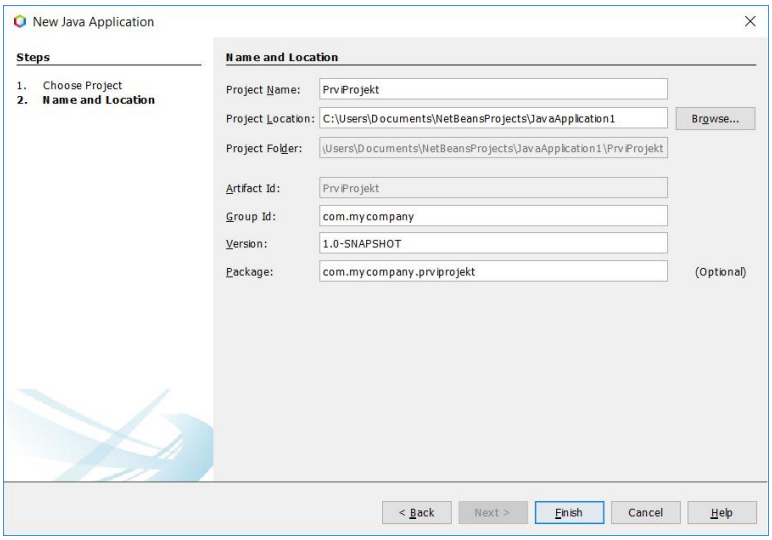


Slika 2.11: Odabir projekta za novu Java-aplikaciju.

Potrebno je izabrati jedan od triju alata za automatizaciju izrade Java-aplikacije. Za više informacija o svi trima alatima, pogledajte dokumentaciju na Internetu.

Za naš primjer izabrat ćemo `Java with Maven` pod kategorijom i `Java Application` pod tipom projekta i odabrati *dalje*.

Ako koristite neku stariju verziju okoline NetBeans ili neki drugi program, možda ćete u izboru imati samo
`File >> New Project >> Java >> Java application`.
Sljedeći je korak odabir imena i lokacije na koju se projekt sprema.



Slika 2.12: Imenovanje i adresa novog projekta.

Projekt smo nazvali PrviProjekt i vidimo da je po imenu projekta dobio ime i direktorij u kojem će projekt biti spremljen, oznaka predmeta (engl. *artifact*) kao i paket u projektu.

Svaki maven projekt kao svoj rezultat daje predmet koji je najčešće datoteka formata jar. Predmet može biti i model, biblioteka nekog operacijskog sustava, itd.

Odabirom gumba `Finish` priveli smo kraju kreiranje projekta u Javi i dobili smo prazan projekt.

2.5.2 Razvoj Java-aplikacije bez razvojnog okruženja

Java-aplikacije mogu se razvijati i bez razvojnog okruženja. Program možemo napisati u bilo kojem uređivaču teksta kao što su Notepad i TextEdit. Sljedeći primjer dovoljno je pretipkati u jedan od njih i spremiti ga kao PrviJavaProgram.java. Sada ćemo objaniti kako ga možete prevesti i izvršiti u konzoli (engl. *command prompt – cmd*).

```
public class PrviJavaProgram {
        public static void main(String[] args){
                System.out.println("Ovo je prvi Java program");
        }
}
```

Primjer kôda 2.1 Jednostavan program

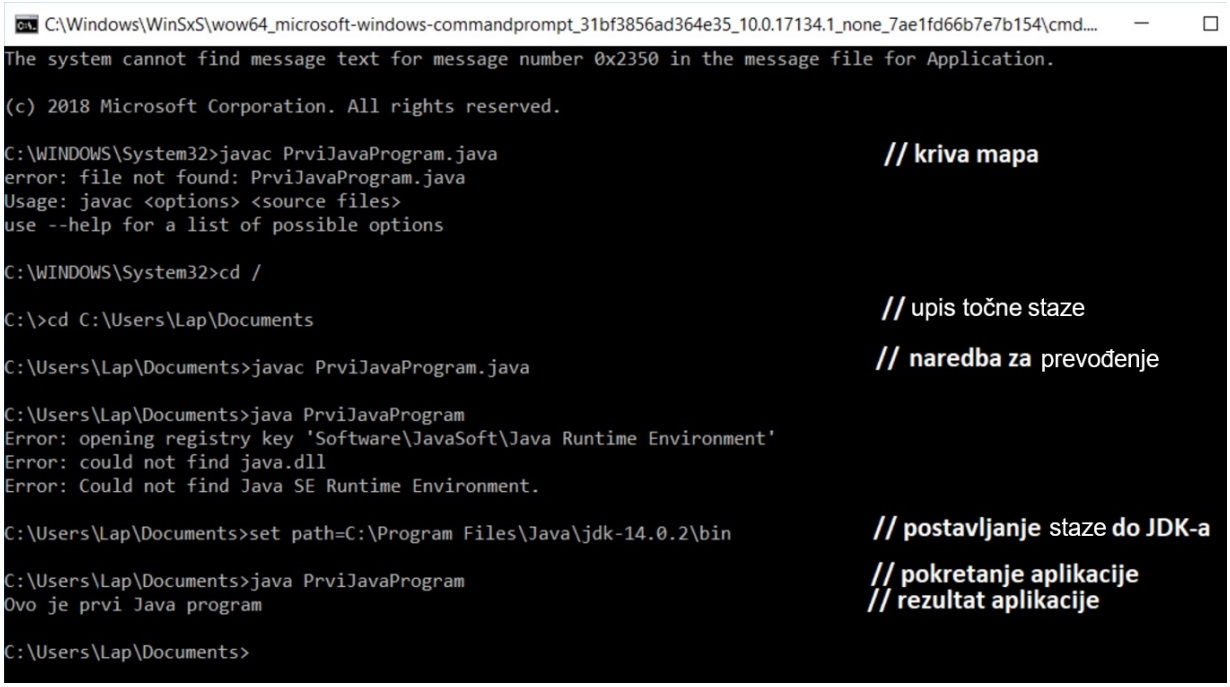
Može se primijetiti da se program i glavna klasa u programu imaju isto ime. To je obvezno kod pisanja programa u Javi. Sljedeći je korak prevođenje programa. Na računalima s operacijskim sustavom Microsoft Windows to se obavlja iz naredbenog retka (engl. *Command Prompt*).

U konzoli prvo treba ući u mapu gdje se nalazi datoteka s nastavkom .java, zatim treba pozvati Java prevoditelj `javac`.

```
javac PrviJavaProgram.java
```

Nakon što prevoditelj odradi svoj dio posla, u mapi se može primijetiti nova datoteku PrviJavaProgram.class. Nju pokrećemo iz konzole pomoću naredbe `java`.

```
java PrviJavaProgram
```



Slika 2.13: Izgled konzole u Windows OS prilikom prevođenja i pokretanja Java programa.

2.6 Tipovi podataka u Javi

Podatkove tipove u Javi možemo podijeliti u dvije kategorije: primitivne tipove te izvedene tipove.

2.6.1 Primitivni tipovi

Tablica 2.1: Primitivni tipovi podataka u Javi.

byte	-128 do 127 (zadana vrijednost 0)
short	-32,768 do 32,767 (zadana vrijednost 0)
int	-2,147,483,648 do 2,147,483,647 (zadana vrijednost 0)
long	-9,223,372,036,854,775,808 do 9,223,372,036,854,775,807 (zadana vrijednost 0L)
float	$-3.4 * 10^{38}$ do $3.4 * 10^{38}$ (zadana vrijednost 0f)
double	$-1.7 * 10^{308}$ do $1.7 * 10^{308}$ (zadana vrijednost 0d)
char	'\u0000' do '\uffff' (zadana vrijednost '\u0000')
boolean	true ili false (zadana vrijednost false)

2.6.2 Java-literali

Java-literali koriste se za izražavanje fiksnih vrijednosti, a mogu se primijeniti na bilo koji od primitivnih tipova npr.:

```
byte a = 22;
char ch = 'T';
```

String-literali navode se kao i u većini drugih programskih jezika koristeći dvostruke navodnike npr.:

```
"Hello world"
```

2.6.3 Posebni znakovi

Tablica 2.2: Popis posebnih znakova i njihovo značenje.

\n	Novi red
\r	Povratak na početak linije
\f	sljedeća stranica
\b	Backspace
\s	Razmak
\t	Tab
\"	Dvostruki navodnik
\'	Jednostruki navodnik
\\	Backslash

2.6.4 Reference i objekti

Reference i objekti stvaraju se koristeći neki od definiranih konstruktora neke klase. Na primjer:

```
Automobil bmw = new Automobil("X6").
```

U ovom primjeru `bmw` je lokalna varijabla koja se čuva na stogu i koja je po tipu podataka referenca. Međutim, također je i u memorijskoj gomili (eng. *heap*) stvoren novi objekt koji je primjerak klase `Automobil` te je i pozvan konstruktor koji će taj objekt popuniti s vrijednosti `"X6"`. Drugim riječima, obavlja se dinamičko zauzimanje memorije pri čemu se adresa dobivena dinamičkim zauzimanjem (operator `new`) zapisuje u lokalnu varijablu `bmw` koja se čuva na memorijskom stogu.

2.6.5 Pretvorba tipova podataka

Potreba za pretvorbom tipova podataka može se pojaviti u različitim situacijama (npr. kada dijelimo dvije varijable cjelobrojnog tipa, ali želimo dobiti decimalni rezultat) prilikom pisanja programskog kôda. U programskom jeziku Java postoje dva načina konverzije tipova podataka i svaki od njih ima i određena pravila:

- automatska pretvorba (implicitna pretvorba ili pretvorba proširivanjem);
- eksplicitna pretvorba (pretvorba skraćivanjem).

Automatska pretvorba tipova podataka može se obaviti u slučajevima kada su tipovi između kojih radimo pretvorbu međusobno kompatibilni, odnosno, udruživi po magnitudi vrijednosti. Isto tako, mora vrijediti da je opseg odredišnog tipa veći od opsega polaznog tipa (zbog čega se i zove pretvorba proširivanjem). Kao primjer možemo navesti pretvaranje tipa `byte` u tip `int`. Oba tipa podatka služe za čuvanje cjelobrojnih vrijednosti, što znači da su kompatibilni i u smislu tipa podatka kojeg čuvaju i u smislu opsega (`byte` može primiti opseg -128 do 127, dok `int` može primiti opseg -2147483648 do 2147483647).

Prema onome što smo naveli, za automatsku pretvorbu primitivnih tipova podataka moguće su sljedeće pretvorbe:

- `byte` → `short`, `char`, `int`, `long`, `float`, `double`;
- `short` → `int`, `long`, `float`, `double`;
- `char` → `int`, `long`, `float`, `double`;
- `int` → `long`, `float`, `double`;
- `long` → `float`, `double`;
- `float` → `double`.

Eksplicitna pretvorba tipova podataka obavlja se u situacijama kada su tipovi između kojih radimo pretvorbu međusobno nekompatibilni, odnosno, kada je opseg odredišnog tipa manji od opsega tipa iz kojeg radimo pretvorbu (zbog čega se i zove pretvorba skraćivanjem). Kao primjer možemo navesti pretvaranje tipa `double` u tip `int`. Navedeni podaci nisu kompatibilni po tipu podatka kojeg čuvaju niti im je opseg jednak pa će prilikom pretvorbe doći do gubitka dijela podataka (skraćivanja), što možemo vidjeti u primjeru kôda 2.2.

```
double d = 2E20;
int i = (int)d;
System.out.println(i);
```

Primjer kôda 2.2 Pretvorba skraćivanjem.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
Pretvorba tipa double u tip int:
i = 44
```

Primjeri koje smo naveli do sada odnose se na primitivne tipove podataka. Međutim, pretvorbu možemo raditi i nad objektima, a o tome će biti govora u nastavku.

2.7 Komentari

Za komentare koristimo znakove // ili /* */ za komentare koji se protežu kroz više linija.

```
//komentar u jednoj liniji
/* komentar koji
se proteze kroz nekoliko
linija */
```

U Javi postoji i posebna vrsta komentara

```
/** dokumentacija
*/
```

Za prevoditelj je ovaj komentar isti kao prethodni, no alat `javadoc` na temelju tako definiranih komentara izrađuje dokumentaciju.

2.8 Tipovi varijabli

Varijable nam omogućavaju imenovanje dijela memorije u koju spremamo podatke kojima upravlja naš program. U Javi svaka varijabla ima svoj tip i raspon vrijednosti koji može u sebe pohraniti.

Postoje tri tipa varijabli:

- Lokalna varijabla - varijabla koja je deklarirana unutar neke metode ili bloka naredbi zove se lokalna varijabla i može se koristit samo unutar te metode, odnosno tog bloka naredbi. Kada metoda (ili blok naredbi) završi s izvršavanjem, vrijednost te varijable je izgubljena.
- Varijaba objekta - varijabla koja se deklarira unutar klase, ali izvan bilo koje metode koja nije označena modifikatorom deklaracije `static`. Memorijski prostor za te varijable zauzima se prilikom stvaranja objekta koji je primjerak te klase. Svaki objekt sadrži odvojenu kopiju varijable u memoriji.
- Statička varijabla - u određenim slučajevima, svi primjerci jedne klase trebaju dijeliti određenu varijablu. U tim slučajevima koristi se statička varijabla. Deklaracija statičke varijable započinje korištenjem ključne riječi `static`.

```
public class Klasa1{
        private String varijabla1;
        private static String varijabla2;
        public void metoda1 (int varijabla3){
```

```
                int varijabla4;
                //tijelo metode
        }
}
```

Primjer kôda 2.3 Tipovi varijabli.

U primjeru kôda 2.3 mogu se vidjeti definicije sva tri tipa varijabli. Varijabla `varijabla4` predstavlja deklaraciju lokalne varijable, `varijabla1` predstavlja deklaraciju varijable objekta i `varijabla2` predstavlja deklaraciju statičke varijable. Varijabla `varijabla3` u ovom je slučaju argument ili parametar metode i ona se tretira kao lokalna varijabla unutar te metode.

2.9 Operatori

U programskom jeziku Java operatori se dijele u sljedeće grupe:

- aritmetički operatori;
- relacijski operatori;
- bitovni (engl. *bitwise*) operatori;
- logički operatori;
- operatori dodjeljivanja.

2.9.1 Aritmetički operatori

Tablica 2.3: Popis aritmetičkih operatora i njihovi primjeri.

Operator	**Primjer** **A = 10; B = 20;**
+ (zbrajanje)	A+B = 30
− (oduzimanje)	A-B = -10
∗ (množenje)	A∗B = 200
/ (dijeljenje)	B/A = 2
% (modul)	B%A = 0
++ (inkrement)	B++ = 21
−− (dekrement)	A−− = 9

2.9.2 Relacijski operatori

Tablica 2.4: Popis relacijskih operatora i njihovi primjeri.

Operator	**Primjer** **A = 1; B = 2;**	**Rezultat**
== (jednakost)	(A == B)	`false`
! = (nejednakost)	(A ! = B)	`true`
> (veće od)	(A > B)	`false`
< (manje od)	(A < B)	`true`
>= (veće ili jednako od)	(A >= B)	`false`
<= (manje ili jednako od)	(A <= B)	`true`

2.9.3 Bitovni operatori

Koriste se za vršenje operacija nad bitovima (bit po bit). Ako pretpostavimo da je vrijednost A=60 i B=13 u binarnom formatu i da su A i B tipa `byte`, to će biti zapisano kao:

```
A = 0011 1100
B = 0000 1101
```

Primjer korištenja I (&) i ILI (—) operatora prikazan je ispod:

```
A & B = 0000 1100
A | B = 0011 1101
```

Tablica 2.5: Popis bitovnih operatora i njihovi primjeri

Operator	Opis	Primjer
& (bitovni I)	Kopira bit u rezultat ako postoji u obama operandima.	(A & B) = 12 (0000 1100)
| (bitovni ILI)	Kopira bit ako postoji u jednom od operanada.	(A | B) = 61 (0011 1101)
^ (bitovni isključivo ILI)	Kopira bit ako postoji u jednom od operanada, ali ne u obama istovremeno.	(A ^ B) = 49 (0011 0001)
~ (bitovni komplement)	Okretanje bitova.	(~A) = -61 (1100 0011)
<< (aritmetički posmak u lijevo)	Vrijednost lijevog operanda pomjera se u lijevo za broj bitova određenih u desnom operandu.	A << 2 = 240 (1111 0000)
>> (aritmetički posmak u desno)	Vrijednost lijevog operanda pomjera se u desno za broj bitova određenih u desnom operandu.	A >> 2 = 15 (0000 1111)
>>> (logički posmak u desno)	Vrijednost lijevog operanda pomiče se u desno za broj bitova definiranih lijevim operandom.	A >>> 2 = 15 (0000 1111)

2.9.4 Logički operatori

Tablica 2.6: Popis logičkih operatora i njihovi primjeri.

Operator	Primjer **A = true; B = false;**
&& (logički I)	(A && B) = false
|| (logički ILI)	(A || B) = true
! (logičko NE)	!A = false

2.9.5 Operatori dodjeljivanja

Tablica 2.7: Popis operatora dodjeljivanja i njihovi primjeri.

Operator	Primjer
=	C = A+B (vrijednost A+B dodjeljuje C)
+=	C+=A (isto kao C = C+A)
−=	C−=A (isto kao C = C−A)
*=	C*=A (isto kao C = C*A)
/=	C/=A (isto kao C = C/A)
%=	C%=A (isto kao C = C%A)
<<=	C<<=2 (isto kao C = C<<2)
>>=	C>>=2 (isto kao C = C>>2)
>>>=	C>>>=2 (isto kao C = C>>>2)
&=	C&=2 (isto kao C = C&2)
^=	C^=2 (isto kao C = C^2)
|=	C|=2 (isto kao C = C|2)

2.9.6 Uvjetni operator ?:

Uvjetni operator predstavlja operator koji se sastoji od trija operanada i koristi se za evaluaciju i odluku koju vrijednost treba upisati u varijablu. Sintaksa je uvjetnog operatora kako slijedi:

```
varijabla v = izraz ? vrijednost ako je true : vrijednost ako je false
```

```
int a, b;
a = 5;
b = a = = 1 ? 10: 15;  // vrijednost b ce biti 15
b = a = = 5 ? 10: 15;  //vrijednost b ce biti 10
```

2.9.7 Operator `instanceof`

Operator `instanceof` predstavlja operator kojim se provjerava može li se na neki objekt gledati kroz referencu određenog tipa (gdje je tip reference neka klasa, sučelje, enum i slično). Sintaksa je operatora `instaceof` kako slijedi:

referenca instanceof tip

```
Integer i = Integer.valueOf(5);
boolean r1 = i instanceof Integer; //DA
boolean r2 = i instanceof Number; //DA
boolean r3 = i instanceof Object; //DA
boolean r4 = i instanceof Long; //NE
```

Primjer kôda 2.4 Korištenje operatora `instanceof`

2.10 Prva Java-aplikacija

Prije nego otvorimo novi projekt, upoznat ćemo se s unosom vrijednosti s tipkovnice i ispisa na zaslon.

Za unos s tipkovnice koristi se klasa `Scanner` koja je naprednija od starijeg načina koji koristi klasu `System` (`System.in`). Dio je paketa `java.util` za dohvaćanje korisničkog unosa primitivnih tipova kao što su `int`, `double` i drugi. Za stvaranje objekta tipa `Scanner` koristi se sljedeća linija kôda:

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

S lijeve strane operatora pridruživanja `Scanner input` deklarira se varijabla koja je referenca tipa `Scanner` i naziva `input`. S desne strane operatora pridruživanja `new Scanner(System.in)` stvara se objekt tipa `Scanner` koji je povezan s objektom na koji pokazuje `System.in`. Drugim riječima, stvoreni objekt tipa `Scanner` spojen je s prethodno definiranim ulaznim uređajem (tipkovnica).

Objekt tipa `Scanner` razdvaja ulazni podatak u dijelove koji se nazivaju tokeni. Tokeni se zatim mogu pretvoriti u vrijednosti različitih tipova korištenjem za to dostupne metode. U tablici 2.8 prikazane su neke od najčešće korištenih metoda gdje svaka od njih dohvaća vrijednost s tipkovnice u željenom tipu podatka.

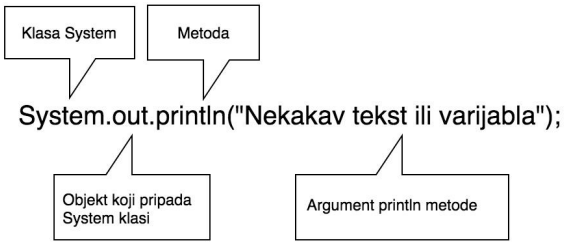
Tablica 2.8: Metode `Scanner` klase

Metoda	Opis
nextDouble()	Dohvaća ulaz kao `double`.
nextInt()	Dohvaća ulaz kao `int`.
nextLine()	Dohvaća sljedeću liniju i vraća ju kao `String`.
next()	Dohvaća kompletan token kao `String`.

```
String string;
int integer;
Scanner input = new Scanner(System.in);
        //za unos "studenti FERIT-a bit ce uspjesni programeri"
string = input.next(); // bit ce spremljeno "studenti"
string = input.nextLine();
        // bit ce spremljeno " FERIT-a bit ce uspjesni programeri"
integer = input.nextInt(); // zablokirat ce cekajuci unos broja
```

Primjer kôda 2.5 Korištenje klase tipa `Scanner`

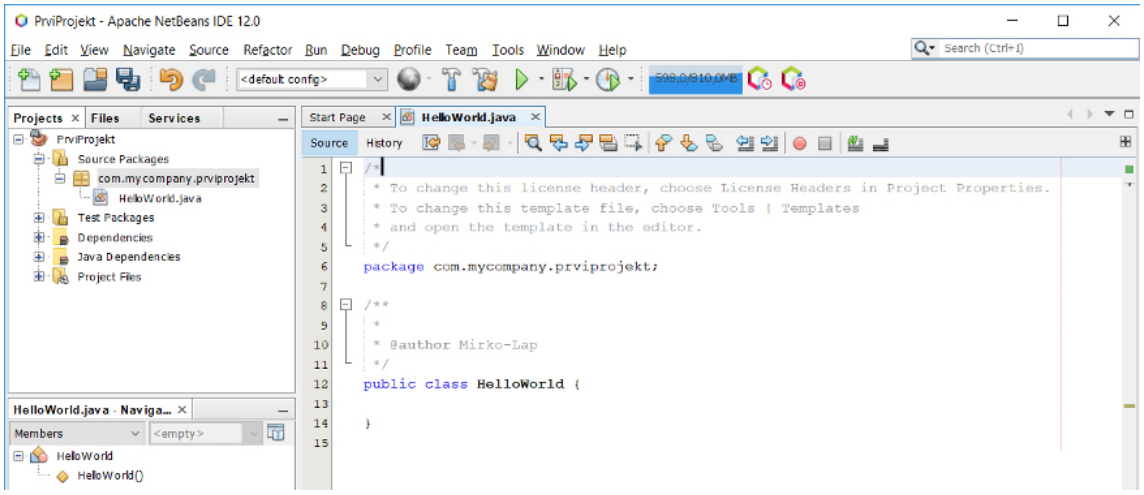
Ispis podataka na zaslon obavlja se pomoću klase tipa `System`.



Slika 2.14: Opis poziva klase za ispis na zaslon.

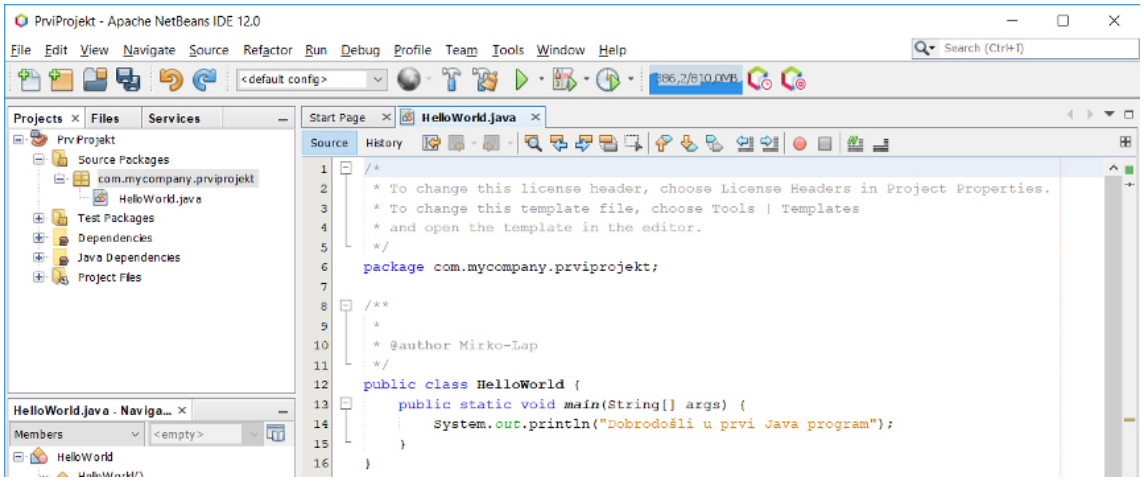
2.11 Prvi primjer "Hello world"

U razvojnom okruženju NetBeans kod popisa otvorenih projekata, potrebno je kliknuti desnom tipkom miša na paket koji nosi ime projekta. U našem slučaju to je `com.mycompany.prviprojekt`. Zatim odabrati `New >> Java Class`.
Klasi treba dati ime i odabirom gumba `Finish` priveli smo kraju kreiranje klase u Javi.



Slika 2.15: Primjer otvorene klase u programskom paketu NetBeans.

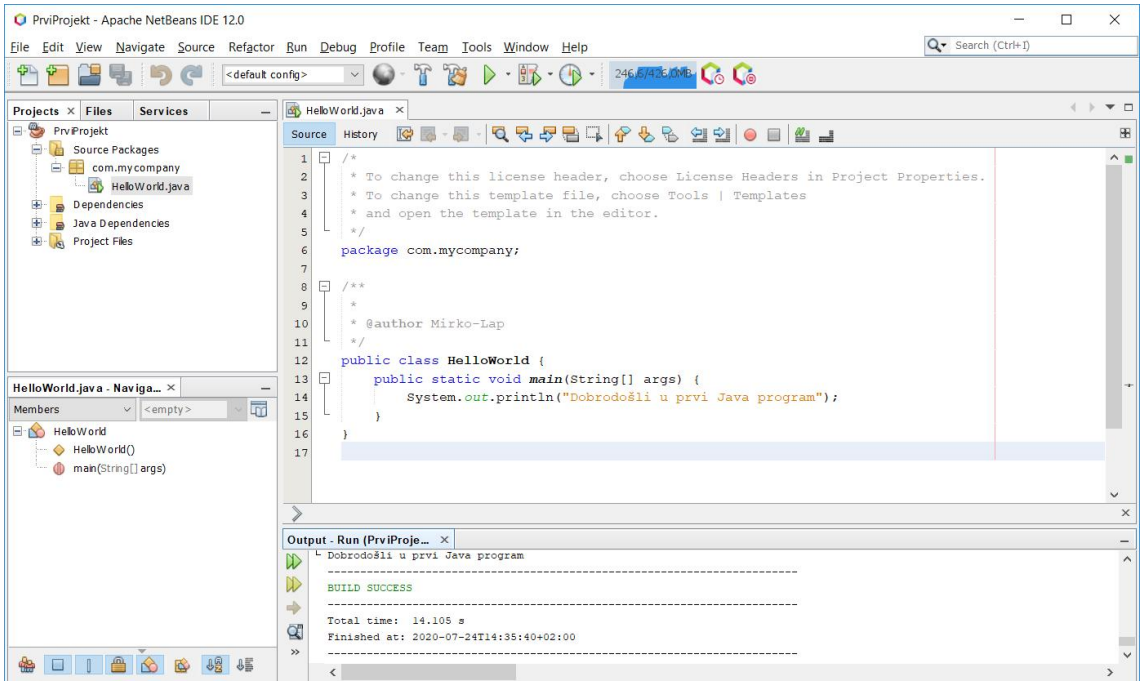
Prvo je potrebno unutar naše klase `HelloWorld` dodati glavnu metodu i u njoj napisati poziv na metode za ispis na zaslon.



Slika 2.16: Dodavanje glavne metode klasi `HelloWorld`.

Projekt možemo pokrenuti na više načina:

- klikom na tipku F6;
- `Run >> Run Project` ili
- klikom na ikonu za `Run Project` (zeleni trokut u alatnoj traci).



Slika 2.17: Izgled sučelja nakon pokretanja Java-aplikacije.

2.12 Zadaci

Zadatak 2.1. Napisati program koji ispisuje sljedeći tekst na zaslon.
Očekivani izlaz:

```
  J   a  v     v  a
  J  a a  v   v  a a
J J aaaaa  V V  aaaaa
 JJ a   a   V   a   a
```

Zadatak 2.2. Napisati program u Javi koji ispisuje tekst 'Hello' na zaslonu i nakon toga Vaše ime u novom redu.
Očekivani izlaz:

```
Hello
Ime Prezime
```

Zadatak 2.3. Napisati program koji ispisuje sumu dvaju brojeva.
Testni podaci: `74+36`
Očekivani izlaz: `110`

Zadatak 2.4. Napisati program koji cjelobrojno dijeli dva cijela broja i ispisuje rezultat na zaslon.
Testni podaci: `50/3`
Očekivani izlaz: `16`

Zadatak 2.5. Napisati program koji uzima dva broja (korisnik unosi) i ispisuje njihov umnožak na zaslon.
Testni podaci:
Unesite prvi broj: `25`
Unesite drugi broj: `5`

Očekivani izlaz: `25 x 5 = 125`

Zadatak 2.6. Napisati program koji korisniku omogućava unos dvaju brojeva s tipkovnice te ispisuje na zaslon njihovu sumu, razliku, umnožak, rezultat dijeljenja i ostatak pri dijeljenju.
Testni podaci:
Unesite prvi broj: `125`
Unesite drugi broj: `24`

Očekivani izlaz:

```
125 + 24 = 149
125 - 24 = 101
125 * 24 = 3000
125 / 24 = 5
125 mod 24 = 5
```

Poglavlje 3

Klase i objekti

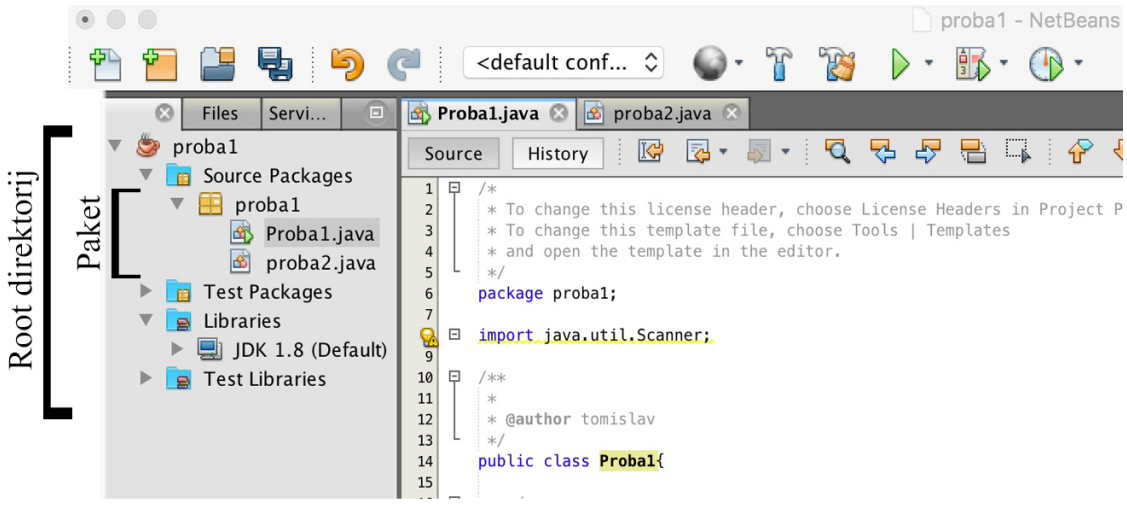
Klase predstavljaju temelj objektno-orijentiranog pristupa programiranju. Kada se razmišlja na objektno-orijentirani način, tada je sve objekt, a svaki je objekt primjerak neke klase. Primjerice, stol u radnoj sobi dio je klase koja sadrži sve stolove, zlatna ribica u akvariju dio je klase koja sadrži sve ribice za akvarij.

Objekt (primjerak ili instanca klase) predstavlja skup vrijednosti svojstava definiranih nekom klasom koja se mogu objediniti u smislenu cjelinu. Objekti međusobno mogu razmjenjivati informacije pri čemu okolina objekta, pa čak ni sami objekti, ne moraju ništa "znati" o unutrašnjoj strukturi pojedinog objekta.

Deklaracijom klase ne stvara se objekt. Ako zamislimo proizvodnju nekog proizvoda, kako bi ga proizveli potrebno je u potpunosti razumjeti sve karakteristike tog proizvoda. Sukladno tome, klasa je samo apstraktni opis kako će objekt izgledati ako se ikada instancira. Prema tome, ako se vratimo na primjer sa stolom i ribicom, stol u radnoj sobi bio bi instanca klase `Desk`, a zlatna ribica bila bi instanca klase `Fish`. Navedeni izrazi predstavljaju "is-a" relaciju u kojoj je objekt konkretan primjerak klase.

3.1 Paketi

Prije definiranja klasa i pripadajućih metoda i varijabli potrebno je upoznati se s mehanizmom koji ima zadatak grupiranja klasa u jedan modul koji se naziva paket (engl. *package*). Paketi mogu sadržavati i podpakete što onda tvori strukturu koja se naziva struktura paketa (engl. *package structure*). Kako bi stvorili paket u Javi prvo je potrebno stvoriti korjenski (engl. *root*) direktorij koji sam po sebi i nije dio paketa, već sadrži sve ostale datoteke kao što su datoteke u Javi koje će se kasnije smjestiti u određeni paket.

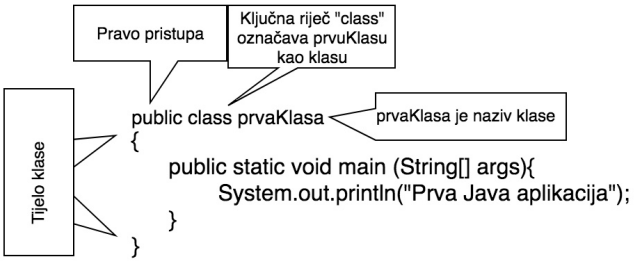


Slika 3.1: Prikaz paketa unutar korjenskog direktorija.

3.2 Definiranje klase

Kada definiramo klasu potrebno je dodijeliti ime klasi te odrediti koji će se podaci i metode nalaziti u klasi. Za početak, zaglavlje klase sastoji se od triju dijelova:

- pravo pristupa (engl. *access modifier*);
- ključna riječ `class`;
- naziv klase.



Slika 3.2: Opis definicije klase u Javi.

Deklariranjem prava pristupa definira se kojim elementima druge klase može pristupiti određena klasa. Mogu se deklarirati odvojeno za klasu, konstruktor, varijable i metode.

Klase, varijable, konstruktori i metode mogu imati dodijeljena četiri različita prava pristupa:

- privatni (ključna riječ `private`);
- privatni pristup unutar paketa;
- zaštićeni (ključna riječ `protected`);
- javni (ključna riječ `public`).

3.3 Privatni pristup

U slučaju dodjeljivanja privatnog (ključna riječ `private`) prava pristupa metodi ili varijabli, samo kôd unutar iste klase može pristupiti toj varijabli ili pozvati tu metodu. Vršni razredi mogu biti definirani isključivo kao javni ili privatni za paket; na sve ostale razrede nema takvih ograničenja (primjerice, mogu biti privatni).

```
public class Klasa{
        private long varijabla = 1;
}
```

Primjer kôda 3.1 Dodjeljivanje privatnog prava pristupa

3.4 Privatni pristup unutar paketa

Privatni pristup unutar paketa deklarira se na način da se uopće ne navodi nikakav identifikator. Dodjeljivanje ove vrste prava pristupa znači da kôd unutar klase, kao i kôd unutar klasa koje se nalaze u istom paketu, mogu pristupiti klasi, varijablama, konstruktoru ili metodama. Radi toga u literaturi se za ovaj tip prava pristupa može naći naziv i modifikator paketnog pristupa (engl. *package access modifier*). Podklase ne mogu pristupati metodama i varijablama nadklase ako su metode i varijable deklarirane kao privatne za paket, osim u slučajevima kada se i podklasa nalazi u istom paketu kao i nadklasa.

```
public class Klasa1{
        int varijabla_v = 1;
}
public class Klasa2{
        Klasa1 ob = new Klasa1();
        public long f(){
                return ob.varijabla_v;
        }
}
```

Primjer kôda 3.2 Dodjeljivanje privatnog prava pristupa unutar paketa

3.5 Zaštićeni pristup

Zaštićeni se pristup (ključna riječ `protected`) prema sigurnosti može smjestiti između privatnoga za paket i javnog. Pruža iste mogućnosti pristupa kao i privatni za paket uz dodatak da podklase mogu pristupiti zaštićenim metodama i varijablama nadklase čak i u slučaju kada se ne nalaze u istom paketu. To se odnosi na nasljeđivanje o kojem će se pričati više u nastavku.

```
public class Klasa1{
        protected long varijabla_v = 1;
}
public class Klasa2 extends Klasa1{
        public long f(){
                return this.varijabla_v;
        }
}
```

Primjer kôda 3.3 Dodjeljivanje zaštićenog prava pristupa

3.6 Javni pristup

Dodjeljivanje javnog pristupa (ključna riječ `public`) znači da kompletan kôd može pristupiti klasi, varijablama, konstruktoru ili metodama bez obzira na to gdje se nalazi. Drugim riječima, može se nalaziti u različitim klasama i različitim paketima.

```
public class Klasa1{
        public long varijabla_v = 1;
}
public class Klasa2{
        Klasa1 ob = new Klasa1();
        public long f(){
                return ob.varijabla_v;
        }
}
```

Primjer kôda 3.4 Dodjeljivanje javnog prava pristupa

Važno je napomenuti da prava pristupa dodijeljena klasama imaju prednost nad pravima koja su dodijeljena varijablama, konstruktorima ili metodama te klase. Drugim riječima, ako je klasi dodijeljen privatni pristup unutar paketa, tada niti jedna druga klasa, ukoliko se ne nalazi u istom paketu, neće moći pristupiti metodi te klase iako je možda postavljeno javno pravo pristupa na tu metodu.

3.7 Definiranje metoda

Metoda predstavlja dio kôda koji u sebi sadrži skup naredbi koje izvršavaju neki zadatak. Kako bi se neka metoda izvršila potrebno ju je pozvati unutar iste klase ili druge klase ili iz neke druge metode unutar iste ili druge klase. Svaka klasa može sadržavati neograničen broj metoda i svaka se metoda može

pozvati neograničeni broj puta. Također, neke metode mogu zahtijevati određene podatke prilikom njihova pozivanja. Ti podaci koji se šalju nazivaju se argumenti.

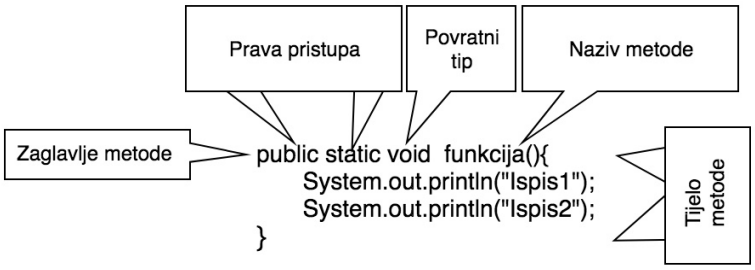
Metoda mora uključivati sljedeće:

- deklaracija – naziva se još zaglavlje (engl. *header*) ili definicija (engl. *definition*);
- otvorenu i zatvorenu vitičastu zagradu { };
- tijelo metode (između vitičastih zagrada).

Deklaracija metode sadrži sljedeće:

- pravo pristupa;
- povratni tip;
- naziv;
- argumente metode (ako postoje).

Važno je napomenuti da se za metodu može postaviti bilo koje od dostupnih prava pristupa (javno, zaštićeno, privatno i privatno unutar paketa) premda je najčešće korišteno javno pravo pristupa.



Slika 3.3: Opis definicije metode u Javi.

Na slici 3.3 može se vidjeti primjer jedne metode koja ne prihvaća niti jedan argument. Važno je istaknuti upotrebu ključne riječi `static`. Ona označava mogućnost pozivanja navedene metode bez instanciranja objekta klase u kojoj se metoda nalazi. Veliki se broj metoda tako piše pa tako u programskom jeziku Java imamo klasu `Math` koja u sebi ima sve metode deklarirane kao `static`. Zbog toga je, primjerice, za računanje korijena nekog broja potrebno napisati samo `Math.sqrt(broj)`. Metoda koja ne sadržava ključnu riječ `static` u svom se zaglavlju naziva metoda instance (engl. *instance method*) zbog toga što se može koristit tek nakon instanciranja objekta te klase.

```
public class OpsegKruga{
        public static void main(String [] args){
                double r = 15;
                double pi = 3.14;
                opsegK(r,pi);
        }
        public static void opsegK(double r, double pi){
                System.out.println(2*r*pi);
        }
}
```

Primjer kôda 3.5 Metoda s ulaznim argumentima

Metoda može primati i varijabilni broj argumenata kao što je to prikazano u primjeru kôda 3.6.

```
public class OpsegKruga{
        public static void main(String [] args){
                double r = 15;
                double pi = 3.14;
                opsegK(r,pi);
```

```
        }
        public static void opsegK(double ... argumenti){
                System.out.println(2*argrumenti[0]*argumenti[1]);
        }
}
```

Primjer kôda 3.6 Definicija i korištenje metode s varijabilnim brojem argumenata

Kako bi metoda primila varijabilan broj argumenata mora se slijediti sintaksa:

```
<tip argumenta> ... <naziv argumenta>
```

Tako proslijeđeni argumenti tretiraju se kao niz argumenata te im se pristupa na isti način, što je moguće vidjeti u primjeru kôda 3.6. Važno je napomenuti da se varijabilan broj argumenata može koristiti u kombinaciji s uobičajenim argumentima uz napomenu da metoda može primiti maksimalno jedan varijabilni argument i on uvijek mora stajati na kraju liste argumenata:

```
public void funkcija(int prvi, double drugi, int ... varijabilni)
```

Metoda završava na tri načina:

- kada se izvede posljednja naredba;
- kada se dogodi iznimka (engl. *exception* - više o iznimkama u nastavku);
- kada dođe do naredbe `return`. Naredba `return` uzrokuje prekidanje izvođenja metode i povratak na dio kôda gdje se metoda pozvala, pri čemu dosta često i vraća neku vrijednost kao rezultat izvršavanja.

```
public static double opsegK(double r, double pi) {
        return(2*r*pi);
}
```

Primjer kôda 3.7 Metoda s return naredbom.

Metode je moguće pozivati i unutar neke druge metode.

```
public static double povrsinaK(double r, double pi){
        return(r*r*pi);
}
public class OpsegKruga{
        public static void main(String [] args){
                double r = 15;
                double pi = 3.14;
                izracunajOpsegiPovrsinuKruga(r,pi);
        }
        public static void
          izracunajOpsegiPovrsinuKruga (double r, double pi){
                System.out.println("Opseg kruga je "
                 +(2*r*pi)+" , a povrsina je "+povrsinaK(r,pi));
        }
}
```

Primjer kôda 3.8 Poziv iz druge metode.

3.8 Prijenos argumenata po vrijednosti

S obzirom da je Java programski jezik nastao na temelju C/C++ jezika, možda bi bilo najbolje kada bismo se kratko osvrnuli na načine prosljeđivanja argumenata podržane u C/C++ jeziku. Kada pričamo o C/C++ jeziku, možemo reći da su podržana tri načina prosljeđivanja:

- po vrijednosti (engl. *pass by value*);

- po adresi (engl. *pass by address*);
- po referenci (engl. *pass by reference*) – nećemo dodatno objašnjavati jer nije potrebno za shvaćanje problematike.

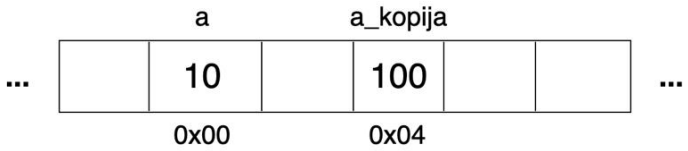
Na jednostavnom primjeru prikazat ćemo razlike između tih triju načina prosljeđivanja argumenata kako biste lakše mogli shvatiti na koji se način prosljeđuju argumenti u programskom jeziku Java.

Zamislite situaciju gdje deklarirate jednu varijablu i nju prosljeđujete u neku funkciju koja pokušava promijeniti vrijednost proslijeđene varijable:

```
void funkcija(int a){ a *= a; }
int main(){
        int a = 10;
        cout<<a; //ispisat ce 10
        funkcija(a);
        cout<<a; //ispisat ce 10
}
```

Primjer kôda 3.9 Prosljeđivanje po vrijednosti (C++ implementacija).

U primjeru kôda 3.9 možemo vidjeti da neće doći do promjene vrijednosti varijable `a`. To se neće dogoditi jer funkcija neće primiti originalnu varijablu `a`, već će se stvoriti kopija varijable `a` na drugoj memorijskoj lokaciji koja će vrijediti samo unutar vitičastih zagrada funkcije `funkcija`. Sve promjene nad varijablom `a` unutar funkcije odnosit će se na kopiju, a ne na originalnu varijablu te iz tog razloga promjena nije vidljiva u `main` funkciji. Stanje u memoriji možemo vidjeti na slici 3.4. Kada biste htjeli



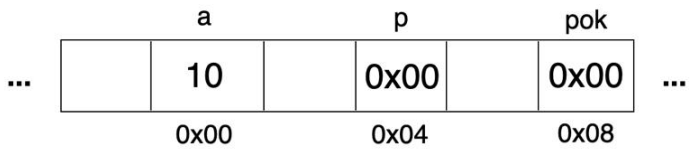
Slika 3.4: Prikaz adresiranja u programskom jeziku C++.

vidjeti promjenu originalne varijable, moramo koristiti prosljeđivanje po adresi ili referenci. U jeziku C/C++ to možemo postići koristeći pokazivače (u programskom jeziku Java pokazivači ne postoje, ali postoje reference koje u velikoj mjeri predstavljaju zamjenu za pokazivače, a u kasnijim primjerima vidjet ćemo kako se koriste reference u Javi). Pokazivači u jeziku C/C++ predstavljaju varijable koje u sebi sadrže adresu na neki blok memorije, a u ovom slučaju sadržavat će adresu varijable `a`.

```
void funkcija(int * pok){ *pok *= *pok; }
int main(){
        int a = 10;
        int * p = &a; //deklaracija pokazivaca
                      //i dodjeljivanje adrese varijable "a"
        cout<<a; //ispisat ce 10
        funkcija(p); // ili funkcija(&a) - prosljedjivanje po adresi
        cout<<a; //ispisat ce 100
}
```

Primjer kôda 3.10 Prosljeđivanje po adresi (C++ implementacija).

U primjeru kôda 3.10 možemo vidjeti deklaraciju pokazivača `p` koji pokazuje na varijablu `a` (sadrži adresu varijable `a`). Funkcija prima pokazivač na cjelobrojnu vrijednost, odnosno prilikom pozivanja iz `main` funkcije, mi radimo prosljeđivanje po adresi. Na slici 3.5 možemo vidjeti stanje u memoriji nakon izvršavanja kôda iz primjera kôda 3.10. Međutim, ovisno o tome koristimo li `funkcija(p)` ili `funkcija(&a)`, u pozadini će se stvari događati na dva različita načina. Ukoliko koristimo `funkcija(p)`, uspjet ćemo promijeniti sadržaj originalne varijable `a`; međutim tu zapravo nije došlo do prosljeđivanja po adresi kao što će to biti u slučaju korištenja `funkcija(&a)`, već po vrijednosti. Onog trenutka kada smo proslijedili pokazivač `p` u funkciju, napravila se kopija tog pokazivača u obliku pokazivača `pok` koji sada pokazuje na istu memorijsku adresu kao i pokazivač `p`. Svaka promjena koju napravimo pokazivačem `pok` odnosit će se na vrijednost na koju je originalno pokazivao pokazivač `p`, odnosno na



Slika 3.5: Prikaz prosljeđivanja po adresi u programskom jeziku C++.

vrijednost varijable a. Ovo je jako važno za zapamtiti kako biste mogli shvatiti prosljeđivanje argumenata u programskom jeziku Java.

Iako se u nekim literaturama može krivo zaključiti da u programskom jeziku Java postoji prosljeđivanje po referenci i po vrijednosti, važno je napomenuti da postoji prosljeđivanje jedino po vrijednosti. Međutim, ovo nekada zna biti zbunjujuće, što će ilustrirati sljedeći primjer:

```
public class Klasa1 {
        static public void funkcija(int a){ a*=a; }
        static public void funkcija2(Broj i){ i.a*= i.a;}
        public static void main(String[] args){
                int a = 10;
                Broj b = new Broj();
                System.out.println(b.a); //ispisat ce 10
                funkcija2(b);
                System.out.println(b.a); //ispisat ce 100
                System.out.println(a); //ispisat ce 10
                funkcija(a);
                System.out.println(a); //ispisat ce 10
        }
        static class Broj{
            public int a=10;
        }
}
```

Primjer kôda 3.11 Prosljeđivanje po vrijednosti u Java programskom jeziku.

U primjeru kôda 3.11 možemo vidjeti da ćemo dobiti dva različita ispisa za naočigled isti primjer koji smo naveli ranije. Međutim, kada u Javi prosljeđujemo primitivne tipove prosljeđuju se njihove kopije i bilo koja promjena neće utjecati na vrijednost originalne varijable. Isto se događa i kada prosljeđujemo reference. Kada proslijedimo referencu na objekt klase `Broj` u metodu, pravi se kopija te reference koja sadrži istu adresu pa bilo koja promjena utječe i na originalnu vrijednost. S obzirom na to, možemo zaključiti da u Javi postoji jedino **prosljeđivanje po vrijednosti**!

3.9 Preopterećivanje metoda

Postoji tri načina preopterećivanja (engl. *overloading*) metoda:

- Broj argumenata
  - `ispis(int, int);`
  - `ispis(int, int, int);`

```
class Klasa1{
        public void funkcija(int a){
                System.out.println(a);
        }
        public void funkcija(int a, int b){
                System.out.println(a + b);
        }
}
class Klasa2{
```

```
        public static void main(String [] args){
                Klasa1 obj = new Klasa1();
                obj.funkcija(1);
                obj.funkcija(1,2);
        }
}
```

Primjer kôda 3.12 Preopterećivanje metode (broj argumenata).

- Tip argumenata
  - ◇ `ispis(int, int);`
  - ◇ `ispis(double, int);`

```
class Klasa1{
        public void funkcija(int a){
                System.out.println(a);
        }
        public void funkcija(double a){
                System.out.println(a);
        }
}
class Klasa2{
        public static void main(String [] args){
                Klasa1 obj = new Klasa1();
                obj.funkcija(1);
                obj.funkcija(1.25);
        }
}
```

Primjer kôda 3.13 Preopterećivanje metode (tip argumenata).

- Redoslijed tipova argumenata
  - ◇ `ispis(int, double);`
  - ◇ `ispis(double, int);`

```
class Klasa1{
        public void funkcija(int a, double b){
                System.out.println((double)a+b);
        }
        public void funkcija(double a, int b){
                System.out.println(a + (double)b);
        }
}
class Klasa2{
        public static void main(String [] args){
                Klasa1 obj = new Klasa1();
                obj.funkcija(1,1.25);
                obj.funkcija(1.25,2);
        }
}
```

Primjer kôda 3.14 Preopterećivanje metode (redoslijed tipova argumenata).

3.10 Konstruktor

Konstruktor predstavlja posebanu vrstu metode koja se koristi za inicijalizaciju objekta, a ukoliko mi ne inicijaliziramo objekt, onda će svi atributi poprimiti pretpostavljene vrijednosti. Konstruktor se poziva svaki puta kada se pravi novi primjerak objekta, tj. kada se instancira. Postoje temeljna dva pravila koja je potrebno poštivati prilikom definiranja konstruktora:

- naziv konstruktora mora biti isti kao i naziv klase čiji je to konstruktor;
- konstruktor ne smije imati eksplicitno navedeni povratni tip.

U programskom jeziku Java moguće je napisati dvije vrste konstruktora:

- pretpostavljeni konstruktor (konstruktor bez argumenata);
- ostali konstruktori.

3.10.1 Pretpostavljeni konstruktor

Pretpostavljeni konstruktor predstavlja konstruktor koji nema argumente. Potrebno je napomenuti da se pretpostavljeni konstruktor automatski generira ukoliko programer drugačije nije naznačio. Glavna svrha pretpostavljenog konstruktora jest inicijalizacija varijabli objekta na pretpostavljenu vrijednost (0, null i slično).

```
public class Klasa1{
        Klasa1(){
                System.out.println("Pretpostavljeni konstruktor");
        }
        public static void main(String [] args){
                //main funkcija
        }
}
```

Primjer kôda 3.15 Pretpostavljeni konstruktor.

3.10.2 Ostali konstruktori

Ostali konstruktori predstavljaju konstruktore koji na ulazu primaju određeni broj argumenata, a glavna im je svrha inicijalizacija varijabli nekog objekta drugačijim početnim vrijednostima. Primjerice, imamo klasu `Automobil` iz koje instanciramo dva objekta koji predstavljaju dva automobila s dvjema različitim vrstama mjenjača. Prilikom instanciranja tih dvaju objekata, u jedan ćemo konstruktor proslijediti da se radi o ručnom mjenjaču, dok ćemo u drugi proslijediti da se radi o automatskom mjenjaču.

```
public class Klasa1{
int var;
        Klasa1(int vrijednost){
                var = vrijednost;
        }
        public static void main(String [] args ){
                //main funkcija
        }
}
```

Primjer kôda 3.16 Pisanje konstruktora.

Ako argument koji se šalje konstruktoru ima isto ime kao varijabla u klasi, tada je potrebno ključnom riječi `this` označiti da se radi o varijabli objekta kao što je prikazano u primjeru kôda 3.18.

```
public class Klasa1{
int var;
        Klasa1(int var){
                this.var = var;
        }
        public static void main(String args []){
                //main funkcija
        }
}
```

Primjer kôda 3.17 Ključna riječ `this`.

Osim u navedenom primjeru (kada varijabla metode sakriva varijablu instance), ključnu riječ `this` možemo koristiti unutar konstruktora radi pozivanja drugog konstruktora unutar iste klase (engl. *explicit constructor invocation*), što se naziva delegiranje. Primjer kôda 3.18 upravo prikazuje takav način upotrebe ključne riječi `this`. Neovisno koji konstruktor pozovemo, uvijek ćemo završiti s inicijalizacijom svih privatnih članova unutar trećeg konstruktora koji prima četiri argumenta.

```
public class Klasa1 {
        private int b1,b2,b3,b4;

        public Klasa1(){
                this(0,0,0,0);
        }
        public Klasa1(int b1){
                this(b1,0,0,0);
        }
        public Klasa1(int b1, int b2, int b3, int b4){
                this.b1 = b1;
                this.b2 = b2;
                this.b3 = b3;
                this.b4 = b4;
        }
}
```

Primjer kôda 3.18 Korištenje ključne riječi `this`.

3.10.3 Razlika između metode i konstruktora

Tablica 3.1: Razlike između metode i konstruktora.

Konstruktor	Metoda
Ne može imati povratni tip	Mora imati povratni tip
Poziva se implicitno	Poziva se eksplicitno
Prevoditelj generira pretpostavljeni konstruktor ukoliko programer nije definirao niti jedan	Metoda se ne generira automatski ni u kojem slučaju
Naziv konstruktora mora biti isti kao i naziv klase u kojoj se definira	Nazivi metoda mogu se proizvoljno definirati

3.11 Preopterećivanje konstruktora

Kao što je to slučaj kod metoda, i konstruktore je moguće preopteretiti. Ideja je omogućiti stvaranje više od jednog konstruktora gdje svaki od konstruktora obavlja drugačiji zadatak.

```
public class Klasa1{
        private int var;
        Klasa1(){
                var = 10;
        }
        Klasa1(int broj){
                this(); //odnosi se pozivanje pretpostavljenog konstruktora
                var += broj;
        }
        public int dohvatiVar(){ return var; }
        public void setVar(int novaVar){ this.var = novaVar; }
        public static void main(String [] args){
                Klasa1 ob = new Klasa1(55);
                System.out.println(ob.dohvatiVar()); //ispisat ce 65
        }
```

```
}
```

Primjer kôda 3.19 Preopterećivanje konstruktora.

3.12 Zadaci

Zadatak 3.1. Napisati klasu `Zarulja` koja ima jedan privatni član tipa `boolean`: `stanje`, pretpostavljeni konstruktor koji postavlja stanje u `false` te javne metode `pritisniPrekidac` koji će uključivati i isključivati žarulju i `provjeri` koja će ispisivati tekst `"Svijetli"` ili tekst `"Mrak je"` ovisno o trenutnom stanju (`false` – mrak, `true` – svijetli). Kreiraj objekt `osram` klase `Zarulja` te uključi i isključi žarulju nekoliko puta.

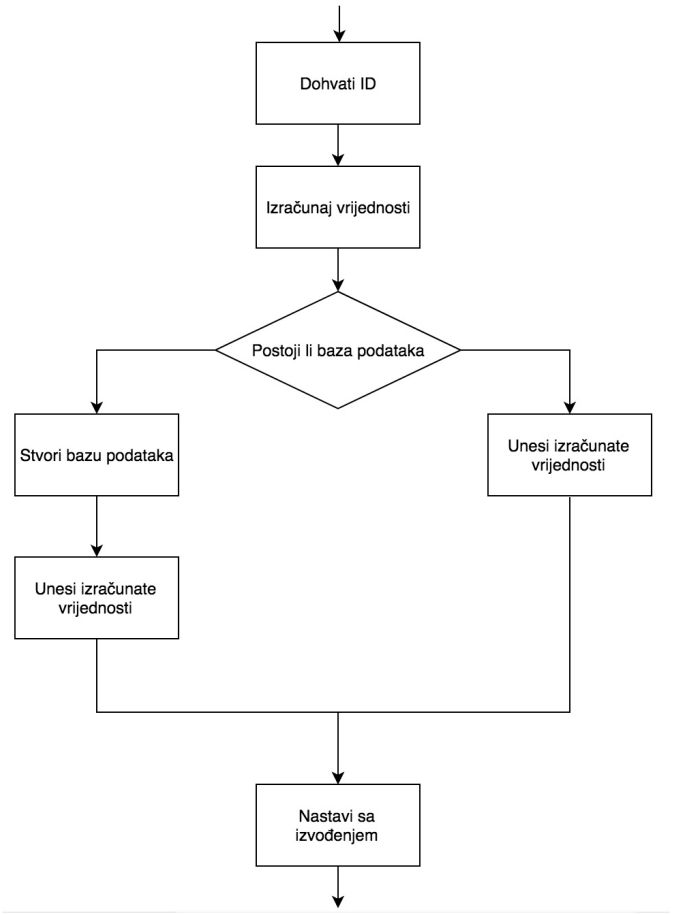
Zadatak 3.2. Napisati klasu `Zaposlenik` koja ima privatne podatkovne članove `staz`, `starost` i `placa`. Napisati konstruktor koji inicijalizira podatkovne članove prilikom stvaranja objekta. Ako se ne proslijede vrijednosti u konstruktor `staz`, `starost` i `placa` se trebaju postaviti na 0. Napisati pristupne metode kojima se mogu promijeniti podatkovni članovi kao i metode kojima se može dobiti njihova vrijednost. Napisati dvije metode koje računaju koliko je do sada zaradio zaposlenik. Jedna metoda ne prima nikakav argument pri pozivu, nego računa s podacima objekta koji ju poziva, a druga je metoda statička i prima referencu na objekt klase `Zaposlenik` za koji onda radi izračun. Obje metode računaju koliko je zaposlenik ukupno zaradio za vrijeme rada (`mjeseci * broj godina staža * mjesečna plaća`).

Zadatak 3.3. Napisati klasu `Kalkulator` koja ima mogućnost računanja matematičkih operacija zbrajanje i množenje (moguće je obaviti operacije za `int` i `double` tip podataka). Ovo je potrebno riješiti preopterećivanjem metoda. Pozvati računanje s npr. `new Kalkulator().zbrajanje(5,10)`. U novoj klasi `Test` isprobati sve mogućnosti i ispisati rezultate.

Poglavlje 4

Uvjeti, petlje i kontrola grešaka

Bilo koji računalni problem može se riješiti izvršavanjem niza radnji određenim redoslijedom. Takav redoslijed izvršavanja zove se algoritam. Određivanje redoslijeda izvođenja radnji naziva se kontrola programa (engl. *program control*), a u ovom poglavlju bavit ćemo se kontrolom tijeka programa (engl. *control statements*). Pri razvoju nekog složenijeg programskog rješenja, programeri nerijetko dugo vremena provedu crtajući razne dijagrame tijeka (engl. *flowchart*) koji predstavljaju buduće ponašanje programa. Primjer jednostavnog dijagrama prikazan je na slici 4.1.



Slika 4.1: Jednostavan dijagram.

4.1 Naredba `if`

Naredba `if` predstavlja najjednostavniji oblik donošenja odluke u programskom jeziku Java. Možemo zamisliti da postoji deklarirana varijabla "varijabla1" i ukoliko je vrijednost te varijable manja ili jednaka od 10 tada se ispisuje određena poruka.

```
public class Klasa1{
        public static void main(String [] args) {
                int varijabla1 = 8;
                if(varijabla1<=10){
                    System.out.println("Ispis neke poruke");
                }
        }
}
```

Primjer kôda 4.1 Naredba `if`

S obzirom da unutar naredbe `if` koristimo relacijske operatore (==, <, >, <=, >= i !=), rezultat usporedbe je `true` ili `false`. Prema tome, primjer iznad mogli smo napisati i kao:

```
public class Klasa1{
        public static void main(String [] args){
                int varijabla1 = 8;
                boolean b = varijabla1 <=10;
                if(b){
                        System.out.println("Ispis neke poruke");
                }
        }
}
```

Primjer kôda 4.2 Naredba `if` – drugi način

Važno je napomenuti da standardne relacijske operatore možemo koristiti za uspoređivanje vrijednosti primitivnih tipova podataka kao što je `int` (cjelobrojni tip). Operatori <, >, <= ili >= ne mogu se koristiti za uspoređivanje objekata jer će inače doći do greške prilikom kompajliranja. Operatori == i != mogu se koristiti nad objektima s tim da će se umjesto njihove vrijednosti uspoređivati sadržaji njihovih referenci. Za uspoređivanje objekata potrebno je pisati specijalizirane metode s obzirom da preopterećenje operatora u Javi nije moguće.

4.2 Naredba `if-else`

U primjeru kôda 4.3 uspoređuje se vrijednost `varijabla1` i u slučaju zadovoljavanja uvjeta, ispisuje se poruka. Međutim, dosta je često potrebno dodati više opcija toka programa ovisno o rezultatu usporedbe. U tu svrhu moguće je koristiti naredbu `if-else` koja predstavlja mehanizam obavljanja jedne radnje u slučaju da je rezultat usporedbe `true` i druge radnje ukoliko je rezultat usporedbe `false`.

```
public class Klasa1{
    public static void main(String [] args){
        int varijabla1 = 10;
        if(varijabla1 <= 10){
            System.out.println("varijabla1 je manja ili jednaka 10");
        }else{
            System.out.println("varijabla1 je veca od 10");
        }
    }
}
```

Primjer kôda 4.3 Naredba `if-else`

4.3 Ugniježđena naredba `if-else`

Osim naredbi `if` i `if-else`, moguće je "testirati" višestruke slučajeve uz pomoć ugniježđenih naredbi `if-else`. Za primjer možemo uzeti slučaj dodjeljivanja ocjene na temelju ostvarenog postotka na pismenom ispitu. (>=85%=5, >=75%=4, >=65%=3, >=50%=2, <50%=1).

```
public class Klasa1{
    public static void main(String [] args){
        int rezultat = 78;
        if(rezultat >= 85){ System.out.println("5");
        }else{
            if(rezultat >= 75){ System.out.println("4");
            }else{
                if(rezultat >= 65){ System.out.println("3");
                }else{
                    if(rezultat >= 50){ System.out.println("2");
                    }else{ System.out.println("1"); }
                }
            }
        }
    }
}
```

Primjer kôda 4.4 Ugniježđena naredba `if-else`

Ili na sljedeći način koji se preporuča:

```
public class Klasa1{
        public static void main(String [] args){
                int rezultat = 78;
                if(rezultat >= 85){
                        System.out.println("5");
                }else if(rezultat >= 75){
                        System.out.println("4");
                }else if(rezultat >= 65){
                        System.out.println("3");
                }else if(rezultat >= 50){
                        System.out.println("2");
                }else{
                        System.out.println("1");
                }
        }
}
```

Primjer kôda 4.5 Naredba `if-else if-else`.

4.4 Petlja `while`

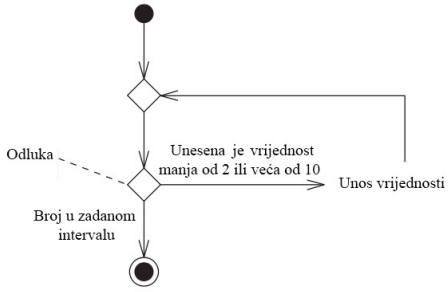
Petlja `while` omogućava izvođenje nekog bloka naredbi dok vrijedi određeni uvjet. Kao primjer možemo navesti situaciju gdje želimo unijeti neki broj koji se nalazi u određenom intervalu. Ukoliko se broj ne nalazi u zadanom intervalu, ponavlja se operacija unosa.

```
public class Klasa1{
        public static void main(String [] args){
                int unos=0;
                Scanner input = new Scanner(System.in);
                while(unos<2 || unos>10){
                        System.out.println("Unesite vrijednost:");
                        unos = input.nextInt();
                }
```

```
        }
}
```

Primjer kôda 4.6 Petlja `while`.

U primjeru kôda 4.6 želimo da unesena vrijednost bude u intervalu između 2 i 10.



Primjer kôda 4.7 Dijagram while petlje.

Petlju `while` također možemo koristiti i za ponavljanja uvjetovana brojačem. Za takav način ponavljanja potrebno je definirati kontrolnu varijablu (brojač), početnu vrijednost kontrolne varijable, inkrement ili dekrement nad kontrolnom varijablom, ovisno u kojem smjeru želimo ići, i uvjet ponavljanja. Kao primjer možemo navesti ispis brojeva od 1 do 10.

```
public class Klasa1{
        public static void main(String [] args){
                int brojac=1;
                while(brojac <= 10){
                        System.out.print(brojac+" ");
                        brojac++;
                }
        }
}
```

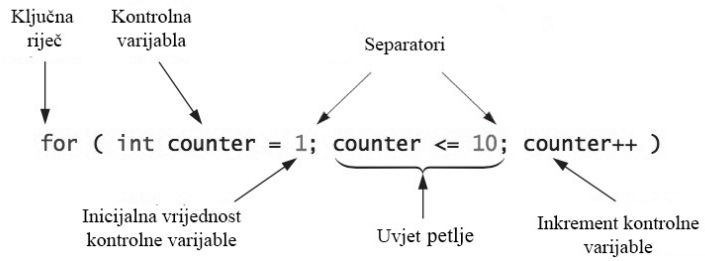
Primjer kôda 4.8 Petlja `while` s inkrementom.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
1 2 3 4 5 6 7 8 9 10
```

4.5 Petlja `for`

Iako je moguće definirati petlju `while` uvjetovanu kontrolnom varijablom, najčešći način na koji se takvo ponavljanje radi je korištenjem petlje `for`.

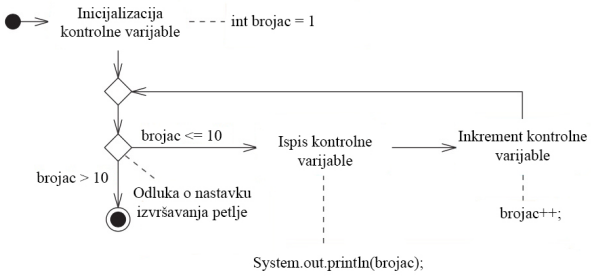


Slika 4.2: Definicija for petlje.

Implementacija primjera koji ispisuje brojeve od 1 do 10 korištenjem petlje `for` dana je u primjeru kôda 4.9:

```
public class Klasa1{
        public static void main(String [] args){
                for(int i=1;i<=10;i++){
                        System.out.println(i);
                }
        }
}
```

Primjer kôda 4.9 Petlja for.



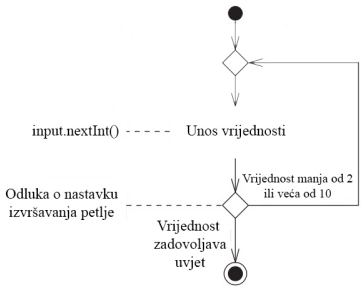
Primjer kôda 4.10 Dijagram petlje for.

4.6 Petlja do-while

Ova vrsta petlje slična je petlji while s razlikom da se petlja while ne mora niti jednom izvesti, dok se petlja do-while izvodi barem jednom. Razlog tome je što se kod petlje while prvo provjerava uvjet petlje te ukoliko uvjet nije zadovoljen, petlja se preskače. Kao primjer najbolje bi bilo ponoviti primjer naveden ranije, a odnosi se na unos broja u određenom intervalu. Uzet ćemo isti uvjet da broj mora biti u intervalu od 2 do 10 te da se unos mora ponavljati sve dok uvjet nije zadovoljen.

```
public class Klasa1{
        public static void main(String [] args){
                int unos = 0;
                Scanner input = new Scanner(System.in);
                do{
                        unos = input.nextInt();
                }while(unos <2 || unos >10);
        }
}
```

Primjer kôda 4.11 Petlja do-while.



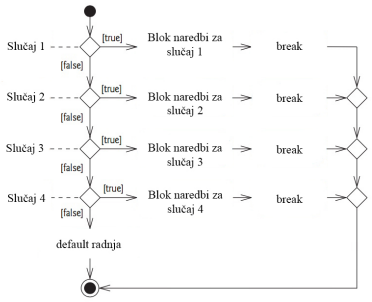
Primjer kôda 4.12 Dijagram petlje do-while.

4.7 Naredba `switch`

Naredba `switch` omogućava izvršavanje više različitih radnji ovisno o odabiru. Kao jednostavan primjer možemo navesti odabir osnovne matematičke operacije (zbrajanje, oduzimanje, dijeljenje, množenje) za neka dva ulazna broja.

```
public class Klasa1{
        public static void main(String [] args){
                int odabir,a,b;
                Scanner input = new Scanner(System.in);
                System.out.println("Unesite vrijednost broja a: ");
                a = input.nextInt();
                System.out.println("Unesite vrijednost broja b: ");
                b = input.nextInt();
                System.out.println("Odaberite jednu od mogucnosti: ");
                do {
                        System.out.println("####################");
                        System.out.println("Zbrajanje - 1 ");
                        System.out.println("Oduzimanje - 2");
                        System.out.println("Dijeljenje - 3 ");
                        System.out.println("Mnozenje - 4");
                        System.out.println("Izlaz - -1");
                        System.out.println("####################");
                        odabir = input.nextInt();
                        switch(odabir){
                                case 1:
                                System.out.println("Rezultat: "+(a+b));
                                break;
                                case 2:
                                System.out.println("Rezultat: "+(a-b));
                                break;
                                case 3:
                                System.out.println("Rezultat: "+(a/b));
                                break;
                                case 4:
                                System.out.println("Rezultat: "+(a*b));
                                break;
                                default:
                                break;
                        }
                } while(odabir!=-1);
}}
```

Primjer kôda 4.13 Switch blok.



Primjer kôda 4.14 Dijagram switch bloka.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
Unesite vrijednost broja a:
2
Unesite vrijednost broja b:
3

Odaberite jednu od mogucnosti:
######################
Zbrajanje - 1
Oduzimanje - 2
Dijeljenje - 3
Mnozenje - 4
Izlaz - -1
######################
1
Rezultat: 5
######################
Zbrajanje - 1
Oduzimanje - 2
Dijeljenje - 3
Mnozenje - 4
Izlaz - -1
######################
2
Rezultat: -1
######################
Zbrajanje - 1
Oduzimanje - 2
Dijeljenje - 3
Mnozenje - 4
Izlaz - -1
######################
4
Rezultat: 6
######################
Zbrajanje - 1
Oduzimanje - 2
Dijeljenje - 3
Mnozenje - 4
Izlaz - -1
######################
3
Rezultat: 0
######################
Zbrajanje - 1
Oduzimanje - 2
Dijeljenje - 3
Mnozenje - 4
Izlaz - -1
######################
-1
```

4.8 Naredbe `break` i `continue`

Obje naredbe koriste se za kontrolu toka izvođenja programa unutar neke petlje. Prilikom izvršavanja naredbi `while`, `for`, `do-while` ili `switch` korištenje naredbe `break` osigurava prekid izvođenja i izlazak iz petlje ili naredbe `switch`. Za primjer možemo navesti situaciju kada imamo petlju koja ispisuje prvih

10 brojeva, ali mi želimo ispisati samo prvih pet.

```
public class Klasa1{
        public static void main(String [] args){
                for(int i=1;i<=10;i++){
                        System.out.print(i+" ");
                        if(i == 5){
                        break;
                        }
                }
        }
}
```

Primjer kôda 4.15 Naredba break.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
1 2 3 4 5
```

Naredbu `continue` koristimo u petljama `while`, `for` ili `do-while` kada želimo preskočiti trenutnu iteraciju u petlji. Primjerice, imamo for petlju koja ispisuje brojeve od 1 do 10, ali ne želimo ispisati broj 5.

```
public class Klasa1{
        public static void main(String [] args){
                for(int i=1;i<=10;i++){
                        if(i == 5){
                                continue;
                        }
                        System.out.println(i+" ");
                }
        }
}
```

Primjer kôda 4.16 Continue naredba.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
1 2 3 4 6 7 8 9 10
```

4.9 Zadaci

Zadatak 4.1. Napisati program koji će imati sljedeće mogućnosti odabira (koristiti naredbu `switch`):
(1) Izračunati opseg pravokutnika.
(2) Izračunati površinu kvadrata.
(3) Izračunati aritmetičku sredinu parnih brojeva. Program na početku treba od korisnika zatražiti unos maksimalnog broja brojeva `n`. Koristeći petlju `do-while` ograničiti unos broja `n` na interval između 2 i 10. Unos se mora ponavljati sve dok nije unesen broj u traženom intervalu. Nakon toga krenuti s unosom i izračunom aritmetičke sredine.

Zadatak 4.2. Doraditi zadatak 3.3. iz poglavlja 3 tako da dodate metodu za matematičku operaciju dijeljenja. Potrebno je provjeriti djelitelj te ako je vrijednost djelitelja nula, onda ispisati poruku da dijeljenje nije moguće.

Zadatak 4.3. Napisati klasu `PogodiBroj` koja sadrži metodu `provjeri(int broj)`. Prilikom instanciranja objekta klase `PogodiBroj` u konstruktoru se treba inicijalizirati jedan slučajan broj u određenom intervalu (zadajete sami). U klasi `Test` korisnika trebate pitati `N` puta neka pogodi koji je broj izgeneriran (pomoću neke od petlji) te da brojite broj pokušaja. Kada korisnik pogodi o kojem se broju radi, na ekran ispisati `"Uspjeh, broj pokušaja je ?`.
Napomena:

- Instanciranje objekta obaviti unutar klase `Test` kao i pogađanje broja.

- Slučajni broj možete generirati uz pomoć sljedeće metode:
```
public int randomN(int gornjaGranica, int donjaGranica) {
   return (int) (Math.random() * (gornjaGranica - donjaGranica)) + donjaGranica;
}
```

Poglavlje 5

Nasljeđivanje

Nasljeđivanje je važno svojstvo objektno-orijentiranog programiranja. Osnovna je ideja nasljeđivanja da se prilikom razvoja identificiraju razredi koji imaju sličnu funkcionalnost te da se u izvedenim razredima samo redefiniraju specifična svojstva, dok se preostala svojstva nasljeđuju u nepromijenjenom obliku. Osnovni pojmovi s kojima se možemo susresti u literaturi, a koji su vezani za nasljeđivanje:

- osnovna klasa (engl. *base class*);
- roditeljska klasa (engl. *parent class*);
- superklasa (engl. *superclass*);

} Klasa iz koje nasljeđujemo

- izvedena klasa (engl. *derived class*);
- podklasa (engl. *subclass*);
- klasa dijete (engl. *child class*).

} Klase koja nasljeđuje

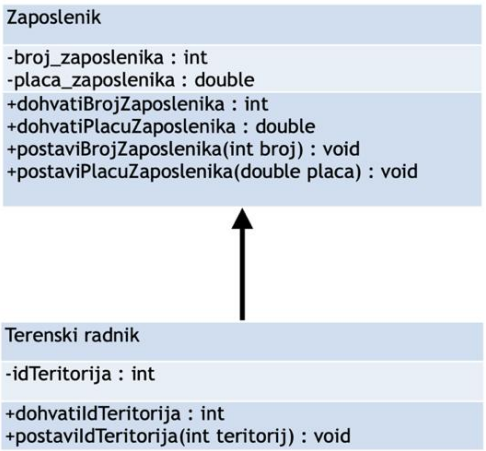
Kao što je napisano na početku, prilikom stvaranja nove klase, umjesto stvaranja potpuno novih članova, možemo navesti da nova klasa nasljeđuje već postojeću klasu. Prilikom toga, već postojeća klasa naziva se osnovna klasa, dok se klasa koja nasljeđuje članove naziva podklasa. Zbog mogućnosti promjene ponašanja osnovne klase, nasljeđivanje se još može nazvati i specijalizacijom osnovne klase.

Važno je napomenuti još dva pojma, a to su:

- direktna osnovna klasa - predstavlja klasu iz koje podklasa eksplicitno nasljeđuje;
- indirektna osnovna klasa - predstavlja bilo koju klasu iznad direktne osnovne klase u hijerarhiji klasa.

Prednosti nasljeđivanja:

- ušteda vremena;
- smanjivanje pogrešaka;
- smanjivanje krivulje učenja i prilagođavanja.



Primjer kôda 5.1 Nasljeđivanje.

5.1 Ključna riječ `extends`

Kako bi se postiglo nasljeđivanje u programskom jeziku Java, koristi se ključna riječ `extends`.

```
public class Zaposlenik {
        private int brojZaposlenika;
        private double placaZaposlenika;
        public int dohvatiBrojZaposlenika(){
                return brojZaposlenika;
        }
        public double dohvatiPlacuZaposlenika(){
                return placaZaposlenika;
        }
        public void postaviBrojZaposlenika(int broj){
                brojZaposlenika = broj;
        }
        public void postaviPlacuZaposlenika(double placa){
                placaZaposlenika = placa;
        }
}
```

Primjer kôda 5.2 Osnovna klasa `Zaposlenik`.

Primjer kôda 5.2 prikazuje opću klasu Zaposlenik koja u sebi sadrži dvije privatne varijable (varijable instance) te `get()` i `set()` metode za dohvat, odnosno postavljanje istih.

```
public class TerenskiRadnik extends Zaposlenik{
        private int idTeritorija;
        public int dohvatiIdTeritorija () {
                return idTeritorija;
        }
        public void postaviIdTeritorija(int teritorij){
                idTeritorija = teritorij;
        }
}
```

Primjer kôda 5.3 Nasljeđivanje osnovne klase `Zaposlenik`.

Primjer kôda 5.3 prikazuje klasu `TerenskiRadnik` koja predstavlja dodatnu specifikaciju klase `Zaposlenik`.

5.2 Operator `instanceof`

Operator `instanceof` u Javi koristi se za određivanje pripadnosti neke instance određenoj osnovnoj klasi, podklasi ili sučelju (upoznat ćemo se u nastavku). Operator `instanceof` poznat je još i kao operator uspoređivanja (engl. *comparison operator*). Kao rezultat, vraća `true` ili `false`.

```
TerenskiRadnik radnik1 = new TerenskiRradnik();
radnik1 instanceof TerenskiRradnik // vraca true //1
radnik1 instanceof Zaposlenik //vraca true //2
Zaposlenik radnik2 = new Zaposlenik();
radnik2 instanceof Zaposlenik //vraca true  //3
radnik2 instanceof TerenskiRadnik //vraca false  //4
```

Primjer kôda 5.4 Upotreba operatora `instanceOf`.

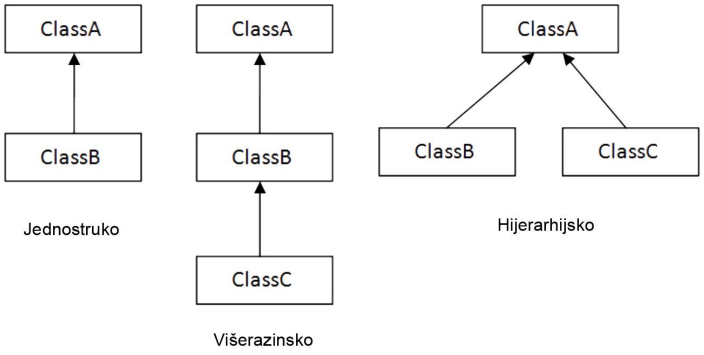
Za primjer kôda 5.4, ovisno o izrazu, dobili bismo sljedeće rezultate:

1. `true` – zato što je `radnik1` instanca klase `TerenskiRadnik`.
2. `true` – zato što je klasa `TerenskiRadnik` podklasa klase `Zaposlenik`.
3. `true` – zato što je `radnik3` instanca klase `Zaposlenik`.
4. `false` – zato što klasa `Zaposlenik` nema IS-A odnos s klasom `TerenskiRadnik`.

5.3 Pravila nasljeđivanja

Programski jezik Java podržava samo jednu upotrebu ključne riječi `extends` za određenu klasu. Prema tome, podržani su sljedeći tipovi nasljeđivanja:

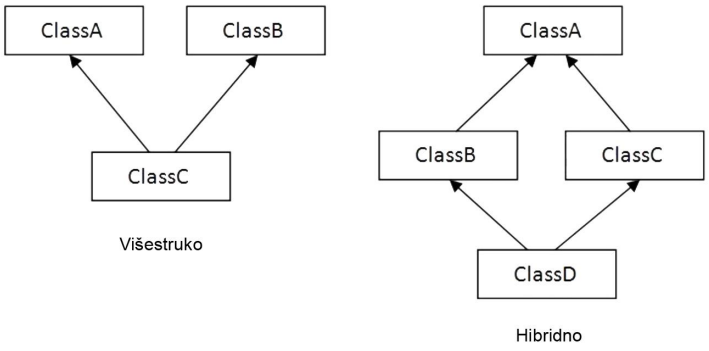
- jednostruko;
- višerazinsko;
- hijerarhijsko.



Slika 5.1: Dozvoljeni načini nasljeđivanja.

Sljedeće vrste nasljeđivanja nisu dopuštene u programskom jeziku Java:

- višestruko;
- hibridno.



Slika 5.2: Nedozvoljeni načini nasljeđivanja.

S obzirom da je jednostruko nasljeđivanje već prikazano u primjeru ranije, u sljedećim dvama primjerima prikazat će se primjer višerazinskog i hijerarhijskog nasljeđivanja.

```
public class Zaposlenik {
        private int brojZaposlenika;
         private double placaZaposlenika;
         public int dohvatiBrojZaposlenika(){
                return brojZaposlenika;
        }
        public double dohvatiPlacuZaposlenika(){
                return placaZaposlenika;
        }
...
}
```

Primjer kôda 5.5 Višerazinsko nasljeđivanje 1.

```
public class TerenskiRadnik extends Zaposlenik{
        private int idTeritorija;
        public int dohvatiIdTeritorija () {
                return idTeritorija;
        }
        public void postaviIdTeritorija(int teritorij){
                idTeritorija = teritorij;
        }
}
```

Primjer kôda 5.6 Višerazinsko nasljeđivanje 2.

```
public class AsistentTerenskiRadnik extends TerenskiRadnik{
        private int idAsistenta;
        public int dohvatiIdAsistenta () {
                return idAsistenta;
        }
        public void postaviIdAsistenta(int idAsist){
                idAsistenta = idAsist;
        }
}
```

Primjer kôda 5.7 Višerazinsko nasljeđivanje 3.

```
public static void main(String [] args){
        AsistentTerenskiRadnik radnik1 = new AsistentTerenskiRadnik();
        radnik1.dohvatiPlacuZaposlenika();
        radnik1.postaviIdTeritorija(23);
        radnik1.dohvatiIdAsistenta();
}
```

Primjer kôda 5.8 Instanciranje objekta klase – višerazinsko nasljeđivanje.

U primjeru koda 5.6 i 5.7 prikazane su dvije klase `TerenskiRadnik` i `AsistentTerenskiRadnik`. Klasa `TerenskiRadnik` nasljeđuje klasu `Zaposlenik`, dok klasa `AsistentTerenskiRadnik` nasljeđuje klasu `TerenskiRadnik`. Lako se može vidjeti kako klase `TerenskiRadnik` i `AsistentTerenskiRadnik` predstavljaju sve dublju specijalizaciju klase `Zaposlenik`, a samim time imaju i pristup članovima i metodama svojih nadklasa (ovisno o pristupnim modifikatorima).

```
public class Zaposlenik {
        private int brojZaposlenika;
        private double placaZaposlenika;
        public int dohvatiBrojZaposlenika(){
                return brojZaposlenika;
        }
        public double dohvatiPlacuZaposlenika(){
                return placaZaposlenika;
        }
...
}
```

Primjer kôda 5.9 Hijerarhijsko nasljeđivanje 1.

```
public class TerenskiRadnik extends Zaposlenik{
        private int idTeritorija;
        public int dohvatiIdTeritorija () {
                return idTeritorija;
        }
    public void postaviIdTeritorija(int teritorij){
        idTeritorija = teritorij;
    }
}
```

Primjer kôda 5.10 Hijerarhijsko nasljeđivanje 2.

```
public class Tajnica extends Zaposlenik{
        private int idTajnice;
        public int dohvatiIdTajnice () {
                return idTajnice;
        }
        public void postaviIdTajnice(int id){ idTajnice = id; }
}
```

Primjer kôda 5.11 Hijerarhijsko nasljeđivanje 3.

Na sličan način kao u prethodnom primjeru, kod višerazinskog nasljeđivanja primjeri kôda 5.10 i 5.11 prikazuju dodatnu specifikaciju klase `Zaposlenik` u obliku klasa `TerenskiRadnik` i `Tajnica` gdje obje navedene klase nasljeđuju istu klasu.

5.4 Nasljeđivanje i modifikatori pristupa

U prethodnom dijelu upoznali smo se s modifikatorima pristupa u programskom jeziku Java. Tablica 5.1 poslužit će nam kao podsjetnik na modifikatore pristupa i što točno oni predstavljaju:

Tablica 5.1: Pravila pristupnih modifikatora.

Pristupni modifikator	**Unutar klase**	**Unutar paketa**	**Podklasa** (isti paket)	**Podklasa** (drugi paket)	**Izvan** paketa
`public`	X	X	X	X	X
`protected`	X	X	X	X	
privatni za paket	X	X	X		
`private`	X				

5.4.1 Zaštićeni članovi

Ključna riječ `protected` predstavlja stupanj zaštite koji se po sigurnosti može smjestiti između javnog i privatnog za paket pristupa gdje je:

- `private` – najveći stupanj zaštite (najmanji stupanj vidljivosti);
- `public` – najmanji stupanj zaštite (najveći stupanj vidljivosti).

Koristimo ga kada želimo dozvoliti pristup nekom članu iz same klase ili iz njenih podklasa. Klase koje su u istom paketu imaju pristup svim zaštićenim (`protected`) članovima neovisno o tome nasljeđuju li spomenutu klasu ili ne.

S obzirom na navedeno, kod nasljeđivanja vrijedi nekoliko pravila:

- podklasa nasljeđuje sve varijable kao i metode osnovne klase, koje su deklarirane kao public ili protected bez obzira na izvorišni paket;
- podklasa nasljeđuje sve varijable kao i metode osnovne klase, koje nemaju deklariran identifikator pristupa (paketni pristup) pod uvjetom da se nalaze u istom paketu;

- podklasa ne može pristupiti nasljeđenim privatnim varijablama iako se u memoriji čuva ta skrivena vrijednost;
- podklasa ne može naslijediti varijablu ili metodu s istim imenom (sakrivanje podataka). Ukoliko je originalna metoda deklarirana kao statička, tada i preopterećena metoda mora biti statička;
- konstruktore nije moguće naslijediti.

5.5 Načini pozivanja konstruktora kod nasljeđivanja

Instanciranje objekta neke podklase pokreće lanac pozivanja konstruktora u kojem konstruktor podklase prije izvršavanja svojih zadataka poziva konstruktor svoje direktne osnovne klase eksplicitno korištenjem ključne riječi `super` ili implicitno pozivanjem zadanog konstruktora. Isto tako, ako osnovna klasa koju nasljeđuje podklasa ima svoju osnovnu klasu, poziva se i njezin konstruktor i tako za sve ostale slučajeve ovisno o hijerarhiji osim za klasu `Object`.

```
public class Zaposlenik {
        Zaposlenik(){ System.out.println("Zaposlenik konstr."); }
}
public class TerenskiRadnik extends Zaposlenik{
        TerenskiRadnik(){ System.out.println("TerenskiRadnik konstr."); }
}
public class proba{
    public static void main(String [] args){
        TerenskiRadnik radnik1 = new Terenski_radnik();
    }
}
```

Primjer kôda 5.12 Pozivanje konstruktora.

Za zadani primjer kôda 5.12 dobit ćemo sljedeći izlaz u konzoli:

```
Zaposlenik konstr.
TerenskiRadnik konstr.
```

Napomena: Svaki korisnički instancirani objekt u Javi automatski sadrži i dio Java `Object` ugrađene klase. To znači da ako imamo klasu `Klasa1` kao osnovnu klasu i klasu `Klasa2` kao podklasu klase `Klasa1`, prilikom instanciranja objekta `Klasa2` prvo se poziva njen vlastiti konstruktor, koji odmah nakon toga poziva konstruktor nadkalse `Klasa1`, a ovaj odmah poziva konstruktor klase `Object`. Kada konstruktor klase `Object` završi, vraća se kontrola na konstruktor klase `Klasa1` koji do kraja izvršava svoje preostale naredbe i na kraju se vraća kontrola na konstruktor klase `Klasa2`. Ovo znači da se pozivaju ukupno tri konstruktora.

Nakon deklariranja klase, ukoliko se eksplicitno ne navede da ta klasa nasljeđuje neku od klasa, tada ta klasa postaje nasljednik klase `Object`.

```
public class Zivotinja{
...
}
// ILI
public class Zivotinja extends Object{
...
}
```

Primjer kôda 5.13 Nasljeđivanje klase `Object` (implicitno i eksplicitno)

5.5.1 Ključna riječ `super`

Ključna riječ `super` odnosi se na osnovnu klasu, a koristi se u situacijama kao što su:

- pozivanje konstruktora osnovne klase;
- pozivanje metoda roditeljske klase.

U sljedećem primjeru pozabavit ćemo se s prvim slučajem upotrebe ključne riječ `super` dok ćemo ostale primjere raditi u nastavku.

Do sada smo se upoznali s načinima stvaranja klasa i instanciranjem objekata tih klasa. Također, upoznali smo se s pretpostavljenim i ostalim konstruktorima. Kod nasljeđivanja postoji razlika u instanciranju konstruktora ovisno o tome ima li klasa u sebi samo pretpostavljeni konstruktor, neki drugi konstruktor ili oboje.

Ako klasa sadrži samo pretpostavljeni konstruktor, tada se on prilikom instanciranja objekta podklase poziva automatski (iako ga i mi možemo eksplicitno pozvati).

Ako klasa ima deklariran neki drugi konstruktor, tada podklasa mora sadržavati barem jedan konstruktor koji će pozvati konstruktor osnovne klase (korištenje ključne riječi `super`).

```
public class Zaposlenik {
        private int brojZaposlenika;
        private double placaZaposlenika;
        Zaposlenik(int argument){
                brojZaposlenika = argument;
                System.out.println("Zaposlenik konstr.");
        }
        public int dohvatiBrojZaposlenika(){
                return brojZaposlenika;
        }
        public double dohvatiPlacuZaposlenika(){
                return placaZaposlenika;
        }
}
public class TerenskiRadnik extends Zaposlenik{
        TerenskiRadnik(){
                super(3);
                System.out.println("TerenskiRadnik konstr.");
        }
}
```

Primjer kôda 5.14 Korištenje ključne riječi `super` (eksplicitno).

Poseban slučaj kada se ne mora pozivati konstruktor osnovne klase, čak i kada ima deklariran jedan ili više konstruktora, je kada ima deklariran pretpostavljeni konstruktor.

```
public class Zaposlenik {
        Zaposlenik(){
                //pretpostavljeni konstruktor;
        }
        Zaposlenik(int argument){
                brojZaposlenika = argument;
                System.out.println("Zaposlenik konstr.");
        }
...
}
```

Primjer kôda 5.15 Korištenje ključne riječi `super` (implicitno).

5.6 Zadaci

Zadatak 5.1. Napravi klasu `Sisavac` s dvijema privatnim članskim varijablama: `dob` i `tezina`. Klasa treba imati konstruktor koji postavlja obje članske varijable na 0 i metode za dohvatiti i postaviti članske varijable (tzv. gettere i settere).
Treba napraviti klase `Pas` i `Macka` koje obje nasljeđuju klasu `Sisavac` i imaju još dodatnu privatni člansku varijablu: `vrsta` u koju se zapisuje pasmina psa ili mačke. Obje klase imaju metode za postavljanje i dohvaćanje vrste, a imaju i metodu za opisivanje koja ispisuje koliko je teška, koliko je stara, koje je vrste. Napravite klasu `Test`, kreirajte objekte sisavca, psa i mačke, postavite im članske varijable i pozovite metode za opis psa i mačke.

Zadatak 5.2. Razdvojite klasu `Sisavac` i klase `Pas` i `Macka` u različite pakete. Pokažite razliku u nasljeđivanju s `protected` pristupnim modifikatorom (pokažite kako on utječe na pristup varijablama kada je naslijeđena klasa u drugom paketu).

Poglavlje 6

Polimorfizam

Polimorfizam predstavlja koncept programiranja pomoću kojeg se jedna radnja može izvesti na više različitih načina. Omogućava općeniti način programiranja umjesto specifičnog načina programiranja. Riječ polimorfizam izvodi se iz dviju grčkih riječi (poly – više i morphe – oblik), što znači više oblika.

Polimorfizam u Java programskom jeziku može se izvoditi pomoću preopterećivanja metoda (engl. *overloading*) i nadjačavanja metoda (engl. *overriding*).

Postoje dvije vrste polimorfizma:

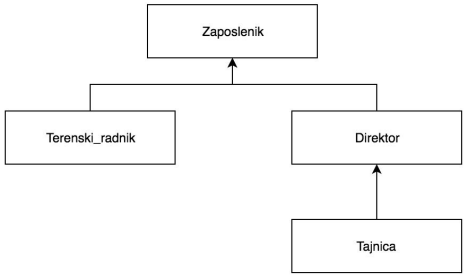
- statički polimorfizam – polimorfizam u vremenu prevođenja;
- dinamički polimorfizam – polimorfizam u vremenu izvršavanja.

Preopterećivanje metoda primjer je statičkog polimorfizma koji omogućava definiranje metoda istog imena s različitim brojem, tipom ili redoslijedom parametara (unutar iste klase).

```
public class Zaposlenik{
        private void dodajZaposlenika(){
        }
        private void dodajZaposlenika(boolean odredjeno){
        }
}
public class Poziv{
        public static void main( String[] args){
                Zaposlenik radnik1 = new Zaposlenik();
                radnik1.dodajZaposlenika();
        radnik1.dodajZaposlenika(true);
        }
}
```

Primjer kôda 6.1 Statički polimorfizam.

Nadjačavanje metoda primjer je dinamičkog polimorfizma koji omogućava dodatnu specifikaciju roditeljskih metoda unutar izvedene klase (preopterećivanje kroz više klasa). Objekt podklase može se tretirati kao objekt svoje roditeljske klase.



Slika 6.1: Primjer nasljeđivanja.

```
public class Direktor extends Zaposlenik{
        public void potpisiDokument () {
                System.out.println("Direktor potpisuje"+" dokument");
        }
}
public class Tajnica extends Direktor{
        public void potpisiDokument () {
                System.out.println("Tajnica potpisuje"+" dokument");
        }
}
public class Firma{
        public static void main(String[] args){
                Direktor direktor = new Direktor();
                Tajnica tajnica = new Tajnica();
                direktor.potpisiDokument();
                tajnica.potpisiDokument();
                Direktor ob = new Tajnica();
                ob.potpisiDokument();
                direktor = tajnica;
                direktor.potpisiDokument();
        }
}
```

Primjer kôda 6.2 Dinamički polimorfizam.

Kada varijabla tipa roditeljske klase (`direktor`) sadrži referencu na objekt podklase i kada se ta referenca koristi za pozivanje metode, poziva se metoda podklase. U trenutku izvođenja tip objekta na koji se odnosi varijabla određuje koja će se metoda pozvati, što se još naziva i dinamičko povezivanje. Za primjer kôda 6.2 dobit ćemo sljedeći izlaz u konzoli:

```
Direktor potpisuje dokument
Tajnica potpisuje dokument
Tajnica potpisuje dokument
Tajnica potpisuje dokument
```

Razlike između preopterećivanja i nadjačavanja dane su u tablici 6.1.

Tablica 6.1: Razlike između preopterećivanja i nadjačavanja.

Preopterećivanje	**Nadjačavanje**
Koristi se za povećanje čitljivosti programskog kôda.	Koriste se za stvaranje detaljnije specifikacije metode koja se već nalazi u roditeljskoj klasi.
Događa se unutar klase.	Obavlja se u dvijema klasama koje imaju IS-A odnos (nasljeđivanje).
Parametri metoda moraju biti različiti.	Parametri metoda moraju biti isti.
Povratni tip metode može ostati jednak ili se može promijeniti.	Povratni je tip najčešće isti, ali može biti i specijalizacija povratnog tipa u roditeljskoj klasi.

Postoji tri vrste metoda koje nije moguće nadjačati:

- statičke metode;
- metode označene modifikatorom `final` (zaključane za nadjačavanje);
- metode definirane u klasi koja je označena ključnom riječi `final` (zaključane za nasljeđivanje).

Statičke metode

Nije moguće nadjačati metodu koju je moguće pozvati bez instanciranja objekta. Moguće je deklarirati statičku metodu s istim potpisom unutar podklase, ali nije moguće pozvati istu tu metodu unutar nadklase čak ni uz korištenje ključne riječi `super`.

```
public class Klasa1 {
        private int var1;
        private double var2;
        public static void funkcija(){
                //tijelo funkcije
        }
}
//primjer 1
public class Klasa2 extends Klasa1 {
        double var1;

        public void funkcija(){ //nije moguce napraviti
                super.funkcija();
                System.out.println("Ispis...");
        }
}
//primjer 2
public class Klasa2 extends Klasa1 {
        double var1;

        public static void funkcija(){
                super.funkcija();  // nije moguce napraviti
                System.out.println("Ispis...");
        }
}
//primjer 3
public class Klasa2 extends Klasa1 {
        double var1;
        public static void funkcija(){
                Klasa1.funkcija();
                System.out.println("Ispis...");
        }
}
```

Primjer kôda 6.3 Nadjačavanje static metode.

Metode označene modifikatorom `final`

Nije moguće nadjačati metodu roditeljske klase ako je ona deklarirana kao `final`, drugim riječima, ključnu riječ `final` prilikom deklaracije metode koristimo kada želimo da svaka podklasa koristi izvorno deklariranu metodu unutar roditeljske klase.

Prednost deklariranja metode označene modifikatorom `final` u konačnici je brže izvođenje kôda, a razlog tomu leži u načinu pozivanja metoda (engl. *virtual method calls*).

```
public class Klasa1 {
        private int var1;

        public final void ispis() {
                System.out.println("Ispis...");
        }
}
public class Klasa2 extends Klasa1 {
        double var1;
        public void ispis(){ //nije moguce napraviti
                System.out.println("Ispis...");
        }
}
```

Primjer kôda 6.4 Metoda označena modifikatorom `final` – primjer.

Metode definirane u klasi koja je označena ključnom riječi `final`

Kada se klasa deklarira pomoću ključne riječi `final`, tada sve metode unutar te klase dobivaju modifikator `final` bez obzira na pristupni modifikator kojeg dodijeli programer. S obzirom na to, takva klasa ne može biti nadklasa niti jednoj drugoj klasi.

```
public final class Klasa1{
        private int var1;
        public void ispis(){
                System.out.println("Ispis...");
        }
}
//nije moguce napraviti
public final class Klasa2 extends Klasa1{
        private double var;
}
```

Primjer kôda 6.5 Metoda definirana u klasi koja je označena ključnom riječi `final`.

6.1 Apstraktne klase

Apstraktna klasa predstavlja klasu koju je moguće naslijediti, ali ne i instancirati objekte. Može sadržavati jednu ili više apstraktnih metoda koje su sadržajno prazne.
Kako bi se naznačilo da se radi o apstraktnoj klasi ili metodi koristi se ključna riječ abstract.

Nakon stvaranja podklase apstraktne klase potrebno je napisati metode s istim potpisom unutar podklase. Moguće je da te metode budu također apstraktne; inače je potrebno napisati implementaciju.

```
public abstract class Zivotinja {
        private String ime;
        public abstract void glasaj();
        public String dohvatiIme(){
                return ime;
        }
        public void postaviIme(String imeZivotinje){
                ime = imeZivotinje;
        }
}
```

Primjer kôda 6.6 Apstraktna klasa.

Kako iz apstraktne klase nije moguće instancirati objekt, sljedeći izraz bio bi neispravan:

```
Zivotinja ljubimac = new Zivotinja("Miki");
```

Nasljeđivanje klase `Zivotinja` bilo bi kako slijedi:

```
public class Pas extends Zivotinja{
        public void glasaj(){
                System.out.println("Wau wau");
        }
}
public class Macka extends Zivotinja{
        public void glasaj(){
                System.out.println("Mijauuu");
        }
}
```

Primjer kôda 6.7 Nasljeđivanje apstraktne klase.

Vidljivo je da je metoda `glasaj()` obvezna u obama primjerima iz razloga što je metoda deklarirana kao `abstract` u roditeljskoj klasi.

U slučaju da se ne osigura implementacija metode `glasaj()` u podklasama nije moguće instancirati objekt te klase. U tom slučaju i podklasa bi se trebala proglasiti apstraktnom, što znači da bi se metoda `glasaj()` morala implementirati u klasi koja ju naslijedi.

6.2 Sučelja

Sučelje (engl. *interface*) predstavlja opis onoga što klasa radi, ali ne i kako radi. Dolazi kao zamjena za višestruko nasljeđivanje koje je omogućeno primjerice u programskom jeziku C++. Za korištenje sučelja koristi se ključna riječ `implements`. Upotrebom ključne riječi `extends` podklasi je dozvoljeno korištenje članova koji nisu privatni ili nadjačani, dok ključna riječ `implements` zahtijeva vlastitu implementaciju svake metode ili u nekoj od podklasa ove podklase.

```
public abstract class Zivotinja {
        private String ime;
        public abstract void glasaj();
        public String dohvatiIme(){
                return ime;
        }
        public void postaviIme(String imeZivotinje){
                ime = imeZivotinje;
        }
}
public class Pas extends Zivotinja{
        public void glasaj(){
                System.out.println("Wau wau");
        }
}
public interface Radni{
        public void radi();
}
public class RadniPas extends Pas implements Radni {
        private int satiTreninga;
        public void postaviSateTreninga(int sati){
                satiTreninga = sati;
        }
        public int dohvatiSateTreninga(){
                return satiTreninga;
        }
        public void radi(){
                glasaj();
                System.out.println("Ja sam pas koji radi");
                System.out.println("Imam " + satiTreninga +"h treninga");
        }
}
```

Primjer kôda 6.8 Deklariranje i korištenje sučelja.

Osim za deklariranje apstraktnih metoda, kao što je to prikazano u primjeru kôda 6.8, sučelje može služiti i kao skladište statičkih podataka kao što je to prikazano u primjeru kôda 6.9.

```
public interface PizzaConstants{
    public static final int mala = 12;
    public static final int velika = 16;
    public static final double porez = 1.25;
    public static final String restoran = "Pizzeria";
}

public class PizzaDemo implements PizzaConstants {
    public static void main(String[] args){
```

```
        double cijena = 40.00;
        System.out.println("Dobrodosli u" + Restoran);
        System.out.println("Trenutno imamo ponudu:\n "
        + mala + " cm pizza sa 3 sastojka\n ili "
        + velika + " cm pizza sa 6 sastojaka\n za " + cijena+" kn");
        System.out.println("Sa porezom to je" + (cijena + cijena * porez));
    }
}
```

Primjer kôda 6.9 Korištenje sučelja za spremanje konstantnih varijabli.

Važne su razlike između apstraktne klase i sučelja:

- apstraktna klasa može sadržavati metode koje nisu apstraktne dok kod sučelja sve metode moraju biti apstraktne (napomena: u novijim verzijama Jave ovo je izmijenjeno pa apstraktna klasa može sadržavati statičke, privatne i pretpostavljene metode koje imaju implementaciju ponašanja);
- klasa može naslijediti samo jednu apstraktnu klasu koristeći ključnu riječ `extends`, dok sučelja može implementirati u neograničenom broju.

Napomena: Apstraktna klasa kao niti sučelje ne dozvoljava instanciranje objekata.

6.3 Zadaci

Zadatak 6.1. Napisati klasu `Zivotinja` koja ima člansku varijablu `protected String vrsta` i metodu `kreciSe` koja ispisuje `"zivotinja se krece"`. Klase `Konj`, `Riba` i `Ptica` nasljeđuju klasu `Zivotinja` i svaka u konstruktoru postavlja vrijednost članske varijable `vrsta` klase `Zivotinja` na svoju vrstu. Također, svaka klasa nadjačava metodu `kreciSe` svojom specifičnom porukom (npr. `"trcim"`, `"plivam"`, `"letim"`). U metodi `main` klase `Test` napraviti objekte svih klasa i isprobati metode kretanja. BONUS: napraviti niz od 4 elemenata i napuniti ga s jednim objektom svake od gore navedenih klasa. Napraviti petlju koja će proći kroz niz i pokrenuti metodu `kreciSe` na svakom od objekata niza.

Zadatak 6.2. Napisati klasu `Zaposlenik`, `Direktor`, `Tajnica` i `Vozac`. Klasa `Zaposlenik` je apstraktna i sadrži članske varijable tipa `private String`: `ID`, `ime` i `prezime`. Sadrži metode za postavljanje i dohvaćanje tih parametara (`get/set` metode) i metode `predstaviSe` i `radi`. Metoda `predstaviSe` ispisuje podatke o zaposleniku (`ime`, `prezime` i `ID`). Metoda `radi` nema implementaciju, prazna je. Klasa `Direktor`, `Tajnica` i `Vozac` nasljeđuju klasu `Zaposlenik` i imaju metodu `radi` u kojoj opisuju način na koji rade (npr. `"Odlucujem"`, `"Dogovaram"`, `"Vozim"` ili nešto slično). Klasa `Direktor` nadjačava metodu `predstaviSe` na način da poziva originalnu metodu klase `Zaposlenik` (ključna riječ `super`) i u novom retku ispisuje svoj tip objekta (koristeći metodu `getSimpleName()` klase `Class`). U metodi `main` klase `Test` napraviti sve objekte i isprobati metode. BONUS: Modificiraj metodu `predstaviSe` klase `Zaposlenik` tako da prima objekt bilo kojeg tipa od klasa koje ju nasljeđuju (`Direktor`, `Tajnica` ili `Vozac`) i onda ispisuje tip tog objekta (koristeći metodu `getSimpleName()` klase `Class`), slično kao što je to radila metoda `predstaviSe` u klasi `Direktor`.

Zadatak 6.3. Napisati implementaciju apstraktne klase `Shape` koja ima jednu metodu `draw()` te pripadajućih klasa `Circle` i `Rectangle` koje nasljeđuju klasu `Shape`. U klasi `Test` instancirati objekte klasa `Circle` i `Rectangle` te pozvati metodu `draw()`. (metoda `draw()` treba samo na ekran ispisivati niz znakova koji predstavljaju radnju npr. `"Drawing circle"` ili `"Drawing rectangle"`).

Zadatak 6.4. Iskopiraj i modificiraj klase iz zadatka 6.1. tako da osnovna klasa (`Zivotinja`) ne bude klasa, nego sučelje. Definiraj sučelje `Radni` s metodom `radi`. Klasa `Konj` neka implementira i to sučelje. U metodi `main` klase `Test` napraviti objekte svih klasa i isprobati metode. BONUS: Modificiraj klase iz zadatka 6.2. tako da osnovna klasa (`Zaposlenik`) ne bude klasa, nego sučelje. U metodi `main` klase `Test` napraviti objekte svih klasa i isprobati metode.

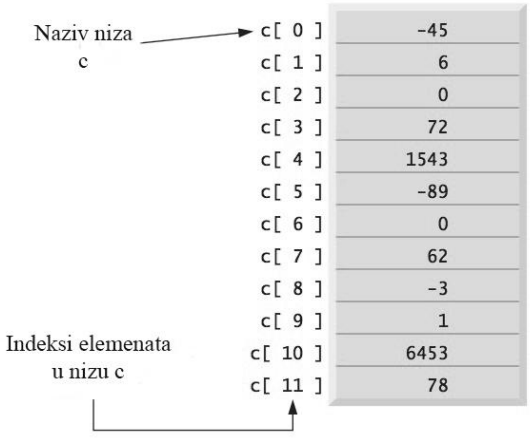
Poglavlje 7

Polja

Polja predstavljaju strukturu podataka koja se sastoji od memorijski uzastopnog niza podataka istog tipa. Koriste se za obradu povezanih grupa podataka. Jednom stvoreno polje ostaje iste veličine kroz cijeli programski kôd s tim da se naziv varijable koja je po tipu referenca i pokazuje na polje može preusmjeriti na drugo polje različite veličine. Elementi polja mogu biti primitivni tipovi kao i reference. Jednodimenzionalna polja nazivaju se još i *nizovi* ili *vektori*. U nastavku će se za jednodimenzionalna polja koristiti kraći naziv – niz.
Pozicija na kojoj se nalazi neki element niza naziva se indeks. Sintaksa kojom se deklarira referenca koja je po tipu niz je kako slijedi:

```
<tip> [] <naziv_niza>;
```

Na slici 7.1 prikazan je primjer niza:



Slika 7.1: Prikaz niza.

Iz slike 7.1 može se vidjeti da se vrijednosti koje može poprimiti indeks kreću od `0` do `n-1` (gdje je `n` broj elemenata niza odnosno duljina niza). Također, vidljivo je da indeks mora biti nenegativan broj.

Primjeri deklariranja niza:

```
//1
int [] niz = new int [10];
//2
int [] niz;
niz = new int [10];
//3
String [] s = new String[10];
String [] s1 = new String[10], s2 = new String[10];
//4
String niz[] = new String[10];
```

Primjer kôda 7.1 Primjeri deklaracije niza.

Niz možemo deklarirati na dva načina ovisno o tome koliko nizova istog tipa želimo deklarirati u istom redu. U primjeru kôda 7.1 možemo vidjeti da se uglate zagrade nalaze odmah iza tipa podatka koji će se spremati u niz. Ako pišemo deklaraciju na taj način, možemo deklarirati jedan ili više nizova u istom redu kao što je vidljivo pod 3 u primjeru kôda 7.1. Ako niz deklariramo na način da uglate zagrade stavimo iza naziva niza, kao što je to prikazano u načinu 4, tada je moguće deklarirati samo jedan niz. Na sljedećem primjeru prikazat ćemo način na koji se unose vrijednosti u niz kao i način na koji se iščitavaju vrijednosti iz niza:

```
public class Klasa1{
    public static void main(String [] args){
        int niz[] = new int[10];
        Scanner input = new Scanner(System.in);
        //unos elemenata u niz
        for(int i=0; i<10;i++) niz[i]=input.nextInt();
        //ispis unesenih elemenata iz niza
        for(int i=0; i<10;i++)
            System.out.println("Element ["+i+"] = "+niz[i]);
    }
}
```

Primjer kôda 7.2 Unos i ispis elemenata iz niza.

Za unesene elemente {2,4,6,8,10,12,14,16,18,20} dobit ćemo sljedeći izlaz u konzoli:

```
Element [0] = 2
Element [1] = 4
Element [2] = 6
Element [3] = 8
Element [4] = 10
Element [5] = 12
Element [6] = 14
Element [7] = 16
Element [8] = 18
Element [9] = 20
```

Međutim, unos elemenata u niz, kao i iščitavanje istih, može se obaviti i na druge načine. Jedan je od načina čitanja elemenata niza petlja `for-each` koja predstavlja skraćenu verziju petlje `for`. Unutar bloka naredbi petlje `for-each` s lijeve strane stavljamo tip podatka koji se nalazi u nizu (ili kolekciji) i u koji će se spremati element po element kako budemo prolazili kroz niz, dok se s desne strane stavlja skup elemenata kroz kojeg prolazimo.

```
//pretpostavimo da postoji popunjeni niz naziva "niz"
for(int broj : niz){
        //ispis elementa niza
        System.out.println(broj);
}
```

Primjer kôda 7.3 Upotreba petlje `for-each`.

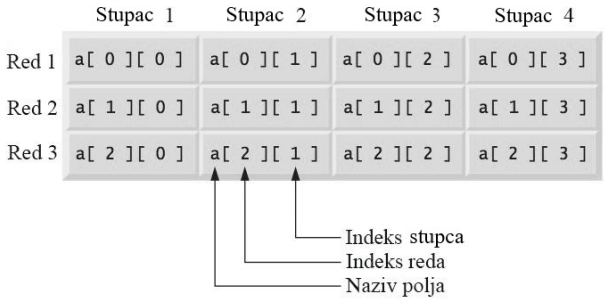
Osim dinamičkog unosa elemenata u niz, vrijednosti je moguće inicijalizirati prilikom same deklaracije niza:

```
int [] niz = {2,4,6,8,10,12,14,16}
```

7.1 Dvodimenzionalna polja

U primjerima do sada spominjali su se samo jednodimenzionalna polja. Osim jednodimenzionalnog polja upoznat ćemo se i s dvodimenzionalnim poljima koja se kraće nazivaju *matrice*.

Na slici 7.2 možemo vidjeti strukturu dvodimenzionalnog polja. Ono što odmah možemo primijetiti dodatni je indeks u usporedbi s jednodimenzionalnim poljem, odnosno u dvodimenzionalnom polju



Slika 7.2: Prikaz dvodimenzionalnog polja – matrice.

imamo jedan indeks koji označava red i drugi koji označava stupac. Npr. element `a[0][0]` označava element koji se nalazi u prvom retku i prvom stupcu.

Primjeri deklariranja dvodimenzionalnog polja:

```
int[][] polje = new int [2][2];
int[][] polje = { {2,4,6}, {1,3,5} }; //inicijalizacija prilikom deklaracije
```

Primjer kôda 7.4 Deklaracija matrice.

```
public class Klasa1{
    public static void main(String [] args){
        int polje[][] = new int[2][3];
        Scanner input = new Scanner(System.in);
        //unos elemenata u polje
        for(int i=0; i<2;i++){
            for(int j=0;j<3;j++){
                System.out.println("Unesi element za polje["+i+"]["+j+"]");
                polje[i][j]=input.nextInt();
            }
        }
        //ispis elemenata iz dvodimenzionalnog polja
        for(int i=0; i<2;i++){
            for(int j=0;j<3;j++){
                System.out.print(polje[i][j]+ " ");
            }
            //dodajemo novi red koji ce razdvajati retke u matrici
            System.out.println();
        }
    }
}
```

Primjer kôda 7.5 Unos i ispis elemenata matrice.

Primjer kôda 7.5 za unesene elemente {{2,4,6}, {1,3,5}} dobit ćemo sljedeći izlaz u konzoli:

```
2 4 6
1 3 5
```

Prosljeđivanje niza u metodu

U određenim slučajevima niz je potrebno poslati na obradu u neku od definiranih metoda. To ćemo učiniti na sljedeći način:

```
public class Klasa1{
public static void kvadriranje(int[] niz){
    for(int i=0;i<niz.length;i++) niz[i]*=niz[i];
}
    public static void main(String [] args){
```

```
        int niz[] = new int[10];
        Scanner input = new Scanner(System.in);
        //unos elemenata u niz
        for(int i=0; i<10;i++) niz[i]=input.nextInt();
        //poziv funkcije i predaja parametra
        kvadriranje(niz); //pozivanje funkcije "kvadriranje"
        for(int i=0; i<10;i++)
            System.out.println("Element ["+i+"] = "+niz[i]);
    }
}
```

Primjer kôda 7.6 Prosljeđivanje niza u metodu.

Za unesene elemente {2,4,6,8,10,12,14,16,18,20} dobit ćemo sljedeći izlaz u konzoli:

```
Element [0] = 4
Element [1] = 16
Element [2] = 36
Element [3] = 64
Element [4] = 100
Element [5] = 144
Element [6] = 196
Element [7] = 256
Element [8] = 324
Element [9] = 400
```

7.2 Klasa `Arrays`

Klasa `Arrays` predstavlja skupinu statičkih metoda koje olakšavaju rad s nizovima. Iako klasa `Arrays` sadrži mnoštvo korisnih metoda, ovdje ćemo prikazati samo neke.

Tablica 7.1: Metode klase Arrays.

Naziv metode	Broj argumenata	Opis
`sort()`	1	Sortira elemente niza.
`fill()`	2	Popunjava niz prethodno definiranim vrijednostima.
`equals()`	2	Uspoređuje dva niza.
`binarySearch()`	2	Pretražuje niz (mora biti sortirano).

```
public static void main(String [] args){
    //inicijalizacija nizova
    int dp[] = {2,1,8,4,10,6};
    int ip [] = new int[6];
    Arrays.sort(dp); //sortiranje niza
    //ispis niza
    for(int x : dp){ System.out.print(x+" "); }
    System.out.println();
    Arrays.fill(ip,1); //popunjavanje niza predefiniranom vrijednosti (1)
    //ispis niza
    for(int x : ip){ System.out.print(x+" "); }
    System.out.println();
    boolean p = Arrays.equals(dp,ip); //provjerava jednakost dva niza
    int indeks = Arrays.binarySearch(dp,3); //primjena binarnog pretr. niza
    System.out.println(p);
    System.out.println(indeks);
}
```

Primjer kôda 7.7 Korištenje ugrađenih metoda klase `Arrays`.

Za navedene nizove `dp` i `ip` u primjeru kôda 7.7 dobit ćemo sljedeći izlaz u konzoli:

```
1 2 4 6 8 10
1 1 1 1 1 1
false
4
```

7.3 Rad sa znakovima i stringovima

Ako pogledamo širu sliku, znakovi su temeljni gradivni blokovi svakog programa u Javi. Drugim riječima, svaki program u Javi sastoji se od niza znakova koji se zatim interpretiraju u prevoditelju koji onda tvore skup instrukcija koje se izvršavaju na računalu. Znak (engl. *character*) se može zadati kao cjelobrojna vrijednost (redni broj u skupu kodiranja) ili kao znakovna konstanta unutar jednostrukih navodinka ili literala (`'a'`, `'b'`, `'c'`).

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

7.4 Klasa `Character`

Klasa `Character` predstavlja jednu od dostupnih ugrađenih klasa (dostupne su još klase `Double`, `Float`, `Byte`, `Short`, `Integer` i `Long`) unutar jezika Java, čija je svrha predstavljanje primitivnih tipova podataka kao objekata. Većina metoda unutar klase `Character` su statičke radi jednostavnosti korištenja, a kao argument primaju znak nad kojim se onda vrši manipulacija. Također, klasa `Character` sadrži konstruktor, koji u svrhe inicijalizacije objekta, prihvaća znak kao argument. Neke od najčešće korištenih metoda iz klase `Character` prikazane su u tablici 7.2.

Tablica 7.2: Metode klase `Character`.

Naziv metode	Broj argumenata	Opis
`isDefined()`	1	Provjerava je li znak definiran unutar Unicode znakovnog skupa.
`isDigit()`	1	Provjerava je li znak definiran kao broj.
`isLetter()`	1	Provjerava je li znak definiran kao slovo.
`isLetterOrDigit()`	1	Provjerava je li znak slovo ili broj.
`isLowerCase()`	1	Provjerava je li znak malo slovo.
`isUpperCase()`	1	Provjerava je li znak veliko slovo.
`toLowerCase()`	1	Pretvara znak u malo slovo.
`toUpperCase()`	1	Pretvara znak u veliko slovo.

```
public static void main(String [] args){
        Scanner input = new Scanner(System.in);
        System.out.println("Unesite znak:");
```

```
        String ulaz = input.next();
        char znak = ulaz.charAt(0);
        System.out.println(Character.isDefined(znak));
        System.out.println(Character.isDigit(znak));
        System.out.println(Character.isLetter(znak));
        System.out.println(Character.isLetterOrDigit(znak));
        System.out.println(Character.isLowerCase(znak));
        System.out.println(Character.isUpperCase(znak));
        System.out.println(Character.toUpperCase(znak));
        System.out.println(Character.toLowerCase(znak));
}
```

Primjer kôda 7.8 Korištenje ugrađenih metoda klase `Character`.

Za navedeni ulaz u primjeru kôda 7.8 dobit ćemo sljedeći izlaz u konzoli:

```
Unesite znak:
A
true
false
true
true
false
true
A
a
```

7.5 Klasa `String`

Klasa `String` koristi se za reprezentaciju stringova u programskom jeziku Java. Klasa `String` ima nekoliko konstruktora koje možemo koristiti.

```
char [] niz = {'a','b','c','d','e','f',' ','g','h','i'};
String s = new String("rijec");
// instanciranje novog String objekta koristeci konstruktor bez argumenata
String s1 = new String();
// instanciranje novog String objekta prosljedjivanjem
// vec postojeceg String objekta
String s1 = new String(s);
//instanciranje novog String objekta iz polja znakova
String s1 = new String(niz);
// instanciranje novog String objekta iz polja znakova pri tome
// specificirajuci od kojeg znaka treba krenuti i koliko znakova
// treba uzeti u obzir (drugi argument je pocetni indeks, treci argument
// predstavlja zeljeni broj znakova)
String s1 = new String(niz,7,3);
```

Primjer kôda 7.9 Ugrađeni konstruktori klase `String`.

Osim konstruktora, važno je navesti s kojim sve metodama raspolaže klasa `String`, a koje se odnose na manipuliranje stringovima (metode `length`, `charAt` i `getChars`).

Tablica 7.3: Metode klase Character.

Naziv metode	Broj argumenata	Opis
length()	0	Vraća duljinu stringa.
charAt()	1	Vraća znak na određenom mjestu u stringu.
getChars()	4	Kopira znakove iz stringa u niz znakova.

```
String s = new String("Nekakav string");
char[] nizZnakova = new char[5];
System.out.println("Duljina stringa je: "+s.length());
System.out.println("Znak koji se nalazi na 4 mjestu je: "+s.charAt(4));
// metoda getChars()
// Prvi parametar - pocetni indeks od kojeg se krece u stringu.
// Drugi parametar - indeks do kojeg se vrsi kopiranje
//    (prva granica se ukljucuje, a druga ne ukljucuje u kopiranje)
// Treci parametar - niz u kojeg kopiramo znakove iz stringa.
// Cetvrti parametar - indeks od kojeg krece spremanje znakova
//                     koje dohvacamo iz stringa.
s.getChars(0,5,nizZnakova,0);
for(char c : nizZnakova) System.out.print(c);
```

Primjer kôda 7.10 Ugrađene metode klase String.

Za navedeni ulaz u primjeru kôda 7.10 dobit ćemo sljedeći izlaz u konzoli:

```
Duljina stringa je: 14
Znak koji se nalazi na 4 mjestu je: k
Nekak
```

7.5.1 Usporedba stringova

Tablica 7.4: Metode klase String.

Naziv metode	Broj argumenata	Opis
equals()	1	Uspoređuje sadržaj objekata.
equalsIgnoreCase()	1	Uspoređuje sadržaj neovisno o velikom/malom slovu.
startsWith()	1	Uspoređuje počinje li string s određenim skupom znakova.
endsWith()	1	Uspoređuje završava li string s određenim skupom znakova.
compareTo()	1	Leksikografski uspoređuje string s onim u parametru.

```
public static void main(String [] args){
    String s1 = new String("rijec");
    String s3 = "Neka Rijec";
    String s4 = "neka rijec";
    // metoda equals() usporedjuje sadrzaj objekata koristeci
    // leksikografsku usporedbu (usporedba integer vrijednosti iz Unicode
    // tablice) i ukoliko sadrzaji objekata nisu isti vraca "false".
    if(s1.equals("rijec")){
        System.out.println("#1 s1 jednak je \"rijec\"");
    }
    // usporedba koristeci operator jednakosti. U navedenom slucaju
    // rezultat ce biti "false" jer se usporedjuje String objekt stvoren
```

```
        // na osnovu String literala sa String objektom stvorenim s new
        // Kada usporedjujemo dva String literala na ovaj
    // nacin, "true" ce biti samo u slucaju kada su obje vrijednosti
    // identicne. Ukoliko usporedjujemo dva String objekta, rezultat ce
    // biti "true" samo ako se reference odnose na iste memorijske adrese.
    if(s1=="rijec"){
        System.out.println("#2 s1 jednak je \"rijec\"");
    }
    if(s3.equalsIgnoreCase(s4)){
        System.out.println("s3 i s4 su jednaki");
    }
    if(s1.startsWith("ri")){
        System.out.println( "Prva 2 znaka stringa s1 jesu \"ri\"");
    }
    if(s1.endsWith("ri")){
        System.out.println( "Zadnja 2 znaka stringa s1 jesu \"ri\"");
    }
    if(s1.compareTo("auto")>0){
        System.out.println( "\"auto\" je leksikografski prije \"rijec\"");
    }
}
```

Primjer kôda 7.11 Ugrađene metode klase `String`.

Za navedeni ulaz u primjeru kôda 7.11 dobit ćemo sljedeći izlaz u konzoli:

```
#1 s1 jednak je "rijec"
s3 i s4 su jednaki
Prva 2 znaka stringa s1 jesu "ri"
"auto" je leksikografski prije "rijec"
```

7.5.2 Dohvaćanje podstringova

Klasa `String` u sebi sadrži dvije metode za stvaranje podstringova iz već postojećih stringova. Rezultat obiju funkcija novi je objekt tipa `String`.

Tablica 7.5: Metode klase String.

Naziv metode	Broj argumenata	Opis
`substring()`	1	Vraća podstring sa svim znakovima od navedenog indeksa pa do kraja.
`substring()`	2	Vraća podstring sa svim znakovima između početnog i krajnjeg indeksa (ne uključujući zadnji indeks).

```
public static void main(String [] args){
        String slova = "abcdefghijklmnopqrstuvwxyz";
        System.out.println(slova.substring(25));
        System.out.println(slova.substring(8,13));
}
```

Primjer kôda 7.12 Ugrađene metode klase `String`.

Za navedeni ulaz u primjeru kôda 7.12 dobit ćemo sljedeći izlaz u konzoli:

```
z
ijklm
```

7.5.3 Spajanje stringa

Ponekad je potrebno dva ili više stringa spojiti u jednu cjelinu. Za tu svrhu na raspolaganju nam je funkcija `concat()` koja također ne mijenja postojeće stringove, već stvara novi čiji je sadržaj jednak nadovezivanju (konkatenaciji) tih stringova.

```
public static void main(String [] args){
        String s1 = "Dobar ";
        String s2 = "dan!";
        System.out.println(s1.concat(s2));
}
```

Primjer kôda 7.13 Ugrađene metode klase `String`.

Za navedeni ulaz u primjeru koda 7.13 dobit ćemo sljedeći izlaz u konzoli:

```
Dobar dan!
```

7.5.4 Ostale korisne metode

Osim do sada navedenih funkcija iz klase `String`, važno je navesti i neke od ostalih korisnih metode koje se često znaju koristiti, a predstavljaju i ekvivalente sličnih ili istih funkcija u klasi `Character`. Niti jedna od metoda iz tablice 7.6 ne mijenja string nad kojim je pozvana, već stvara novi koji je modificiran u skladu s pozvanom metodom.

Tablica 7.6: Metode klase `String`.

Naziv metode	Broj argumenata	Opis
`replace()`	2	Zamjenjuje znakove u stringu sa zadanim znakom.
`toUpperCase()`	0	Pretvara mala u velika slova.
`toCharArray()`	0	Pretvara string u niz znakova.

```
...
String s1 = "zanzibar";
String s2 = "  razmak     razmak";

// funkcija koja zamjenjuje znakove u stringu sa zadanim znakom
System.out.println(s1.replace('z','Z'));
// pretvara mala u velika slova
System.out.println(s1.toUpperCase());
// pretvara String u niz znakova
char[] niz_znakova = s1.toCharArray();
for(char c : niz_znakova){
        System.out.print(c);
}
```

Primjer kôda 7.14 Ugrađene metode klase `String`.

Za navedeni ulaz u primjeru 7.14 dobit ćemo sljedeći izlaz u konzoli:

```
ZanZibar
ZANZIBAR
zanzibar
```

7.6 Zadaci

Zadatak 7.1. Napisati program koji će izračunati aritmetičku sredinu sporedne dijagonale i sumu neparnih elemenata prvog retka i prvog stupca matrice. Ukoliko ne postoje neparni elementi, ispisati da nije bilo moguće izračunati sumu. Program na početku treba od korisnika zatražiti dimenzije kvadratne matrice (`m`×`m`). Koristeći petlju `do-while` ograničiti unos za broj `m` na interval između 1 i 5. Unos se mora ponavljati sve dok nije unesen broj u traženom intervalu.

Zadatak 7.2. Napisati program koji će za neki proizvoljan pozitivan broj `N` tražiti od korisnika unos brojeva u vektor `V`. Nakon toga, program treba izračunati i na ekran ispisati koji broj ima najveću sumu znamenki.

Zadatak 7.3. Napisati program koji od korisnika traži unos jedne rečenice. Nakon toga program treba ispisati na ekran sljedeće podatke:

a) najdulju riječ u rečenici;

b) najkraću riječ ispisanu obrnuto;

c) broj pojavljivanja brojeva u rečenici;

d) broj istih riječi u rečenici;

e) broj riječi koje završavaju na neki od samoglasnika;

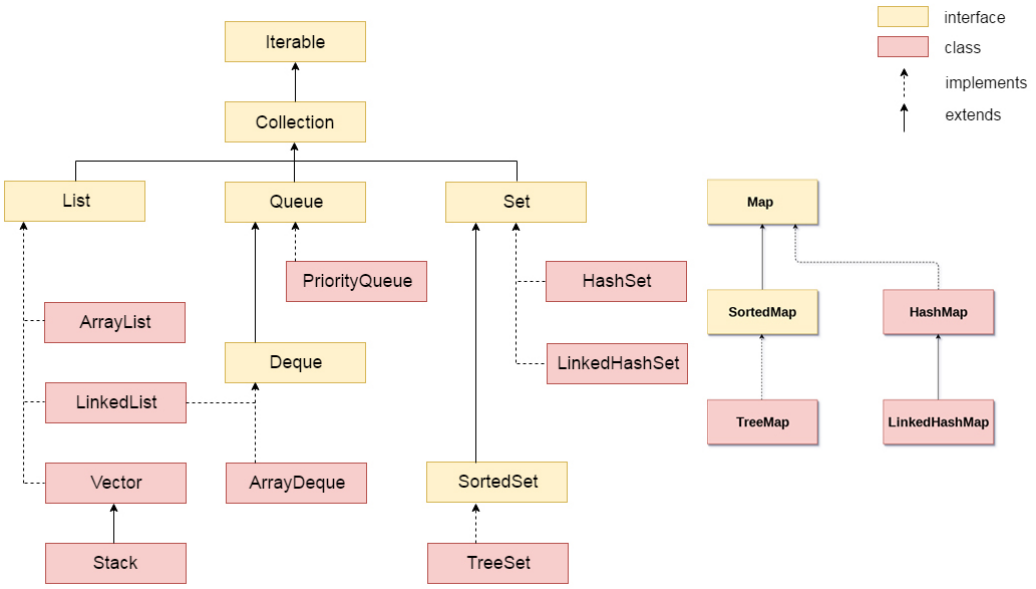
f) broj riječi koje započinju velikim početnim slovom.

Zadatak 7.4. Napisati program koji od korisnika traži unos `N` jedinstvenih brojeva u neki vektor `V` (potrebna provjera). Nakon toga treba obaviti sortiranje (po izboru) i ispis sortiranih brojeva na ekran.

7.7 Uvod u kolekcije

U prethodnim primjerima bavili smo se različitim tipovima nizova kao i ugrađenim klasama koje nam olakšavaju rad s istima. Međutim, problem nizova je što moramo unaprijed znati ukupan broj elemenata koje ćemo smjestiti unutar niza. Ako zauzmemo premalo mjesta, imamo problem, a isto to vrijedi ako zauzmemo previše, samo što je tada to problem bespotrebnog zauzimanja resursa. Kao rješenje tog problema nameću se kolekcije koje su dostupne još od verzije 1.2 Jave (proširene u verziji 5 s tehnikom Java Generics te se još uvijek nadograđuju), a predstavljaju nekakvu strukturu podataka, odnosno objekt koji može sadržavati reference na druge objekte. One nam omogućuju spremanje i manipuliranje promjenjivim brojem elemenata.

Okvir kolekcija (engl. *collection framework*) deklarira operacije koje se mogu provoditi generički nad različitim vrstama kolekcija, a sastoji se od sučelja, implementacijskih klasa i algoritama kao što je prikazano na slici 7.3.



Slika 7.3: Hijerarhijski prikaz okvira `Collection`.
Izvor: `https://www.wikitechy.com/tutorials/java/collections-in-java`, pristupljeno 3. 4. 2022.

Izravni nasljednici sučelja `Collection` dijele se na četiri glavna dijela.

Tablica 7.7: Nasljednici sučelja `Collection`.

Sučelje	Opis
`Set()`	Opisuje kolekcije koje su skupovi.
`List()`	Opisuje kolekciju poredanih elemenata (svaki element ima redni broj, na različitim pozicijama mogu se nalaziti isti elementi)
`Map()`	Opisuje kolekciju uređenih parova (ključ, vrijednost). Ne može sadržavati više zapisa koji imaju isti ključ.
`Queue()`	FIFO (engl. *first in first out*) struktura podataka.

7.7.1 Sučelje `Collection`

Sučelje `Collection` je osnovno sučelje koje modelira najopćenitiju kolekciju kao što je to prikazano na slici 7.3. Možemo reći da ćemo rijetko kada koristiti direktno metode ovog sučelja, već će se umjesto toga koristiti specifičnije implementacije ovisno o potrebama koje sadrže sve metode sučelja `Collection` uz nekoliko dodatnih. Metode sučelja `Collection` prikazane su u tablici 7.8.

Tablica 7.8: Metode sučelja `Collection<E>`.

Metoda	Opis
`boolean add(E element)`	Dodaje element u kolekciju. Osigurava da kolekcija sadrži određeni element (opcionalno).
`boolean addAll(Collection<? Extends E> c)`	Dodaje sve elemente predane kolekcije.
`void clear()`	Briše sve elemente iz kolekcije (opcionalno).
`boolean contains(Object o)`	Vraća `true`/`false` ovisno o prisutnosti traženog elementa.
`boolean containsAll(Collection<?> c)`	Vraća `true`/`false` ovisno o prisutnosti više traženih elemenata.
`boolean equals(Object o)`	Uspoređuje zadani objekt s kolekcijom.
`int hashCode()`	Vraća hash vrijednost kolekcije.
`boolean isEmpty()`	Vraća `true`/`false` ovisno o popunjenosti kolekcije.
`Iterator<E> iterator()`	Vraća `Iterator<E>` za iteriranje nad elementima kolekcije.
`boolean remove(Object o)`	Ako postoji, briše jedan element iz kolekcije (opcionalno).
`boolean removeAll(Collection<?> c)`	Briše sve elemente koji su sadržani u predanoj kolekciji (opcionalno).
`boolean retainAll(Collection<?> c)`	Zadržava sve elemente koji su sadržani u predanoj kolekciji (opcionalno).
`int size()`	Vraća broj elemenata u kolekciji.
`Object[] toArray()`	Vraća niz `Object[]` koji sadržava elemente kolekcije.
`<T> T[] toArray(T[] a)`	Vraća niz `<T> T[]` koji sadržava elemente kolekcije.

7.7.2 Sučelje `Set`

Sučelje `Set` je proširenje sučelja `Collection` i predstavlja neporedanu kolekciju jedinstvenih elemenata. Postoji nekoliko različitih implementacija ovog sučelja kao što su klase `HashSet`, `LinkedHashSet` i `TreeSet`. Metode sučelja `Set` identične su sučelju `Collection` i nema niti jedne dodatne metode kao što je vidljivo u tablici 7.9.

Tablica 7.9: Metode sučelja `Set<E>`.

Metoda	Opis
`boolean add(E element)`	Dodaje element u set ako već nije prisutan.
`boolean addAll(Collection<? Extends E> c)`	Dodaje sve elemente koji su sadržani u predanoj kolekciji ako već nisu prisutni u setu (opcionalno).
`void clear()`	Briše sve elemente iz seta (opcionalno).
`boolean contains(Object o)`	Vraća `true`/`false` ovisno o prisutnosti traženog elementa.
`boolean containsAll(Collection<?> c)`	Vraća `true`/`false` ovisno o prisutnosti više traženih elemenata.
`boolean equals(Object o)`	Uspoređuje zadani objekt sa setom.
`int hashCode()`	Vraća hash vrijednost seta.
`boolean isEmpty()`	Vraća `true`/`false` ovisno o popunjenosti seta.
`Iterator<E> iterator()`	Vraća `Iterator<E>` za iteriranje nad elementima seta.
`boolean remove(Object o)`	Ako postoji, briše jedan element iz seta (opcionalno).
`boolean removeAll(Collection<?> c)`	Briše sve elemente koji su sadržani u predanoj kolekciji (opcionalno).

`boolean retainAll(Collection<?> c)`	Zadržava sve elemente koji su sadržani u predanoj kolekciji (opcionalno).
`int size()`	Vraća broj elemenata u setu.
`Object[] toArray()`	Vraća niz `Object[]` koji sadržava elemente seta.
`<T> T[] toArray(T[] a)`	Vraća niz `<T> T[]` koji sadržava elemente seta.

```
public static void main(String[] args){
        // deklariramo varijablu tipa Set<string> i incijaliziramo
        // je da pokazuje na novostvoreni objekt tipa HashSet<String>
        Set<String> dani = new HashSet<String>();
        // dodajemo vrijednosti u kolekciju
        dani.add("Ponedjeljak");
        dani.add("Utorak");
        dani.add("Srijeda");
        dani.add("Ponedjeljak");
        dani.add("Cetvrtak");
        dani.add("Utorak");
        dani.add("Petak");
        dani.add("Subota");
        dani.add("Nedjelja");
        // ispisujemo ukupan broj elemenata
        System.out.println(dani.size()+"\n");
        // provjeravamo postojanost elementa u kolekciji
        System.out.println(dani.contains("Ponedjeljak")+"\n");
        // dohvat iteratora
        Iterator<String> iterator = dani.iterator();
        while(iterator.hasNext()){
                System.out.println(iterator.next());
        }
        // brisanje elementa iz kolekcije
        dani.remove("Utorak");
        System.out.println(dani+"\n");
        / /brisanje svih elemenata iz kolekcije
        dani.clear();
        System.out.println(dani.size()+"\n");
        System.out.println(dani.isEmpty()+"\n");
}
```

Primjer kôda 7.15 Upotreba jedne od implementacija sučelja `Set`.

Primjer koda 7.15 prikazuje upotrebu jedne od implementacije sučelja `Set`, odnosno sučelja `Collection` sa samo jednim dijelom dostupnih metoda.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
7

true

Ponedjeljak
Cetvrtak
Nedjelja
Srijeda
Subota
Utorak
Petak
[Ponedjeljak, Cetvrtak, Nedjelja, Srijeda, Subota, Petak]

0
```

```
true
```

7.7.3 Sučelje List

Sučelje `List` proširuje sučelje `Collection` i predstavlja poredanu kolekciju elemenata (može sadržavati duplicirane elemente). Postoji nekoliko različitih implementacija ovog sučelja kao što su klase `ArrayList`, `LinkedList`, `Vector` i `Stack`. Metode sučelja `List` slične su sučelju `Collection` uz dodatak drugih metoda kao što je vidljivo u tablici 7.10.

Tablica 7.10: Metode sučelja `List<E>`.

Metoda	Opis
`boolean add(E element)`	Dodaje element na kraj liste (opcionalno).
`void add(int index, E element)`	Dodaje element na točno određeno mjesto u listi.
`boolean addAll(Collection<? Extends E> c)`	Dodaje sve elemente koji su sadržani u predanoj kolekciji na kraj liste.
`boolean addAll(int index, Collection<? Extends E> c)`	Dodaje sve elemente koji su sadržani u predanoj kolekciji počev od zadanog mjesta. Konceptualno se radi o umetanju elemenata.
`void clear()`	Briše sve elemente iz liste (opcionalno).
`boolean contains(Object o)`	Vraća `true`/`false` ovisno o prisutnosti traženog elementa.
`boolean containsAll(Collection<?> c)`	Vraća `true`/`false` ovisno o prisutnosti više traženih elemenata.
`boolean equals(Object o)`	Uspoređuje zadani objekt s listom.
`E get(int index)`	Dohvaća element E s određenog mjesta u listi.
`int hashCode()`	Vraća hash vrijednost liste.
`int indexOf(Object o)`	Vraća indeks prvog pojavljivanja traženog elementa u listi.
`boolean isEmpty()`	Vraća `true`/`false` ovisno o popunjenosti seta.
`Iterator<E> iterator()`	Vraća `Iterator<E>` za iteriranje nad elementima liste.
`int lastIndexOf(Object o)`	Vraća indeks zadnjeg pojavljivanja traženog elementa u listi.
`ListIterator<E> listIterator()`	Vraća `ListIterator<E>` za iteriranje nad elementima liste.
`ListIterator<E> listIterator(int index)`	Vraća `ListIterator<E>` za iteriranje nad elementima liste od točno određenog mjesta.
`E remove(int index)`	Briše element s točno određenog mjesta u listi.
`boolean remove(Object o)`	Briše prvo pojavljivanje određenog elementa (ako postoji).
`boolean removeAll(Collection<?> c)`	Briše sve elemente koji su sadržani u predanoj kolekciji (opcionalno).
`boolean retainAll(Collection<?> c)`	Zadržava sve elemente koji su sadržani u predanoj kolekciji (opcionalno).
`E set(int indeks, E element)`	Zamjenjuje element na točno određenom mjestu novim.
`int size()`	Vraća broj elemenata u listi.
`List<E> subList(int fromIndex, int toIndex)`	Vraća `List<E>` za točno određeni dio liste određen indeksima.
`Object[] toArray()`	Vraća niz `Object[]` koji sadržava elemente liste.
`<T> T[] toArray(T[] a)`	Vraća niz `<T> T[]` koji sadržava elemente liste.

```
public static void main(String[] args){
        //jedan nacin inicijalizacije liste elemenata
        String[] dani = {"Ponedjeljak", "Utorak", "Srijeda", "Cetvrtak",
                         "Petak", "Subota", "Nedjelja"};
        List<String> daniTmp = Arrays.asList(dani);
        //drugi nacin inicijalizacije liste elemenata
        ArrayList<String> daniLista = new ArrayList<>();
        daniLista.add("Ponedjeljak");
        daniLista.add("Utorak");
        daniLista.add("Srijeda");
        daniLista.add("Cetvrtak");
        daniLista.add("Ponedjeljak");
        daniLista.add("Petak");
        daniLista.add("Subota");
        daniLista.add("Nedjelja");
        //deklariranje kolekcije koju cemo obrisati iz druge
        ArrayList<String> obrisi = new ArrayList<>();
        obrisi.add("Petak");
        obrisi.add("Subota");
        //ispis sadrzaja liste
        System.out.println(daniLista);
        //dohvat treceg elementa
        System.out.println(daniLista.get(2));
        //dohvat indeksa prvog pojavljaivanja za "Ponedjeljak"
        System.out.println(daniLista.indexOf("Ponedjeljak"));
        //dohvat indeksa zadnjeg pojavljivanja za "Ponedjeljak"
        System.out.println(daniLista.lastIndexOf("Ponedjeljak"));
        //brisanje dijela liste
        daniLista.removeAll(obrisi);
        //ispis sadrzaja liste
        System.out.println(daniLista);
        //postavljanje zadnjeg elementa na "Ponedjeljak"
        daniLista.set(daniLista.size()-1, "Ponedjeljak");
        //ispis liste
        System.out.println(daniLista);
        //provjera postojanja nekog elementa u listi
        System.out.println(daniLista.contains("Srijeda"));
        //dohvat podliste
        List<String> subList = daniLista.subList(0, 2);
        //ispis podliste
        System.out.println(subList);
}
```

Primjer kôda 7.16 Upotreba jedne od implementacija sučelja `List`.

Primjer kôda 7.16 prikazuje način upotrebe dijela dostupnih metoda sučelja `List`. Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Ponedjeljak, Petak, Subota,
Nedjelja]
Srijeda
0
4
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Ponedjeljak, Nedjelja]
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Ponedjeljak, Ponedjeljak]
true
[Ponedjeljak, Utorak]
```

Ovdje je važno napomenuti da metoda `asList`, koja je korištena za kreiranje liste `daniTmp` u primjeru kôda 7.16 vraća listu kojoj nije moguće mijenjati broj elemenata (moguće je zamijeniti element na nekoj poziciji drugim), ali brisanja i dodavanja nisu podržana.

7.7.4 Sučelje `Queue`

Sučelje `Queue` proširenje je sučelja `Collection` koje predstavlja strukturu podataka FIFO. Podaci se dohvaćaju onim redoslijedom kojim su dodani u ovakvu strukturu podataka (to ne mora nužno biti tako u svim slučajevima). Jedna je od implementacije `PriorityQueue`. Isto kao i kod sučelja `List`, sučelje `Queue` sadrži i dodatne metode koje se razlikuju od metoda unutar općeg sučelja `Collection` kao što je vidljivo u tablici 7.11.

Tablica 7.11: Metode sučelja `Queue<E>`.

Metoda	Opis
`boolean add(E element)`	Dodaje element u red ako je moguće. Ako nema mjesta, baca `IllegalStateException` grešku.
`E element()`	Dohvaća element iz reda, ali ga ne briše.
`boolean offer(E element)`	Dodaje element u red, ako je moguće.
`E peek()`	Dohvaća element iz reda, ali ga ne briše. Vraća `null` ako je red prazan.
`E poll()`	Dohvaća element iz reda i briše glavu reda. Vraća null ako je red prazan.
`E remove()`	Dohvaća element iz reda i briše glavu reda.

```
public static void main(String[] args){
        Queue<String> dani = new PriorityQueue<String>();
        dani.add("Ponedjeljak");
        dani.add("Utorak");
        dani.add("Srijeda");
        dani.add("Cetvrtak");
        dani.add("Petak");
        dani.add("Subota");
        dani.add("Nedjelja");
        System.out.println(dani);
        Queue<String> dani2 = new ArrayDeque<String>();
        dani2.add("Ponedjeljak");
        dani2.add("Utorak");
        dani2.add("Srijeda");
        dani2.add("Cetvrtak");
        dani2.add("Petak");
        dani2.add("Subota");
        dani2.add("Nedjelja");
        System.out.println(dani2);
        System.out.println(dani2.peek());//dohvat elementa iz reda
        System.out.println(dani2);
        dani2.offer("Nepostojeci"); //dodavanje u red
        System.out.println(dani2);
        System.out.println(dani2.remove());//brisanje iz reda
        System.out.println(dani2);
}
```

Primjer kôda 7.17 Upotreba jedne od implementacija sučelja `Queue`.

Primjer kôda 7.17 prikazuje upotrebu dviju različitih implementacija sučelja `Queue`, a razlika je u načinu dodavanja elemenata, odnosno u održavanju reda prilikom dodavanja. Također, prikazani su i načini upotrebe nekih od dostupnih funkcija iz tablice 7.11.

Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
[Nedjelja, Ponedjeljak, Petak, Cetvrtak, Utorak, Subota, Srijeda]
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja]
Ponedjeljak
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja]
[Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja,
Nepostojeci]
Ponedjeljak
[Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja, Nepostojeci]
```

7.7.5 Sučelje `Map`

Sučelje `Map` predstavlja skup elemenata u kojem su vrijednosti povezane ključevima. Također, nije moguće ubacivanje dupliciranih vrijednosti, odnosno ključeva. Postoji nekoliko različitih implementacija ovog sučelja kao što su klase `HashMap`, `LinkedHashMap` i `tTreeMap`. Metode sučelja `Map` slične su sučelju `Collection` uz dodatak drugih metoda kao što je vidljivo u tablici 7.12.

Tablica 7.12: Metode sučelja `Map<K,V>`.

Metoda	Opis
`void clear()`	Briše sve elemente iz mape (opcionalno).
`boolean containsKey(Object key)`	Provjerava postojanje određenog ključa u mapi.
`boolean containtValue(Object value)`	Provjerava postojanje neke vrijednosti u mapi.
`set<Map.Entry<K,V>> entrySet()`	Vraća set elemenata sadržanih u mapi.
`boolean equals(Object o)`	Uspoređuje zadani objekt s mapom.
`V get(Object key)`	Vraća vrijednost povezanu s određenim ključem.
`int hashCode()`	Vraća hash vrijednost mape.
`boolean isEmpty()`	Provjerava sadrži li mapa elemente.
`Set<K> keySet()`	Vraća skup ključeva sadržanih u mapi.
`V put(K key, V value)`	Dodaje vrijednost u mapu i povezuje je s ključem (opcionalno).
`void putAll(Map<? Extends K, ? extends V> m)`	Kopira sve elemente iz jedne mape i stavlja ih u drugu (opcionalno).
`V remove(Object key)`	Briše element iz mape s odgovarajućim ključem (opcionalno).
`int size()`	Vraća broj ključ-vrijednost parova u mapi.
`Collection<V> values()`	Vraća kolekciju vrijednosti sadržanih u mapi.

```
public static void main(String[] args){
        Map<String,String> dani = new HashMap<String,String>();
        System.out.println(dani.isEmpty());
        //dodavanje parova kljuc-vrijednost
        dani.put("PON", "Ponedjeljak");
        dani.put("PON", "Ponedjeljak");
        dani.put("UTO", "Utorak");
        dani.put("SRI", "Srijeda");
        System.out.println(dani.values());//ispis vrijednosti mape
        System.out.println(dani.isEmpty());//provjera popunjenosti
        System.out.println(dani.get("PON")); //dohvat vrijednosti po kljucu
        dani.remove("SRI"); //brisanje po kljucu
        System.out.println(dani.keySet());
}
```

Primjer kôda 7.18 Upotreba jedne od implementacija sučelja `Map`.

Primjer kôda 7.18 prikazuje upotrebu ugrađenih metoda u klasi `HashMap` kao implementaciji sučelja `Map`. Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
true
[Utorak, Srijeda, Ponedjeljak]
false
Ponedjeljak
[UTO, PON]
```

7.7.6 Klasa `Collections`

Klasa `Collections`, kao dio okvira kolekcija, predstavlja klasu koja sadrži nekoliko optimiziranih statičkih metoda za manipulaciju elementima kolekcija.

Tablica 7.13: Metode klase `Collections`.

Metoda	Opis
`sort()`	Sortira elemente liste.
`binarySearch()`	Traži objekt u listi.
`reverse()`	Mijenja redoslijed elemenata u listi.
`shuffle()`	Randomizira poredak elemenata u listi.
`fill()`	Postavlja sve elemente liste na neku vrijednost.
`copy()`	Kopira reference jedne liste u drugu.
`min()`	Vraća najmanji element u kolekciji.
`max()`	Vraća najveći element u kolekciji.
`addAll()`	Dodaje sve elemente polja u kolekciju.
`frequency()`	Vraća broj elemenata kolekcije koji odgovaraju nekom određenom elementu.
`disjoint()`	Vraća vrijednost `true` ako kolekcije nemaju zajedničkih elemenata, a `false` inače.

```
public static void main( String[] args ){
        String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
        //Stvori i prikazi listu koja sadrzava elemente polja suits
        List< String > list = Arrays.asList( suits );
        System.out.printf( "Lista: %s\n", list );
        //sortiranje uzlano
        Collections.sort(list);
        // sortiranje silazno
        Collections.sort( list, Collections.reverseOrder() );
        // ispis elemenata
        System.out.printf( "Sortirana lista: %s\n", list );
}
```

Primjer kôda 7.19 Upotreba metoda klase `Collections`.

Primjer kôda 7.19 prikazuje upotrebu metode `sort` klase `Collections`. Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
Lista: [Hearts, Diamonds, Clubs, Spades]
Sortirana lista: [Spades, Hearts, Diamonds, Clubs]
```

```
public static void main( String[] args ){
        // stvori i prikazi List< Character >
        Character[] letters = {'P','C','M'};
        List<Character> lista = Arrays.asList(letters);
        ispis(lista );
        // obrni i prikazi List< Character >
        Collections.reverse(lista ); // obrnuti redoslijed
        ispis(lista );
        // stvaranje copyList iz polja s 3 znaka
```

```
        Character[] lettersCopy = new Character[ 3 ];
        List< Character > copyList = Arrays.asList(lettersCopy);
        //kopiranje elemenata list u copyList
        Collections.copy(copyList, lista );
        ispis(copyList);
        // popunjavanje liste s 'R'
        Collections.fill(lista,'R');
        ispis(lista );
} // end main

private static void ispis( List< Character > listRef ){
        for (Character element : listRef){
                System.out.printf( "%s ", element );
        }
        System.out.printf( "\nMax: %s ", Collections.max( listRef ) );
        System.out.printf( "Min: %s\n", Collections.min( listRef ) );
} // end method output
```

Primjer kôda 7.20 Upotreba metoda klase `Collections`.

Primjer 7.20 prikazuje upotrebu metoda `reverse`, `fill`, `copy`, `max` i `min` klase `Collections`. Za navedeni primjer dobit ćemo sljedeći izlaz u konzoli:

```
P C M
Max: P Min: C
M C P
Max: P Min: C
M C P
Max: P Min: C
R R R
Max: R Min: R
```

7.8 Generičke klase i metode

Do sada smo se prilikom korištenja nekih od ugrađenih klasa i metoda susreli s generičkim klasama i metodama, a sada se možemo i pobliže s njima upoznati. Java generici (engl. *Java Generics*) odnose se na skup značajki programskog jezika koje nam omogućavaju pisanje generičkih tipova i metoda. Podrška za generičke klase i metode uvedena je od Jave 5, a sve s ciljem smanjenja broja grešaka u kôdu kao i dodavanjem dodatnog sloja apstrakcije nad tipovima. Upotreba generika osigurava točno korištenje predviđenih tipova podataka (engl. `type safety`) na način da prevoditelj provjerava jesu li svi korišteni tipovi ispravni te onemogućava pojavljivanje greške `ClassCastException`. Što to točno znači možemo vidjeti na sljedećim dvama primjerima.

```
public static void main(String[] args){
        Object[] lista = new Object[10];
        lista[0]="String";
        lista[1]=100;
        for(int i=0;i<2;i++)  int s = (int) lista[0];
}
```

Primjer kôda 7.21 Upotreba liste objekata tipa `Object`.

Za navedeni primjer dobit ćemo sljedeću poruku u konzoli:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
        at genericsTest.Test.main(Test.java:22)
/Users/Library/Caches/NetBeans/8.2/executor-snippets/run.xml:53: Java
returned: 1
```

```
static void funkcija(Object b1, Object b2){
        int broj1 = (int) b1;
        int broj2 = (int) b2;
        System.out.println("Zbroj: "+(broj1+broj2));
}
public static void main(String[] args){
        List<Object> lista = new ArrayList<>();
        // ili List lista = new ArrayList<>();
        lista.add(1);
        lista.add(2.00);
        lista.add("String");
        int b1 = (int) lista.get(0);
        double b2 = (double) lista.get(1);
        String s = (String) lista.get(2);
        System.out.println(b1+" "+b2+" "+s);
        funkcija(lista.get(0),lista.get(1));
}
```

Primjer kôda 7.22 Dodavanje i dohvat elemenata iz liste tipa `Object`.

Za navedeni primjer dobit ćemo sljedeću poruku u konzoli:

```
1 2.0 String
Exception in thread "main" java.lang.ClassCastException: java.lang.Double
cannot be cast to java.lang.Integer
        at genericsTest.Test.funkcija(Test.java:20)
        at genericsTest.Test.main(Test.java:37)
/Users/tomislavgalba/Library/Caches/NetBeans/8.2/executor-snippets/run.xml:
53: Java returned: 1
```

U primjerima 7.21 i 7.22 možemo vidjeti upotrebu tipa podataka `Object`, što je potpuno ispravno jer klasa `Object` je nadklasa svim ostalim klasama (uključujući i korisnički definirane klase). Na ovaj način moguće je definirati generičke metode ili klase i prije službenog uvođenja generičkih metoda i klasa u Javi 5. Međutim, takav način pisanja kôda može u nekom trenutku dovesti do greške s obzirom da nema provjere tipova podataka koji se spremaju u neku strukturu podataka te ukoliko se dovoljno dobro ne pazi, može dovesti do pojave greške kao u primjerima 7.21 i 7.22. Kako bi se to izbjeglo i kako bi se provjera tipova obavila prilikom prevođenja, koriste se generičke metode i klase.

7.8.1 Generičke klase

Generički klasu deklariramo jednako kao i običnu klasu uz razliku što pored imena klase dodajemo imena parametara u kutne zagrade (engl. *type parameter*) kojih može biti jedan ili više odvojenih zarezom. Takve klase još nazivamo i parametrizirane klase.

Kada pričamo o označavanju parametara, iako ne postoji pisano pravilo, odnosno možemo koristiti bilo koje slovo, ipak postoji određena konvencija:

- `T` – tip;
- `E` – element (npr. `ArrayList<E>`);
- `K` – ključ (npr. `Map<K,V>`);
- `V` – vrijednost (npr. `Map<K,V>`).

Prema navedenom, možemo napisati jednostavnu generičku klasu:

```
public class Vrijednost<T>{
        private T t;

        public void postavi(T t){
        this.t = t;
```

```
        }
        public T dohvati(){
                return t;
        }
}
```

Primjer kôda 7.23 Definicija generičke klase.

Primjer kôda 7.23 prikazuje definiciju generičke klase s jednim parametrom. Upotrebom tog primjera možemo napraviti sljedeće deklaracije:

```
public static void main(String[] args){
        Vrijednost<Integer> v1 = new Vrijednost<Integer>();
        Vrijednost<String> v2 = new Vrijednost<String>();

        v1.postavi(1);
        v2.postavi("String");

        System.out.println("Prva: "+v1.dohvati());
        System.out.println("Druga: "+v2.dohvati());
}
```

Primjer kôda 7.24 Instanciranje i upotreba objekata generičke klase.

Za primjer koda 7.24 dobit ćemo sljedeći izlaz u konzoli:

```
Prva: 1
Druga: String
```

Ako bi primjer koda 7.23 htjeli napisati koristeći tip podatka `Object`, tada bi definicija klase izgledala:

```
public class Vrijednost{
        private Object t;
        public void postavi(Object t) {
                this.t = t;
        }
        public Object dohvati() {
                return t;
        }
}
```

Primjer kôda 7.25 Definicija klase `Vrijednost` korištenjem tipa `Object`.

7.8.2 Generičke metode

Definicija generičke metode vrlo je slična definiciji generičke klase. Svaka metoda koja bilo kako koristi parametre generička je metoda. Ono što bi ovdje trebalo istaknuti da metoda može uvesti i vlastite (”nove”) parametre. Oni mogu biti nezavisni ili mogu biti dovedeni u neki odnos s parametrom klase (ako je klasa parametrizirana). Parametari se mogu koristiti za deklariranje povratnog tipa kao i lokalnih varijabli, dok naziv definiran u zaglavlju mora ostati isti kroz cijelu metodu.

```
public static void printArray( Integer[] inputArray ){
        for ( Integer element : inputArray )
                System.out.printf( "%s ", element );
        System.out.println();
}
public static void printArray( Double[] inputArray ){
        for ( Double element : inputArray )
                System.out.printf( "%s ", element );
        System.out.println();
}
public static void printArray( Character[] inputArray ){
        for ( Character element : inputArray )
```

```
                System.out.printf( "%s ", element );
        System.out.println();
}
public static void main( String[] args ){
        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
        printArray( integerArray );
        printArray( doubleArray );
        printArray( characterArray );
}
```

Primjer kôda 7.26 Definicija metoda (zamjena za generičke metode).

Do uvođenja generičkih tipova, ukoliko smo htjeli deklarirati metodu za ispis podataka iz liste, a da lista može biti različitog tipa (npr. `Integer`, `Double`, `Character`), morali bismo napisati zasebne metode (ili koristiti `Object` kao tip pa raditi provjeru tipa). Primjer kôda 7.26 prikazuje upravo takvu situaciju. Kôd u primjeru kôda 7.26 u potpunosti je ispravan i kao takav se može koristiti, ali očigledno je da može prouzrokovati nepotrebno gomilanje kôda, što može povećati mogućnost greške kao i otežano održavanje u kasnijim fazama.

Ukoliko koristimo generičke metode, primjer kôda 7.26 mogli bismo napisati na sljedeći način:

```
public static < T > void printArray( T[] inputArray ){
        for ( T element : inputArray ) System.out.printf( "%s ", element );
        System.out.println();
}
public static void main( String[] args ){
        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
        printArray( integerArray );
        printArray( doubleArray );
        printArray( characterArray );
}
```

Primjer kôda 7.27 Definicija generičke metode.

Izlaz u konzoli bit će jednak za oba primjera.

```
1 2 3 4 5 6
1.1 2.2 3.3 4.4 5.5 6.6 7.7
H E L L O
```

7.8.3 Ograničenja generika

Postoje određena ograničenja koja se odnose na generike (metode, klase, sučelja itd.), a neka od njih su:

- Nije moguće koristiti statičke tipove.

```
public class Vrijednost<T> {
        private static T t;
}
```

- Nije moguće koristiti primitivne tipove podataka.

```
int[] integerArray = { 1, 2, 3, 4, 5, 6 };
printArray( integerArray );
...
```

- Nije moguće napraviti instancu tipa parametra (engl. *type parameter instance*).

```
T inst = new T();
```

- Nije moguće koristiti `instanceof` nad tipovima parametara.

```
Vrijednost<Integer> v1 = new Vrijednost<Integer>();
if(v1 instanceof Vrijednost<Integer>){ }
```

- Nije moguće napisati generičku klasu koja predstavlja iznimku.

```
public class Iznimka<T> extends Exception
```

7.9 Zadaci

Zadatak 7.5. Napisati klasu `Kosarica` koja ima jedan privatni član (koristiti odgovarajuću kolekciju) te dvije metode `dodaj` i `ispisi`. U košaricu se dodaju razni artikli (nekada i više istih artikala). Prilikom poziva metode `ispisi` na ekran bi se trebali ispisati samo jedinstveni artikli dodani u košaricu.

Zadatak 7.6. U trgovinu elektroničkom opremom stiže dugoočekivani mobilni telefon. Ispred trgovine stvara se veliki red. Međutim, vlasnik trgovine u jednom je trenutku odlučio primati stranke od kraja reda. Napisati program koji će omogućiti unošenje stranki u neku od kolekcija (koristiti odgovarajuću kolekciju). Nakon unosa stranki potrebno je obrnuti redoslijed i ispisati podatke na ekran.

Zadatak 7.7. Prilagoditi zadatak 7.6. tako da red ispred trgovine bude predstavljen klasom `RedStranki` s pripadajućim metodama za dodavanje stranki i dohvaćanje sljedeće stranke koja je na redu. Npr. ako u redu stoje Pero, Marko i Ivan, kada se pozove metoda za dohvat sljedeće stranke, rezultat bi trebao biti Pero. Koristiti odgovarajuću kolekciju.

Zadatak 7.8. Napisati program koji omogućava unos ocjena s tipkovnice u kolekciju podataka (koristiti odgovarajuću kolekciju) sve dok se ne unese riječ `"STOP"`. Nakon završetka unosa potrebno je ispisati koliko je uneseno pojedinih ocjena.

Zadatak 7.9. Napisati generičku klasu s odgovarajućim generičkim metodama koje će predstavljati jednostavnu listu u koju će se moći spremiti različiti tipovi podataka (po uzoru na implementacije sučelja `List`). Lista bi trebala imati mogućnost dodavanja novih elemenata, dohvat unesenih elemenata, dohvat veličine liste i brisanje svih elemenata. Na primjer:

```
KorisnickaLista<String> lista = new KorisnickaLista<>(10)
```

gdje vrijednost argumenta `10` predstavlja maksimalan broj elemenata koje možemo spremiti u listu.

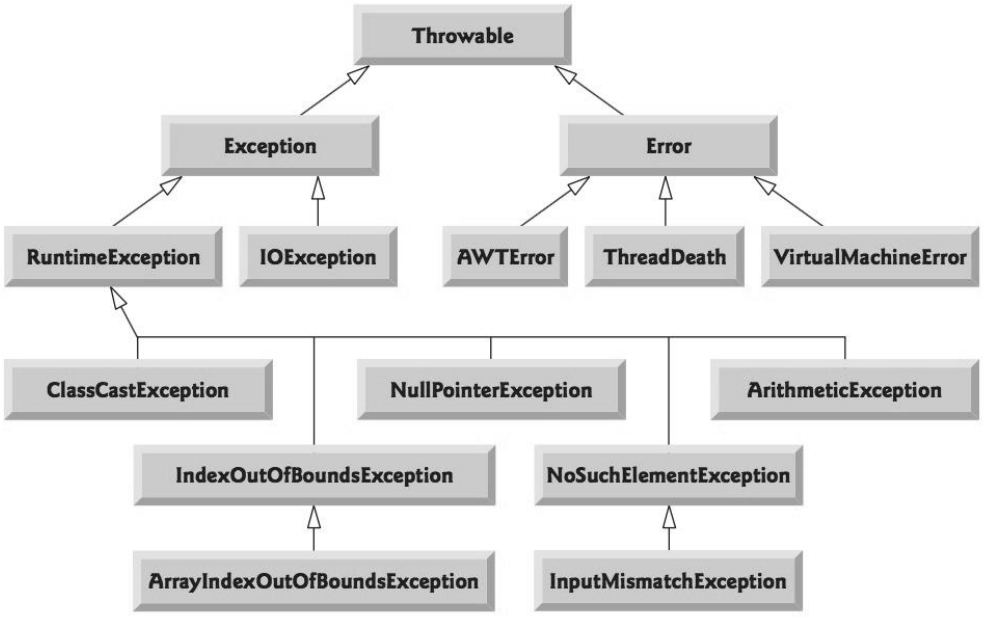
Poglavlje 8

Iznimke

Iznimka predstavlja prekid u izvođenju određenog bloka programskog kôda uslijed nekog problema. Svaki program podložan je greškama, a to se posebno odnosi na interaktivne programe koji omogućavaju korisnicima unos putem tipkovnice ili grafičkog sučelja. Osim navedenog moguće su i greške prilikom izrade programa ili greške prouzrokovane okruženjem u kojem se izvodi program (npr. program zahtijeva više resursa, nego što je dostupno). Pravilno rukovanje iznimkama omogućava normalno izvođenje programskog kôda usprkos nastaloj grešci.
Napomena: Iznimke predstavljaju greške koje se događaju prilikom izvršavanja ispravno napisanog i ispravno prevedenog programa. Iznimke se ne odnose na greške koje se pojavljuju prilikom prevođenja programskog kôda.

8.1 Podjela iznimaka

Na slici 8.1 možemo vidjeti da se iznimke dijele na dvije glavne klase, klasu `Exception` i klasu `Error`. Drugim riječima, bilo koja iznimka predstavlja direktnu instancu klase `Throwable` ili indirektnu u slučaju nekih njezinih naslijeđenih klasa (npr. `NullPointerException`).



Slika 8.1: Hijerarhija klasa koje predstavljaju iznimke. Izvor: "Java How to Program, Late Objects Version", Tenth Edition, Paul Deitel; Harvey Deitel, 2015.

Klasa `Error` i njezine podklase predstavljaju iznimke koje ne možemo ispraviti u programu, odnosno to su greške koje su se dogodile zbog, primjerice, lošeg hardvera (manjak resursa) ili zbog greške unutar virtualnog stroja (JVM). U slučaju nastanka takve greške, izvršavanje će se programa prekinuti.

```
public class Test {
        public static void main(String[] args) {
                List<String> lista = new ArrayList<>();
                while(true){
                        lista.add("Probni string");
                }
        }
}
```

Primjer kôda 8.1 Izazivanje greške uzrokovane nepravilnim oslobađanjem memorije (engl. *memory leak*)

Primjer kôda 8.1 prikazuje nastajanje instance klase `Error`, odnosno iznimku uzrokovanu nedovoljnom količinom raspoložive radne memorije. Iako je ova iznimka namjerno isprovocirana lošim programiranjem, ona se itekako može dogoditi i kada mislite da radite dobar posao. Zamislite na primjer da imate nekakav program koji omogućava korisniku otvaranje kartica (podprozora) u kojima se obavljaju nekakve radnje kao što je primjerice uređivanje dokumenata. Svaka takva kartica zahtijeva određenu količinu radne memorije i ukoliko se resursima ne upravlja dobro i korisniku ne ograniči broj otvorenih kartica, vrlo se brzo nađemo u situaciji koja će prouzrokovati iznimku klase `Error`.

Klasa `Exception` ili njezine podklase modeliraju pogreške čiji se uzroci mogu otkloniti u programu, odnosno mogu se riješiti programskim putem. Primjerice, kada imamo program koji vrši interakciju s korisnikom i zahtijeva od korisnika unos nekog podatka u točno određenom formatu, aplikacija može provjeriti što je korisnik unio, a ukoliko greška i nastane, aplikacija ju može uhvatiti i pravilno obraditi tako da ne dođe do prekida izvođenja programa.

Na jednostavnom primjeru prikazat će se tijek programa bez korištenja kontrole iznimki kao i tijek programa s kontrolom iznimaka.

```
package test;
import java.util.Scanner;
public class Klasa1{

    public static int dijeljenje(int prvi, int drugi){
        return prvi/drugi;
    }
    public static void main(String [] args){
        int prvi=0, drugi=0, rezultat=-1;
        Scanner input = new Scanner(System.in);
        System.out.println("Unesite prvi broj: ");
        prvi = input.nextInt();
        System.out.println("Unesite drugi broj: ");
        drugi = input.nextInt();
        rezultat = dijeljenje(prvi,drugi);
        if(rezultat!=-1){
            System.out.println("Rezultat dijeljenja je: "+rezultat);
        }else{
            System.out.println("Dijeljenje nije uspjelo");
        }
    }
}
```

Primjer kôda 8.2 Programski kod bez upotrebe bloka `try-catch`

Izlaz je kako slijedi:

```
Unesite prvi broj:
3
Unesite drugi broj:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at test.Klasa1.dijeljenje(Klasa1.java:6)
        at test.Klasa1.main(Klasa1.java:15)
```

Za unos 3 i 0 može se vidjeti da je došlo do greške uzrokovane dijeljenjem nulom. Također, vidljivo je da se program nije izvršio do kraja zbog nastale greške. Kada se pojavi iznimka u konzoli nam se ispisuje trag stoga (engl. *stack trace*) koji nam omogućava uvid u put kojim je došlo do iznimke, što uvelike omogućava pronalazak mjesta na kojem se dogodila greška. Prva upisana linija u tragu stoga predstavlja zadnju liniju koja se izvršila prije nego se pojavila iznimka. Za primjer iznad, možemo vidjeti da se iznimka dogodila na retku 6. Redak 15 je poziv `rezultat = dijeljenje(prvi,drugi);` što upućuje da je problem u funkciji `dijeljenje`, a to je vidljivo i u tragu stoga u retku iznad `at test.Klasa1.dijeljenje(Klasa1.java:6)`. Ako povežemo ta dva retka s vrstom iznimke `ArithmeticException` i porukom `/ by zero`, možemo zaključiti da je problem u brojevima koji se prosljeđuju u metodu.

Prilikom razvoja ozbiljnog programskog rješenja nikako nije preporučljivo prekidanje programa zbog nekakve greške (neuspjela uspostava veze na bazu ili poslužitelj, nepostojeća datoteka, nepostojeći korisnik i slično). Iz toga razloga, poželjno je na svim ključnim dijelovima za koje znamo i za koje pretpostavljamo da mogu uzrokovati grešku, postavimo kontrolu iznimki (blok `try-catch`).

```
public class Klasa1{
    public static int dijeljenje(int prvi, int drugi){
        return (prvi/drugi);
    }
    public static void main(String [] args){
        int prvi=0, drugi=0, rezultat=-1;
        Scanner input = new Scanner(System.in);
        System.out.println("Unesite prvi broj: ");
        prvi = input.nextInt();
        System.out.println("Unesite drugi broj: ");
        drugi = input.nextInt();
        try{
            rezultat = dijeljenje(prvi,drugi);
        catch(Exception greska){
            System.out.println("Dogodila se greska");
            greska.printStackTrace();
        }
        if(rezultat!=-1){
            System.out.println("Rezultat dijeljenja je: "+rezultat);
        }else{
            System.out.println("Dijeljenje nije uspjelo");
        }
    }
}
```

Primjer kôda 8.3 Programski kôd uz upotrebu bloka `try-catch`

U primjeru kôda 8.3 možemo vidjeti upotrebu bloka `try-catch` kojim ćemo uhvatiti eventualnu iznimku. Na taj način osiguravamo da će se naš program nastaviti izvršavati i nakon nastanka iznimke.

```
Unesite prvi broj:
3
Unesite drugi broj:
0
Dogodila se greska
java.lang.ArithmeticException: / by zero
Dijeljenje nije uspjelo
```

```
        at test.Klasa1.dijeljenje(Klasa1.java:6)
        at test.Klasa1.main(Klasa1.java:16)
```

Na izlazu možemo vidjeti da se iznimka dogodila i da smo ju uspješno uhvatili te da se program izvršio do kraja bez prekida.

Osim jedne iznimke, u programu se vrlo često mogu dogoditi više različitih vrsta iznimki ovisno o zadatku koji obavlja program. Primjerice, u jednom bloku naredbi možemo obavljati računske operacije, a rezultat spremati u nekakvu datoteku ili ga slati na nekakav udaljeni poslužitelj. U tom slučaju greške možemo obraditi na više različitih načina, a svi rezultiraju istim ishodom.

Prvi je način hvatanje više vrsta iznimaka.

```
public static void main(String [] args){
    Scanner input = new Scanner(System.in);
    int broj=0, rezultat=-1;
    int niz[] = {2,4,6,8,10};
    try{
        for(int i=0;i<6;i++){
            System.out.println("Unesite djelitelj za broj "+niz[i]+": ");
            broj = input.nextInt();
            System.out.println("Rezultat je: "+(niz[i]/broj));
        }
    } catch(ArrayIndexOutOfBoundsException e){ e.printStackTrace();
    } catch(ArithmeticException e){ e.printStackTrace();
    } catch(Exception e){ e.printStackTrace(); }
}
```

Primjer kôda 8.4 Hvatanje grešaka prouzrokovanih nepravilnim pisanjem kôda

U primjeru kôda 8.4 možemo vidjeti da prolazimo kroz niz koji sadrži nekakve brojeve gdje postoji opasnost pokušaja pristupa indeksu izvan niza, a u isto vrijeme broj iz niza pokušavamo podijeliti s nekim brojem gdje postoji opasnost pokušaja dijeljenja s nulom. To znači da se mogu dogoditi minimalno dvije vrste iznimaka koje program uspješno hvata. Zadnji blok `catch` odnosi se na sve ostale greške koje nismo predvidjeli i dobra je praksa koristiti takav način hvatanja iznimaka.

Isti primjer možemo napisati i na način da u jednoj liniji navodimo sve vrste iznimaka koje se mogu pojaviti, a da se odnose na bilo koju podklasu klase `Exception` (ne smije uključivati klasu `Exception`).

```
try{
        ...
}catch(ArrayIndexOutOfBoundsException | ArithmeticException e)
        { e.printStackTrace();
}catch(Exception e){ e.printStackTrace(); }
```

Primjer kôda 8.5 Različiti načini definiranja bloka `catch`

Napomena: Redoslijed blokova `catch` važan je ukoliko želimo na ispravan način obraditi greške i informirati korisnika.

Iako je preporučeno razdvajati i hvatati točno one vrste iznimaka koje se eventualno mogu pojaviti (na taj način možemo obrađivati greške na način na koji je to potrebno, a uz to korisniku možemo prikazati točnije poruke o problemu koji se dogodio), greške možemo hvatati korištenjem jednog bloka `catch` (hvatanjem greške bilo koje vrste).

```
try{
        ...
catch(Exception e){ e.printStackTrace(); }
```

Primjer kôda 8.6 Korištene opće klase `Exception`

8.2 Blok `finally`

Iako koristimo blok `try-catch` za hvatanje iznimaka, sav kôd koji se nalazi ispod mjesta gdje je nastala iznimka neće se izvršiti (pogledati sliku 8.2).

Slika 8.2: Izvršavanje kôda nakon nastajanje greške.

U nekim slučajevima to neće predstavljati nikakav poseban problem; međutim, možda se u tom dijelu kôda trebaju osloboditi ranije zauzeti resursi, što onda predstavlja jednu vrstu curenja memorije (engl. *memory leak*). Kako bismo izbjegli taj problem i kako bismo u slučaju greške ipak uspjeli osloboditi sve eventualno zauzete resurse, to možemo napraviti unutar bloka `catch` ili još praktičnije unutar bloka `finally`. Blok `finally` predstavlja blok koji će se svakako izvršiti nakon završetka bloka `try-catch`.

```
try{
        ...
}catch(ArrayIndexOutOfBoundsException | ArithmeticException e){
        e.printStackTrace();
}finally{
        System.out.println("Blok finally");
}
```

Primjer kôda 8.7 Korištenje bloka `finally`.

Alternativa za upotrebu bloka `finally` uvedena je u novijim verzijama Jave (od verzije 7), a u svrhu smanjivanja količine kôda. Novi način pisanja bloka `try-catch` (konstrukt `try-with-resource`) omogućava zauzimanje jednog ili više resursa koji će se koristiti unutar bloka `try` te se nakon izvršavanja automatski zatvoriti. Preduvjet za to jest da je resurs primjerak klase koji implementira sučelje `AutoCloseable` s pravilno implementiranom metodom `close`.

```
try(ClassName objekt = new ClassName(); drugi resurs){
        ...
}catch(Exception e){}
```

Primjer kôda 8.8 Korištenje konstrukta `try-with-resource` (automatsko oslobađanje resursa).

8.3 Naredba `throw` i deklaracija `throws`

Do sada opisani načini rukovanja iznimkama odnosili su se na otklanjanje grešaka nakon što su se dogodile. Međutim, što ako želimo dojaviti grešku iako to možda „službeno“ i nije greška. Npr. ako imamo funkciju koja računa korijen nekog broja, mi možemo zahtijevati da broj kojeg metoda prima bude pozitivan iako računanje korijena nad negativnim brojem neće baciti grešku. Ukoliko korisnik ipak unese nulu, metoda može izazvati iznimku. Takav način rada može se postići korištenjem naredbe `throw`.

```
public static void main(String [] args){
        try{
                System.out.println(Math.sqrt(-9.00));
        }catch(Exception ex){
                ex.printStackTrace();
        }
}
public static double vratiKorijen(double broj) throws Exception{
        if(broj<0){
                throw new Exception("Negativan broj");
        }
        return Math.sqrt(broj);
}
```

Primjer kôda 8.9 Upotreba naredbe `throw` i deklaracije `throws`.

U primjeru kôda 8.9 jasno se vidi korištenje naredbe `throw` kojom stvaramo novu instancu klase `Exception` sa korisnički definiranim parametrom. Deklaracija `throws` dodaje se u deklaraciju metode nakon čega se navodi koje je vrste iznimka (ili više njih) koju će metoda eventualno vratiti.
Napomena: s obzirom da takva funkcija može izazvati iznimku, poziv takve funkcije mora se staviti unutar bloka `try-catch`.

8.4 Vlastite iznimke

Iako Java JDK ima na raspolaganju širok raspon prethodno definiranih iznimaka, ipak je nekada potrebno izraditi vlastitu iznimku. Razloga za to može biti više, od posebnih korisničkih zahtjeva pa sve do bolje čitljivosti kôda. Glavni je preduvjet za izradu vlastite iznimke nasljeđivanje klase `Throwable` ili neke od podklasa (npr. `Exception`).

```
public class Iznimka extends Exception{
        private String identifikator;

        public Iznimka(String identifikator){
                this.identifikator = identifikator;
        }
        public String getIdentifikator() {
                return identifikator;
        }
        public void setIdentifikator(String identifikator) {
                this.identifikator = identifikator;
        }
}
```

Primjer kôda 8.10 Definicija vlastite iznimke.

Korištenje vlastite iznimke isto je kao i korištenje prethodno definiranih iznimaka.

```
public static void main(String [] args){
        try{
                funkcija();
        }catch(Iznimka ex){ ex.printStackTrace(); }
}
public static void funkcija() throws Iznimka{
        if(uvjet) throw new Iznimka("Vlastita iznimka");
}
```

Primjer kôda 8.11 „Bacanje“ vlastite iznimke.

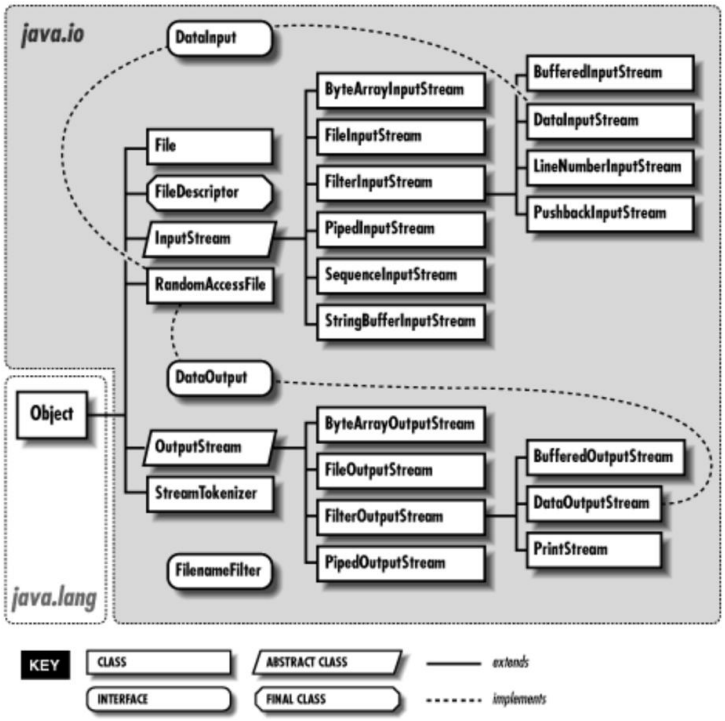
8.5 Zadaci

Zadatak 8.1. Napisati program koji omogućava korisniku unos imena i prezimena s tipkovnice. Ukoliko su početna slova imena ili prezimena napisana malim slovom, potrebno je baciti vlastitu iznimku s pripadajućom porukom koja opisuje grešku. Isto je potrebno napraviti ukoliko korisnik ne unese potpune podatke (npr. unese samo ime ili unese ime i prezime bez razmaka).

Poglavlje 9

Rad s datotekama

Prethodno je spomenuto pohranjivanje nekakvih vrijednosti/podataka u varijable, što je u potpunosti ispravno, ali varijable imaju jedan nedostatak. Nakon što program završi s izvršavanjem, sve što je bilo pohranjeno u varijablu nepovratno se gubi. Za to postoji nekoliko rješenja, a ovdje će se govoriti o datotekama kao jednom od dostupnih rješenja. Datoteke predstavljaju način pohranjivanja nekih informacija koje su dostupne i nakon što program završi s izvršavanjem (engl. *persistent data*). Za rad s datotekama Java koristi klase koje su dostupne u paketu `java.io`.



Slika 9.1: Hijerarhija paketa `java.io`. Izvor: O'Reilly, `https://docstore.mik.ua/orelly/java/exp/ch08_01.htm`, pristupljeno 3. 4. 2022.

9.1 Standardni tokovi `Input`, `Output` i `Error`

Unutar programa u Javi, datoteka se otvara na način da se stvara objekt na koji se veže tok okteta ili znakova. Nakon pokretanja Java programa stvaraju se tri toka:

- `System.in` – standardni ulazni tok koji omogućava čitanje okteta s tipkovnice;
- `System.out` – standardni izlazni tok koji omogućava ispis znakova na zaslon;

- `System.err` – standardni izlazni tok za pogreške koji omogućava ispis znakovno temeljenih poruka vezanih za nastale greške na zaslon.

Svaki od navedenih tokova može se i preusmjeriti u smislu da `System.in` čita oktete iz drugog izvora, `System.out` i `System.err` mogu slati izlaz na drugu lokaciju kao što je to datoteka na disku (HDD/SSD).

9.2 Paket `java.io`

Obrada datoteka obavlja se korištenjem klase iz paketa `java.io` kao što su:

- `FileInputStream` – ulazni tok okteta koji oktete čita iz datoteke;
- `FileOutputStream` – izlazni tok okteta koji oktete zapisuje u datoteku;
- `FileReader` – ulazni tok znakova koji znakove čita iz datoteke;
- `FileWriter` – izlazni tok znakova koji znakove zapisuje u datoteku.

U binarnom načinu rada podaci se prenose kao okteti (onako kako su zapisani u memoriji). Unutar paketa `java.io` možemo pronaći klase `InputStream` i `OutputStream` koje se odnose na rad s binarnim tokovima, dok su sve ostale klase namijenjene za drugu upotrebu (npr. rad s datotekama) nasljednici tih klasa.

```
public static void main(String [] args){
    try{
        InputStream inputStream = new FileInputStream("./datoteka.txt");
        byte b = (byte) inputStream.read();
        while(b!=-1){
            b = (byte) inputStream.read(); }
        inputStream.close();
    }catch(Exception e){ e.printStackTrace(); }
}
```

Primjer kôda 9.1 Tok temeljen na oktetima (čitanje podataka)

Primjer kôda 9.1 prikazuje čitanje podataka iz datoteke (binarni tok) korištenjem klase `FileInputStream`. Na početku navodimo putanju do datoteke te nakon toga otvaramo tok prema datoteci pozivom metode `read()`. Zatim se učitavaju podaci oktet po oktet sve dok ne dođemo do kraja (-1 signalizira da smo došli do kraja).
Napomena: nakon svake upotrebe bilo koji otvoreni resurs, pa tako i tok, potrebno je zatvoriti kako bismo oslobodili resurs i spriječili pojavu curenja memorije (engl. *memory leak*).

Slično kao u primjeru iznad moguće je zapisati podatke u datoteku korištenjem binarnog toka.

```
public static void main(String [] args){
    try{
        String s = "nekakav tekst kojeg zelimo zapisati u datoteku.";
        OutputStream outputStream = new FileOutputStream("./datoteka.txt");
        outputStream.write(s.getBytes());
        outputStream.close();
    }catch(Exception e){ e.printStackTrace(); }
}
```

Primjer kôda 9.2 Tok temeljen na oktetima (pisanje podataka)

Za razliku od binarnog toka, znakovni tokovi, kao što i sam naziv govori, koriste se kod čitanja ili pisanja znakovnih zapisa iz ili u datoteku. U tu svrhu, u paketu `java.io` nalaze se klase `Reader` i `Writer`.
Napomena: Kod binarnih tokova prijenos podataka vrši se u sirovom (engl. *raw*) obliku te nije potrebna nikakva dodatna obrada podataka pa je samim time prijenos podataka upotrebom binarnih tokova brži.

```
public static void main(String [] args){
        try{
                Reader inputStream = new FileReader("./datoteka.txt");
                int i = inputStream.read();
                while(i!=-1){
                        char c = (char)i;
                        System.out.println(c);
                        i = inputStream.read();
                }
                inputStream.close();
        }catch(Exception e){ e.printStackTrace(); }
}
```

Primjer kôda 9.3 Tok temeljen na znakovima (čitanje podataka)

```
public static void main(String [] args){
        try{
                Writer outputStream = new FileWriter("./datoteka.txt");
                outputStream.write("Tekst koji upisujemo u datoteku");
                outputStream.close();
        }catch(Exception e){ e.printStackTrace(); }
}
```

Primjer kôda 9.4 Tok temeljen na znakovima (pisanje podataka)

Kao dodatak klasama paketa `java.io` znakovno temeljeni ulaz/izlaz može se obaviti uz pomoć klasa `Scanner` i `Formatter` gdje `Formatter`, za razliku od klase `Scanner`, omogućava formatirani izlaz na bilo koji tekstualno temeljeni tok (slično metodi `System.out.printf` u C/C++).

Za ovaj primjer napravit ćemo klasu koja predstavlja korisnika banke te ćemo unesene podatke korisnika unositi u datoteku na formatirani način. Isto to napravit ćemo i prilikom čitanja istih podataka. Klasa će sadržavati metode za postavljanje i dohvaćanje podataka o korisniku kao što je broj računa, ime, prezime i stanje računa.

```
public class Korisnik{
    private int racun;
    private String ime, prezime;
    private double stanje;
    public Korisnik(){}
    public Korisnik(int racun, String ime, String prezime, double stanje){
        this.racun = racun;
        this.ime = ime;
        this.prezime = prezime;
        this.stanje = stanje;
    }
    public void setRacun( int racun ){ this.racun = racun; }
    public int getRacun() { return racun; }
    public void setIme( String ime ){ this.ime = ime; }
    public String getIme() { return ime; }
    public void setPrezime( String prezime ){ this.prezime = prezime; }
    public String getPrezime(){ return prezime; }
    public void setStanje( double stanje ){ this.stanje = stanje; }
    public double getStanje(){ return stanje; }
}//zavrsetak klase
```

Primjer kôda 9.5 Definicija klase `Korisnik`

```
public class Zapisi{
    private Formatter output; // objekt za unos u datoteku
    // otvori datoteku
```

```
    public void openFile(){
        try {
            output = new Formatter( "clients.txt" );
        } catch (FileNotFoundException ex) {}
    }
    // dodaj u datoteku
    public void addRecords(){
        // objekt koji zapisujemo u datoteku
        Korisnik korisnik = new Korisnik();
        korisnik.setRacun(1 );
        korisnik.setIme("Pero");
        korisnik.setPrezime("Peric");
        korisnik.setStanje( 10.00 );

        if ( korisnik.getRacun() > 0 ){
            // dodaj novi zapis na formatirani nacin
            output.format( "%d %s %s %.2f\n", korisnik.getRacun(),
            korisnik.getIme(), korisnik.getPrezime(),
            korisnik.getStanje() );
        }else{ System.out.println("Broj racuna mora biti veci od 0" ); }
    }
    public void closeFile(){
        if ( output != null ) output.close();
    }
}
```

Primjer kôda 9.6 Zapisivanje podataka iz objekta u datoteku na formatirani način.

Primjer kôda 9.6 prikazuje klasu `Zapisi` u kojoj su deklarirane tri metode: `openFile()` za otvaranje toka na datoteku u koju želimo zapisivati podatke, `addRecords()` za spremanje podataka iz objekta u datoteku na formatirani način te `closeFile()` za zatvaranje toka kako bi se oslobodili zauzeti resursi.

```
public static void main( String[] args ){
        Zapisi zapis = new Zapisi();
        zapis.openFile();
        zapis.addRecords();
        zapis.closeFile();
}
```

Primjer kôda 9.7 Stvaranja objekta i unos podataka u datoteku.

Izlaz datoteke nakon pokretanja programskog kôda u navedenim primjerima izgleda ovako (za potrebe primjera zadani su podaci za više korisnika):

```
1             Pero            Peric           10.00
2             Marko           Markovic        10.00
3             Ivan            Ivankovic       10.00
4             Mario           Maric           10.00
5             Ivan            Ivanovic        10.00
...
```

Nakon što smo unijeli podatke u datoteku, čitanje istih obavlja se na sljedeći način (navedeni način nije jedini na koji se može pristupiti podacima iz datoteke).

```
public class Citanje{
    private Scanner input;
    //otvori datoteku
    public void openFile(){
        try {
            input = new Scanner( new File( "clients.txt" ) );
        } catch (FileNotFoundException ex) {}
    }
```

```
    //citaj iz datoteke
    public void readRecords(){
        Korisnik korisnik = new Korisnik();
        System.out.printf( "%-10s%-12s%-12s%10s\n",
                           "Racun", "Ime", "Prezime", "Stanje" );
        while ( input.hasNext() ){
            korisnik.setRacun( input.nextInt() );
            korisnik.setIme( input.next() );
            korisnik.setPrezime( input.next() );
            korisnik.setStanje( input.nextDouble() );
            //prikazi procitane podatke
            System.out.printf( "%-10d%-12s%-12s%10.2f\n",
            korisnik.getRacun(), korisnik.getIme(),
            korisnik.getPrezime(), korisnik.getStanje() );
        }
    }
    //zatvori datoteku
    public void closeFile(){
        if ( input != null ) input.close();
    }
}
```

Primjer kôda 9.8 Čitanje podataka iz datoteke (spremanje u objekt).

Na sličan način, kao kod pisanja podataka u primjeru kôda 9.6, primjer kôda 9.8 sadrži tri metode `openFile()`, `readRecords()` i `cloeseFile()` gdje `readRecords` služi za čitanje podataka iz datoteke. Treba napomenuti da ovakav način čitanja podataka iz datoteke i spremanja u objekt može funkcionirati samo u slučaju kada znamo koji tip podatka se nalazi na kojem mjestu u datoteci. U suprotnom, ovo ne bi bilo moguće (rješenje možemo pronaći u serijalizaciji objekata o čemu će biti govora u sljedećim primjerima).

```
public static void main( String[] args ){
        Citanje citanje = new Citanje();
        citanje.openFile();
        citanje.readRecords();
        citanje.closeFile();
}
```

Primjer kôda 9.9 Stvaranje objekta i unos podataka iz datoteke.

Podaci nakon čitanja iz datoteke izgledaju kako slijedi:

```
1             Pero            Peric           10.00
2             Marko           Markovic        10.00
3             Ivan            Ivankovic       10.00
4             Mario           Maric           10.00
5             Ivan            Ivanovic        10.00
...
```

9.3 Serijalizacija objekata

Postoje klase koje omogućavaju ulaz/izlaz objekata ili različitih primitivnih tipova podataka. Na taj način omogućeno je čitanje i pisanje u obliku `int`, `char`, `String` ili nekog drugog tipa. Klase koje se koriste u tu svrhu su `ObjectInputStream` i `ObjectOutputStream`.

U prethodnim primjerima prikazan je način stvaranja datoteke, pisanja podataka u datoteku te čitanje istih. Nedostatak je takvog načina zapisivanja gubitak određenih informacija kao što je tip podatka (npr. ako prilikom čitanja datoteke za broj računa dobijemo broj 1, ne možemo sa sigurnošću odrediti radi li se o tipu podataka `int` ili `String`). Naravno, problem neće nastati ukoliko Vi radite i operaciju spremanja podataka u datoteku i operaciju čitanja podataka iz datoteke jer na taj način znate koje podatke i kojim

ih redoslijedom koristite. Problem nastaje ukoliko netko drugi sprema podatke, a Vi ih trebate čitati i iskoristiti u Vašem programskom okruženju.

Kako bismo riješili takav problem, Java ima mehanizam koji se naziva serijalizacija objekata. Serijalizirani objekt predstavlja reprezentaciju objekata kao niza okteta koji uključuju tip objekta kao i tip podataka spremljenih u objektu. Serijalizirani objekt može se pročitati iz datoteke i deserijalizirati te rekreirati.

Klase `ObjectInputStream` (implementira sučelje `ObjectInput`) i `ObjectOutputStream` (implementira sučelje `ObjectOutput`) omogućavaju čitanje ili pisanje objekata u datoteku. Sučelje `ObjectOutput` sadrži metodu `writeObject` koja za argument uzima objekt i piše njegove informacije u `OutputStream`. Klasa koja implementira sučelje `ObjectOutput` deklarira metodu `writeObject` i osigurava da objekt koji se šalje u `OutputStream` implementira sučelje `Serializible`. Sučelje `ObjectInput` sadrži metodu `readObject` koja čita i vraća referencu na objekt iz toka `InputStream`. Nakon uspješnog čitanja objekta, referenca se može ukalupiti na stvarni tip.

```
import java.io.Serializable;
public class Korisnik implements Serializable{
    private int racun;
    private String ime, prezime;
    private double stanje;
    public Korisnik(int racun, String ime, String prezime, double stanje){
        this.racun = racun;
        this.ime = ime;
        this.prezime = prezime;
        this.stanje = stanje;
    }
    // postavi broj racuna
    public void setRacun( int racun ){
        this.racun = racun;
    }
    // dohvati broj racuna
    public int getRacun() {
        return racun;
    }
...
```

Primjer kôda 9.10 Definicija serijalizabilne klase `Korisnik`.

U primjeru kôda 9.10 prikazan je isječak već postojeće klase `Korisnik` koju smo definirali ranije u primjerima uz blagu izmjenu, a to je omogućavanje serijalizacije objekata te klase. Kako bi se objekte klase moglo serijalizirati, klasa mora implementirati sučelje `Serializable` koje se nalazi u paketu `java.io.Serializable`. U klasi koja implementira sučelje`Serializable` sve varijable instance moraju biti serijalizabilne. U suprotnom, mora se eksplicitno navesti da nisu kako bi se u procesu serijalizacije mogle ignorirati. Prethodno je definirano da svi primitivni tipovi varijabli jesu serijalizibilni. Reference su uvijek serijalizabilne. Objekti (ovisno o tome koje su klase primjerak) mogu i ne moraju biti. String je, na primjer, serijalizibilan.

```
public class Zapisi{
    private ObjectOutputStream output; // objekt za unos u datoteku
    public void openFile(){
        try {
            output = new ObjectOutputStream(
                new FileOutputStream("datoteka.ser") );
        } catch (FileNotFoundException ex) {}
        try {
            input = new ObjectInputStream(
                new FileInputStream( "datoteka.ser" ) );
        } catch (FileNotFoundException ex) {}
    }
    public void readRecords(){
        Korisnik korisnik;
```

```
        while (true){
            record = (Korisnik) input.readObject();
            //daljnja obrada podataka
        }
    }
    public void closeFile(){
        if ( input != null ) input.close();
    }
}
```

Primjer kôda 9.11 Čitanje serijaliziranih objekata iz datoteke.

9.4 Korištenje spremnika

Korištenje spremnika (engl. *buffering*) predstavlja metodu poboljšanja perfomansi ulazno/izlaznih operacija. Korištenjem `BufferedOutputStream` (podklasa klase `FilterOutputStream`) pojedinačna naredba za upis ne mora nužno rezultirati fizičkim prijenosom podataka. Umjesto toga, preusmjerava se u dio memorije koji se naziva spremnik (engl. *buffer*), koji je dovoljno velik i može sadržavati veći dio izlaznih operacija (podataka). Premda se spremnik prazni nakon što se napuni (engl. *logical output operation*), pražnjenje se može i eksplicitno zatražiti pozivanjem metode `flush`.

```
FileInputStream fis = new FileInputStream("datoteka.txt");
//Stvaranje BufferedInputStream objekta.
BufferedInputStream bis = new BufferedInputStream(fis);
int i;
//citanje datoteke.
while((i=bis.read())!=-1){
        System.out.print((char)i);
}
```

Primjer kôda 9.12 Čitanje datoteke uz pomoć `BufferedInputStream`.

```
String str = "Nekakav primjer recenice";
FileOutputStream fos = new FileOutputStream("datoteka.txt");
//Stvaranje BufferedOutputStream objekta.
BufferedOutputStream bos = new BufferedOutputStream(fos);
byte b[]=str.getBytes();
bos.write(b);
bos.flush();
bos.close();
```

Primjer kôda 9.13 Pisanje u datoteku uz pomoć `BufferedOutputStream`.

9.5 Klasa `File`

Klasa `File` koristi se za dohvaćanje informacija o datotekama ili direktorijima. Sastoji se od četiriju konstruktora gdje:

- prvi konstruktor (`File(String)`) – prima `String` kao argument koji definira naziv datoteke ili direktorija. Naziv može sadržavati informacije o putanji (apsolutna – uključuje sve direktorije počinjući od korjenskog direktorija, relativna – počinje iz direktorija iz kojeg se aplikacija izvodi);
- drugi konstruktor (`File(String, String)`) – prima dva stringa gdje je prvi argument relativna ili apsolutna putanja, a drugi datoteka ili direktorij;
- treći konstruktor (`File(File, String)`) – prvi argument odnosi se na postojeći objekt tipa `File` koji definira roditeljski direktorij ili direktorij određen drugim argumentom

```
    File(direktorij, "datoteka");
```

- četvrti konstruktor (`File(URI)`) – argument predstavlja URI (engl. *Uniform Resource Identifier*).

```
File("file://C:/datoteka.txt");
File("file:/usr/home/dokumenti/dokument.txt");
```

Jednom kada smo instancirali objekt klase `File`, na raspolaganju nam je mnoštvo korisnih metoda pomoću kojih možemo rukovati datotekama / direktorijima.

Tablica 9.1: Metode klase `File`.

Naziv metode	Opis
`boolean canRead()`	Provjerava je li datoteka čitljiva iz trenutne aplikacije (vraća `true` ili `false`).
`boolean canWrite()`	Provjerava je li omogućeno pisanje u datoteku iz trenutne aplikacije (vraća `true` ili `false`).
`boolean exists()`	Provjerava postoji li datoteka ili direktorij (vraća `true` ili `false`).
`boolean isFile()`	Provjerava radi li se o datoteci (vraća `true` ili `false`).
`boolean isDirectory()`	Provjerava radi li se o direktoriju (vraća `true` ili `false`).
`boolean isAbsolute()`	Provjerava radi li se o apsolutnoj putanji (vraća `true` ili `false`).
`String getAbsolutePath()`	Vraća apsolutnu putanju datoteke ili direktorija.
`String getName()`	Vraća naziv datoteke ili direktorija.
`String getPath()`	Vraća putanju datoteke ili direktorija.
`String getParent()`	Vraća roditeljski direktorij datoteke ili direktorija (direktorij u kojem se nalaze).
`long length()`	Vraća duljinu datoteke u oktetima. Može se pozvati i za direktorije, ali je rezultat platformski ovisan.
`long lastModified()`	Vraća platformski ovisno vrijeme u kojem su datoteka ili direktorij manipulirani.
`String[] list()`	Vraća sadržaj direktorija u obliku polja. Ne odnosi se na datoteke (vraća null).

```
public static void analizirajPutanju(String putanja){
    // stvori File objekt na temelju korisnickog unosa
    File naziv = new File(putanja);
    if ( naziv.exists() ) {
        // ispisi informacije o datoteci ili direktoriju
        System.out.println(naziv.getName()+" postoji!");
        System.out.println(naziv.isFile()?" je datoteka":"nije datoteka");
        System.out.println(naziv.isDirectory()?" je direktorij":
                           "nije direktorij");
        System.out.println(naziv.isAbsolute()?"je apsolutna putanja":
                           "nije apsolutna putanja");
        System.out.println("Modificirano: "+naziv.lastModified());
        System.out.println("Duljina: "+naziv.length());
        System.out.println("Putanja: "+naziv.getPath());
        System.out.println("Apsolutna putanja: "+naziv.getAbsolutePath());

        if(naziv.isDirectory()){
            String[] directory = naziv.list();
            System.out.println( "\n\nSadrzaj direktorija:\n" );
            for(String directoryName : directory){
                System.out.println(directoryName);
            }
        }
    }else{
        System.out.println( "%s %s", putanja, "ne postoji." );
```

```
    }
}
```

Primjer kôda 9.14 Upotreba metoda klase `File`.

U primjeru kôda 9.14 prikazana je upotreba navedenih metoda u tablici 9.1.
Za ulazni parametar `"./"` u metodu `analizirajPutanju()` dobit ćemo sljedeći izlaz u konzoli:

```
. postoji!
nije datoteka
 je direktorij
nije apsolutna putanja
Modificirano: 1595335002000
Duljina: 288
Putanja: .
Apsolutna putanja: /Users/tomislavgalba/NetBeansProjects/Test/.

Sadrzaj direktorija:

manifest.mf
test
build.xml
datoteka.ser
nbproject
build
src
```

Napomena: prikazani izlaz u konzoli odnosi se na računalo na kojem je rađeno testiranje. Sadržaj direktorija kao i ostali podaci u izlazu mogu se razlikovati na drugim računalima.

9.6 Zadaci

Zadatak 9.1. Napisati program koji će ispisivati hijerarhiju sistemskog diska (ispisati dodatne informacije kao što su `"direktorij"`, `"datoteka"`). Nakon što se ispiše hijerarhija, ponovno proći kroz sve, ali prilikom ispisa provjeriti

- ako naziv direktorija ili datoteke počinje slovom između `'a'` – `'n'`, onda treba malo početno slovo prepraviti u veliko;
- ako počinje slovom između `'o'` – `'z'`, onda ako je zadnje slovo malo, prepraviti kompletan naziv u velika slova, inače ostaviti nepromijenjeno.

Zadatak 9.2. Implementirati klasu koja predstavlja studenta (`maticniBroj`, `ime`, `prezime`, `fakultet`, `smjer`, `godina`, `prosjecnaOcjena`) s pripadajućim `get` i `set` metodama. Kroz petlju unijeti podatke za 10 studenata, zatim učitati te podatke u datoteku te nakon toga učitati te iste podatke i izračunati prosjek ocjena za sve studente. Koristiti serijalizaciju.

Zadatak 9.3. Napisati klasu `Opcina` koja u sebi sadrži privatne članove `kod1`, `kod2`, `kod3` i `nazivOpcine` s pripadajućim `get` i `set` metodama. Nakon toga, u klasi `Test` učitati podatke iz datoteke koja ima sadržaj kao što je to prikazano ispod ovog zadatka i spremiti ih u objekte. Objekte je potrebno spremiti u neku od kolekcija (odaberite sami odgovarajuću kolekciju).

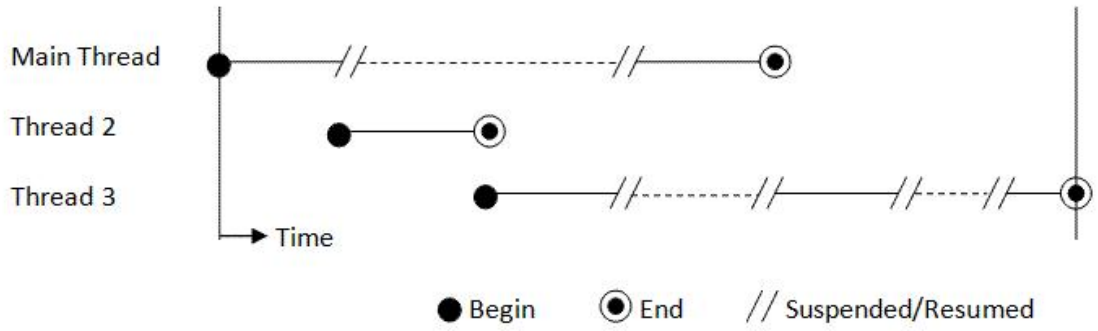
```
001    0019    00019    ANDRIJAŠEVCI
002    0027    00027    ANTUNOVAC
```

Poglavlje 10

Uvod u višenitnost

Do sada smo radili programe koji su se izvršavali u jednoj niti ili dretvi. U ovom priručniku koristit će se termin nit. Ulazna točka glavne niti (engl. *main thread*) jeste metoda `main` koju smo do sada imali u svakom programu u Javi. Svi takvi programi nazivaju se jednonitni (engl. *single-threaded*) programi. Osim jednonitnih programa programski jezik Java omogućava i razvoj višenitnih (engl. *multi-threaded*) aplikacija.
Jednonitni program ima jednu ulaznu točku i jednu izlaznu točku (kraj programa).
S druge strane, višenitni program inicijalno ima jednu ulaznu točku (metoda `main`) koju prati više ostalih ulaznih i izlaznih točaka koje se izvršavaju usporedno s metodom `main`.

Pojam višenitnosti predstavlja mogućnost obavljanja više radnji u isto vrijeme.



Slika 10.1: Dijagram izvršavanja višenitnog programa.

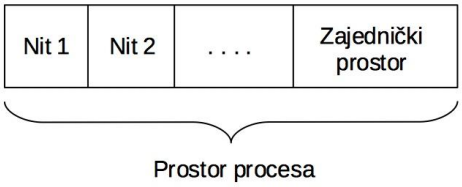
Napomena: Iako je glavna nit završila, program će se izvršavati sve dok postoje nezavršene niti koje je pokrenula glavna nit. Ako je potrebno završiti program sa završetkom glavne niti, onda moramo novu nit proglasiti pozadinskom niti, a to u Javi možemo vrlo jednostavno napraviti postavljajući svojstvo `.setDeamon(true)` za novu nit.

Proces predstavlja instancu izvršnog programa koji se izvršava u memoriji računala (svako pokretanje programa stvara novi proces). Svaki proces koristi zaseban memorijski prostor u kojem se nalaze podaci tog procesa (programski kôd, varijable i slično).

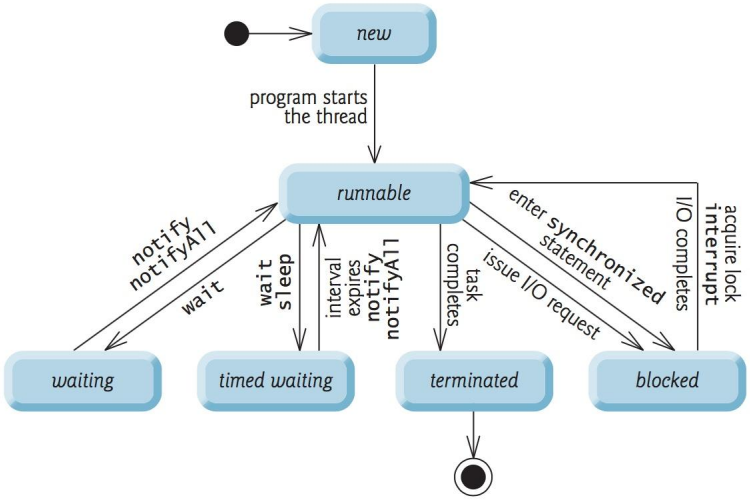
Niti predstavljaju najmanji dio procesa te služe za paralelizaciju na razini procesa. Drugim riječima, kôd procesa podijeljen je na više dijelova koji se izvršavaju paralelno. Sve niti unutar procesa dijele isti dio dodijeljene memorije.
Napomena: Stvaranje niti unutar procesa pripada operacijskom sustavu, što znači kako bi bili u mogućnosti stvoriti niti unutar programa u Javi, operacijski sustav to mora podržavati.

10.1 Stanja niti

Od svojeg stvaranja, svaka nit može se nalaziti u jednom od stanja koja su prikazana na slici 10.3.



Slika 10.2: Raspodjela dodijeljenog memorijskog prostora.



Slika 10.3: Moguća stanja niti i načini prelaska iz jednog u drugo stanje.
Izvor: `https://howtodoinjava.com/java/multi-threading/java-thread-life-cycle-and-thread-states/`, pristupljeno 3.4.2022.

10.1.1 Stanja `new` i `runnable`

Nova nit svoj životni vijek započinje u stanju `new` te ostaje u tom stanju dok ju program ne pokrene. Nakon pokretanja, nit prelazi u stanje `runnable`. Za nit u stanju `runnable` smatra se da obavlja svoj zadatak.

10.1.2 Stanje `waiting`

Ponekad je potrebno da nit prijeđe iz stanja `runnable` u stanje `waiting` kako bi mogla pričekati neku od niti koje se trenutno izvode da završe s izvođenjem zadatka. Kako bi se nit vratila u stanje `runnable` iz stanja `waiting`, potrebno je pričekati da ju druga nit obavijesti da nastavi s izvođenjem.

10.1.3 Stanje `timed waiting`

Nit iz stanja `runnable` može prijeći u stanje `timed waiting` na određeni interval vremena. Povratak u stanje `runnable` može se dogoditi nakon što određeno vrijeme prođe ili ukoliko dođe do događaja kojeg nit očekuje. Niti koje se nalaze u stanju `waiting` ili `timed waiting` ne mogu koristiti procesor iako je možda slobodan. Drugi način na koji je moguće nit staviti iz stanja `runnable` u `timed waiting` stanje je pozivanje metode `sleep`. Nit ostaje u tom stanju sve dok ne istekne zadano vrijeme (engl. *sleep interval*) nakon kojeg se nit vraća u stanje `runnable`.

10.1.4 Stanje `blocked`

Nit iz stanja `runnable` prelazi u stanje `blocked` kada pokušava izvršiti zadatak koji se ne može izvršiti odmah, nego se mora privremeno pričekati do završetka. Kao primjer može se navesti ulazno/izlazni zahtjev gdje operacijski sustav blokira nit dok se zahtjev ne obradi do kraja. Nakon toga, nit prelazi u stanje `runnable`. Nit koja se nalazi u stanju `blocked` ne može koristiti procesor iako je možda slobodan.

10.1.5 Stanje `terminated`

(engl. *dead state*)

Nit prelazi u stanje `terminated` (engl. *dead state*) nakon što uspješno izvrši zadatak ili u slučaju nastanka greške.

10.2 Stvaranje i pokretanje niti

Za stvaranje i pokretanje niti koristi se klasa Thread. Klasa Thread sadrži korisne metode koje ćemo korisne za rad s nitima:

- `start()` – započinje izvršavanje nove niti;
- `sleep(long vrijeme)` – zaustavlja vlastitu nit na određeni interval vremena;
- `Thread currentThread()` – vraća referencu na nit koja se trenutno izvršava;
- `String getName()` – vraća naziv niti;
- `setPriority(int prioritet)` – postavlja prioritet izvršavanja niti (1 za najmanji i 10 za najveći prioritet).

Postoje dva načina stvaranja niti:

- prosljeđivanje objekta `Runnable` – sučelje `Runnable` definira metodu `run()` u koju se treba implementirati kôd kojeg želimo da nit izvrši. Stvoreni objekt `Runnable` prosljeđuje se konstruktoru klase `Thread` kao što je prikazano u primjeru kôda 10.1.

```
public class Proba implements Runnable {
        public void run() { System.out.println("Nit se izvrsava"); }
        public static void main(String [] args) {
                new Thread(new proba()).start();
        }
}
```

Primjer kôda 10.1 Implementacija sučelja `Runnable`.

- nasljeđivanje klase `Thread` - klasa `Thread` sama po sebi također implementira sučelje `Runnable`. Nasljeđivanje klase `Thread` prikazano je u primjeru kôda 10.2.

```
public class Proba extends Thread {
        public void run() { System.out.println("Nit se izvrsava"); }
        public static void main(String args[]) {
                (new proba()).start();
        }
}
```

Primjer kôda 10.2 Nasljeđivanje klase `Thread`.

U pravilu i jedan i drugi primjer obavljaju istu radnju s istim rezultatom. Kada će se koristiti koji primjer ovisi isključivo o programeru iako se preporučuje modeliranje posla implementiranjem sučelja `Runnable` jer tako razdvajamo posao od načina njegovog izvođenja. Prvi primjer koristi objekt `Runnable`, koji je općenitiji primjer, jer na taj način klasa može naslijediti neku drugu klasu osim klase `Thread`.

10.2.1 Pauziranje izvođenja korištenjem metode `sleep()`

Metoda `Thread.sleep()` uzrokuje privremeno zaustavljanje izvođenja trenutne niti na određeni interval vremena. Koristi se kada želimo osloboditi resurse (procesor) za izvođenje drugih niti ili za određivanja koraka.

```
public static void main(String [] args){
        String sekunde[] = {"Jedan","Dva","Tri","Cetiri", "Pet"};
        for (int i = 0; i < sekunde.length; i++) {
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException ex) {}
                System.out.println(sekunde[i]);
        }
}
```

Primjer kôda 10.3 Pauziranje izvođenja niti korištenjem metode `sleep()`.

10.2.2 Korištenje prekida

Prekid (engl. *interrupt*) predstavlja signal niti da treba prestati raditi trenutni posao i početi raditi drugi posao (najčešće nit kompletno prestaje s izvođenjem). Kako bi mehanizam prekida radio kako treba, nit mora podržavati prekidanje (programer mora definirati ponašanje). Mehanizam prekida implementiran je korištenjem zastavica (engl. *interrupt status*). Klasa `Thread` ima implementirane tri metode za upravljanje prekidima:

- `public void interrupt()` – koristi se za slanje prekida. Pozivanje ove metode postavlja prekidnu zastavicu na `true`;
- `public static boolean interrupted()` – vraća prekidnu zastavicu i postavlja ju na `false` ako je bila `true`;
- `public boolean isInterrupted()` – vraća stanje prekidne zastavice (`true` ili `false`).

```
public class proba extends Thread {
        public void run() {
                try{
                        for(int i=0;i<10;i++){
                                System.out.println("Ispisujem poruku...");
                                Thread.sleep(1000);
                        }
                }catch(InterruptedException greska){ throw
                        new RuntimeException("Nit prekinuta...");}
        }
        public static void main(String [] args) {
                proba nit1 = new proba();
                nit1.start();
                try{ nit1.interrupt();
                }catch(Exception greska){}
        }
}
```

Primjer kôda 10.4 Korištenje Thread.sleep() metode uz Interrupt

```
...
// static metoda interrupted provjerava stanje trenutne niti
if (Thread.interrupted()) {
  throw new InterruptedException();
}
...
Thread nit;
// isInterrupted non-static metoda provjerava stanje niti za
// koju se poziva (instance metoda)
```

```
if(nit.isInterrupted()){
//...
}
```

Primjer kôda 10.5 Korištenje interrupt metode

10.2.3 Korištenje metode `join()`

Metoda `join()` omogućava da jedna ili više niti pričeka dok druga nit ne završi s poslom koji obavlja.

```
public class klasa_s_nitima {
        public static void main(String [] args){
                Thread th1 = new Thread(new proba(), "t1");
                Thread th2 = new Thread(new proba(), "t2");
                th1.start();
                try {
                        th1.join();
                } catch (InterruptedException ie) {
                        ie.printStackTrace();
                }
                th2.start();
                try {
                        th2.join();
                } catch (InterruptedException ie) {
                        ie.printStackTrace();
                }
                System.out.println("Sve niti su zavrsile s izvrsavanjem");
        }
}
```

Primjer kôda 10.6 Korištenje metode `join()`.

U primjeru 10.6 deklariramo dvije dodatne niti `th1` i `th2`. Nakon pokretanja svake od njih odmah pozivamo metodu `join()`, što znači da će se nakon pokretanja niti `th1` pričekati na njezin kraj i tek onda pozvati nit `th2` te također pričekati na njezin kraj prije bilo kojeg sljedećeg koraka.

10.2.4 Sinkronizacija niti

U situacijama kada više niti dijeli isti objekt, tada se treba posebno posvetiti načinu na koji se obrađuju podaci. Ako jedna nit obavlja proces ažuriranja vrijednosti dok u isto vrijeme i druga nit pokušava napraviti istu stvar, nije moguće odrediti koja će promjena biti napravljena.
Navedeni problem može se riješiti tako da se dodijeli ekskluzivno pravo pristupa samo jednoj niti u isto vrijeme. Za to vrijeme druge niti moraju čekati. Takav pristup naziva se sinkronizacija niti.

Najčešći je pristup sinkronizaciji korištenje monitora (engl. *monitors*) koji su ugrađeni u programski jezik Java. Svaki objekt ima monitor i bravu, čime se osigurava da samo jedna nit istovremeno može imati pristup.
Kako bi se osiguralo da svaka nit mora imati bravu kako bi izvršila dio kôda, kôd se mora smjestiti unutar bloka `synchronized`. Drugim riječima, nit mora zatražiti bravu kako bi izvršila naredbe unutar bloka `synchronized`. Nakon što blok `synchronized` završi brava se otpušta te neka od blokiranih niti može pristupiti (brava se otpušta i ukoliko dođe do greške unutar bloka).

Postoje dva načina sinkronizacije u Javi. Sinkronizacija metoda i sinkronizacija izraza (engl. *statements*).

```
class ProbaIspis {
        public void ispis() {
                try {
                        for(int i = 1; i <= 5; i++) {
                        System.out.println("Brojac ---"+i );
                }
                } catch (Exception e) { /* interrupt */ }
        }
```

```
}
class Proba extends Thread {
        private Thread t;
        private String naziv;
        ProbaIspis pi;
        Proba( String naz, ProbaIspis pisp) {
                naziv = naz;
                pi = pisp;
        }
        public void run() {
        pi.ispis();
        System.out.println("Nit " + naziv + " zavrsava.");
        }
        public void start () {
                System.out.println("Pocinje s izvrsavanjem nit "+naziv );
                if (t == null) {
                        t = new Thread (this, naziv);
                        t.start ();
                }
        }
}
public class TestNiti {
        public static void main(String [] args) {
                ProbaIspis pi = new ProbaIspis();
                proba t1 = new proba( "Nit - 1 ", pi );
                proba t2 = new proba( "Nit - 2 ", pi );
                t1.start();
                t2.start();
        }
}
```

Primjer kôda 10.7 Jednostavan program bez korištenja sinkronizacije

Za navedeni primjer kôda 10.7 koji prikazuje pozivanje metode na nesinkronizirani način mogući izlaz (svakim pokretanjem može biti drugačiji izlaz) u konzoli je:

```
Pocinje s izvrsavanjem nit Nit - 1
Pocinje s izvrsavanjem nit Nit - 2
Brojac ---1
Brojac ---1
Brojac ---2
Brojac ---3
Brojac ---2
Brojac ---3
Brojac ---4
Brojac ---5
Brojac ---4
Nit Nit - 2 zavrsava.
Brojac ---5
Nit Nit - 1 zavrsava.
```

Primjer kôda 10.8 prikazuje sinkronizaciju izraza. Obavlja se na način da nit dohvaća bravu na objekt ili klasu na koju se referencira u izrazu. Kao što je prikazano u primjeru, više niti ne može istovremeno koristiti objekt, već se mora čekati na izvršavanje niti (jedna po jedna).

Drugi način bio bi sinkronizacija metode gdje se blokira izvođenje određenog bloka naredbi neke klase i taj blok može izvoditi samo jedna nit istovremeno od strane više niti u isto vrijeme.

```
public void run() {
        synchronized(pi){
                pi.ispis();
```

```
        }
        System.out.println("Nit " + naziv + " zavrsava.");
}
```

Primjer kôda 10.8 Jednostavan program uz korištenje sinkronizacije.

Za primjer koda 10.8 i sinkronizirani način rada dobijamo mogući sljedeći izlaz u konzoli:

```
Pocinje s izvrsavanjem nit Nit - 1
Pocinje s izvrsavanjem nit Nit - 2
Brojac ---1
Brojac ---2
Brojac ---3
Brojac ---4
Brojac ---5
Nit Nit - 1 zavrsava.
Brojac ---1
Brojac ---2
Brojac ---3
Brojac ---4
Brojac ---5
Nit Nit - 2 zavrsava.
```

10.3 Korištenje okruženja `Executor`

Sučelje `Executor` opisuje objekte koji su sposobni izvršavati zadane poslove predstavljene pomoću sučelja `Runnable`. Način na koji se ti poslovi izvršavaju nije specificiran samim sučeljem te ne mora nužno uključivati pokretanje novih niti ili asinkrono izvršavanje.

Sučelje `Executor` deklarira metodu `execute()` koja prima objekt tipa `Runnable` i delegira njegovo izvršavanje konkretnoj implementaciji. Implementacija odlučuje hoće li se posao izvršiti u istoj niti, u novoj niti ili pomoću neke druge strategije izvršavanja.

Proširenje mogućnosti ovog sučelja definirano je kroz sučelje `ExecutorService`, koje dodatno omogućuje upravljanje životnim ciklusom izvršavanja te praćenje statusa zadanih poslova. Iako ni `ExecutorService` ne zahtijeva nužno izvršavanje poslova u posebnim nitima, njegove konkretne implementacije (poput onih temeljenih na bazenima niti) često koriste višenitno izvršavanje.

Prednosti korištenja okruženja `Executor`, poput ponovne iskoristivosti niti, kontrole broja aktivnih niti i poboljšanja performansi, proizlaze iz konkretnih implementacija sučelja, a ne iz samog sučelja `Executor`.

```
public class Proba implements Runnable{
    public void run(){
        try{
            System.out.println(Thread.currentThread().getName()
                               +" ulazi u sleep");
            Thread.sleep( 1000 );
        } catch ( InterruptedException exception ){ }
        System.out.println(Thread.currentThread().getName()
                               +" izlazi iz sleep-a");
    }
}
public class TaskExecutor{
    public static void main( String[] args ){
        Proba zadatak1 = new Proba( );
        Proba zadatak2 = new Proba( );
        Proba zadatak3 = new Proba( );

        ExecutorService threadExecutor = Executors.newCachedThreadPool();
        threadExecutor.execute( zadatak1 );
```

```
        threadExecutor.execute( zadatak2 );
        threadExecutor.execute( zadatak3 );
        threadExecutor.shutdown();
    }
}
```

Primjer kôda 10.9 Korištenje okruženja `Executor`.

Za primjer kôda 10.9 dobijemo sljedeći izlaz u konzoli:

```
pool-1-thread-1 ulazi u sleep
pool-1-thread-3 ulazi u sleep
pool-1-thread-2 ulazi u sleep
pool-1-thread-3 izlazi iz sleep-a
pool-1-thread-2 izlazi iz sleep-a
pool-1-thread-1 izlazi iz sleep-a
```

Napomena: Nazivi "pool-1-thread-1" i slični generirani su od strane implementacije sučelja `ExecutorService`.

10.4 Zadaci

Zadatak 10.1. Napisati klasu `LansirnaRampa` koja ima metodu `lansiraj` koja odbrojava od 10 do 0 i ispisuje brojač. Napisati klasu `Raketa` koja nasljeđuje klasu `Thread` i ima 2 argumenta (objekta klase): argument `ime` tipa `String` i argument `LansirnaRampa`. U konstruktoru se postavi `ime` i `LansirnaRampa`, a u metodi `run` ispiše ime rakete i najavi početak lansiranje te pokrene lansiranje (metoda `lansiraj` objekta `LansirnaRampa`). Nakon što metoda za lansiranje završi, ispiše ime rakete i poruku o polijetanju. Napisati klasu `Test` koja kreira nekoliko raketa i pokreće ih paralelno. Pokrenite program više puta i promotrite kako se rakete/niti kreiraju (redoslijed), kako izvršavaju i kako završavaju.

Zadatak 10.2. Izmijeniti gornji primjer tako da `Raketa` implementira sučelje `Runnable` (umjesto da naslijeđuje klasu `Thread`).

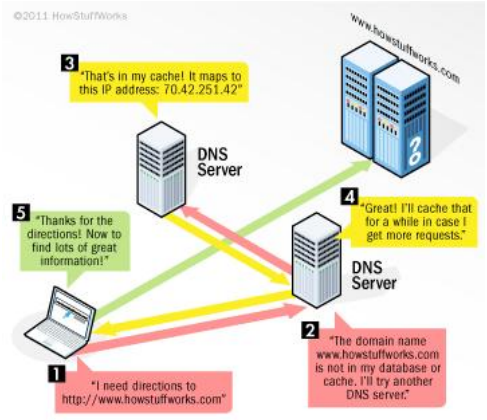
Zadatak 10.3. Napraviti metodu `lansiraj` u klasi `LansirnaRampa` tako da ju samo jedna nit može izvršavati u jednom trenutku (nema paralelnog izvršavanja više niti).

Zadatak 10.4. Napisati višenitni program koji se sastoji od klase `Test` i od klase `Brojac` koja nasljeđuje klasu `Thread`. Klasa `Brojac` (konstruktor) kao argument prima `0` ili `1`. Ako primi `0`, nit treba ispisivati jedinstvene parne brojeve 15 sekundi (svake sekunde jedan broj). Ako primi 1 treba ispisivati jedinstvene neparne brojeve 10 sekundi (svake sekunde jedan broj).

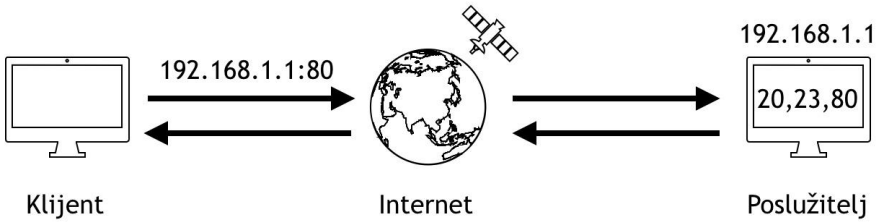
Poglavlje 11

Izrada mrežnih aplikacija

Danas većina aplikacija komunicira s barem jednim uređajem (npr. poslužitelj) putem lokalne mreže ili interneta. Kako bi dva uređaja mogla uspješno komunicirati potrebno je poznavanje njihovih IP (engl. *Internet protocol*) adresa. IP adresa sastoji se od 32-bitnog broja koji se najčešće predstavlja kao 4 dekadska broja razdvojena točkom (npr. 192.168.21.44). U svakodnevnom korištenju umjesto IP adresa predstavljenih brojem koristimo nazive koji se lakše pamte (npr. `ferit.hr`). U tu svrhu koriste se posebni poslužitelji koji se zovu DNS (engl. *Domain Name Server*) poslužitelji i oni služe kako bi preveli naš naziv lokacije na internetu u IP adresu.



Slika 11.1: DNS poslužitelj. Izvor: `https://computer.howstuffworks.com/dns.htm`, pristupljeno 3.4.2022.



Slika 11.2: Spajanje na udaljeno računalo (poslužitelj).

S obzirom da se na poslužitelju može nalaziti više usluga kojima se može pristupiti, potrebno ih je na neki način razlikovati. To radimo uz pomoć ulaza (engl. *ports*). Svakoj aplikaciji koja se nalazi na poslužitelju (npr. web server, ssh, ftp) dodijeljen je ulaz. Jedan takav primjer je web-poslužitelj Apache Tomcat koji dolazi s pridruženim ulazom 8080, što znači da on sluša na ulazu 8080 i čeka uspostavu veze.
Kada klijent želi otvoriti komunikaciju s poslužiteljem otvara se pristupna točka (engl. *socket*). U tekstu će se dalje koristiti termin socket. Socket predstavlja krajnju točku komunikacije (klijent/poslužitelj).
Komunikacija zasnovana na socketima omogućava aplikacijama komunikaciju na mreži na način da aplikacija može čitati ili pisati u socket (sličan princip kao i čitanje ili pisanje u datoteku).
Za komunikaciju računala koriste propisane protokole koji su zasnovani na TCP (engl. *Transmission Control Protocol*) ili UDP (engl. *User Datagram Protocol*) protokolima. TCP predstavlja fiksni kanal komunikacije između dvaju računala koji omogućava uspostavljanje veze i razmjenu tokova podataka. TCP jamči da će svi podaci biti isporučeni istim redosljedom kojim su poslani, a u slučaju gubitka ili nemogućnosti prijenosa, javlja se greška. Na protokolu TCP temelje se svima poznati protokoli kao što su HTTP, FTP, TELNET, SSH i drugi. UDP, s druge strane, prisiljava korisnika cijepkati podatke u pakete

(engl. *datagrams*). Poslani paketi ne moraju nužno doći istim redoslijedom kojim su poslani te se ne jamči njihova dostava. S druge strane, TCP nudi apstrakciju toka podataka te sam radi poslove cjepkanja, prijenosa, potvrđivanja, retransmisije u slučaju gubitka i slično.

Programski jezik Java pruža veliki broj ugrađenih mogućnosti za rad na mreži, što olakšava razvoj mrežnih aplikacija. Osnovne mogućnosti rada na mreži deklarirane su unutar paketa `java.net` koji omogućava aplikacijama dva načina komunikacije gdje je jedan zasnovan na toku podataka (engl. *stream based*), dok je drugi zasnovan na prijenosu paketa (engl. *packet based*).

Korištenjem komunikacije zasnovane na toku proces ostvaruje vezu s drugim procesom. Dok se veza održava, podaci se razmjenjuju neprekinutim tokom. Protokol koji se koristi za prijenos je TCP. Korištenjem komunikacije zasnovane na prijenosu paketa, prijenos se ostvaruje slanjem pojedinačnih paketa. Protokol koji se koristi za prijenos je UDP.

11.1 Struktura poslužitelja koji koristi protokol TCP

Korak 1 – stvaranje objekta `ServerSocket`

```
ServerSocket posluzitelj = new ServerSocket(brojUlaza, brojKorisnika);
```

Prvi argument (`brojUlaza`) predstavlja broj ulaza (port) za komunikaciju putem TCP protokola, a koristi se na strani klijenta kako bi se identificirala aplikacija na poslužitelju. Broj ulaza može biti između 0 i 65535 pri čemu dodjeljivanje broja ulaza može ovisiti o operacijskom sustavu koji može imati rezerviran određeni interval (najčešće su rezervirani brojevi 0 - 1024). Drugi parametar predstavlja maksimalni broj korisnika koji može čekati na uspostavu veze.

Korak 2 – čekanje na uspostavu veze

```
Socket connection = posluzitelj.accept();
```

Aplikacija upravlja sa svakim klijentom koristeći objekt klase `Socket`. Poslužitelj putem objekta `ServerSocket` čeka dolazak klijentske veze, pri čemu poziv metode `accept()` blokira izvršavanje dok se klijent ne spoji. Nakon uspostave veze, komunikacija između poslužitelja i klijenta odvija se putem ulaznog i izlaznog toka podataka povezanih s dobivenim objektom tipa `Socket`.

Korak 3 – dohvaćanje tokova okteta za čitanje i pisanje

Objekti `OutputStream` i `InputStream` omogućavaju komunikaciju (slanje i primanje podataka) između poslužitelja i klijenta. Poslužitelj poziva metodu `getOutputStream()` kako bi dobio referencu na socket `OutputStream` i metodu `getInputStream()` kako bi dobio referencu na socket `InputStream`. Objekti `Stream` mogu se koristiti za slanje/primanje niza okteta ili pojedinačnih okteta kao i za slanje/primanje primitivnih tipova ili serijaliziranih objekata.

```
ObjectInputStream input = new ObjectInputStream(connection.getInputStream());
ObjectOutputStream output = new ObjectOutputStream(connection.getOutputStream());
```

Korak 4 – obrada

Četvrti korak odnosi se na komunikaciju između poslužitelja i klijenta korištenjem objekata `OutputStream` i `InputStream`.

Korak 5 – zatvaranje veze

Nakon završetka toka poslužitelj raskida vezu pozivanjem metode `close()` na tok i socket.

```
try (ServerSocket echoServer = new ServerSocket()) {
    echoServer.bind(new InetSocketAddress("127.0.0.1", 9999));
    System.out.println("Postavljeni posluzitelj: " + echoServer);

    // cekamo klijenta
    try (Socket clientSocket = echoServer.accept();
        BufferedReader is = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
        PrintWriter os = new PrintWriter(
            new OutputStreamWriter(clientSocket.getOutputStream()), true)) {
        System.out.println("Server>> Imamo klijenta: "
                + clientSocket.getInetAddress());
```

```
        boolean done = false;
        while (!done && (linija = is.readLine()) != null) {
            os.println("Echo: " + linija.toUpperCase());
            if (linija.trim().equals("BYE")) {
                done = true;
            }
        }
    }

} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Primjer kôda 11.1 Implementacija poslužitelja.

11.2 Struktura klijenta koji koristi protokol TCP

Korak 1 – stvaranje socketa za spajanje na poslužitelj

```
Socket connection = new Socket(adresaPosluzitelja, ulaz);
```

Ukoliko je spajanje bilo uspješno, vratit će se socket spreman za komunikaciju tipa `Socket`. U suprotnom, izazvat će se iznimka `IOException`.

Korak 2 – dohvaćanje ulazno/izlaznog toka

U ovom koraku klijent koristi metode `getInputStream` i `getOutputStream` objekta tipa `Socket` kako bi dobio reference na `InputStream` i `OutputStream`.

Korak 3 – obrada

Ovaj korak odnosi se na komunikaciju između poslužitelja i klijenta korištenjem objekata `OutputStream` i `InputStream`.

Korak 4 – raskid veze

U četvrtom koraku klijent raskida vezu nakon što prijenos završi pozivanjem metode `close()` na tok i socket. Klijent mora prepoznati kada je poslužitelj završio sa slanjem podataka.

```
String host = "127.0.0.1";
int port = 9999;
try{
    // kreiramo soket za komunikaciju sa serverom
    Socket klijent = new Socket(host, port);
    System.out.println("Klijent: "+klijent);
    // input i output strimovi
    InputStream in = klijent.getInputStream();
    OutputStream out = klijent.getOutputStream();
    BufferedReader bis = new BufferedReader(new InputStreamReader(in));
    PrintWriter ps = new PrintWriter(new PrintStream(out),
                                     true /* autoflush */);
    // poruke koje ce nam posluzitelj vratiti nazad
    String[] eho = {"Poruka 1", "Poruka 2", "BYE"};
    // posaljemo poruke posluzitelju
    for(int i=0;i<eho.length; ++i) ps.println(eho[i]);
    // sad nam tu istu poruku vraca posluzitelj
    String linija=null;
    while((linija=bis.readLine()) != null) {
        System.out.println(linija);
    }
}
catch (Exception e){ e.printStackTrace(); }
```

Primjer kôda 11.2 Implementacija klijenta.

Primjer naveden ranije predstavlja tokovno-orijentiranu spojnu uslugu s pouzdanim prijenosom (engl. *connection-oriented transmission*) koji se odnosi na otvaranje toka između klijenta i poslužitelja prilikom čega veza ostaje otvorena bez obzira na to šalju li se podaci ili ne.
Suprotno tome, postoji paketno-orijentirana spojna ulsuga bez pouzdanog prijenosa (engl. *connectionless transmission*). Ovakav način prijenosa ne jamči da će podaci stići u obliku u kakvom su poslani niti da će stići u bilo kojem obliku.

```
public class Server {
    // socket za spajanje na klijenta
    private DatagramSocket socket;

    public Server(){
    try {
        socket = new DatagramSocket(5000);
    } catch (SocketException ex) {
        Logger.getLogger(Server.class.getName()).log(
                                        Level.SEVERE, null, ex);
    }
}
// cekanje na paket, prikaz sadrzaja, povrat klijentu
public void cekanjePaketa(){
    while(true){
        try{
            byte[] podaci = new byte[100]; //deklaracija paketa
            DatagramPacket prihvatniPaket = new
                            DatagramPacket(podaci, podaci.length);
            // cekanje na primitak paketa
            socket.receive(prihvatniPaket);

            prikaziPoruku("\nPaket primljen:"+
            "\nHost: "+prihvatniPaket.getAddress()+
            "\nPort: "+prihvatniPaket.getPort()+
            "\nDuljina: "+prihvatniPaket.getLength()+
            "\nSadrzi:\n\t"+new String(prihvatniPaket.getData(),
            0,prihvatniPaket.getLength()));

            posaljiPoruku(prihvatniPaket);
        }catch(IOException ioException){
            ioException.printStackTrace();
        }
    }
}
private void posaljiPoruku(DatagramPacket prihvatniPaket)
                                            throws IOException{
    // stvaranje paketa za slanje
    DatagramPacket sPaket = new DatagramPacket(prihvatniPaket.getData(),
    prihvatniPaket.getLength(), prihvatniPaket.getAddress(),
                                prihvatniPaket.getPort());
    // slanje paketa
    socket.send(sPaket);

    prikaziPoruku("Paket poslan\n");
}

private void prikaziPoruku(final String messageToDisplay){
    System.out.println("Poruka: "+messageToDisplay);
}
```

Primjer kôda 11.3 Implementacija poslužitelja (datagrami).

```
public class Client {
    // socket za spajanje na posluzitelj
    private DatagramSocket socket;
    // zadani konstruktor
    public Client(){
        try{
            socket = new DatagramSocket();
        }
        catch (SocketException socketException){ System.exit( 1 ); }
    }
    // cekanje na pakete, prikaz sadrzaja
    public void cekanjePaketa(){
        try{
            byte[] podaci = new byte[ 100 ]; // deklaracija paketa
            DatagramPacket prihvatniPaket = new
                            DatagramPacket(podaci, podaci.length );
            // cekanje na paket (blokira dok se paket ne primi)
            socket.receive(prihvatniPaket);
            // prikaz sadrzaja paketa
            prikaziPoruku( "\nPacket primljen:" +
            "\nHost: " + prihvatniPaket.getAddress() +
            "\nPort: " + prihvatniPaket.getPort() +
            "\nDuljina: " + prihvatniPaket.getLength() +
            "\nSadrzi:\n\t" + new String( prihvatniPaket.getData(),
            0, prihvatniPaket.getLength() ) );
            }
        catch ( IOException exception ){ exception.printStackTrace(); }
        posaljiPoruku();
    }
    // slanje poruke na posluzitelj
    public void posaljiPoruku(){
        Scanner input = new Scanner(system.in);
        System.out.println("Poruka: ");
        String unos = input.nextLine();
        byte[] podaci = unos.getBytes();
        try {
            // stvaranje paketa za slanje
            DatagramPacket sPaket = new DatagramPacket(podaci,
                podaci.length, InetAddress.getLocalHost(),5000);
            // slanje paketa
            socket.send(sPaket);
            cekanjePaketa();
        } catch (UnknownHostException ex) { ex.printStackTrace();
        } catch (IOException ex) { ex.printStackTrace(); }
    }
    // prikaz poruke
    private void prikaziPoruku(final String poruka){
        System.out.println("Poruka: "+poruka);
    }
}
```

Primjer kôda 11.4 Implementacija klijenta (datagrami).

11.3 Zadaci

Zadatak 11.1. Preraditi primjer kôda 11.1 tako da može opsluživati više klijenata u isto vrijeme (jedan klijent = jedna odvojena nit). Nakon toga potrebno je napraviti sljedeće:

a) podatke o klijentima spremati u listu i prilikom svakog novog spajanja, novom klijentu poslati podatke o svim spojenim klijentima;

b) implementirati jednostavnu funkcionalnost razmjene kratkih poruka (nakon uspješnog spajanja na poslužitelj i slanja poruke, poslužitelj bi svim spojenim klijentima trebao poslati sadržaj te poruke i podatak o pošiljatelju).

Poglavlje 12

Izrada grafičkog sučelja tehnologijom `Swing`

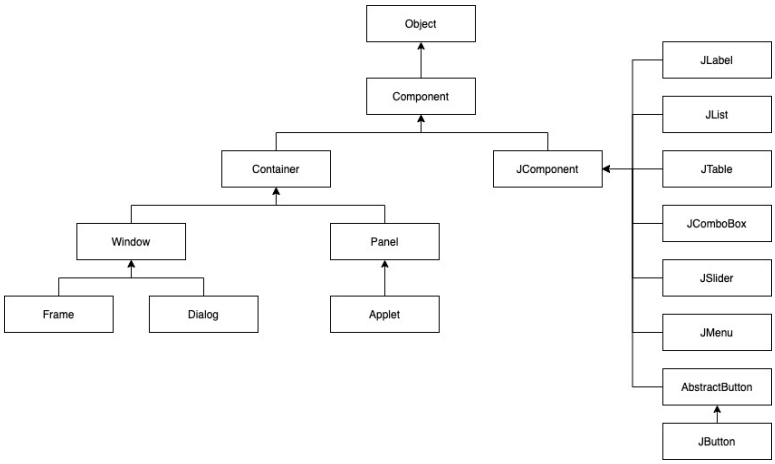
Kada pričamo o izradi grafičkog sučelja u programskom jeziku Java možemo spomenuti tri glavna načina za izradu sučelja koji su se mijenjali tijekom godina. Prvi paket za izradu grafičkog sučelja bio je `AWT` (engl. *Abstract Window Toolkit*). Odmah iza njega u Javi 1.2 dolazi paket `Swing` te on ostaje standard jako dugo vremena. U kasnijim verzijama Jave dolazi treći kao najnapredniji paket za izradu grafičkog sučelja `JavaFX` (o njemu će biti više govora u sljedećim poglavljima). S obzirom da paket `AWT` možemo smatrati zastarjelim, započet ćemo priču s paketom `Swing`.
Paket `Swing` (`javax.swing`) sadrži definicije svih komponenata za izradu grafičkog sučelja aplikacije. U potpunosti zamjenjuje paket `AWT`. Komponente `Swinga` smještene su u paketu `javax.swing`, dok se komponente `AWT` nalaze u paketu `java.awt`. Razlikuju se i po imenu, tako da nazivi svih komponenti `Swing` počinju sa slovom "J". Npr. AWT klasa koja definira gumb nazvana je `Button`, dok ekvivalentna Swing klasa nosi naziv `JButton`.

Svaki program koji koristi grafičko sučelje `Swing` mora na najvišoj razini sadržavati najmanje jedan Swing spremnik (engl. *container*). Na najvišoj razini postoje tri tipa spremnika:

- `JFrame` – implementira osnovni prozor aplikacije;
- `JDialog` – implementira pomoćne (dialog) prozore;
- `JApplet` – implementira područje za prikaz appleta unutar prozora internetskog preglednika (danas su ih napustili skoro svi preglednici).

12.1 Hijerarhija spremnika

Na slici 12.1 prikazana je hijerarhija komponenti (nisu prikazane sve komponente).



Slika 12.1: Hijerarhija klase `Component`.

Klasa `JFrame` (kao i `JDialog` i `JApplet`) je spremnik najviše razine i definira osnovni prozor aplikacije.
Klasa `JPanel` spremnik je srednje razine čija je jedina uloga pojednostavljenje pozicioniranja elementarnih komponenti.
Elementarne komponente (`JButton`, `JLabel` i dr.) ne koriste se za pohranjivanje ostalih komponenti grafičkog sučelja, nego sučelju daju funkcionalnost.

Na početku programa potrebno je uključiti paket s komponentama `Swinga`:

```
import javax.swing.*;
```

Budući da se `Swing` u nekim dijelovima još uvijek oslanja na `AWT`, često je potrebno uključiti i paket:

```
java.awt.*
```

Prikazani primjer sadrži samo jedan spremnik, JFrame. Osnovni prozor, definiran kao instanca klase JFrame, sadrži sve uobičajene dijelove: okvir, naslov i kontrolne gumbove.

Sljedeći kôd definirat će i prikazati jednostavan prozor:

```
JFrame prozor = new JFrame("Swing Program");
 . . .
prozor.pack();
prozor.setVisible(true);
```

Primjer kôda 12.1 Definicija najjednostavnijeg Swing prozora.

Nadalje, kako bi dodali nekakvu komponentu, npr. natpis (labelu) koja sadrži tekst `"Ovo je naš prvi Swing program"`, potrebno je napisati sljedeći dio:

```
// deklariramo natpis
JLabel natpis = new JLabel( "Ovo je naš prvi Swing program!");
// dodajemo labelu u prozor
prozor.getContentPane().add(natpis);
```

Primjer kôda 12.2 Deklaracija natpisa (engl. *label*).

Na kraju nam ostaje definicija ponašanja prozora nakon zatvaranja. Zašto je to potrebno napraviti? Svaki prozor kojeg stvorimo može imati nekoliko stanja nakon zatvaranja pa se, primjerice, prozor može sakriti (ali da sve vezano za prozor ostane zadržano u pozadini), a možemo ga i zatvoriti (i uništiti sve vezano za njega). Ako želimo zatvoriti i uništiti sve vezano za njega, nakon zatvaranja napisat ćemo sljedeći dio:

```
prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Primjer kôda 12.3 prikazuje osnovne dijelove koje treba sadržavati svaki Swing program, a koje smo prikazali u primjerima kôda 12.1 i 12.2:

```
public class PrviSwingProgram {
    public static void main(String[] args) {
        JFrame prozor = new JFrame("SwingProgram");
        JLabel natpis = new JLabel("Ovo je naš prvi Swing program!");
        prozor.getContentPane().add(natpis);
        prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        prozor.pack();
        prozor.setVisible(true);
    }
}
```

Primjer kôda 12.3 Jednostavno Swing sučelje.

Ako pogledamo primjer kôda 12.3, možemo primijetiti da se komponente dodaju uz pomoć metode `add`:

```
JLabel natpis = new JLabel("Ovo je naš prvi Swing program!");
prozor.getContentPane().add(natpis);
```

Primjer kôda 12.4 Dodavanje komponente pomoću metode add.

Metoda add inače sadrži najmanje jedan parametar (komponenta koja se dodaje). Međutim, ponekad se koriste dodatni parametri koji određuju smještaj komponente kao što je to prikazano u primjeru kôda 12.5.

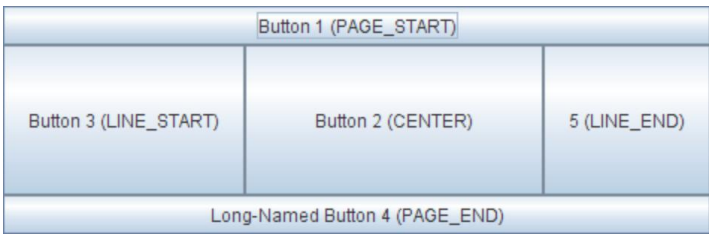
```
...
frame = new JFrame(...);
button = new JButton(...);
label = new JLabel(...);
pane = new JPanel();
pane.add(button);
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
...
```

Primjer kôda 12.5 Smještanje komponente na točno određeno mjesto.

12.2 Razmještaj komponenti

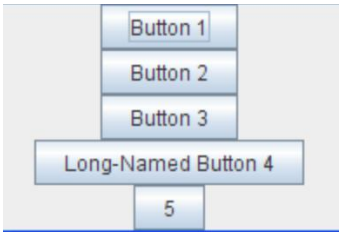
Za svako kvalitetnije grafičko sučelje potrebno je koristiti neki od predefiniranih načina razmještaja komponenti po sučelju (engl. *layouts*). Definirano je nekoliko načina smještaja komponenti od kojih su neki definirani za upotrebu u grafičkim alatima za izradu sučelja dok su neki predviđeni za primjenu iz kôda. Isto tako, većina načina razmještaja radi automatsko smještanje komponenti na sučelju, dok neki podržavaju i ručni način kao što je to `GridBagLayout`:

- `BorderLayout` – smješta komponente na način da prostor za smještaj podijeli na pet glavnih regija (PAGE_START, LINE_START, CENTER, LINE_END i PAGE_END).



Slika 12.2: Border layout razmještaj komponenti.

- `BoxLayout` – smješta komponente u redak ili stupac pri čemu uzima u obzir definirane dimenzije svake komponente.

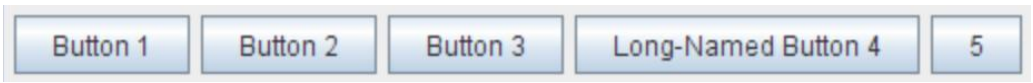


Slika 12.3: BoxLayout razmještaj komponenti.

- `CardLayout` – smješta komponente na način da omogućava prikaz drugačijeg sadržaja ovisno o korisnikovom unosu. Npr. ovisno o odabiru iz `ComboBoxa` prikazat će se određeni panel od kojih svaki ima drugačiji skup komponenti.

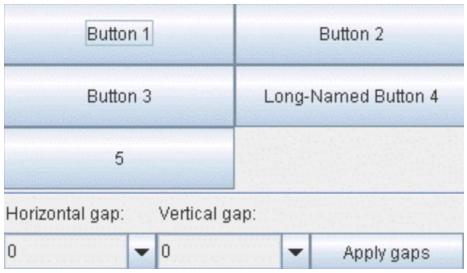
Slika 12.4: CardLayout razmještaj komponenti.

- `FlowLayout` – smješta komponente u jedan red (ukoliko jedan red nije dovoljan za smještaj svih komponenti stvorit će se dodatan). Ovaj razmještaj ujedno predstavlja i zadani razmještaj za svaki panel.



Slika 12.5: FlowLayout razmještaj komponenti.

- `GridLayout` – smješta komponente u određeni broj redaka i stupaca.

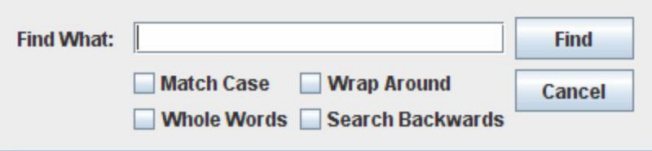


Slika 12.6: GridLayout razmještaj komponenti.

- `GridBagLayout` – možemo reći da ovaj razmještaj predstavlja napredniju inačicu razmještaja `GridLayout`, a omogućava smještaj komponenti u retke i stupce na fleksibilan način. Retci i stupci mogu imati različite dimenzije, a isto tako možemo raditi i spajanje (engl. *merge*) istih.

Slika 12.7: GridBagLayout razmještaj komponenti.

- `GroupLayout` – predviđen za upotrebu kao dio grafičkog sučelja za raspoređivanje komponenti, ali se može koristiti i iz kôda. Svaka se os tretira zasebno (horizontalno i vertikalno), odnosno svaka komponenta mora postojati u objema grupama (horizontalnoj i vertikalnoj).

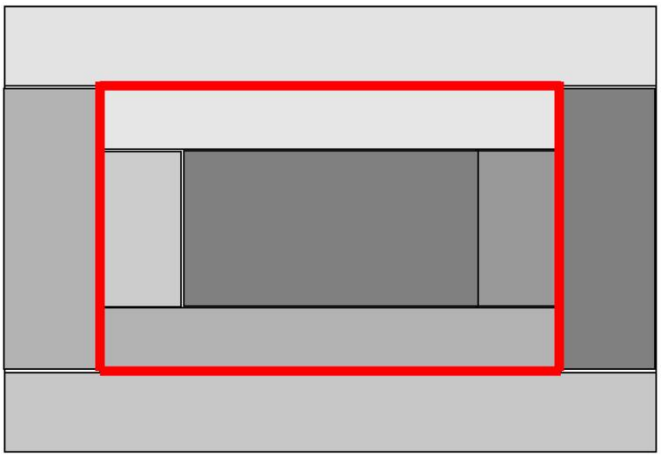


Slika 12.8: GroupLayout razmještaj komponenti.

- `SpringLayout` – predviđen za upotrebu kao dio grafičkog sučelja za raspoređivanje komponenti. Omogućava raspoređivanje komponenti ovisno o skupu korisnički zadanih ograničenja (engl. *constraints*), kao npr. udaljenost komponente od ruba susjedne komponente.

Slika 12.9: SpringLayout razmještaj komponenti.

Važno je napomenuti kako jedan panel može imati samo jedan upravljač razmještajem (engl. *LayoutManager*). Ako se želi složeniji razmještaj, onda se koristi niz panela koji se međusobno ugnijezde (ali i dalje svaki od njih ima jedan upravljač razmještajem koji se, naravno, može razlikovati od panela do panela).



Slika 12.10: Kombiniranje načina smještanja komponenti (layout-a).

Slika 12.10 prikazuje primjer kombinacije načina smještanja komponenti. Panelu je dodijeljen `BorderLayout` te se onda centralni dio podijelio na dodatnih pet regija s novim panelom kojim upravlja novi `BorderLayout` (označeno crvenom bojom).

Način smještaja komponenti odabire se metodom setLayout, npr:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

Primjer kôda 12.6 Postavljanje načina smještanja komponenti (layout-a).

Komponente se u pravilu smještaju na panel redoslijedom dodavanja. Način dodatne kontrole nad smještajem ovisi o odabranom načinu smještaja. Na primjer, za `BorderLayout` komponente se dodaju na sljedeći način:

```
panel.add(component1, BorderLayout.CENTER);
panel.add(component2, BorderLayout.SOUTH);
```

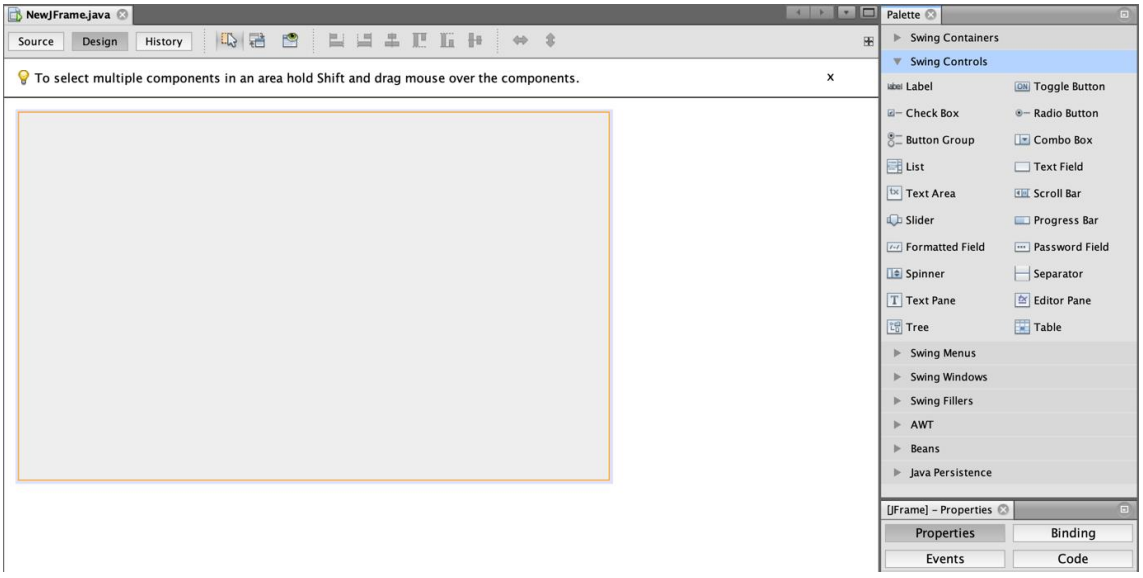
Primjer kôda 12.7 Dodavanje komponenti na točno određeno mjesto određeno načinom smještanja komponenti.

Iako se preporuča korištenje automatskog smještanja komponenti, postavljanjem na `null` moguće ga je isključiti. U tom slučaju svakoj komponenti potrebno je zadati veličinu i poziciju unutar prozora. Nedostatak je takvog načina smještanja komponenti gubitak mogućnosti automatske prilagodbe kod promjene veličine prozora ili promjene veličine teksta, npr. izvođenjem aplikacije na drugom računalu.

Korištenjem automatskog smještaja komponenti zadržava se mogućnost kontrole razmještaja – svakoj komponenti moguće je zadati minimalnu, preporučenu i maksimalnu veličinu, moguće je definirati prazan prostor oko komponente, poravnanje susjednih komponenti (npr. po gornjem rubu) i sl. Ukoliko nijedan od raspoloživih načina smještanja ne odgovara, moguće je definirati vlastiti.

12.3 Grafičko sučelje za uređivanje razmještaja komponenata

Kako je već spomenuto, osim slaganja sučelje iz kôda (programskim putem), sučelje možemo sastaviti i koristeći grafičko sučelje za smještanje komponenti. Svako malo bolje programsko okruženje koje podržava Javu ima već u sebi ugrađeno takvo grafičko sučelje (npr. NetBeans, Eclipse, IntelliJ itd.).



Slika 12.11: Grafičko sučelje za slaganje komponenti grafičkog sučelja.

Slika 12.11 je prikaz izgleda prozora razvojnog programskog okruženja NetBeans. U sredini se nalazi dio koji predstavlja panel, dok se s desne strane nalaze komponente i ostale postavke vezane za dodane komponente. Komponente se prevlače metodom drag & drop. Ukoliko se ne odredi neki od načina smještanja, automatski će se koristiti razmještaj `GroupLayout`.

12.3.1 Izgled sučelja

Budući da je većina Java-aplikacija direktno prenosiva između različitih platformi, posebna pažnja posvećena je izgledu grafičkog sučelja (engl. *look and feel*).

Izvorno su definirana tri izgleda sučelja prikazana slikama:

Napomena: Postoji još i prethodno definirani izgled pod nazivom „Nimbus“, a predstavlja moderniji izgled u odnosu na navedene. Za moderniji izgled sučelja potrebno je koristiti JavaFX, o kojem će biti govora u sljedećim poglavljima, ili je potrebno ručno redefinirati izgled svake komponente (što je moguće, ali iziskuje puno posla).

Aplikacija postavlja izgled sučelja prema sljedećim pravilima:

Slika 12.12: Metal sučelje.

Slika 12.13: Motif sučelje.

Slika 12.14: Windows sučelje.

1. Ako program inicijalno postavi izgled sučelja prije postavljanja komponenti, upravitelj korisničkog sučelja (engl. *UI manager*) pokušava stvoriti instancu zadane klase. Ukoliko u tome uspije, program koristi zadani izgled sučelja.
2. Ako program nije postavio izgled sučelja, upravitelj provjerava je li korisnik definirao željeni izgled sučelja u datoteci swing.properties, npr.:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

3. Ako niti program niti korisnik nisu postavili izgled sučelja, koristi se Java ("Metal") izgled.

Izgled sučelja postavlja se iz aplikacije pozivom metode `setLookAndFeel`. Izgled sučelja moguće je definirati direktno:

```
UIManager.setLookAndFeel( "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
// ili koristenjem neke od za to predvidjenih metoda, npr:
UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
```

12.3.2 Obrada događaja

Za razliku od klasičnih aplikacija u kojima se podaci u pravilu obrađuju slijedno (ciklus unos – obrada – ispis), aplikacije koje rade u grafičkom okruženju upravljane su događajima (engl. *events driven*). Događaji su najčešće generirani akcijama koje korisnik provodi nad sučeljem aplikacije: klik ili pomak miša, unos znaka i sl. Aplikacija se sastoji od niza međusobno nezavisnih metoda koje se aktiviraju određenim događajima.

Svaki događaj predstavljen je objektom koji sadrži informacije o događaju i izvoru događaja. Najčešće su izvori događaja komponente sučelja, ali to mogu biti i druge vrste objekata.
Svaka komponenta ima svoje specifičnosti vezane uz obradu događaja.

Budući da se cjelokupan kôd za obradu svih događaja izvršava u jednoj niti (engl. *event dispatching thread*), važno je da se kôd za obradu događaja izvršava brzo. U suprotnom će odziv aplikacije biti loš. Ako je kao rezultat određenog događaja potrebno izvršiti određenu operaciju duljeg trajanja, za to treba koristiti zasebnu nit.

Tablica 12.1: Primjeri događaja.

Akcija koja generira događaj	Vrsta događaja
Klik mišem na dugme, pritisak na tipku kod unosa teksta, odabir stavke izbornika.	`ActionListener`
Zatvaranje osnovnog prozora.	`WindowListener`
Pritisak na tipku miša dok je kursor iznad komponente.	`MouseListener`
Pomak kursora miša iznad komponente.	`MouseMotionListener`
Komponenta postaje vidljiva.	`ComponentListener`
Komponenta dobiva fokus tipkovnice.	`FocusListener`
Promjena odabira u tablici ili listi.	`ListSelectionListener`

Obrada događaja zahtijeva tri segmenta kôda:

- Kod deklaracije klase za obradu događaja potrebno je specificirati implementira li klasa sučelje `ActionListener` ili nasljeđuje klasu koja implementira to sučelje, npr.:

```
public class MojaKlasa implements ActionListener
```

- Potrebno je registrirati instancu klase za obradu događaja kao objekt koji osluškuje događaje za jednu ili više komponenti (engl. *event listener*), npr.:

```
component.addActionListener(instancaMojaKlasa /* ili: this */ );
```

- Dodati kôd koji implementira metode sučelja slušača (engl. *listener*), npr.:

```
public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if(source == komponenta){
                ...
        }
}
```

Napomena: slušač kao i odgovor na akciju možemo definirati korištenjem anonimnih klasa (o kojima će biti govora kasnije), a primjer bi izgledao:

```
dugme.addActionListener(new ActionListener( ) {
        public void actionPerformed(ActionEvent e) {
                ...
        }
});
```

Primjer kôda 12.8 Dodavanje slušača i definiranje akcije za gumb.

Primjer 12.9 prikazuje jednostavno sučelje koje u sebi ima dugmić koji reagira na korisnikovu akciju.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Dogadjaj {
    private static String pocetakTeksta = "Pritisnuto je: ";
    // koliko je puta korisnik pritisnuo gumb
    private int brojPritisaka = 0;
    // metoda kreira graficke komponente na kontejneru

    public Dogadjaj() {
        // sve se nalazi na:
        JFrame frame = new JFrame("Pritisni gumb!");
        // labela mora biti deklarirana s final jer je lokalna
```

```
        // koja se poziva unutar inner klase, uocite da varijabla
        // brojPritisaka nije mijenjana na taj nacin
        final JLabel label = new JLabel(pocetakTeksta + "0 ");
        JButton button = new JButton("Pritisni");
        // Postavimo i labelu i button. Kako imamo dva elementa moramo
        // voditi racuna o njihovom polozaju pa koristimo GridLayout
        frame.getContentPane().setLayout(new GridLayout(0, 1));
        frame.getContentPane().add(label);
        frame.getContentPane().add(button);
        frame.pack();
        // prikazimo ih
        frame.setVisible(true);
        // da mozemo zatvoriti graficku aplikaciju
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // metoda koja procesira pritiske misom na gumb
        button.addActionListener(new ActionListener() {
            // unutarnja klasa
            public void actionPerformed(ActionEvent e) {
                brojPritisaka++;
                label.setText(pocetakTeksta + brojPritisaka);
            }
        });
    }
    public static void main(String[] args) {
        Dogadjaj dogadjaj = new Dogadjaj();
    }
}
```

Primjer kôda 12.9 Jednostavno grafičko sučelje – kôd za stvaranje.

Rezultat pokretanja aplikacije može se vidjeti na slici 12.15.

Slika 12.15: Rezultat pokretanja primjera koda 12.9.

12.4 Zadaci

Zadatak 12.1. Izraditi jednostavan kalkulator s brojevima i osnovnim matematičkim operacijama. Najjednostavnija verzija: korisnik unosi prvi broj, operaciju i drugi broj i ispisuje se rezultat na pritisak znaka "=". Rješenje se može proširiti tako da prima `N` brojeva s različitim operacijama među njima.

Zadatak 12.2. Izraditi grafičko sučelje za jednostavnu igru pogađanja. Korisnik na početku odabire/unosi jedan broj `N` koji će predstavljati `N×N` matricu. U sljedećem prozoru generira se matrica u obliku `N×N` gumba koji imaju dodijeljene vrijednosti (ali ih ne prikazuju). Također, jedna od generiranih vrijednosti odabrana je u pozadini i korisnik mora pogoditi u kojem se polju ta vrijednost nalazi (npr. slučajno odabrani broj kojeg korisnik mora pogoditi može se ispisati na vrhu prozora). Kada korisnik pogodi polje gumb treba promijeniti boju u zelenu i ispisati poruku korisniku (poruka po izboru), a ukoliko vrijednost nije pogođena, gumb se treba obojati u crveno.

Zadatak 12.3. (BONUS ili za kod kuće): napraviti dvije aplikacije (klijent i poslužitelj) koje razmjenjuju poruke preko socketa. Poruka unesena u jednoj aplikaciji ispiše se na drugoj i obrnuto.

Poglavlje 13

Anonimne klase i lambda-izrazi

Pored ugniježdenih i unutarnjih klasa, Java podržava i mogućnost *anonimnih* klasa. Deklaracija ovakvih klasa nalazi se uvijek unutar neke već postojeće klase (poput ugniježdenih ili unutarnjih), ali s tom razlikom da nije potrebno zadati ime, već samo napravimo (instanciramo) primjerak objekta već postojeće klase ili nekog Java sučelja. Glavna je prednost korištenja anonimnih klasa veća sažetost programskog kôda.

Kako bi Java išla u korak s modernim programskim jezicima, od verzije SE 8 uvodi se podrška za tzv. *lambda-izraze*. Lambda-izrazi su težnja za podrškom funkcijskog programiranja kao već dobro poznate paradigme koja se uvodi kako bi se postigla:

- još veća sažetost i razumljivost kôda;
- alternativa korištenju anonimnih klasa;
- što jednostavnija podrška za suvremene hardverske mogućnosti paralelne obrade podataka.

13.1 Anonimne klase

Zanimljiva stvar kod korištenja anonimnih klasa jest mogućnost pravljenja klasa 'za jednu uporabu' točno na određenom mjestu gdje nam treba. Sama je sintaksa takva da reducira nepotreban programski kôd te tako obično pojednostavljuje i samo razumijevanje i daljnje održavanje.

Ako se anonimne klase pišu kao lokalne klase, one samim time imaju i sve restrikcije koje vrijede za takve klase. Primjerice, u anonimnim klasama nije dopušteno pisanje nikakvih statičkih metoda, klasa, polja, itd. Moguće je korištenje statičkih konstanti tipa `final`. Dodatna je restrikcija da anonimne klase ne mogu imati konstruktor. Baš zbog tog razloga uvedeni su tzv. inicijalizatori primjerka klase koji su uvedeni u sintaksu upravo zbog toga što nije moguće imati konstruktor. Naposljetku, ako imamo potrebu za višestrukim primjercima objekata, onda je bolje koristiti lokalne klase, nego anonimne klase.

13.1.1 Sintaksa anonimnih klasa

Najprije bi trebalo spomenuti obavezno korištenje ključne riječi `new` kojom uvijek započinje definicija anonimne klase. Samim time odmah je jasno da se pri definiciji anonimne klase odmah stvara i novi primjerak objekta. Uporaba je uvijek unutar nekakve 'vanjske' klase, pa možemo reći da osnovna sintaksa izgleda ovako:

```
class VanjskaKlasa {
        mojObjekt = new Tip( listaParametara ) {
                // tijelo anonimne klase
        };
}
```

Primjer kôda 13.1 Generički primjer sintakse anonimne klase.

Neke su posebnosti koje možemo odmah uočiti iz ovog primjera sljedeće:

- stvaranje novog objekta klase `Tip`;

- moguće je imati više parametara odvojenih zarezom;
- postojanje znaka ';' iza vitičaste zagrade – to je zato što se anonimna klasa uvijek definira unutar izraza (naredbe) unutar vanjske klase, zato je ovaj znak obavezan.

Tip ovdje može biti u dvama oblicima:

- *nadklasa* koju naša anonimna klasa nasljeđuje;
- *sučelje* koje naša anonimna klasa implementira.

Uobičajenija je uporaba implementacija sučelja, pa zato slijedi takav primjer:

```
interface Automobil {
        public void vozi();
}
class vanjskaKlasa {
        public void createClass() {
                Automobil a = new Automobil() {
                        public void vozi() {
                                System.out.println("Brm brm brm...");
                        }
                };
                a.vozi();
        }
}
class Main {
        public static void main(String[] args) {
                vanjskaKlasa vk = new vanjskaKlasa();
                vk.createClass();
        }
}
```

Primjer kôda 13.2 Anonimna klasa koja implementira sučelje.

Posebno je označen dio kôda koji implementira sučelje `Automobil`.

Zadatak 13.1. Pokušajte umjesto sučelja napraviti nadklasu `Automobil` tako da implementirate metodu `vozi()` koja ispisuje neki drugi tekst. Pokrenite program. Koji tekst program ispisuje i zašto?

13.1.2 Uporaba anonimnih klasa u Java FX aplikacijama

Anonimne klase često se koriste kod grafičkih korisničkih sučelja (engl. *GUI – graphical user interface*). Java FX je moderna biblioteka za izgradnju GUI-a koji se danas vrlo često koristi za izradu aplikacija u Javi. Kada počinjemo učiti Java FX, jedan je od osnovnih primjera tzv. "`Hello world`" primjer:

```
public class HelloWorld extends Application {
        public static void main(String[] args) {
                launch(args);
        }
        @Override
        public void start(Stage primaryStage) {
                primaryStage.setTitle("Hello World!");
                Button btn = new Button();
                btn.setText("Say 'Hello World'");

                btn.setOnAction(new EventHandler<ActionEvent>() {
                        @Override
                        public void handle(ActionEvent event) {
                                System.out.println("Hello World!");
                        }
                });
```

```
                StackPane root = new StackPane();
                root.getChildren().add(btn);
                primaryStage.setScene(new Scene(root, 300, 250));
                primaryStage.show();
        }
}
```

Primjer kôda 13.3 Jednostavna `JavaFX` aplikacija koja koristi anonimnu klasu.

U ovom je primjeru naglašen dio kôda koji predstavlja anonimnu klasu. Ona je ovdje zadana kao parametar metode `setOnAction`.
Ova anonimna klasa zapravo implementira sučelje `EventHandler<ActionEvent>`. Ovo sučelje ima samo jednu metodu metodu tipa `void` koju smo implementirali tako da ispisuje tekst u konzolu: `handle(ActionEvent event)`.
Uočimo također u ovom primjeru i anotaciju `@Override` koja nije postojala u prethodnom primjeru. U općem slučaju, sučelja mogu imati i više metoda te ih sve možemo preopteretiti (engl. *override*) korištenjem anonimnih klasa. Isto vrijedi i ako pomoću anonimnih klasa nasljeđujemo neku nadklasu; ona isto tako može imati više metoda koje možemo preopteretiti (ponovno implementirati) u našoj anonimnoj klasi.

Zadatak 13.2. Preuredite program tako da dodate još jedan gumb u gore danu aplikaciju koji će, kad se pritisne, u konzolu ispisati pozdrav na hrvatskom jeziku. Posebno obratite pozornost na to da je potrebno napraviti anonimnu klasu na isti način kao parametar funkcije `setOnAction` za taj drugi gumb. Pokrenite aplikaciju i isprobajte.

Zadatak 13.3. Pokušajte uključiti klasu `TextField` koja je dio biblioteke `Java FX` i koja omogućava prikaz kućice za unos tekstualnog podatka. Proučite kako napraviti takav objekt te ga ugraditi u Vašu aplikaciju. Nakon toga implementirajte još jedan gumb te funkcionalnost da nakon pritiska tog gumba u konzolnom ispisu dobivamo tekst `"Pozdrav "` + ono što je korisnik unio u polje tipa `TextField`.

13.2 Lambda-izrazi

Prije verzije Java SE 8, podržane su bile samo tri programerske paradigme:

- proceduralno programiranje;
- objektno-orijentirano programiranje;
- generičko programiranje.

Java SE 8 uvodi novu paradigmu: *funkcijsko programiranje*.
Neke od prednosti funkcijskog programiranja mogu biti brže programiranje, sažetiji programski kôd te puno manja mogućnost za pogreške u kôdu. Također, treba spomenuti i puno lakši način paralelne obrade podataka, koju možemo vrlo lako implementirati na ovaj način.
Glavni je naglasak kod funkcijskog programiranja na tome da stavimo naglasak na to što želimo učiniti, a ne na to kako ćemo to postići. Uzmimo primjerice situaciju gdje želimo izračunati sumu cjelobrojnih vrijednosti nekog niza cijelih brojeva `V`. Tradicionalni bi programski kôd otprilike izgledao ovako:

```
int suma = 0;
for (int brojac = 0; brojac < V.length; brojac++)
        suma += V[brojac];
```

Većina programera bila bi sasvim zadovoljna ovim rješenjem. Međutim, u ovom rješenju na više mjesta je moglo doći do pogreške. Primjerice, `suma` ili `brojac` nisu inicijalizirani na pravu vrijednost ili korak promjene brojača je pogrešan, a isto tako mogli smo pogriješiti i kod samog zbrajanja u sumu. Ovakav način rješavanja možemo nazvati još i vanjska ili eksterna iteracija. Njene su karakteristike postojanje varijabli koje se tijekom izvođenja mijenjaju, koje još dodatno trebamo i deklarirati.

Unutarnja ili interna iteracija bila bi ono što funkcijsko programiranje nudi kao alternativno i još jednostavnije rješenje. Ideja je da jednostavno kažemo „evo imam niz, izračunaj mi sumu njegovih elemenata". Ta bi se suma računala interno, tj. mi ne bismo morali raditi petlju `for`, a osim toga niti

deklarirati nikakve pomoćne varijable. Samim tim izbjegli bismo mnoge potencijalne pogreške u kôdu, ali tu nije kraj. Vrlo jednostavno, možemo lako reći jednoj takvoj biblioteci da to sumiranje napravi na paralelan način, tj. iskoristi prednosti višejezgrene arhitekture današnjih računala te tako značajno poveća performanse. Isto tako, biblioteka bi trebala voditi računa i o tome da se naš niz, ili bilo kakav drugi izvor podataka, ne promijeni nakon obavljene operacije. O tome kako postići takvu funkcionalnost upravo će biti riječi u nastavku.

13.2.1 Funkcijska sučelja

Posebna vrsta sučelja koja imaju samo *jednu* apstraktnu metodu nazivaju se funkcijska sučelja. Mogu se označavati i kraticom SAM (engl. *Single Abstract Method*). Ovakva se sučelja često koriste kod funkcijskog programiranja jer ona zapravo predstavljaju objektno-orijentirani model funkcije.
U paketu `java.util.function` postoji definirano 6 osnovnih generičkih funkcijskih sučelja koja koriste generičke tipove podataka, tako da ih možemo koristiti s bilo kakvim klasama kao parametrima. Postoji još puno drugih specijaliziranih funkcijskih sučelja koja su definirana u ovom paketu.

Tablica 13.1: Šest osnovnih generičkih funkcijskih sučelja u paketu `java.util.function`.

Sučelje	**Metoda** koju sadrži	**Parametri** meteode	**Povratni** tip	**Opis**
`BinaryOperator <T>`	`apply`	`T,T`	`T`	Obavlja operaciju (obično računsku) s parametrima i vraća rezultat.
`Consumer <T>`	`accept`	`T`	`void`	Obavlja nekakvu zadaću sa zadanim objektom, npr. ispisuje ga.
`Function <T,R>`	`apply`	`T`	`R`	Poziva neku metodu na argumentu te vraća rezultat te metode.
`Predicate <T>`	`test`	`T`	`boolean`	Ispituje zadovoljava li argument određeni uvjet.
`Supplier <T>`	`get`		`T`	Često se koristi za stvaranje objekta kolekcije koji je povezan s tokom.
`UnaryOperation <T>`	`get`	`T`	`T`	Obavlja unarnu operaciju te vraća rezultat.

13.2.2 Sintaksa lambda-izraza

Lambda-izraz (engl. *lambda expressions*) zapravo predstavlja anonimnu metodu na određenoj razini apstrakcije, tj. kraću notaciju za implementaciju funkcijskog sučelja, slično kao što to radimo s anonimnim klasama koje su opisane u prethodnoj cjelini uz napomenu da se na implementacijskoj razini detalji mogu razlikovati u odnosu na anonimne klase. Tip lambda-izraza je upravo tip funkcijskog sučelja kojeg taj izraz implementira. Na mjestima na kojima se mogu koristiti funkcijska sučelja, tu možemo koristiti i lambda-izraze.

Osnovna sintaksa lambda-izraza sastoji se od liste parametara nakon koje slijedi znak strelice `->` i tijela izraza:

```
(listaParametara) -> {naredbe}
```

Sljedeći lambda-izraz prima dva cjelobrojna parametra i vraća njihovu sumu kao rezultat:

```
(int x, int y) -> {return x+y;}
```

Tijelo je u stvari tipični blok naredbi koji može sadržavati jednu ili više naredbi unutar vitičastih zagrada. Postoji nekoliko verzija sintakse. Jedna je od njih da se tipovi parametara ne navode te time prepuštamo prevoditelju da odredi tip podataka. U tom slučaju naš bi primjer izgledao ovako:

```
(x, y) -> {return x+y;}
```

Ukoliko tijelo ima samo jednu naredbu, izostavljamo i ključnu riječ `return` i isto tako vitičaste zagrade. U tome slučaju naš lambda-izraz svodi se na:

```
(x,y) -> x+y
```

Ovdje je vrijednost izraza implicitno vraćena. Ako lista parametara ima samo jedan parametar, onda se čak i obične zagrade mogu izostaviti. Primjer za to je:

```
cijena -> System.out.printf("%.2f",cijena)
```

Postoji i situacija kada uopće nemamo parametara. U tome slučaju moramo staviti otvorenu i zatvorenu zagradu na mjestu liste parametara, kao što je u sljedećem primjeru:

```
() -> System.out.println("Dobrodosli u svijet lambda-izraza!")
```

Treba još spomenuti kako postoje i specijalne kratice koje se nazivaju reference na metode, a predstavljaju specijalizirane forme lambda-izraza. Reference na metode spomenut ćemo poslije.

Zadatak 13.4. Definirajte klasu kompleksnog broja koja se sastoji od realnog i imaginarnog dijela koji su tipa `float`. Napisati lambda-izraze za:

a) zbrajanje dvaju kompleksnih brojeva;

b) množenje dvaju kompleksnih brojeva;

c) rotaciju kompleksnog broja za $+45^o$ oko ishodišta.

Pokušajte koristiti što sažetiju sintaksu ako je to moguće.

Zadatak 13.5. Napisati lambda-izraz koji ima `String` kao parametar, a vraća kao rezultat broj samoglasnika u tom stringu (bez obzira na mala i velika slova).

13.3 Kolekcijski tokovi

Java SE 8 uvodi i koncept *tokova* (engl. *streams*). Paket `java.util.stream` sadrži sučelje `Stream`, a tokovi su upravo objekti klasa koje implementiraju ovo sučelje ili specijalizirana sučelja za obradu kolekcija tipa `int`, `long` ili `double`. Tokovi omogućavaju izvođenje raznih zadaća nad kolekcijama, a tako i lambda-izrazi.

Tijek operacija ide obično u tri osnovna koraka

- na početku imamo izvor podataka (nekakva kolekcija objekata ili elementarnih tipova podataka);
- na toj kolekciji izvršavamo nekakve međuoperacije;
- naposljetku pozivamo terminalnu ili završnu operaciju.

Cjelokupna je naredba kao ulančani poziv metoda. Dok se obrađuje tok podataka, on se ne može ponovno koristiti jer se ne stvaraju kopije originalnog izvora podataka.

Osnovna razlika između međuoperacija i terminalnih operacija je u tome što međuoperacije uvijek kao rezultat vraćaju novi tok podataka, dok terminalne većinom vraćaju konkretan rezultat. Isto tako, terminalne se operacije izvršavaju odmah po pozivu, dok su međuoperacije tzv. 'lijene' operacije koje se izvršavaju samo po potrebi kada ih terminalne operacije trebaju. To omogućava bolje upravljanje performansama obrade toka podataka. U nastavku su dane tablice s najčešće korištenim međuoperacijama i terminalnim operacijama.

Tablica 13.2: Najčešće korištene međuoperacije kod toka podataka Međuoperacija.

Međuoperacija	Opis
`filter`	Vraća tok koji sadrži samo elemente koji zadovoljavaju neki uvjet.
`distinct`	Vraća tok koji sadrži samo jedinstvene elemente (bez istih vrijednosti!).
`limit`	Vraća tok koji sadrži točno određeni broj elemenata od početka.
`map`	Vraća tok u kojem su elementi preslikani u neku novu vrijednost (često je u pitanju i sasvim drugi tip podataka). Broj je elemanata zadržan.
`sorted`	Vraća tok u kojem su elementi poredani. Broj je elemenata zadržan.

Tokove u kontekstu funkcijskog programiranja ne treba miješati s ulazno/izlaznim tokovima podataka (engl. *I/O streams*) koji su spominjani u ranijim poglavljima. Ipak, funkcijskim programiranjem možemo obrađivati i podatke koji su smješteni u datotekama.

Tablica 13.3: Najčešće korištene terminalne (završne) operacije kod toka podataka.

Skupina	**Terminalna** operacija	**Opis**
	`forEach`	Obavlja obradu svih elemenata toka podataka.
Operacije redukcije	`average`	Računa srednju vrijednost svih elemenata.
	`count`	Vraća broj elemenata toka podataka.
	`min`	Locira najmanju vrijednost u brojčanom toku podataka.
	`max`	Locira najveću vrijednost u brojčanom toku podataka.
	`reduce`	Reducira elemente kolekcije u jednu vrijednost koristeći asocijativnu funkciju akumulacije (npr. lambda-izraz koji zbraja dva elementa).
Kreiranje spremnika	`collect`	Stvara novu kolekciju koja sadrži elemente koji su rezultati prethodne operacije s tokom podataka.
	`toArray`	Stvara niz koji sadrži rezultate prethodne operacije s tokom podataka.
Operacije pretraživanja	`findFirst`	Pronalazi *prvi* element na osnovi prethodnih međuoperacija i odmah prekida daljnju obradu toka.
	`findAny`	Pronalazi *bilo koji* element na osnovi prethodnih međuoperacija i odmah prekida daljnju obradu toka.
	`anyMatch`	Određuje zadovoljava li *bilo koji* element traženi uvjet i odmah prekida daljnju obradu ako takav element postoji.
	`allMatch`	Određuje zadovoljavaju li *svi* elementi u toku zadni uvjet.

13.3.1 Primjer korištenja lambda-izraza uz klasu `IntStream`

Neka je zadan primitivni niz cijelih brojeva:

```
int[] V = {5, 17, 6, 19, 22, 4, 8, 11, 35, 46, 92, 2, 27, 54};
```

Ispis ovog niza sada možemo ostvariti bez uobičajene petlje `for`.

Koristit ćemo gotovu klasu `IntStream` koja je definirana u paketu tokova podataka `java.util.stream.IntStream`. Lambda-izraz ćemo iskoristiti kako bismo ostvarili ispis:

```
IntStream.of(V).forEach( x -> System.out.printf("%d\n",x));
```

Rezultat je ispis podataka iz zadanog niza tako da je svaki podatak ispisan u novi red. Ukoliko, primjerice, želimo ispisati samo najveći podatak u tome nizu, onda nam čak niti ne treba lambda-izraz:

```
System.out.println( "Najveci je: " + IntStream.of(V).max().getAsInt() )
```

Izračunavanje sume kvadrata svih elemenata možemo postići odgovarajućim korištenjem međuoperacije `reduce` i lambda-izraza:

```
int sumakv = IntStream.of(V).reduce( 0, (x,y) -> x+y*y );
```

Primijetimo kako se u ovom slučaju koristi međuoperacija `reduce` te kako izgleda ispravan lambda-izraz.

Malo složenije bilo bi, primjerice, ispisati sve parne vrijednosti u sortiranom poretku:

```
IntStream.of(V).filter( x -> x%2==0 )
    .sorted()
    .forEach( x -> System.out.printf("%d\n",x) );
```

U ovom smo primjeru koristili dvije međuoperacije `filter` i `sorted` te na kraju terminalnu operaciju `forEach`. Isto tako na dva smo mjesta koristili lambda-izraze koji su posebno naglašeni.

Zadatak 13.6. Za gore zadani niz `V`, pokušajte napisati naredbu koja bi ispisala (koristiti lambda-izraze!)

a) broj elemenata u nizu `V` koji su veći od 50;

b) najveći neparan broj;

c) sve elemente koji su manji od srednje vrijednosti.

13.3.2 Korištenje referenci na metode

Kako bi pokazali korištenje reference na metode, uzmimo primjer niza `V` iz prošle točke i pokušajmo izračunati sumu korijena svih elemenata. To možemo postići na sljedeći način:

```
double sumakor = IntStream.of(V).asDoubleStream()
      .map(Math::sqrt)
      .sum();
```

U ovom primjeru posebno je označena referenca na metodu koja se koristi kako bi se pozvala statička metoda `sqrt` iz klase `Math`. Koristi se posebna sintaksa `Math::sqrt`. Isto tako, primijetimo da smo prvo morali prebaciti tok podataka cijelih brojeva u novi tok podataka tipa `double` pomoću metode `asDoubleStream()`. Nakon toga smo preslikali sve podatke u iznose njihovih korijena te naposljetku izračunali sumu terminalnom operacijom `sum`. Alternativno smo umjesto reference na metodu `Math::sqrt` mogli zapisati lambda-izraz poput

```
(double x) -> {return Math.sqrt(x);}
```

ili još kraće

```
x -> Math.sqrt( x )
```

Osim reference na statičke metode postoji ukupno četiri vrste referenci na metode opisanih u sljedećoj tablici.
Napomena: Značenje lambda-izraza i referenci na metode u Javi ovisi o kontekstu u kojem se izraz koristi, odnosno o funkcijskom sučelju na koje se izraz preslikava. Isti zapis reference na metodu može u različitim kontekstima rezultirati pozivom različitih metoda ili konstruktora.

Tablica 13.4: Vrste referenci na metode.

Vrsta reference na metodu	Primjer zapisa	Opis reference na metode
Referenca na statičku metodu.	`String::toUpperCase`	Referenca na metodu koja je instanca metode neke klase. Kreira jednoparametarski lambda-izraz koji poziva instancu metode za parametar lambda-izraza te vraća rezultat koji ta metoda vrati.
Referenca na instancijsku metodu proizvoljnog objekta određene klase.	`System.out::println`	Referenca na metodu za instancu metode koja treba biti pozvana na specifičnom objektu. Kreira jednoparametarski lambda-izraz koji poziva metodu na specifičnom objektu prenoseći parametar lambda-izraza instanci metode te vraća rezultat koji metoda vrati.
Referenca na instancijsku metodu određenog objekta.	`Math::sqrt`	Referenca na metodu koja je statička metoda klase. Kreira jednoparametarski lambda-izraz koji prenosi parametar lambda-izraza kao parametar statičke metode te vraća rezultat koji ta statička metoda vrati.
Referenca na konstruktor.	`TreeMap::new`	Referenca na konstruktor. Kreira lambda-izraz koji poziva konstruktor bez parametara specifične klase kako bi se kreirao i inicijalizirao novi objekt te klase.

Zadatak 13.7. Definirajte niz stringova. Koristeći klasu `java.util.Arrays` pretvorite taj niz u tok podataka metodom `stream` te preslikajte vrijednosti tog niza u vrijednosti s velikim slovima. U tu svrhu iskoristite referencu na metodu. Tako dobiveni niz ispišite.

Zadatak 13.8. Neka je zadana matrica cijelih brojeva:

```
Integer[][] nat = new Integer[][]{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Napisati izraz za pretvaranje ove matrice u matricu tipa `double [][]` korištenjem tokova podataka.
Savjet: koristite klasu `Arrays` i metodu `stream` kao u prošlom zadatku kako biste dobili tok cijelih brojeva, a nakon toga preslikajte taj tok u tok vrijednosti tipa `double` i na kraju taj tok preslikajte u traženu matricu. U zadnjem koraku koristite referencu na konstruktor klase `Arrays` kako bi izraz bio što sažetiji.

13.3.3 Manipulacija tokovima objekata

Do sada smo promatrali samo tokove elementarnih tipova podataka. Sada ćemo koristiti korisnički definiranu klasu `Student` te od nje napraviti tok podataka. Promotrimo sljedeći primjer programa:

```
class Student{
      protected String ime;
      protected String prezime;
      protected double prosjekOcjena;
      protected String JMBAG;

      public Student( String ime, String prezime,
                      double prosjekOcjena, String JMBAG ) {
          this.ime = ime; this.prezime = prezime;
          this.prosjekOcjena = prosjekOcjena; this.JMBAG = JMBAG;
      }

      @Override
      public String toString() {
          return String.format("-12s -12s 5.2f s", ime, prezime,
          prosjekOcjena, JMBAG );
      }
}
public class Main {
        public static void main(String[] args) {
            Student[] studenti = {
                new Student("Pero","Peric",4.22,"0165012345"),
                new Student("Marica","Maric",3.61,"0165000001"),
                new Student("Tiho","Tihic",4.07,"0165099999"),
                new Student("Miso","Misic",3.27,"0165055447"),
                new Student("Ana","Anic",4.60,"0165098765")};
            List<Student> lista = Arrays.asList(studenti);
            lista.stream().forEach(System.out::println);
        }
}
```

Primjer kôda 13.4 Manipulacija tokom objekata tipa `Student`.

U ovom smo primjeru formirali klasu `Student` u kojoj smo preopteretili metodu `toString` koja ispisuje podatke o studentu na neki poseban formatiran način koji smo mi zadali. U glavnoj klasi `Main` najprije smo definirali niz studenata te nakon toga, koristeći klasu `Arrays`, 'zamotali' niz studenata u listu i onda pretvorili dobivenu listu u tok korištenjem metode `stream()`. Posebno je označen dio kôda u kojem koristimo referencu na metodu s kojom smo se upoznali u prethodnoj cjelini, a pomoću nje ostvarujemo ispis svih studenata na vrlo jednostavan način.

Pokušajmo sada koristiti filter pomoću kojeg ćemo odabrati samo one studente kojima je prosjek ocjena u određenom rasponu. Pretpostavimo da želimo odabrati studente kojima je prosjek veći ili jednak od 3,5 i manji od 4,5 (studenti koji su 'prošli s 4'). Programski bi kôd izgledao ovako:

```
lista.stream().filter( e -> e.prosjekOcjena>=3.5 && e.prosjekOcjena<4.5)
            .sorted(Comparator.comparing((Student s) -> s.prosjekOcjena))
            .forEach(System.out::println);
```

U ovom primjeru dobivamo ispis svih studenata koji su 'prošli s 4' poredan od najmanjeg prosjeka prema najvećem. Posebno su označeni lambda-izrazi i referenca na funkciju koja pojednostavljuje ispis. Kako bismo mogli poredati po određenom članu nekog objekta, koristimo posebno sučelje `Comparator` u

kojem pomoću lambda-izraza određujemo član s obzirom na koji sortiramo podatke. Sučelje `Comparator` nudi niz statičkih metoda za stvaranje gotovih komparatora i ovdje je iskorištena upravo jedna od tih koja kao argument očekuje objekt koji će iz objekta koji se uspoređuje izvaditi i vratiti ključ prema kojem se radi usporedba. U ovom je primjeru takav objekt dan lambda-izrazom.

Sada ćemo izračunati zajednički prosjek svih studenata:

```
double ukPros = lista.stream().mapToDouble(s -> s.prosjekOcjena).average()
        .getAsDouble();
```

Tok objekata studenti najprije smo preslikali u tok vrijednosti `double`, a onda smo taj tok preslikali u kolekciju tipa `double` te na toj kolekciji pozvali završnu operaciju `average`. Poziv metode `getAsDouble()` neophodan je kako bismo dobili pravi povratni tip.

Zadatak 13.9. Proširite prethodni primjer, tako da pokušate dobiti:

a) ispis najboljeg studenta (po prosjeku);

b) ispis svih studenata koji su poredani od najboljeg uspjeha prema lošijem (koristiti metodu `reversed()` nakon poziva metode `comparing()` u sučelju `Comparator`);

c) ispisati studente poredane po prezimenu pa onda po imenu (proučiti sučelje `Comparator` i njegovu metodu `thenComparing()`);

d) preslikati sve studente u tok jedinstvenih stringova koji su jednaki njihovom prezimenu.

Zadatak 13.10. Dodajte člansku varijablu `spol` u klasu `Student`. Sukladno tome, izmijenite i konstruktor te u glavnoj klasi dodajte svakom objektu studenta i odgovarajući spol. Pokušajte sada grupirati studente po spolu. Ispišite koliki je postotak studenata, koliki studentica i ispišite grupirani ispis studenata po spolu. U što većoj mjeri koristiti tokove i lambda-izraze!

Savjeti:

– za grupiranje koristiti mogućnosti klase `Collectors`;

– posebno je zanimljiva metoda `groupingBy()`;

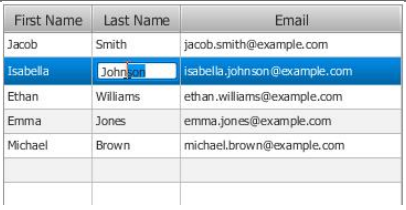
– za ispis po grupama spola najbolje je spremiti podatke u kolekciju `Map<String,List<Student>>`, gdje je prvi `String` naziv spola, a u listi je popis studenata toga spola.

Poglavlje 14

Biblioteka Java FX

Osim već spomenutog programskog okvira Swing, Java SE od 2008. godine uvodi Java FX kao alternativu za razvoj GUI desktop aplikacija (treba napomenuti kako je Java FX bio distribuiran unutar JDK-a, ali nije bio dio standardne platforme programskog jezika Java). Glavna je namjera autora zamijeniti dosadašnji Swing kao standard za takve aplikacije. Međutim, još su uvijek oba ova programska okvira aktualna.

Tablica 14.1: Neke od Java FX kontrola.

Naziv kontrole	Vrsta	Klasa	Mogući izgled
Gumb	Ulazna	`Button`	
Lista	Ulazno/Izlazna	`ListView`	
Postotak napredovanja nekog procesa	Izlazna	`ProgressBar`	
HTML uređivač teksta	Ulazna	`HTMLEditor`	
Tablica	Ulazno/Izlazna	`TableView`, `TableColumn`, `TableCell`	

Kao što je i sam jezik Java, i Java FX je osmišljen kao multiplatformski. Razvojem svojih verzija, krenuvši od verzije 1.0 koja je izdana u prosincu 2008., polako se uvodila podrška za većinu poznatijih platformi. Linux i Solaris podržani su već 2009. godine u verziji 1.2, a od 2012. službeno je podržan Mac

OS X. Java FX se razvijao u smjeru podrške što većeg broja raznih tehnologija za prikaz i unos podataka. Tako on sada podržava CSS, 3D grafiku, MathML, manipulaciju slikama, podršku za rad s dodirnim zaslonima i druge tehnologije.

Jedna je od najvećih prednosti programskog okvira Java FX mnoštvo raznih ulazno/izlaznih kontrola koje nam omogućavaju izradu i najsloženijih programskih sučelja. Sve te kontrole mogu biti vrlo jednostavno stilski oblikovane te na taj način možemo ostvariti vizualni dizajn aplikacije visoke kvalitete. Primjeri nekih ulazno/izlaznih kontrola prikazani su u tablici 14.1.

14.1 Struktura aplikacije izrađene tehnologijom Java FX

Primjer jednostavne aplikacije izrađene tehnologijom Java FX je "Hello world" koju smo vidjeli u poglavlju o anonimnim klasama.
Ovdje ćemo dati još jedan primjer aplikacije koja prikazuje tablicu s imenima i prezimenima osoba.

```
public class MainApp extends Application {
        private TableView<Person> table = new TableView<Person>();
        private final ObservableList<Person>
        data = FXCollections.observableArrayList(
            new Person("Pero", "Perić"),
            new Person("Ana", "Anić"),
            new Person("Marko", "Marković"),
            new Person("Iva", "Ivić")
        );
        public static void main(String[] args) {launch(args);}
@Override
public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(450);
        stage.setHeight(500);
        final Label label = new Label("Popis imena");
        label.setFont(new Font("Arial", 20));
        table.setEditable(true);

        TableColumn firstNameCol = new TableColumn("Ime");
        firstNameCol.setMinWidth(100);
        firstNameCol.setCellValueFactory(
        new PropertyValueFactory<Person, String>("ime"));

        TableColumn lastNameCol = new TableColumn("Prezime");
        lastNameCol.setMinWidth(100);
        lastNameCol.setCellValueFactory(
        new PropertyValueFactory<Person, String>("prezime"));

        table.setItems(data);
        table.getColumns().addAll(firstNameCol, lastNameCol);

        final VBox vbox = new VBox();
        vbox.setSpacing(5);
        vbox.setPadding(new Insets(10, 0, 0, 10));
        vbox.getChildren().addAll(label, table);
         ((Group) scene.getRoot()).getChildren().addAll(vbox);
        stage.setScene(scene);
        stage.show();
}
public static class Person {
        private final SimpleStringProperty ime, prezime;
        private Person(String fName, String lName) {
```

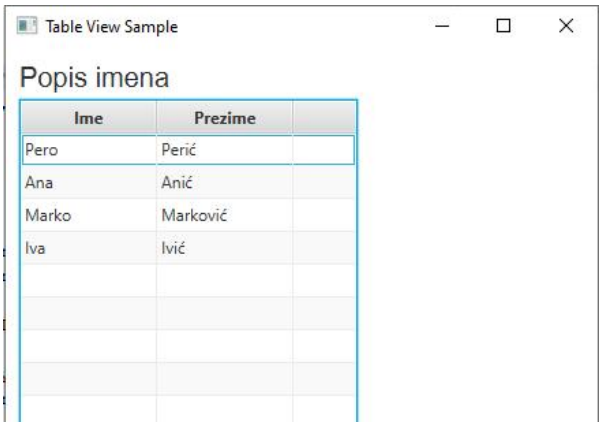
```
                ime = new SimpleStringProperty(fName);
                prezime = new SimpleStringProperty(lName);
        }
        public String getIme() { return ime.get(); }
        public void setIme(String Name) { ime.set(Name); }
        public String getPrezime() { return prezime.get(); }
        public void setPrezime(String Name) { prezime.set(Name); }
        }
}
```

Primjer kôda 14.1 Jednostavna aplikacija koja tablično prikazuje podatke o osobama napravljena korištenjem tehnologije Java FX.

Najvažnije su mogućnosti koje iz ovoga primjera možemo vidjeti sljedeće:

- ovakve aplikacije nasljeđuju klasu `Application` koju uključujemo odgovarajućom naredbom `import`;
- glavna funkcija samo poziva metodu `launch()` za pokretanje aplikacije;
- klasa `Stage` predstavlja svojevrsnu 'pozornicu' koja predstavlja podlogu za prikaz sadržaja naše aplikacije;
- klasa `Scene` isto je tako obavezna klasa i nju koristimo kako bismo definirali 'scenu' na pozornici koju možemo po potrebi i mijenjati;
- kontrole i razmještaji koje dodajemo na scenu (`Label`, `TableView`, `VBox`) konstruirani su kao objekti;
- postoje razne metode kojima definiramo stilski izgled pozornice i drugih kontrola (`setWidth()`, `setFont()`, ...).

U ovom je primjeru također korištena statički ugniježđena klasa `Person` koja nam je omogućila stvaranje objekata čije vrijednosti prikazujemo u tablici. Izgled aplikacije na Windows 10 prikazan je idućom slikom.



Slika 14.1: Izgled Java FX aplikacije na Windows 10 operacijskom sustavu.

Važno je napomenuti da, osim 'vidljivih' kontrola, poput `Label` ili `TableView`, u Java FX-u se koriste i razmještaji (engl. *layouts*) koji nisu vidljivi, ali omogućavaju grupiranje i pravilan raspored kontrola na samoj sceni. U ovom je primjeru korišten vertikalni kontejner `VBox` koji slaže elemente vertikalno. Zato je `Label "Popis imena"` iznad same tablice. Osim ovog kontejnera postoje još razni drugi kontejneri u Java FX-u. Neki od njih su su:

- `HBox` (horizontalni raspored elemenata);
- `BorderPane` (raspored sa zaglavljem podnožjem, lijevim i desnim dijelom te glavnim centralnim dijelom);
- `GridPane` (matrični raspored elemenata) i drugi kontejneri.

14.2 Naprednije mogućnosti tehnologije Java FX

Ozbiljniji rad s aplikacijama koje koriste tehnologiju Java FX obavlja se tako da se cijela scena kreira u odvojenoj datoteci vrste `.fxml`. To je tipična XML datoteka u kojoj su zapisani podaci o objektima kontrola i razmještaja u obliku XML-elemenata i atributa, a koji se onda mogu direktno ugraditi u aplikaciju koja koristi ovu tehnologiju.
Na taj način olakšavamo samu izradu ulazno izlaznog sučelja aplikacije, za razliku od 'ručnog' kreiranja kako smo to vidjeli u prethodnoj cjelini.
Kako bismo mogli povezati takav dokument vrste `.fxml` s našom aplikacijom, važno je navesti naziv upravljača (kontrolera), tj. upravljačke klase koja će biti zadužena za rad tog dijela aplikacije čiju smo scenu definirali.
Osim što je moguće ručno izmijeniti sam dokument vrste `.fxml` zato što je on kao i svaki XML-dokument tekstualna datoteka, postoji još jednostavniji način, a to je upotreba programa *SceneBuilder*. Taj nam program omogućava još jednostavniji način kreiranja samog korisničkog sučelja, a rezultat se uvijek sprema u odgovarajuću datoteku tipa `.fxml`.
Programski kôd pomoću kojeg možemo datoteku tipa `.fxml` ugraditi u našu aplikaciju je sljedeći:

```
Parent root = FXMLLoader.load(getClass().getResource("/fxml/Scene.fxml"));
Scene scene = new Scene(root);
scene.getStylesheets().add("/styles/Styles.css");
```

Prva naredba služi za učitavanje sadržaja iz konkretne datoteke `Scene.fxml` koja se nalazi na navedenoj putanji u projektu. Nakon toga se taj sadržaj pridružuje novoj sceni, a posljednja naredba nam omogućava da toj sceni pridružimo i odgovarajuću stilizaciju koja je definirana klasičnom CSS datotekom.

Primjer je jednostavne datoteke `Scene.fxml` sljedeći:

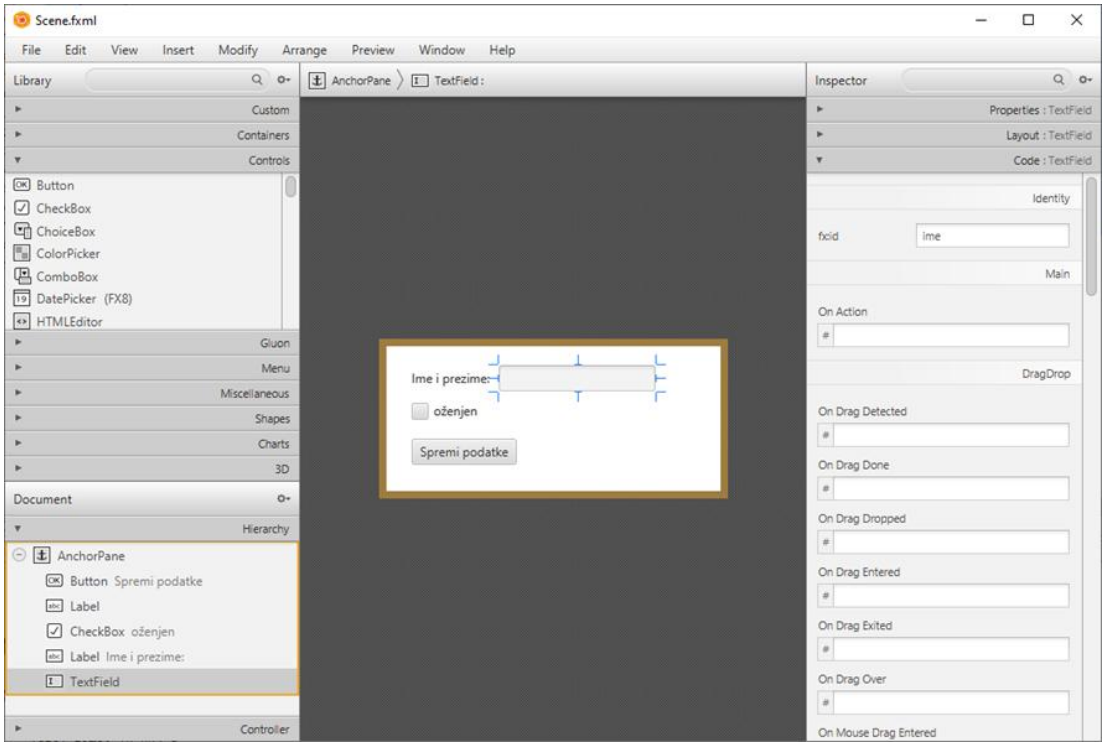
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane id="AnchorPane" prefHeight="138.0" prefWidth="318.0"
            xmlns="http://javafx.com/javafx/8.0.171"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="com.mycompany.testapptable.FXMLController">
    <children>
        <Button fx:id="spremi" layoutX="24.0" layoutY="88.0"
            onAction="#handleButtonAction" text="Spremi podatke" />
        <Label fx:id="label" layoutX="126" layoutY="120"
            minHeight="16" minWidth="69" />
        <CheckBox fx:id="ozenjen" layoutX="24.0" layoutY="53.0"
            mnemonicParsing="false" text="oženjen" />
        <Label layoutX="24.0" layoutY="22.0" text="Ime i prezime:" />
        <TextField fx:id="ime" layoutX="107.0" layoutY="18.0" />
   </children>
</AnchorPane>
```

Primjer kôda 14.2 Jednostavna datoteka vrste `.fxml` koja opisuje scenu.

Prikaz ove datoteke u programu SceneBuilder prikazan je slikom 14.2.



Slika 14.2: Program SceneBuilder za izradu scene za aplikacije koje koriste Java FX tehnologiju.

U izvornom kôdu datoteke vrste `.fxml` posebno je označen naziv upravljačke klase za prikazanu scenu. Ova upravljačka klasa mora biti smještena u projektu točno na navedenoj lokaciji i naziv same klase mora biti odgovarajući. Vrlo često ovaj naziv trebamo ručno provjeriti uređivanjem .fxml datoteke, bez pokretanja SceneBuilder programa. Primjer kôda upravljačke klase dan je sljedećim kôdom:

```
public class FXMLController implements Initializable {
        @FXML
        private TextField ime;
        @FXML
        private CheckBox ozenjen;
        @FXML
        private Button spremi;
        @FXML
        private void handleButtonAction(ActionEvent event) {
                String status = " nije ";
                if (ozenjen.isSelected()) status = " je ";
                System.out.println( ime.getText() + status + "oženjen." );
        }
        @Override
        public void initialize(URL url, ResourceBundle rb) {
        // TODO
        }
}
```

Primjer kôda 14.3 Upravljačka klasa za jednostavnu scenu definiranu datotekom vrste `.fxml`.

Najvažniji su detalji koje trebamo primijetiti u ovoj upravljačkoj klasi sljedeći:

- svaka upravljačka klasa implementira sučelje `Initializable`;
- najvažnije je postaviti imena kontrola kojima će se upravljati te se svaki takav objekt označava posebnom anotacijom `@FXML`;
- imena objekata kontrola moraju odgovarati onome što je zadano u datoteci vrste `.fxml` (primijetite naziv `ime` koji se nalazi u izvornom kôdu elementa `TextField` XML-datoteke; isti taj prikazan je i

u slici programa SceneBuilder te je pokazano gdje se to zadaje, a naposljetku, u samom upravljaču imamo taj isti naziv posebno istaknut);

- postoje razne metode za postavljanje i dohvaćanje vrijednosti samih kontrola poput `setText`, `getText`, `isSelected` i mnoge druge;
- u važnoj metodi `initialize` programski postavljamo sve početne vrijednosti u kontrolu te vrlo često koristimo jednostavnije lambda-izraze za razne funkcionalnosti pojedinih elemenata na sceni.

14.3 Zadaci

Zadatak 14.1. Preuredite navedeni program tako da osim imena i prezimena prikazuje još i e-adresu i broj telefona u tabličnom prikazu. Tablica će onda imati četiri stupca.

Zadatak 14.2. Napravite aplikaciju koja koristi tehnologiju Java FX koja će omogućiti unos podataka o osobi: ime, prezime, e-adresa i broj telefona. Za unos koristiti klasu `TextField`, a isto tako i dodati gumb koji će omogućiti spremanje unosa. Samo spremanje nije potrebno napraviti.

Zadatak 14.3. Pokušajte riješiti prethodni zadatak tako da sada umjesto ručno, pomoću programa SceneBuilder, napravite traženu scenu. Dodatno napravite i upravljač koji će sve unesene podatke ispisati u terminal pomoću metode `System.out.println` kada se pritisne odgovarajući gumb. Pokušajte lijepo rasporediti podatke za unos tako da oznake i kućice za unos `TextFiled` rasporedite pravilno koristeći kontejner `GridPane`.

Zadatak 14.4. Napravite aplikaciju koja koristi tehnologiju Java FX koja će omogućiti unos korisničkog imena i zaporke (zaporka se ne smije vidjeti prilikom unosa). Nakon toga provjerite je li korisnik `"Pero"` i je li lozinka `"123"`. U slučaju odgovarajućih podataka završite program, a u suprotnom, neka program ostane raditi.

Zadatak 14.5. Napravite aplikaciju koja koristi tehnologiju Java FX 'mini kalkulator' koja će imati mogućnost unosa dvaju brojeva u dvije kućice tipa `TextFiled` te četiri dugmića za gumba osnovne računske operacije. Pritiskom na bilo koju od tih operacija potrebno je u odgovarajućoj kontroli tipa `Label` ispisati rezultat izvršene operacije.
Savjet: u metodi `initialize` definirajte za svaki gumb računske operacije njegovo ponašanje.

Popis slika

Popis tablica