PRIRUČNIK ZA LABORATORIJSKE VJEŽBE IZ KOLEGIJA "PROGRAMIRANJE U JAVI"

Pred. Aleksander Radovan, dipl. ing.

Sadržaj

1. UVOD

Ovaj priručnik je prvenstveno namijenjen studentima Specijalističkog studija Veleučilišta u Velikog Gorici koji su upisali kolegij "Programiranje u programskom jeziku Java". U njemu se opisuje programski jezik Java kao jedan od najvažnijih programskih jezika današnjice i koji ima vrlo široku primjenu u svijetu.

Od korisnika ovog priručnika očekuje se da posjeduju solidno znanje programiranja u programskom jeziku C, te osnove objektno-orijentiranog programiranja.

U poglavljima priručnika opisuju se osnovna svojstva programskog jezika Java, razvojno okruženje Eclipse za pisanje programa u Javi i dodaci koji općenito olakšavaju programiranje u Javi.

Drugo poglavlje opisuje što je sve potrebno napraviti kako bi se uspostavila razvojna okolina za pisanje Java aplikacija, instalirali dodaci za lakše programiranje, te način korištenja nekih opcija koje su izuzetno korisne i ubrzavaju sam razvoj Java aplikacija u Eclipse razvojnoj okolini.

Treće poglavlje sadrži osnovne podatke o tipovima podataka koji se koriste u Javi sa ciljem usporedbe Jave s programskim jezikom C, operatorima koji se mogu koristiti prilikom pisanja programa i ostale osnovne značajke ključnih riječi koje je potrebno koristiti i kod najjednostavnijih Java programa.

Četvrto poglavlje opisuje osnovne principe klasa i objekata, te konstruiranje programa podijeljenih na više klasa iz kojih se kreiraju objekti pomoću posebnih metoda koje se nazivaju konstruktori. Svi elementi unutar klase (varijable i metode) označavaju se modifikatorima koji im daju posebna svojstva koja definiraju ponašanje, vidljivost te način pozivanja i dohvaćanja tih elemenata.

Peto poglavlje odnosi se na objektno-orijentirano programiranje u Javi koje se temelji na nasljeđivanje klasa, implementiranju sučelja, nadjačavanju metoda itd. Objektno-orijentirani principi vrlo su važan temelj za savladavanje naprednijih principa programiranja u Javi koje se temelji na korištenju tzv. programskih okvira (engl. *framework*). Programski okviri omogućavaju programerima olakšano kreiranje složenijih sustava, bez potrebe da se svi detalji i podsustavi nanovo razvijaju, već da se koriste gotove komponente, čime se značajno ubrzava razvoj Java aplikacija.

U šestom poglavlju opisuju se iznimke kao vrlo značajan mehanizam u programskom jeziku Java za izgradnju robusnih aplikacija. Iznimke najčešće označavaju neku vrstu pogreške ili iznimne situacije u Java aplikaciji. Zbog toga je jako bitno predvidjeti sve ključne situacije unutar aplikacije i definirati primjerene radnje koje je potrebno obaviti kako bi aplikacija mogla nastaviti s radom.
Sedmo poglavlje opisuje zbirke (engl. *Collections*) kao vrlo važnu cjelinu programskog jezika Java, pomoću kojih se mogu organizirati različiti podaci koji se prikazuju na ekranu ili se dohvaćaju iz baze podataka. Zbirke su podijeljene na tri osnove skupine, liste, mape i setovi, a svaka od njih obilježena je svojstvenim karakteristikama koje omogućavati

organiziranje svih vrsta podataka koji se pojavljuju u praktičnoj primjeni.

U osmom poglavlju prikazana je mogućnost korištenja datoteka u programskom jeziku Java, pri čemu datoteke mogu sadržavati tekstualni ili binarni sadržaj. Osim slijednog čitanja sadržaja datoteka, može se iskoristiti posebni način za pristupanje sadržaju slučajnim rasporedom. Osim toga Java omogućava tzv. serijalizaciju objekata kojom je moguće objekte unutar programa spremiti u datoteku, a kasnije procesom deserijalizacije vratiti u program kod naknadnog pokretanja programa.

Deveto poglavlje se odnosi organiziranje elemenata za oblikovanje grafičkog sučelja Swing, kao važnog dijela same Jave u inačici *Standard Edition*. O tom poglavlju opisuju se najvažnije komponente za izradu grafičkog korisničkog sučelja, te načini organiziranja istih pomoću organizatora rasporeda komponenti (engl. *Layout Managers*). Postoje dva načina dizajniranja izgleda grafičkog sučelja: izravno pisanje programskog koda ili korištenjem pomoćne komponente koja omogućava kreiranje korisničkog sučelja grafičkim putem (bez pisanja koda). Svaka metoda ima svoje prednosti i mane, te su obje opisane unutar devetog poglavlja.

Deseto poglavlje opisuje način povezivanja Java aplikacija s bazom podataka, konkretno s *Apache Derby* bazom. Svaka aplikacija koja mora spremati podatke u bazu podataka kako bi bili sačuvani i nakon prekida rada s aplikacijom. Način spajanja, konfiguriranja veze s bazom podataka, te izvršavanja upita nad bazom podataka opisani su upravo u ovom poglavlju.

2. RAZVOJNO OKRUŽENJE ECLIPSE

U ovom poglavlju opisuje se razvojno okruženje Eclipse, koraci potrebni za instalaciju, njene najvažnije funkcionalnosti za razvijanje Java aplikacija, te nadogradnja razvojnog okruženja raznim dodacima koji olakšavaju razvijanje Java aplikacija.

2.1. Instalacija razvojnog okruženja Eclipse za Java aplikacije

Eclipse predstavlja platformu za razvijanje aplikacija u Javi i drugim jezicima, te zbog mogućnosti nadograđivanja pomoću dodataka (engl. *plugins*) može se prilagoditi velikom broju potreba i zahtjeva kod razvijanja aplikacija. Više informacija o samom razvojnom okruženju (engl. *Integrated Development Environment - IDE*) moguće je pronaći na stranicama http://www.eclipse.org/.

2.1.1. Instalacija Java alata za razvijanje

Prije preuzimanja instalacije Eclipse razvojnog okruženja potrebno je preuzeti i instalirati razvojne biblioteke za Javu (engl. *Java Development Kit*) sa stranica http://www.oracle.com/technetwork/java/javase/downloads/index.html. Od ponuđenih resursa za preuzimanje potrebno je odabrati "*Java Platform (JDK)*". Nakon preuzimanja potrebno je pokrenuti izvršnu datoteku i izvršiti instalacijski postupak prema zadanim koracima (potrebno je instalirati *Java Development Kit* i *Java Runtime Environment* koji omogućavaju razvijanje Java programa i pokretanje Java programa, respektivno). Nakon završetka instalacije potrebno je provjeriti je li pokretanje Java virtualnog stroja (engl. *Java Virtual Machine*) moguće obaviti upisivanjem naredbe u naredbenu liniju (engl. *Command prompt*). To je moguće isprobati npr. upisivanjem naredbe "java -version" u naredbenoj liniji:

```
C:\Users\Aleksander>java -version
```

Ukoliko se nakon unošenja naredbe ispiše odgovor poput sljedećeg (razlika može biti u broju inačice Jave), znači da je instalacija prošla u redu:

```
java version "1.7.0_03"
Java(TM) SE Runtime Environment (build 1.7.0_03-b05)
Java HotSpot(TM) Client VM (build 22.1-b02, mixed mode, sharing)
```

Ako to nije slučaj i ispiše se poruka o pogrešci, potrebno dodati mapu "bin" (koja ima putanju "C:\Program Files\Java\jdk1.7.0_03\bin" ili onu koja je zadana kod instalacije) u varijablu okruženja (engl. *Environment variable*) pod imenom PATH. U operacijskom sustavu "Windows" varijable okruženja moguće je mijenjati odabirom opcije "Control Panel->System->Advanced system setting->Environment variables". Ako varijabla PATH nije kreirana, potrebno je dodati novu varijablu i postaviti putanju "C:\Program Files\Java\jdk1.7.0_03\bin;%path%" ili proširiti postojeću.

2.1.2. Instalacija razvojnog okruženja Eclipse

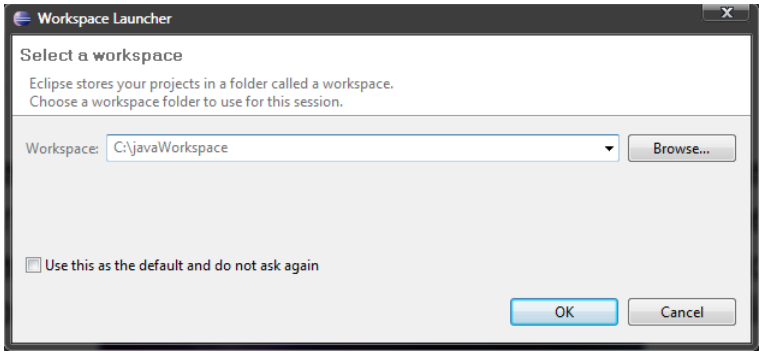
Eclipse razvojno okruženje moguće je besplatno preuzeti sa stranica http://www.eclipse.org/downloads/, pri čemu je potrebno preuzeti inačicu pod nazivom "*Eclipse IDE for Java Developers*".

Nakon preuzimanja arhive istu je potrebno raspakirati na željenu lokaciju (podmapa s nazivom "eclipse" kreira se automatski pa je nije potrebno posebno kreirati). Samim raspakiravanjem preuzete arhive instalacijska procedura završava i razvojno okruženje moguće je pokrenuti izvršavanjem datoteke pod nazivom "eclipse.exe" koja se nalazi unutar mape "eclipse".

2.2. Korištenje razvojnog okruženja Eclipse

2.2.1. Otvaranje radnog prostora

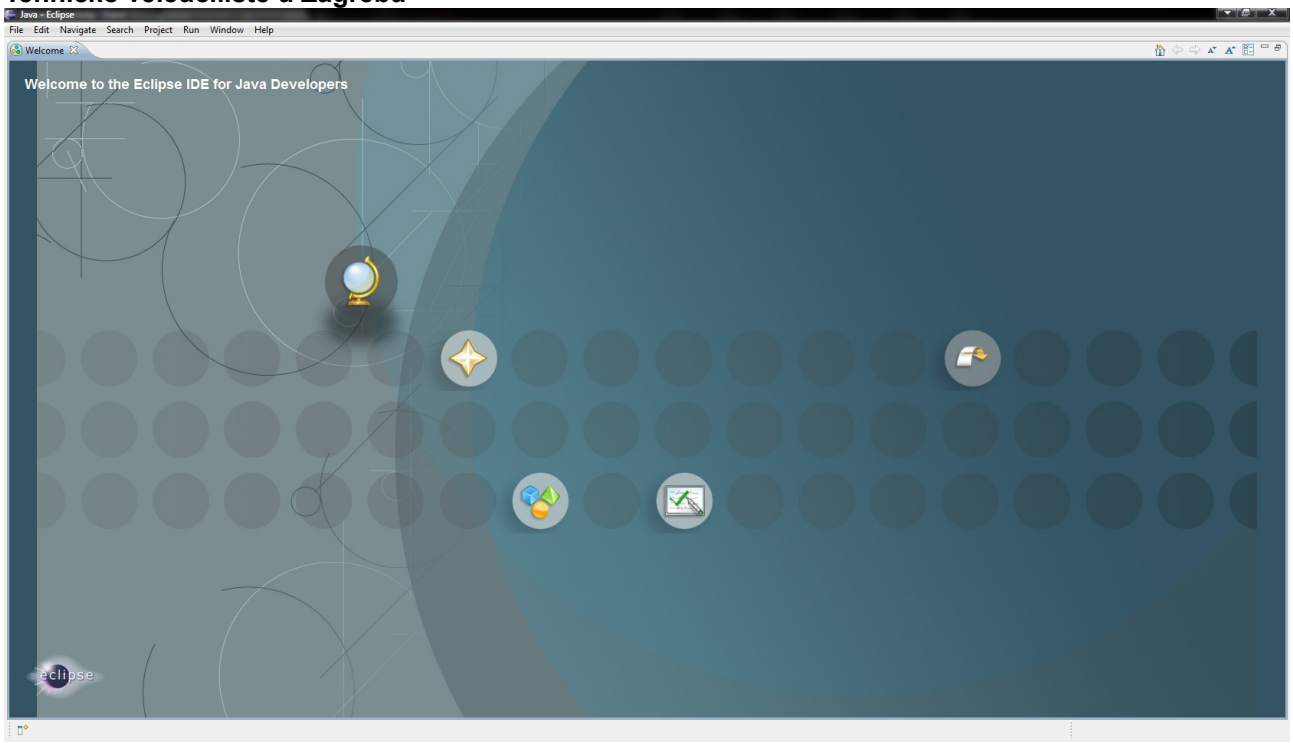
Nakon pokretanja razvojnog okruženja Eclipse pojavljuje se ekran prozor s vizualnim identitetom samog razvojnog okruženja i nazivom inačice (posljednja inačica Eclipse 3.7 nosi kodni naziv "Indigo"). Ubrzo nakon toga pojavljuje se prozor koji omogućava odabir lokacije za radni prostor (engl. *workspace*) koji je prikazan na slici 2.1.



Slika 2.1. Prozor za odabir radnog prostora

Radni prostor je moguće promijeniti i unutar Eclipse-a, bez potrebe gašenja i ponovnog pokretanja, odabirom opcije "Switch Workspace" iz izbornika "File".

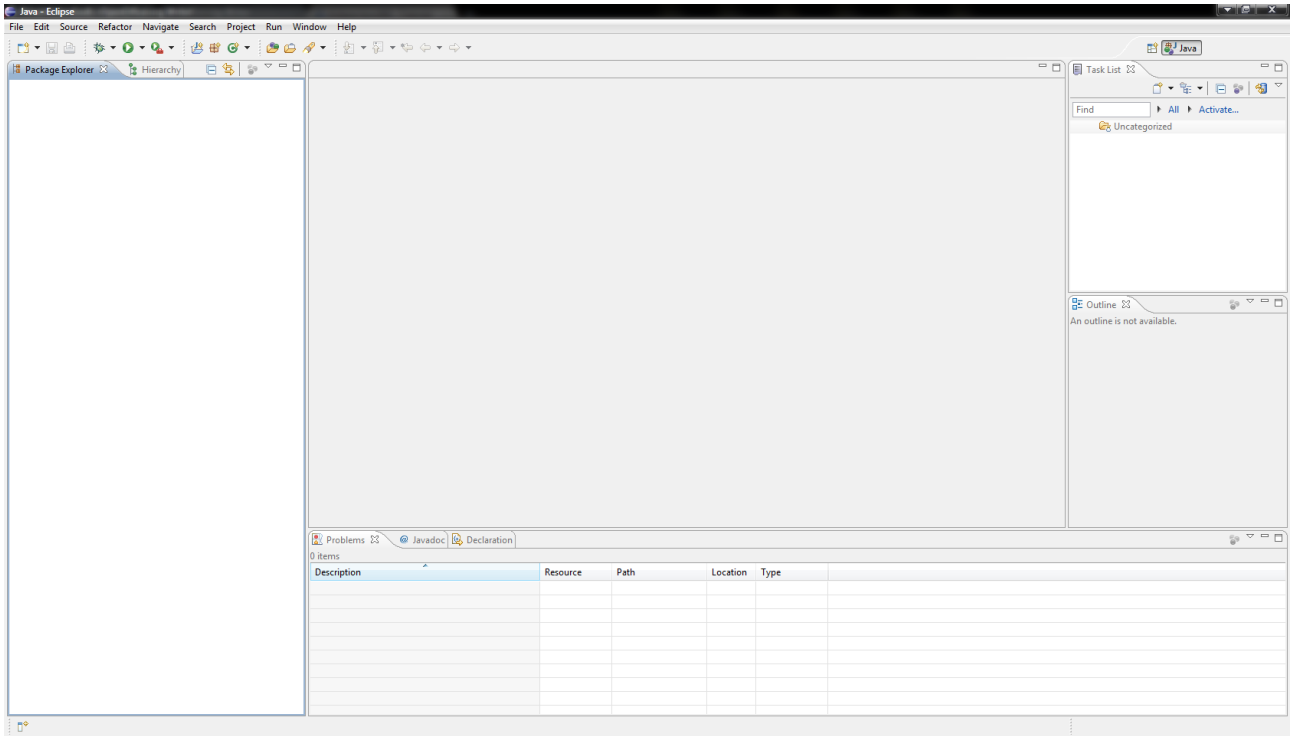
Odabirom radnog prostora i pritiskom na gumb "OK" prikazuje se razvojno okruženje Eclipse. Prilikom prvog pokretanja Eclipse-a prikazuje se pozdravni ekran koji nudi osnovne informacije o navedenoj inačici i pomoć kod korištenja Eclipse-a (slika 2.2.).



Pozdravni prozor je moguće aktivirati i nakon prvog korištenja Eclipse-a, odabirom opcije "Welcome" iz izbornika "Help".

Slika 2.2. Pozdravni ekran s dodatnim informacijama o Eclipse-u

Nakon gašenja "Welcome" prozora prikazuje se Java perspektiva Eclipse-a koja je optimizirana za razvijanje Java aplikacija (slika 2.3.). Razvojno okruženje Eclipse je organizirano po perspektivama, a perspektive se sastoje od pogleda (engl. *View*) i *editora*. Svaki pogled i *editor* je moguće dodati, maknuti ili premjestiti.



Slika 2.3. Java perspektiva

Ukoliko je potrebno vratiti početni izgled perspektive, potrebno je odabrati opciju

"Window->Reset Perspective".

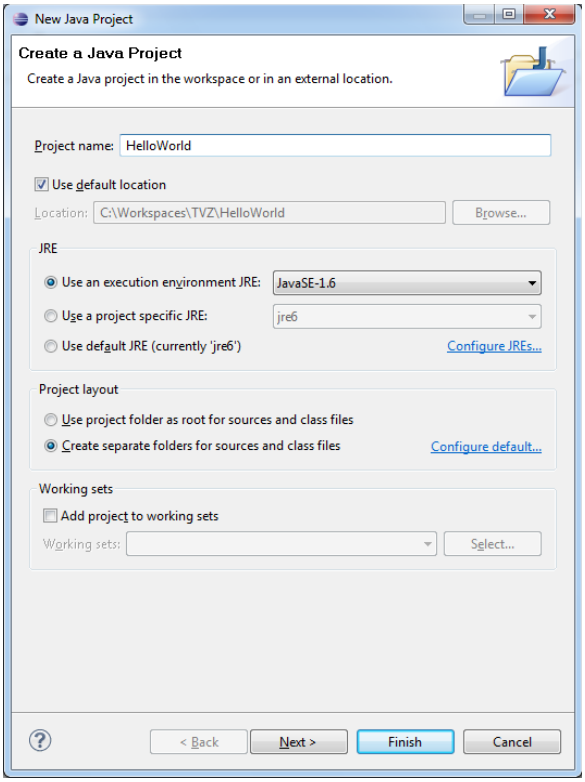
Osim prikazane Java perspektive, kod razvijanja Java aplikacije često se koristi i "Debug" perspektiva koja će detaljnije biti objašnjena u nastavku. Debug perspektiva je optimizirana za praćenje rada i pronalaženje pogrešaka unutar Java aplikacija.

2.2.2. Kreiranje prve Java aplikacije

Osnovni dijelovi svakog objektno-orijentiranog jezika su klase. Za kreiranje najjednostavnije moguće Java aplikacije koja samo ispisuje tekst na ekran (unutar Eclipse-a se nalazi posebni pogled za tu svrhu – *Console View*), potrebno je kreirati klasu koja će sadržavati programski kod aplikacije.

Unutar jednog radnog prostora Eclipse-a moguće je imati više aplikacija koje su organizirane po projektima. Ima više vrsta projekata, a za Java aplikacije postoji zasebna vrsta projekta pod nazivom "*Java Project*".

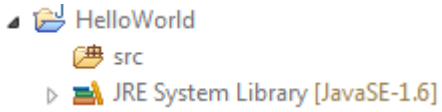
Prvi korak u izradi najjednostavnije "Hello World" aplikacije je kreiranje Java projekta unutar Eclipse-a. Projekte je moguće kreirati unutar "*Package explorer*" pogleda unutar lijevog dijela Java perspektive (slika 2.3.) odabirom opcije "File->New->Java Project". Na slici 2.4. prikazan je prozor za kreiranje novog Java projekta.



Slika 2.4. Prozor za kreiranje Java projekta

Prozor osim obvezatnog unošenja naziva projekta ("HelloWorld") nudi razne druge opcije kod kreiranja novog projekta kao što je sadržaj projekta, inačica Jave koja će se koristiti za pokretanje projekta (JRE), izgled projekta i kreiranje radnih skupina (engl. *Working sets*). Za kreiranje Java projekta koji će sadržavati samo najjednostavniju Java aplikaciju dovoljno je unijeti naziv projekta "Hello World" i pritisnuti gumb "Finish", nakon čega se

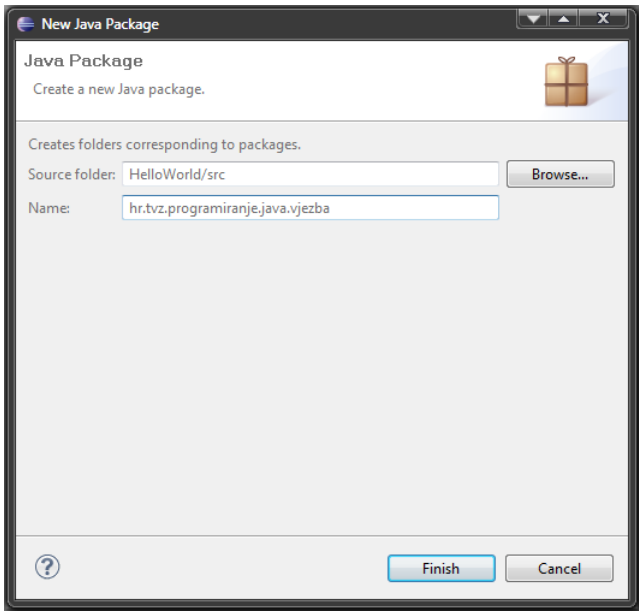
unutar "Package Explorer"-a pojavljuje navedeni projekt (slika 2.5.).



Slika 2.5. Java projekt

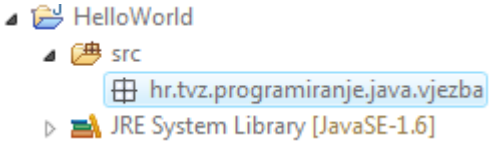
Prazan Java projekt sastoji se od mape za izvorni kod (engl. *Source folder*) označen sa "src" te oznaku JRE-a koji se koristi za pokretanje aplikacije.

Nakon kreiranja Java projekta, drugi korak je kreiranje paketa u kojem će se nalazi klasa s programskim kodom koji ispisuje poruku "Hello World!". Paket je logička cjelina programa koja objedinjava klase i ostale resurse koje povezuje zajednička funkcionalnost. Svaka klasa u programskom jeziku Java mora se nalaziti u svom paketu kako bi se, između ostalog, razlikovala od istoimenih klasa nekih drugih projekata ili aplikacija. Nazivi paketa u sebi obično nose i puno drugih informacija, npr. naziv ustanove ili organizacije koja je autor dotičnih klasa. Tako bi se klase kreirane za vježbu u sklopu kolegija "Programiranje u Javi" na Tehničkom veleučilištu smjestile u paket pod nazivom "hr.tvz.programiranje.java.vjezba". Naziv paketa obično se sastoji od više riječi ili kratica odvojenih točkama.



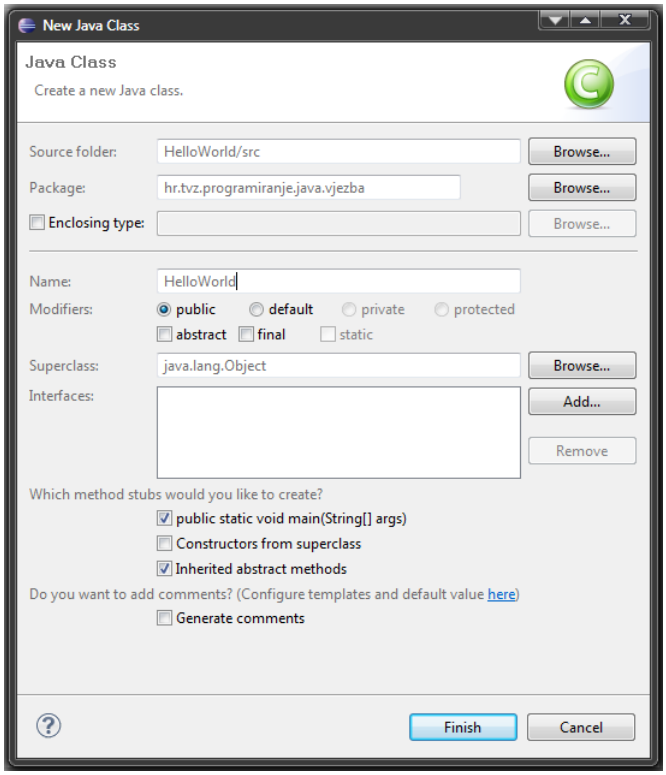
Slika 2.6. Prozor za kreiranje novog paketa

Paketi se kreiraju odabirom opcije "File->New->Package", nakon čega se pojavljuje prozor kao na slici 2.6.
Prvo je potrebno odabrati mapu za izvorni kod u koju će se paket dodati (ako već nije ponuđena), a nakon toga i naziv paketa. Ukoliko se pokuša kreirati paket čije ime nije u skladu s pravilima (na primjer, kad završava s točkom), na prozoru se prikazuje poruka o pogrešci i paket nije moguće kreirati. Ukoliko se navede ispravan naziv i pritisne gumb "Finish", unutar mape izvornog koda "src" pojavljuje se prazan paket kao na slici 2.7.



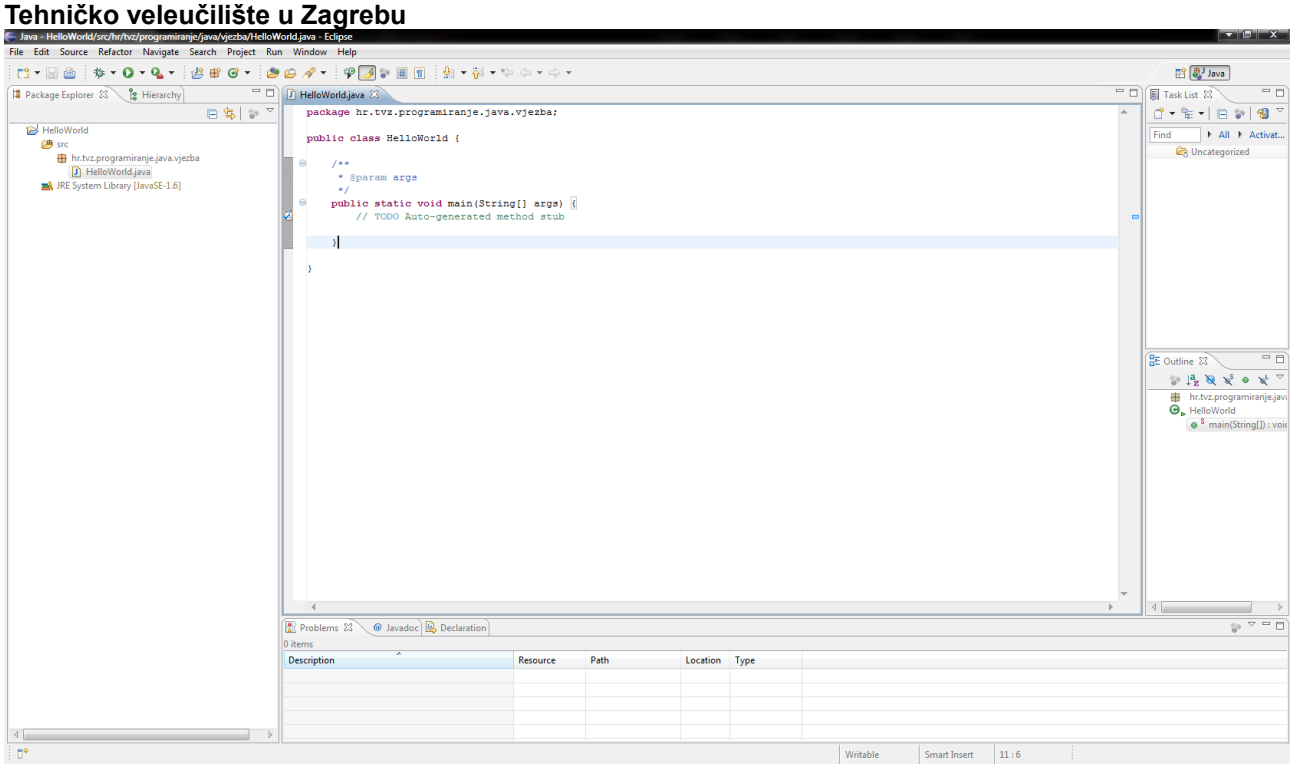
Slika 2.7. Java projekt s praznim paketom

Treći korak nakon kreiranja paketa je kreiranje Java klase koja će sadržavati programski kod koji ispisuje poruku "Hello World!". Klasu je moguće kreirati odabirom opcije "File->New->Class" nakon čega se prikazuje prozor kao na slici 2.8.



Slika 2.8. Prozor za kreiranje nove klase

Prozor sadrži puno svojstava klase koja se kreira, kao što su modifikatori i nadklase, međutim, radi jednostavnosti je potrebno upisati samo ime klase "HelloWorld" i označiti *checkbox* kod metode "public static void main(String[] args)" koja omogućava klase da se izvrši, što je i potrebno kako bi se ispisao željeni tekst. Ostala svojstva klase objašnjena su u narednim poglavljima.



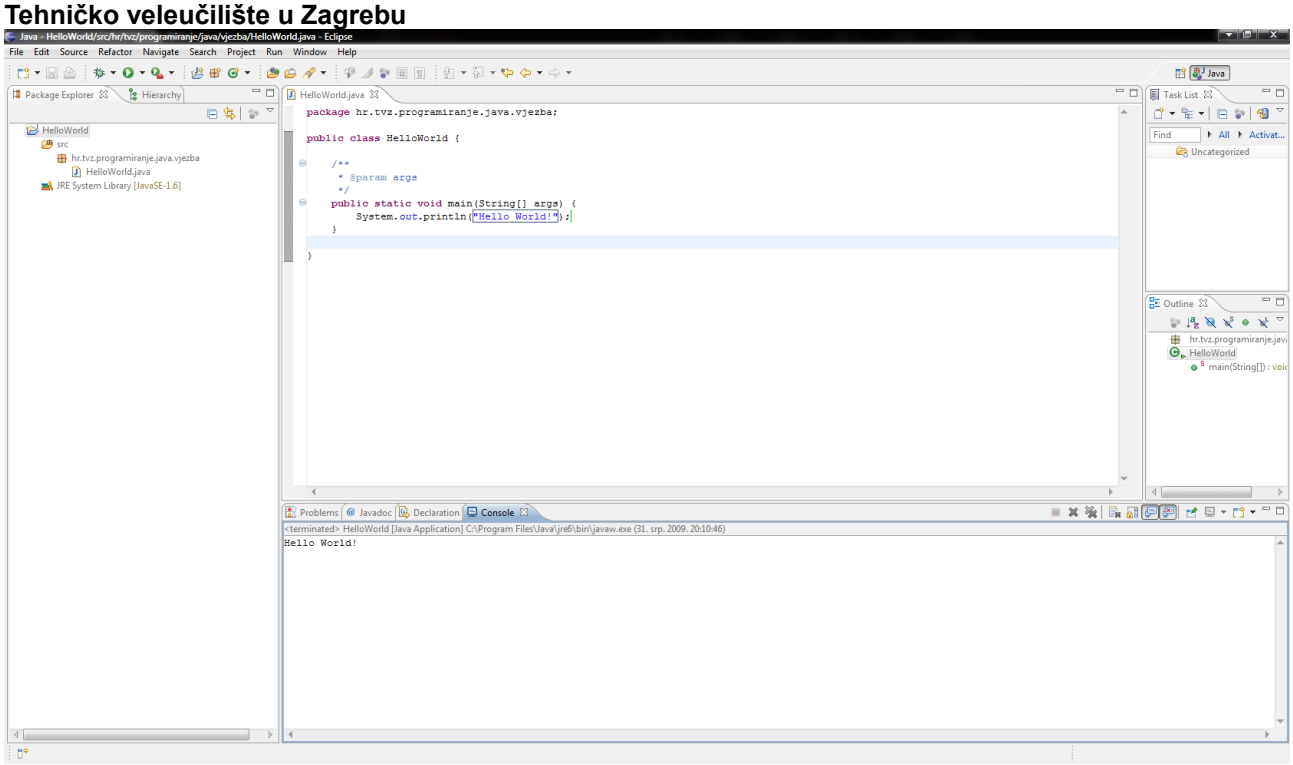
Slika 2.9. Izgled razvojnog okruženja Eclipse nakon kreiranja klase "HelloWorld"

Slično kao i kod imenovanja paketa, klase također moraju zadovoljavati određene kriterije po pitanju imenovanja pa se na gornjem dijelu prozora pojavljuju određene poruke ukoliko se ti kriteriji ne poštuju (na primjer, Java klase moraju započeti velikim slovom, u suprotnom se pojavljuje upozorenje). Ako se navede ispravan naziv klase ("HelloWorld") i pritisne tipka "Finish", kreira se prazna klasa unutar paketa "hr.tvz.programiranje.java.vjezba", kao što je i prikazano na slici 2.9.

Unutar *Package explorer*-a je dodana klasa, a na sredini razvojnog okruženja Eclipse se pojavio sadržaj klase "HelloWorld". Svaka klasa u Javi mora na samom početku imati naznačeno ime paketa u kojem se nalazi, modifikatore i naziv klase (koji će biti detaljnije opisani u nastavku), te tijelo klase u koje se upisuje programski kod koji klasa izvršava. Klasa "HelloWorld" posjeduje i *main* metodu koja klasi daje svojstvo izvršavanja. Iznad metode nalazi se automatski generirana *javadoc* dokumentacija koja je detaljnije opisana u nastavku.

Naredbu koji ispisuje poruku "Hello World!" potrebno je ubaciti u tijelo *main* metode između vitičastih zagrada, umjesto komentara "//TODO Auto-generated method stub". Naredba za ispis poruke "Hello World!" izgleda ovako:

```
System.out.println("Hello World!");
```



Nakon ubacivanja navedene naredbe u *main* metodu, sve je spremno za pokretanje programa koji će u konzolu ispisati pozdravnu poruku. Pokretanje programa moguće je odabirom opcije "Run->Run As->Java Application".

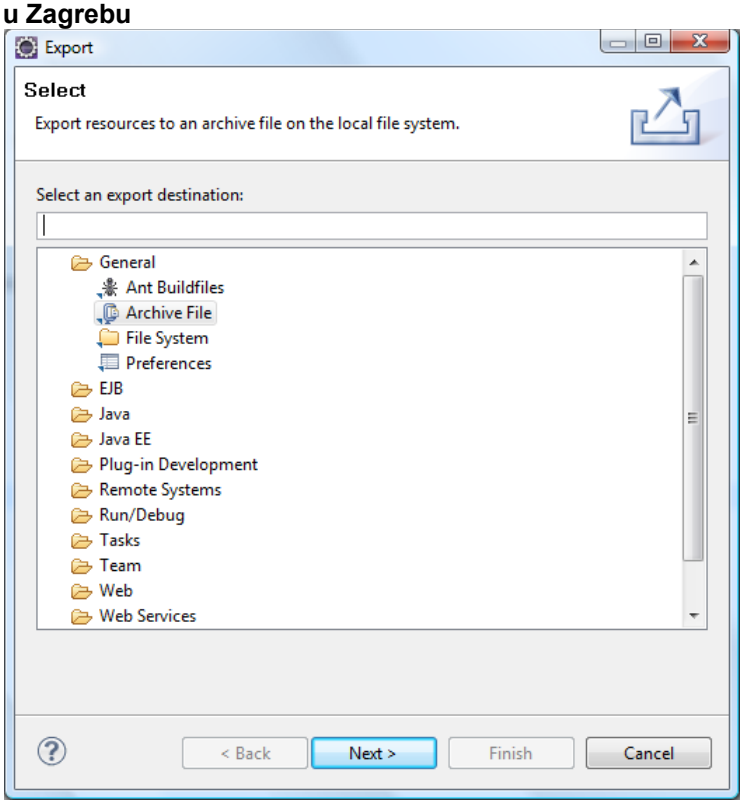
Slika 2.10. Izgled razvojnog okruženja Eclipse nakon izvršenja programa HelloWorld

Nakon uspješnog pokretanja u donjem dijelu razvojnog okruženja Eclipse, točnije *Console* pogledu, pojavljuje se poruka koju ispisuje program (slika 2.10). Konzola služi za ispis poruka i rezultata programa, te služi kao standardni izlaz Java programa.

2.3. Izvoz i uvoz postojećih projekata u korisničko radno okruženje

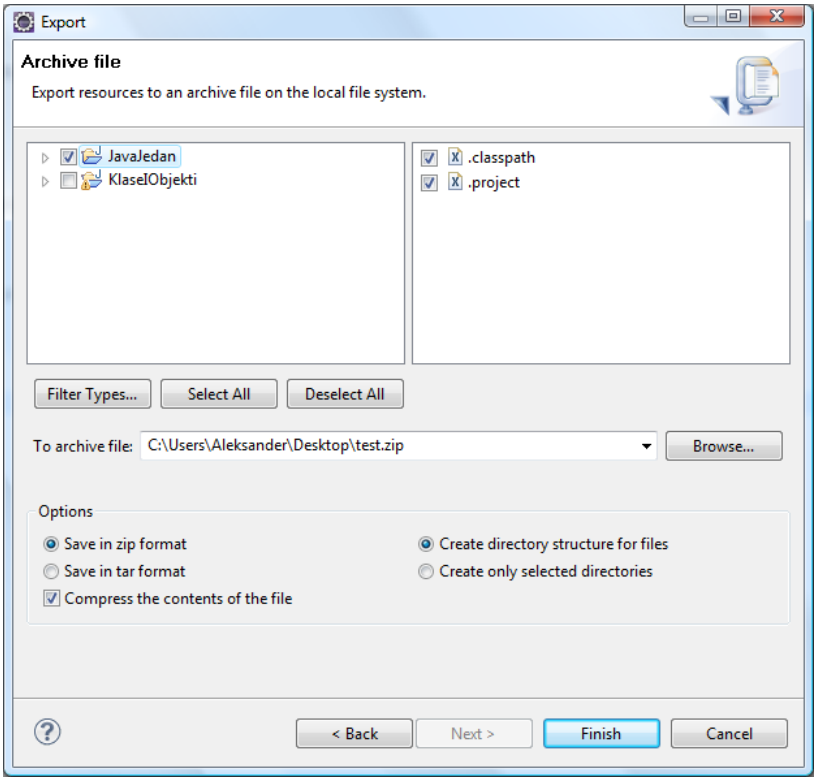
Često je potrebno raditi s više radnih okruženja istovremeno i na različitim lokacijama pa je bitno da se projekti i izvorni kod mogu jednostavno prenositi i koristiti. Razvojno okruženje Eclipse nudi mogućnost izvoza (engl. *export*) projekata u arhivirane datoteke (npr. na računalu kod kuće) i uvoza (engl. *import*) tih arhiviranih datoteka u neko drugo radno okruženje (npr. na računalu u laboratoriju fakulteta).

Za izvoz projekta u Eclipse radnom okruženju potrebno je odabrati opciju File->Export nakon čega će se pojaviti prozor kao na slici 2.11.



Slika 2.11. Prozor za izvoz arhivirane datoteke Eclipse projekta

Na prozoru je potrebno odabrati ikonu "Archive File" unutar mape "General" i pritisnuti gumb "Next" nakon čega će se prikazati prozor kao na slici 2.12.

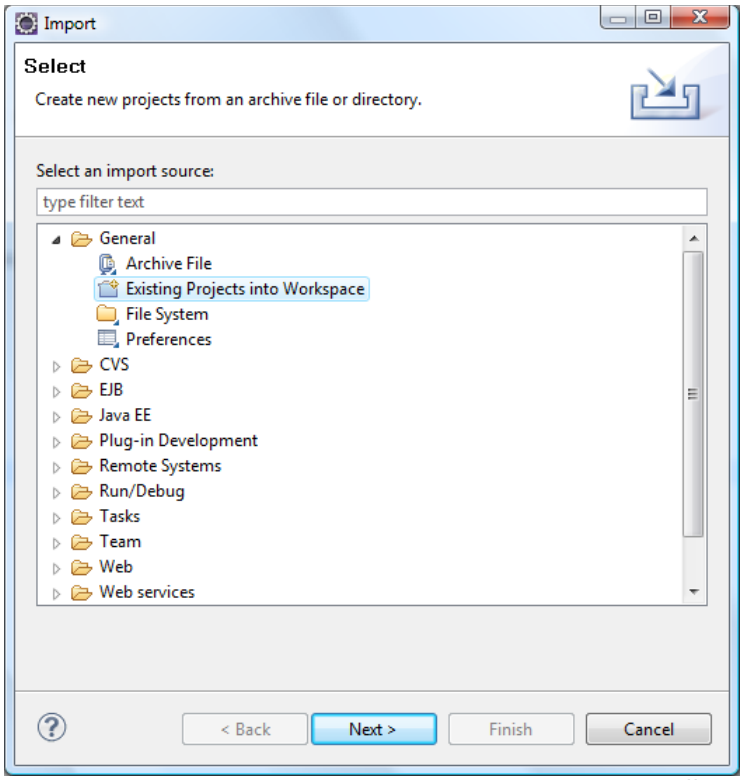


Slika 2.12. Prozor za odabir opcija za izvoz arhive Eclipse projekta

Na prozoru prikazanom na slici 2.12. potrebno je odabrati koje sve projekte je potrebno izvesti (na slici 2.12. odabran je projekt "JavaJedan"), naziv i lokaciju ("C:\Users\Aleksander\Desktop\test.zip") i vrstu arhive ("Save in zip format"), te ostale

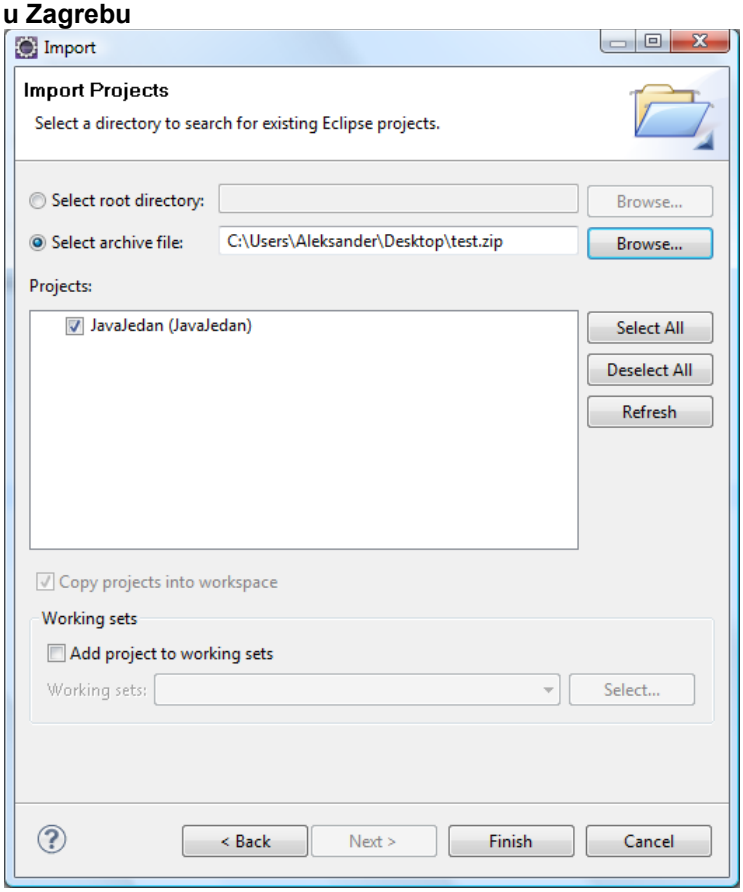
opcije koje su prikazane i na kraju pritisnuti gumb "Finish". Prema primjeru na slici stvoriti će se zip arhiva na navedenoj lokaciji.

Navedenu arhivu je moguće uvesti u neko drugo razvojno okruženje korištenjem opcije File->Import. Odabirom te opcije prikazuje je ekran kao na slici 2.13.



Slika 2.13. Ekran za uvez Eclipse projekata u razvojno okruženje

Na prikazanom ekranu je potrebno odabrani opciju "Existing Projects into Workspace" unutar mape "General" i pritisnuti gumb "Next". Drugi korak kod uvoza arhivirane datoteke u razvojno okruženje se prikazan na slici 2.14.

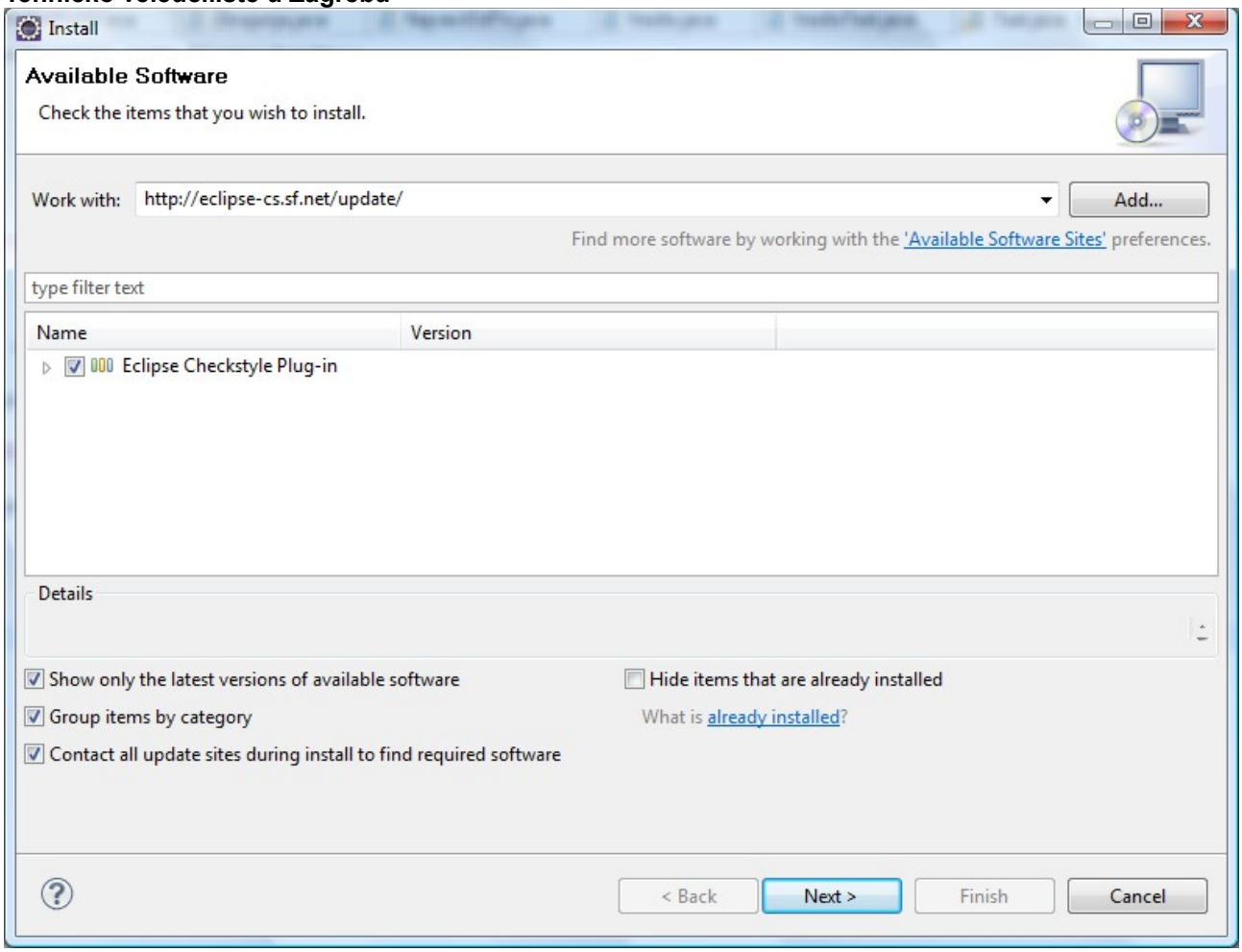


Slika 2.14. Ekran za odabir arhivirane datoteke za uvod u radno okruženje

Na ekranu je potrebno označiti opciju "Select archive file" i odabrati putanju do arhivirane datoteke, nakon čega će se prikazati projekti unutar arhive, te pritisnuti tipku "Finish". Nakon toga se unutar "Package explorer" dijela Eclipse razvojnog okruženja pojavljuje uvezeni projekt koji je istovjetan projektima koji su kreirati unutar tog istog radnog prostora.

2.4. Instalacija dodataka u razvojno okruženje Eclipse

Jedna od najvažnijih karakteristika Eclipse razvojnog okruženja je proširivost pomoću dodataka (engl. *plugins*). Najjednostavniji način dodavanja novih dodataka u Eclipse je pomoću opcije "Help->Install New Software", nakon čega se pojavljuje prozor kao na slici 2.15.



Slika 2.15. Prozor za instalaciju dodataka u razvojno okruženje Eclipse

Na početku je potrebno u tekstualno polje pod nazivom "Work with" upisati URL stranice za ažuriranje (engl. *update site*) uz pomoć koje je moguće preko Interneta dohvatiti potrebne resurse za instalaciju željenih dodataka.

Na slici 2.15. odabran je dodatak pod imenom "Checkstyle" koji služi za provjeru načina pisanja programskog koda. Na primjer, ukoliko programer kod pisanja programskog koda ne poštuje neku od konvencija pisanja Java koda koje je definirala tvrtka Sun, dodatak "Checkstyle" dojavi pogrešku programeru u obliku upozorenja.

Za instaliranje Eclipse dodatka "Checkstyle" potrebno je koristiti stranicu za ažuriranje "http://eclipse-cs.sf.net/update/". Stranice za ažuriranje ostalih dodataka moguće je pronaći na pripadajućim web stranicama za dotične Eclipse dodatke, koje je moguće vrlo lako pronaći pomoću neke od tražilica kao što je Google. Osim toga, uz pomoć opcije "Available Software Sites" koji se u obliku poveznice (engl. *link*) nalazi na prozoru prikazanom na slici 2.15., također je moguće pronaći URL stranica koje sadržavaju dodatke za razvojno okruženje Eclipse.

Nakon unošenja navedenog URL-a u pripadajuće polje, nakon kratkog vremena prikazuje se naziv "Eclipse Checkstyle Plug-in" u središnjem dijelu prozora, kao što je i prikazano na slici 2.15. Označavanjem tog dodatka i pritiskom na tipku "Next" započinje proces

instalacije dodatka. Naravno, prilikom instalacije računalo mora biti spojeno na Internet. Nakon ekrana s detaljima samog dodatka i ekrana s popisom licenci pojavljuje se gumb "Finish" kojim instalacijska procedura dodatka "Checkstyle" ulazi u završnu fazu instalacije kod koje se dohvaćaju i instaliraju potrebni resursi. Na sličan način instaliraju se i ostali dostupni dodaci.

Osim pomoću stranice za ažuriranje, Eclipse dodaci mogu se instalirati i pomoću zip arhive u kojoj se nalazi dodatak. Arhivu je nakon preuzimanja s Interneta potrebno raspakirati u instalacijsku mapu Eclipse-a (npr. "C:\eclipse" ili lokacija u kojoj se nalazi Eclipse). Dodaci su unutar zip arhive najčešće organizirani u mape "plugins" i "features" pa je sadržaje tih mapa potrebno ubaciti u istoimene mape unutar lokacije gdje je instaliran Eclipse na tvrdom disku računala.

Osim instalacije dodatka, pomoću stranice za ažuriranje moguće je i ažurirati dodatke na posljednju dostupnu inačicu pomoću opcije "Help->Check for Updates", čije korištenje je vrlo slično korištenju opcije za instaliranje dodataka u Eclipse.

Nakon instalacije dodatka u Eclipse, vrlo često je potrebno ugasiti i ponovno pokrenuti Eclipse kako bi se instalacijska procedura dodatka uspješno dovršila. To je moguće napraviti "ručnim" gašenjem i pokretanje Eclipse-a, a osim toga i korištenjem opcije "File->Restart".

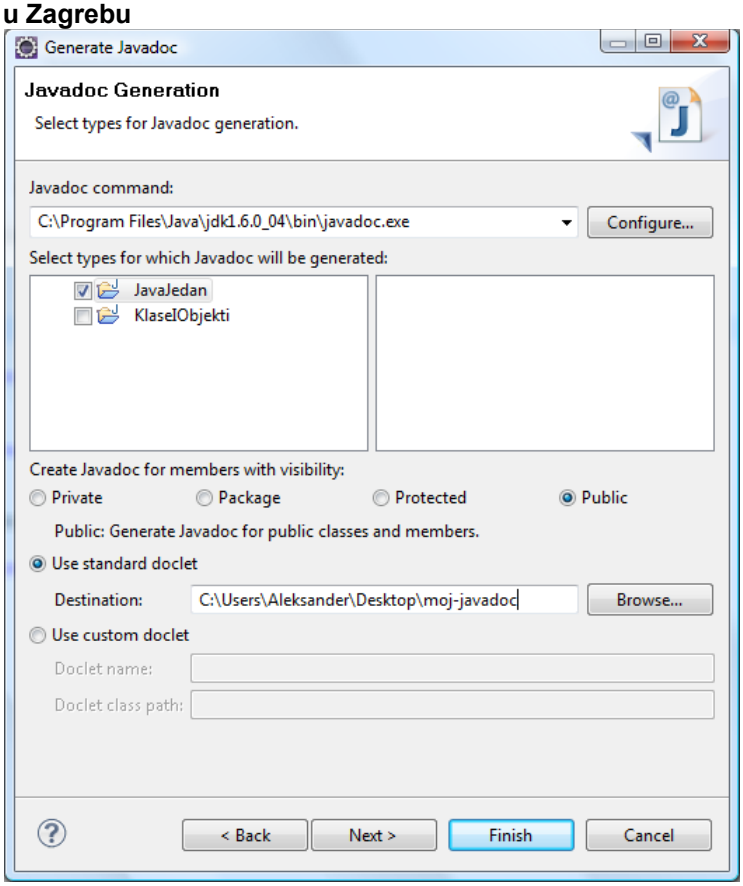
Instalirani "Checkstyle" dodatak moguće je aktivirati zasebno na svakom projektu pritiskom desne tipke miša nad ikonom projekta unutar "Package explorer"-a, te odabirom opcije "Properties". Nakon toga je potrebno odabrati opciju "Checkstyle" i označiti opciju "Checkstyle active for this project". Potvrđivanjem aktivacije dodatak "Checkstyle" na svim datotekama Java izvornog koda pomoću oznaka žute boje označi mjesta gdje je potrebno napraviti neke izmjene kako bi Java kod bio u skladu s propisanim normama.

Dodatne informacije o dodatku "Checkstyle" mogu se pronaći na službenim stranicama:

http://eclipse-cs.sourceforge.net/

2.5. Generiranje Javadoc dokumentacije

Eclipse razvojno okruženje omogućava generiranje *Javadoc* dokumentacije na vrlo jednostavan način. Za to je potrebno imati projekt koji je detaljno dokumentiran i instaliran JDK koji sadrži alat za generiranje *Javadoc* dokumentacije. Generiranje dokumentacije pokreće se pritiskom desne tipke miša nad projektom za koji se želi generirati *Javadoc* dokumentacije i odabrati opciju Export. Nakon toga se pojavljuje prozor naizgled kao na slici 2.13., samo što je potrebno odabrati opciju "Javadoc" unutar mape "Java" i pritisnuti tipku "Next". Nakon toga se pojavljuje prozor kao na slici 2.16.



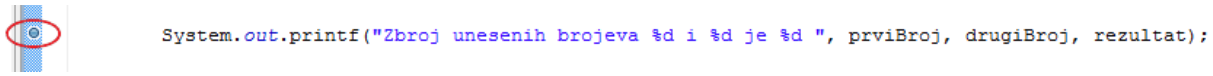
Slika 2.16. Prozor s postavkama za generiranje Javadoc dokumentacije

Na prozoru je najprije potrebno odabrati lokaciju datoteke "javadoc.exe" koja služi na generiranje *Javadoc* dokumentacije. Ukoliko je JDK instaliran u mapu "Program Files\Java", onda je potrebno unutar "bin" podmape pronaći datoteku "javadoc.exe" i označiti je.

Nakon toga je potrebno odabrati projekte za koje je potrebno generirati *Javadoc* dokumentaciju i odabrati lokaciju na koju će Javadoc dokumentacija biti spremljena, te pritisnuti tipku "Finish".

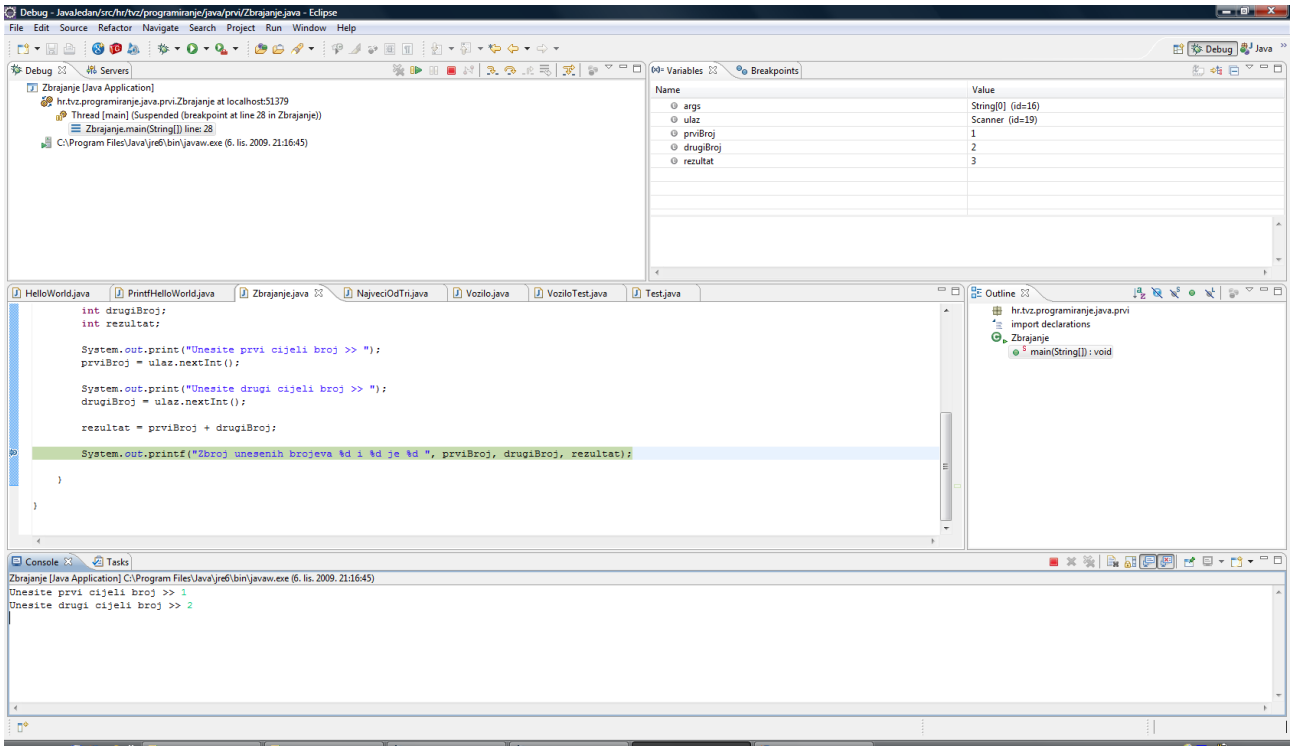
2.6. Pronalaženje i ispravljanje pogrešaka u programima

Jedno od najvažnijih i najmoćnijih karakteristika Eclipse razvojnog okruženja je pronalaženje i ispravljanje pogrešaka u programima (engl. *debugging*). Java program je osim u *Run* način rada (opisanog na 11. stranici) moguće pokrenuti i u *Debug* načinu rada. Razlika između ta dva načina je sastoji se u tome što *Debug* način rada omogućava zaustavljanje izvođenja programa na točkama prekida (engl. *breakpoints*). Kad se izvođenje programa zaustavi na određenoj točki prekida, unutar razvojnog okruženja Eclipse moguće je pregledavati stanja svih varijabli programa koje se u tom trenutku koriste. Prije pokretanja Java programa u *Debug* načinu rada potrebno je postaviti jednu ili više točki prekida. To je moguće napraviti dvostrukim klikom miša pokraj površine za pisane izvornog koda, kako je i prikazano na sljedećoj slici:



Slika 2.17. Dodavanje točaka prekida

Ako se nakon toga program pokrene pomoću opcije "Debug As->Java application", izvođenje programa će se zaustaviti na točki prekida te pitati korisnika želi se otvoriti novu perspektivu pod nazivom *Debug* čiji izgled je optimiziran za pronalaženje i ispravljanje pogrešaka u programima (slika 2.18.).



Slika 2.18. Debug perspektiva unutar razvojnog okruženja Eclipse

Unutar *Debug* perspektive vidi se programski kod s točkama prekida gdje je zelenom bojom označen redak u kojem je prekinuto izvođenje programa. U tom trenutku izvođenja programa sve dotad korištenje varijable imaju neke vrijednosti koje je moguće vidjeti u gornjem desnom dijelu prozora. Kako bi se provjerila ispravnost programa, vrijednosti tih varijabli je moguće i promijeniti korištenjem opcije "Change Value" koju je moguće odabrati nakon pritiska desne tipke miša nad varijablom.

Rukovanje programom zaustavljenim na nekoj od točaka prekida moguće je obavljati pomoću gumbića lijevo od dijela prozora s varijablama koji su u obliku strelice zelene boje (koja omogućava izvođenje programa do kraja ili do sljedeće točke prekida) i kvadratića crvene boje (koja služi za prekidanje programa). Pored njih postoje i gumbići za odvijanje programa liniju po liniju ili ulazak programa u funkciju, slično kao i u nekim drugim alatima za razvijanje u nekim drugim programskim jezicima.

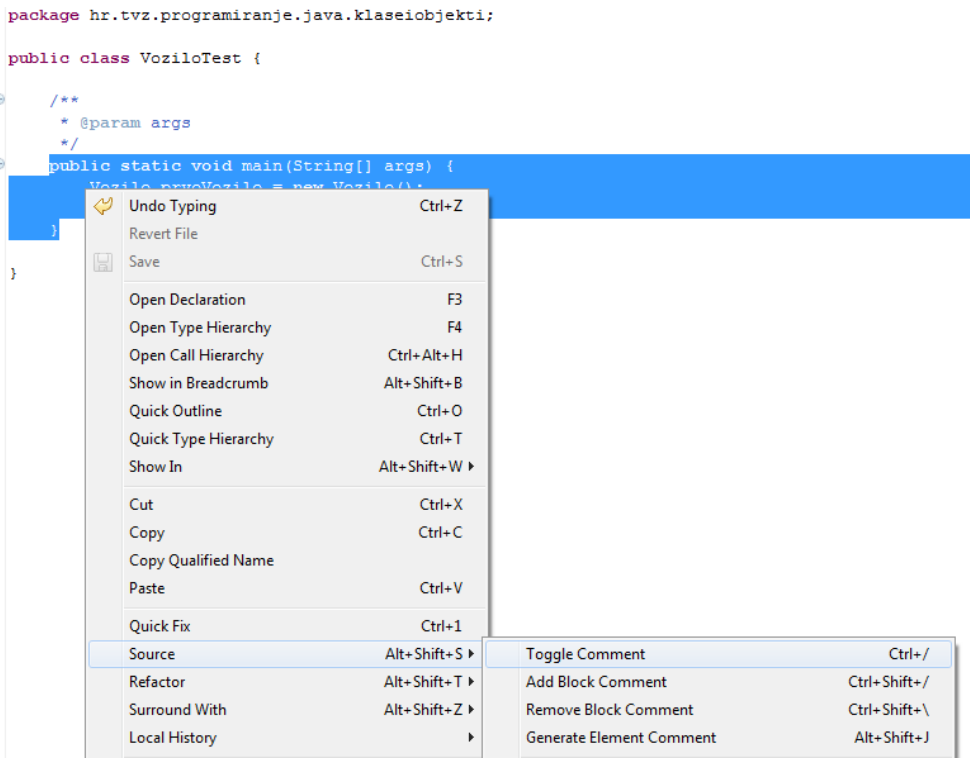
2.7. Korisne funkcionalnosti razvojnog okruženja Eclipse

2.7.1. Zakomentiravanje programskog koda

Tijekom razvoja aplikacija često je potrebno privremeno ili trajno "deaktivirati" dio programskog koda. To je moguće učiniti tako da se zakomentirava "linija po linija" pomoću komentara za jednu liniju ("//"), zakomentirava cijeli blok programskog koda komentarom za više linija ("/*" i "*/) ili možda najjednostavnije, korištenjem opcije unutar Eclipse

razvojnog okruženja.

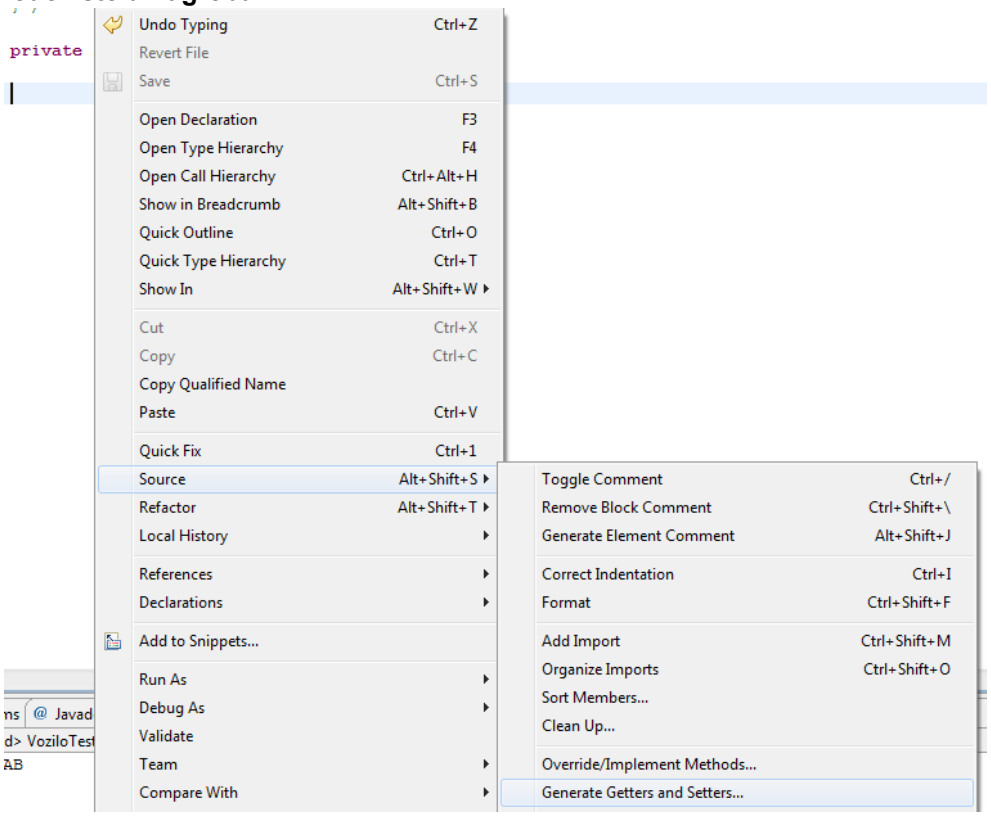
Potrebno je mišem označiti programski kod koji se želi zakomentirati, pritisnuti desnu tipku miša i odabrati opciju "Source->Toggle Comment" ili neku od drugih opcija vezanih uz zakomentiravanje programskog koda (slika 2.19.).



Slika 2.19. Opcije za dodavanje komentara

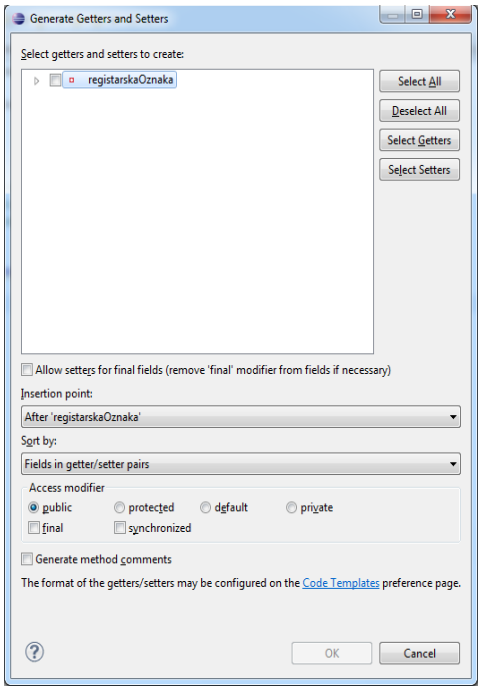
2.7.2. Automatsko generiranje "*getter*" i "*setter*" metoda

Klase u Javi često imaju funkciju pohranjivanja podataka o određenim entitetima kao što je vozilo, trokut, karta ili slično. Svaka takva "domenska" klasa ima nekoliko privatnih varijabli (polja) koje se ne mogu dohvaćati izravno, već se njihove vrijednosti dohvaćaju pomoću tzv. "*getter*" i "*setter*" metoda. Te metode uvijek imaju isti oblik (ulazni parametri i povratne vrijednosti definirane su tipom varijable) te ih je radi uštede vremena moguće generirati i automatski korištenjem opcije unutar razvojnog okruženja Eclipse.



Slika 2.20. Opcija za automatsko generiranje "*getter*" i "*setter*" metoda

Pritiskom na desnu tipku miša na površini za pisanje programskog koda otvara se izbornik kao na slici 2.20., nakon čega je potrebno odabrati opciju "Source->Generate Getters and Setters...". Odabirom opcije prikazuje se ekran kao na slici 2.21.



Slika 2.21. Ekran za automatsko generiranje "getter" i "setter" metoda

Na tom ekranu moguće je definirati za koje privatne varijable je potrebno generirati "getter" i "setter" metode, poziciju gdje će te metode biti smještene i slično. Pritiskom na gumb "OK" generiraju se metode prema zadanim parametrima. Primjer generiranja "getter" i "setter" metode za privatnu varijablu "registarskaOznaka" tipa String prikazan je u nastavku:

```
private String registarskaOznaka;

public String getRegistarskaOznaka() {
      return registarskaOznaka;
}

public void setRegistarskaOznaka(String registarskaOznaka) {
      this.registarskaOznaka = registarskaOznaka;
}
```

2.7.3. Automatsko strukturiranje programskog koda

Tijekom razvijanja aplikacija često se programski kod ne piše na način da bude što čitljiviji, odnosno, tako da je svaki blok omeđen vitičastim zagrada uvučen i tako vizualno prepoznatljiv. Takav ispravan način pisanja programskog koda naziva se "strukturiran programski kod". Razvojno okruženje Eclipse na vrlo lagani način omogućava pretvorbu nestrukturiranog koda u strukturirani kod. Na primjer, ukoliko je napisan sljedeći programsko odsječak koji nije napisan strukturirano:

```
if (vrstaEkranaA.equals(vrstaEkrana)) {
a_inch = (float) 3 * dijagonala_inch / 5;
b_inch = (float) 4 * dijagonala_inch / 5;
a_cm = a_inch / CM_U_INCH;
b_cm = b_inch / CM_U_INCH;} else {
a_inch = (float) dijagonala_inch / (float) Math.sqrt((float)337/81);
b_inch = (float) 16 * a_inch / 9;
a_cm = a_inch / CM_U_INCH;
b_cm = b_inch / CM_U_INCH;}
```

Navedeni programski kôd se korištenjem opcije "Source->Format" (koju je moguće odabrati iz izbornika nakon što se pritisne desna tipka miša nad samim programskim kodom – opcija je prikazana na slici 2.20.) može vrlo brzo strukturirati nakon čega izgleda puno čitljivije:

```
if (vrstaEkranaA.equals(vrstaEkrana)) {
      a_inch = (float) 3 * dijagonala_inch / 5;
      b_inch = (float) 4 * dijagonala_inch / 5;
      a_cm = a_inch / CM_U_INCH;
      b_cm = b_inch / CM_U_INCH;
} else {
      a_inch = (float) dijagonala_inch/(float) Math.sqrt((float) 337 / 81);
      b_inch = (float) 16 * a_inch / 9;
      a_cm = a_inch / CM_U_INCH;
      b_cm = b_inch / CM_U_INCH;
}
```

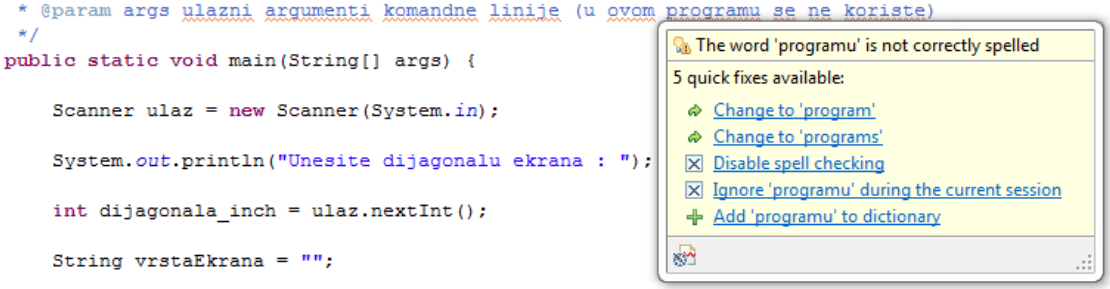
2.7.4. Isključivanje provjere pravopisa kod "javadoc" komentara

U razvojnom okruženju Eclipse prema pretpostavljenim (engl. *default*) postavkama uključeno je automatsko provjeravanje pravopisa kod "javadoc" komentara. Javadoc komentare je vrlo često potrebno pisati na engleskom jeziku kako bi svi korisnici mogli koristiti programski kôd, međutim, u fazi učenja pisanja "javadoc" dokumentacije često se tolerira pisanje dokumentacije i na hrvatskom jeziku.

```
/**
 * Glavna metoda koja služi za pokretanje programa.
 *
 * @param args ulazni argumenti komandne linije (u ovom programu se ne koriste)
 */
public static void main(String[] args) {

    Scanner ulaz = new Scanner(System.in);

    System.out.println("Unesite dijagonalu ekrana : ");

    int dijagonala_inch = ulaz.nextInt();

    String vrstaEkrana = "";
```

Slika 2.22. Prijavljene pravopisne pogreške kod "javadoc" komentara

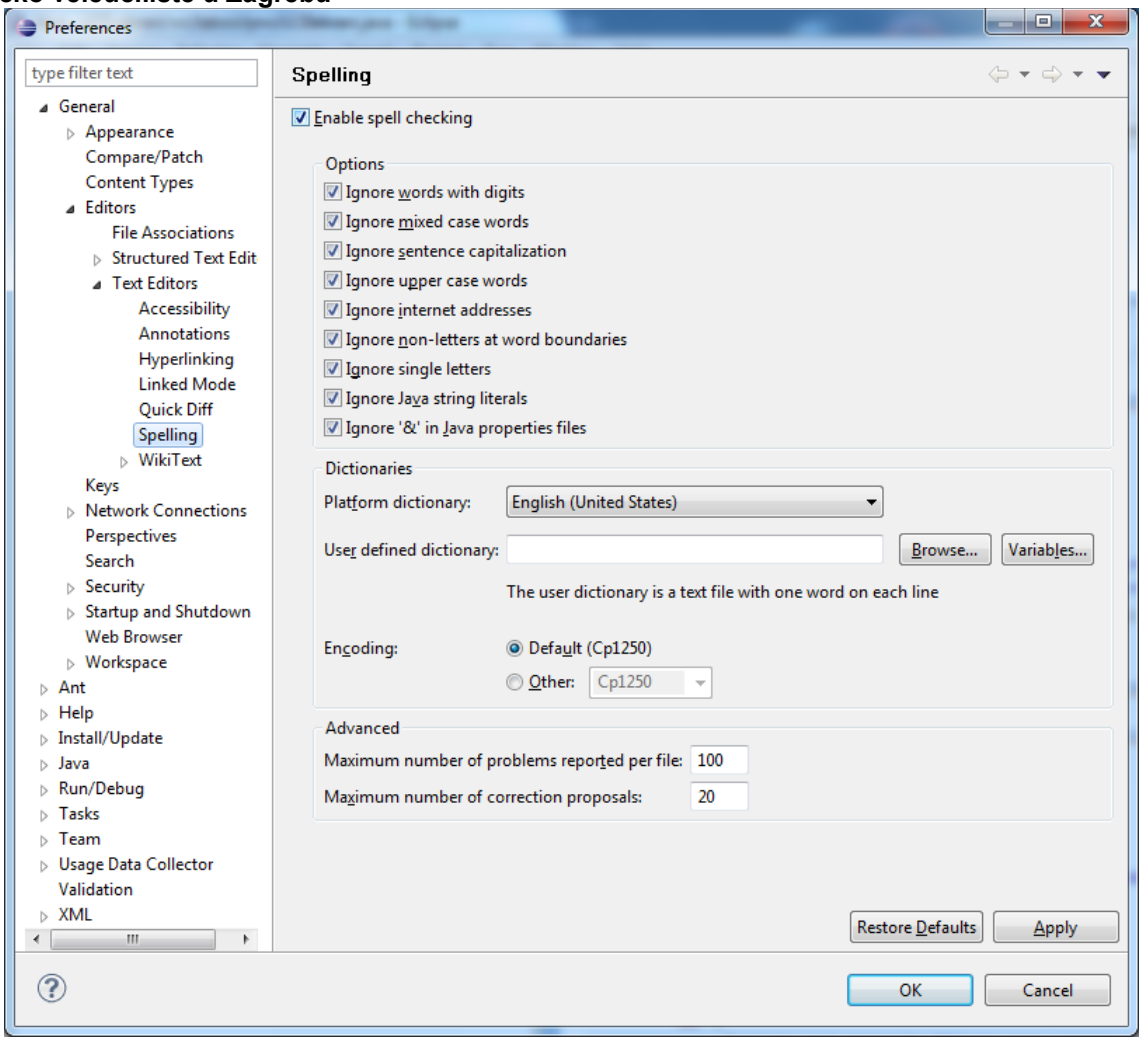
Isključivanje provjere pravopisa kod "javadoc" komentara najlakše je moguće isključiti odabirom opcije "Disable spell checking" koju je moguće odabrati iz izbornika prikazanog na slici 2.23. (navedeni izbornik prikazuje se prilikom postavljanja kursora miša na jednu od prijavljenih pravopisnih pogrešaka u "javadoc" komentarima).



Slika 2.23. Odabir opcije za isključivanje provjere pravopisa "javadoc" komentara

Osim tog najkraćeg puta, moguće je odabrati i opciju "Window->Preferences->General->Editors->Text Editors->Spelling", te na prikazanom ekranu (slika 2.24.) isključiti *checkbox* "Enable spell checking".

Osim toga u izborniku "Window->Preferences" postoji mnoštvo drugih opcija vezanih uz konfiguriranje i prilagođavanje razvojne okoline Eclipse.



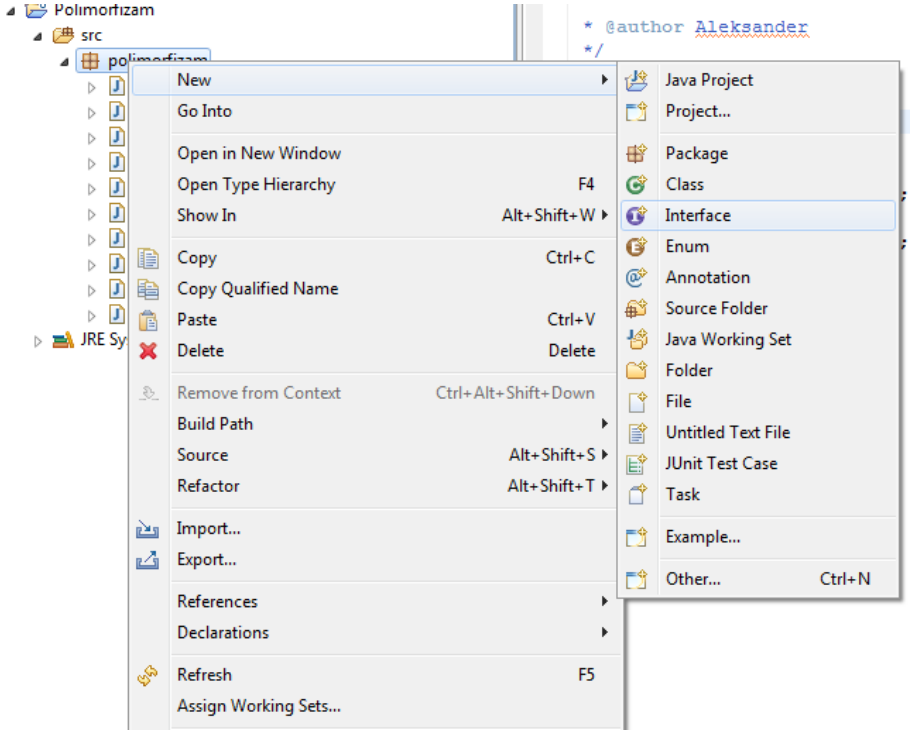
Slika 2.24. Ekran s konfiguracijskim postavkama za provjeravanje pravopisa "javadoc" komentara

2.7.5. Dodavanje Java sučelja u radno okruženje

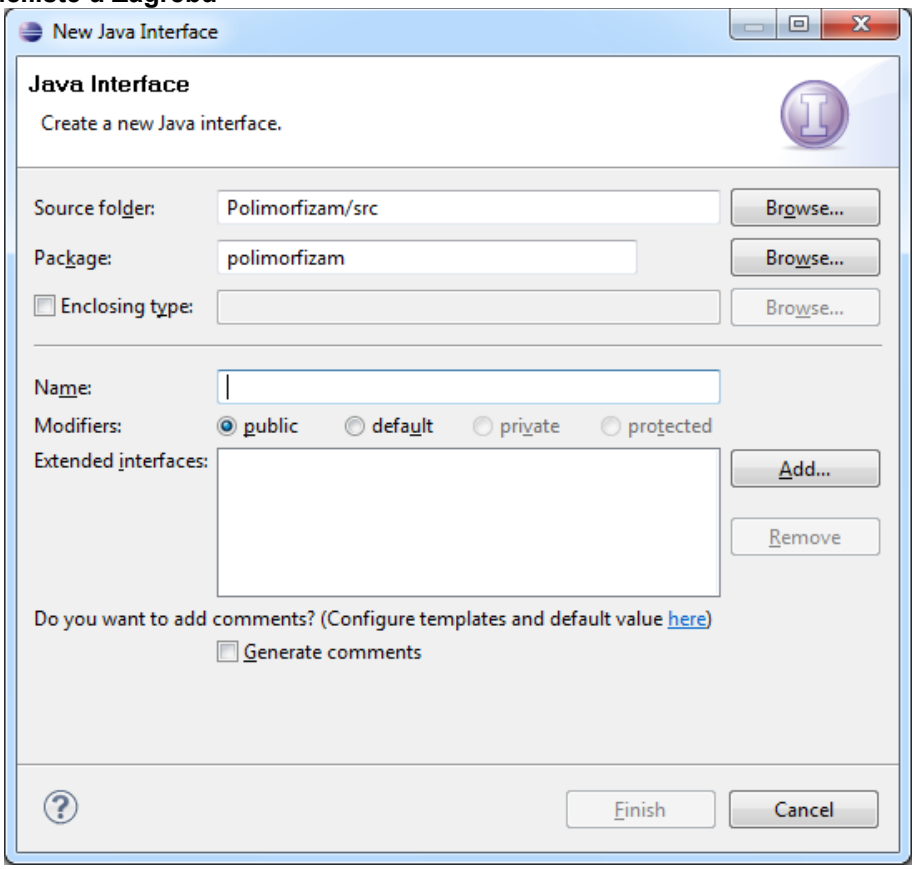
Osim dodavanja samih Java klasa u pakete koji su dio nekog Java projekta u razvojnom okruženju Eclipse (što je opisano slikom 2.8.), često je potrebno kreirati i sučelja (engl. *interface*) koja neke klase nasljeđuju u kontekstu objektno-orijentiranog programiranja.

Dodavanje novih sučelja moguće je obaviti odabirom opcije "File->New->Interface" ili pritiskom desne tipke miša nad ikonom koja predstavlja paket u razvojnoj okolini i odabrati "New->Interface".

Nakon toga će se prikazati dijalog kao na slici 2.26. na kojem je moguće unijeti parametre sučelja kao što su ime, paket, sučelja koje nasljeđuje novo sučelje itd.



Slika 2.25. Dodavanje sučelja u razvojno okruženje



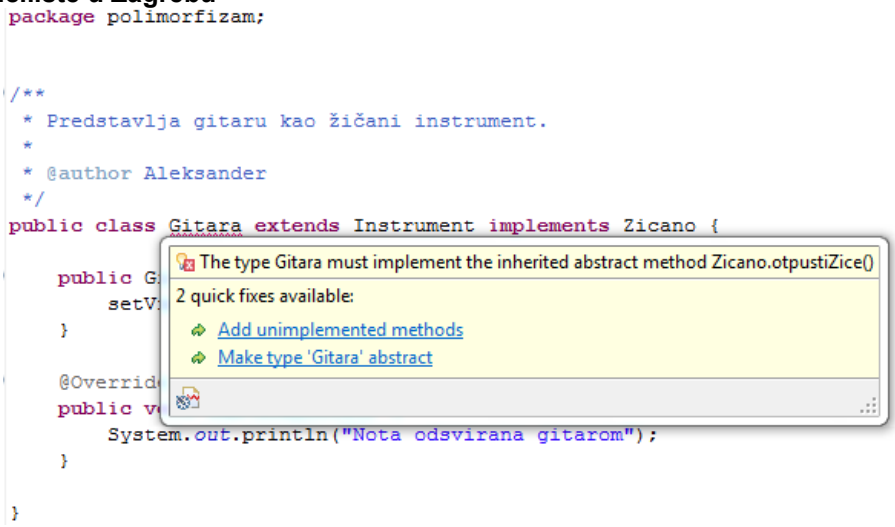
Slika 2.26. Dijalog za definiranje parametara sučelja

2.7.6. Generiranje kostura metoda iz sučelja u implementacijskoj klasi

Vrlo često se tijekom razvoja tek naknadno dodaje sučelje pa je u klasama koje implementiraju to sučelje potrebno "ručno" dodavati kosture metoda iz sučelja kako bi se mogle implementirati.

Osim "ručnog" dodavanja kostura metoda, razvojno okruženje Eclipse nudi i mogućnost automatskog generiranja tih kostura. Nakon dodavanja ključnih riječi "**`implements Zicano`**", klasa "**`Gitara`**" ne implementira sve potrebne metode zbog čega razvojno okruženje Eclipse dojavljuje pogrešku kod prevođenja.

Tu pogrešku moguće je automatski ispraviti na način da se pokazivač miša pozicionira iznad naziva klase koji je podcrtan crvenom linijom koja označava pogrešku kod prevođenja, nakon čega se pojavljuje prozor koji omogućava opciju "Add unimplemented methods", kao što je prikazano na slici 2.27.



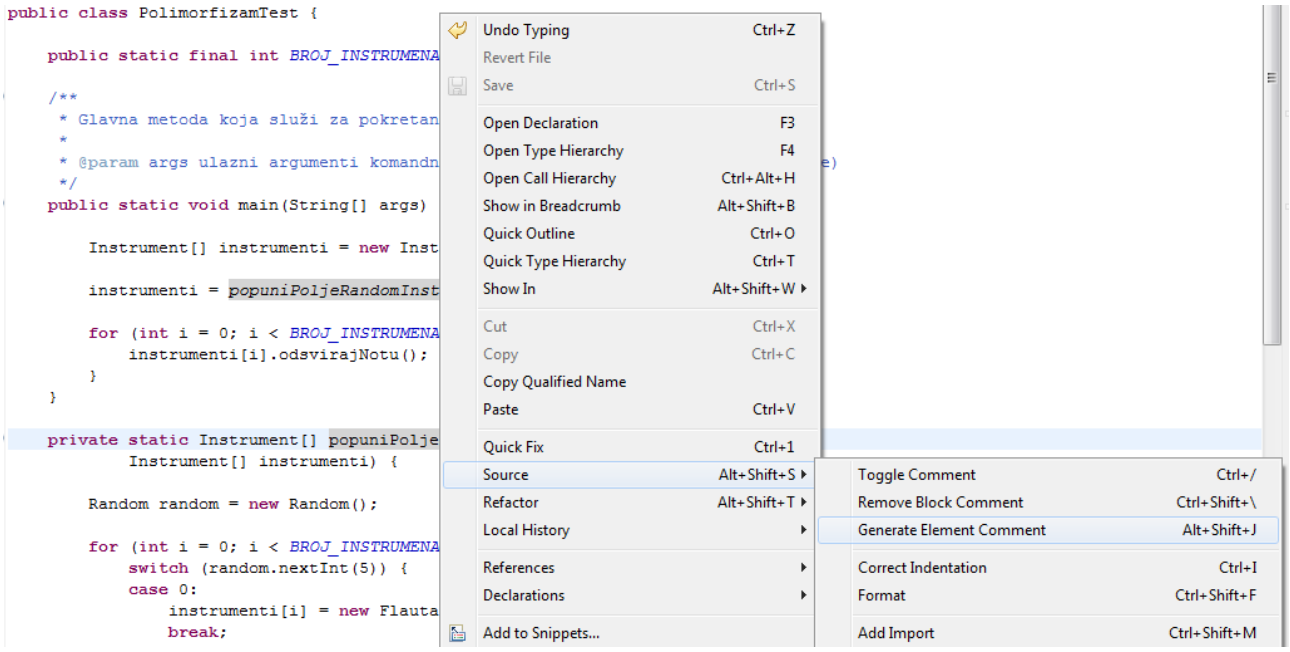
Slika 2.27. Opcija za automatsko generiranje kostura metoda iz sučelja

Nakon odabira te opcije unutar tijela klase pojavljuju se kosturi svih metoda koje je potrebno implementirati unutar klase.

2.7.7. Automatsko generiranje kostura za "Javadoc" dokumentaciju

Osim kostura metoda iz sučelja, razvojno okruženje Eclipse omogućava i automatsko generiranje kostura za javadoc dokumentaciju. Javadoc dokumentacija može se generirati za metode i klase unutar prostora za pisanje programskog koda.

Javadoc dokumentaciju moguće je generirati na način da se pokazivač miša pozicionira iznad naziva klase ili metode, pritisne desna tipka miša, te odabere opcija "Source->Generate Element Comment", kao što je i prikazano na slici 2.28.



Slika 2.28. Automatsko generiranje Javadoc komentara

Osim pritiska desne tipke miša, istu akciju je moguće obaviti i korištenjem kombinacije

tipki "Alt+Shift+J".

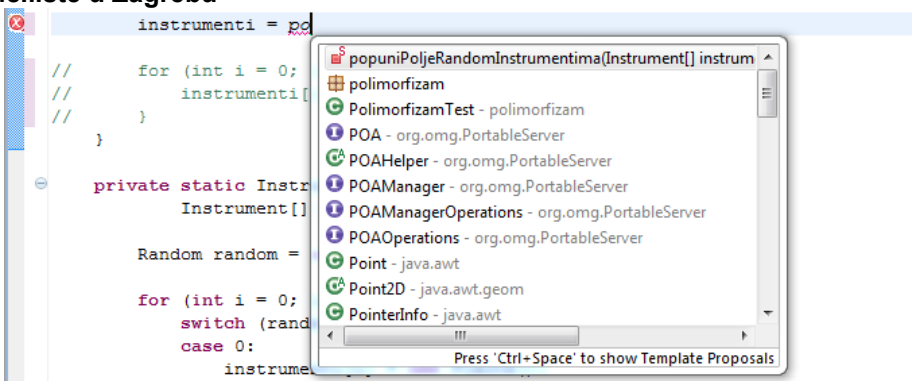
2.7.8. Automatsko dovršavanje naredbi

Jedna od najkorisnijih opcija u razvojnom okruženju Eclipse je automatsko dovršavanje naredbi. Ono se svodi na dovršavanje započetih naziva varijabli, metoda ili klasa tijekom pisanja programskog koda, što može značajno unaprijediti brzinu programiranja.

Na primjer, ukoliko je potrebno u kodu pozvati metodu dugačkog imena "**`popuniPoljeRandomInstrumentima`**", dosta je izgledno da se tijekom pisanja naziva metode dogodi pogreška. Kako bi se iskoristila opcija automatskog dovršavanja naredbi, dovoljno je napisati samo početni dio imena metode i pritisnuti kombinaciju tipki "Ctrl + Space bar".

Nakon toga pojavljuje se lista metoda, klasa, paketa, sučelja i ostalih tipova čiji naziv počinje sa zadanim prefiksom koji je upisan (slika 2.29.).

Vrlo je korisno korištenje kratice za generiranje naredbe "System.out.println", bez potrebe za pisanjem cijelog naziva metode. Ako se upiše samo "sysout" i pritisne kombinacija tipku "Ctrl + Space bar", izgenerira se cijeli naziv naredbe pa još samo potrebno upisati tekst koji se želi ispisati u konzolu.



Slika 2.29. Korištenje opcije automatskog generiranja programskog koda u Eclipse-u

3. OSNOVNA SVOJSTVA PROGRAMSKOG JEZIKA JAVA

3.1. Tipovi podataka

Java podržava osam osnovnih tipova podataka koji se također nazivaju i **primitivni tipovi** (engl. *primitive types*) – jednostavni neobjektni tipovi koji predstavljanju neku vrijednost. Primitivni tipovi (tablica 3-1) uključuju logički tip (engl. *boolean type*), znakovni tip (engl. *charachter type*), četiri cjelobrojna tipa (engl. *integer types*) i dva tipa s pomičnim zarezom (engl. *floating-point types*).

Tablica 3.1. Svojstva primitivnih tipova podataka u Javi

Tip	***Vrijednost***	***Default***	***Veličina***	***Raspon***
boolean	**true** ili **false**	false	1 bit	-
char	Unicode znak	\u0000	16 bitova	od \u0000 do \uFFFF
byte	Cijeli broj s predznakom	0	8 bitova	od -2^7 do 2^7-1
short	Cijeli broj s predznakom	0	16 bitova	od -2^{15} do $2^{15}-1$
int	Cijeli broj s predznakom	0	32 bitova	od -2^{31} do $2^{31}-1$
long	Cijeli broj s predznakom	0	64 bitova	od -2^{63} do $2^{63}-1$
float	Decimalni broj (IEEE 754)	0.0	32 bitova	od 1.4E-45 do 3.4028235E+38
double	Decimalni broj (IEEE 754)	0.0	64 bitova	od 4.9E-324 do 1.7976931348623157E+308

Osim primitivnih tipova, Java podržava i neprimitivne tipove kao što su klase (engl. *classes*), sučelja (engl. *interfaces*) te polja (engl. *arrays*).

3.1.1. Logički tip

Logički tip (engl. *boolean*) predstavlja logičku vrijednost prikazanu s vrijednostima **true** ili **false** (istina ili laž). Za razliku od jezika *C* i *C++*, Java ima vrlo stroga pravila vezana uz logički tip. Vrijednosti **true** ili **false** ne mogu se pretvoriti (engl. *convert*) iz drugih tipova podataka ili u neke druge tipove podataka. Primjer korištenja:

```
boolean logickiTip = true;
```

3.1.2. Znakovni tip

Znakovni tip (engl. *char*) predstavlja *Unicode* znakove. Koriste se unutar jednostrukih navodnika (apostrofa):

```
char znak = 'A';
```

Osim navođenja konkretnih znakova, varijabla znakovnog tipa može poprimati i vrijednosti neispisivih (engl. *nonprinting*) znakova koji se definiraju pomoću *escape* sekvence:

```
char tabulator = '\t', noviRed = '\n', backspace = '\b';
```

Također postoji i *Unicode escape* sekvenca pomoću koje se mogu ispisati svi znakovi. Ovo svojstvo se vrlo često koristi za ispisivanje hrvatskih znakova *č, ć, ž, đ* i *š*.

Tablica 3.2. Unicode-kodovi za hrvatske znakove

Znak	Unicode
Ć	\u0106
Č	\u010C
Đ	\u0110
Š	\u0160
Ž	\u017D
ć	\u0107
č	\u010D
đ	\u0111
š	\u0161
ž	\u017E

3.1.3. String

Osim znakovnog tipa, Java podržava tip podatka za rad s nizom znakova (engl. *String*). Tip *String* nije primitivan tip podatka, već je predstavljen istoimenom klasom. *String* može biti bilo koji tekst omeđen dvostrukim navodnicima:

```
String pozdravnaPoruka = "Ovo je 'pozdravna' poruka";
```

3.1.4. Cjelobrojni tipovi

Cjelobrojni tipovi u Javi su **byte**, **short**, **int** i **long**, a razlikuju se samo po veličini, odnosno rasponu brojeva čije vrijednosti mogu poprimiti (tablica 3.1.). Svi navedeni tipovi mogu poprimiti pozitivne i negativne vrijednosti (ne postoji ključna riječ **`unsigned`** kao u jezicima C i C++). Primjeri korištenja:

```
byte dva = 2;
short stojedanaest = 111;
int desetTisuca = 10000;
long desetMilijuna = 10000000;
```

Aritmetika cjelobrojnih brojeva u Javi je modularna (engl. *modular*), što znači da nije moguće prekoračenje maksimalne (engl. *overflow*) ili minimalne vrijednosti (engl. *underflow*). Umjesto toga, kod prekoračenja vrijednosti dolazi do efekta kružnog nastavljanja niza vrijednosti (engl. *wrap around*):

```
// Najveća vrijednost tipa byte je 127
byte b1 = 127, b2 = 1;

// Suma postaje -128, što je najmanji byte broj
byte sum = (byte)(b1 + b2);
```

3.1.5. Tipovi podataka s pomičnim zarezom

Realni brojevi se u Javi prikazuju pomoću tipova podataka **float** i **double**. Kao što je prikazano u tablici 3.1., **float** predstavlja realne brojeve s 32-bitnom preciznosti, dok **double** predstavlja realne brojeve s dvostrukom, 64-bitnom preciznosti.

Decimalni brojevi deklariraju se kao niz brojeva s decimalnom točkom koja označava decimale broja (primjeri: **`123.45`**, **`0.0`**, **`.01`**). Osim toga, varijable koje sadrže jako velike ili jako male realne vrijednosti mogu se prikazati uz pomoć eksponencijalnog zapisa, npr.

```
double varijabla = 1.23E41; //predstavlja vrijednost 1.23 * 10^41
```

3.1.6. Konverzija primitivnih tipova

Java omogućava konverziju između cjelobrojnih i realnih tipova podataka. Postoje dva osnovna tipa konverzije, konverzija s proširenjem (engl. *widening conversion*) i konverzija sa suženjem (engl. *narrowing conversion*).

Konverzija s proširenjem obavlja se prilikom pretvorbe vrijednosti podatka u tip podatka koji pokriva veći skup podataka. Java vrši automatsku konverziju s proširenjem prilikom, na primjer, konverzije podataka cjelobrojnog tipa (*int*) u podatak *double* tipa. Primjer konverzije s proširenjem dan je u sljedećem kodu (vrijednost varijable **`drugi`** nakon izvršenja koda iznosi **`345.0`**).

```
int prvi = 345;
double drugi = prvi;
```

Konverzija sa suženjem obavlja se prilikom pretvorbe vrijednosti podatka u tip koji pokriva manji skup podataka. Međutim, takve pretvorbe nisu uvijek pouzdane. Ukoliko se želi konvertirati, na primjer, cjelobrojna vrijednost '1500' u tip podatka `byte`, koji može poprimiti samo vrijednosti od -128 do 127, kompajler (engl. *compiler*) će programeru dojaviti pogrešku:

```
int i = 13;
byte b = i;    // Kompajler ne dopušta ovakvu pretvorbu
```

Međutim, ukoliko je potrebno izvršiti konverziju prilikom koje je programer siguran da neće doći do gubitka podataka ili preciznosti, moguće je koristiti konverziju pomoću jezične konstrukcije zvane *cast*. Takva konverzija obavlja se ukoliko se navede tip podatka unutar oblih zagrada ispred varijable koja se konvertira:

```
int i = 13;
byte b = (byte) i;   // Konverzija int u byte
i = (int) 13.456;    // Konverzija double vrijednosti u int 13
```

Tablica 3.3. sadrži informacije o tome koji primitivni tipovi se mogu konvertirati. Oznaka *'N'* označava da nije moguća konverzija, oznaka *'D'* označava da je konverzija moguća, oznaka *'C'* označava konverziju sa suženjem kod koje se koristi konstrukcija *cast*, dok oznaka *'D*'* označava automatsku konverziju s proširenjem, međutim, moguć je gubitak znamenki s najmanjom težinom (engl. *least significant digits*).

Tablica 3.3. Konverzije primitivnih tipova u Javi

Konverzija	**Konverzija u:**							
iz:	**boolean**	**byte**	**short**	**char**	**int**	**long**	**float**	**double**
boolean	-	N	N	N	N	N	N	N
byte	N	-	D	C	D	D	D	D
short	N	C	-	C	D	D	D	D
char	N	C	C	-	D	D	D	D
int	N	C	C	C	-	D	D*	D
long	N	C	C	C	C	-	D*	D*
float	N	C	C	C	C	C	-	D
double	N	C	C	C	C	C	C	-

3.2. *Operatori*

Vrsta izraza koji se može oblikovati u programskom jeziku ovisi o operatorima koje podržava taj jezik. U tablici 3.4. navedeni su operatori koje podržava programski jezik Java. U stupcima ***P*** i ***A*** nalaze se informacije o prednostima, odnosno asocijativnosti operatora. Detaljnije objašnjenje operatora i njihovih svojstava dan je u nastavku. Operatori i svojstva koja su identična kao i u programskom jeziku C nisu posebno navedena.

Tablica 3.4. Operatori u Javi

P	**A**	**Operator**	**Tipovi operanada**	**Operacija**
15	L	**.**	objekt, članske varijable	dohvaćanje objekata i članskih varijabli
		[ ]	polja, cijeli brojevi	dohvaćanje elemenata polja
		(*argumenti*)	metode, liste argumenata	pozivanje metoda
		++, --	varijable	post-inkrementiranje, dekrementiranje
14	D	**++, --**	varijable	post-inkrementiranje, dekrementiranje
		+, -	brojevi	unarni plus, unarni minus
		~	cijeli broj	komplementiranje bitova
		!	logičke varijable	logička negacija
13	D	**new**	klase, liste argumenata	instanciranje objekata
		(tipovi)	razne vrste tipova	*cast* (konverzija tipova)

P	A	Operator	Tipovi operanada	Operacija
12	L	***, /, %***	brojevi	množenje, dijeljenje, cjelobrojni ostatak
11	L	***+, -***	brojevi	zbrajanje, oduzimanje
		+	*string*, razni tipovi	konkatenacija *stringova*
10	L	***<<***	cijeli brojevi	pomicanje *(shift)* bitova ulijevo
		>>	cijeli brojevi	pomicanje *(shift)* bitova udesno s predznakom
		>>>	cijeli brojevi	pomicanje *(shift)* bitova udesno bez predznaka
9	L	***<, <=***	cijeli brojevi	manje, manje ili jednako
		>, >=	cijeli brojevi	veće, veće ili jednako
		instanceof	reference i tipovi	uspoređivanje tipova
8	L	**= =**	primitivni tipovi	jednakost (identične vrijednosti)
		!=	primitivni tipovi	nejednakost (različite vrijednosti)
		= =	reference	jednakost (pripadnost istom objektu)
		!=	reference	nejednakost (pripadnost različitim objektima)
7	L	**&**	cijeli brojevi	operacija **AND** nad bitovima
		&	logičke vrijednosti	logička operacija **AND**
6	L	**^**	cijeli brojevi	operacija **XOR** nad bitovima
		^	logičke vrijednosti	logička operacija **XOR**
5	L	**|**	cijeli brojevi	operacija **OR** nad bitovima
		|	logičke vrijednosti	logička operacija **OR**
4	L	**&&**	logičke vrijednosti	uvjetna operacija **AND**
3	L	**||**	logičke vrijednosti	uvjetna operacija **OR**
2	R	**?:**	razni tipovi	ternarni uvjetni operator
1	R	**=**	razni tipovi	Pridruživanje
		***=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=**	razni tipovi	pridruživanje s operacijom

3.2.1. Razina prednosti operatora

Stupac **P** tablice 3.4. predstavlja razinu prednosti svakog operatora, odnosno redoslijed obavljanja operacija. Na primjer, u izrazu

```
a + b * c
```

operator množenja ima veći prioritet (**P** vrijednost 12) od operatora zbrajanja (**P** vrijednost 11). Razina prednosti operatora može se promatrati kao mjera vezanosti operatora uz operand, odnosno, što je veća vrijednost razine prednosti u tablici 1-4, to je operator jače vezan uz operand. Prednost operatora također može biti poništena korištenjem oblih zagrada, pa operacija zbrajanja može imati prednost pred operacijom množenja u sljedećem obliku izraza:

```
(a + b) * c
```

Pretpostavljene (engl. *default*) razine prednosti operatora određena je tako da bude kompatibilna s programskim jezikom C.

3.2.2. Asocijativnost

Ukoliko se u izrazu pojavljuju operatori jednake razine, asocijativnost operatora određuje redoslijed po kojem će se obavljati operacije s tim operatorima. Većina operatora je asocijativna s lijeva na desno, što znači da se operacije obavljaju s lijeva na desno. Jedino su pridruživanja i unarni operatori asocijativni s desna na lijevo. Stupac **A** u tablici 3.4. označava vrstu asocijativnosti operatora ili grupe operatora: oznaka **L** označava asocijativnost s lijeva na desno, dok oznaka **R** označava asocijativnost s desna na lijevo.

Operatori zbrajanja su asocijativni s lijeva na desno pa se izraz `a+b-c` izvršava s lijeva na desno: `(a+b)-c`. Unarni operatori i pridruživanja izvršavaju se s desna na lijevo. Na primjer, izraz

```
a = b += c = -~d
```

izvršava se sljedećim redoslijedom:

```
a = (b += (c = -(~d)))
```

3.2.3.Tipovi operanada

Četvrti stupac tablice predstavlja tipove operanada na koje se može primijeniti pojedini operator. Neki operatori obavljaju operacije samo nad jednim operandom; takvi operatori se zovu ***unarni***. Na promjer, unarni operator minus mijenja predznak broja kojemu je pridodijeljen:

```
-n // unarni minus operator
```

Većina operatora su ***binarni*** operatori koji primjenjuju operaciju na vrijednosti od dva operanda. Operator – dolazi i kao binarni operator:

```
a – b //binarni operator za oduzimanje vrijednosti varijabli a i b
```

Osim unarnih i binarnih, Java definira i jedan ***ternarni*** operator, koji se često naziva i ***uvjetni*** operator. Na njega se može gledati kao na **`if`** uvjet unutar izraza. Ternarni operator uključuje tri operanda odijeljena upitnikom (?) i dvotočkom (:):

```
x > y ? x : y // ternarni operator koji vraća veći broj, x ili y
```

3.2.4. Operator za konkatenaciju stringova

Osim za zbrajanja brojeva, operator + (kao i pripadajući operator +=) također služe za konkatenaciju, odnosno, spajanje *stringova*. Ukoliko je bilo koji od operanada tipa *String*, operator + sve ostale operande također pretvara u *string*. Na primjer, naredba

```
System.out.println("Kvocijent brojeva 45 i 12 je '" + 45/12.
+ "'.");
```

ispisuje

```
Kvocijent brojeva 45 i 12 je '3.75'.
```

Java interpreter ima ugrađenu konverziju svih primitivnih tipova u *string*, tako da se mogu koristi kao u gornjem primjeru. Objekti također imaju svoju *string* reprezentaciju koja se dobiva pozivanjem metode **`toString()`**. Tu metodu moguće je nadjačati (engl. *override*) i time definirati na koji način će se objekt konvertirati u *string*.

Kao u programskom jeziku C, u Javi je također potrebno pretvoriti cjelobrojne brojeve u decimalni format ukoliko kod dijeljenja ne želimo izgubiti decimalni dio rezultata. U gornjem izrazu to je postignuto umetanjem točke ('.') iza broja '`12`'. U suprotnom, bez umetanja točke, konačni rezultat bio bi '`3`' umjesto '`3.75`'.

3.2.5. Operator 'instanceof'

Operator ***instanceof*** zahtijeva objekt ili polje kao lijevi operand, te ime tipa s kojim se vrši usporedba kao desni operand. Izraz vraća logičku vrijednost ***true*** ukoliko je lijevi operand (objekt ili polje) instanca navedenog tipa, u suprotnom vraća logičku vrijednost ***false***. Ukoliko je lijevi operand `null`, operator *instanceof* uvijek vraća *false.* Ukoliko operator *instanceof* vrati *true*, to znači da lijevi operand može poprimiti vrijednost tipa desnog operanda.

Operator *instanceof* može se koristiti samo s objektima i tipova referenci, a ne s primitivnim tipovima i vrijednostima. Primjeri korištenja operatora:

```
//True – svi stringovi su instance klase String
"string" instanceof String

//True – stringovi su također instance klase Object
"" instanceof Object

//False – null nikad nije instanca nijedne klase
null instanceof String

Object o = new int[] {1,2,3};
o instanceof int[]    //True – polje je tipa int
o instanceof byte[]   //False – polje nije tipa byte
o instanceof Object   //True – sva polja su instance tipa Object

//Korištenje instanceof operatora kao provjera prije operacije
//"cast"
if (object instanceof Point) {
   Point p = (Point) object;
}
```

3.2.6. Posebni operatori

Java ima nekoliko jezičnih konstrukcija koje se mogu interpretirati kao operatori ili kao dio osnovne jezične sintakse. Ti "operatori" također su uključeni u tablicu 3.4. gdje se vide njihova svojstva kao što je prednost izvođenja u odnosu na druge operatore.

3.2.6.1. Dohvaćanje članova objekata (.)

Objekt je kolekcija (engl. *collection*) podataka i metoda koje koriste te podatke. Podaci i

metode objekta nazivaju se članovi (engl. *members*) objekta. Operator točka (.) služi za dohvaćanje tih članova. Na primjer, ako **`objekt`** predstavlja referencu nekog objekta, a **`varijabla`** predstavlja člansku varijablu tog objekta, onda je vrijednost članske varijable moguće dohvatiti pomoću operatora točke:

```
objekt.varijabla
```

Ukoliko je **`metoda`** ime metode koju sadrži objekt **`objekt`**, izraz

```
objekt.metoda
```

omogućava pozivanje metode s tim na je ulazne parametre nužno predati unutar oblih zagrada ( ) ukoliko ih ima.

3.2.6.2. Kreiranje objekata (`new`)

U Javi se objekti i polja kreiraju pomoću **`new`** operatora, nakon čega slijedi tip objekta koji se želi kreirati, te lista ulaznih parametra konstruktora (engl. *constructor*) tog objekta. Konstruktor je posebna metoda objekta koja služi za kreiranje, odnosno instanciranje, novih objekata te klase. Sintaksa za kreiranje objekata vrlo je slična sintaksi za pozivanje metode. Na primjer, instanciranje objekta koji predstavlja listu obavlja se na ovaj način (konstruktor na prima nikakve parametre):

```
List lista = new ArrayList();
```

dok bi kreiranje objekta za točku izgledalo ovako (konstruktor prima dva argumenta koji određuju koordinate novokreirane točke):

```
Point tocka = new Point(1, 2);
```

3.2.6.3. Konvertiranje tipova ili operacija cast (`()`)

Kao što je već opisano prije, oble zagrade također se koriste kao operator konverzije za suženjem, ili *cast* operacija. Prvi operand tog operatora je tip podatka u koji se pretvara i on se nalazi unutar oblih zagrada. Drugi operand je vrijednost koja se pretvara u taj tip i nalazi se iza oblih zagrada. Evo nekoliko primjera:

```
byte novaVrijednost = (byte) 28; //konverzija integera u byte
//konverzija decimalnog u cijeli broj
int cijeliBroj = (int) (x + 3.14f)
//konverzija objekta u string
String objektKaoString = (String) objekt;
```

3.3. Programske strukture u Javi

Java podržava niz programskih struktura koje su vrlo slične ili iste onim strukturama koje se koriste u programskom jeziku C. U taj skup spadaju uvjetne petlje **`if`** i **`switch`**, petlje **`do-while`**, **`while`** i **`for`**, naredbe **`break`**, **`continue`** i **`return`**, te korištenje labela. Za navedene naredbe vrijede ista pravila kao i kod programskog jezika C pa se neće posebno opisivati i razmatrati, već će se naglašavati bitne specifičnosti programskog jezika Java koje će se koristiti u izradi zadatka za ovu vježbu i koje su različite od onih u programskom jeziku C.

3.3.1. Petlja for/in

Petlja **for/in** je korisna i moćna petlja koja je uvedena u jezik Java od inačice 5.0. Pomoću nje se obavlja iteriranje (prolazak) po elementima polja (engl. *array*) ili zbirke (engl. *collection*). Prilikom svake iteracije dohvaća jedan element polja ili zbirke, te izvršava tijelo petlje s tim elementom. Petlja *for/in* obavlja iteraciju bez brojača ili iteratora pa programer ne mora razmišljati o inicijalizaciji ili završetku petlje.

Petlja *for/in* koristi se pomoću ključne riječi *for* iza koje slijede zagrade unutar kojih se nalaze deklaracije varijabli (bez inicijalizacije), te izraz koji označava samo polje ili zbirku:

```
for ( deklaracija : izraz )
     tijelo petlje
```

Iako se u nazivu petlja *for/in* koristi riječ *"in"*, prilikom korištenja petlja ne spominje se ključna riječ *"in"*. Da bi se izbjegle eventualne zamjene s "običnom" *for* petljom, uvedena je ta notacija. Često se kod čitanja petlje umjesto znaka ":" izgovara "in", odnosno "u", pa za petlju **`for(int broj : skup)`** možemo pročitati "za svaki broj u skupu čini..". Osim toga petlja *for/in* često se naziva "proširena for petlja" (engl. *enhanced for*) ili petlja "za-svaki" (engl. *foreach*).

Na primjer, ako želimo ispisati članove polja prim brojeva, to možemo učiniti na ovaj način:

```
// Brojevi koje želimo ispisati
int[] poljePrimBrojeva = new int[] { 2, 3, 5, 7, 11, 13, 17, 19,
23, 29 };
// petlja za ispisivanje tih brojeva
for(int n : poljePrimBrojeva)
   System.out.println(n);
```

3.3.2. Metode

Kao i u programskom jeziku C, metoda predstavlja imenovani niz naredbi koji može pozvati iz drugog programskog koda u Javi. Metoda prima ulazne vrijednosti koje se još nazivaju i parametri (engl. *parameters*), te može vratiti neku vrijednost kao rezultat izvođenja metode.

Definicija metode sastoji se od imena metode, definicije ulaznih parametara (broj, poredak, tip i imena parametara), tipa povratne vrijednosti, iznimaka koje metoda može baciti (engl. *throw*), te atributa koji dodatno određuju svojstva metode (koji će biti detaljno objašnjeni u sljedećim laboratorijskim vježbama). Definicija metode u Javi izgleda ovako:

```
atributi tip ime (lista parametara) [iznimke]
```

Svaka definicija metoda može se, ali ni ne mora sastojati od atributa kao što su **`public`**, **`static`** ili **`abstract`**. U nastavku je dana lista mogućih atributa metode i njihova značenja:

3.3.2.1. Atribut "abstract"

Metoda označena atributom **`abstract`** predstavlja specifikaciju metode bez implementacije. Vitičaste zagrade ({ }) unutar kojih bi se nalazilo tijelo samo metode u tom su slučaju zamijenjene znakom točka-zarez (';'). Pripadajuća klasa koja sadržava apstraktnu metodu također mora biti apstraktna. Takve klase su nepotpune i ne mogu biti instancirane.

3.3.2.2. Atribut "final"

Metoda označena atributom **`final`** ne može biti nadjačana (engl. *overridden*) u klasi koja nasljeđuje tu klasu. Sve **`private`** metode su implicitno također **`final`**, kao i sve metode unutar **`final`** klase.

3.3.2.3. Atribut "native"

Metoda označena atributom **`native`** govori da je implementacija metode napisana u nekom drugom programskom jeziku kao što je npr. C, te je programskom jeziku Java predana pomoću vanjskog resursa. Isto kao i **`abstract`** metode, **`native`** metoda nema implementacije.

3.3.2.4. Atributi "public", "protected" i "private"

Ovo su atributi koji definiraju razinu pristupa (engl. *access*) i time određuju način na koji se može pristupiti metodi izvan klase u kojoj je definirana. Detaljna objašnjenja ovih atributa nalaze se u narednim poglavljima.

3.3.2.5. Atribut "static"

Metoda označena atributom **`static`** može se pozivati iz klase bez potrebe njenog instanciranja.

3.3.2.6. Atribut "strictfp"

Metoda označena atributom **`strictfp`** koristi precizniju aritmetiku decimalnih brojeva (engl. ***`floating-point`***) .

3.3.2.7. Atribut "synchronized"

Atribut **`synchronized`** omogućava sigurno izvođenje metode u višenitnim sustavima (engl. *threadsafe*).

Nakon definicije metode slijedi njena implementacija, odnosno tijelo metode koje se sastoji od niza naredbi. Primjer osnovne metode za pokretanje Java programa dan je u nastavku:

```
/**
 * Prima polje Stringova kao ulazni parametar, nema povratne
 * vrijednosti i ispisuje pozdravnu poruku na ekran. Pozdravna
 * poruka sastoji se od prefiksa "Hello" te prvog String elementa
 * u predanom polju. Ukoliko je polje Stringova prazno, ispisati
 * defaultnu poruku "Hello world!".
 */

public static void main(String args[]) {
   if (args.length > 0) {
      System.out.println("Hello " + args[0] + "!");
   }
   else {
      System.out.println("Hello world!");
   }
}
```

4. KLASE I OBJEKTI U JAVI

4.1. Uvod u klase i objekte

Klasa (engl. *class*) je imenovana skupina polja (engl. *fields*), odnosno varijabli, koje sadržavaju podatke, te metoda (funkcija) koje koriste te podatke. Svaka klasa definira novi tip podatka. Na primjer, moguće je definirati klasu **`Point`** koja definira točku u koordinatnom sustavu. Klasa bi sadržavala dva polja tipa **`double`** koja bi označavala vrijednost koordinata X i Y, te metode za manipuliranje i korištenje točke. Jedna od mogućih definicija klase **`Point`** izgledala bi ovako:

```
/**
 * Klasa koja predstavlja točku u koordinatnom sustavu
 */

public class Point {

   //koordinate točke, odnosno, polja x i y
   public double x,y;

   /**
    * Konstruktor klase koji inicijalizira vrijednosti
    * koordinata točke
    */
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   /**
    * Metoda koja izračunava udaljenost točke od ishodišta
    */
   public double distanceFromOrigin() {
      return Math.sqrt(x*x + y*y);
   }
}
```

Kao što je prethodno opisano, prije kreiranja klase unutar razvojnog okruženja Eclipse SDK potrebno je kreirati paket u kojem će se klasa nalaziti (ukoliko već ne postoji), te nakon toga kreirati samu klasu. Nakon uspješnog kreiranja klase **`Point`** stvorena je datoteka **`Point.java`**, a nakon kompajliranja klase (što se u Eclipse SDK obavlja automatski aktiviranjem opcije spremanja datoteke *Save*) nastaje datoteka **`Point.class`**. Eclipse SDK razvojno okruženje u pogledu *Package explorer* radi preglednosti ne prikazuje datoteke sa "**`.class`**" ekstenzijom, međutim, mogu se pronaći na tvrdom disku gdje je kreirana lokacija za sam Java projekt, unutar podmape "bin".

Nakon definiranja klase, moguće je kreirati instancu, odnosno, objekt te klase. Kao što je opisano u prvoj laboratorijskoj vježbi, za kreiranje objekta klase koristi se operator **`new`**. Za kreiranje objekta klase **`Point`** potrebno je konstruktoru klase predati dva argumenta koji određuju koordinate točke na sljedeći način:

```
//kreiranje objekta klase Point koja se nalazi na koordinatama
//X= 2.0, Y=3.5
Point tocka = new Point(2.0, 3.5);
```

Isto tako, ukoliko je potrebno kreirati neki objekt klase koja je sastavni dio Java API sučelja (npr. klase **`Date`** za sadržavanje podataka o nekom datumu), postupak je identičan:

```
Date trenutniDatum = new Date();
```

Osim korištenja operatora **`new`**, što je najčešći slučaj, postoji još nekoliko načina za kreiranje objekata (kao što su pozivanje metoda **`getInstance()`** ili **`newInstance()`**) koji ovise o implementaciji same klase koja se instancira.

Nakon kreiranja objekta, moguće je dohvaćati polja samog objekta koja sadržavaju neke informacije i koristiti te informacije u određene svrhe. Svaki objekt sadržava svoje kopije polja koja su definirana u samoj klasi i uz pomoć operatora '.' može dohvatiti stanje (vrijednosti) polja. Polja objekata dohvaćaju se sljedećom sintaksom:

```
//kreiranje objekta
Point tocka = new Point(2.0, 3.0);

//dohvaćanje vrijednosti polja x iz objekta 'tocka' i spremanje te
//vrijednosti u lokalnu varijablu 'x'
double x = tocka.x;

//dodjeljivanje nove vrijednosti polju 'y' objekta
//'tocka'
tocka.y = p.x * p.x;
```

```
//pozivanje metode 'distanceFromOrigin' objekta 'tocka'
//čime se izračunava udaljenost točke od ishodišta i spremanje
//izračunate vrijednosti u lokalnu varijablu 'd'
double d = tocka.distanceFromOrigin();
```

4.2. Definiranje klase

Kod osnovnog načina definiranja klase kao u prošlom poglavlju potrebna je ključna riječ class prije koje se navodi modifikator (npr. public), te nekoliko polja i metoda unutar vitičastih zagrada ({ }).

Ukoliko klasa nasljeđuje neku drugu klasu, navodi se ključna riječ "extends" nakon koje se navodi naziv klase koja se nasljeđuje. Osim nasljeđivanja neke druge klase, moguća je implementacija sučelja pomoću ključne riječi "implements" nakon koje se navode sučelja (može ih biti više za razliku od klasa koje se nasljeđuju) odvojena zarezima ','. Na primjer, definicija klase cjelobrojnih brojeva Integer izgleda ovako:

```
public class Integer extends Number implements Serializable {
   //članovi klase (polja i metode) u tijelu same klase
}
```

Deklaracija klase prije ključne riječi class može sadržavati nijedan, jedan ili više navedenih modifikatora:

4.2.1. public

Klasa označena ovim modifikatorom vidljiva je ostalim klasama koje se nalaze izvan paketa u kojem je ta klasa definirana

4.2.2. abstract

Klasa označena ovim modifikatorom je apstraktna klasa čija je implementacija nepotpuna zbog čega se ne može izravno instancirati (ne mogu se kreirati objekti), već je predviđena za nasljeđivanje

4.2.3. final

Modifikator označava klasu koja nije predviđena za nasljeđivanje

4.2.4. strictfp

Ukoliko je klasa označena ovim modifikatorom, sve njene metode ponašaju se kao metode koje koriste posebnu aritmetiku decimalnih brojeva (koristi se vrlo rijetko)

Iako je moguće korištenje više navedenih modifikatora odjednom, klasa ne može istovremeno biti final i abstract.

4.3. Polja i metode (članovi) klase

Postoje dva tipa polja i metoda (koji se jednom riječju nazivaju članovi): **članovi klase** koji su vezani sa samom klasom i **članovi instance** koji su vezani pojedine instance (objekte) klase. Zbog toga postoje četiri vrste članova: **polja klase**, **metode klase**, **polja instance** i **metode instance**. U sljedećem primjeru klase prikazana su sve četiri vrste članova:

```
public class Circle {

   //polje klase koje predstavlja korisnu konstantu PI
   public static final double PI = 3.14159;

   /**
    * Pretvara radijane u stupnjeve. Radi se o metodi klase.
    */
   public static double radiansToDegrees(double radians) {
      return radians * 180 / PI;
   }

   //polje instance koje sadržava vrijednost polumjera kružnice
   public double r;

   /**
    * Služi za izračunavanje površine kružnice. Radi se o metodi
    * instance.
    */
   public double area() {
      return PI * r * r;
   }
}
```

Prema gornjem primjeru može se primijetiti da **članovi klase koriste modifikator** `static` i mogu se koristiti bez potrebe instanciranja same klase navođenjem imena klase `Circle` i korištenjem operatora '.':

```
//dohvaćanje i upotreba vrijednosti konstante PI definirane kao
//polje metode unutar klase Circle
double piNaKvadrat = Circle.PI * Circle.PI;

//korištenje metode radiansToDegrees iz klase Circle
double stupnjevi = Circle.radiansToDegrees(2.34);
```

Za razliku od članova klasa, **članovi instance** mogu se koristiti samo kod instanci (objekata) klase i njihove definicije **ne sadržavaju modifikator** `static`:

```
//instanciranje dva različita objekta
Circle c = new Circle();
Circle d = new Circle();

//postavljanje polja 'r' (polumjera kružnice) instance 'c' na
//vrijednost '2.0'
```

```
c.r = 2.0;

//postavljanje polja 'r' (polumjera kružnice) instance 'd' na
//dvostruku vrijednost od 'c.r'
d.r = 2 * c.r;

//pozivanje metoda instanci pomoću kojih se izračunavaju površine
//pojedine kružnice (objekti 'c' i 'd')
double povrsinaKruzniceC = c.area();
double povrsinaKruzniceD = d.area();
```

Pristup sa članovima instance koristi se ukoliko je potrebno koristiti informacije koje su pohranjene u samom objektu (stanje objekta), međutim, ukoliko je potrebno koristiti članove koji nisu vezani uz stanje pojedinog objekta (neke konstante ili općenite metode), koristi se pristup sa članovima klase.

4.3.1. Ključna riječ 'this'

Iako metoda instance **`area()`** ne prima nijedan ulazni parametar, ipak za izračun površine kružnice koristi iznos polumjera te kružnice. Zbog toga je potrebno prije pozivanja metode postaviti parametar '**`r`**' instance klase **`Circle`** na neku vrijednost. U metodi **`area()`** koja je definirana unutar klase **`Circle`**, navedena je formula za izračunavanje površine kružnice **`PI * r * r`** koja koristi parametar '**`r`**'. Bez eksplicitnog označavanja u ovom slučaju jasno da je riječ o polju instance '**`r`**', a ne nekoj drugoj varijabli. Međutim, u nekim slučajevima moramo posebno naglasiti da želimo koristiti upravo polje instance, a to je moguće pomoću ključne riječi "**`this`**":

```
public void setRadius(double r) {
  //dodjeljivanje argumenta r (ulazni parametar metode) polju
  //instance pomoću ključne riječi 'this' (nije moguće napisati
  //samo 'r = r')
  this.r = r;
}
```

4.3.2. Modifikatori polja

Polja također kod deklaracije mogu imati sljedeće modifikatore:

4.3.2.1. **public**, **protected**, **private**

Ovi modifikatori određuju da li se polje može koristiti izvan klase gdje je definirano i gdje se može koristiti.

4.3.2.2. static

Ukoliko je naveden, ovaj identifikator označava da je polje vezano uz samu klasu gdje je definirano (polje klase).

4.3.2.3. final

Ovaj modifikator označava polje čija vrijednost se nakon inicijalizacije ne može promijeniti.

4.3.2.4. transient

Ovaj modifikator označava polje koje nije dio perzistentnog stanja serijaliziranog (pohranjenog) objekta.

4.3.2.5. volatile

Ovaj modifikator označava polje koje se pouzdano može koristiti u višedretvenom okruženju.

Neki primjeri inicijalizacije polja s opisanim modifikatorima dani su u nastavku:

```
int x = 1;
private String name;
public static final DAYS_PER_WEEK = 7;
String[] daynames = new String[DAYS_PER_WEEK];
private int a = 17, b = 37, c = 53;
```

4.4. Kreiranje i inicijalizacija objekata

Svaka klasa u Javi sadržava barem jedan konstruktor (engl. *constructor*), odnosno, metodu koja ima isto ime kao i sama klasa, te obavlja potrebnu inicijalizaciju kod kreiranja novog objekta klase. Ukoliko se eksplicitno ne navede konstruktor, koristi se podrazumijevani (engl. *default*) konstruktor koji ne prima nikakve parametre i ne izvodi nikakvu posebnu inicijalizaciju.

Sljedeći primjer prikazuje definiciju **Circle** klase u kojoj se koristi konstruktor koji prima informaciju u polumjeru kružnice:

```
public class Circle {
   //definiranje konstante PI
   public static final double PI = 3.14159;

   //polje instance koja sadržava informaciju o polumjeru kružnice
   public double r;

   /**
    * Kreira objekt klase kojem je definiran polumjer kružnice.
    */
   public Circle(double r) { this.r = r; }

   /**
    * Služi za za računanje površine kružnice. Radi se o metodi
    * instance.
    */
   public double area() { return PI * r*r; }

}
```

S novodefiniranim konstruktorom instanciranje objekata izgledalo bi ovako:

```
Circle kruznica = new Circle(2.5);
```

U nekim složenijim klasa uobičajeno je definirati više različitih konstruktora koji se pozivaju u ovisnosti o okolnostima u kojima se koriste. U tom slučaju svi konstruktori imali bi isti naziv, međutim, razlikovali bi se po ulaznim parametrima.

Osim izvršavanja logike unutar konstruktora, prilikom kreiranja novih objekata potrebno je i inicijalizirati polja klase (npr. u prošlom primjeru klase **Circle** to je potrebno učiniti s konstantom **PI**). Java kompajler automatski generira inicijalizaciju polja klase i izvodi je u sklopu konstruktora onim redoslijedom kako je navedena u samoj klasi, ali prije prve naredbe unutar konstruktora te klase. Na primjer, inicijalizacija polja klase u sljedećem programskom kodu:

```
public class TestClass {

  public int len = 10;
  public int[] table = new int[len];

  public TestClass() {
    for(int i = 0; i < len; i++) table[i] = i;
  }
}
```

obavila bi se kao da su inicijalizacijske naredbe umetnute u sam konstruktor:

```
public TestClass() {
```

```
  len = 10;
  table = new int[len];
  for(int i = 0; i < len; i++) table[i] = i;
}
```

Osim inicijalizacije koja se obavlja automatski, moguće je eksplicitno definirati inicijalizacijski blok koji se još naziva statički inicijalizator (engl. *static initializer*). Statički inicijalizator je konstrukcija u programskom kodu koja se sastoji od ključne riječi static i vitičastih zagrada unutar kojih se nalazi blok naredbi za inicijalizaciju polja:

```
/**
 * Klasa koja sadržava statički inicijalizator
 */
public class TrigCircle {

  //polja klase koja se koriste u statičkom inicijalizatoru
  private static final int NUMPTS = 500;
  private static double sines[] = new double[NUMPTS];
  private static double cosines[] = new double[NUMPTS];

  //statički inicijalizator koji popunjava polja klase
  static {
    double x = 0.0;
    double delta_x = (Circle.PI/2)/(NUMPTS-1);
    for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
      sines[i] = Math.sin(x);
      cosines[i] = Math.cos(x);
    }
  }
}
```

Statičkim inicijalizatorom omogućena je inicijalizacija polja (varijabli) koja su zajednička za sve objekte te klase i takva inicijalizacija se obavlja samo jednom.

5. NASLJEĐIVANJE U JAVI

5.1. Podklase i nasljeđivanje

Klasa **`Circle`** opisana u prethodnom poglavlju jednostavna je klasa koja predstavlja kružnicu i pomoću koje je moguće definirati više kružnica koje se razlikuju samo po njihovim polumjerima. Međutim, ukoliko je potrebno kreirati kružnicu koja je osim s polumjerom određena i ishodištem kružnice u koordinatnom sustavu, potrebno je definirati novu klasu koja se zove npr. **`PlaneCircle`**, ali koja i dalje poprima svojstva klase **`Circle`**. U tom slučaju bi klasa **`PlaneCircle`** bila **podklasa** (engl. *subclass*) klase **`Circle`**, odnosno, klasa **`PlaneCircle`** nasljeđuje sve metode i polja svoje **nadklase** (engl. *superclass*) **`Circle`**. Funkcionalnost zadržavanja svojstva od neke klase i dodavanje novih svojstava je ključan princip objektno orijentiranih programskih jezika kao što je Java.

Sljedeći primjer prikazuje klasu **`PlaneCircle`** kao podklasu klase **`Circle`**:

```
public class PlaneCircle extends Circle {
  //metode i polja klase 'Circle' automatski se nasljeđuju, tako
  //da je u ovoj klasi potrebno navesti samo nove metode i polja
  //nova polja klase koja će sadržavati koordinate ishodišta
  //kružnice
  public double cx, cy;

  /**
   * Novi konstruktor za inicijalizaciju novih polja u klasi
   * koristi se posebna sintaksa 'super' za pozivanje konstruktora
   * iz svoje nadklase 'Circle'
   */
  public PlaneCircle(double r, double x, double y) {
    super(r);       //pozivanje instruktora nadklase 'Circle'
    this.cx = x;    //inicijalizacija polja instance cx
    this.cy = y;    //inicijalizacija polja instance cy
  }

  /**
   * Metoda 'area' naslijeđena je iz nadklase 'Circle'
   * nova metoda instance provjerava je li točka unutar ili izvan
   * kružnice
   */
  public boolean isInside(double x, double y) {
    double dx = x - cx, dy = y - cy;//udaljenost od ishodišta
    double distance = Math.sqrt(dx*dx + dy*dy); //Pitagorin poučak
    return (distance < r);//vraćanje true ili false
  }
}
```

Ključna riječ **extends** u definiciji metode označava da klasa **`PlaneCircle`** nasljeđuje klasu **`Circle`** i samim time nasljeđuje sva njena polja i metode. Metoda klase **`isInside`** zorno prikazuje nasljeđivanje polja '**`r`**' – polje je u klasi **`PlaneCircle`** nije definirano, već je naslijeđeno iz nadklase **`Circle`**. Isto tako je moguće pozvati metodu **`area`** iz klase

`PlaneCircle` iako ona u njoj nije definirana.

Osim inicijalizacije vrijednosti polja `cx` i `cy`, konstruktor klase `PlaneCircle` poziva i konstruktor svoje nadklase `Circle` kako bi se inicijalizirala naslijeđena polja iz te klase. To je postignuto korištenjem metode `super(r)` koja prima iste parametre koje prima i konstruktor klase `Circle`. Metodu `super()` moguće je koristiti samo kao prvu naredbu konstruktora i isključivo unutar konstruktora podklase.

Još jedno vrlo važno svojstvo kod nasljeđivanja je to što je svaki `PlaneCircle` objekt isto tako i objekt klase `Circle`. Ako referenca objekta `pc` referencira objekt tipa `PlaneCircle`, moguće ga je dodijeliti klasi `Circle` i time "zaboraviti" na sva nova svojstva klase `PlaneCircle` (podataka i ishodištu i metode za izračunavanje udaljenosti od ishodišta). Ova tehnika se često naziva i *upcasting* jer od objekta konkretnije klase `PlaneCircle` dobivamo objekt općenitije klase `Circle` koju klasa `PlaneCircle` nasljeđuje:

```
//instanciranje klase 'PlaneCircle'
PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0);

//dodjeljivanje objekta 'pc' klasi 'Circle' bez cast operacije
Circle c = pc;
```

Dodjeljivanje objekta klase `PlaneCircle` objektu klase `Circle` kao u prošlom primjeru može se obaviti bez potrebe korištenja operacije *cast*. Objekt pohranjen u varijabli `c` i dalje je validan (ispravan) objekt klase `PlaneCircle`, međutim, kompajler to "ne može znati" pa je kod suprotne konverzije nužno korištenje operatora *cast*:

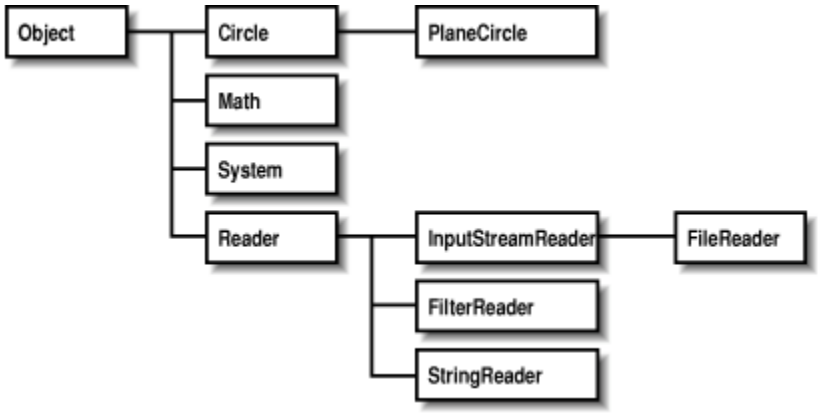
```
PlaneCircle pc2 = (PlaneCircle) c;
boolean inside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

Osim klasa koje se mogu nasljeđivati, postoje i klase koje se ne mogu nasljeđivati. Takve klase u svojoj deklaraciji imaju modifikator `final`. Na primjer, klasa `String` jedna je od klasa koje se ne mogu nasljeđivati:

```
public final class String { ... }
```

Svaka klasa ima svoju nadklasu. Ukoliko se u definiciji nove klase ne navede nadklasa pomoću riječi "**`extends`**", podrazumijevana nadklasa je **`java.lang.Object`** - jedina klasa koja nema nadklasu i sve je ostale klase u Javi nasljeđuju.

Kako sve klase u Javi (osim klase **`java.lang.Object`**) imaju svoju nadklasu, moguće ih je hijerarhijski prikazati počevši od klase **`Object`**. Na primjer, sljedeća slika prikazuje hijerarhiju klasa **`Circle`** i **`PlaneCircle`**, te još nekih u Javi:



Slika 3.1. Primjer dijagrama hijerarhije klasa

Ukoliko je u klasu **`PlaneCircle`** potrebno uvesti polje koje se naziva '**`r`**' kao i polje za polumjer koje je naslijeđeno iz klase **`Circle`**, tada dolazi do **sakrivanja** (engl. *hiding*) **polja iz nadklase**. Iako nije poželjno polja u podklasi nazivati imenima polja iz nadklase radi izbjegavanja dvosmislenih situacija, programski jezik Java to ipak omogućava. Vrijednosti polja u tom slučaju moguće je dohvatiti na sljedeće načine unutar klase **`PlaneCircle`**:

```
//referenciranje polja 'r' podklase PlaneCircle
r

//referenciranje polja 'r' podklase PlaneCircle pomoću ključne
//riječi 'this'
this.r

//referenciranje polja 'r' nadklase Circle pomoću ključne riječi
//'super'
super.r
```

Postoji još jedan način za dohvaćanje polja klase **`Circle`** unutar klase **`PlaneCircle`**, pomoću operacije *cast*:

```
//referenciranje polja 'r' nadklase Circle unutar podklase
//PlaneCircle
((Circle) this).r
```

Ukoliko se unutar podklase **`PlaneCircle`** definira metoda instance koja ima jednako ime, povratni tip i prima parametre identične metodi koja je definirana u nadklasi **`Circle`** (npr. metoda **`area()`** opisana u prethodnom poglavlju), to se zove **nadjačavanje** (engl.

overriding) **metode iz nadklase**. U tom slučaju se pozivanjem nadjačane metode iz objekta klase **`PlaneCircle`** poziva upravo ta novodefinirana metoda u podklasi, a ne više istoimena metoda u nadklasi **`Circle`**.

Nadjačavanje metoda nije moguće ukoliko su one metode klase (u definiciji sadržavaju modifikator **`static`**), već moraju isključivo biti metode instance.

Ponekad u programu nije moguće precizno odrediti koja metoda instance će se pozvati prije samog izvođenja programa (npr. metoda nadklase ili nadjačana metoda u podklasi), već se to može odrediti tek kod samog izvođenja programskog koda. U tom slučaju dolazi do **dinamičkog određivanja metode** (engl. *dynamic method lookup*) koju je potrebno pozvati **prilikom samog izvođenja** (engl. *runtime*) programskog koda. Na primjer, kad Java interpreter u nekom trenutku izvršava naredbu **`o.area()`** prvo mora odrediti tip podatka koji referencira varijabla **`o`** (objekt klase **`Circle`** ili **`PlaneCircle`**) i tek onda pozvati odgovarajuću metodu **`area()`**.

Slično kao i dohvaćanje polja iz nadklase, moguće je i pozivati nadjačane metode iz nadklase pomoću ključne riječi '**`super`**':

```
class A {
  int i = 1;
  int f() { return i; }
}

class B extends A {
  int i;

  //nadjačavanje metode 'f()'
  int f() {

    //korištenje polja 'i'
    i = super.i + 1;

    //pozivanje metode 'f()' iz nadklase 'A'
    return super.f() + i;
  }
}
```

5.2. Sakrivanje podataka i enkapsulacija

Jedna od ključnih objektno-orijentiranih tehnika je **sakrivanje podataka** unutar klase i mogućnost dohvaćanja tih podataka pomoću metoda. Ta tehnika zove se **enkapsulacija** (engl. *encapsulation*) zbog toga što su podaci zatvoreni unutar klase i mogu se dohvatiti isključivo na način (pomoću metoda) koji je odredio autor te klase. Time je omogućeno sakrivanje detalja same implementacije klase, neovisnost programskog koda koji koristi tu klasu o samoj implementaciji, te lakše razumijevanje klase kao cjeline.

Sva polja i metode uvijek se mogu koristiti unutar klase u kojoj su definirani, međutim, kad se koriste izvan te klase Java definira **pravila za kontrolu pristupa** (engl. *access control rules*) koja ograničavaju načine korištenja članova klase izvan klase u kojoj su definirani. U dosadašnjim laboratorijskim vježbama često se koristila ključna riječ **`public`**, a osim nje još postoje **`protected`** i **`private`**, te zajedno čine **modifikatore za kontrolu pristupa** (engl. *access control modifiers*) koji definiraju pravila pristupa članovima klase.

Osim unutar klase u kojoj su definirani, članovi klase mogu se koristiti i unutar cijelog paketa (engl. *package*) u kojem je ta klasa definirana. To znači da klase stavljene u isti paket "vjeruju" jedna drugoj, odnosno, imaju pristup detaljima implementacije druge klase unutar istog paketa. Ta pretpostavljena (engl. *default*) razina pristupa često se naziva i **razina pristupa unutar paketa** (engl. *package access level*) i čini jednu od četiri moguće razine pristupa, a preostale tri definirane su pomoću modifikatora **`public`**, **`protected`** i **`private`**. Primjer korištenja tih modifikatora prikazan je u sljedećem primjeru:

```
public class Laundromat {//Klasa koju mogu koristiti svi
  //'private' polje koje se ne može izravno koristiti
  private Laundry[] dirty;
  //'public' metode koje mogu koristiti svi
  public void wash() { ... }
  public void dry() { ... }
  //'protected' polje koje se može mijenjati u podklasama koje
  //nasljeđuju ovu
  protected int temperature;
}
```

Pravila pristupa određuju se na sljedeći način:

- Ukoliko je član klase deklariran pomoću modifikatora "**public**", znači da ga je moguće dohvatiti svugdje gdje je moguće pristupiti klasi u kojoj je definiran. Ovaj tip kontrole pristupa najmanje je restriktivan, odnosno, postavlja najmanja ograničenja na korištenje članova koji su deklarirani pomoću modifikatora "public".

- Ukoliko je član klase deklariran pomoću modifikatora "**private**", moguće ga je dohvatiti samo unutar klase u kojoj je definiran. Ovaj tip kontrole pristupa najviše je restriktivan od svih ostalih načina.

- Ukoliko je član klase deklariran pomoću modifikatora "**protected**", moguće ga je

dohvatiti unutar svih klasa koje su definirane unutar paketa s klasom u kojoj je definiran taj član (isto kao i razina pristupa unutar paketa). Osim toga, moguće ga je dohvatiti i unutar svih podklasa koje nasljeđuju klasu kojoj je definiran taj član, bez obzira u kojim paketima se nalazile te podklase. Ovaj tip kontrole je više restriktivan od modifikatora "public", međutim, manje restriktivan od modifikatora "private".

- Ukoliko prilikom deklaracije člana nije naveden nijedan od modifikatora "public", protected ili private, član posjeduje pretpostavljenu razinu pristupa: dohvatljiv je iz svih klasa koje su definirane unutar paketa u kojem je definirana klasa s tim članom, međutim, ne može se dohvatiti izvan tog paketa.

Sljedeća tablica prikazuje pravila o pristupanju članovima klase:

Tablica 5-1. Dostupnost članova klase

	Vidljivost članova klase			
Dostupno	**Public**	**Protected**	**Package**	**Private**
Klasa u kojoj je član klase definiran	Da	Da	Da	Da
Klasa u istom paketu	Da	Da	Da	Ne
Podklasa u drugom paketu	Da	Da	Ne	Ne
Neovisna klasa u drugom paketu	Da	Ne	Ne	Ne

Nakon navođenja pravila o pristupanju članovima klase slijede jednostavna pravila kad je poželjno i potrebno koristiti pojedini modifikator:

- Modifikator **`public`** upotrebljava se samo kod metoda i konstanti koje su dio javnog sučelja (engl. *application programming interface - API*) klase. Neka važna ili vrlo često korištena polja također mogu biti **`public`**, međutim, uobičajena praksa je da polja nisu **`public`**, već metode s kojima se dohvaća vrijednost tih polja.

- Modifikator **`protected`** koristi se za polja i metode koja nisu nužna programerima kod korištenja same klase, ali su potrebna u slučaju nasljeđivanja te klase koja se nalazi u drugom paketu.

- Podrazumijevana (engl. *default*) dostupnost koristi se kod polja i metoda koja su dio implementacijskih detalja, ali ih također koriste klase u istom paketu.

- Modifikator **`private`** koristi se kod polja i metode koje se koriste isključivo unutar klase i moraju biti sakriveni od preostalog programskog koda.

U primjeru klase **`Circle`** polje za pohranjivanje polumjera kružnice **`r`** deklarirano je modifikatorom **`public`**. Osim što je zbog toga moguće pristupiti polju **`r`** s bilo kojeg mjesta, moguće je postaviti i bilo koju vrijednost za polumjer, pa i negativnu vrijednost koja nema smisla. Da bi se to onemogućilo, moraju se implementirati dvije **`public`** metode za neizravno postavljanje i dohvaćanje vrijednosti polumjera (**`getRadius()`** i

`setRadius()`), a samo polje `r` mora biti deklarirano modifikatorom `protected` da bi mu se izravno moglo pristupiti iz podklasa koje nasljeđuju klasu `Circle`.

Promijenjena implementacija klase `Circle` izgledala bi ovako:

```
//Određivanje paketa u kojem će se nalaziti klasa
package shapes;

/**
 * Klasa je i dalje 'public'
 */
public class Circle {

  //Vrlo korisna konstanta koja se često koristi pa je 'public'
  public static final double PI = 3.14159;

  //Polumjer je sakriven, ali ga mogu vidjeti podklase koje
  //nasljeđuju ovu klasu
  protected double r;

  /**
   * Provjerava vrijednost polumjera (ne smije biti
   * negativan). Metoda je 'protected' da bi je mogle koristiti
   * podklase
   */
  protected void checkRadius(double radius) {
    if (radius < 0.0)
      throw new IllegalArgumentException("radius may not be " +
          "negative.");
  }

  /**
   * Konstruktor kojemu je dodana metoda za provjeravanje
   * predznaka polumjera 'checkRadius'.
   */
  public Circle(double r) {
    checkRadius(r);
    this.r = r;
  }

  /**
   * 'public' metoda za dohvaćanje vrijednosti polumjera.
   */
  public double getRadius() {
    return r;
  }

  /**
   * 'public' metoda za postavljanje vrijednosti polumjera koja
   * koristi metodu za provjeravanje pozitivnosti polumjera
   * 'checkRadius'.
```

```
   */
  public void setRadius(double r) {
    checkRadius(r);
    this.r = r;
  }

  /**
   * Služi za izračunavanje površine kružnice.
   */
  public double area() {
    return PI * r * r;
  }
}
```

Kako je polje za označavanje polumjera kružnice **`protected`**, sve klase unutar paketa **`shapes`** imaju izravan pristup tom polju. Metoda **`checkRadius`** je također **`protected`** i onemogućava postavljanje negativne vrijednosti za polumjer, a moguće ju je nadjačati u podklasama i time eventualno proširiti kontrole same vrijednosti.

5.3. Apstraktne klase i metode

U prošlom primjeru opisana je klasa **`Circle`** koja se nalazi unutar paketa **`shapes`**. Osim te klase moguće je implementirati i još neke klase koje označavaju oblike kao što su pravokutnik, kvadrat, elipsa, trokut i sl. Svaka klasa bi također imala metodu za izračunavanje površine oblika **`area()`** kao i klasa **`Circle`** koja označava kružnicu. Kako bi omogućili korištenje različitih oblika unutar neke strukture podataka kao što je npr. polje (koje će biti detaljnije opisano u sljedećim laboratorijskim vježbama), moramo definirati zajedničku nadklasu koja bi se zvala **`Shape`**. Tada bi svaki oblik (pravokutnik, kvadrat, elipsa, trokut i sl.) mogao koristiti sva polja i varijable klase **`Shape`** i metode koje implementira ta klasa. Kod definiranja klase **`Shape`** potrebno je voditi računa o zajedničkim obilježjima svih oblika i pokušati ih što smislenije ugraditi u klase na način kako bi nasljeđivanje takve nadklase bilo što lakše i razumljivije. Međutim, klasa **`Shape`** ne predstavlja određeni oblik pa ne može implementirati nikakve korisne metode. Pristup kod kojeg se ne implementiraju metode u Javi zove se korištenje *apstraktnih metoda* (engl. *abstract methods*).

Java dopušta definiranje metoda bez njenih implementacija pomoću modifikatora **`abstract`**. Apstraktna metoda nema tijela (engl. *method body*), već jednostavno sadržava potpis metode iza koje slijedi znak "točka-zarez" (';').

Postoji nekoliko vrlo važnih pravila za apstraktne metode i klase koje sadržavaju takve klase:

- Svaka klasa koja sadržava apstraktnu metodu automatski je apstraktna klasa i mora se deklarirati na taj način (navođenjem modifikatora **`abstract`** prilikom definiranja klase).

- Apstraktne klase ne mogu se instancirati (ne mogu se kreirati objekti iz tih klasa).

- Podklasa apstraktne klase može se instancirati isključivo ukoliko nadjačava i implementira (definira tijelo metode) svaku apstraktnu metode iz apstraktne klase koju nasljeđuje. Takva klase se tada često naziva i *konkretna podklasa* (engl. *concrete subclass*) kako bi se naglasilo da ta klasa nije apstraktna.

- Ukoliko neka podklasa ne implementira sve apstraktne metode apstraktne klase koju nasljeđuje, ona je i dalje apstraktna klasa i mora se deklarirati na taj način.

- Klase s modifikatorima metoda **`static`**, **`private`** i **`final`** ne mogu biti apstraktne zbog toga što metode s tim modifikatorima ne mogu biti nadjačane u podklasi. Isto tako ni klase označene modifikatorom **`final`** ne mogu sadržavati apstraktne metode.

- Klasa također može biti deklarirana kao apstraktna ukoliko nema nijedne apstraktne metode. Takav način definiranja klase označava da je implementacija nepotpuna i služi kad nadklasa za jednu ili više podklasa koje dovršavaju implementaciju. Takva vrsta klase također ne može biti instancirana.

Poštujući ova pravila, klasa **`Shape`** može biti definirana tako da sadržava apstraktnu metodu za izračunavanje površine oblika (**`abstract area()`**) . Time svaka klasa koja nasljeđuje klasu **`Shape`** mora sadržavati implementaciju te metode da bi se mogla instancirati (površina različitih oblika izračunava se na različite načine pa je i logično da svaki konkretni oblik ima svoju metodu za izračunavanje površine). Na kraju bi takva implementacija mogla izgledati ovako:

```
public abstract class Shape {
  //apstraktne metode – umjesto tijela metode nalazi se znak ";"
  public abstract double area();
}
```

```
public class Circle extends Shape {
  public static final double PI = 3.14159265358979323846;
  //podaci instance
  protected double r;
  //konstruktor
  public Circle(double r) { this.r = r; }
  //metode za dohvaćanje
 public double getRadius() { return r; }
  //implementacije apstraktnih metoda
  public double area() { return PI*r*r; }
}

public class Rectangle extends Shape {
  //podaci instance
  protected double w, h;

  //konstruktor
  public Rectangle(double w, double h) {
    this.w = w;
    this.h = h;
  }
  //metode za dohvaćanje vrijednosti
  public double getWidth() { return w; }
  public double getHeight() { return h; }
  //implementacije apstraktnih metoda
  public double area() { return w*h; }
```

Svaka definicija apstraktne metode u klasi **`Shape`** ima znak ';' (točka-zarez) koji slijedi odmah nakon oblih zagrada ('()'). Takve metode nemaju vitičaste zagrade, što znači da nemaju ni tijela metode. Koristeći prethodno definirane klase moguće je napisati sljedeći programski kod:

```
// Kreiranje polja s objektima Shape klase
Shape[] shapes = new Shape[3];

// Popunjavanje polja
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

double total_area = 0;

//izračunavanje površine svakog objekta (pozivaju se konkretne
//metode iz klasa Circle i Rectangle)
for(int i = 0; i < shapes.length; i++)
    total_area += shapes[i].area();
```

Vrlo je važno je uočiti da se pojedinom članu polja mogu dodijeliti i podklase klase **`Shape`** i da nije potrebna operacija *cast*-anja. To je još jedan primjer konverzije tipa proširenjem koja je detaljnije prikazan u prvoj laboratorijskoj vježbi.

Isto tako je moguće pozvati metodu **`area()`** nad svakim objektom klase **`Shape`**, usprkos tome što je to apstraktna klasa i ne definira tijelo te metode. Prilikom toga koristi se dinamičko povezivanje i određivanje metode koja se poziva pa se tako zapravo pozivaju metode **`area()`** objekata tipa **`Circle`** i **`Rectangle`** koji se nalaze u tom polju.

5.4. Sučelja

Poput klase, sučelje (engl. *interface*) također definira novi tip reference. Međutim, glavna razlika je u tome što sučelje nikad ne sadržava implementaciju tipova koje definira, odnosno, sve metode sučelja su apstraktne i ne sadržavaju tijela metoda.

Nadalje, nije moguće izravno instanciranje sučelja i kreiranje objekta tipa sučelja. Umjesto toga, klasa može implementirati (engl. *implement*) sučelje čime se naglašava da će se implementacije (tijela) metoda koje se nalaze u tom sučelju nalaziti unutar tijela same klase.

Sučelja programskom jeziku Java daju ograničenu ali vrlo moćnu alternativu *višestrukom nasljeđivanju* (koja se specifična za programski jezik C++, međutim, unosi i veliku razinu kompleksnosti u sam jezik). Klase u Javi mogu **nasljeđivati samo jednu nadklasu**, međutim, mogu **implementirati više sučelja** – to je **osnovna razlika između apstraktnih klasa i sučelja**.

Definicija sučelja vrlo je slična definiciji klase u kojoj su sve metode apstraktne i ključna riječ **`class`** zamijenjena je ključnom riječju **`interface`**. Na primjer, sljedeći programski kod prikazuje definiciju sučelja koje se naziva **`Centered`**.

```
public interface Centered {
    void setCenter(double x, double y);
    double getCenterX();
    double getCenterY();
}
```

Za članove sučelja (metode i polja) vrijedi nekoliko ograničenja:

- Sučelje ne sadržava nikakve implementacije metoda, odnosno, sve metode sučelja implicitno su apstraktne. Modifikator **`abstract`** je ipak dozvoljen, međutim, prema konvenciji se obično ne navodi. Kako **`static`** metode ne mogu biti apstraktne, ni metode sučelja ne mogu biti statičke.

- Sučelje je u pravilu javno (**`public`**) i svi njegovi članovi implicitno su označeni modifikatorom **`public`** pa je i u ovom slučaju nepotrebno navoditi taj modifikator. Neispravno je koristiti druge modifikatore (protected i private) i njihovo korištenje unutar sučelja prouzročiti će pogrešku u programu.

- Jedina polja koja su dozvoljena u sučelju su konstante koje moraju biti **`static`** i **`final`**. Nije dopušteno koristiti objekte kao polja unutar samog sučelja, jer ona

predstavljaju detalje same implementacije što ne spada u javno sučelje.

Sučelja se ne mogu instancirati (odnosno, iz njih se ne mogu kreirati objekti), pa nemaju ni konstruktor.

Sučelja mogu nasljeđivati druga sučelja pa isto kao i kod nasljeđivanja klasa, definicija sučelja može sadržavati ključnu riječ **`extends`**. Sučelje kod nasljeđivanja drugog sučelja nasljeđuje konstante i metode od *nadsučelja* (engl. *superinterface*) i može definirati nove konstantne i metode. Isto tako, sučelje može nasljeđivati više od jednog sučelja (za razliku od nasljeđivanja kod klasa). U nastavku su dani neki primjeri nasljeđivanja sučelja:

```
public interface Positionable extends Centered {
    void setUpperRightCorner(double x, double y);
    double getUpperRightX();
    double getUpperRightY();
}

public interface Transformable extends Scalable, Translatable,
Rotatable {}
```

Slično kao što je kod definicije klase prilikom nasljeđivanje nadklase potrebno koristiti ključnu riječ **`extends`**, tako je kod implementiranja jednog ili više sučelja potrebno koristiti ključnu riječ **`implements`**. Ukoliko klasa nasljeđuje više sučelja, iza ključne riječi **`implements`** moraju slijediti nazivi samih sučelja odvojeni zarezom. Klase koje ne implementiraju sve metode iz sučelja, implicitno su apstraktnog tipa i na taj način moraju biti deklarirane.

U sljedećem programskom kodu prikazan je način definiranja klase **`CenteredRectangle`** koja nasljeđuje klasu **`Rectangle`** (iz prethodnog dijela vježbe) i implementira sučelje **`Centered`**:

```
public class CenteredRectangle extends Rectangle implements
Centered {
  // Nova polja instance
  private double cx, cy;



// Konstruktor
public CenteredRectangle(double cx, double cy, double w, double h)
{
    super(w, h);
    this.cx = cx;
    this.cy = cy;
}

// Nasljeđuju se sve metode iz klase Rectangle, ali se trebaju
//implementirati i sve metode iz sučelja Centered
  public void setCenter(double x, double y) { cx = x; cy = y; }
  public double getCenterX() { return cx; }
```

```
  public double getCenterY() { return cy; }
}
```

5.5. Usporedba apstraktnih klasa i sučelja

Prilikom definiranja klase (npr. **`Shape`**) koja bi mogla imati puno podklasa (npr. **`Circle`**, **`Rectangle`**, **`Square`**) moguće je koristiti pristup s apstraktnim klasama ili pristup sa sučeljima zbog sličnih mogućnosti koje pružaju.

Sučelje je korisno zbog toga što ga mogu implementirati sve klase, čak i one koje nasljeđuju klasu koja nema nikakve veze s tim sučeljem. Međutim, **sučelje pruža samo specifikaciju, odnosno, ne implementira konkretne metode** pa klasa mora implementirati sve metode iz sučelja koja nasljeđuje.

Apstraktna klasa ne mora nužno biti u potpunosti apstraktna; **može sadržavati djelomičnu implementaciju metoda** koju podklase mogu iskoristiti. U nekim slučajevima brojne podklase mogu se u potpunosti osloniti na pretpostavljenu (engl. *default*) implementaciju metode koja se nalazi u apstraktnoj klasi. Međutim, klasa može nasljeđivati samo jednu nadklasu.

U nekim situacijama je lako razlučiti što je bolje koristiti, apstraktne klase ili sučelja. U ostalim slučajevima koriste se oba pristupa.

6. IZNIMKE

6.1. Uvod u rad s iznimkama

U nekim slučajevima izvođenje programa ne završi očekivano, već se zbog pogrešaka koje mogu nastati zbog propusta u kodu neplanirano prekida izvođenje programa. Na primjer, korisnik unese ime datoteke koja ne postoji ili sadrži neispravan sadržaj, što rezultira pogreškom prilikom čitanja podataka s konzole ili program pristupa zaštićenom dijelu memorije. Takvi slučajevi se u Javi nazivaju **uvjeti iznimke** (engl. *exception conditions*) i predstavljeni su pomoću objekata koji se nazivaju **iznimke**. Svaki objekt koji označava iznimku je izravno ili neizravno instanca klase koja nasljeđuje klasu **`java.lang.Throwable`**.

Klasa **`Throwable`** sadrži varijablu **`String`** koji se koristi za opisivanje same iznimke i predaje se kao argument konstruktora, a može se dohvatiti i uz pomoć **`getMessage`** metode.

Većina novih programera u programskom jeziku Java prvi put se susretne s iznimkama kad žele koristiti neku metodu za koju su saznali iz Java API dokumentacije. Na primjer, ukoliko se radi o metodi

```
file.getCanonicalFile();
```

kompajler javlja pogrešku "**`Unhandled exception type IOException`**". Detaljnijim proučavanjem API dokumentacije **`File`** klase moguće je primijetiti da deklaracija metode **`getCanonicalFile()`** uključuje riječi "**`throws IOException`**". Svaka metoda koja u svojoj deklaraciji sadrži ključnu riječ **`throws`** mora se pozivati u dijelu koda koji je pripremljen za **hvatanje** (engl. *catch*) iznimki ili prosljeđivanje iznimaka u ostale dijelove programa.

6.2. Hvatanje iznimki

Jedan način korištenja metoda koje bacaju iznimke je pomoću blokova **`try`**, **`catch`** i **`finally`**. U općenitom smislu korištenje tih blokova izgleda ovako:

```
try {
    //pozivanje metoda koje bacaju iznimke
} catch (tip_iznimke1 identifikator1) {
    //programski kod za analizu iznimke1
} catch (tip_iznimke2 identifikator2) {
    //programski kod za analizu iznimke2
...
} finally {
    //programski kod koji se obavlja nakon analize iznimaka
}
```

Ukoliko poziv neke metode unutar **`try`** bloka baci iznimku (što se nužno ne događa kod svakog poziva metode, već samo u određenim slučajevima), provjerava se svaki **`catch`** blok tako da se usporedi bačena iznimka s onom koju obrađuje pojedini **`catch`** blok. Na primjer, pri pozivu metode

```
File.createTempFile("test", "txt", "C:\\temp");
```

mogu dogoditi čak tri iznimke (**`IllegalArgumentException`**, **`IOException`** i **`SecurityException`**) pa se taj slučaj onda obrađuje na sljedeći način:

```
try {
    File file = File.createTempFile("test", "txt", "C:\\temp");
    System.out.println("Završetak 'try' bloka!");
} catch (IllegalArgumentException iznimka1) {
    iznimka1.printStackTrace();
} catch (IOException iznimka2) {
    iznimka2.printStackTrace();
} catch (SecurityException iznimka3) {
    iznimka3.printStackTrace();
} finally {
    //završne operacije
}
```

Ukoliko je poziv metode

```
File.createTempFile("test", "txt", "C:\\temp");
```

unutar **`try`** bloka prouzročio bacanje iznimke **`IllegalArgumentException`**, izvesti će se programski kod za obradu te iznimke, odnosno metoda

```
iznimka1.printStackTrace();
```

Metoda **`printStackTrace`** je također vrlo korisna metoda koja ispisuje sadržaj staze

stoga (engl. *stack trace*) Java virtualnog stroja (engl. *Java Virtual Machine*). **Staza stoga** sadrži vrlo precizan opis iznimke, odnosno sadržava liniju koda i klasu u kojoj se iznimka dogodila. S tim informacijama je ispravljanje pogrešaka (engl. *debugging*) u kodu znatno olakšano.

Ukoliko je poziv metode

```
File.createTempFile("test", "txt", "C:\\temp");
```

prouzročio bacanje iznimke **`IOException`**, poziva se metoda

```
iznimka2.printStackTrace();
```

unutar drugog **`catch`** bloka, a iznimka **`SecurityException`** poziva metodu

```
iznimka3.printStackTrace();
```

unutar trećeg **`catch`** bloka.

U slučaju kad se dogodi iznimka prilikom poziva metode

```
File.createTempFile("test", "txt", "C:\\temp");,
```

ispis poruke koji slijedi iza nje (**`System.out.println("Završetak 'try' bloka!");`**) se ne obavlja. Međutim, ukoliko se ne dogodi nijedna od iznimaka, **`try`** blok obavi se do kraja i nema pozivanja nijednog **`catch`** bloka.

Osim **`catch`** blokova, iza **`try`** bloka može se nalaziti **`finally`** blok koji sadržava programski kod koji se uvijek obavlja nakon završetka koda u **`try`** bloku. Programski kod unutar **`finally`** bloka obavlja se **uvijek**, bez obzira kako je završio **`try`** blok, bez obzira dogodila li se iznimka ili ne. Unutar njega se obično stavlja programski kod koji npr. zatvara datoteku, zatvara tok bajtova ili znakova, ili zatvara konekciju prema bazi podataka i sl.

Vrlo važno svojstvo kod hvatanja iznimki unutar **`catch`** blokova je poštivanje hijerarhije klasa koje definiraju iznimke. Naime, ukoliko se unutar **`try`** bloka baci iznimka npr. **`java.util.InputMismatchException`**, a **`catch`** blok hvata npr. iznimku **`java.lang.RuntimeException`**, izvesti će se kod unutar tog **`catch`** bloka zbog toga što je klasa **`InputMismatchException`** podklasa klase **`RuntimeException`**. Tako će se programski kod iz prve laboratorijske vježbe ukoliko metoda **`scanner.nextDouble()`** baci iznimku **`InputMismatchException`** izvesti na način da se iznimka obradi unutar **`catch`** bloka:

```
try {
     broj = scanner.nextDouble();
}
catch(InputMismatchException ex) {
     System.out.println("Došlo je do pogreške kod učitavanja " +
     "podatka!");
```

```
}
```

6.3. Deklariranje iznimaka

Osim korištenja **`try`**, **`catch`** i **`finally`** blokova postoji još jedan način korištenja metoda koje bacaju iznimke. Deklaracija metode može se sastojati od dijela koji govori koje sve iznimke mogu biti bačene unutar metode. U tom slučaju iza deklaracije metode slijedi ključna riječ **throws** te tip iznimke ili više tipova iznimaka odvojenih zarezom. U nastavku slijedi primjer deklaracije metode s ključnom riječi **`throws`**:

```
public File kreirajTempDatoteku() throws IllegalArgumentException,
IOException, SecurityException
{
   return File.createTempFile("test", "txt", "C:\\temp");
}
```

Ukoliko metoda deklarira mogućnost bacanja navedenih iznimaka, pozivi programski kod unutar te metode koji može baciti navedene iznimke ne mora biti unutar **`try`** bloka. Ključna riječ **`throws`** označava prosljeđivanje eventualnih iznimaka u ostale dijelove koda koji pozivaju tu metodu pa zbog toga nije potrebno obraditi iznimke unutar metode.

6.4. Vrste iznimaka

U Javi postoje dvije osnovne vrste iznimaka s kojima se programeri najčešće susreću: **označene iznimke** (engl. *checked exceptions*) i **iznimke kod izvođenja** (engl. *runtime exceptions*). Sve klase u Javi koje označavaju neku iznimku imaju duga imena koja opisuju namjenu te iznimke. Na primjer, klasa **`ArrayIndexOutOfBoundsException`** označava iznimku koja se baca ukoliko se dohvaća element izvan zadanih granica polja, a klasa **`FileNotFoundException`** označava iznimku koja se baca ukoliko se pokušava dohvatiti datoteka koja ne postoji.

Iznimka **`ArrayIndexOutOfBoundsException`**, koja se baca kad se dohvaća element izvan granica polja, prouzročena je isključivo od strane programera zbog nepažnje. Takve iznimke koje je moguće izbjeći pažljivijim programiranjem nazivaju se **iznimkama kod izvođenja**. Na primjer, iznimke **`ArithmeticException`** koja se baca kod dijeljena brojeva s nulom ili **`NegativeArraySizeException`** koja se baca kod kreiranja polja s negativnom veličinom, također spadaju u iznimke kod izvođenja.

Npr. pokušaj inicijalizacije polja s negativnom veličinom

```
char[] polje = new char[-234];
```

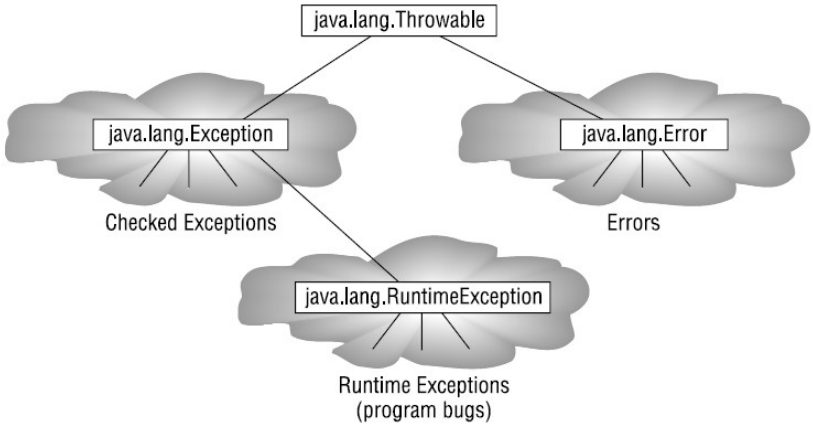
tek kod izvođenja bi prouzročilo iznimku:

```
java.lang.NegativeArraySizeException
            at
```

```
hr.tvz.programiranje.java.labosi.cetvrti.Test.main(Test.java:30)
Exception in thread "main"
```

Bacanje iznimki kod izvođenja mora se izbjegavati jer su one uvjetovane pogreškama programera. Kod njih nije nužno **postojanje `try` blokova koji ih obrađuju ili `throws` deklaracija** – po tome se znatno razlikuju od ostalih iznimaka i pravila koja su do sada opisana. Time je izbjegnuta izrazita nepreglednost programskog koda koja bi nastala ukoliko bi svaku iznimku kod izvođenja bilo potrebno obraditi pomoću `try` bloka ili `throws` deklaracije.

Sve iznimke kod izvođenja imaju jedno zajedničko svojstvo: **nasljeđuju istu klasu `java.lang.RuntimeException`**. Slika 6.1. prikazuje hijerarhiju iznimaka:



Slika 6.1. Hijerarhija iznimaka

Osim iznimaka kod izvođenja (engl. *runtime exceptions*) postoje i označene iznimke (engl. *checked exceptions*). Svaka iznimka koja izravno ili neizravno **ne nasljeđuje klasu `java.lang.RuntimeException`**, naziva se **označena iznimka**. Iznimke tog tipa moraju se obrađivati unutar **`try-catch-finally`** blokova ili **`throws`** deklaracijama kod metoda. Iznimke koje izravno ili neizravno nasljeđuju **`java.lang.Error`** klasu (desna strana slike 6.1.) također predstavljaju označene iznimke. Klasa **`java.lang.Error`** predstavlja iznimke koje se događaju uslijed ozbiljnih pogrešaka unutar JVM-a.

6.5. Bacanje iznimaka

Osim hvatanja i obrađivanja iznimaka koje bacaju metode iz Java API sučelja, moguće je napisati metode koje također bacaju iznimke. Na primjer, u slučaju kad je unutar metode potrebno baciti neku iznimku koja označava neko izvanredno stanje u programu (npr. **`IOException`**), to je moguće obaviti koristeći ključnu riječ **`throw`**:

```
IOException x = new IOException("Pogreška kod kreiranja" +
"datoteke.");
throw x;
```

Sve klase koje označavaju iznimke imaju tri vrste konstruktora, jedan koji ne prima nikakve

parametre, drugi koji prima **`String`** poruku koja detaljnije opisuje samu iznimku (kao u navedenom programskom kodu) i treći koji prima objekt iznimke koja je prouzročila tu drugu iznimku.. Uvijek se preporuča korištenje konstruktora koji prima poruku koja detaljnije opisuje iznimke zbog toga što je tu poruku moguće dohvatiti iz iznimke preko metode **`getMessage`** unutar nekog **`catch`** bloka koji hvata tu iznimku.

Ukoliko metoda baca neku označenu iznimku, ona se mora nalaziti unutar **`throws`** deklaracije ili se u **`throws`** deklaraciji mora nalazi tip iznimke koji nasljeđuje iznimka koja se baca unutar metode.

Ukoliko nijedna od već otprije definiranih iznimaka (koje je moguće naći u *javadoc* dokumentaciji Java API-a) ne zadovoljava korisničke potrebe, potrebno je kreirati novu vrstu iznimke. Kod kreiranja nove vrste iznimke, prvo je potrebno odlučiti kakve će vrste biti iznimka, označena ili iznimka kod izvođenja. Ukoliko je potrebno kreirati označenu iznimku, novokreirana klasa koja označava novu iznimku mora nasljeđivati klasu **`java.lang.Exception`** ili neku njenu podklasu, a iznimke kod izvođenja moraju nasljeđivati klasu **`java.lang.RuntimeException`** ili neku njenu podklasu.

Svaka klasa koja definira novu iznimku mora sadržavati naziv koji opisuje samu iznimku i koji završava nazivom **`Exception`**. Na primjer, ukoliko je potrebno kreirati iznimku koja se baca ukoliko korisnik prekorači maksimalno dozvoljeni broj pokušaja logiranja u neku aplikaciju, iznimka bi se mogla nazvati **`TooManyUnsuccessfulLoginAttemptsException`**.

Svaka novokreirana iznimka mora imati tri konstruktora. Prvi konstruktor mora primati **`String`** poruku koja pobliže opisuje iznimku, drugi mora primati objekt koji označava drugu iznimku koja je prouzročila pozivanje te iznimke, te treći konstruktor koji prima poruku i objekt koji označava uzročnu iznimku. Poruke koje se predaju u konstruktor iznimke mogu se dohvatiti pomoću **`getMessage`** metode, a uzročne iznimke koje se predaju u konstruktor mogu se dohvatiti pomoću **`getCause`** metode.

Na primjer, definiranje nove iznimke izgledalo bi ovako:

```
public class TooManyUnsuccessfulLoginAttemptsException
             extends Exception
{
   public TooManyUnsuccessfulLoginAttemptsException(
     String message)
   {
      super(message);
   }

   public TooManyUnsuccessfulLoginAttemptsException(
     Throwable cause)
   {
      super(cause);
   }

   public TooManyUnsuccessfulLoginAttemptsException(
```

```
      String message, Throwable cause)
   {
      super(message, cause);
   }
}
```

Nakon definiranja nove iznimke, bacanje iste bi moglo izgledati ovako:

```
throw new TooManyUnsuccessfulLoginAttemptsException("Korisnik" +
"Pero Perić!");
```

6.6. Ulančavanje iznimaka

Iznimke su ponekad prouzročene nekom drugom iznimkom, a ta druga iznimka je prouzročena opet nekom trećom iznimkom itd. Takva struktura naziva se ulančavanje iznimaka (engl. *exception chaining*).

Ulančavanje iznimaka je vrlo važno zbog podizanja razine apstrakcije i zbog dobivanja dodatnih informacija kod ispravljanja pogrešaka. Kad programer zna točan razlog pogreške koja se dogodila iz staze stoga (engl. *stack trace*), ispravljanje iste znatno mu je olakšano.

Zbog toga svaka klasa koja definira iznimku ima dva konstruktora (poglavlje 6.5.) koji primaju **uzročnu iznimku** (engl. *cause exception*). Isto tako sve iznimke imaju metodu **`initCause`** koju nasljeđuju iz klase **`Throwable`** pomoću koje također mogu postaviti uzročnu iznimku:

```
} catch (IOException e) {
    BadDataSetException bdse = new BadDataSetException();
    bdse.initCause(e);
    //ili korištenje konstruktora
    //BadDataSetException bdse = new BadDataSetException(e);
    throw bdse;
} finally {
    // ...
}
```

Poziv metode **`bdse.initCause(e);`** služi za spremanje uzročne iznimke koja programeru daje dodatne informacije koje mogu biti vrlo korisne kod ispravljanje pogrešaka.

6.7. Nadjačavanje kod iznimaka

Kad se nasljeđuje klasa i nadjačava (engl. *override*) metoda, Java kompajler inzistira da sve iznimke koje baca nova metoda moraju biti podklase ili jednake klasama iznimaka koje baca originalna metoda. Sljedeći primjer klasa prikazuje nekoliko ispravnih i neispravnih načina nasljeđivanja metoda koje bacaju iznimke:

```
public class OsnovnaKlasa {
     public void method() throws IOException {
     }
}

public class PrvaKlasa extends OsnovnaKlasa {
     public void method() throws IOException {
     }
}

public class DrugaKlasa extends OsnovnaKlasa {
     public void method() {
     }
}

public class TrecaKlasa extends OsnovnaKlasa {
     public void method() throws EOFException,
               MalformedURLException
     {
     }
}
```

```
public class CetvrtaKlasa extends OsnovnaKlasa {
     public void method() throws IOException,
                          IllegalAccessException
     {
     }
}

public class PetaKlasa extends OsnovnaKlasa {
     public void method() throws Exception {
     }
}
```

Metoda **`method()`** u klasi **`OsnovnaKlasa`** koju nasljeđuju ostale klase iz primjera baca iznimku **`IOException`**. Takva deklaracija omogućava da metode u podklasi koje nadjačavaju metodu iz nadklase bacaju **`IOException`** iznimke ili podklase iznimke **`IOException`**.

Prema tom pravilu metoda **`method`** iz klase **`PrvaKlasa`** ispravno nadjačava metodu iz nadklase koja baca iznimku **`IOException`**.

Klasa **`DrugaKlasa`** u kojoj se nalazi metoda **`method`** koja nadjačava metodu iz nadklase **`OsnovnaKlasa`** to radi na ispravan način, jer ne baca nikakve iznimke i samim time ne može baciti iznimku koja nije podklasa klase **`IOException`**.

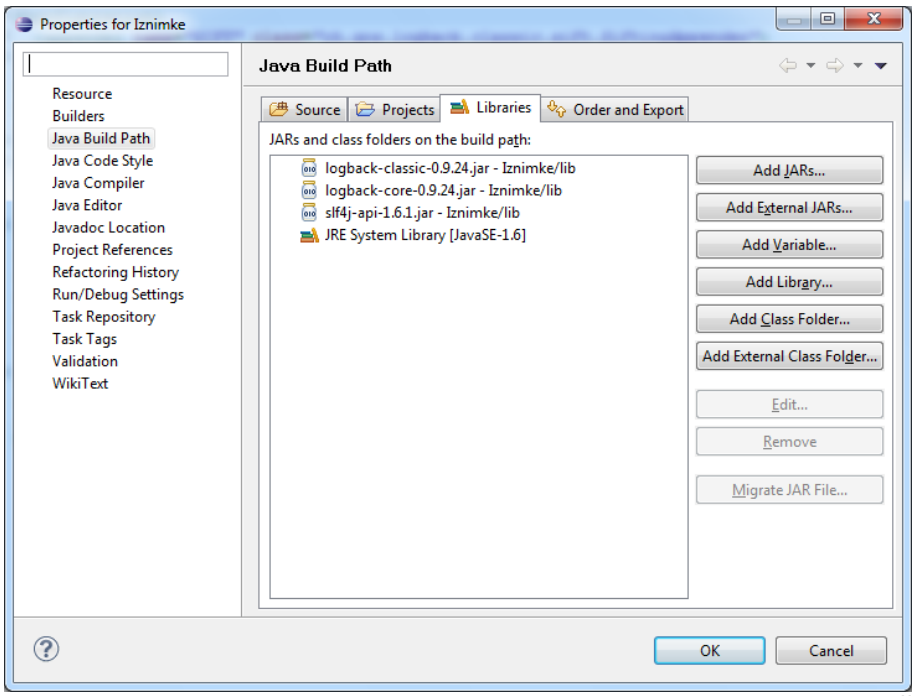
Metoda **`method`** iz klase **`TrecaKlasa`** također ispravno nadjačava metodu iz nadklase jer baca iznimke **`EOFException`** i **`MalformedURLException`** koje su podklase iznimke **`IOException`**.

Klase `CetvrtaKlasa` i `PetaKlasa` ne nadjačavaju metodu `method` na ispravan način zbog toga što iznimke koje bacaju nisu podklase klase `IOException`. `IllegalAccessException` je podklasa klase `Exception` i nije podklasa klase `IOException`, kao i `Exception` koja je zapravo nadklasa, a ne podklasa klase `IOException`.

6.8. Dodatna biblioteka za kreiranje logova – "Logback"

Gotovo svaka aplikacija ima potrebu za bilježenje svih događaja, pa ima ugrađeno vlastito praćenje aktivnosti koja se zapisuju u datoteke, što se vrlo često naziva **logiranje** (engl. *logging*). Programski jezik Java ima sustav za logiranje pod licencom otvorenog koda (engl. *open source license*) koji se naziva **Logback** i dolazi u obliku datoteka tipa Java-arhive (engl. *Java archive – JAR*) koje ima ekstenziju **.jar**. Pomoću datoteka tog tipa u aplikacije je moguće uključiti i koristiti programski kod koji je napisao neki drugi programer ili skupina programera. Njih je potrebno uključiti u **putanju klasa** (engl. *classpath*) kako bi razvojna okolina koristila klase koje se nalaze unutar *jar* datoteka.

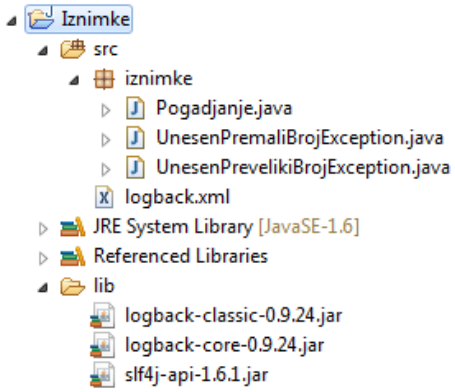
Za korištenje Logback-a potrebne su *jar* datoteka **`logback-classic-*.jar`**, **`logback-core-*.jar i slf4j-api-*.jar`**, pri čemu "**`*`**" označava inačicu samih datoteka (npr. "0.9.24" za "Logback" ili "1.6.1" za "slf4j" datoteke). Datoteke je moguće preuzeti s lokacije gdje se nalazi i sama laboratorijska vježba. Da bi se ta datoteka uključila u *classpath* aplikacije koja se razvija, *jar* datoteku potrebno je najprije spremiti na lokalni disk, a radi mogućnosti prenošenja Eclipse projekta na druga računala, poželjno je da se smjeste u mapu "lib" unutar samog projekta. Nakon toga je unutar razvojne okoline Eclipse SDK potrebno odabrati projekt u koji se nove klase žele uključiti i pritisnuti desnu tipku miša i izabrati opciju "**`Properties`**" ili odabrati opciju "Build Path -> Configure Build Path..." (slika 6.2.).



Slika 6.2. Konfiguriranje *classpath*-a na projektima unutar Eclipse razvojnog okruženja

Nakon odabiranja opcije "Java Build Path" s lijeve strane dijaloga sa slike 6.2. potrebno je pomoću odabrati opciju "Add JARs..." (ako se "jar" datoteke nalaze unutar mape "lib" Eclipse projekta, što se preporuča radi prenosivosti – prikazano na slici 6.3.), odnosno "Add External JARs..." opcije (ukoliko se "jar" datoteke nalaze izvan Eclipse radne okoline). Nakon uspješnog dodavanja "jar" datoteka u *classpath*, situacija s "Build Path" putanjom trebala bi izgledati kao na slici 6.2.

Logback biblioteka omogućava bilježenje događaja u konzolu ili "log" datoteke. Prema pretpostavljenim postavkama Logback zapisuje ispisuje u konzolu, a ukoliko je potrebno događaje u aplikaciji zapisati u datoteku, nužno je kreirati XML konfiguracijsku datoteku pod imenom "logback.xml" koja sadrži postavke za datoteke. Tu datoteku potrebno je kreirati korištenjem izbornika "File->New->Other..." i pronaći "XML File" datoteku i smjestiti je unutar "src" mape Java projekta unutar Eclipse razvojnog okruženja (slika 6.3.).



Slika 6.3. Izgled strukture Java projekta u Eclipse-u

Datoteku "logback.xml" potrebno je otvoriti i u nju ubaciti sljedeći sadržaj:

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <appender name="SIFT" class="ch.qos.logback.classic.sift.SiftingAppender">
    <discriminator>
      <Key>screen</Key>
      <DefaultValue>unknown</DefaultValue>
    </discriminator>
    <sift>
      <appender name="FILE-${screen}" class="ch.qos.logback.core.FileAppender">
        <File>./logs/${screen}.log</File>
        <Append>false</Append>
        <layout class="ch.qos.logback.classic.PatternLayout">
          <Pattern>%d [%thread] %level %mdc %logger{35} - %msg%n</Pattern>
        </layout>
      </appender>
    </sift>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="SIFT" />
  </root>
</configuration>
```

Logback ima tri osnovne komponente: komponente za logiranje (engl. *loggers*), komponente za proširivanje logova (engl. *appenders*) i komponente za definiranje izgleda log zapisa (engl. *layouts*).

`Logger` je objekt koji služi za zapisivanje logova tijekom korištenja programa u **`log`** datoteke. Unutar Java koda je taj objekt moguće dohvatiti pozivanjem statičke metode **`LogFactory.getLogger`** i poželjno ga je deklarirati **`private`** i **`static`** modifikatorima da bi bio dostupan samo unutar klase koja se logira (npr. **`Pogadjanje.class`**) i

dostupan unutar **main** metode:

```
private static final Logger logger = LoggerFactory.getLogger(Pogadjanje.class);
```

Logger mora biti iz paketa **org.slf4j.Logger**. Nakon inicijaliziranja **Logger** objekta pod nazivom **logger**, moguće ga je koristiti uz pomoć nekoliko metoda od kojih svaka označava određenu razinu logiranja: **trace()**, **debug()**, **info()**, **warn()** i **error()**. Ukoliko je potrebno npr. logirati događaje u aplikaciji na razini **debug()**, logovi će se popunjavati pomoću sljedeće metode:

```
logger.debug("Korisnik je pokrenuo aplikaciju!");
```

ili

```
logger.error(poruka, prevelikiException);
```

gdje varijabla "poruka" predstavlja tekst koji opisuje izvanredni događaj (iznimku) u aplikaciji, dok varijabla "prevelikiException" predstavlja iznimku koja je prouzročila pogrešku u programu. Ovdje je prikazan cijeli "catch" blok u kojem se koristi navedeno logiranje:

```
...
} catch (UnesenPrevelikiBrojException prevelikiException) {
     String poruka = "Unijeli ste preveliki broj!";
     System.out.println(poruka);
     logger.error(poruka, prevelikiException);
     brojPokusaja++;
}
...
```

Razine logiranja su poredane prema količini detalja koji se ispisuju u log datoteke: **TRACE < DEBUG < INFO < WARN < ERROR**. Ukoliko se npr. u programskom kodu logiraju zapisi razina **DEBUG** i **ERROR**, a u konfiguracijskoj datoteci je navedena osnovna kategorija (engl. *root category*) **ERROR**, u datoteke će se zapisivati samo zapisi razine **ERROR** jer su zapisi razine **DEBUG** zbog toga ignorirani. Međutim, to ne znači da je **DEBUG** zapise potrebno izbrisati iz koda, već ih samo ignorirati. Ukoliko se pokaže potreba za naknadnim logiranjem i svih **DEBUG** zapisati radi otkrivanja pogrešaka u programu, ona je potrebno promijeniti samo konfiguracijsku datoteku. To pravilo vrijedi i za sve ostale kombinacije – ključna informacija je razina logiranja u konfiguracijskoj datoteci ispod koje se sve razine ignoriraju.

Razina logiranja u već prikazanom sadržaju konfiguracijske datoteke "logback.xml" određuje se pomoću XML "*tag*"-a:

```
<root level="DEBUG">
```

Appender komponente definiraju se na sljedeći način:

```
<appender name="SIFT" class="ch.qos.logback.classic.sift.SiftingAppender">
```

```
  <discriminator>
    <Key>screen</Key>
    <DefaultValue>unknown</DefaultValue>
  </discriminator>
  <sift>
    <appender name="FILE-${screen}" class="ch.qos.logback.core.FileAppender">
      <File>./logs/${screen}.log</File>
      <Append>false</Append>
      <layout class="ch.qos.logback.classic.PatternLayout">
        <Pattern>%d [%thread] %level %mdc %logger{35} - %msg%n</Pattern>
      </layout>
    </appender>
  </sift>
</appender>
```

Appender nazvan imenom "SIFT" definira svojstvo logiranja kod kojeg se za svaki ekran u aplikaciji definira vlastita log datoteka. U složenijim aplikacijama vrlo je korisno razdvajati različite logove u zasebne datoteke kako bi pretraživanje logova i pogrešaka u aplikaciji bilo olakšano. Jedan način da se to postigne je organiziranje datoteka prema ekranima (funkcionalnostima) unutar aplikacije. To je moguće postići kreiranjem tzv. *discriminator* komponente unutar *appender* komponente koja definira parametar s kojim je moguće utjecati na naziv datoteke u koju se zapisuju događaji. U ovom slučaju se parametar koji utječe na naziv datoteke naziva "screen" i definiran je na sljedeći način:

```
<discriminator>
  <Key>screen</Key>
  <DefaultValue>unknown</DefaultValue>
</discriminator>
```

dok se unutar programskog koda Java može postaviti na sljedeći način:

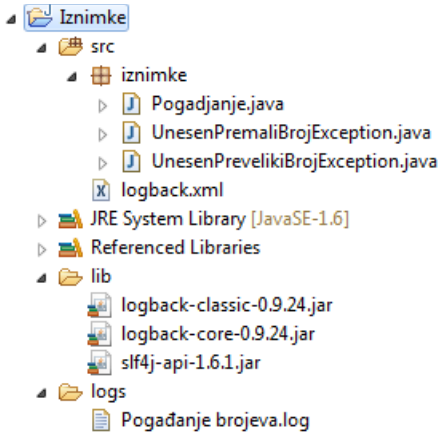
```
MDC.put("screen", "Pogađanje brojeva");
```

Lokacija na kojoj se kreiraju datoteke s logovima postavlja se na sljedeći način:

```
<File>./logs/${screen}.log</File>
```

čime se označava mapa "logs" unutar projekta u Eclipse-u.

Nakon pokretanja, korištenja i završetka rada s aplikacijom logovi ostanu zapisani u specificiranim datotekama unutar Java projekta u Eclipse-u. Međutim, često se dogodi da mapa i datoteke nisu odmah vidljive u Eclipse-u, zbog čega je potrebno odabrati opciju "Refresh" nad projektom unutar *Package Explorer*-a:



Slika 6.4. Java projekt u Eclipse-u s mapom "logs" i datotekom s logovima

Sadržaj datoteke "Pogađanje brojeva.log" sadrži sve zapise koji su zapisani pomoću **`logger`** objekta i izgleda slično ovome:

```
2010-10-08 18:26:36,207 [main] INFO screen=Pogađanje brojeva iznimke.Pogadjanje
- Korisnik je pokrenuo aplikaciju!
2010-10-08 18:26:36,254 [main] DEBUG screen=Pogađanje brojeva iznimke.Pogadjanje
- Aplikacije je generirala broj 87
2010-10-08 18:26:38,725 [main] ERROR screen=Pogađanje brojeva iznimke.Pogadjanje
- Unijeli ste premali broj!
iznimke.UnesenPremaliBrojException: Unesen je premali broj!
      at iznimke.Pogadjanje.provjeriBroj(Pogadjanje.java:91) ~[bin/:na]
      at iznimke.Pogadjanje.main(Pogadjanje.java:55) ~[bin/:na]
2010-10-08 18:26:42,334 [main] ERROR screen=Pogađanje brojeva iznimke.Pogadjanje
- Unijeli ste preveliki broj!
iznimke.UnesenPrevelikiBrojException: Unesen je preveliki broj!
      at iznimke.Pogadjanje.provjeriBroj(Pogadjanje.java:89) ~[bin/:na]
      at iznimke.Pogadjanje.main(Pogadjanje.java:55) ~[bin/:na]
2010-10-08 18:26:44,714 [main] ERROR screen=Pogađanje brojeva iznimke.Pogadjanje
- Unijeli ste premali broj!
iznimke.UnesenPremaliBrojException: Unesen je premali broj!
      at iznimke.Pogadjanje.provjeriBroj(Pogadjanje.java:91) ~[bin/:na]
      at iznimke.Pogadjanje.main(Pogadjanje.java:55) ~[bin/:na]
2010-10-08 18:26:46,651 [main] ERROR screen=Pogađanje brojeva iznimke.Pogadjanje
- Unijeli ste preveliki broj!
iznimke.UnesenPrevelikiBrojException: Unesen je preveliki broj!
      at iznimke.Pogadjanje.provjeriBroj(Pogadjanje.java:89) ~[bin/:na]
      at iznimke.Pogadjanje.main(Pogadjanje.java:55) ~[bin/:na]
2010-10-08 18:26:48,525 [main] ERROR screen=Pogađanje brojeva iznimke.Pogadjanje
- Unijeli ste premali broj!
iznimke.UnesenPremaliBrojException: Unesen je premali broj!
      at iznimke.Pogadjanje.provjeriBroj(Pogadjanje.java:91) ~[bin/:na]
      at iznimke.Pogadjanje.main(Pogadjanje.java:55) ~[bin/:na]
2010-10-08 18:26:49,777 [main] INFO screen=Pogađanje brojeva iznimke.Pogadjanje
- BRAVO! Pogodili ste broj 87, za što Vam je trebalo 6 pokušaja!
```

Priručnik za naprednije razumijevanje i korištenje moguće je pronaći na sljedećoj web stranici: http://logback.qos.ch/manual/index.html .

7. POLJA I DINAMIČKE STRUKTURE PODATAKA

7.1. Polja

U programskom jeziku Java polje (engl. *array)* predstavlja poseban tip **objekta** koji sadrži nula ili više vrijednosti primitivnog tipa, te nula ili više objekata (referenci). Te vrijednosti pohranjene su među elementima polja koji predstavljaju neimenovane varijable određene pozicijom ili indeksom (engl. *index*) unutar polja. Tip polja određen je tipom elemenata koji se u njemu nalaze i svi elementi unutar polja moraju biti tog tipa.

Elementi polja numerirani su indeksima počevši od nule pa sve do indeksa koji označava vrijednost ukupne veličine polja umanjenog za jedan (0..n-1 gdje n predstavlja dimenziju polja). Element polja s indeksom 1 zapravo označava drugi po redu član unutar polja. Broj elemenata u polju određen je podatkom **`length`** polja koji je određen kod kreiranja polja i nakon toga se više ne može promijeniti.

Polje može sadržavati elemente bilo kojeg tipa podatka iz Jave, a osim toga element polja može biti i drugo polje (polje unutar polja). To znači da Java podržava i višedimenzionalna polja.

7.1.1. Tipovi polja

Polja mogu biti primitivnih tipova i tipova referenci (kao objekti instance klasa). Da bi se definiralo polje potrebno je iza tipa navesti uglate zagrade ('[' i ']'). Na primjer, sljedeći programski kod definira tri polja različitih tipova:

```
byte b;                         // byte je primitivan tip
byte[] arrayOfBytes;            // byte[] je tip polja: polja byteova
byte[][] arrayOfArrayOfBytes;// byte[][] je drugi tip: polje koje
                                // sadrži polja byte-ova
String[] points;                // String[] polje String objekata
```

Polja se često koriste pri konverziji s proširenjem. Kao što je opisano u prethodnim poglavljima, polje može biti tipa **`Shape`**, a može sadržavati objekte koji nasljeđuju tu klasu:

```
// Kreiranje polja "shapes" tipa Shape
Shape[] shapes = new Shape[3];

// Popunjavanje polja objektima klase koji nasljeđuju klasu Shape
// (Circle i Rectangle)
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
double total_area = 0;

// izračunavanje površine svakog objekta (pozivaju se konkretne
```

```
// metode iz klasa Circle i Rectangle koje su određene dinamičkim
// povezivanjem za vrijeme izvođenja)
for(int i = 0; i < shapes.length; i++)
  total_area += shapes[i].area();
```

Unutar **`for`** petlje pozivaju se metode **`area()`** za izračunavanje površine svakog konkretnog oblika. Iako se naizgled pozivaju metode **`area()`** od objekata klase **`Shape`** (zbog toga što je samo polje tipa **`Shape`**), dinamičkim pozivanjem (engl. *dynamic linking*) za vrijeme izvođenja programa (engl. *runtime*) kompajler prije samog izvođenja metoda provjerava tipove objekata u polju i tek nakon toga poziva odgovarajuće metode.

U usporedbi s programskim jezikom C++, u Javi se uglate zagrade stavljaju iza tipa polja (**`byte[] arrayOfBytes;`**) dok se u C++-u uglate zagrade stavljaju iza varijable koja označava polje (**`byte arrayOfBytes[]`**). Međutim, radi kompatibilnosti i reduciranja nepotrebnih zabuna, u Javi je također omogućeno navođenje uglatih zagrada iza varijable koja označava naziv polja.

7.1.2. Kreiranje i inicijalizacija polja

Da bi se kreiralo polje u Javi, baš kao i kod kreiranja objekata koristi se ključna riječ **`new`**. Tipovi polja nemaju konstruktora, međutim, kod svakog kreiranja polja potrebno je navesti veličinu polja. Veličina polja definirana je nenegativnim cjelim brojem koji se navodi između uglatih zagrada:

```
// Kreiranje polja od 1024 byte-ova
byte[] buffer = new byte[1024];

// Kreiranje polja od 50 referenci tipa String
String[] lines = new String[50];
```

Nakon što je polje kreirano na navedeni način, svaki element polja inicijalizira se na vrijednost koja je podrazumijevana za taj tip podatka: **`false`** za **`boolean`** tipove, '\u0000' za elemente tipa **`char`**, 0 za elemente tipa **`int`**, 0.0 za elemente tipa **`float`**, te **`null`** za tipove referenci (instance klasa).

Ukoliko je polje potrebno kreirati i inicijalizirati u jednom izrazu, potrebno je iza uglatih zagrada (bez navođenja veličine polja unutar njih) navesti vitičaste zagrade unutar kojih se navodi lista izraza (ili samo vrijednosti zadanog tipa) odvojenih zarezom:

```
String[] greetings = new String[] { "Hello", "Hi", "Howdy" };
int[] smallPrimes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19 };
```

Osim navedenog načina inicijalizacija, polje je moguće inicijalizirati i bez korištenja ključne riječi '**`new`**':

```
String[] greetings = { "Hello", "Hi", "Howdy" };
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

Arhitektura Java virtualnog stroja (engl. *Java Virtual Machine*) ne podržava efikasnu inicijalizaciju polja, odnosno, polja se kreiraju u trenutku izvođenja programa, a ne za vrijeme njegovog kompajliranja. Zbog toga se izraz:

```
int[] perfectNumbers = {6, 28};
```

kompajlira u Java *bytecode* koji je ekvivalentan skupini izraza:

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

7.1.3. Korištenje polja

Elementi polja su varijable koje nemaju imena već ih određuju redni brojevi mjesta unutar polja. Elementi polja dohvaćaju se navođenjem uglatih zagrada ('[' i ']') unutar kojih se navodi redni broj elementa unutar polja. Sljedeći primjer programskog koda prikazuje postavljanje i dohvaćanje elemenata polja:

```
// Kreiranje polja s dva String objekta
String[] responses = new String[2];

// Postavljanje prvog člana polja
responses[0] = "Yes";

// Postavljanje drugog člana polja
responses[1] = "No";

// Dohvaćanje elemenata polja
System.out.println(question + " (" + responses[0] + "/" +
                   responses[1] + " ): ");
```

Izraz koji predstavlja indeks za dohvaćanje elemenata polja (npr. unutar uglatih zagrada moguće je napisati i izraz **`i + 1`**, gdje je "**`i`**" varijabla) mora biti primitivnog tipa **`int`** ili tipa koji može biti proširen do tipa **`int`**, **`byte`**, **`short`** ili **`char`**. Veličina polja **`length`** također je tipa **`int`**, a polje ne smije biti veće od konstante **`Integer.MAX_VALUE`**.

Prvi element polja "a" je **`a[0]`**, drugi element polja je **`a[1]`**, a posljednji element polja dohvaća se izrazom **`a[a.length - 1]`**. Za razliku od programskog jezika C++ gdje je za indekse elemenata polja moguće navesti negativne i indekse veće od veličine polja (prilikom čega ne dolazi do pogreške, odnosno baca se iznimka), Java ne dopušta dohvaćanje elemenata polja čiji indeks nije unutar granica samog polja. Ukoliko je indeks elementa koji se dohvaća premalen (negativan) ili prevelik (veći od **`a.length - 1`**), Java baca (engl. *throws*) iznimku **`ArrayIndexOutOfBoundsException`** čime završava izvršavanje programa.

Analiziranje sadržaja i obavljanje operacija nad elementima polja najčešće se obavlja pomoću **`for`** petlji. Na primjer, sljedeći programski kod "iterira" (engl. *iterates*) po svakom elementu i računa sumu svih elemenata polja:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
int sumOfPrimes = 0;
for(int i = 0; i < primes.length; i++)
    sumOfPrimes += primes[i];
```

Sva polja implementiraju sučelje **`Cloneable`** koje omogućava kopiranje sadržaja polja pozivanjem metode **`clone()`**. Općenito, objekti klasa koje implementiraju sučelje **`Cloneable`** mogu se kopirati korištenjem metode **`clone()`**. Pri tome je važno uočiti da je korištenjem sučelja **`Cloneable`** potrebno obaviti i *cast* operaciju (zbog toga što **`clone()`**

metoda vraća tip **`java.lang.Object`**, pa je potrebno specificirati koji tip se zapravo kopira – detaljnije informacije u *javadoc* dokumentaciji). Sljedeći programski kod prikazuje *kloniranje* (kopiranje) polja **`data`** i korištenje *cast* operacije:

```
int[] data = { 1, 2, 3 };
int[] copy = (int[]) data.clone();
```

Za razliku od kloniranja polja, kloniranje objekata (tipova referenci) kopira samo reference objekata, odnosno njihove adrese, a ne same objekte. To znači da kopirani objekt i dalje sadržava adresu od "originalnog" objekta pa se svaka promjena na kopiranom objektu odražava i na originalu. Takva kopija zove se *plitka* (engl. *shallow*) kopija.

Ponekad je potrebno samo kopirati elemente iz jednog u drugo polje. Za tu svrhu koristi se metoda **`System.arraycopy()`** (detalji funkcije nalaze se u *javadoc* dokumentaciji).

Osim klase **`System`**, postoji i klasa **`java.util.Arrays`** koja podržava niz korisnih statičkih metoda za rad s poljima. Npr. metode **`sort()`** i **`bynarySearch()`** su korisne za sortiranje i pretraživanje polja, metoda **`equals()`** omogućava usporedbu sadržaja dva polja, **`Arrays.toString()`** metoda koristi se ukoliko je sadržaj polja potrebno pretvoriti u **`String`** oblik (ukoliko je to potrebno kod *debuggiranja* programa). Detalji pojedine metode mogu se pronaći u *javadoc* dokumentaciji.

7.1.4. Višedimenzionalna polja

U slučaju kad su elementi polja neka druga polja, takva polja nazivaju se višedimenzionalnim (engl. *multidimensional*) poljima. Višedimenzionalna polja označavaju se s više parova uglatih zagrada, koji ovisi o samom broju dimenzija polja. Na primjer,

```
int[][] products;
```

definira dvodimenzionalno polje (svaki par uglatih zagrada definira jednu dimenziju) . Za dohvaćanje jednog elementa polja potrebno je definirati dva indeksa polja, po jedan za svaku dimenziju. Ukoliko je potrebno definirati novo višedimenzionalno polje, potrebno je koristiti ključnu riječ **`new`** i specificirati veličinu svake od dimenzija polja:

```
int[][] products = new int[10][10];
```

Ukoliko se prilikom definiranja polja ne navedu inicijalne vrijednosti elemenata, one su postavljenje na podrazumijevane vrijednosti (npr. za primitivni tip **`int`** to je vrijednost 0 itd.). Isto kao i kod jednodimenzionalnih polja, elementi višedimenzionalnih polja također mogu biti inicijalizirani pomoću statičkog inicijalizatora:

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

Potrebno je samo koristiti ugnježđene parove vitičastih zagrada unutar kojih je potrebno navesti vrijednosti pojedinih članova odvojenih zarezom.

7.2. Dinamičke strukture podataka

Osim polja, u Javi postoji niz drugih vrsta struktura podataka koje služe za pohranjivanje objekata koje se nazivaju **zbirke objekata** (engl. *object collections*) ili **spremnici objekata** (engl. *object containers*). Java programski okvir za zbirke (engl. *Java Collections Framework*) je skupina važnih i korisnih klasa i sučelja iz paketa **`java.util`** za rad sa zbirkama objekata. Programski okvir za zbirke definira dva fundamentalna tipa zbirki objekata.

Sučelje **`Collection`** predstavlja osnovno sučelje za sve zbirke objekata, odnosno, sve zbirke nasljeđuju to sučelje. Sadržava metode kao što su **`add`**, **`remove`**, **`size`**, **`toArray`** itd.

Postoji niz različitih vrsti zbirki među kojima su najvažnije četiri skupine: **`Set`**, **`List`**, **`Queue`** i **`Map`**.

`Set` je vrsta **`Collection`** zbirke koja ne može sadržavati duplikate objekata, ali objekti u zbirci ne moraju nužno biti sortirani. **`SortedSet`** je inačica **`Set`** zbirke kod koje objekti moraju biti sortirani.

`List` predstavlja zbirku čiji elementi zadržavaju određeni poredak, npr. poredak po kojem su dodavani u listu.

`Queue` je zbirka sa sortiranim elementima u skladu sa specifičnim poretkom. Svaki *queue* ima svoj glavni element (engl. *head element*) koji se koristi u operacijama poput **`peek`** i **`pool`**.

`Map` predstavlja zbirku objekata u kojoj se nalaze elementi koji su određeni parovima ključ-vrijednost (engl. *key-value*), pri čemu ne smiju postojati duplikati ključeva. Npr. ako postoji mapa u kojoj su pohranjeni objekti klase **`Osoba`** (koji sadrže podatke o osobama), a ključevi su objekti klase **`String`** koji označavaju JMBG osoba, onda se korištenjem određenom JMBG-a kao ključa može dohvatiti samo jedan objekt **`Osoba`** (ne postoje dva elementa s istim ključem – JMBG-om). Sučelje **`Map`** ne nasljeđuje sučelje **`Collection`**, međutim, imaju ista imena metoda koja se koriste za rukovanje elementima polja pa se i na mape može gledati isto kao na zbirke elemenata.

`SortedMap` je vrsta mape u kojoj su ključevi sortirani.

`Iterator` je sučelje koje služi za dohvaćanje objekata jednog po jednog iz neke zbirke.

`ListIterator` je iterator **`List`** objekata koji ima korisne metode za korištenje listi.

Programski okvir za zbirke u Javi postoji od inačice 1.2, dok su se prije te verzije koristile **`Vector`** i **`Hashtable`** zbirke objekata, koje su približno iste kao i zbirke **`ArrayList`** i **`HashMap`**.

Od Jave inačice 5 uvedeni su parametrizirani tipovi (engl. *parameterized types*) kao npr. **`List<String>`** čime se definira zbirka objekata tipa **`List`** u kojoj se nalaze isključivo objekti klase **`String`**. Time je otklonjen jedan od nedostataka Jave da programer mora znati kakvi objekti su spremljeni u zbirci da bi je uopće mogao koristiti. Međutim, radi zadržavanja kompatibilnosti s prethodnim inačicama Jave (engl. *backward compatibility*), kompajler tolerira i notaciju zbirki bez eksplicitnog navođenja tipova s pripadajućim upozorenjem kako bi programer bio svjestan da postoji način kako bi se unaprijedio programski kod.

U specifikacijama se vrlo često koristi notacija npr. **`java.util.List<E>`**, gdje zbirka parametriziranog tipa, u ovom slučaju lista, sadržava elemente tipa označenog oznakom **`E`**. Metode **`get()`** i **`add()`** u tom slučaju vraćaju i dodaju elemente tipa **`E`** u polje.
Kako bi se mogao koristiti ovakav generički tip (engl. *generic type*), potrebno je specificirati tip varijable kako bi se dobio parametrizirani tip kao što je npr. **`List<String>`**. U tom slučaju kompajler može obaviti bolju provjeru tipa čime je povećana sigurnost tipova (engl. *type safety*) cijelog programa. U takvu zbirku moguće je dodavati samo objekte tipa **`String`**.

Na primjer, sljedeći programski kod ne koristi parametrizirane tipove i krije potencijalne pogreške koje je moguće otkriti tek kod izvođenja programa (engl. *run-time errors*):

```
public static void main(String[] args) {
   //Ova lista je predviđena da sadrži samo objekte tipa String
   //Kompajler ne može znati koja je namjena ove zbirke (liste)
   List wordlist = new ArrayList();

   //Zabunom je u polje dodan polje Stringova (String[])
   //Kompajler ne može znati da je radi o pogrešci
   wordlist.add(args);
   //Metoda get() vraća objekt tipa Object kojem je pomoću cast
   //operacije potrebno dodijeliti ispravan tip. Kako je lista
   //predviđena da koristi objekte tipa String, pokušaj obavljanja
   //cast operacije rezultirati će iznimkom ClassCastException
   //zbog toga što je prvi član polja tipa  String[], a ne
   //očekivanog tipa String.
   String word = (String) wordlist.get(0);
}
```

Isti programski kod moguće je napisati korištenjem parametriziranih tipova, gdje se izbjegavaju te pogreške:

```
public static void main(String[] args) {
    // Ova lista može sadržavati samo objekte tipa String
    List<String> wordlist = new ArrayList<String>();

    // varijabla args je tipa String[], a ne String, pa kompajler
    // ne dopušta  dodavanje ovog tipa u listu koja može primati
    // samo objekte tipa String
    wordlist.add(args);  // Kompajlerska pogreška

    // Međutim, moguće je napraviti sljedeće:
    // Pomoću petlje for/in obavlja se iteriranje kroz polje
    // Stringova i spremanje pojedinog Stringa u listu (jednog po
    // jednog)
    for(String arg : args) wordlist.add(arg);

    // Ovdje nije potrebno koristiti cast operaciju, jer metoda
    // List<String>.get() parametrizirane liste uvijek vraća
    //objekt tipa String
    String word = wordlist.get(0);
}
```

Kao što metode mogu imati više ulaznih parametara, klase također mogu imati više tipova varijabli. Sučelju **`java.util.Map`**, koje predstavlja zbirke koje služe za spremanje objekata pod nekim ključem, kod parametriziranog tipa potrebno je definirati tip varijable koji predstavlja tip ključeva u mapi, te tip varijable koji predstavlja tip objekata koji se nalaze u samoj mapi. Na primjer, korištenje mape u kojoj su ključevi tipa **`String`**, a objekti u mapi tipa **`Integer`** izgledalo bi ovako:

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

```
    // Mapa Stringova iz polja args, te njihovih pozicija u polju
    // 'args'(Integer)
    Map<String,Integer> map = new HashMap<String,Integer>();

    // Dodavanje Stringova i objekata Integer u mapu. Metoda put()
    // kao drugi parametar prima tip int (primitivni tip), koji se
    // na kraju pretvori u tip Integer -> ta automatska pretvorba
    // naziva se "autoboxing"
    for(int i=0; i < args.length; i++) map.put(args[i], i);

    // Dohvaćanje pozicije Stringa "hello" unutar mape. Operacija
    // cast u ovom  slučaju također nije potrebna.
    Integer position = map.get("hello");

    // Osim automatske pretvorbe "autoboxing", moguća je i
    // automatska pretvorba "autounboxing" koja pretvara objekt
    // tipa Integer u tip int (primitivni tip). Ukoliko u mapi ne
    // postoji ključ "world" ili bilo koji drugi ključ koji se
    // dohvaća, događa se iznimka "NullPointerException"
    int pos = map.get("world");
}
```

Paket **`java.util`** osim navedenih zbirki sadržava i neke konkretne implementacije (za razliku od dosad navedenih sučelja koja se moraju implementirati da bi ih se moglo iskoristiti) navedenih sučelja koje pokrivaju veliku većinu svih potreba programera:

- **`HashSet`** je vrsta **`Set`** implementacije kao *hash-tablice* (engl. *hashtable*). Svaki objekt ima svoj *hash* kod (neki oblik kodiranog sažetka objekta) kojim je objekt određen i označava ključ koji se koristi unutar zbirki objekata. **`HashSet`** predstavlja općenitu implementaciju kod koje operacije pretraživanja, dodavanja i brisanja objekata iz zbirke nisu ovisna o veličini same zbirke.

- **`TreeSet`** predstavlja **`SortedSet`** implementaciju kao binarnog stabla. Sporije se pretražuje ili mijenja njen sadržaj nego u slučaju **`HashSet`**-a, međutim, elementi su sortirani.

- **`ArrayList`** je implementacija **`List`** sučelja pri čemu se koristi polje promjenjive veličine. Operacija brisanja elemenata s početka liste može biti vrlo zahtjevna ukoliko je lista vrlo velika, no operacije kreiranja i slučajnog pristupanja (engl. *radnom access*) elementima nisu zahtjevne.

- **`LinkedList`** je implementacija **`List`** i **`Queue`** sučelja kod koje operacija modificiranja nije zahtjevna za bilo koju veličinu liste, međutim, operacija slučajnog pristupanja je spora.

- **`HashMap`** je implementacija **`Map`** sučelja pomoću hash-tablice kod koje su operacije pregledavanja i dodavanja elemenata nezahtjevne.

- **`TreeMap`** je implementacija **`SortedMap`** sučelja kao binarnog stabla u kojem su elementi sortirani po ključu.

7.2.1. Sučelje Collection<E>

Kao što je već opisano, većina tipova zbirki objekata nasljeđuje sučelje **`Collection<E>`**, iznimka su samo podklase od klase **`Map<K,V>`**. U nastavku ovog poglavlja opisuju se najkorisnije i najvažnije metode sučelja **`Collection<E>`** koje je moguće pozvati prilikom rada sa svim zbirkama objekata (dokumentacija o ostalim metodama nalazi se u *javadoc*-u).

- **`public int size()`**

Vraća veličinu zbirke objekata, odnosno, broj objekata koje zbirka sadrži. Vrijednost koju funkcija vraća ograničena je veličinom **`Integer.MAX_VALUE`** i u slučajevima kad zbirka sadrži više objekata od tog maksimalnog broja.

- **`public boolean isEmpty()`**

Vraća **`true`** ukoliko zbirka ne sadržava nikakve objekte.

- **`public boolean contains(Object elem)`**

Vraća **`true`** ukoliko zbirka sadržava traženi objekt, odnosno, ukoliko zbirka sadržava element koji pozivanjem metode **`equals`** s elementom **`elem`** vraća **`true`**. Ukoliko je parametar **`elem`** jednak **`null`**, metoda **`contains`** vraća **`true`** u slučaju kad postoji **`null`** element unutar zbirke.

- **`public Iterator<E> iterator()`**

Vraća iterator pomoću kojega je moguće iterirati po objektima unutar zbirke.

- **`public Object[] toArray()`**

Vraća novo polje koje sadrži sve elemente iz zbirke.

- **`public boolean add(Object elem)`**

Osigurava da element **`elem`** postoji u zbirci i vraća **`true`** ukoliko je **`elem`** dodan u zbirku. Ako dotična zbirka dozvoljava duplikate, ova metoda uvijek vraća **`true`**. Ukoliko zbirka ne podržava duplikate, metoda **`add`** vraća **`false`** ako takav objekt već postoji u zbirci.

- **`public boolean remove(Object elem)`**

Briše jednu instancu elementa **`elem`** iz zbirke i vraća **`true`** ukoliko je obrisan element iz zbirke (ukoliko je element **`elem`** postojao u zbirci).

- **`public boolean addAll(Collection<E> collection)`**

Dodaje svaki element iz zbirke **`collection`** u dotičnu zbirku i vraća **`true`** ukoliko je došlo do izmjene u zbirci.

- **`public void clear()`**

Briše sve elemente iz zbirke.

7.2.2. Sučelje Set<E>

`Set<E>` je zbirka objekata koja ne dozvoljava duplikate: ne smije sadržavati dvije reference (adrese) na isti objekt, dvije reference na **`null`** ili reference na dva objekta **`a`** i **`b`** takva da izraz **`a.equals(b)`** vraća **`true`**. Većina općenitih implementacija **`Set<E>`** sučelja ne uvjetuje sortiranje objekata unutar seta, međutim, moguće je koristiti i sortirane implementacije setova (**`SortedSet<E>`** i **`LinkedHashSet<E>`**) koje koriste sortiranje.

`Set<E>` ne definira dodatne metode osim onih koje već postoje u sučelju **`Collection<E>`** (to su metode **`add()`** i **`addAll()`**), međutim, dodaje neka ograničenja na te metode. Metode **`add()`** i **`addAll()`** izričito provode pravila o sprečavanju duplikata u samoj zbirci objekata: ukoliko se pokušava dodati novi objekt koji već postoji u zbirci, metode taj element neće dodati u zbirku, odnosno metode će u tom slučaju vratiti **`false`** (umjesto vrijednosti **`true`** koju vrate kad se novi objekt uspješno doda u set). **`SortedSet<E>`** je sučelje koje nasljeđuje zbirku **`Set<E>`** i sadržava logiku koja vraća elemente u određenom redoslijedu. Redoslijed je definiran **`compareTo`** metodom iz sučelja **`Comparable<T>`**, a ukoliko je potrebno kreirati specifične uvjete uspoređivanja, moguće je definirati objekt klase **`java.util.Comparator<T>`** koji sadrži kriterije uspoređivanja.

`SortedSet<E>` sučelju **`Set<E>`** dodaje još neke metode kao što su **`first()`**, **`last()`**, te **`subset()`** koje služe za dohvaćanje prvog (najmanjeg), zadnjeg (najvećeg) ili elemente u nekom rasponu vrijednosti. Opisi ostalih metoda nalaze se u *javadocu*.

7.2.2.1. HashSet<E>

`HashSet<E>` je implementacija sučelja **`Set<E>`** proširena *hash*-tablicom kod koje vrijeme dohvaćanja elemenata ne ovisi o veličini same zbirke. Npr. sljedeći programski kod pokazuje da nije moguće isti objekt (referencu) dva puta staviti u zbirku:

```
// instanciranje HashSet-a
Set<String> hashSet = new HashSet<String>();

// definiranje objekta koji se sprema u hashSet
```

```
String jedan = "1";

// ispis vrijednosti koje vraća add metoda
System.out.println(hashSet.add(jedan));
System.out.println(hashSet.add(jedan));

// ispis vrijednosti koje vraća contains metoda
System.out.println(hashSet.contains(jedan));
```

Ispisuju se vrijednosti:

```
true //objekt 'jedan' je prvi put uspješno dodan
false //objekt 'jedan' drugi put nije dodan – sprečavanje
      //duplikata u Set-u
true //objekt 'jedan' postoji u zbirci hashSet
```

7.2.2.2. LinkedHashSet<E>

`LinkedHashSet<E>` nasljeđuje **`HashSet<E>`** pri čemu sortira elemente u skladu s poretkom kako su dodavani u zbirku. Radi malo sporije od **`HashSet<E>`**-a zbog potrebe stvaranja strukture povezane liste.

7.2.2.3. TreeSet<E>

`TreeSet<E>` je implementacija sučelja **`SortedSet<E>`** koja sadržaj sprema u strukturi stabla (engl. *tree structure*). Pretraživanje elemenata je sporije nego kod **`HashSet<E>`**-a, međutim, elementi zbirke su sortirani. Sljedeći programski kod demonstrira sortiranje elemenata koji se ubacuju u zbirku:

```
// instanciranje TreeSet-a
Set<Integer> treeSet = new TreeSet<Integer>();

// dodavanje elemenata koje je potrebno sortirati
treeSet.add(new Integer(1));
treeSet.add(new Integer(5));
treeSet.add(new Integer(8));
treeSet.add(new Integer(50));
treeSet.add(new Integer(3));
treeSet.add(new Integer(12));
treeSet.add(new Integer(7));

// ispis sadržaja treeSet-a koji mora biti sortiran
System.out.println(treeSet.toString());
```

Ispisuje se sadržaj:

```
[1, 3, 5, 7, 8, 12, 50] // sadržaj zbirke je sortiran
```

7.2.3. Sučelje List<E>

Sučelje **`List<E>`** definira zbirku gdje svaki element ima svoju poziciju u listi koja je određena rednim brojem (indeksom) od 0 do **`list.size() - 1`**. Drugim riječima, **`List<E>`** definira sekvencu (engl. *sequence*) elemenata. Novi element koji se dodaje stavlja se na kraj liste, a ukoliko se neki element briše iz liste, element koji je bio iza njega dolazi na njegovo mjesto itd. (svi elementi pomiču se za jedno mjesto "unaprijed").

`List<E>` proširuje sučelje **`Collection<E>`** s metodama od kojih su najvažnije **`get`** (dohvaćanje elementa liste prema njegovom indeksu u listi), **`set`** (postavljanje novog elementa u listu na određeni indeks s izbacivanjem elementa koji je bio na toj poziciji), **`add`** (dodavanje novog elementa u listu s pomicanjem ostalih "unazad"), **`remove`** (brisanje elemenata iz liste), **`indexOf`** (za određivanje indeksa određenog objekta u zbirci), te **`subList`** (dohvaćanje podliste). Detaljnije informacije o metodama nalaze se u *javadoc*-u.

Sve metode kojima se preda indeks elementa koji je manji od nule ili veći od veličine liste bacaju iznimku **`IndexOutOfBoundsException`**.

U paketu **`java.util`** nalaze se dvije implementacije sučelja **`List<E>`**: **`ArrayList<E>`** i **`LinkedList<E>`**.

7.2.3.1. ArrayList<E>

`ArrayList<E>` je osnovna implementacija **`List<E>`** sučelja koja elemente liste sprema u polje. Dodavanje i brisanje elemenata na kraju liste vremenski je manje zahtjevno od dodavanja i brisanja elemenata sa sredine liste, jer je zbog polja potrebno pomicati sve elemente za jedno mjesto unazad/unaprijed (brisanje/dodavanje).

`ArrayList<E>` ima svoj kapacitet koji je određen brojem elemenata koji može sadržavati bez potrebe alociranja (engl. *allocating*) nove memorije za veća polja. Ukoliko se taj kapacitet prekorači, mora se alocirati novo zamjensko polje (što se odrađuje automatski). Da bi se takve situacije izbjegle, korisno je definirati inicijalni kapacitet liste predavanjem parametra u **`ArrayList<E>`** konstruktor.

Sljedeći programski kod opisuje jednostavno korištenje zbirke **`ArrayList<E>`** i ispis njenog sadržaja nakon popunjavanja:

```
// instanciranje ArrayList-e
List<String> arrayList = new ArrayList<String>();

// dodavanje elemenata u listu
arrayList.add("1");
arrayList.add("1");
arrayList.add("2");
arrayList.add("1");
arrayList.add("1");
arrayList.add("4");
arrayList.add("7");

// ispis sadržaja liste
System.out.println(arrayList);
```

Ispisuje se sadržaj:

```
[1, 1, 2, 1, 1, 4, 7] // sadržaj polja – poredak je isti kao i
                      // redoslijed dodavanja elemenata u listu
```

7.2.3.2. LinkedList<E>

`LinkedList<E>` je dvostruko povezana lista čije se operacije dohvaćanja i brisanja elemenata dosta razlikuju od **`ArrayList<E>`**-e u performansama. Dodavanje i brisanje elemenata sa sredine liste je brzo zbog toga što nije potrebno kopiranje elemenata, dok je dohvaćanje elemenata zahtjevno jer je potrebno proći sve elemente od početka liste dok se ne pronađe traženi element.
`LinkedList<E>` implementira sučelje **`Queue<E>`** (koje se opisuje u sljedećem poglavlju) i optimizirano je za zbirke kod kojih se malo operacija obavlja na rubnim dijelovima liste (na početku i na kraju).

7.2.4. Sučelje Queue<E>

Sučelje **`Queue<E>`** predstavlja zbirku objekata koja definira glavnu poziciju (engl. *head position*) elementa koji će sljedeći biti obrisan. Redovi (engl. *queues*) često rade na principu FIFO (engl. *first-in-first-out*) strategije, no moguća je i strategija LIFO (engl. *last-in-first-out*) koja je poznatija kao stog (engl. *stack*). Svaka implementacija sučelja **`Queue<E>`** mora imati definirane kriterije sortiranja.

Sučelje **`Queue<E>`** definira metode za rad s elementom na glavnoj poziciji (engl. *head element*) kao što su **`element`** (vraćanje *head* elementa i bacanje **`NoSuchElementException`** iznimke u slučaju praznog reda), **`peek`** (vraćanje *head* elementa i vraćanje **`null`** vrijednosti u slučaju praznog reda), **`remove`** (vraćanje i brisanje *head* elementa i bacanje **`NoSuchElementException`** iznimke u slučaju praznog reda), te **`pool`** (vraća i briše *head* element i vraćanje null vrijednosti u slučaju praznog reda). Osim navedenih postoji i metoda **`offer`** koja dodaje element u red. Detaljniji opisi metoda nalaze se u *javadocu*.

Redovi općenito ne prihvaćaju **`null`** elemente zbog toga što je to vrijednost koju vraćaju metode **`peek`** i **`pool`** u slučaju praznog reda.

Zbirka **`LinkedList<E>`** predstavlja najjednostavniju implementaciju sučelja **`Queue<E>`**, a još postoji i implementacija **`PriorityQueue<E>`** koja predstavlja red temeljen na *prioritetnoj hrpi* (engl. *priority heap*). Glava reda je najmanji element unutar zbirke, a ostatak elemenata je sortiran od najmanjeg prema najvećem. Detaljniji opis nalazi se u *javadocu*.

7.2.5. Sučelje Map<K,V> i SortedMap<K,V>

Sučelje **`Map<K,V>`** ne nasljeđuje **`Collection<E>`** jer se razlikuje u nekoliko važnih karakteristika. Najvažnija razlika je u tome što se u mapu dodaju parovi *ključ-vrijednost* (engl. *key-value pairs*), umjesto dodavanja samo objekata kao u implementaciji **`Collection<E>`**. Sukladno tome, objekti se dohvaćaju pomoću ključeva. Povezivanje ključa s pripadajućim objektom u mapi često se naziva i mapiranje (engl. *mapping*). Svaki ključ vezan je samo uz jednu ili nijednu vrijednost i nije moguće imati više identičnih ključeva, međutim, više različitih ključeva može vratiti istu vrijednost.

Na primjer, ukoliko se mapa koristi za spremanje korisnika i njihovih adresa, ključevi u mapi bila bi imena korisnika, dok bi vrijednosti (objekti) u mapi bile adrese korisnika. U mapi ne postoje dva ista imena (ne postoje identični ključevi), dok bi više korisnika mogla imati istu adresu (više istih objekata za nekoliko različitih ključeva).

Osnovne metode u sučelju **`Map<K,V>`** su **`size`** (vraća veličinu mape, odnosno, broj parova ključ-vrijednost unutar zbirke), **`isEmpty`** (vraća **`true`** ukoliko je mapa prazna), **`containsKey`** (vraća **`true`** ukoliko mapa sadržava zadani ključ), **`containsValue`** (vraća

`true` ukoliko mapa sadržava zadani objekt), **`get`** (vraća objekt koji je mapiran zadanim ključem), **`put`** (dodavanje zadanog objekta u mapu pod zadanim ključem), te **`remove`** (brisanje mapinga za zadani ključ). Ostale metode i detalji navedenih metoda nalaze se u *javadocu*.

Iako **`Map<K,V>`** ne nasljeđuje **`Collection<E>`**, metode s istom funkcionalnošću imaju ista imena u oba sučelja, te analogne metode imaju analogna imena. Isto tako u sučelju **`Map<K,V>`** postoje i metode koje služe za dohvaćanje sadržaja mape koji se vraća pomoću zbirki koje su implementacije sučelja **`Collection<E>`**: metoda **`keySet`** vraća **`Set`** ključeva (nema duplikata) iz dotične mape, te metoda **`values()`** koja vraća **`Collection<E>`** objekata (vrijednosti) iz mape.

`SortedMap<K,V>` sučelje nasljeđuje **`Map<K,V>`** i definira mapu u kojoj su ključevi sortirani. Osim toga dodaje i neke metode karakteristične za sortiranu mapu od kojih su najvažnije: **`comparator`** (vraća **`Comparator<T>`** objekt koji definira kriterije sortiranja), **`firstKey`** (vraća ključ s najmanjom vrijednosti), **`lastKey`** (vraća ključ s najvećom vrijednosti), te **`subMap`** (vraća podskup parova ključ-vrijednost koja je definirana ulaznim parametrima **`minKey`** i **`maxKey`**).

Sučelje **`SortedMap<K,V>`** je prema **`Map<K,V>`** upravno ono što je **`SortedSet<E>`** prema sučelju **`Set<E>`**, osim što mape koriste ključeve za spremanje i dohvaćanje elemenata zbirke.

Paket **`java.util`** sadržava nekoliko osnovnih implementacija sučelja **`Map<K,V>`** od kojih su najvažnije **`HashMap<K,V>`**, **`LinkedHashMap<K,V>`** i **`TreeMap<K,V>`**.

7.2.5.1. HashMap<K,V>

`HashMap<K,V>` predstavlja implementaciju sučelja **`Map<K,V>`** pomoću hash tablice, pri čemu se hash vrijednost svakog ključa (koje se određuje pozivanjem metode **`hashCode`**) koristi za određivanje pozicije unutar mape. **`Hashmap<K,V>`** je zbirka kod koje su operacije dodavanja, brisanja i traženja elemenata vrlo učinkovite, što je čini jednom od najčešće korištenih zbirki u Javi.

Sljedeći programski kod opisuje jednostavno korištenje mape **`HashMap<K,V>`** (dodavanje ključeva koji predstavljaju prezimena osoba, te objekata koji predstavljaju adrese osoba):

```
// instanciranje HashMap-e
Map<String,String> hashMap = new HashMap<String,String>();

// dodavanje prve vrijednosti s ključem "Peric"
System.out.println(hashMap.put("Peric", "Trg bana Jelacica 12"));

// dodavanje druge vrijednosti s ključem "Perić" (pokušaj
// dupliciranja ključa)
System.out.println(hashMap.put("Peric", "Trg bana Jelacica 12"));
```

```
// dodavanje novog para ključ-vrijednost
System.out.println(hashMap.put("Horvat", "Ilica 345"));


// dohvaćanje elementa koji je dvaput dodan u mapu
System.out.println(hashMap.get("Peric"));

// ispisivanje seta ključeva ("dupli" ključ se pojavljuje samo
// jednom)
System.out.println(hashMap.keySet());

// ispisivanje vrijednosti svih objekata u mapi
System.out.println(hashMap.values());
```

Ispisuje se sadržaj:

```
null      // prvo umetanje ključa "Peric" i pripadajuće adrese –
        // povratna vrijednost "null" označava da do tada u mapi
        // nije bilo vrijednosti s ključem "Peric"

Trg bana Jelacica 12 // drugo umetanje ključa "Peric" – ovaj put
                     // se ne vraća "null" zbog toga što već
                     // postoji adresa "Trg bana Jelacica 12" u
                     // mapi pod tim ključem pa se stari element s
                     // tim ključem zamjenjuje novim

null // umetanje novog ključa "Horvat" koji još ne postoji u mapi

Trg bana Jelacica 12 // dohvaćanje elementa za ključ "Peric"

[Peric, Horvat] // ispis svih ključeva ("Peric" je samo jednom u
                // mapi)

[Trg bana Jelacica 12, Ilica 345] // ispis svih elemenata
```

7.2.5.2. LinkedHashMap<K,V>

`LinkedHashMap<K,V>` nasljeđuje **`HashMap<K,V>`** i definira mapu koja zadržava redoslijed umetanja elemenata u mapi. **`LinkedHashMap<K,V>`** zbog potrebe kreiranja strukture povezane liste ima malo lošije performanse od **`HashMap<K,V>`**, međutim, operacija iteriranja ovisna je samo o veličini mape. Detalji ove implementacije mogu se naći u *javadocu*.

7.2.5.3. TreeMap<K,V>

`TreeMap<K,V>` klasa implementira sučelje **`SortedMap<K,V>`** i sortira ključeve na isti način kao i **`SortedSet<E>`**. **`TreeMap<K,V>`** se koristi samo u slučajevima kad je potrebno sortiranje elemenata ili u slučajevima kad je metoda **`hashCode`** za generiranje

hash koda ključeva nadjačana i nije optimalno implementirana. Detalji implementacije mogu se naći u *javadocu*.

7.2.6. Iteracija

Iteracija kod zbirki objekata je operacija pomoću koje se prolazi po sadržaju zbirke i dohvaćaju pojedini objekti unutar nje. U Javi za potrebe iteriranja po zbirkama objekata postoji sučelje **`Iterator<E>`** koje sadrži tri osnovne metode za iteraciju:

- **`hasNext()`** - metoda vraća true ukoliko iteracija ima još elemenata.
- **`next()`** - metoda vraća sljedeći element u iteraciji. Ukoliko više nema elemenata, baca se iznimka **`NoSuchElementException`**.
- **`remove()`** - metoda briše element koji je netom vraćen iteracijom iz zbirke po kojoj se iterira. Metoda **`remove()`** može se pozvati samo jednom nakon svakog pozivanja metode **`next()`**, u protivnom se baca iznimka **`IllegalStateException`**.

Sljedeći primjer koda opisuje kako je pomoću for petlje i iteratora moguće iterirati kroz zbirku **`String`** objekata (npr. **`ArrayList<E>`**) i traženje zapisa "Trazeni":

```
// definiranje liste
List<String> lista = new ArrayList<String>();

// popunjavanje liste (dodavanje elemenata)
lista.add("Prvi");
lista.add("Drugi");
lista.add("Treci");
lista.add("Trazeni");

// unutar for petlje obavlja se inicijalizacija iteratora i
// provjeravanje postojanja sljedećeg elementa u listi
for (Iterator<String> iterator = lista.iterator();
iterator.hasNext();) {
   // dohvaćanje sljedeg elementa pomoću iteratora
   String string = iterator.next();
   // provjeri je li trenutni element traženi
   if (string.equals("Trazeni")) {
      System.out.println("Pronadjen je '" + string
                         + "' u listi.");
      break;
   }
}
```

7.3. Enumeracije

Enumeracije su tipovi za koje su sve vrijednosti koje može poprimiti taj tip poznate kod definiranja. Na primjer, enumeracija za dane u tjednu bio bi skup vrijednosti: Ponedjeljak, Utorak, Srijeda, Četvrtak, Petak , Subota i Nedjelja.

U nekim programskim jezicima enumeracije su setovi imenovanih cjelobrojnih vrijednosti, međutim, u Javi enumeracije predstavljaju poseban tip klase s instancama koje poprimaju jednu od vrijednosti iz enumeriranog skupa vrijednosti.

Definicija enumeracije koja definira dane u tjednu bila bi ovakva:

```
enum Days { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY }
```

U osnovnoj izvedbi deklaracija enumeracije sastoji se od ključne riječi **`enum`**, identifikatora koji označava enumeraciju (**`Days`**), te tijela enumeracije koje sadrži elemente enumeracije. Konvencija nalaže da su elementi enumeracije navedeni velikim slovima. Enumeracijske konstante su statička polja klase (mogu se dohvatiti bez potrebe instanciranja klase):

```
Day currentDayOfTheWeek = ...;
if (currentDayOfTheWeek == Days.MONDAY) {
   System.out.println("Danas je ponedjeljak!");
}
```

Enumeracije mogu sadržavati i neke metode, međutim, u tom slučaju lista elemenata enumeracije mora završavati znakom ';' :

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY;
   public static int getSize() { return 7; }
}
```

Iako strukturom može podsjećati na običnu klasu (sadržava statičke konstante i statičke metode), enumeracija ne može nasljeđivati druge tipove zbog toga što sve enumeracije implicitno implementiraju klasu **`java.lang.Enum`**. Isto tako druge klase ne mogu nasljeđivati enumeracije zbog toga što je svaka enumeracija implicitno deklarirana modifikatorom **`final`**.

Svaka enumeracija ima dvije statičke metode koje se automatski generiraju: metodu **`values`** vraća polje koje sadržava sve elemente enumeracije i metodu **`valueOf`** koja vraća konstantu enumeracije za zadano ime. Detalji metoda nalaze se u *javadoc* dokumentaciji.

8. ULAZNO-IZLAZNE OPERACIJE I RAD S DATOTEKAMA

8.1. ULAZNO-IZLAZNE OPERACIJE

Paket **`java.io`** sadržava veliki broj klasa za pisanje i čitanje podataka koji mogu biti u obliku toka podataka (engl. *streaming data*) ili u obliku slijednih podataka (engl. *sequential data*). Tokovi (engl. *streams*) su poredani slijedovi podataka koji imaju izvor (engl. *input stream*) i odredište (engl. *output stream*). Klase **`InputStream`** i **`OutputStream`** služe za čitanje toka *byte*-ovima ili pisanje u tok *byte*-ova (engl. *streams of bytes*), a klase **`Reader`** i **`Writer`** služe za čitanje i pisanje toka znakova (engl. *streams of characters*). Tokovi mogu biti i ugnježđeni, što znači da je moguće znakove čitati iz objekta klase **`FilterReader`** koji čita i procesira podatke iz nekog pozadinskog **`Reader`** toka. Taj pozadinski tok također može čitati bajtove iz objekta **`InputStream`** klase i pretvarati ih u znakove.

8.1.1. Tokovi bajtova

Paket **`java.io`** definira nekoliko apstraktnih klasa za osnovne ulazne i izlazne tokove *byte*-ova. Te apstraktne klase su nasljeđene u drugim klasama kako bi tvorile korisne implementacije koje je moguće iskoristiti. Tipovi tokova najčešće postoje u parovima: na primjer, postoji klasa **`FileInputStream`** za čitanje sadržaja binarnih datoteka, pa postoji i klasa **`FileOutputStream`** za zapisivanje podataka u binarne datoteke.

Svi tokovi bajtova imaju nekoliko zajedničkih karakteristika. Na primjer, svi tokovi moraju se **otvarati prije korištenja** i **zatvarati nakon korištenja**.

8.1.1.1. Klasa `InputStream`

Apstraktna klasa **`InputStream`** deklarira metoda koje se koriste za čitanje bajtova iz određenog izvora. **`InputStream`** je nadklasa većini klasa ulaznih tokova u paketu **`java.io`** i definira sljedeće metode:

- **`public abstract int read() throws IOException`**

Metoda čita jedan *byte* podataka i vraća taj *byte* koji je pročitan u obliku cjelobrojnih brojeva od 0 do 255 (a ne od -128 do 127). Ukoliko je prilikom čitanja došlo do kraja toka bajtova, vraća se -1. Metoda vraća cjelobrojne vrijednosti umjesto stvarnih byte vrijednosti zbog toga jer je potrebno vratiti sve vrijednosti i znak koji označava kraj toka *byte*-ova. To zahtijeva korištenje većeg skupa vrijednosti koji se može spremiti i podatak tipa *byte* pa se koristi veći skup vrijednosti koji pokriva **`int`**. Ukoliko tijekom čitanja dođe do ulazno/izlazne pogreške, baca se **`IOException`**.

- **`public int read(byte[] buf, int offset, int count) throws IOException`**

Metoda čita *byte*-ove u dio polja. Maksimalan broj pročitanih *byte*-ova određen je ulaznim parametrom **`count`**. Pročitani *byte*-ovi spremaju se u polje od mjesta **`buf[offset]`** do mjesta **`buf[offset + count – 1]`**, a sva ostala mjesta u polju ostaju nepromijenjena. Metoda vraća broj pročitanih *byte*-ova. Ukoliko nije pročitan nijedan *byte* zbog kraja toka *byte*-ova, vraća se vrijednost -1. Ukoliko je ulazni parametar **`count`** jednak nuli, vraća se 0. Ukoliko tijekom čitanja prvog *byte*-a dođe do ulazno/izlazne pogreške, baca se **`IOException`**. Međutim, ukoliko tijekom čitanja ostalih *byte*-ova dođe do pogreške, ne baca se iznimka, već metoda vraća broj pročitanih bajtova do trenutka pogreške.

- **`public int read(byte[] buf) throws IOException`**

Metoda je ekvivalent pozivu metode **`read(buf, 0, buf.length)`**.

- **`public long skip(long count) throws IOException`**

Metoda preskače količinu *byte*-ova definiranu ulaznih parametrom **`count`** tako dugo dok se ne dođe do kraja toka *byte*-ova, a vraća broj preskočenih *byte*-ova. Ukoliko je ulazni parametar **`count`** negativan, ne preskače se nijedan *byte*.

- **`public int available() throws IOException`**

Metoda vraća broj *byte*-ova koji se može pročitati (ili preskočiti).

- **`public void close() throws IOException`**

Metoda zatvara ulazni tok bajtova, a predviđena je za oslobađanje resursa kao što je opisnik datoteke (engl. *file descriptor*) koji je povezan s tokom *byte*-ova. Kad se tok *byte*-ova zatvori, svaka sljedeća operacija s tokom *byte*-ova prouzročiti će bacanje iznimke

`IOException`. Zatvaranje toka *byte*-ova koji je već zatvoren nema učinka.

Sljedeći programski kod opisuje korištenje ulaznog toka *byte*-ova za prebrojavanje *byte*-ova u datoteci ili iz **`System.in`** toka ako datoteka ne postoji:

```
import java.io.*;

public class CountBytes {

    public static void main(String[] args) throws IOException {

        InputStream in;
        if (args.length == 0)
            in = System.in;
        else
            in = new FileInputStream(args[0]);

        int total = 0;
        while (in.read() != -1)
            total++;
        System.out.println(total + " bytes");
    }
}
```

Program prima ime datoteke iz komandne linije. Varijabla **`in`** predstavlja ulazni tok *byte*-ova. Ukoliko nije predano ime datoteke, upotrebljava se standardni ulazni tok **`System.in`**, a ukoliko je predano ime datoteke, kreira se objekt klase **`FileInputStream`** koja je podklasa klase **`InputStream`**. Petlja **`while`** prebrojava *byte*-ove u datoteci i na kraju programa se ispisuje rezultat, odnosno sumu bajtova.

8.1.1.2. Klasa `OutputStream`

Apstraktna klasa **`OutputStream`** je analogna klasi **`InputStream`** i predstavlja mehanizam za zapisanje *byte*-ova podataka u neko odredište. Klasa definira sljedeće metode:

- **`public abstract void write(int b) throws IOException`**

Metoda zapisuje **`b`** kao **`byte`**. Podatak tipa **`byte`** predan je kao podatak tipa **`int`** zbog toga što je često rezultat neke aritmetičke operacije s *byte*-ovima. Izrazi koji uključuju operacije *byte*-ovima su tipa int pa rezultate tih operacija nije potrebno *cast*-ati u tip **`byte`**.

- **`public void write(byte[buf], int offset, int count) throws IOException`**

Metoda zapisuje dio polja *byte*-ova, počevši od pozicije **`buf[offset]`** i zapisuje **`count`** *byte*-ova.

- **`public void write(byte[] buf) throws IOException`**

Metoda je ekvivalentna poziciju metode **`write(buf, 0, buf.length)`**.

- **`public void flush() throws IOException`**

Metoda sprema trenutno promjene u izlazni tok *byte*-ova. Ukoliko tok *byte*-ova drži u međuspremniku (engl. *buffer*) neke promjene, metodom **`flush`** zapisuje sve promjene u ciljno odredište (datoteku, neki drugi tok *byte*-ova ili sl.)

- **`public void close() throws IOException`**

Metoda zatvara izlazni tok *byte*-ova, a predviđena je za oslobađanje resursa kao što je opisnik datoteke (engl. *file descriptor*) koji je povezan s tokom *byte*-ova. Nakon što se tok *byte*-ova zatvori, sve operacije s tokom izazivaju bacanje iznimku **`IOException`**. Zatvaranje

Sljedeći programski kod prikazuje kopiranje ulaznog toka *byte*-ova u izlazni tok *byte*-ova, s tim da se prevodi određeni *byte* u neki drugi. Program **`TranslateByte`** prima dva parametra: **`from`** *byte* i **`to`** *byte*. Ukoliko uneseni *byte* na standardnom ulazu (tipkovnici) odgovara prvom *byte*-u prvog ulaznog argumenta, ispisuje se prvi *byte* drugog ulaznog parametra. Inače se ispisuje uneseni *byte*.

```
import java.io.*;

public class TranslateByte {
   public static void main(String[] args) throws IOException {
      byte from = (byte) args[0].charAt(0);
      byte to   = (byte) args[1].charAt(0);
      int b;
      while ((b = System.in.read()) != -1)
         System.out.write(b == from ? to : b);
      }
   }
```

8.1.2. Tokovi znakova

Apstraktne klase za čitanje i pisanje tokova znakova su **`Reader`** i **`Writer`**. Svaka klasa ima metode slične onima iz klasa **`InputStream`** i **`OutputStream`**. Na primjer, **`InputStream`** ima **`read`** metodu koja vraća tip podatka **`byte`**, a **`Reader`** ima **`read`** metodu koja vraća podatak tipa **`char`**. Isto tako, **`OutputStream`** ima metode koje zapisuju polja *byte*-ova, dok **`Writer`** ima metode koje zapisuju polja znakova. Tokovi znakova su dizajnirani nakon tokova *byte*-ova da bi omogućili podršku *Unicode* znakova i olakšali korištenje povezanih klasa.

8.1.2.1. Klasa `Reader`

Apstraktna klasa **`Reader`** sadržava metode za rukovanje tokom znakova, analogne onima u klasi **`InputStream`**:

- **`public int read() throws IOException`**

Metoda čita jedan znak i vraća ga u obliku cjelobrojnog broja u rasponu od 0 do 65535 koji predstavlja *Unicode* kod znaka. Ukoliko se došlo do kraja toka znakova, vraća se -1.

- **`public abstract int read(char[] buf, int offset, int count) throws IOException`**

Metoda čita u dio polja koje sadržava znakove (**`char`**). Maksimalan broj znakova koji se čita je određen ulaznim parametrom **`count`**. Pročitani znakovi spremaju se od pozicije **`buf[offset]`** pa sve do pozicije **`buf[offset + count – 1]`**, a sva ostala mjesta u polju ostaju nepromijenjena. Metoda vraća broj pročitanih znakova. Ukoliko nije pročitan nijedan znak zbog kraja toka znakova, metoda vraća -1. Ukoliko je parametar **`count`** jednak nuli, ne pročita se nijedan znak i metoda vraća 0.

- **`public int read(char[] buf) throws IOException`**

Metoda je ekvivalentna pozivu metode **`read(buf, 0, buf.length)`**.

- **`public int read(java.nio.CharBuffer) throws IOException`**

Metoda pokušava čitati koliko je god moguće znakova i sprema ih u određeni međuspremnik znakova, bez mogućnosti prekoračenja veličine međuspremnika i vraća broj pročitanih znakova. Ukoliko nije pročitan nijedan znak zbog kraja toka znakova, metoda vraća -1.

- **`public long skip(long count) throws IOException`**

Metoda preskače broj znakova koji je određen ulaznim parametrom count ili ukoliko je kod čitanja došlo do kraja toka znakova. Vraća broj preskočenih znakova. Parametar **`count`** ne smije biti negativan.

- **`public boolean ready() throws IOException`**

Metoda vraća **`true`** ukoliko tok znakova spreman za čitanje, odnosno, ukoliko je barem jedan znak raspoloživ za čitanje.

- **`public abstract void close() throws IOException`**

Metoda zatvara tok znakova, a predviđena je za oslobađanje resursa kao što je opisnik datoteke (engl. *file descriptor*) koji je povezan s tokom znakova.

Sljedeći primjer programskog koda izračunava broj praznih znakova (engl. *space*) u toku

znakova:

```
import java.io.*;

public class CountSpace {
    public static void main(String[] args) throws IOException {
        Reader in;
        if (args.length == 0)
            in = new InputStreamReader(System.in);
        else
            in = new FileReader(args[0]);
        int ch, total, spaces = 0;
        for (total = 0; (ch = in.read()) != -1; total++) {
           if (Character.isWhitespace((char) ch))
                spaces++;
           }
        System.out.println(total + " chars, " + spaces
                                 + " spaces");
     }
}
```

Program prima ime datoteke iz komandne linije. Varijabla **`in`** predstavlja tok znakova. Ukoliko se ne navede ime datoteke, koristi se standardni ulazni tok *omotan* (engl. *wrapped*) u **`InputStreamReader`** koji pretvara ulazni tok bajtova u ulazni tok znakova. Ukoliko se ime datoteke navede, kreira se objekt klase **`FileReader`** koja je podklasa klase **`Reader`**.

Petlja **`for`** izračuna broj znakova u datoteci i broj praznih znakova koristeći metodu statičku metodu **`isWhitespace`** iz klase **`Character`**.

8.1.2.2. Klasa **`Writer`**

Apstraktna klasa **`Writer`** sadržava metode za rukovanje tokom znakova, analogne onima u klasi **`OutputStream`**, samo što se umjesto *byte*-ova koriste znakovi:

- **`public void write(int ch) throws IOException`**

Metoda zapisuje **`ch`** kao znak. Znak je predan kao **`int`** reprezentacija znaka.

- **`public abstract void write(char buf[], int offset, int count) throws IOException`**

Metoda zapisuje dio polja znakova, počevši od pozicije **`buf[offset]`** i zapisivanje broja
- znakova određenog parametrom **`count`**.

- **`public void write(char[] buf) throws IOException`**

Metoda je ekvivalentna pozivu metode **`write(buf, 0, buf.length)`**.

- **`public void write(String str, int offset, int count) throws`**

`IOException`

Metoda zapisuje broj znakova određen ulaznim parametrom iz **`String`** parametra **`str`** u

- tok znakova, počevši od pozicije **`str.charAt(offset)`**.

- **`public void write(String str) throws IOException`**

Metoda je ekvivalent pozivu metode **`write(str, 0, str.length)`**.

- **`public abstract void flush() throws IOException`**

Metoda sprema trenutne promjene u izlazni tok znakova. Ukoliko tok znakova drži u

- međuspremniku (engl. *buffer*) neke promjene, metodom **`flush`** zapisuje sve promjene u
- ciljno odredište (datoteku, neki drugi tok znakova ili sl.).
-

- **`public abstract void close() throws IOException`**

Metoda zatvara tok znakova i po potrebi poziva metodu **`flush`**. Predviđena je za

- oslobađanje resursa kao što je opisnik datoteke (engl. *file descriptor*) koji je povezan s
- tokom znakova.

8.1.2.3. Klase InputStreamReader i OutputStreamWriter

Klase **`InputStreamReader`** i **`OutputStreamWriter`** predstavljaju tokove za pretvorbu između tokova znakova i tokova *byte*-ova koristeći određeni način kodiranja (engl. *encoding*). **`InputStreamReader`** objekt prima ulazni tok *byte*-ova čiji sadržaj pretvara u odgovarajuće znakove kodirane **`UTF-16`** kodom. Objekt **`OutputStreamWriter`** ispisuje tok *byte*-ova u odredište koji se pretvara iz zadanog toka znakova kodiranih **`UTF-16`** kodom. Na primjer, sljedeći programski kod čita tok *byte*-ova kodiran pomoću **`windows-1250`** načina kodiranja i pretvara ih u znakove kodirane **`UTF-16`** načina kodiranja:

```
public static Reader readCroatian(String file) throws IOException
{
     InputStream fileIn = new FileInputStream(file);
     return new InputStreamReader(fileIn, "windows-1250");
}
```

Detaljnije informacije o navedenim klasama nalaze se u *javadoc* dokumentaciji.

8.1.3. Implementacije klasa za korištenje tokova

8.1.3.1. `Filter` *tokovi*

`Filter` tokovi su apstraktne klase koje predstavljaju tokove s određenim operacijama filtriranja nad podacima koji se čitaju ili zapisuju od drugog toka. Na primjer, objekt **`FilterReader`** dobiva ulazni tok od nekog drugog **`Reader`** objekta, procesira (filtrira) znakovni tok po određenim kriterijima i vraća filtrirani rezultat.

Konkretne implementacije su **`FilterInputStream`**, **`FilterOutputStream`**, **`FilterReader`** i **`FilterWriter`** koje rade s tokovima *byte*-ova i tokovima znakova. Detalji pojedine klase nalaze se u *javadoc* dokumentaciji.

8.1.3.2. `Buffered` *tokovi*

`Buffered` tokovi dodaju svojstvo privremenog spremanja (engl. *buffering*) podataka zbog kojeg tijekom korištenja metoda **`read`** i **`write`** ne moraju neprestano imati pristup datotečnom sustavu. Konkretne implementacije su **`BufferedInputStream`**, **`BufferedOutputStream`**, **`BufferedReader`** i **`BufferredWriter`** koje rade s tokovima bajtova i tokovima znakova. Detalji pojedine klase nalaze se u *javadoc* dokumentaciji.

Primjer programskog koda koji čita unesene linije teksta u konzolu (engl. *Console*), a koristi implementaciju **`BufferedReader`** klase dan je u nastavku:

```
import java.io.*;

...

BufferedReader console = new BufferedReader(
                    new InputStreamReader(System.in));
System.out.print("Unesite svoje ime: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) {
  //slučaj pogreške kod čitanja
  ime = "<" + e + ">";
}

System.out.println("Dobar dan, " + name);
```

8.1.3.3. `Piped` *tokovi*

Cjevododi (engl. *pipes*) tvore mehanizam komunikacije između dviju niti (engl. *threads*), pri čemu jedna nit zapisuje podatke u cjevovod, a druga nit čita podatke iz cjevododa.

Konkretne implementacije su **`PipedInputStream`**, **`PipedOutputStream`**, **`PipedReader`** i **`PipedWriter`** koje rade s tokovima *byte*-ova i tokovima znakova. Detalji pojedine klase nalaze se u *javadoc* dokumentaciji.

8.1.3.4. **`ByteArray`** *tokovi bajtova*

Ukoliko je potrebno koristiti polja *byte*-ova kao izvor ili odredište tokova *byte*-ova, koriste se **`ByteArray`** tokovi. Konkretne implementacije su **`ByteArrayInputStream`** i **`ByteArrayOutputStream`**, a detalji pojedine klase mogu se naći u *javadoc* dokumentaciji.

8.1.3.5. **`CharArray`** *tokovi znakova*

`CharArray` tokovi znakova analogni su **`ByteArray`** tokovima, pri čemu se koriste kao izvor ili odredište koristi polje znakova. Konkretne implementacije su **`CharArrayReader`** i **`CharArrayWriter`**, a detalji pojedine klase mogu se naći u *javadoc* dokumentaciji.

8.1.3.6. **`String`** *tokovi znakova*

Implementacija **`StringReader`** čita znakove iz **`String`** objekta, dok implementacija **`StringWriter`** zapisuje znakove u međuspremnik koji može biti **`String`** ili **`StringBuffer`** objekt. Detalji pojedine klase mogu se naći u *javadoc* dokumentaciji.

8.1.3.7. **`Print`** *tokovi*

Implementacije **`Print`** tokova **`PrintStream`** i **`PrintWriter`** imaju metode koje olakšavaju zapisivanje vrijednosti primitivnih tipova i objekata u tok. **`PrintStream`** klasa koristi se za tokove *byte*-ova, a **`PrintWriter`** klasa koristi se za tokove znakova. Detalji pojedine klase mogu se naći u *javadoc* dokumentaciji.

8.1.3.8. ***`StreamTokenizer`***

Tokeniziranje ulaznog teksta (razdvajanje u dijelove odijeljene određenim znakom – delimiterom) često se koristi u programskom jeziku Java. Klasa **`StreamTokenizer`** iz paketa **`java.io`** služi za jednostavno tokeniziranje ulaznog teksta. Općenitija klasa za učitavanje i pretvorbu ulaznog teksta je **`java.util.Scanner`**.

8.1.3.9. Podatkovni tokovi bajtova

Čitanje i pisanje znakova je korisno, međutim, često je potrebno slati binarne podatke

određenog tipa među tokovima. Sučelja **`DataInput`** i **`DataOutput`** definiraju metode koje šalju primitivne tipove između tokova. Klase **`DataInputStream`** i **`DataOutputStream`** predstavljaju podrazumijevanu implementaciju svakog sučelja. Detalji navedenih klasa i sučelja mogu se pronaći u *javadoc* dokumentaciji.

8.2. RAD S DATOTEKAMA

Paket **`java.io`** sadržava niz klasa koje olakšavaju rad s datotekama. Klase **`File`** tokova omogućavaju čitanje iz datoteka i zapisivanje u datoteke, a **`FileDescriptor`** klasa omogućava datotečnom sustavu da datoteke koristi poput objekata. Klasa **`RandomAccessFile`** omogućava korištenje datoteka kao tokove bajtova ili znakova sa slučajnim pristupom (engl. *randomly access streams*). Interakcija s datotekom koja je dio datotečnog sustava obavlja se uz pomoć **`File`** klase koja omogućava apstrakciju lokacije datoteka i koristi metode za manipuliranje imenima datoteka.

8.2.1. `File` tokovi i `FileDescriptor`

`File` tokovi **`FileInputStream`**, **`FileOutputStream`**, **`FileReader`** i **`FileWriter`** omogućavaju tretiranje datoteke kao ulazni ili izlazni tok podataka. Svaki tip instancira se pomoću jednog od tri konstruktora:

- Konstruktor koji prima **`String`** objekt koji predstavlja ime datoteke
- Konstruktor koji prima **`File`** objekt koji označava datoteku
- Konstruktor koji prima **`FileDescriptor`** objekt

Ukoliko datoteka s navedenim imenom i lokacijom ne postoji, ulazni tokovi bacaju iznimku **`FileNotFoundException`**. Pristupanje datoteci zahtijeva određene sigurnosne provjere pa u slučaju nedostatka prava za pristupanje datoteci baca se **`SecurityException`**.

`FileDescriptor` objekt predstavlja identifikator ovisan o datotečnom sustavu koji opisuje otvorenu datoteku. Moguće ga je dohvatiti pozivanjem metode **`getFD`** nad objektom **`File`** i to samo u slučaju **`File`** tokova bajtova, ali ne i **`File`** tokova znakova. Ispravnost **`FileDescriptor`** objekta moguće je provjeriti pozivanjem metode **`valid`** koja vraća **`boolean`** rezultat. Opisnici datoteka kreirani pomoću konstruktora **`FileDescriptor`** koji ne prima nikakve ulazne parametre nisu ispravni.

8.2.2. `RandomAccessFile` klasa

Klasa **`RandomAccessFile`** podržava napredniji datotečni mehanizam od **`File`** tokova. Datoteka sa slučajnim pristupom (engl. *random access file*) ponaša se kao veliko polje *byte*-ova spremljeno u datotečnom sustavu. Polju se pristupa pomoću neke vrste kursora ili indeksa polja koji se naziva pokazivač na datoteku (engl. *file pointer*): ulazne operacije čitaju *byte*-ove počevši od pokazivača na datoteku i pomiču ga iza pročitanih *byte*-ova.

Ukoliko je datoteka sa slučajnim pristupom kreirana za čitanje i pisanje, moguće su i izlazne operacije koje zapisuju *byte*-ove počevši od pokazivača na datoteku i pomiču ga iza zapisanih bajtova.

`RandomAccessFile` klasa nije podklasa **`InputStream`**, **`OutputStream`**, **`Reader`** ili **`Writer`** klasa, jer može čitati i pisati, te raditi s tokovima znakova i tokovima *byte*-ova. Konstruktor prima parametar koji određuje radi li se o ulaznom ili ulazno/izlaznom toku podataka.

`RandomAccessFile` podržava metode za čitanje i pisanje (**`read`** i **`write`**) koje imaju ista imena i potpis (engl. *signature*) kao i metode koje rade s tokovima bajtova. Osim toga **`RandomAccessFile`** implementira i sučelja **`DataInput`** i **`DataOutput`**. Detaljnije informacije moguće je pronaći u *javadoc* dokumentaciji.

8.2.3. `File` klasa

Klasa **`File`** (koja se ne odnosi na **`File`** tokove iz poglavlja 6.1.2.1.) sadržava niz funkcionalnosti koja se koriste s imenima datoteka. Osim toga sadržava i metode koje odvajaju imena putanje to datoteke (engl. *pathnames*) u zasebne komponente i dohvaćanje informacija o određenoj datoteci.

`File` objekt predstavlja zapravo putanju do datoteke, a ne nužno samu datoteku. Na primjer, kako bi se provjerilo da li zadana putanja do datoteke zaista označava postojeću datoteku u datotečnom sustavu, nad kreiranim **`File`** objektom sa zadanom putanjom do datoteke potrebno je pozvati metodu **`exists`**.

Putanja se razdjeljuje na dijelove koji označavaju direktorij (engl. *directory*) i dijelove koji označavaju datoteku pomoću znaka spremljenog u polju klase (statičkom polju) **`separatorChar`** ili pomoću **`String`**-a u polju klase **`separator`**.

`File` objekti kreiraju se pomoću jednog od sljedeća četiri konstruktora: **`File(String path)`**, **`File(String dirName, String name)`**, **`File(File fileDir, String name)`** i **`File(java.net.URI)`**. Datoteka je u njima određena putanjom (**`path`**), imenom direktorija i imenom datoteke (**`dirname + File.separator + name`**), imenom direktorija iz **`File`** objekta i imenom datoteke (**`fileDir.getPath() + name`**), te jedinstvenim identifikatorom resursa (engl. *Uniform Resource Identifier* – URI). Detaljnije informacije o konstruktorima mogu se naći u *javadoc* dokumentaciji.

Klasa **`File`** podržava i pet "*getter*" metoda kojima se dohvaćaju informacije o komponentama putanje do datoteke. Sljedeći kod koristi svaku od tih pet metoda nakon što je kreiran objekt **`File`** za datoteku **`FileInfo.java`** koja se nalazi unutar "ok" poddirektorija koji se nalazi direktorija iznad trenutnog direktorija (određenog znakovima ".."):

```
File src = new File(".." + File.separator + "ok","FileInfo.java");
System.out.println("getName() = " + src.getName());
System.out.println("getPath() = " + src.getPath());
```

```
System.out.println("getAbsolutePath() = "+src.getAbsolutePath());
System.out.println("getCanonicalPath() = " +
src.getCanonicalPath());
System.out.println("getParent() = " + src.getParent());
```

a na izlazu se ispisuje npr.:

```
getName() = FileInfo.java
getPath() = ../ok/FileInfo.java
getAbsolutePath() = /vob/java_prog/src/../ok/FileInfo.java
getCanonicalPath() = /vob/java_prog/ok/FileInfo.java
getParent() = ../ok
```

Detaljnije razlike između apsolutne putanje (engl. *absolute path*) i kanonske putanje (engl. *canonical path*), kao i informacije o ostalim metodama koje podržava klasa File moguće je naći u *javadoc* dokumentaciji.

8.3. PRIMJERI S DATOTEKAMA

8.3.1. Čitanje tekstualne datoteke

Slična operacija čitanja ulaza iz konzole je čitanje teksta iz tekstualne datoteke. Sljedeći programski kod čita datoteku sve dok ne dođe do njenog kraja:

```
String filename = "C:\\input.txt";
try {
  BufferedReader in = new BufferedReader(
                                        new FileReader(filename));
  String line;
  // Čitanje jedne linije i provjera kraja datoteke
  while((line = in.readLine()) != null) {
    //  Ispis linije
    System.out.println(line);
  }

  //  Svaki tok je potrebno zatvoriti nakon njegovog korištenja
  in.close();
}
catch (IOException e) { /* Obrada eventualne pogreške */}
```

8.3.2. Zapisivanje teksta u datoteku

Slično kao što naredba `System.out.println()` za ispisivanje u konzolu koristi izlazni podatkovni tok, tekst je moguće ispisivati u bilo koji podatkovni tok. Sljedeći programski kod prikazuje kako je moguće ispisati tekst u datoteku:

```
try {
  File f = new File("C:\\output.txt");
  PrintWriter out = new PrintWriter(new FileWriter(f));
```

```
  out.println("Prvi redak");
  out.close();  // Upisivanje je završeno
}
catch (IOException e) { /* Obrada eventualne pogreške */ }
```

8.3.3. Čitanje binarne datoteke

Osim teksta, datoteke mogu imati i binarni sadržaj. U sljedećem programskom kodu datoteka se tretira kao tok *byte*-ova i čita *byte*-ove u veliko polje:

```
try {
  File f = new File("test.bin"); // Binarna datoteka koja se čita
  int filesize = (int) f.length(); // Određivanje veličine podatka
  byte[] data = new byte[filesize]; // Kreiranje polja

  // Kreiranje toka bajtova za čitanje datoteke
  DataInputStream in = new DataInputStream(
                                        new FileInputStream(f));
  in.readFully(data);// Čitanje sadržaja binarne datoteke u polje
  in.close();
}
catch (IOException e) { /* Obrada eventualnih pogrešaka */ }
```

8.4. SERIJALIZACIJA OBJEKATA

Svojstvo spremanja objekata u tokove bajtova koji mogu biti prenošeni preko mreže, spremljeni na disk u obliku datoteke ili spremljeni u bazu podataka, te kasnije natrag oblikovati "živi" objekt naziva se **serijalizacija objekata**.

U Javi postoje specijalizirani tokovi bajtova za objekte `ObjectInputStream` i `ObjectOutputStream` koji omogućavaju serijalizaciju i deserijalizaciju cijelih objekata.

8.4.1. `Object` tokovi bajtova

`Object` tokovi omogućavaju čitanje i zapisivanje **grafova objekata** (engl. *object graphs*) zajedno s primitivnim tipovima, stringovima i poljima. Graf objekta je stablasta struktura objekta koji se zapisuje pomoću metode `writeObject` u `ObjectOutputStream` uključujući sve druge objekte koje sadržava taj objekt. Proces pretvorbe objekta u tok *byte*-ove naziva se **serijalizacija**. Kako serijalizirani objekt dolazi isključivo u obliku byte-ova, a ne znakova, `Object` tokovi ne dolaze u `Reader` i `Writer` inačicama.

Ukoliko se *byte*-ovi koji predstavljaju serijalizirani graf objekata čitaju pomoću metode **`readObject`** iz **`ObjectInputStream`** toka, deserijalizirani rezultat je graf objekata ekvivalentan "živom" objektu koji je bio serijaliziran.

Na primjer, ukoliko postoji **`HashMap`** objekt koji je potrebno spremiti u datoteku za kasniju upotrebu, graf objekata može se zapisati na ovaj način:

```
FileInputStream fileIn = new FileInputStream("tab");
ObjectInputStream in = new ObjectInputStream(fileIn);
HashMap newHash = (HashMap) in.readObject();
```

8.4.2. Omogućavanje serijaliziranja klasa

U slučaju kad **`ObjectOutputStream`** zapisuje serijalizirani neki objekt, taj objekt mora implementirati sučelje **`Serializable`**. Time se naznačava da je klasa iz koje je instanciran taj objekt predviđena za serijalizaciju.

Podrazumijevani (engl. *default*) proces serijalizacije serijalizira sva polja objekta osim onih koja su označena modifikatorom `transient` ili `static`. Primitivni tipovi i stringovi zapisani su pomoću istog *encodinga* koji koristi i `DataOutputStream`, dok se objekti serijaliziraju pozivanjem metode `writeObject`. Podrazumijevana serijalizacija također nalaže da nadklase klasa koje se serijaliziraju imaju konstruktore bez ulaznih parametara ili da također implementiraju sučelje `Serializable`. Primjer klase koja implementira sučelje `Serializable`:

```
public class Name implements java.io.Serializable {
    private String name;
    private long id;
    private transient boolean hashSet = false;//ne serijalizira se
    private transient int hash;//ne serijalizira se
    private static long nextID = 0;//ne serijalizira se

    public Name(String name) {
        this.name = name;
        synchronized (Name.class) {
            id = nextID++;
        }
    }


    public int hashCode() {
        if (!hashSet) {
            hash = name.hashCode();
            hashSet = true;

        }
        return hash;
```

```
        }
}
```

U nekim slučajevima klase se mogu generalno serijalizirati, međutim, neke posebne instance ne mogu se serijalizirati. Na primjer, zbirka objekata kao takva može se serijalizirati, ali ukoliko sadržava objekte koji se ne mogu serijalizirati, svaki pokušaj serijalizacije završio bi bacanjem iznimke **`NotSerializableException`**.

9. JAVA SWING

Apstraktni skup alata za razvijanje grafičkog sučelja (engl. *Abstract Window Toolkit* – AWT) dio je programskog jezika Java koji je dizajniran za izradu korisničkih sučelja i crtanje slika i grafika. Sastoji se od skupine klasa koje pružaju sve što je programerima potrebno da bi izradili grafičko sučelje za bilo koji Java aplet ili aplikaciju. Većina AWT komponenti nasljeđuje java.awt.Component klasu.

Swing predstavlja veliki skup komponenti od najjednostavnijih kao što je labela (engl. label) koja predstavlja tekstualni ili slikovni element grafičkog sučelja, pa sve do vrlo složenih komponenti kao što su tablice, stabla (engl. trees), te stilizirani tekstualni dokumenti (engl. styled textual documents). Gotovo sve komponente u Swingu nasljeđuju klasu `JComponent` koja nasljeđuje klasu `Component` iz AWT-a. Upravo zbog te činjenice Swing je najbolje opisati kao "sloj" iznad AWT-a, a ne kao njegovu zamjenu.

Svaka AWT komponenta ima svoju **ekvivalentnu komponentu u Swingu koja počinje slovom "J"**. Jedina iznimka je AWT klasa `Canvas` umjesto koje se mogu koristiti `JComponent`, `JLabel` ili `JPanel`. Međutim, ne vrijedi obrnuto, odnosno klase iz Swinga nemaju ekvivalentne klase u AWT-u.

Swing komponente su **nezahtjevne** (engl. *lightweight*), a AWT komponente su **zahtjevne** (engl. *heavyweight*). Jedna od razlika između nezahtjevnih i zahtjevnih komponenti je u dubini (engl. *depth*), odnosno, slojevitosti (engl. *layering*): svaka zahtjevna komponenta zadržava svoj "z-poredak" (z predstavlja treću dimenziju kod grafičkih sučelja – dubinu).

Svaka nezahtjevna komponenta nalazi se unutar zahtjevne komponente i zadržavaju svoju shemu slojevitosti koju definira Swing. Zahtjevne komponente unutar zahtjevnog **spremnika** (engl. *containter*) prekrivaju sve zahtjevne komponente pa se preporučuje **izbjegavanje istovremenog korištenja zahtjevnih i nezahtjevnih komponenata u istom spremniku**.

Jedna od važnijih svojstava Swinga je **neovisnost o platformi**, jer su sve komponente pisane u Javi. Kod AWT-a to nije slučaj, jer većina njegovih komponente ovise o komponentama platforme (engl. *peer components*). To znači da Swing komponente kao što je komponenta za unošenje teksta (engl. *text area*) identično izgledaju na Macintosh, Solaris, Linux i Windows platformama.

9.1. Paketi sa Swing komponentama

U nastavku poglavlja navedeni su paketi sa Swing komponentama, te kratki opis namjena klasa u tim paketima:

`javax.swing`

Paket sadrži većinu osnovnih Swing komponenti, podrazumjevanih modela komponenti (engl. *default component models*) i sučelja.

`javax.swing.border`

Paket sadrži klase i sučelja koje se koriste za definiranje specifičnih stilova rubova (engl. *border styles*). Rubove može dijeliti neodređeni broj Swing komponenti, jer rubovi zapravo nisu prave komponente.

`javax.swing.colorchooser`

Paket sadrži klase i sučelja koja podržavaju **`JColorChooser`** komponentu, koja se koristi za odabir boja.

`javax.swing.event`

Paket sadrži sve tipove događaja (engl. *event types*) i primatelje događaja (engl. *listeners*) koji su specifični za Swing. Osim toga Swing komponente podržavaju tipove događaja i primatelje događaja definiranih u paketima **`java.awt.event`** i **`java.beans`**.

`javax.swing.filechooser`

Paket sadrži klase i sučelja koje podržavaju komponentu **`JFileChooser`** koja se koristi za odabir datoteka (engl. *file selection*).

`javax.swing.plaf`

Paket sadrži programsko sučelje (engl. ***P**luggable **L**ook **A**nd **F**eel API*) koje se koristi za definiranje izgleda korisničkih komponenata grafičkog sučelja. Većina klasa u ovom paketu su apstraktne klase, a nasljeđuju i implementiraju ih komponente poput **`Metal`**, **`Motif`** ili **`Basic`**. Klase u ovom paketu predviđene su za korištenje u slučajevima kad programer ne želi koristiti postojeći izgled (engl. *look and feel*) komponenata.

`javax.swing.plaf.basic`

Paket sadržava osnovne implementacije izgleda (engl. *Basic look and feel implementation*) koji predstavlja temelj svih ostalih izgleda u Swingu. Ukoliko je potrebno definirati korisnički izgled, moraju se nasljeđivati klase iz ovog paketa.

`javax.swing.plaf.metal`

`Metal` je podrazumijevani (engl. *default*) izgled Swing komponenata (često se naziva i Java izgledom). To je jedini izgled komponenti u Swingu koji nije dizajniran da izgleda jednako na svim platformama.

`javax.swing.plaf.multi`

Paket sadrži multipleksirani izgled (engl. *Multiplexing look and feel*), koji nije uobičajena implementacija izgleda zbog toga što ne definira izgled komponenata. Umjesto toga omogućava simultano kombiniranje nekoliko izgleda. Na primjer, to može biti izgled koji se temelji na zvuku (engl. *audio-based*) u kombinaciji s **`Metal`** ili **`Motif`** izgledom.

`javax.swing.table`

Paket sadrži klase i sučelja koje podržavaju **`JTable`** kontrolu komponentu koja služi za prikazivanje podataka u tabličnom obliku. Komponenta je u velikoj mjeri prilagodljiva svim zahtjevima.

`javax.swing.text`

Paket sadržava klase i sučelja koje koriste tekstualne komponente, uključujući jednostavnije i složenije komponente, izgled tih komponenata, naglašavanje i sl.

`javax.swing.text.html`

Paket sadržava elemente za podržavanje parsiranje, kreiranje i pregled HTML dokumenata.

`javax.swing.text.rtf`

Paket sadržava elemente za podržavanje RTF (engl. *Rich Text Format*) dokumente.

`javax.swing.tree`

Paket sadržava klase i sučelja koja podržavaju **`JTree`** komponentu koja se koristi za prikaz i manipuliranje podataka u hijerarhijskom obliku.

`javax.swing.undo`

Paket sadržava elemente koji podržavaju implementaciju funkcionalnost poništavanja i ponavljanja postupka (engl. *undo/redo functionality*).

9.2. MVC arhitektura

MVC arhitektura (engl. *Model-View-Controller*) je jako poznata objektno-orijentirana arhitektura koja se koristi kod dizajniranja korisničkih sučelja (engl. *user interface*). Komponente sučelja podijeljene su u tri dijela: model (engl. *model*), izgled (engl. *view*) i kontroler (engl. *controller*). Svaka komponenta Swinga temelji se na tom dizajnu. U nastavku poglavlja objašnjena je pojedini dio arhitekture.

9.2.1. Model

Model predstavlja detalje vezane uz **stanje komponente**. Na primjer, vrijednost koja opisuje je li gumb (engl. *button*) pritisnut ili ne (engl. *pressed/unpressed*) ili tekstualni podaci upisani u tekstualnu komponentu. Model izravno ne koristi komponente izgleda i kontrolera, već njima komunicira pomoću događaja (engl. *events*).

9.2.2. Izgled

Izgled (engl. view) određuje vizualni izgled modela komponente, odnosno, izgled same komponente. Na primjer, izgled ispisuje boju komponente ili prikaz određenog fonta. Izgled je također odgovoran za ažuriranje izgleda komponenti na ekranu, na koji može utjecati model ili kontroler slanjem neizravnih poruka.

9.2.3. Kontroler

Kontroler (engl. *controller*) određuje hoće li komponenta reagirati na događaje (engl. *input events*) koje generiraju periferni uređaji (engl. *input devices*) kao što su tipkovnica i miš. Osim toga kontroler može primiti poruku od izgleda i neizravnu poruku od modela.

Na primjer, ukoliko korisnik označi *checkbox*, kontroler ustanovi da je korisnik kliknuo mišem i pošalje poruku izgledu. Ukoliko izgled ustanovi da se klik mišem dogodio na površini *checkbox*-a, pošalje poruku modelu. Model ažurira svoje stanje i pošalje poruku koju prima izgled, koji je također mora ažurirati na novo stanje (iscrtati kvačicu unutar *checkbox*-a).

Može se primijetiti da model nije vezan za određeni izgled ili kontroler, čime je omogućeno korištenje istog modela u više izgleda i kontrolera.

9.3. Obrada događaja

Događaj (engl. *event*) predstavlja svaku korisničku akciju na grafičkom sučelju poput pritiska tipke miša ili tipke na tipkovnici. Swing komponente također mogu generirati niz različitih tipova događaja, uključujući i one iz paketa **`java.awt.event`** i **`java.swing.event`**. Mnogi događaji iz paketa **`java.swing.event`** su vezani uz specifičnu komponentu. Svaki tip događaja predstavljen je objektom koji označava izvor događaja. Neki događaji nose dodatne informacije kao što su tip događaja i identifikator, kao i stanje izvora događa prije i poslije samog događaja.

Kako bi primili obavijest o događaju (engl. *notification of events*), potrebno je **povezati primatelje događaja** (engl. *event listeners*) **s objektima koji predstavljaju izvor događaja**. Primatelj događaja je implementacija svakog sučelja čije ime završava s "**`Listener`**", a prvi dio imena primatelja događaja opisuje tip događaja (npr. **`MouseListener`**). Sučelja koja implementiraju primatelji događaja definirana su u paketima **`java.awt.event`**, **`java.beans`** i **`javax.swing.event`**. U svakom sučelju postoji barem jedna metoda koja je vezana uz objekt koji završava s "**`Event`**" i definira tip događaja (npr. **`MouseEvent`**). Klase koje podržavaju neki događaj implementiraju odgovarajuće sučelje i podržavaju povezivanje i raskidanje (engl. *registering and unregistering*) primatelja događaja pomoću metoda npr. **`addMouseListener()`** i **`removeMouseListener()`**.

Većina izvora događaja dozvoljava povezivanje neograničenog broja primatelja događaja. Isto tako svako sučelje primatelja događaja može biti povezano s neograničenim brojem izvora događaja.

Na primjer, ukoliko želimo povezani gumb (izvor događaja) button s primateljem događaja **`ActionListener`**, potrebno je kreirati gumb, kreirati **`ActionListener`** i povezati ga s gumbom:

```
//kreiranje gumba
JButton button = new JButton();

//kreiranje ActionListenera
ActionListener act = new ActionListener() {
     public void actionPerformed(ActionEvent e) {
          System.out.println("Gumb je pritisnut!");
     }
};

//povezivanje izvora i primatelja događaja
button.addActionListener(act);
```

U slučaju pritiska gumba, ovaj jednostavni primatelj događaja će u konzolu ispisati poruku "**`Gumb je pritisnut!`**". Ovaj princip povezivanja primatelja i izvora događaja koristi se kod svih komponenti Swinga, razlikuju se samo vrste komponenti i vrste događaja.

9.4. Okviri i paneli u Swingu

Swing aplikacije izgrađene su od osnovnih komponenata programskog okvira (engl. *framework*) kao što su okviri i paneli koji će biti objašnjeni u nastavku poglavlja.

9.4.1. JFrame

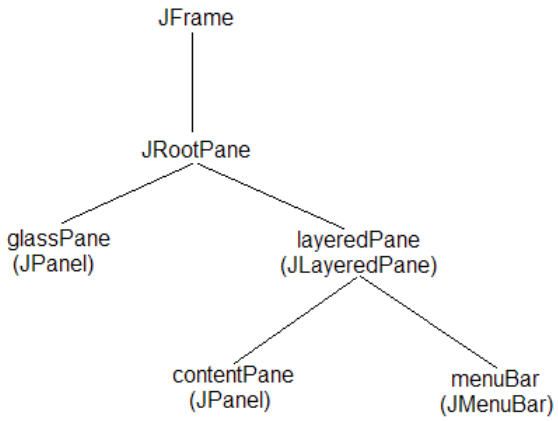
Glavni spremnik (engl. *container*) za aplikacije koje se temelje na Swingu je implementiran klasom **`javax.swing.JFrame`**. Komponente koje se dodaju **`JFrame`**-u zapravo se dodaju jedinom panelu koji je vezan uz njega, **`JRootPane`**, koji je osnovni panel kojem se mogu dodavati ostali paneli. Definiranje nekih osnovnih obilježja **`JFrame`**-a i dodavanje komponenata osnovnom panelu **`JRootPane`**-u moguće je izvesti sljedećim naredbama:

```
JFrame frame = new JFrame(); //instanciranje JFrame klase

//metoda za postavljanje veličine okvira
frame.setSize(300,300);

//metoda za postavljanje naslova na okviru
frame.setTitle("JFrame");

//metoda za postavljanje vidljivosti okvira
frame.setVisible(true);

//dodavanje JPanel komponente
frame.getContentPane().add(new JPanel());
```

Isto tako je moguće definirati i novi raspored komponenti (engl. *layout*) koje se nalaze unutar JFrame spremnika (npr. **`FlowLayout`**), detaljnije informacije o pojedinom rasporedu komponenti nalaze se u nastavku vježbe:

```
frame.getContentPane().setLayout(new FlowLayout());
```

Svaki **`JFrame`** spremnik sadržava svoj **`JRootPane`** panel koji se može dohvatiti pomoću metode **`getRootPane()`**. Na slici 9.1 prikazan je jednostavan graf koji opisuje kako je organizirana hijerarnije komponenti na primjeru koji će se opisivati u nastavku ovog poglavlja (sve komponente biti će detaljnije objašnjene u nastavku):

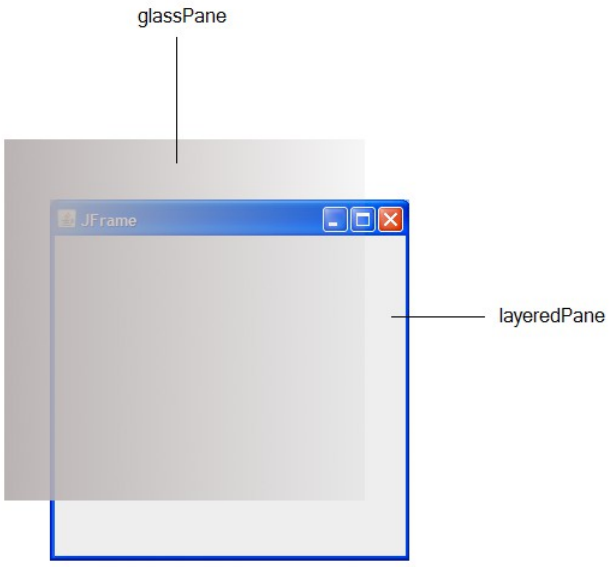


Slika 9.1. Primjer hijerarhije komponenti u Swing aplikaciji

Na slici je moguće uočiti da je **`JFrame`** spremnik koji sadrži panel **`JRootPane`**, a **`JRootPane`** sadržava sve ostale panele u koji se mogu nalaziti drugi paneli i komponente itd.

9.4.2. JRootPane

Svaki **`JRootPane`** panel (implementiran klasom **`javax.swing.JRootPane`**) sadrži nekoliko komponenti koje će biti opisane u sljedećem primjeru: **`glassPane`** (koji je zapravo instanca **`JPanel`** panela), **`layeredPane`** (instanca **`JLayeredPane`** panela), **`contentPane`** (instanca **`JPanel`** panela) i **`menuBar`** (instanca **`JMenuBar`** panela). Komponenta **`glassPane`** je postavljena iznad komponente **`layeredPane`** kako je i prikazano na slici 9.2.



Slika 9.2. Prikaz položaja komponente **`glassPane`** s obzirom na komponentu **`layeredPane`**

To može biti vrlo korisno ukoliko je potrebno "presresti" događaj mišem (engl. *mouse event*) ili je potrebno promijeniti trenutni fokus aplikacije (engl. *application focus*). Komponenta **`glassPane`** može biti bilo koja komponenta, a u podrazumijevanom (engl. *default*) slučaju je to instanca **`JPanel`** klase. Ukoliko je potrebno postaviti neku drugu

komponentu različitu od **`JPanel`** komponente, potrebno je pozvati sljedeću metodu:

```
frame.setGlassPane(drugaKomponenta);
```

Iako je komponenta **`glassPane`** iznad komponente **`layeredPane`**, nije vidljiva ukoliko se ne promijenje podrazumijevane postavke. Međutim, to je moguće promijeniti pozivanjem sljedeće metode:

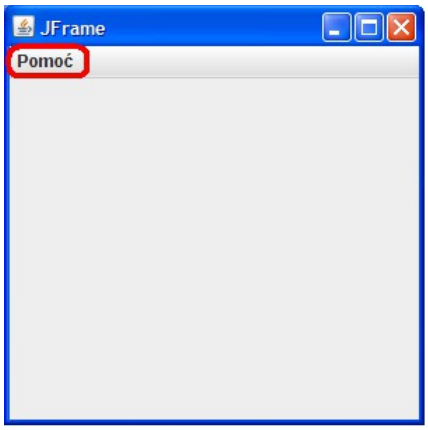
```
frame.getGlassPane().setVisible(true);
```

Time je omogućeno da **`glassPane`** panel može prikazivati komponente ispred sadržaja okvira **`JFrame`**.

Okviru **`JFrame`** također je moguće dodati izbornik **`JMenuBar`**, koji se pojavljuje iznad **`contentPane`** panela, pomoću sljedećeg programskoj koda:

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(new JMenu("Pomoć"));
frame.setJMenuBar(menuBar);
```

Prema grafičkom prikazu hijerarhije komponenata na slici 9.1, panel **`layeredPane`** sadrži **`menuBar`** i **`contentPane`**, gdje je "**`Pomoć`**" opcija unutar izbornika **`menuBar`**, a površina ispod izbornika zapravo **`contentPane`** koji je u ovom trenutku prazan:



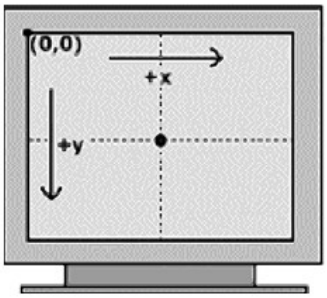
Slika 9.3. Prikaz **`menuBar`**-a i **`contentPane`**-a unutar okvira **`JFrame`**

Ukoliko je potrebno definirati novi raspored komponenti (engl. *layout*) unutar **`JRootPane`** panela, moguće je podrazumijevani organizator komponenti (engl. *default layout manager*) zamijeniti drugim organizatorom komponenti koji podržava pozicioniranje tih komponenti. Podrazumijevani organizator komponenti definiran je klasom **`javax.swing.JRootPane.RootLayout`** koja se nalazi unutar klase **`JRootPane`** (engl. *inner class*).

9.4.3. Definiranje mjesta postavljanja okvira na ekranu

Bez promjene podrazumijevanih postavki pozicije, okvir **`JFrame`** postavljen je u gornjem lijevom kutu ekrana. Ishodište koordinatnog sustava na ekranu za smještaj okvira se nalazi

upravo u gornjem lijevom kutu, a koordinate se povećavaju u pozitivnom smjeru prema dolje (os ordinata), te prema desno (os apscisa), kako je prikazano na slici 9.4.



Slika 9.4. Koordinatni sustav na ekranu za smještaj okvira

Jedan od najčešćih zahtjeva kod aplikacija je da se okvir postavi na sredinu ekrana. Korištenjem metode **`getToolkit()`** koju klasa **`JFrame`** nasljeđuje iz klase **`java.awt.Window`**, moguće je komunicirati s operacijskim sustavom računala. Metode unutar klase **`ToolKit`** koju vraća metoda **`getToolKit()`** tvore most između Java komponenata grafičkog sučelja i komponenata operacijskog sustava.

Metodom **`getScreenSize()`** moguće je dohvatiti dimenzije ekrana, te prema tome odrediti gdje se nalazi središte ekrana:

```
Dimension dim = frame.getToolkit().getScreenSize();
```

Da bi okvir smjestili upravo u središte ekrana, potrebno je izračunati polovicu širine ekrana i polovicu dužine ekrana, te dobivene vrijednosti oduzeti od koordinata središta ekrana:

```
frame.setLocation(dim.width/2 – frame.getWidth()/2,
          dim.height/2 – frame.getHeight/2);
```

9.4.4. Dekoriranje izgleda prozora

Od inačice Jave 1.4 omogućeno je dekoriranje izgleda prozora (engl. *look and feel window decorations*) kod okvira **`JFrame`** i prozora dijaloga **`JDialog`**. Da bi se to aktiviralo, potrebno je pozvati sljedeće statičke metode:

```
JFrame.setDefaultLookAndFeelDecorated(true);
JDialog.setDefaultLookAndFeelDecorated(true);
```

Nakon pozivanja tih metoda svaka nove instance **`JFrame`** i **`JDialog`** klasa imati će dekoriran izgled. Međutim, svaki okvir i prozor dijaloga koji su kreirani prije pozivanja navedenih metode neće biti promijenjeni.

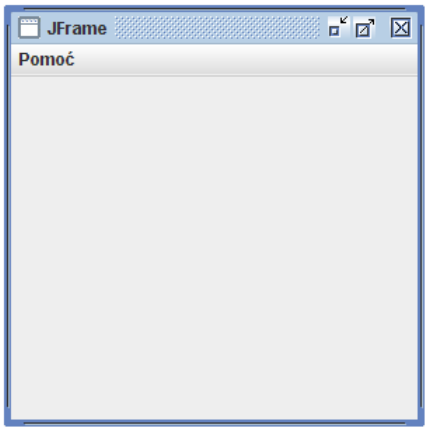
Ukoliko je potrebno dekoriranje primijeniti samo na jednom okviru ili prozoru dijaloga, to je moguće postići pozivanjem sljedećeg programskog koda:

```
frame.setUndecorated(true);
frame.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
```

Kod poziva metode **`setWindowDecorationStyle`** korištena je konstanta

`JRootPane.FRAME` pomoću koje je definiran stil dekoriranja. U Swingu i u Javi općenito, korištenje postojećih konstanti je vrlo raširena pojava (slično kao i kod konstanti koje se koriste kod klase **`Calendar`**).

Nakon aktiviranja navedene dekoracije prozora, jednostavan okvir iz poglavlja 9.3.2. poprimio bi sljedeći izgled:



Slika 9.5. Dekorirani izgled okvira

9.4.5. JWindow

Klasa **`javax.swing.JWindow`** predstavlja komponentu koja je vrlo slična okviru **`JFrame`**, samo što nema naslov (engl. *title bar*), ne podržava promjenu veličine (engl. *resize*), ne može se minimizirati (engl. *minimize*), maksimizirati (engl. *maximize*) i zatvoriti (engl. *close*).

Komponenta **`JWindow`** najčešće se koristi za ispisivanje privremene poruke korisniku aplikacije ili iscrtavanje slike koja predstavlja logo (engl. *splash screen logo*). Kako **`JWindow`** klasa implementira sučelje **`RootPaneContainer`**, moguće je manipulirati njen sadržaj isto kao i kod klase **`JFrame`**.

9.4.6. JPanel

Klasa **`javax.swing.JPanel`** predstavlja jednostavnu komponentu spremnika (engl. *simple container component*) koja se koristi za organiziranje grupe ili više grupa komponenata (engl. *child components*). **`JPanel`** je integralni dio **`JRootPane`** panela i koristi se u svakom primjeru u ovoj laboratorijskoj vježbi. Svim komponentama unutar **`JPanel`** spremnika upravlja organizator komponenti (engl. *layout manager*), koji određuje veličinu i poziciju svake komponente unutar panela. Podrazumijevani organizator komponenti kod **`JPanel`** panela je **`FlowLayout`** (detalji o organizatorima komponenti nalaze se u sljedećem poglavlju).

9.5. Rubovi komponenata u Swingu

Paket **`javax.swing.border`** sadržava klase koje definiraju nekoliko vrsti rubova komponenata (engl. *borders*) u Swingu:

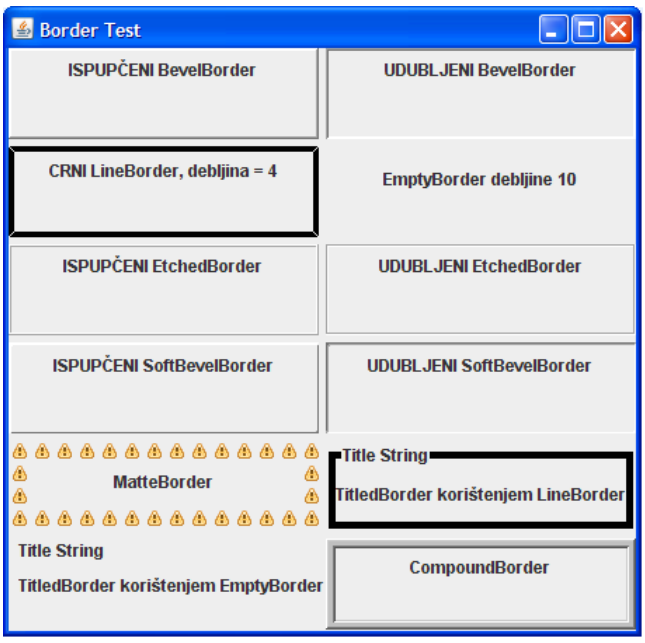
Naziv ruba	Opis ruba

`BevelBorder`	3-D rub koji može biti ispupčen ili udubljen.
`CompoundBorder`	Kombinacija dvaju rubova; vanjskog i unutarnjeg ruba.
`EmptyBorder`	Transparentni rub koji se koristi za definiranje praznog mjesta oko komponente.
`EtchedBorder`	Rub s izgledom bakroreza (engl. *eched line appearance*).
`LineBorder`	Ravni rub s određenom debljinom, bojom i mogućnošću zaobljivanja rubova.
`MatteBorder`	Rub koji se sastoji od niza slika (engl. tiled image) ili običan obojeni rub.
`SoftBevelBorder`	3-D rub koji može biti ispupčen ili udubljen, te imati oštre ili zaobljene rubove.
`TitledBorder`	Rub s naslovom na određenom mjestu, određene vrsta slova, boje slova itd.

Za postavljanje rubova Swing komponentama potrebno je pozvati samo metodu **`setBorder()`** iz klase **`JComponent`** koju nasljeđuju sve komponente u Swingu. Osim toga postoji i klasa **`javax.swing.BorderFactory`** koja sadrži niz statičkih metoda za brzo kreiranje rubova komponenti. Na primjer, ukoliko želimo kreirati rub tipa **`EtchedBorder`**, koristili bi sljedeću metodu:

```
component.setBorder(BorderFactory.createEtchedBorder());
```

Klase koje predstavljaju rubove ne sadržavaju metode za postavljanje željenih svojstava kao što su dimenzija ili boja ruba. Umjesto da se modificiraju postojeći rubovi, uobičajeno je da se kreiraju novi i oni zamijene stari rub koji je potrebno zamijeniti. Na slici 9.6. prikazani su rubovi koje je moguće kreirati, a nakon slike dan je cijeli programski kod koji kreira prikazane rubove.



Slika 9.6. Primjeri rubova

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderTest extends JFrame {

     public BorderTest() {

          setTitle("Border Test");
          setSize(455, 450);

          JPanel content = (JPanel) getContentPane();
          content.setLayout(new GridLayout(6, 2, 5, 5));

          JPanel p = new JPanel();
          p.setBorder(new BevelBorder (BevelBorder.RAISED));
          p.add(new JLabel("ISPUPČENI BevelBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new BevelBorder (BevelBorder.LOWERED));
          p.add(new JLabel("UDUBLJENI BevelBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new LineBorder (Color.black, 4, true));
          p.add(new JLabel("CRNI LineBorder, debljina = 4"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new EmptyBorder (10,10,10,10));
          p.add(new JLabel("EmptyBorder debljine 10"));
          content.add(p);

          p = new JPanel();
          p.setBorder(
                          new EtchedBorder(EtchedBorder.RAISED));
          p.add(new JLabel("ISPUPČENI EtchedBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(
                         new EtchedBorder(EtchedBorder.LOWERED));
          p.add(new JLabel("UDUBLJENI EtchedBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(
```

```
                    new SoftBevelBorder(SoftBevelBorder.RAISED));
          p.add(new JLabel("ISPUPČENI SoftBevelBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(
                    new SoftBevelBorder(SoftBevelBorder.LOWERED));
          p.add(new JLabel("UDUBLJENI SoftBevelBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new MatteBorder
               (new ImageIcon("uslicnik.gif")));
          p.add(new JLabel("MatteBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new TitledBorder (
                    new LineBorder (Color.black, 5),
                    "Title String"));
          p.add(new JLabel(
                         "TitledBorder korištenjem LineBorder"));
          content.add(p);

          p = new JPanel();
          p.setBorder(new TitledBorder(
               new EmptyBorder(0, 0, 0, 0), "Title String"));
          p.add(new JLabel(
                        "TitledBorder korištenjem EmptyBorder"));
          content.add(p);

          Color c1 = new Color(86, 86, 86);
          Color c2 = new Color(192, 192, 192);
          Color c3 = new Color(204, 204, 204);

          Border b1 = new BevelBorder(
                                     EtchedBorder.RAISED, c3, c1);
          Border b2 = new MatteBorder(3,3,3,3,c2);
          Border b3 = new BevelBorder(
                                    EtchedBorder.LOWERED, c3, c1);
          p = new JPanel();
          p.setBorder(new CompoundBorder(
               new CompoundBorder(b1, b2), b3));
          p.add(new JLabel("CompoundBorder"));
          content.add(p);
     }
     public static void main(String args[]) {
          BorderTest frame = new BorderTest();

             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setVisible(true);
     }
```

```
}
```

9.6. Organizatori komponenti

Svi organizatori komponenti (engl. *layout managers*) implementiraju jedno od dva sučelja iz **`java.awt`** paketa: **`LayoutManager`** i **`LayoutManager2.LayoutManager`**. Kod definiranja organizatora komponenti u pojedinim spremnicima komponenti (engl. container), potrebno je držati se sljedećih pravila: **svaki spremnik komponenata može imati samo jedan organizator komponenti, a svaki organizator komponenti smije biti dodijeljen samo jednom spremniku komponenti**.

9.6.1. LayoutManager

Svaki organizator komponenti mora implementirati apstraktno sučelje **`java.awt.LayoutManager`**, koji sadržava metode od kojih su najvažnije:

`layoutContainer(Container parent)`:

Izračunava i postavlja optimalnu veličinu komponenata u spremniku u kojem se nalaze.

`preferredLayoutSize(Container parent)`:

Izračunava i postavlja optimalnu veličinu komponenata u spremniku u kojem se nalaze i vraća instancu klase **`Dimension`** klase koja opisuje tu veličinu.

9.6.2. LayoutManager2

Apstraktno sučelje **`java.awt.LayoutManager2`** nasljeđuje sučelje **`LayoutManager`** kako bi omogućilo stvaranje programskog okvira (engl. *framework*) za organizatore komponenata koji koriste organizaciju s ograničenjima (engl. *constraints-based layouts*).

Metoda **`addLayoutComponent`**(**`Component comp, Object constraints`**) dodaje novu komponentu i objekt koji opisuje ograničenja kod dodavanja te komponente u spremnik.

Karakteristična implementacija tog sučelja je klasa **`BorderLayout`** koja zahtijeva informaciju u smjeru pozicioniranja komponente (npr. sjever ili zapad). U tom slučaju objekt koji opisuje ograničenja je zapravo **`String`** koji može biti npr. **`BorderLayout.NORTH`** ili **`BorderLayout.WEST`**.

9.6.3. BoxLayout

Organizator **`javax.swing.BoxLayout`** organizira komponente po X-osi i Y-osi

koordinatnog sustava panela. Jedini konstruktor, **`BoxLayout(Container target, int axis)`**, uzima referencu komponente **`Container`** koju organizira, te smjer (**`BoxLayout.X_AXIS`** ili **`BoxLayout.Y_AXIS`**).

9.6.4. Box

Za lakše korištenje organizatora **`BoxLayout`**, Swing sadrži klasu **`Box`** koja predstavlja organizator koji automatski ima postavljen **`BoxLayout`** organizator. Za kreiranje organizatora tipa **`Box`**, u konstruktor je potrebno predati samo željeno poravnanje (engl. *alignment*) kao parametar konstruktora. **`Box`** organizator također podržava dodavanje tzv. nevidljivih blokova (engl. *invisible blocks*), odnosno instanci klase **`Box.Filler`** (objašnjeno u sljedećem poglavlju), koji omogućavaju definiranje veličine prostora koji neće biti iskorišten. To su obično komponente s veličinom i mjestom, ali bez izgleda.

9.6.5. Filler

Statička ugnježđena (engl. *inner*) klasa **`javax.swing.Box.Filler`** definira nevidljive komponente koje utječu na izgled spremnika. Klasa **`Box`** sadrži nekoliko korisnih statičkih metoda za kreiranje triju različitih varijacija: zalijepljeno područje (engl. *glue area*), razvučeno područje (engl. *struts area*) i kruto područje (engl. *rigid area*).

Metode **`createHorizontalGlue()`** i **`createVerticalGlue()`** vraćaju komponentu koja popunjava područje između njenih susjednih komponenata tako da ih "gura od sebe" kako bi zauzelo svo područje koje može zauzeti.

Metode **`createHorizontalStrut(int width)`** i **`createVerticalStrut(int height)`** vraćaju komponentu fiksne širine (visine) koja predstavlja fiksan razmak između njenih susjednih komponenti.

Metoda **`createRigidArea(Dimension d)`** vraća nevidljivu komponentu fiksne širine i visine.

Sve navedene metode su statičke i mogu biti dodane svakom spremniku koji je organiziran pomoću **`BoxLayout`** organizatora, a ne samo instance klase **`Box`**.

9.6.6. FlowLayout

Organizator predstavljen klasom **`java.awt.FlowLayout`** jednostavan je organizator koji postavlja komponente slijeva nadesno u retke koristeći preferirane veličine komponenti (engl. *preferred component size*) koje vraća metoda **`getPreferredSize()`**. Nakon popunjavanja jednog retka, organizator nastavlja dodavati komponente u sljedeći redak. Zbog toga što postavljanje komponenti u spremnik ovisi o veličini samog spremnika, nije uvijek moguće predvidjeti u koji redak će se staviti koja komponenta.

Organizator **`FlowLayout`** je prejednostavan da bi se koristio u nekim složenijim aplikacijama u kojima želimo biti sigurni, na primjer, da će svi potrebni gumbi biti na dnu prozora za dijalog, a ne na njegovoj desnoj strani. Međutim, ponekad može biti koristan ukoliko je neku komponentu potrebno postaviti u sredinu spremnika. **`FlowLayout`** je podrazumijevani organizator u svim **`JPanel`** komponentama (jedina iznimka je **`JRootPane`** kojem je podrazumijevani organizator **`BorderLayout`**).

9.6.7. GridLayout

Organizator predstavljen klasom **`java.awt.GridLayout`** postavlja komponente u pravokutnu mrežu. Postoje tri konstruktora:

`GridLayout()`:

Kreira organizator koji koristi samo jedan redak i jedan stupac.

`GridLayout(int rows, int cols)`:

Kreira organizator koji koristi zadani broj redaka i stupaca.

`GridLayout(int rows, int cols, int hgap, int vgap)`:

Kreira organizator koji koristi zadani broj redaka i stupaca, te postavlja zadani razmak između svakog retka i stupca komponenata.

`GridLayout` postavlja komponente slijeva na desno i od vrha prema dnu, s tim da svakoj komponenti dodjeljuje jednaku veličinu. Nastoji zauzeti cijelo dostupno područje u spremniku pri čemu svakoj komponenti dodjeljuje jednako područje. Ukoliko se ne koristi pažljivo, moguće je dobiti nepredviđene veličine komponenata, kao na primjer, komponenta za unos teksta može postati tri puta viša od očekivane visine.

9.6.8. GridBagLayout

Organizator predstavljen klasom **`java.awt.GridBagLayout`** nasljeđuje svojstva klase **`GridLayout`**, te dodaje ograničenja koja se predstavljaju klasom **`java.awt.GridBagConstraints`**. Mjesto unutar spremnika podijeljeno je na jednake pravokutne dijelove (poput cigli unutar zida), te svaku komponentu postavlja na mjesto jednog ili više tih pravokutnih dijelova. Osim toga potrebno je popuniti objekt klase **`GridBagConstraints`** za svaku komponentu koji opisuje na koji način je potrebno staviti komponentu u spremnik.

`GridBagLayout` može efektivno biti iskorišten za postavljanje komponenti ukoliko se ne postavlja nikakvo ograničenje na veličinu komponenti. Međutim, kako je ovaj organizator prilično kompleksan, obično zahtijeva neke pomoćne metode i klase kako bi se opisala sva potrebna ograničenja.

9.6.9. BorderLayout

Organizator **`java.awt.BorderLayout`** podijeli spremnik u pet područja: centralno područje (engl. *center*), sjeverno područje (engl. *north*), južno područje (engl. *south*), istočno područje (engl. *east*) i zapadno područje (engl. *west*). Za određivanje područja u

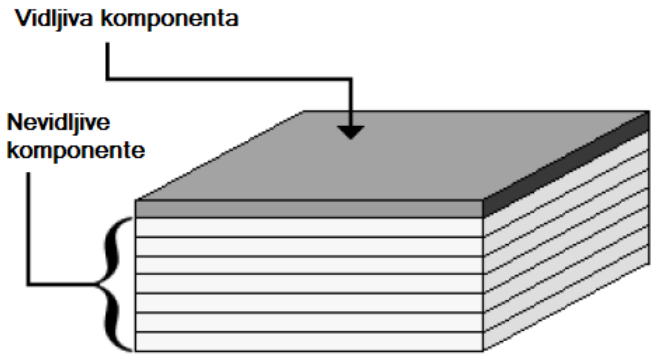
koji se želi postaviti komponenta koriste se nizovi (engl. *String*) u obliku "**`Center`**", "**`North`**" itd. Osim toga moguće je koristiti i statičke String konstante definirane unutar klase **`BorderLayout`** (**`BorderLayout.CENTER`**, **`BorderLayout.NORTH`** itd.).

Tijekom procesa organiziranja prednost kod zauzimanja mjesta u spremniku imaju komponente koje se nalaze u sjevernim i južnim područjima, te će im se dodijeliti preferirana visina i širina (ukoliko je to moguće). Nakon što komponente u sjevernim i južnim područjima zauzmu svoje mjesto, komponente u istočnim i zapadnim područjima će također pokušati zauzeti potreban prostor za smještaj koji je ostao između sjevernog i južnog područja. Na kraju će komponente u centralnom području također zauzeti preostalo mjesto u spremniku.

Organizator **`BorderLayout`** je vrlo koristan, osobito u kombinaciji s ostalim organizatorima, kao što će biti objašnjeno u sljedećim poglavljima.

9.6.10. CardLayout

Organizator **`java.awt.CardLayout`** tretira sve komponente slična kartama iste veličine koje jedna drugu preklapaju. U svakom trenutku je vidljiva samo jedna karta, odnosno, komponenta, kao što je prikazano na sljedećoj slici:



Slika 9.7. Komponente kod organizatora CardLayout

Metode **`first()`**, **`last()`**, **`next()`**, **`previous()`** i **`show()`** mogu se pozvati kako bi se napravile zamjene između komponenata u spremniku.

9.6.11. SpringLayout

Organizator **`javax.swing.SpringLayout`** organizira komponente prema zadanom skupu ograničenja (četiri ograničenja za svaku komponentu u spremniku), pri čemu je svako ograničenje definirano u objektu **`javax.swing.Spring`**.

Instanca klase **`SpringLayout.Constraints`** koristi se kao općenito ograničenje prilikom dodavanja komponente u spremnik koji je organiziram prema **`SpringLayout`**-u u sljedećem programskom kodu:

```
container.setLayout(new SpringLayout());
```

```
container.add(new JButton("Button"),
new SpringLayout.Constraints(
     Spring.constant(10),
     Spring.constant(10),
     Spring.constant(120),
     Spring.constant(70)));
```

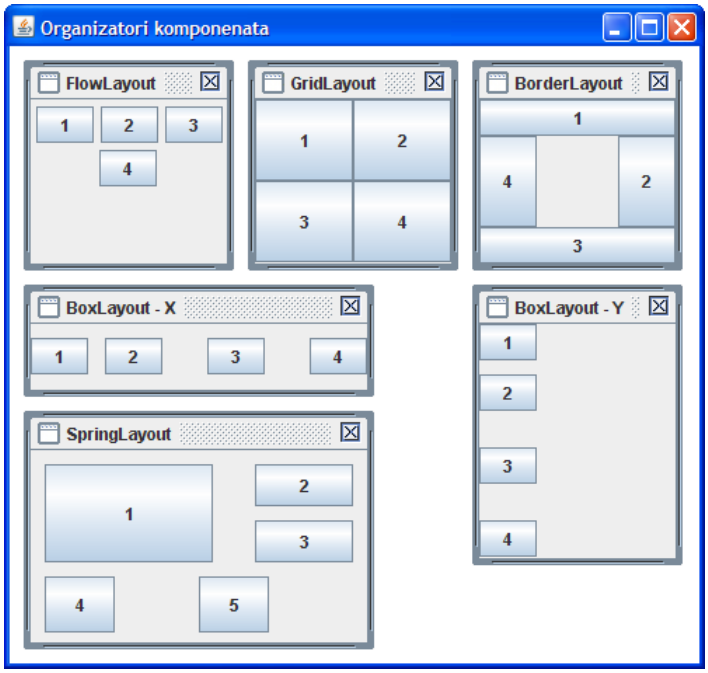
Četiri ulazna parametra `SpringLayout.Constraints` konstruktora su četiri `Spring` objekta, u ovom slučaju kreirana pomoću statičke `constants()` metode kako bi odredila minimalne, maksimalne i preferirane vrijednosti svakog ograničenja. Prvi parametar određuje lokaciju komponente po X-osi, drugi parametar određuje lokaciju komponente po Y-osi, treći parametar predstavlja širinu komponente, a četvrti parametar predstavlja visinu komponente.

9.6.12. JPanel

Klasa `javax.swing.JPanel` predstavlja generički jednostavan spremnik (engl. *generic lightweight container*), koji je vrlo važan faktor kod funkcioniranja organizatora komponenti. Podrazumijevani konstruktor `JPanel` klase postavlja `FlowLayout` organizator, međutim, moguće je postaviti i druge organizatore pomoću `setLayout()` metode.

9.7. Usporedba najčešćih organizatora komponenti

U sljedećem primjeru korišteni su organizatori komponenti koji se najčešće koriste kod razvijanja Swing aplikacija. Primjer pokazuje skupinu `JInternalFrame` okvira koji sadrže identične komponente, međutim, svaki od njih koristi različiti organizator komponenti. Svrha ovog primjera je omogućavanje izravne usporedbe korištenjem spremnika s promjenjivom veličinom. Na slici 9.8. nalazi se izgled okvira koji nastaje pokretanjem programa u nastavku:



Slika 9.8. Okvir s različitim organizatorima komponenti

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;

import javax.swing.*;

public class LayoutManagerTest extends JFrame {

     public Integer LAYOUT_FRAME_LAYER = new Integer(1);

     public LayoutManagerTest() {

          super("Organizatori komponenata");
          setSize(500, 460);
          JDesktopPane desktop = new JDesktopPane();
          getContentPane().add(desktop);

          JInternalFrame fr1 =
               new JInternalFrame("FlowLayout", true, true);

          fr1.setBounds(10, 10, 150, 150);

          Container c = fr1.getContentPane();
          c.setLayout(new FlowLayout());
          c.add(new JButton("1"));
          c.add(new JButton("2"));
          c.add(new JButton("3"));
          c.add(new JButton("4"));
          desktop.add(fr1, 0);
          fr1.show();

          JInternalFrame fr2 =
                new JInternalFrame("GridLayout", true, true);

          fr2.setBounds(170, 10, 150, 150);

          c = fr2.getContentPane();
          c.setLayout(new GridLayout(2, 2));
          c.add(new JButton("1"));
          c.add(new JButton("2"));
          c.add(new JButton("3"));
          c.add(new JButton("4"));
          desktop.add(fr2, 0);
          fr2.show();

          JInternalFrame fr3 =
              new JInternalFrame("BorderLayout", true, true);

          fr3.setBounds(330, 10, 150, 150);

          c = fr3.getContentPane();
          c.add(new JButton("1"), BorderLayout.NORTH);
          c.add(new JButton("2"), BorderLayout.EAST);
          c.add(new JButton("3"), BorderLayout.SOUTH);
          c.add(new JButton("4"), BorderLayout.WEST);
```

```
        desktop.add(fr3, 0);
        fr3.show();

        JInternalFrame fr4 =
           new JInternalFrame("BoxLayout - X", true, true);
        fr4.setBounds(10, 170, 250, 80);

        c = fr4.getContentPane();
        c.setLayout(new BoxLayout(c, BoxLayout.X_AXIS));
        c.add(new JButton("1"));
        c.add(Box.createHorizontalStrut(12));
        c.add(new JButton("2"));
        c.add(Box.createGlue());
        c.add(new JButton("3"));
        c.add(Box.createHorizontalGlue());
        c.add(new JButton("4"));
        desktop.add(fr4, 0);
        fr4.show();
        JInternalFrame fr5 =
            new JInternalFrame("BoxLayout - Y",true, true);

        fr5.setBounds(330, 170, 150, 200);

        c = fr5.getContentPane();
        c.setLayout(new BoxLayout(c, BoxLayout.Y_AXIS));
        c.add(new JButton("1"));
        c.add(Box.createVerticalStrut(10));
        c.add(new JButton("2"));
        c.add(Box.createGlue());
        c.add(new JButton("3"));
        c.add(Box.createVerticalGlue());
        c.add(new JButton("4"));
        desktop.add(fr5, 0);
        fr5.show();

        JInternalFrame fr6 =
            new JInternalFrame("SpringLayout", true, true);

        fr6.setBounds(10, 260, 250, 170);

        c = fr6.getContentPane();
        c.setLayout(new SpringLayout());
        c.add(new JButton("1"), new
             SpringLayout.Constraints(
                       Spring.constant(10),
                       Spring.constant(10),
                       Spring.constant(120),
                       Spring.constant(70)));
        c.add(new JButton("2"), new
             SpringLayout.Constraints(
                       Spring.constant(160),
        Spring.constant(10),
        Spring.constant(70),
        Spring.constant(30)));
```

```
            c.add(new JButton("3"), new
                        SpringLayout.Constraints(
                                    Spring.constant(160),
                                    Spring.constant(50),
                                    Spring.constant(70),
                                    Spring.constant(30)));
            c.add(new JButton("4"), new
                        SpringLayout.Constraints(
                              Spring.constant(10),
                              Spring.constant(90),
                              Spring.constant(50),
                              Spring.constant(40)));
                        c.add(new JButton("5"),
                             new SpringLayout.Constraints(
                                    Spring.constant(120),
                                    Spring.constant(90),
                                    Spring.constant(50),
                                    Spring.constant(40)));
                        desktop.add(fr6, 0);
                        fr6.show();
                        desktop.setSelectedFrame(fr6);
            }

      public static void main(String argv[]) {
            LayoutManagerTest frame = new LayoutManagerTest();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
}
```

Primjer najsloženih organizatora komponenti, **`GridBagLayout`**-a, nalazi se na kraju u poglavlju "Dodaci".

9.8. Labele i gumbi

9.8.1. JLabel

Klasa **`javax.swing.JLabel`** predstavlja jednu od najjednostavnijih Swing komponenti i najčešće se koristi za dodjeljivanje značenja drugim komponentama. **`JLabel`** može prikazivati tekst, ikonu ili oboje u svim kombinacijama pozicioniranja.

Sljedeći primjer opisuje kako kreirati četiri različite labele (engl. *label*) i staviti ih u organizator komponenti **`GridLayout`** kao što je prikazano na slici 7.9:

9.9. Četiri različite labele

Prikazani prozor s labelama mogu će dobiti izvođenjem sljedećeg programskog koda:

```
package hr.tvz.programiranje.java.labosi.swing;

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

public class JLabelTest extends JFrame  {

     public JLabelTest() {

          super("JLabel Demo");
          setSize(600, 150);
          JPanel content = (JPanel) getContentPane();
          content.setLayout(new GridLayout(1, 4, 4, 4));
          JLabel label = new JLabel();
          label.setText("JLabel");
          label.setBackground(Color.white);
          content.add(label);
          label = new JLabel("JLabel",SwingConstants.CENTER);
          label.setOpaque(true);
          label.setBackground(Color.white);
          content.add(label);
          label = new JLabel("JLabel");
          label.setFont(new Font("Helvetica", Font.BOLD,18));
          label.setOpaque(true);
          label.setBackground(Color.white);
          content.add(label);
          ImageIcon image = new ImageIcon("javalogo.gif");
          label = new JLabel("JLabel", image,
                    SwingConstants.RIGHT);
          label.setVerticalTextPosition(SwingConstants.TOP);
          label.setOpaque(true);
          label.setBackground(Color.white);
          content.add(label);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
     }

     public static void main(String args[]) {
          new JLabelTest();
     }
}
```

Prva labela u primjeru kreirana je pomoću podrazumijevanog konstruktora (engl. *default constructor*) i korištenjem metode **`setText()`**. Usprkos tome što je boja pozadine postavljena na bijelu, u prikazanom prozoru pozadina je siva. To se događa zbog toga što labela nije proglašena "neprozirnom" (engl. *opaque*). Ukoliko komponente nisu neprozirne, neće popuniti svoju pozadinu.

Labelama je moguće definirati font (engl. *font*) i boju (engl. *foreground color*). Osim toga je moguće postaviti i poravnanje (engl. *alignment*). Podrazumijevano poravnanje za tekst je lijevo, a za slike centrirano.

Poravnanje slika i tekstova je moguće definirati korištenjem konstanti iz **`javax.swing.SwingConstants`** sučelja unutar konstruktora ili metoda **`setHorizontalAlignment()`** i **`setVerticalAlignment()`**:

```
SwingConstants.LEFT
SwingConstants.CENTER
SwingConstants.RIGHT
SwingConstants.TOP
SwingConstants.BOTTOM
```

Tekst može biti poravnan horizontalno ili vertikalno, neovisno o slici (tekst će prekriti sliku ukoliko je to potrebno) pomoću metoda **`setHorizontalTextAlignment()`** i **`setVerticalTextAlignment()`**.

9.8.2. AbstractButton

Apstraktna klasa **`javax.swing.AbstractButton`** je temeljna klasa za sve gumbe koji su definirani u Swingu: gumbi za pritiskanje (engl. *push buttons*), preklopni gumbi (engl. *toggle buttons*), simboli za označavanje (engl. *check boxes*), radio gumbi (engl. *radio buttons*) i izbornici (engl. *menu items and menus*).

Klase koje nasljeđuju klasu **`AbstractButton`** su **`JButton`**, **`JToggleButton`** i **`JMenuItem`**. Osim tih klasa **`JButton`** nema nijednu podklasu, **`JToggleButton`** ima podklase **`JCheckBox`** i **`JRadioButton`**, a klasa **`JMenuItem`** ima tri podklase: **`JCheckBoxMenuItem`**, **`JRadioButtonMenuItem`** i **`JMenu`**.

9.8.3. Sučelje ButtonModel

Svaka klasa koja predstavlja gumb koristi model za spremanje informacija o stanju tog gumba koji se može dohvatiti i postaviti metodama **`getModel()`** i **`setModel()`** iz klase **`AbstractButton`**. Sučelje **`javax.swing.ButtonModel`** predstavlja temeljnu klasu u kojoj su definirani svi modelu gumbiju.

Stanje gumba predstavljeno je sljedećim svojstvima boolean tipa, koje imaju metode za dohvaćanje i postavljanje (npr. **`isSelected()`** i **`setSelected()`**):

`selected`: promjena stanja na svaki klik (koristi se samo kod **`JToggleButtons`** gumba – *checkbox* i *radio button*)

`pressed`: vraća **`true`** kada je gumb pritisnut dolje pomoću miša.

`rollover`: vraća **`true`** ukoliko je kursor miša iznad gumba.
`armed`: zaustavlja generiranje događanja (engl. *events*) kad se gumb pritisne mišem, te otpusti tipka miša kada se kursor nalazi izvan površine gumba.

`enabled`: vraća true kada je gumb aktivan, a kada gumb nije aktivan, nije moguće mijenjati njegova svojstva.

9.8.4. JButton

Klasa **`javax.swing.JButton`** predstavlja osnovni gumb za pritiskanje (engl. *push button*) i jedna je od najjednostavnijih Swing komponenata. Gotovo sva svojstva koja vrijeme za labelu **`JLabel`**, vrijede i za gumb **`JButton`**: moguće je dodavati slike, definirati poravnanje teksta i slike, postavljati boje (engl. *foreground and background colors*) i postavljati fontove. Osim toga moguće je dodavati **`ActionListener`**, **`ChangeListener`** i **`Itemlistener`** primatelje događaja (engl. *event listeners*) koji će biti objašnjeni u nastavku poglavlja.

Ukoliko je potrebno definirati koji od gumbiju na okviru mora biti označen (fokusiran) da bi ga mogli pritisnuti pritiskom tipke "Enter" s tipkovnice, potrebno je pozvati metodu **`setDefaultButton()`** iz klase **`JRootPane`**:

```
frame.getRootPane().setDefaultButton(myButton);
```

Za kreiranje **`ActionListener`** primatelja događaja potrebno je kreirati klasu koja implementira sučelje **`ActionListener`** i definira metodu **`actionPerformed()`**. Nakon toga moguće je dodijeliti taj primatelj događaja gumbu pomoću metode **`addActionListener()`** iz klase **`JComponent`** koju **`JButton`** nasljeđuje. Sljedeći dio koda je uobičajena implementacija ugnježđene (engl. *inner*) klase. U trenutku kad se primi događaj **`ActionEvent`**, na standardni izlaz ispisuje se tekst "Gumb je pritisnut!":

```
JButton button = new Jbutton();

ActionListener act = new ActionListener() {
     public void actionPerformed(ActionEvent e) {
          System.out.println("Gumb je pritisnut!");
     }
};

button.addActionListener(act);
```

Korištenje ugnježđene klase preporuča je kad svaka **`ActionListener`** klasa vezana uz samo jednu komponentu, u suprotnom je bolje kreirati zasebnu klasu u zasebnoj datoteci.

Gumb može biti onemogućen (engl. *disabled*) ili omogućen (engl. *enabled*) kao i labele, što je moguće postaviti pomoću metode **`setEnabled()`**. Onemogućeni gumb ne reagira na nikakve akcije korisnika.

Osim pritiskom miša, gumb je moguće aktivirati i tipkovnicom u slučaju kad mu se dodijeli slovo s tipkovnice (engl. *keyboard mnemonic*) pomoću metode **`setMnemonic()`**:

```
button.setMnemonic('R');
```

U tom slučaju navedeni gumb moguće je osim mišem aktivirati i pritiskom na kombinaciju tipku ALT + 'R'. Prvo pojavljivanje dodijenjenog slova ako postoji, biti će podcrtano u tekstu koji se nalazi na gumbu.

9.8.5. JToggleButton

Klasa **`javax.swing.JToggleButton`** predstavlja mehanizam za označavanje stanja komponenti koje nasljeđuju tu klasu (**`JCheckBox`** i **`JRadioButton`**), konkretno, svojstvo **`selected`** koje označava je li komponenta označena ili nije. Stanje gumba može se provjeriti metodom **`isSelected()`**, a postaviti pomoću metode **`setSelected()`**.

9.8.6. ButtonGroup

Komponente **`JToggleButton`** najčešće se koriste u komponentama **`ButtonGroup`** koje su predstavljenje klasom **`javax.swing.ButtonGroup`**. **`ButtonGroup`** upravlja skupinom gumbiju na način da dopušta označavanje samo jednog gumba u svakom trenutku (nije moguće označiti više gumbiju odjednom). To svojstvo ima smisla samo s **`JToggleButton`** gumbima (**`JCheckBox`** i **`JRadioButton`**).

Sljedeći prozor sadržava četiri **`JToggleButton`** komponente pri čemu ne mogu biti označene istovremeno dvije komponente:

Slika 9.10. Četiri **`JToggleButton`** komponente

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;

import javax.swing.*;
public class ToggleButtonTest extends JFrame {

     public ToggleButtonTest () {
          super("ToggleButton Test");
          getContentPane().setLayout(new FlowLayout());
          ButtonGroup buttonGroup = new ButtonGroup();
          int k = 0;
          for (; k < 4; k++) {
               char ch = (char) ('1'+ k);
               JToggleButton button = new JToggleButton(
                    "Button " + ch, k == 0);
               button.setMnemonic(ch);
               button.setEnabled(k < 3);
               button.setToolTipText(
                                          "This is button " + ch);
               button.setIcon(
                      new ImageIcon("C://not_selected.gif"));
               button.setSelectedIcon(
                    new ImageIcon("C://selected.gif"));
               button.setRolloverIcon(
                    new ImageIcon("C://rollover.gif"));
```

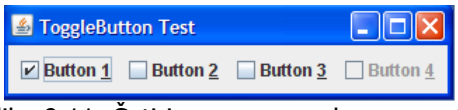
```
                button.setRolloverSelectedIcon(
                            new ImageIcon(
                                        "C://rolloverSelected.gif"));
                getContentPane().add(button);
                buttonGroup.add(button);
          }
          pack();
     }

     public static void main(String args[]) {
          ToggleButtonTest frame = new ToggleButtonTest();
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
     }
}
```

9.8.7. JCheckBox i JRadioButton

Klase **javax.swing.JCheckBox** i **javax.swing.JRadioButton** nasljeđuju sve funkcionalnosti klase **JToggleButton**. Jedina razlika između ove tri komponente je njihov izgled, odnosno, prikaz na ekranu. Ako se programski kod iz prošlog poglavlja modificira tako da se liniju za dodavanje **JToggleButton** komponenata zamijeni s linijom za dodavanje **JCheckBox** komponenata

```
JToggleButton button = new JCheckBox("Button " + ch, k == 0);
```

i nakon toga izbace četiri linije koje pridjeljuju ikone gumbima, prikazuje se sljedeći okvir:



Slika 9.11. Četiri **JCheckBox** komponente

Na sličan način možemo dobiti i četiri **JRadioButton** komponente, ako liniju za ubacivanje **JToggleButton** komponenti zamijenimo linijom:

```
JToggleButton button = new JRadioButton("Button " + ch, k == 0);
```

Slika 9.12. Četiri **JRadioButton** komponente

Međutim, ukoliko nije potrebno korištenje svojstva da samo jedan gumb smije biti označen, nema potrebe koristiti **ButtonGroup** komponentu.

9.9. Tabovi

Klasa **javax.swing.JTabbedPane** predstavlja komponentu koja sadržava skupine komponenti podijeljene u slojeve koji se mogu odabrati. Svaki sloj može sadržavati jednu komponentu koja je najčešće spremnik (engl. *container*). Proširenja tabova (engl. *tab extensions*) koriste se za postavljanje odabranog taba na površinu, odnosno, postavljanje

tog sloja na vrh. Proširenja tabova su vrlo slična labelama jer mogu sadržavati neki tekst, ikonu i boju.

Za dodavanje komponenti spremniku tabova (engl. *tabbed pane*), potrebno je koristiti metodu **`add()`**. Pomoću te metode dodaje se novi tab u spremnik tabova i po potrebi reorganizira ostala proširenja tabova, ukoliko je to potrebno. Osim toga moguće je koristiti i metode **`addTab()`** ili **`insertTab()`**. Ako je potrebno maknuti neki od tabova, dovoljno je pozvati metodu **`remove()`** nad komponentom koja predstavlja tab.

Proširenja tabova mogu biti na sjevernoj, južnoj, istočnoj ili zapadnoj strani sadržaja spremnika tabova. Taj parametar moguće je podesiti pozivanjem metode **`setTabPlacement()`** i predavanjem jedne od **`SwingConstants`** konstanti koje predstavljaju orijentaciju proširenja tabova.

Komponenti **`JTabbedPane`** moguće je pridijeliti jednog ili više primatelja događaja promjene **`ChangeListeners`**. U trenutku kad se označi jedan od tabova unutar spremnika tabova, šalje se događaj **`ChangeEvent`** svim pridjeljenim primateljima događaja i poziva se metoda **`stateChanged()`** svakog od primatelja događaja.

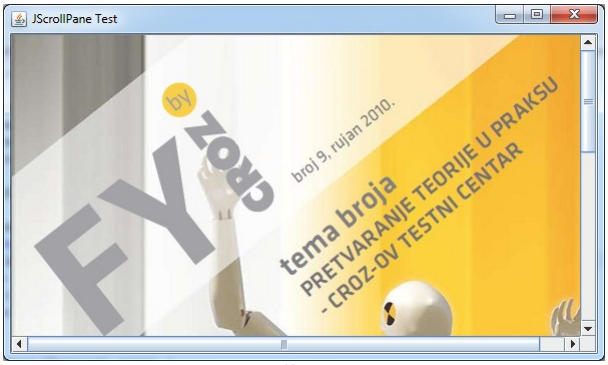
Unutar poglavlja "Dodaci – Primjer korištenja tabova" naveden je primjer korištenja tabova.

9.10. Preglednici

Klasa **`javax.swing.JScrollPane`** predstavlja komponentu pomoću koje je moguće pregledavati sadržaj prozora i u slučaju kad ne stane unutar okvira prozora (engl. *scrolling*). Korištenje komponente **`JScrollPane`** je vrlo jednostavno. Svaka komponenta ili spremnik komponenata može biti spremljena u **`JScrollPane`** komponentu i pregledavati se na taj način. Da bi se to postiglo, u konstruktor **`JScrollPane`** klase potrebno je predati komponentu koja se želi pregledavati:

```
JScrollPane jsp = new JScrollPane(myLabel);
```

Sljedeći prozor prikazuje primjer korištenja vertikalnih i horizontalnih preglednima za sliku koja je puno veća od same veličine prozora u kojem se nalazi:



Slika 9.13. Korištenje preglednika

Programski kod pomoću kojeg je moguće dobiti navedenu funkcionalnost izgleda ovako:

```
package hr.tvz.programiranje.java.labosi.swing;
```

```
import javax.swing.*;

public class ScrollPaneTest extends JFrame {

     public ScrollPaneTest() {
          super("JScrollPane Test");
          ImageIcon ii = new ImageIcon("FYI.jpg");
          JScrollPane jsp = new JScrollPane(new JLabel(ii));
          getContentPane().add(jsp);
          setSize(300,250);
          setVisible(true);
     }

     public static void main(String[] args) {
          new ScrollPaneTest();
     }
}
```

9.11. Komponente za odabir

Klasa **javax.swing.JComboBox** predstavlja osnovnu komponentu grafičkog sučelja koja se sastoji od dva dijela:

1. *pop-up* izbornika koji sadržava komponentu **JList** (objašnjenu u sljedećem poglavlju) unutar komponente **JScrollPane.**

2. gumba koji služi kao spremnik komponente za promjenu (engl. *editor*) ili komponente za prikaz, te strelice pomoću koje se dolazi do pop-up izbornika.

Jedna od najvažnijih značajki komponente za odabir je omogućavanje odabira samo jedne od ponuđenih opcija, za razlike od liste koja omogućava odabir više ponuđenih opcija istovremeno (liste se obrađuju u sljedećem poglavlju).

Za klasu **JComboBox** postoji nekoliko konstruktora. Podrazumijevani konstruktor koristi se za kreiranje komponente za odabir (engl. *combo box*) s praznom listom, a ostali konstruktori kreiraju komponentu za odabir predavanjem kao ulazni argument konstruktora jednodimenzionalno polje, zbirku **Vector** ili implementaciju sučelja **ComboBoxModel**.

Klasa **JComboBox** koristi **ListDataEvent** događaje za prijenos informacije o promjenama stanja liste za odabir (engl. *drop-down list*). Događaji **ItemEvent** i **ActionEvent** šalju se u slučaju promjene odabira u komponenti, a potrebno ih je obrađivati pomoću primatelja događaja (engl. *event listeners*) **ItemListener** i **ActionListener**.

Ako komponenta **JComboBox** ima mogućnost promjene (engl. *editable*), što nije podrazumijevana postavka, osim označavanja vrijednosti u komponenti za odabir nudi mogućnost i promjene tih vrijednosti. Tu postavku moguće je promijeniti pozivanjem metode **setEditable().**

Primjer korištenja komponenti za odabir prikazan je u poglavlju 7.2.3 Dodaci – primjer

korištenja komponente za odabir.

9.12. Liste

Klasa **`javax.swing.JList`** prestavlja osnovnu komponentu grafičkog sučelja koja omogućava odabir više opcija istovremeno. **`JList`** ima dva modela: **`ListModel`** za podatke u listi i **`ListSelectionModel`** za odabir opcija u listi. Za razliku od komponente za odabir (engl. *combo box*), lista ne omogućava promjenu vrijednosti unutar liste.

Za kreiranje **`JList`** komponente postoji nekoliko konstruktora. Moguće je koristiti podrazumijevani konstruktor, konstruktoru predati podatke u obliku jednodimenzionalnog polja, u obliku zbirke **`Vector`** ili neku implementaciju sučelja **`ListModel`**. Pomoću implementacije sučelja **`ListModel`** omogućena je maksimalna kontrola sadržaja i izgleda liste. Podatke je moguće dodati listi i pomoću metoda **`setModel()`** ili **`setListData()`**. **`JList`** ne omogućava izravan pristup svojim elementima, već je potrebno dohvatiti dodijeljenu implementaciju **`ListModel`** sučelja da bi se moglo upravljati tim podacima. Međutim, ukoliko se implementiraju sve metode sučelja **`ListSelectionModel`**, **`JList`** omogućava izravan pristup podacima o odabiru (engl. *selection data*).

`JList` također podržava metodu **`locationToIndex()`** koja vraća indeks ćelije u kojoj se nalazi određena točka (o obliku objekta klase **`Point`**) u koordinatnom prostoru same liste. Iako **`JList`** ne podržava dvostruki klik mišem, moguće je implementirati to ponašanje na jednostavan način. Pomoću objekata klasa **`MouseAdapter`**, **`MouseEvent`** i metode **`locationToIndex()`**, dvostruki klik na neku ćeliju liste moguće je detektirati na sljedeći način:

```
myList.addMouseListener(new MouseAdapter() {
   public void mouseClicked(MouseEvent e) {
      if (e.getClickCount() == 2) {
     //indeks ćelije na kojoj je obavljen dvostruki klik mišem
         int cellIndex = myJList.locationToIndex(e.getPoint());
      }
   }
});
```

Sljedeći prozor prikazuje primjer liste sa sadržajem većim od veličine prozora zbog čega se s desne strane pojavio preglednik (engl. *scrollbar*):

Slika 9.14. Okvir s listom

Ekran na prošloj slici moguće je dobiti izvođenjem sljedećeg programskog koda:

```
package hr.tvz.programiranje.java.labosi.swing;

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

public class JListTest extends JFrame {
     protected JList m_statesList;
     public JListTest() {
          super("Swing List");
          setSize(500, 240);
          String [] states = {
                    "AK Alaska Juneau",
                    "AL Alabama Montgomery",
                    "AR Arkansas Little Rock",
                    "AZ Arizona Phoenix",
                    "CA California Sacramento",
                    "CO Colorado Denver",
                    "CT Connecticut Hartford",
                    "DE Delaware Dover",
                    "FL Florida Tallahassee",
                    "GA Georgia Atlanta",
                    "HI Hawaii Honolulu",
                    "IA Iowa Des Moines",
                    "ID Idaho Boise",
                    "IL Illinois Springfield",
                    "IN Indiana Indianapolis",
                    "KS Kansas Topeka",
                    "KY Kentucky Frankfort",
                    "LA Louisiana Baton Rouge",
                    "MA Massachusetts Boston",
                    "MD Maryland Annapolis",
                    "ME Maine Augusta",
                    "MI Michigan Lansing",
                    "MN Minnesota St.Paul",
                    "MO Missouri Jefferson City",
                    "MS Mississippi Jackson",
                    "MT Montana Helena",
                    "NC North Carolina Raleigh",
                    "ND North Dakota Bismarck",
                    "NE Nebraska Lincoln",
                    "NH New Hampshire Concord",
                    "NJ New Jersey Trenton",
                    "NM New Mexico SantaFe",
                    "NV Nevada Carson City",
                    "NY New York Albany",
                    "OH Ohio Columbus",
                    "OK Oklahoma Oklahoma City",
                    "OR Oregon Salem",
                    "PA Pennsylvania Harrisburg",
                    "RI Rhode Island Providence",
                    "SC South CarolinaColumbia",
```

```
                    "SD South Dakota Pierre",
                    "TN Tennessee Nashville",
                    "TX Texas Austin",
                    "UT Utah Salt Lake City",
                    "VA Virginia Richmond",
                    "VT VermontMontpelier",
                    "WA WashingtonOlympia",
                    "WV West VirginiaCharleston",
                    "WI WisconsinMadison",
                    "WY WyomingCheyenne"
          };

          m_statesList = new JList(states);
          JScrollPane ps = new JScrollPane();
          ps.getViewport().add(m_statesList);
          getContentPane().add(ps, BorderLayout.CENTER);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
     }

     public static void main(String argv[]) {
          new JListTest();
     }
}
```

9.13. Tekstualne komponente

9.13.1. JTextComponent

Apstraktna klasa **`javax.swing.text.JTextComponent`** služi kao nadklasa svakoj tekstualnoj komponenti u Swingu. Funkcionalnost svake tekstualne komponente definirana je tom klasom, zajedno s nizom drugih klasa i sučelja u tom paketu. Osim u paketu **`javax.swing.text`**, u paketu **`javax.swing`** također postoje tekstualne komponente: **`JTextField`**, **`JPasswordField`**, **`JTextArea`**, **`JEditorPane`** i **`JTextPane`**.

Tekstualni sadržaj moguće je manipulirati instancama klasa koje implementiraju sučelje **`javax.swing.text.Document`**, koji se ponaša kao model tekstualnih komponenata. Paket **`text`** uključuje dvije implementacije sučelja **`Document`**: **`PlainDocument`** i **`StyledDocument`**. Implementacija **`PlainDocument`** podržava samo jedan font i jednu boju teksta, te je ograničen na tekstualni sadržaj. Implementacija **`StyledDocument`** je puno složenija i omogućava niz naprednijih opcija.

Komponente **`JTextField`**, **`JPasswordField`** i **`JTextArea`** koriste **`PlainDocument`** model, a komponente **`JEditorPane`** i **`JTextPane`** koriste **`StyledDocument`** model. Model komponenti moguće je dohvatiti pomoću metode **`getDocument()`**, a postaviti pomoću metode **`setDocument()`**. Osim toga implementacijama **`Document`** sučelja moguće je dodijeliti primatelje događaja **`DocumentListeners`** koji će primati događaje koje generiraju promjene sadržaja dokumenta.

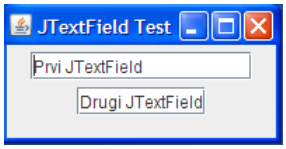
Sve tekstualne komponente sadržavaju informacije o svom trenutnom stanju. Trenutno označeni niz znakova moguće je dohvatiti u **`String`** obliku pomoću metode **`getSelectedText()`**.

9.13.2. JTextField

Klasa **`javax.swing.JTextField`** predstavlja tekstualnu komponentu veličine jednog retka koja koristi model **`PlainDocument`**. Atribut **`horizontalAlignment`** definira poravnanje teksta unutar komponente. U tu svrhu koriste se konstante **`JTextField.LEFT`**, **`JTextField.RIGHT`** i **`JTextField.CENTER`**.

Postoji nekoliko različitih konstruktora **`JTextField`** klase od kojih dva omogućavaju definiranje broja znakovnih polja koja sadržava ta tekstualna komponenta. Taj podatak moguće je dohvatiti i postaviti pomoću metoda **`getColumns()`** i **`setColumns()`**.

Sljedeći prozor prikazuje primjer **`JTextField`** komponenata, a u nastavku je prikazan programski kod pomoću kojeg se kreiraju te tekstualne komponente:



Slika 9.15. Okvir s **`JTextField`** komponentama

```
package hr.tvz.programiranje.java.labosi.swing;

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


public class JTextFieldTest extends JFrame {

     public JTextFieldTest() {
          super("JTextField Test");

          getContentPane().setLayout(new FlowLayout());
          JTextField textField1 = new JtextField(
                                             "Prvi JTextField",14);
          JTextField textField2 = new JTextField();
          textField2.setText("Drugi JTextField");
          getContentPane().add(textField1);
          getContentPane().add(textField2);

          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setSize(200,100);
          setVisible(true);
     }

     public static void main(String argv[]) {
          new JTextFieldTest();
     }

}
```

9.13.3. JPasswordField

Klasa **javax.swing.JPasswordField** predstavlja jednostavno proširenje **JTextField** komponente pri čemu umjesto upisanih znakova u tekstualnu komponentu upisuje znakove '*' čime se zapravo sakriva ulazni niz znakova. Ukoliko je potrebno promijeniti znak koji se prikazuje umjesto stvarnih znakova, dovoljno je pozvati metodu **setEchoChar()** i promijeniti tu podrazumijevanu postavku.
Za razliku od ostalih tekstualnih komponenata, nije moguće dohvatiti sadržaj **JPasswordField** komponente pomoću metode **getText()**. Umjesto te metode potrebno je koristiti metodu **getPassword()** koja vraća polje znakova.

9.13.4. JTextArea

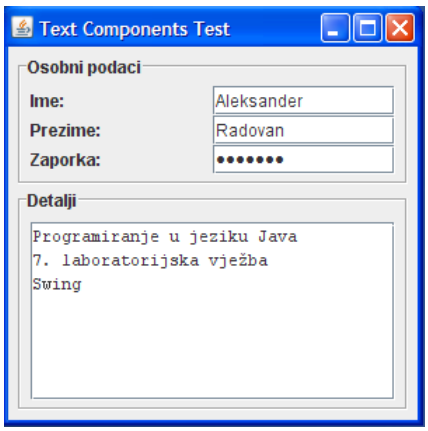
Klasa **javax.swing.JTextArea** predstavlja komponentu koja omogućava unošenje više linija teksta i koja koristi model **PlainDocument**. Postoji nekoliko načina na koje je moguće dodati tekst **JTextArea** komponenti. Tekst je moguće dodati pomoću parametra konstruktora, pomoću metode **append()** na kraj tekstualne komponente, pomoću metode **insert()** na neko određeno mjesto unutar komponente, te zamijeniti dio teksta pomoću metode **replaceRange()**. Osim tih metoda moguće je koristiti i metodu **setText()**,

kao i kod komponente **`JTextComponent`**.

`JTextArea` sadrži atribute **`lineCount`** i **`rows`** pri čemu **`lineCount`** označava broj linija koji komponenta sadrži, a **`rows`** označava broj komponenta u zadanom trenutku ispisuje.

9.13.5. Primjer korištenja osnovnih tekstualnih komponenata

Sljedeća slika prikazuje okvir u kojem se koriste osnovne tekstualne komponente iza koje slijedi programski kod kojim je kreiran okvir:



Slika 9.16. Okvir s osnovnim tekstualnim komponentama

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;

import javax.swing.*;

public class BasicTextComponentTest extends JFrame {

     protected JTextField m_firstTxt;
     protected JTextField m_lastTxt;
     protected JPasswordField m_passwordTxt;
     protected JTextArea m_commentsTxt;

     public BasicTextComponentTest() {

          super("Text Components Test");
          Font monospaced = new Font(
                    "Monospaced", Font.PLAIN, 12);
          JPanel pp = new JPanel(new BorderLayout());

          JPanel p = new JPanel(new BorderLayout());
          p.setBorder(new CompoundBorder(
                    new TitledBorder(new EtchedBorder(),
               "Osobni podaci"),
                    new EmptyBorder(1, 5, 3, 5)));

          JPanel panel = new JPanel(new GridLayout(3, 3));
          panel.add(new JLabel("Ime:"));
          m_firstTxt = new JTextField(20);
          panel.add(m_firstTxt);

          panel.add(new JLabel("Prezime:"));
          m_lastTxt = new JTextField(20);
          panel.add(m_lastTxt);

          panel.add(new JLabel("Zaporka:"));
          m_passwordTxt = new JPasswordField(20);
          m_passwordTxt.setFont(monospaced);
          panel.add(m_passwordTxt);

          p.add(panel);

          pp.add(p, BorderLayout.NORTH);
          m_commentsTxt = new JTextArea("", 4, 20);
          m_commentsTxt.setFont(monospaced);
          m_commentsTxt.setLineWrap(true);
          m_commentsTxt.setWrapStyleWord(true);
          p = new JPanel(new BorderLayout());
          p.add(new JScrollPane(m_commentsTxt));
          p.setBorder(new CompoundBorder(
                    new TitledBorder(
                                   new EtchedBorder(), "Detalji"),
                    new EmptyBorder(3, 5, 3, 5))
```

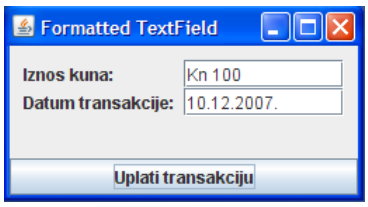
```
            );
            pp.add(p, BorderLayout.CENTER);
            pp.setBorder(new EmptyBorder(5, 5, 5, 5));
            getContentPane().add(pp);
            pack();
      }
      public static void main(String[] args) {
            JFrame frame = new BasicTextComponentTest();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(300, 300);
            frame.setVisible(true);
      }
}
```

9.13.6. Tekstualne komponente s formatiranjem

Klasa **`javax.swing.JFormattedTextField`** predstavlja komponentu koja nasljeđuje tekstualnu komponentu **`JTextField`** koju nadopunjava svojstvom formatiranja teksta.

Najjednostavniji način korištenja komponente **`JFormattedTextField`** je predavanje instance klase **`java.text.Format`** kao ulaznog parametra konstruktora. Pomoću te instance klase omogućeno je formatiranje ispisa poput zadanog formata broja, datuma i sl. Podklase klase **`Format`** su **`DateFormat`**, **`NumberFormat`**, **`MessageFormat`** i dr.

Sljedeća slika prikazuje okvir u kojem se koriste dvije tekstualne komponente tipa **`JFormattedTextField`**, a u nastavku se nalazi programski kod kojim se kreiraju te komponente:



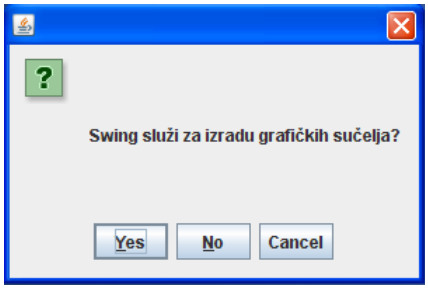
Slika 9.17. Tekstualne komponente s formatiranjem

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

public class FormattedTextFieldTest extends JFrame {

     public FormattedTextFieldTest() {

          super("Formatted TextField");
          JPanel p = new JPanel(new GridLayout(3, 3));
          p.setBorder(new EmptyBorder(10, 10, 10, 10));
          p.add(new JLabel("Iznos kuna:"));
          NumberFormat formatMoney =
               NumberFormat.getCurrencyInstance();
          JFormattedTextField ftMoney =
               new JFormattedTextField(formatMoney);
          ftMoney.setColumns(10);
          ftMoney.setValue(new Double(100));
          p.add(ftMoney);
          p.add(new JLabel("Datum transakcije:"));
          DateFormat formatDate = new
                                  SimpleDateFormat("dd.MM.yyyy.");
          JFormattedTextField ftfDate =
               new JFormattedTextField(formatDate);
          ftfDate.setColumns(10);
          ftfDate.setValue(new Date());
          p.add(ftfDate);
          getContentPane().add(p, BorderLayout.CENTER);
          JButton btn = new JButton("Uplati transakciju");
          getContentPane().add(btn, BorderLayout.SOUTH);
          pack();
     }

     public static void main( String args[] ) {
          FormattedTextFieldTest mainFrame =
               new FormattedTextFieldTest();
          mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          mainFrame.setVisible(true);
     }
}
```

9.14. Dijalozi

Klasa **`java.swing.JDialog`** predstavlja komponentu koja omogućava kreiranje dijaloga aplikacije s korisnikom. Jednostavni dijalozi sastoje se od prikazivanja jednostavnih poruka korisniku ili primanja jednostavnih informacija od korisnika. Postoji nekoliko standardiziranih dijaloga koji se dobijaju pomoću klase **`JOptionPane`**.

Jedan od najjednostavnijih dijaloga je dijalog s pitanjem prikazan na slici:



Slika 9.18. Primjer dijaloga s pitanjem

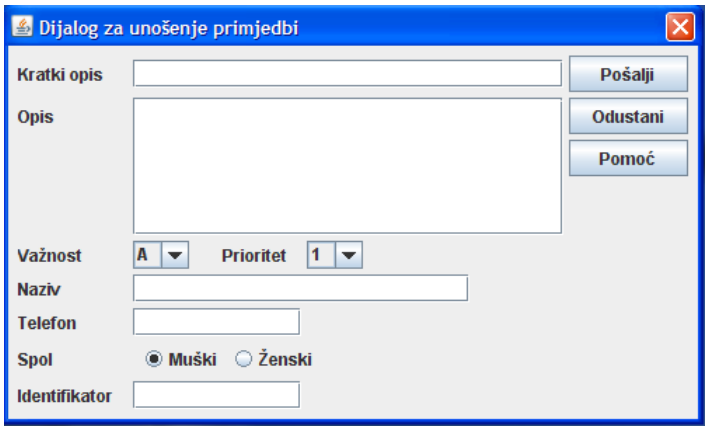
koji se može dobiti s vrlo jednostavnim programskim kodom:

```
package hr.tvz.programiranje.java.labosi.swing;

import javax.swing.*;

public class JDialogTest {

     public static void main(String[] args) {

          JDialog dialog = new JDialog();
          JOptionPane pane = new JoptionPane(
               "Swing služi za     izradu grafičkih sučelja?",
          JOptionPane.QUESTION_MESSAGE,
     JOptionPane.YES_NO_CANCEL_OPTION);
          dialog.setContentPane(pane);
          dialog.setSize(300, 200);
          dialog.setVisible(true);
     }
}
```

9.15. DODACI

9.15.1. Primjer korištenja organizatora komponenti GridBagLayout

Sljedeća slika prikazuje prozor u kojem se koristi **`GridBagLayout`** organizator komponenti, a u nastavku se nalazi programski kod pomoću kojeg se on kreiran:



Slika 9.19. Primjer korištenja GridBagLayout organizatora komponenti

```
package hr.tvz.programiranje.java.labosi.swing;

import java.awt.*;

import javax.swing.*;

public class GridBagLayoutTest extends JFrame {
     public GridBagLayoutTest() {
          setTitle( "Dijalog za unošenje primjedbi" );
          setSize( 500, 300 );

          // Kreiranje panela koji će sadržavati sve
                // komponente
          JPanel panel = new JPanel( new BorderLayout() );
          panel.setLayout( new GridBagLayout() );
          panel.setBorder( new EmptyBorder(
                                    new Insets( 5, 5, 5, 5 ) ) );
          getContentPane().add( BorderLayout.CENTER, panel );
          GridBagConstraints c = new GridBagConstraints();

          // Dimenzije tekstualnih komponenata
          Dimension shortField = new Dimension( 40, 20 );
          Dimension mediumField = new Dimension( 120, 20 );
          Dimension longField = new Dimension( 240, 20 );
          Dimension hugeField = new Dimension( 240, 80 );

          // Prazna mjesta između komponenata
          EmptyBorder border = new EmptyBorder(
                                             new Insets(0,0,0,10));
          EmptyBorder border1 = new EmptyBorder(
                                            new Insets(0,20,0,10));
```

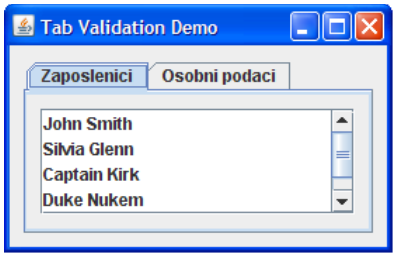
```
        // Dodavanje praznih mjesto oko komponenata
        c.insets = new Insets( 2, 2, 2, 2 );

        // Stavljanje komponenata na "Zapad"
        c.anchor = GridBagConstraints.WEST;
        JLabel lbl1 = new JLabel( "Kratki opis" );
        lbl1.setBorder( border );
        panel.add( lbl1, c );
        JTextField txt1 = new JTextField();
        txt1.setPreferredSize( longField );
        c.gridx = 1;
        c.weightx = 1.0; // Korištenje horizontalnog mjesta
        c.gridwidth = 3; // Proširivanje na 3 stupca

        //Popunjavanje 3 stupca
        c.fill = GridBagConstraints.HORIZONTAL;

        panel.add( txt1, c );
        JLabel lbl2 = new JLabel( "Opis" );
        lbl2.setBorder( border );
        c.gridwidth = 1;
        c.gridx = 0;
        c.gridy = 1;
        // Ne upotrebljava se dodatno horizontalna mjesta
        c.weightx = 0.0;
        panel.add( lbl2, c );
        JTextArea area1 = new JTextArea();
        JScrollPane scroll = new JScrollPane( area1 );
        scroll.setPreferredSize( hugeField );
        c.gridx = 1;
        //Korištenje svih horizontalnih mjesta
        c.weightx = 1.0;
        //Korištenje svih vertikalnih mjesta
        c.weighty = 1.0;
        //Proširenje kroz 3 stupca
        c.gridwidth = 3;
        //Proširenje kroz 2 retka
        c.gridheight = 2;
        //Popunjavanje stupaca i redaka
        c.fill = GridBagConstraints.BOTH;
        panel.add( scroll, c );
        JLabel lbl3 = new JLabel( "Važnost" );
        lbl3.setBorder( border );
        c.gridx = 0;
        c.gridy = 3;
        c.gridwidth = 1;
        c.gridheight = 1;
        c.weightx = 0.0;
        c.weighty = 0.0;
        c.fill = GridBagConstraints.NONE;
        panel.add( lbl3, c );
        JComboBox combo3 = new JComboBox();
        combo3.addItem( "A" );
```

```
        combo3.addItem( "B" );
        combo3.addItem( "C" );
        combo3.addItem( "D" );
        combo3.addItem( "E" );
        combo3.setPreferredSize( shortField );
        c.gridx = 1;
        panel.add( combo3, c );
        JLabel lbl4 = new JLabel( "Prioritet" );
        lbl4.setBorder( border1 );
        c.gridx = 2;
        panel.add( lbl4, c );
        JComboBox combo4 = new JComboBox();
        combo4.addItem( "1" );
        combo4.addItem( "2" );
        combo4.addItem( "3" );
        combo4.addItem( "4" );
        combo4.addItem( "5" );
        combo4.setPreferredSize( shortField );
        c.gridx = 3;
        panel.add( combo4, c );
        JLabel lbl5 = new JLabel( "Naziv" );
        lbl5.setBorder( border );
        c.gridx = 0;
        c.gridy = 4;
        panel.add( lbl5, c );
        JTextField txt5 = new JTextField();
        txt5.setPreferredSize( longField );
        c.gridx = 1;
        c.gridwidth = 3;
        panel.add( txt5, c );
        JLabel lbl6 = new JLabel( "Telefon" );
        lbl6.setBorder( border );
        c.gridx = 0;
        c.gridy = 5;
        panel.add( lbl6, c );
        JTextField txt6 = new JTextField();
        txt6.setPreferredSize( mediumField );
        c.gridx = 1;
        c.gridwidth = 3;
        panel.add( txt6, c );
        JLabel lbl7 = new JLabel( "Spol" );
        lbl7.setBorder( border );
        c.gridx = 0;
        c.gridy = 6;
        panel.add( lbl7, c );
        JPanel radioPanel = new JPanel();

        radioPanel.setLayout(
                            new FlowLayout(FlowLayout.LEFT,5,0));
        ButtonGroup group = new ButtonGroup();
        JRadioButton radio1 = new JRadioButton( "Muški" );
        radio1.setSelected( true );
```

```
        group.add( radio1 );
        JRadioButton radio2 = new JRadioButton( "Ženski" );
        group.add( radio2 );
        radioPanel.add( radio1 );
        radioPanel.add( radio2 );
        c.gridx = 1;
        c.gridwidth = 3;
        panel.add( radioPanel, c);
        JLabel lbl8 = new JLabel( "Identifikator" );
        lbl8.setBorder( border );
        c.gridx = 0;
        c.gridy = 7;
        c.gridwidth = 1;
        panel.add( lbl8, c );
        JTextField txt8 = new JTextField();
        txt8.setPreferredSize( mediumField );
        c.gridx = 1;
        c.gridwidth = 3;
        panel.add( txt8, c );
        JButton submitBtn = new JButton( "Pošalji" );
        c.gridx = 4;
        c.gridy = 0;
        c.gridwidth = 1;
        c.fill = GridBagConstraints.HORIZONTAL;
        panel.add( submitBtn, c );
        JButton cancelBtn = new JButton( "Odustani" );
        c.gridy = 1;
        panel.add( cancelBtn, c );
        JButton helpBtn = new JButton( "Pomoć" );
        c.gridy = 2;
        c.anchor = GridBagConstraints.NORTH;
        panel.add( helpBtn, c );
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setVisible( true );
              this.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent we) {
                  System.exit(0);
             }
        });
    }

    public static void main( String[] args ) {
        new GridBagLayoutTest();
    }
}
```

9.15.2. Primjer korištenja tabova

Sljedeća slika prikazuje prozor u kojem se koriste tabovi, a u nastavku se nalazi programski kod pomoću kojeg se on kreiran. Tab "Osobni podaci" moguće je aktivirati samo ukoliko je odabran jedan zapis unutar taba "Zaposlenici".



9.20. Primjer korištenja tabova

```
package hr.tvz.programiranje.java.labosi.swing;

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

public class TabbedPaneTest extends JFrame {

     public static final int LIST_TAB = 0;
     public static final int DATA_TAB = 1;

     protected Person[] m_employees = {
               new Person("John", "Smith", "111-1111"),
               new Person("Silvia", "Glenn", "222-2222"),
               new Person("Captain", "Kirk", "333-3333"),
               new Person("Duke", "Nukem", "444-4444"),
               new Person("James", "Bond", "000-7777")
     };
     protected JList m_list;
     protected JTextField m_firstTxt;
     protected JTextField m_lastTxt;
     protected JTextField m_phoneTxt;
     protected JTabbedPane m_tab;

     public TabbedPaneTest() {
          super("Tab Validation Demo");
          JPanel p1 = new JPanel(new BorderLayout());
          p1.setBorder(new EmptyBorder(10, 10, 10, 10));
          m_list = new JList(m_employees);
          m_list.setVisibleRowCount(4);
          JScrollPane sp = new JScrollPane(m_list);
          p1.add(sp, BorderLayout.CENTER);
          MouseListener mlst = new MouseAdapter() {
               public void mouseClicked(MouseEvent evt) {
                    if (evt.getClickCount() == 2)
                      m_tab.setSelectedIndex(DATA_TAB);
               }
          };
          m_list.addMouseListener(mlst);
          JPanel p2 = new JPanel(new BorderLayout());
          p2.setBorder(new EmptyBorder(10, 10, 10, 10));
          p2.add(new JLabel("Ime:"));
          m_firstTxt = new JTextField(20);
          p2.add(m_firstTxt);
          p2.add(new JLabel("Prezime:"));
```

```
        m_lastTxt = new JTextField(20);
        p2.add(m_lastTxt);
        p2.add(new JLabel("Telefon:"));
        m_phoneTxt = new JTextField(20);;
        p2.add(m_phoneTxt);
        m_tab = new JTabbedPane();
        m_tab.addTab("Zaposlenici", p1);
        m_tab.addTab("Osobni podaci", p2);
        m_tab.addChangeListener(new TabChangeListener());
        JPanel p = new JPanel();
        p.add(m_tab);
        p.setBorder(new EmptyBorder(5, 5, 5, 5));
        getContentPane().add(p);
        pack();
    }

    public Person getSelectedPerson() {
        return (Person)m_list.getSelectedValue();
    }

    public static void main(String[] args) {
        JFrame frame = new TabbedPaneTest();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    private class TabChangeListener implements ChangeListener {

        public void stateChanged(ChangeEvent e) {
            Person sp = getSelectedPerson();
            switch(m_tab.getSelectedIndex()) {
            case DATA_TAB:
                 if (sp == null) {
                m_tab.setSelectedIndex(LIST_TAB);
                      return;
                 }
                 m_firstTxt.setText(sp.m_firstName);
                      m_lastTxt.setText(sp.m_lastName);
                      m_phoneTxt.setText(sp.m_phone);
                      break;

            case LIST_TAB:
                 if (sp != null) {
                      sp.m_firstName =
          m_firstTxt.getText();
                      sp.m_lastName =
           m_lastTxt.getText();
                      sp.m_phone =
                            m_phoneTxt.getText();
                      m_list.repaint();
                      }
                      break;
```

```
                    }
                }
    }

    private class Person {

        public String m_firstName;
        public String m_lastName;
        public String m_phone;

        public Person(String firstName, String lastName,
                  String phone)
        {
            m_firstName = firstName;
            m_lastName = lastName;
            m_phone =phone;
        }

        public String toString() {
            return m_firstName + " " + m_lastName;
        }
    }
}
```

9.15.3. Primjer korištenja komponente za odabir

Sljedeća slika prikazuje prozor u kojem se koriste komponente za odabir, a u nastavku se nalazi programski kod pomoću kojeg se on kreiran:

9.21. Primjer korištenja komponenata za odabir

```
package hr.tvz.programiranje.java.labosi.swing;

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

```
public class ComboBoxTest extends JFrame {

    public ComboBoxTest() {
        super("ComboBoxes [Usporedba auta]");
        getContentPane().setLayout(new BorderLayout());
        Vector cars = new Vector();

        Car maxima = new Car("Maxima", "Nissan", new
             ImageIcon("maxima.gif"));
        maxima.addTrim("GXE", 21499, 19658,
                           "3.0L V6 190-hp");
        maxima.addTrim("SE", 23499, 21118,
                           "3.0L V6 190-hp");
        maxima.addTrim("GLE", 26899, 24174,
                           "3.0L V6 190-hp");
        cars.addElement(maxima);

        Car accord = new Car("Accord", "Honda", new
        ImageIcon("accord.gif"));
        accord.addTrim("LX Sedan", 21700, 19303,
                           "3.0L V6 200-hp");
        accord.addTrim("EX Sedan", 24300, 21614,
                           "3.0L V6 200-hp");
        cars.addElement(accord);

        Car camry = new Car("Camry", "Toyota", new
        ImageIcon("camry.gif"));
        camry.addTrim("LE V6", 21888, 19163,
                           "3.0L V6 194-hp");
        camry.addTrim("XLE V6", 24998, 21884,
                           "3.0L V6 194-hp");
        cars.addElement(camry);

        Car lumina = new Car("Lumina", "Chevrolet", new
        ImageIcon("lumina.gif"));
        lumina.addTrim("LS", 19920, 18227,
                           "3.1L V6 160-hp");
        lumina.addTrim("LTZ", 20360, 18629,
                           "3.8L V6 200-hp");
        cars.addElement(lumina);

        Car taurus = new Car("Taurus", "Ford", new
        ImageIcon("taurus.gif"));
        taurus.addTrim("LS", 17445, 16110,
                           "3.0L V6 145-hp");
        taurus.addTrim("SE", 18445, 16826,
                           "3.0L V6 145-hp");
        taurus.addTrim("SHO", 29000, 26220,
                           "3.4L V8 235-hp");
        cars.addElement(taurus);


        Car passat = new Car("Passat", "Volkswagen", new
```

```
          ImageIcon("passat.gif"));
          passat.addTrim("GLS V6", 23190, 20855,
                              "2.8L V6 190-hp");
          passat.addTrim("GLX", 26250, 23589,
                              "2.8L V6 190-hp");
          cars.addElement(passat);

          getContentPane().setLayout(
                    new GridLayout(1, 2, 5, 3));
          CarPanel pl = new CarPanel("Osnovni model", cars);
          getContentPane().add(pl);
          CarPanel pr = new CarPanel(
                    "Usporedba s modelom", cars);
          getContentPane().add(pr);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          pl.selectCar(maxima);
          pr.selectCar(accord);
          setResizable(false);
          pack();
          setVisible(true);
     }

     public static void main(String argv[]) {
          new ComboBoxTest();
     }

     private class Car {

          protected String m_name;
          protected String m_manufacturer;
          protected Icon m_img;
          protected Vector m_trims;

          public Car(String name, String manufacturer,
          Icon img) {
               m_name = name;
               m_manufacturer = manufacturer;
               m_img = img;
               m_trims = new Vector();
          }

          public void addTrim(String name, int MSRP, int
                    invoice, String engine) {
               Trim trim = new Trim(this, name, MSRP,
          invoice, engine);
               m_trims.addElement(trim);
          }

          public String getName() { return m_name; }

          public String getManufacturer() { return
m_manufacturer; }
```

```
        public Icon getIcon() { return m_img; }

        public Vector getTrims() { return m_trims; }

        public String toString() { return m_manufacturer+"
                        "+m_name; }
    }

    private class Trim {
        protected Car m_parent;
        protected String m_name;
        protected int m_MSRP;
        protected int m_invoice;
        protected String m_engine;

        public Trim(Car parent, String name, int MSRP, int
            invoice, String engine)
        {
            m_parent = parent;
            m_name = name;
            m_MSRP = MSRP;
            m_invoice = invoice;
            m_engine = engine;
        }

        public Car getCar() { return m_parent; }

        public String getName() { return m_name; }

        public int getMSRP() { return m_MSRP; }

        public int getInvoice() { return m_invoice; }

        public String getEngine() { return m_engine; }

        public String toString() { return m_name; }
    }

    private class CarPanel extends JPanel {
        protected JComboBox m_cbCars;
        protected JComboBox m_cbTrims;
        protected JLabel m_lblImg;
        protected JLabel m_lblMSRP;
        protected JLabel m_lblInvoice;
        protected JLabel m_lblEngine;

        public CarPanel(String title, Vector cars) {
            super();
            setLayout(new BoxLayout(this,
                BoxLayout.Y_AXIS));
            setBorder(new TitledBorder(
                        new EtchedBorder(),
                title));
            JPanel p = new JPanel();
```

```
            p.add(new JLabel("Model:"));
            m_cbCars = new JComboBox(cars);
            ActionListener lst = new ActionListener() {
             public void actionPerformed(ActionEvent e)
     {
                 Car car =
    (Car)m_cbCars.getSelectedItem();
                 if (car != null)
                      showCar(car);
                 }
            };
            m_cbCars.addActionListener(lst);
            p.add(m_cbCars);
            add(p);
            p = new JPanel();
            p.add(new JLabel("Trim:"));
            m_cbTrims = new JComboBox();
            lst = new ActionListener() {
             public void actionPerformed(ActionEvent e)
     {
                 Trim trim =
    (Trim)m_cbTrims.getSelectedItem();
                       if (trim != null)
                            showTrim(trim);
                 }
            };
            m_cbTrims.addActionListener(lst);
            p.add(m_cbTrims);
            add(p);
            p = new JPanel();
            m_lblImg = new JLabel();

            m_lblImg.setHorizontalAlignment(
                 JLabel.CENTER);
            m_lblImg.setPreferredSize(new
         Dimension(140, 80));
            m_lblImg.setBorder(new
    BevelBorder(BevelBorder.LOWERED));
            p.add(m_lblImg);
            add(p);
            p = new JPanel();
            p.setLayout(new GridLayout(3, 2, 10, 5));
            p.add(new JLabel("MSRP:"));
            m_lblMSRP = new JLabel();
            p.add(m_lblMSRP);
            p.add(new JLabel("Invoice:"));
            m_lblInvoice = new JLabel();
            p.add(m_lblInvoice);
            p.add(new JLabel("Engine:"));
            m_lblEngine = new JLabel();
            p.add(m_lblEngine);
            add(p);
```

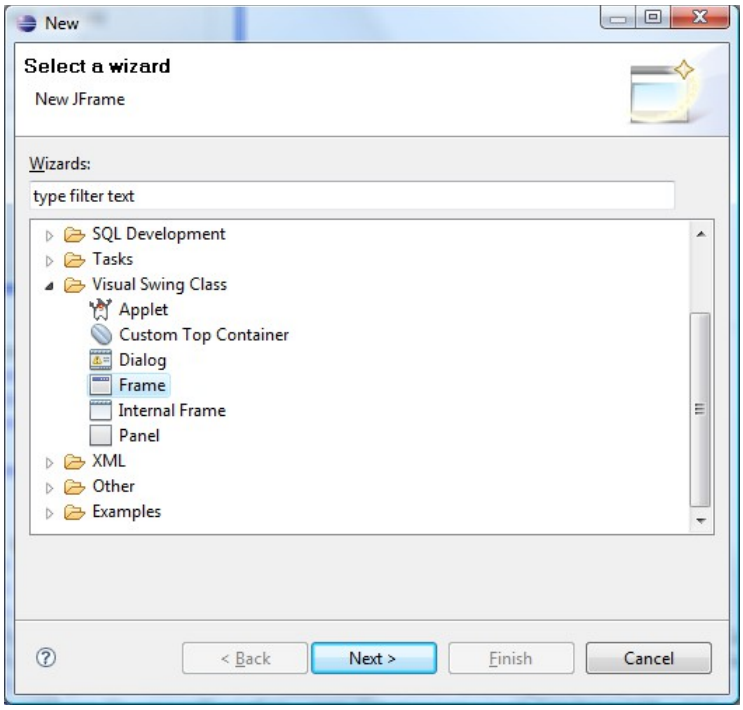
```
        }

        public void selectCar(Car car)
    { m_cbCars.setSelectedItem(car); }

        public void showCar(Car car) {
            m_lblImg.setIcon(car.getIcon());
            if (m_cbTrims.getItemCount() > 0)
                m_cbTrims.removeAllItems();
            Vector v = car.getTrims();
            for (int k=0; k<v.size(); k++)
                m_cbTrims.addItem(v.elementAt(k));
            m_cbTrims.grabFocus();
        }

        public void showTrim(Trim trim) {
            m_lblMSRP.setText("$"+trim.getMSRP());
            m_lblInvoice.setText("$"+trim.getInvoice());
            m_lblEngine.setText(trim.getEngine());
        }
    }
```

9.16. KORIŠTENJE DIZAJNERA ZA KREIRANJE GRAFIČKOG SUČELJA

Osim kreiranja grafičkog sučelja isključivo pisanjem programskog koda, moguće je korištenje i interaktivni dodatak razvojnom okruženju Eclipse koji omogućava vizualno kreiranje grafičkog sučelja. Takvi dodaci nazivaju se "dizajneri" i omogućavaju automatizirano generiranje programskog koda koji odgovara grafičkom sučelju koje je korisnik kreirao pomoću komponenti dizajnera.

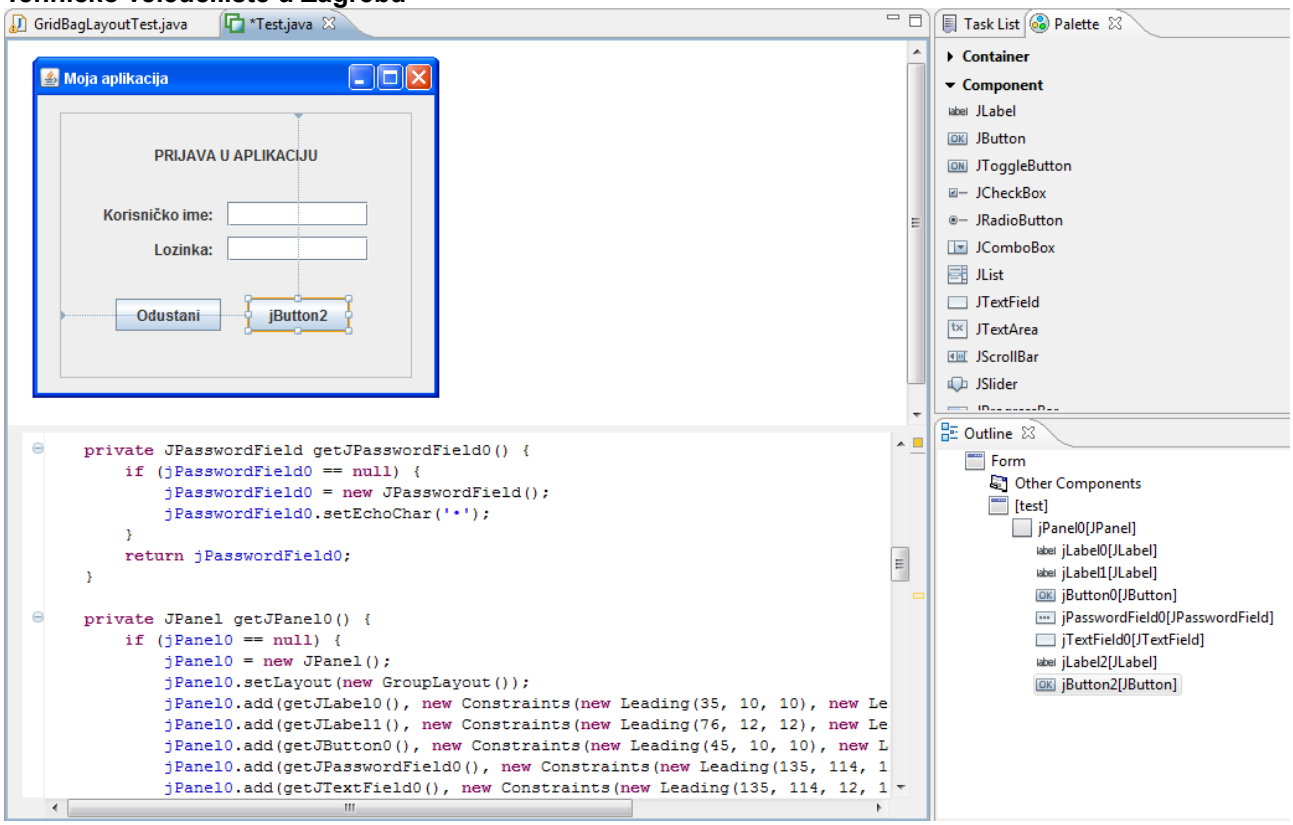
Ima nekoliko različitih dizajnera, međutim, u besplatne se ubrajaju "Visual Swing for Eclipse" (`http://eclipse.dzone.com/announcements/visual-swing-designer-eclipse`) i "Jigloo" (besplatan za nekomercijalne svrhe: `http://www.cloudgarden.com/jigloo/`). Oba dodatka je moguće instalirati pomoću "*Update site*"-a ili preuzimanja arhive i raspakirati je u mapu gdje je instalirano razvojno okruženje Eclipse (sadržaje mapa "plugins" i "features" potrebno je kopirati u istoimene mape unutar Eclipse mape).

U nastavku će ukratko biti objašnjeno korištenje dodatka "Visual Swing for Eclipse". Nakon uspješne instalacije, za kreiranje Java aplikacije sa Swing grafičkim sučeljem potrebno je kreirati posebnu vrstu klase koja se odnosi na grafičko sučelje. Takve klase je moguće pronaći unutar File->New->Other... izbornika, te odabirom skupine klasa unutar mape "Visual Swing Class" (slika 9.22):



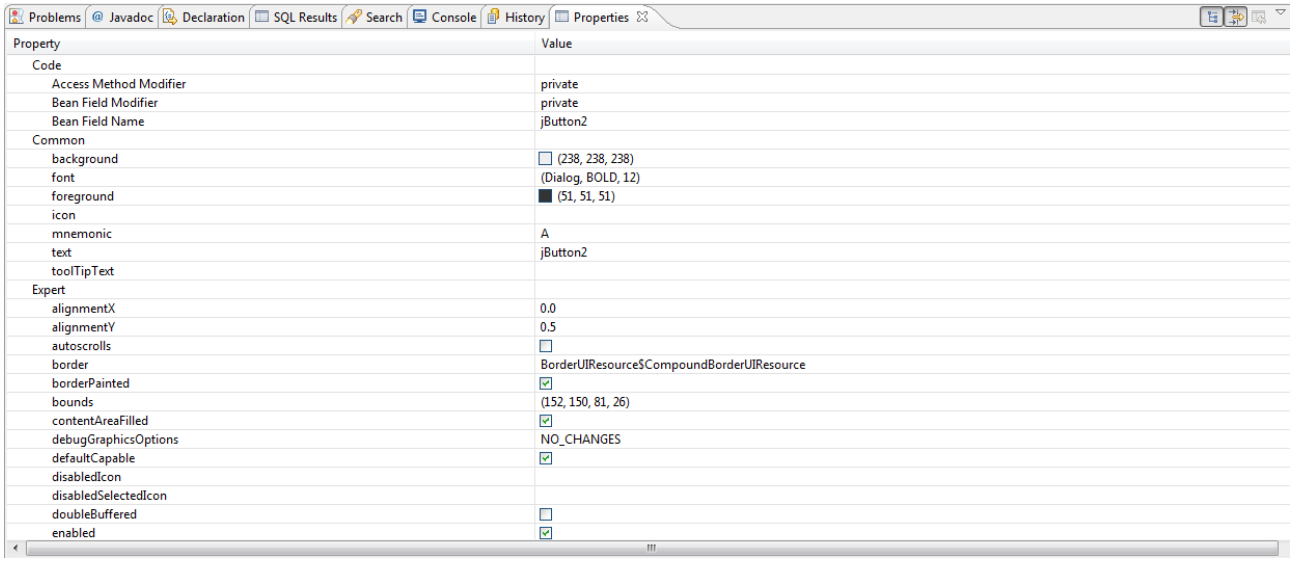
9.22. Dijalog za odabir klase vezane uz *Visual Swing* dizajner

Na primjer, odabirom klase "Frame" dobije se pripremljena klasa čiji sadržaj je moguće dinamički mijenjati dodavanjem grafičkih elemenata na sučelje. Dio ekrana unutar razvojnog okruženja Eclipse prikazuje "dizajnerski" način izgradnje grafičkog sučelja, a drugi dio prikazuje programski kod koji se izgenerirao dodavanjem grafičkih elemenata uz pomoć dizajnera (slika 9.23):



Slika 9.23. Izgled razvojnog okruženja Eclipse pri korištenju dizajnera "Visual Swing For Eclipse"

Na slici je 9.23. je prikazano već izgrađeno jednostavno grafičko sučelje od nekoliko komponenti koje su dodane na okvir (komponente je moguće odabrati iz "*Pallete view*"-a), a svojstva pojedine komponente je moguće vidjeti unutar "*Properties view*"-a (slika 9.24):



Slika 9.24. Svojstva svake od komponente grafičkog sučelja

Kod korištenja dizajnera vrlo često se koristi neki unaprijed definirani organizator raspoređivanja komponenti (engl. *layout manager*) ili ga korisnik mora odabrati. U slučaju dodatka "Visual Swing For Eclipse" koristi se **`GroupLayout`**.

Nakon dizajniranja grafičkog sučelja potrebno je u automatizirano izgenerirani programski kod ubaciti dijelove koda koje dizajner ne može generirati. U taj kod spadaju funkcionalnosti koje aplikacija mora obavljati uslijed nekih događaja (klik miša na gumb i

sl.), odnosno *evenata*. Svakoj komponenti moguće je dodati tzv. primatelj događaja (engl. *event listener*) pritiskom desne tipke miša nad komponentom (npr. gumbom) i odabirom opcije "Add/Edit events", te odabirom vrste događaja i pripadajuće metode koja će se obavljati u slučaju tog događaja (npr. *action->actionPerformed*). Nakon odabira se u programskom kodu generira samo definicija metoda, a implementaciju je potrebno napisati "ručno", jer to nije moguće obaviti pomoću dizajnera.

Nakon završetka sa svim njezinim dijelovima aplikaciju je moguće pokrenuti kao i svaku drugu klasu koja sadrži *main* metodu.

Detaljnije informacije o samom dodatku i ostalim dodacima za dizajniranje grafičkog sučelja moguće je pronaći na službenim stranicama tih dodataka za Eclipse.

10. PRISTUPANJE BAZAMA PODATAKA IZ JAVA APLIKACIJA

10.1. JDBC

JDBC (engl. *Java Database Connectivity*) je aplikativno programsko sučelje (engl. *Application Programming Interface – API*) za programski jezik Java koji definira na koji način klijentske aplikacije mogu pristupati bazi podataka, posebno relacijskoj bazi podataka.

JDBC omogućava spajanje na bazu podataka, izvršavanje upita nad bazom podataka i dohvaćanje rezultata upita nad bazom podataka.

U nastavku se opisuje način na koji je moguće iz razvojnog okruženja Eclipse ostvariti komunikaciju s bazom podataka i opisuju se detalji korištenja same baze podataka.

10.1.1. Kreiranje okruženja potrebnog za rad s bazom podataka

Za rad s bazama podataka u razvojnom okruženju Eclipse potrebno je ugraditi neke dodatke koji se besplatno mogu preuzeti s web stranica opisanih u poglavlju 10.3. Radi se o relacijskoj bazi podataka Derby i dodatku za Eclipse pod nazivom "Data Tools Platform" (DTP).

Detalji vezani uz instalaciju Derby baze podataka i ugrađivanje DTP dodatka u Eclipse, te detalji korištenja navedenih alata opisani u poglavljima 10.3.1. , 10.3.2. i 10.3.3. u nastavku.

Ostatak poglavlja se u potpunosti oslanja na korištenje dodataka opisanih u poglavlju 10.2., tako da je instalaciju tih resursa nužno potrebno obaviti prije prelaska na ostatak poglavlja kako bi primjeri isječaka programskog koda radili ispravno.

10.1.2. Ostvariranje veze s bazom podataka iz programskog jezika Java

Prije početka rada s bazom podataka iz programskog koda potrebno je ostvariti vezu (engl. *Connection*) programa i baze podataka, odnosno, sustava za upravljanje bazom podataka (engl. *Database Management System – DBMS*). To je moguće postići korištenjem klase **`DriverManager`** ili **`DataSource`**.

Korištenjem klase **`DriverManager`** vezu s bazom podataka moguće ostvariti korištenjem statičke metode **`getConnection`** koja vraća objekt tipa **`Connection`** koji predstavlja otvorenu vezu prema bazi podataka:

```
Connection veza = DriverManager.getConnection(
     "jdbc:derby:C:/Users/Aleksander/MyDB", "test","test");
```

Metoda **`getConnection`** u gornjem obliku prima tri parametra:

•"**`jdbc:derby:C:/Users/Aleksander/MyDB`**" – opisuje tip baze podataka ("**`jdbc:derby`**") i lokaciju baze podataka (na lokalnom računalu na lokaciji "**`C:/Users/Aleksander/MyDB`**")
•korisničko ime "**`test`**"
•lozinku "**`test`**"

Prije uspostavljanja veze s bazom podataka iz programskog koda (aplikacije) **nužno je potrebno prekinuti vezu iz DTP dodatka za Eclipse**, jer u suprotnom dolazi do iznimke (**`java.sql.SQLException: Failed to start database 'C:/Users/Aleksander/MyDB', see the next exception for details.`**). Bazu podataka u ovom primjeru moguće je koristiti samo u slučaju kad se na nju spaja samo jedan korisnik (Java aplikacija ili DTP *plugin*).

10.1.3. Izvođenje SQL naredbi i dohvaćanje rezultata

U poglavlju 10.3.3. opisan je postupak kako kreirati tablicu u bazi podataka i u nju dodati nekoliko redaka s podacima, a u ovom poglavlju se opisuje kako u programskom jeziku Java napisati programski kod koji će izvoditi SQL naredbe na bazom podataka s kojom je iz programskog koda ostvarena veza.

Rezultati izvršavanja SQL naredbi vrlo često sadržavaju više od jednog retka zbog čega je u programskom jeziku Java potrebno koristiti objekte klasa koji podržavaju takve funkcionalnosti – objekti klasa koje implementiraju sučelje **`ResultSet`**.

Sučelje **`ResultSet`** sadržava metode za dohvaćanje i korištenje rezultata izvršenih SQL naredbi i ti objekti mogu imati različite karakteristike i funkcionalnosti kako će biti opisano u nastavku.

Postoje **tri različita tipa** `ResultSet` objekata:

•**`TYPE_FORWARD_ONLY`** – rezultatni set (engl. *Result set*) dobiven izvođenjem SQL naredbe može se dohvaćati samo u redoslijedu od prvog do zadnjeg retka, bez mogućnosti vraćanja unatrag s obzirom na poziciju kursora (engl. *cursor*). **Početna pozicija kursora je prije prvog retka, a krajnja pozicija kursora je poslije posljednjeg retka.**

•**`TYPE_SCROLL_INSENSITIVE`** – rezultatni set je moguće dohvaćati i pregledavati u oba smjera, odnosno, kursor je moguće pomicati prema naprijed i prema nazad s obzirom na trenutnu poziciju, a može se pomaknuti i na neku apsolutnu poziciju unutar rezultatnog seta. Rezultatni set nije osjetljiv na promjene po podacima unutar rezultatnog seta.

•**`TYPE_SCROLL_SENSITIVE`** – slično kao i prošli rezultatni set, samo što je osjetljiv na promjene po podacima unutar rezultatnog seta.

Osim tipa, rezultatni set ima još dva atributa koje je moguće definirati: **konkurentnost rezultatnog seta** (engl. *result set concurrency*) i **upravljanje kursorom** (engl. *cursor holdability*).

Konkurentnost rezultatnog seta može poprimiti vrijednost **`ResultSet.CONCUR_READ_ONLY`** iz gornjeg primjera koja označava da se rezultatni set koristi samo za čitanje i nije moguće ažuriranje (engl. *update*) redaka ili može poprimiti vrijednost **`ResultSet.CONCUR_UPDATABLE`** koja označava da se rezultatni set može koristiti i za ažuriranje redaka koje sadrži (u nastavku laboratorijske vježbe biti će primjer ažuriranja redaka).

Svojstvo upravljanja kursorom može poprimiti vrijednost **`HOLD_CURSORS_OVER_COMMIT`** koje označava da će kursor biti otvoren (engl. *open*) i nakon što se transakcija potvrdi (engl. c*ommit*) ili vrijednost **`CLOSE_CURSOR_AT_COMMIT`** koja označava da se kursor zatvara s potvrđivanjem transakcije.

SQL naredbe u programskom jeziku Java izvode se korištenjem objekata koji implementiraju sučelje **`Statement`**. Ti objekti najčešće se ne instanciraju "ručno", već se dobivaju izvršavanjem metode **`createStatement`** iz objekta **`Connection`**:

```
// otvaranje veze (konekcije) s bazom podataka
Connection veza = DriverManager.getConnection(
             "jdbc:derby:C:/Users/Aleksander/MyDB","test","test");

// kreiranje objekta za izvršavanje upita
Statement stmt = veza.createStatement(
          ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
```

Metoda **`createStatement`** iz gornjeg primjera prima dva parametra (može ih primati i više i manje od dva, detalje je moguće pronaći u javadoc dokumentaciji), prvi **`ResultSet.TYPE_FORWARD_ONLY`** opisuje tip rezultatnog seta kojemu je kursor moguće pomicati samo unaprijed, a drugi **`ResultSet.CONCUR_READ_ONLY`** da se rezultatni set

koristi samo za čitanje (bez ažuriranja redaka).

Nakon kreiranja objekta tipa **`Statement`**, moguće je izvršavati SQL naredbe nad bazom pozivanjem sljedeće metode:

```
// izvođenje upita i dohvaćanje rezultata upita
ResultSet rs = stmt.executeQuery("SELECT * FROM RAZVOJ.STUDENTI");
```

Metoda **`executeQuery`** prima SQL upit u obliku String-a iz kao rezultat vraća objekt tipa **`ResultSet`** koji sadržava rezultat izvršenja navede SQL naredbe. Sučelje **`ResultSet`** sadržava niz metoda koje je moguće koristiti kako bi se dohvatili podaci iz dobivenog rezultatnog seta nakon izvođenja zadane SQL naredbe:

•**`next()`** - pomiče kursor unaprijed za jedan redak. Vraća logičku vrijednost **`true`** ukoliko kursor nakon pomicanja pokazuje na neki redak ili vraća logičku vrijednost **`false`** ukoliko je kursor pozicioniran iza posljednjeg retka.

•**`previous()`** - pomiče kursor unatrag za jedan redak. Vraća logičku vrijednost **`true`** ukoliko kursor nakon pomicanja pokazuje na neki redak ili vraća logičku vrijednost **`false`** ukoliko je kursor pozicioniran prije prvog retka.

•**`first()`** - pomiče kursor na prvi redak rezultatnog seta. Vraća logičku vrijednost **`true`** ukoliko rezultatni set sadrži barem jedan redak ili vraća logičku vrijednost **`false`** ukoliko rezultatni set ne sadrži nijedan redak.

•**`last()`** - pomiče kursor na posljednji redak rezultatnog seta. Vraća logičku vrijednost **`true`** ukoliko rezultatni set sadrži barem jedan redak ili vraća logičku vrijednost **`false`** ukoliko rezultatni set se sadrži nijedan redak.

•**`beforeFirst()`** - pozicionira kursor na početak rezultatnog seta, odnosno, prije prvog retka. Ukoliko rezultatni set ne sadrži nijedan redak, pozivanje ove metode u tom slučaju nema učinka.

•**`afterLast()`** - pozicionira kursor na kraj rezultatnog seta, odnosno, nakon posljednjeg retka. Ukoliko rezultatni set ne sadrži nijedan redak, pozivanje ove metode u tom slučaju nema učinka.

•**`relative(int rows)`** – pomiče kursor relativno s obzirom na trenutnu poziciju za zadani broj redaka.

•**`absolute(int rows)`** – pomiče kursor na redak zadan ulaznim parametrom metode.

Osim pomicanja kursora po retcima rezultatnog seta, sučelje **`ResultSet`** sadrži i niz metoda za dohvaćanje podataka (stupaca) iz pojedinog retka na koji u tom trenutku pokazuje kursor. Postoji zasebna metoda za svaki tip podatka, tzv. *getter* metode: **`getBoolean`**, **`getLong`**, **`getString`**, **`getDate`** itd. Svaka od metoda može primiti **redni broj stupca** (počevši od 1.) koji se dohvaća iz trenutnog retka ili **naziv stupca** u bazi podataka (naziv stupca može biti pisan velikim ili malim slovima – *case insensitive*). Povratna vrijednost svake od metoda ovisi o vrsti podataka koji se dohvaća (npr. **`getString`** vraća **`String`** itd.).

Koristeći navedene metode, rezultatni set moguće je na konzolu ispisati na sljedeći način:

```
// izvođenje upita i dohvaćanje rezultata upita
ResultSet rs = stmt.executeQuery("SELECT * FROM RAZVOJ.STUDENTI");

// analiza rezultata koji može predstavljen u više redaka (rows)
while (rs.next()) {
     int id = rs.getInt("id");
     String jmbag = rs.getString("jmbag");
     String ime = rs.getString("ime");
     String prezime = rs.getString("prezime");
     Date date = rs.getDate("datum_rodjenja");

     System.out.println("Pročitani redak: " + id + " "
     + jmbag + " " + ime + " " + prezime + " " + date);
}
```

10.1.4. Ažuriranje podataka u bazi podataka

Ažuriranje podataka obavlja se u dvije faze: **postavljanje nove vrijednosti** u stupac trenutnog retka i **spremanje nove vrijednosti** u bazu podataka. Nova vrijednost ne sprema se u bazu podataka sve dok ne završi druga faza ažuriranja.

Sučelje **`ResultSet`** sadrži po dvije metode za ažuriranje svakog tipa podatka. Jedna prima redni broj stupca koji se ažurira, a druga prima naziv stupca koji se ažurira. Na primjer, za ažuriranje stupca koji sadržava datum može se postići korištenjem jedne od ove dvije metode:

```
updateDate(int columnIndex, Date x);
updateDate(String columnLabel, Date x);
```

Prije samog ažuriranje potrebno je kreirati **`ResultSet`** objekt koji ima svojstvo ažuriranja podataka. To je moguće definirati ulaznim parametrom **`ResultSet.CONCUR_UPDATABLE`** metode **`createStatement`** koji sadrži objekt **`Connection`**:

```
Statement stmt = veza.createStatement(
     ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
```

Nakon toga potrebno je kreirati objekt tipa **`ResultSet`** i ažurirati željene stupce u retku na koji pokazuje kursor pomoću metode **`updateString`** (prva faza ažuriranja), te zapisivanje podataka u bazu pomoću metode **`updateRow`** (druga faza ažuriranja).

```
// izvođenje upita i dohvaćanje rezultata upita
ResultSet rs = stmt.executeQuery("SELECT * FROM RAZVOJ.STUDENTI");

// analiza rezultata koji može biti predstavljen u više redaka (rows)
while (rs.next()) {
...
     if (jmbag.equals("0036374849")) {
          rs.updateString("ime", "Ivan");
          rs.updateString("prezime", "Horvat");
          rs.updateRow();
     }
}
```

10.1.5. Korištenje pripremljenih naredbi

Pripremljene naredbe (engl. *Prepared statements*) koriste se u slučaju kad je jednu SQL naredbu potrebno pozvati više puta. Za razliku kad je SQL naredbu potrebno izvesti samo jedanput pomoću objekta tipa **`Statement`**, pripremljene naredbe koriste se pomoću objekta tipa **`PreparedStatement`**.

Osnovna prednost pripremljenih naredbi (**`PrepredStatement`**) s obzirom na "obične" naredbe (**`Statement`**) je u tome što se pripremljene naredbe moraju prevoditi (engl. *compile*) samo jednom, a više puta izvoditi. Za razliku od nepripremljenih "običnih" naredbi koje je potrebno prevoditi prilikom svakog izvođenja (manje učinkovitije nekog kod pripremljenih naredbi).

Pripremljene naredbe najčešće se koriste s promjenjivim parametrima koje je moguće definirati prije izvođenja same SQL naredbe. Svaki od promjenjivih parametara unutar SQL naredbe označen je sa znakom "?". Na primjer, pripremljenu naredbu koja ažurira podatke o studentima moguće je kreirati na sljedeći način, pomoću metode **`preparedStatement`** iz objekta tipa **`Connection`**:

```
// otvaranje veze (konekcije) s bazom podataka
Connection veza = DriverManager.getConnection(
           "jdbc:derby:C:/Users/Aleksander/MyDB", "test", "test");

// kreiranje pripremljene naredbe
PreparedStatement updateStudenti = veza.prepareStatement("UPDATE
RAZVOJ.STUDENTI SET
IME = ? WHERE JMBAG = ?");
```

Prije svakog pozivanja pripremljene naredbe potrebno je definirati promjenjive parametre kako bi SQL naredba bila potpuna. Definiranje vrijednosti moguće je obaviti pomoću metoda čiji naziv ovisi o tipu podatka koji se definira (npr. ako se definira parametar tipa String, koristi se metoda **`setString`** itd.):

```
updateStudenti.setString(1, "Željko");
updateStudenti.setString(2, "0024568238");
updateStudenti.executeUpdate();
```

Metoda **`executeUpdate`** koja ne prima nikakve parametre koristi se za izvođenje naredbi pomoću **`Statement`** i **`PreparedStatement`** objekata i potrebno ju je pozvati nakon definiranja vrijednosti svih parametara.

Za razliku od metode **`executeQuery`** koja vraća objekt tipa **`ResultSet`**, metoda **`executeUpdate`** vraća cjelobrojni broj koji označava koliko je redaka ažurirano nakon izvođenja SQL naredbe.

10.1.6. Zatvaranje i oslobađanje resursa za komunikaciju s bazom podataka

Slično kao i kod datoteka, kod baza podataka je također poželjno oslobađati sve resurse nakon njihovog korištenja. To se odnosi na objekte tipa **`Connection`**, **`Statement`**, **`PreparedStatement`** i **`ResultSet`**.

Svako od navedenih sučelja sadržava metodu **`close()`** kojom se vrši oslobađanje resursa. *Javadoc* dokumentacija sadržava detalje svake od metoda, odnosno, informacije što je potrebno napraviti prije pozivanje metode i što se postiže pozivanjem metode.

10.1.7. Korištenje transakcija

U nekim slučajevima nije poželjno da se SQL naredbe odmah izvrše i promjene budu odmah vidljive u bazi podataka, jer bi time podaci postali nekonzistentni. Na primjer, ako je istovremeno potrebno ažurirati više redaka u bazi podataka pomoću različitih SQL naredbi, a potrebno je da se sve te naredbe obave u potpunosti ili da se uopće ne obave.

Takvo ponašanje moguće je postići korištenjem **transakcija** (engl. *transaction*). Transakcija je skup od jedne ili više SQL naredbi (engl. s*tatements*) koje se izvode zajedno kao jedna operacija (atomarno), što znači da se izvode ili sve naredbe zajedno ili nijedna.

Prilikom kreiranja veze s bazom podataka, ona se nalazi u **načinu rada s automatskim spremanjem u bazu podataka** (engl. auto-commit mode). To znači da se svaka SQL naredba tretira kao jedna transakcija i odmah nakon izvršenja naredbe promjene se spremaju (engl. *commit*) u bazu.

Za isključivanje tog načina rada kako bi se moglo koristiti više SQL naredbi unutar jedne transakcije moguće je postići pozivanjem sljedeće metode (objekt veza je tipa `Connection`):

```
veza.setAutoCommit(false);
```

Nakon što se isključi način rada s automatskim spremanjem u bazu podataka, promjene se ne spremaju u bazu podataka iako se SQL naredbe izvršavaju. Tek nakon poziva metode `commit()` sve promjene koje su do tada izvršene, a nisu spremljene, spremaju se u bazu:

```
//isključivanje načina rada s automatskim spremanjem
veza.setAutoCommit(false);

//kreiranje pripremljene naredbe za ažuriranje podataka
PreparedStatement updateStudenti1 =
veza.prepareStatement(
         "UPDATE RAZVOJ.STUDENTI SET IME = ? WHERE JMBAG = ?");
updateStudenti1.setString(1, "Petar");
updateStudenti1.setString(2, "0024568238");
updateStudenti1.executeUpdate();

//kreiranje pripremljene naredbe za ažuriranje podataka
PreparedStatement updateStudenti2 =
veza.prepareStatement(
        "UPDATE RAZVOJ.STUDENTI SET PREZIME = ? WHERE JMBAG = ?");
updateStudenti2.setString(1, "Ivičić");
updateStudenti2.setString(2, "0024568238");
updateStudenti2.executeUpdate();

//spremanje podataka u bazu (do tog trenutka promjene nisu
vidljive u bazi)
veza.commit();
```

Gornji primjer pokazuje jedan od načina kako se koriste transakcije u slučaju kad programer sam odlučuje kada će promjene biti vidljive u bazi. Osim tog načina postoji i

način u kojem se koristi metoda **`setTransactionIsolation`** koja definira na koji način se tretiraju promjene nad bazom koje nisu spremljene u bazu. Više detalja o tom načinu nalazi se u *javadoc* dokumentaciji.

Transakcije je moguće i podijeliti na više dijelova, odnosno, **točaka spremanja** (engl. *savepoint*) koje mogu odlučivati koji dio transakcije će biti spremljen u bazu podataka, a koji dio neće u slučaju pogrešaka ili nepredviđenih situacija.

Za tu svrhu koriste se metode **`setSavePoint`** i **`rollback`** iz sučelja **`Connection`**. Metoda **`setSavePoint`** služi za postavljanje točke spremanja u određenoj fazi transakcije, a metoda **`rollback`** služi za odbacivanje svih promjena koje do tada nisu bile spremljene u bazu podataka. Metoda **`rollback`** se često koristi i bez točaka spremanja, ukoliko tijekom izvršavanja transakcije dođe do bacanja iznimke i potrebno je poništiti sve promjene. Na primjer, u sljedećem programskom odsječku odbacuju se sve promjene izvršene do prve točke spremanja:

```
veza.setAutoCommit(false);
PreparedStatement updateStudenti1 =
veza.prepareStatement(
      "UPDATE RAZVOJ.STUDENTI SET IME = ? WHERE JMBAG = ?");
updateStudenti1.setString(1, "Petar");
updateStudenti1.setString(2, "0024568238");
updateStudenti1.executeUpdate();

//kreiranje točke spremanja
Savepoint tockaSpremanja =
veza.setSavepoint("PrvaTockaSpremanja");

PreparedStatement updateStudenti2 =
veza.prepareStatement(
      "UPDATE RAZVOJ.STUDENTI SET PREZIME = ? WHERE JMBAG = ?");
updateStudenti2.setString(1, "Ivičić");
updateStudenti2.setString(2, "0024568238");
updateStudenti2.executeUpdate();

//odbacivanje svih promjena od prve točke spremanja pa nadalje
veza.rollback(tockaSpremanja);

//spremanje svih promjene koje su izvršene do prve točke spremanja
veza.commit();
```

Osim postavljanja, točke spremanja moguće se i **poništiti** (engl. *release*) pomoću metode **`releaseSavePoint`**.

10.2. Java datoteke za spremanje svojstava

Java datoteke za spremanje svojstava (engl. *Java properties files*) služe za spremanje parametara za konfiguriranje Java aplikacija. Mogu se koristiti i za spremanje poruka za višejezične aplikacije. Datoteke imaju ekstenziju ".properties" i njoj se u

tekstualnom obliku spremaju parova ključ-vrijednost, na primjer:

```
aplikacija.verzija = 10.4.5
```

gdje predstavlja ključ **`aplikacija.verzija`** prema kojem se dohvaća vrijednost **`10.4.5`**.

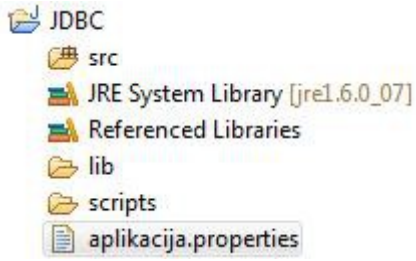
Svaka linija u datoteci za spremanje svojstava predstavlja jedan zapis. Osim gornjeg primjera zapisi mogu biti i u sljedećim oblicima:

```
aplikacija.verzija=10.4.5
aplikacija.verzija:10.4.5
aplikacija.verzija 10.4.5
```

Komentari u datotekama za spremanje svojstva počinju sa znakom: "**`#`**":

```
#Ovo je komentar u properties datoteci
```

Za korištenje datoteke za spremanje svojstava najprije je potrebno kreirati datoteku s ekstenzijom "**`.properties`**" u Eclipse razvojnom okruženju i dodati je u korijen (engl. *root*) Java projekta kao što je prikazano na sljedećoj slici (**`aplikacija.properties`**):



Slika 10.1. Dodavanje datoteke **`aplikacija.properties`** u Java projekt

Na primjer, u datoteci može biti sljedeći sadržaj:

```
#Osnovni podaci za spajanje na bazu podataka
bazaPodatakaUrl = jdbc:derby:C:/Users/Aleksander/MyDB
korisnickoIme = test
lozinka = test
```

Nakon dodavanja datoteke potrebno je korištenjem klase **`java.util.Properties`** inicijalizirati i dohvatiti potrebne vrijednosti iz datoteke:

```
//kreiranje objekta klase java.util.Properties
Properties svojstva = new Properties();

//inicijalizacija parova ključ-vrijednost
svojstva.load(new FileInputStream("aplikacija.properties"));

//dohvaćanje vrijednosti pomoću ključeva
String urlBazePodataka = svojstva.getProperty("bazaPodatakaUrl");
```

```
String korisnickoIme = svojstva.getProperty("korisnickoIme");
String lozinka = svojstva.getProperty("lozinka");
```

Nakon dohvaćanja vrijednosti pomoću ključeva u obliku **`String`** objekata, te vrijednosti mogu se iskoristiti u ostatku programskog koda, na primjer za povezivanje s bazom podataka umjesto zapisivanja tih vrijednosti u Java programski kod

```
Connection veza = DriverManager.getConnection(
                urlBazePodataka, korisnickoIme, lozinka);
```

Prednost Java datoteka za spremanje svojstava je u tome što se izbjegava pisanje nepotrebnih podataka u izvorni kod (engl. *source code*), već se ti podaci nalaze u vanjskim datotekama koje je puno lakše održavati. Nadalje, promjenom sadržaja tih datoteka nije potrebno prevoditi cijeli programski kod, što bi bilo potrebno napraviti uvijek ukoliko se ne bi koristile *properties* datoteke. Najraširenija primjena *properties* datoteka je u lokaliziranim (višejezičnim) aplikacijama gdje se prijevod poruka na određeni jezik nalazi u zasebnoj datoteci. Java datoteke za spremanje svojstava korištene za tu namjenu često se nazivaju i *Property Resource Bundles*.

10.3. Dodaci

U ovom poglavlju opisuju se dodaci (engl. *plugins*) za razvojno okruženje Eclipse koji olakšavaju rad s bazom podataka.

10.3.1. Apache Derby baza podataka

Apache Derby baza podataka je relacijska baza podataka zajednice otvorenog koda (engl. *open source relational database*) u potpunosti implementirana u programskom jeziku Java i dostupna pod licencom *Apache License, Version 2.0* (http://db.apache.org/derby/license.html).

Neke od prednosti Derby baze podataka su sljedeće:

•Mehanizam baze podataka (engl. *base engine*) i ugrađeni pogonski program (engl. *embedded JDBC driver*) zauzimaju vrlo malo diskovnog prostora

•Bazirana je na Java, JDBC i SQL standardima

•Ugrađeni pogonski program omogućava korištenje Derby baze podataka u svim programskim rješenjima baziranih na programskom jeziku Java

•Može se koristiti u klijent-poslužitelj načinu rada (engl. *client-server mode*)

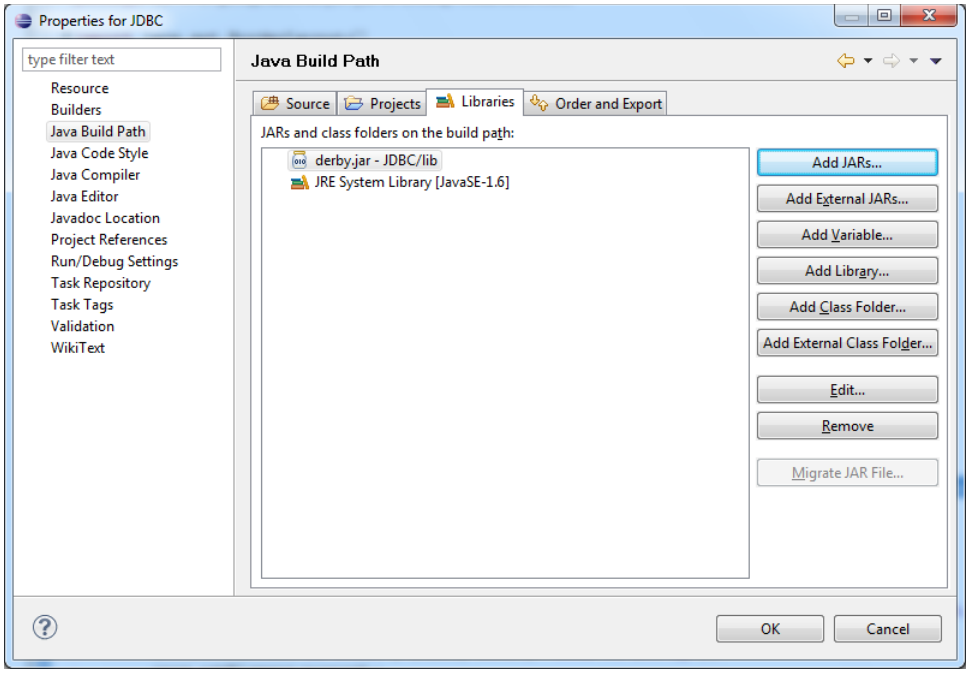
•Jednostavna za instalaciju i korištenje

Osnovne informacije o Derby bazi podataka moguće je pronaći na web adresi http://db.apache.org/derby/, a upute i osnove korištenja nalaze se na web adresi http://db.apache.org/derby/quick_start.html.

Posljednju inačicu Derby baze podataka moguće je preuzeti (engl. *download*) s web adrese http://db.apache.org/derby/derby_downloads.html.

Nakon preuzimanja potrebno je otpakirati zip arhivu, iz "lib" mape uzeti datoteku "**derby.jar**", kopirati je u "lib" mapu Java projekta unutar Eclipse razvojnog sučelja koji će koristiti bazu podataka i dodati u putanju klasa (engl. *classpath*) tog projekta. Način na koji se datoteke dodaju u putanju klasa opisan je poglavlju 6.8. (koje opisuje Logback).

Java putanja klasa projekta u koji se dodaju navedene *jar* datoteke nakon dodavanja mora izgledati ovako:



10.2. Ekran s Java putanjom klasa Eclipse projekta nakon dodavanja *jar* datoteka za Derby bazu podataka

Za Derby bazu podataka postoji i mnoštvo dokumentacije koju je moguće besplatno preuzeti s web stranice http://db.apache.org/derby/manuals/index.html. Jedan od najkorisnijih dokumenata je "Derby Reference Manual" koji sadržava veliki broj korisnih informacije o tome kako koristiti Derby bazu podataka.

10.3.2. "Data Tools Platform" dodatak za Eclipse

Data Tools Platform (DTP) predstavlja dodatak (engl. *plugin*) za razvojno okruženje Eclipse koji omogućava obavljanje osnovnih operacija s bazom podataka kao što su spajanje ili izvršavanje upita (engl. *Queries*). Opširnije informacije mogu se pronaći na web adresi http://www.eclipse.org/datatools/.

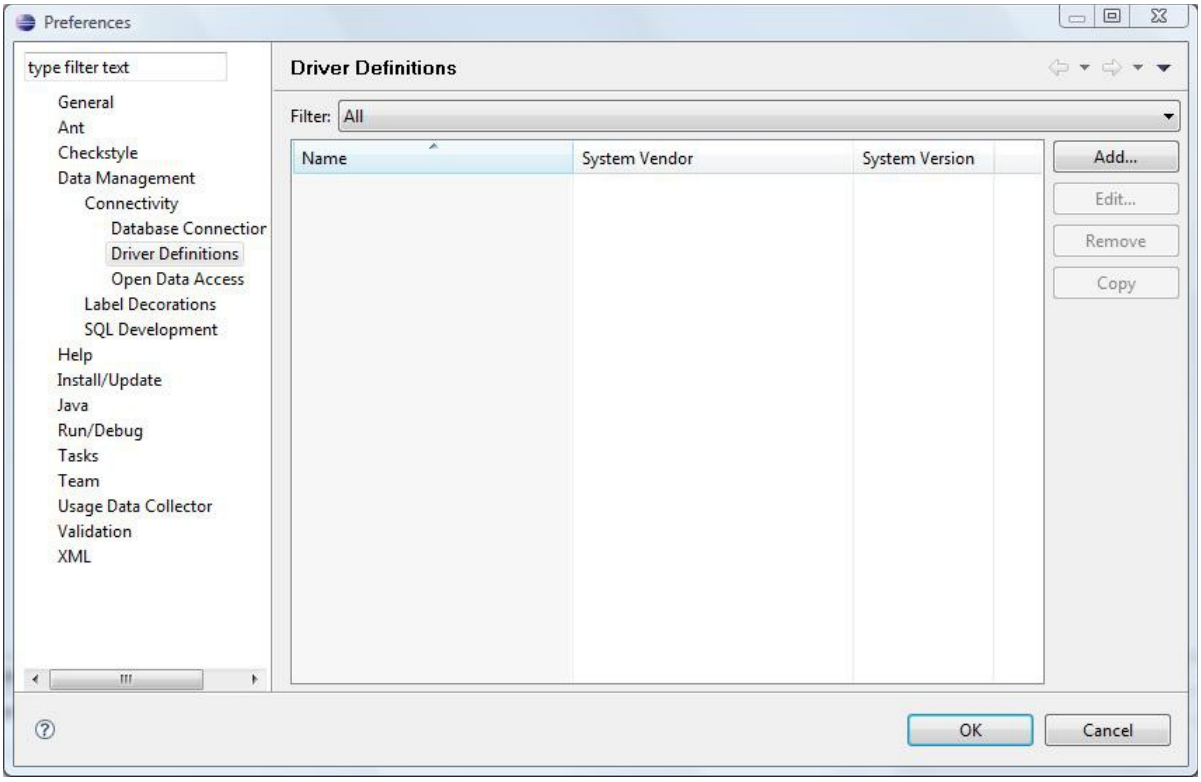
Posljednju inačicu DTP dodatka moguće se preuzeti s web adrese http://www.eclipse.org/datatools/downloads.php.

Za rješavanje zadatka za laboratorijsku vježbu dovoljno je preuzeti datoteku bez izvornog koda i Javadoc dokumentacije, međutim, na istoj web stranici moguće je preuzeti zip arhivu koja uključuje i navedene dijelove.

Nakon preuzimanja i raspakiravanja zip arhive, mapa "eclipse" sadrži podmape "features" i "plugins" čije sadržaje je potrebno kopirati u istoimene mape na lokaciji gdje je instalirano Eclipse razvojno okruženje (npr. C:\eclipse\features i C:\eclipse\plugins). Postupak instalacije dodatka u razvojno okruženje Eclipse opisan je u poglavlju 2.4.

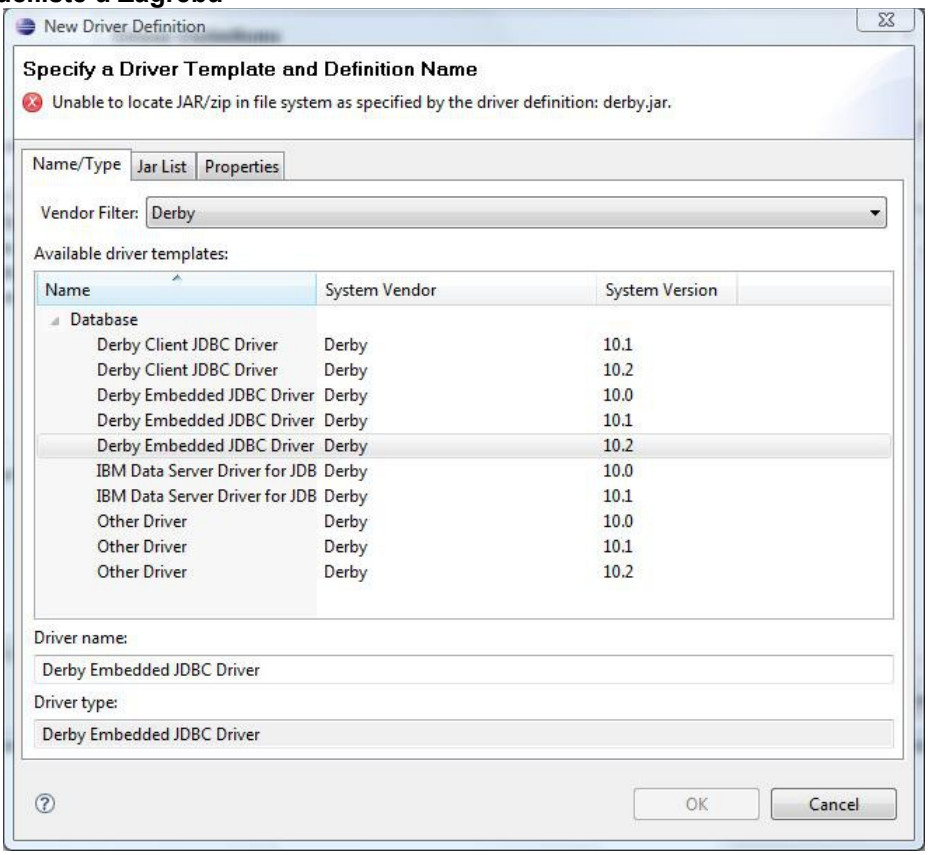
Nakon instalacije DTP dodatka potrebno je pokrenuti Eclipse razvojno okruženje i definirati

pogonski program (engl. *driver*) za Derby bazu podataka koja će se koristiti prilikom razvijanja programa. To je moguće postići otvaranjem prozora Window->Preferences i odabirom opcije Data Management->Connectivity->Driver Definitions:



Slika 10.3. Ekran za definiranje pogonskih programa za bazu podataka

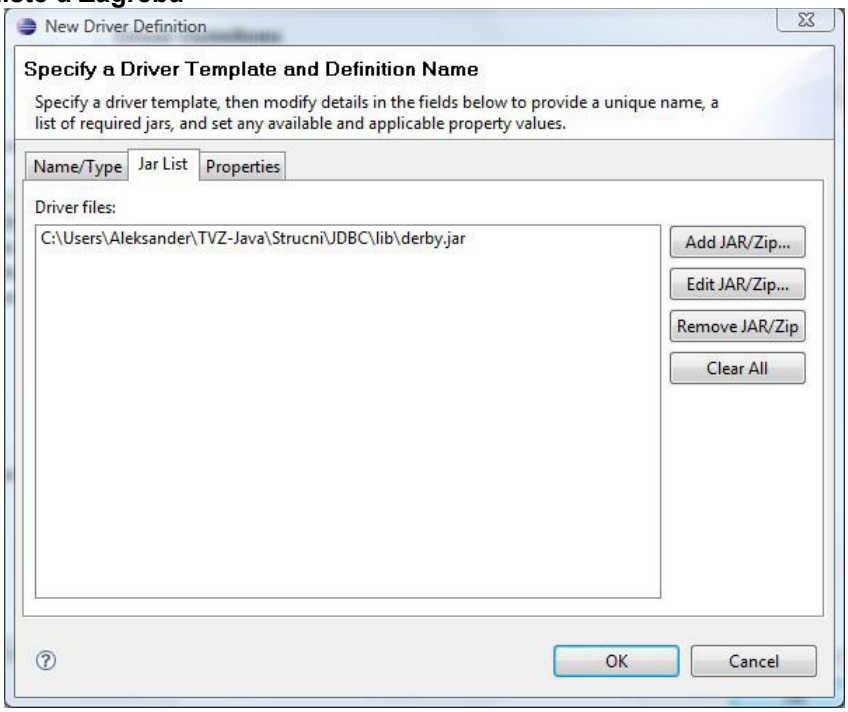
Na ekranu prikazanom na slici 10.3. potrebno je definirati novi pogonski program pritiskom na gumb "Add...". Slika 10.4. prikazuje ekran koji se pojavljuje nakon toga na kojem je potrebno odabrati pogonski program za Derby bazu podataka inačice 10.2. Prvo je pod opcijom "Vendor Filter" potrebno odabrati bazu podataka "Derby", nakon čega je u dijelu prozora "Available driver templates" potrebno odabrati "Derby Embedded JDBC driver" s inačicom sustava (engl. *System version*) 10.2. kao što je i prikazano na slici 10.3.



Nakon toga se u gornjem dijelu ekrana prikazanog na slici 10.4. pojavljuje poruka o pogrešci koja govori da nije moguće pronaći datoteku "derby.jar" koja sadrži navedeni pogonski program.

10.4. Ekran za dodavanje novog pogonskog programa

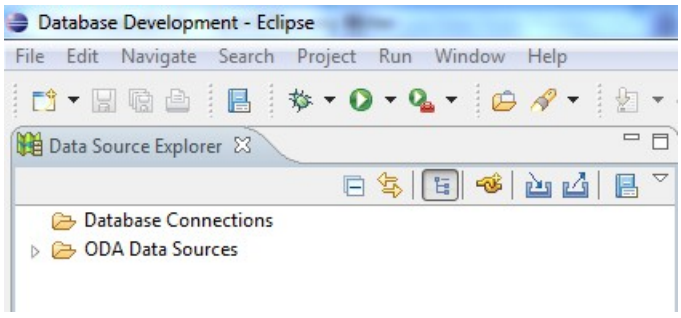
Navedenu pogrešku je moguće ispraviti dodavanjem potrebne datoteke odabirom *tab*-a "Jar List" i pomoću opcije "Add JAR/Zip..." dodati datoteku "derby.jar" koja se nalazi u "lib" mapi unutar Java projekta koji koristi bazu podataka (mapa "lib" bilo je potrebno kreirati u sklopu poglavlja 10.3.1). Slika 10.5. prikazuje izgled ekran na kojem je ta datoteka dodana:



10.5. Ekran s dodanom "derby.jar" datotekom

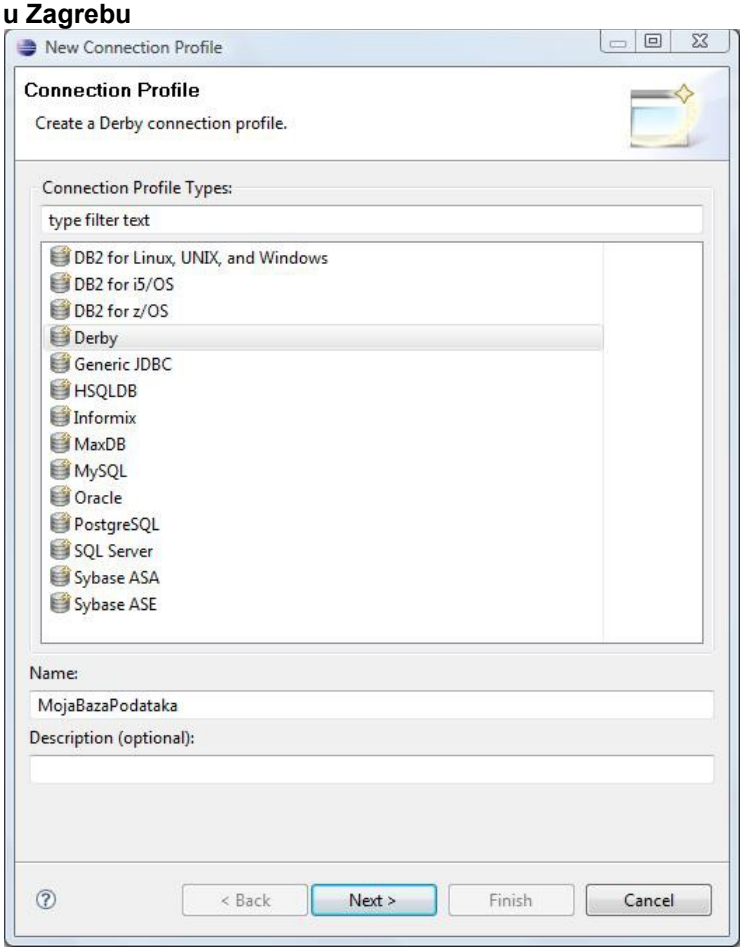
Nakon potvrde dodavanja datoteke prikazuje se ekran sličan onome na slici 10.3., samo što je dodan pogonski program za Derby bazu podataka.

Sljedeći korak nakon definiranja pogonskog programa je kreiranje veze (engl. *connection*) na bazu podataka. Da bi se to postiglo, potrebno je otvoriti novu perspektivu pod nazivom "*Database Development*" (Window->Open Perspective->Database Development). Dio navedene perspektive je pogled *Data Source Explorer* koji sadržava mapu "Databases" u kojem je potrebno kreirati vezu na bazu podataka (slika 10.6.).



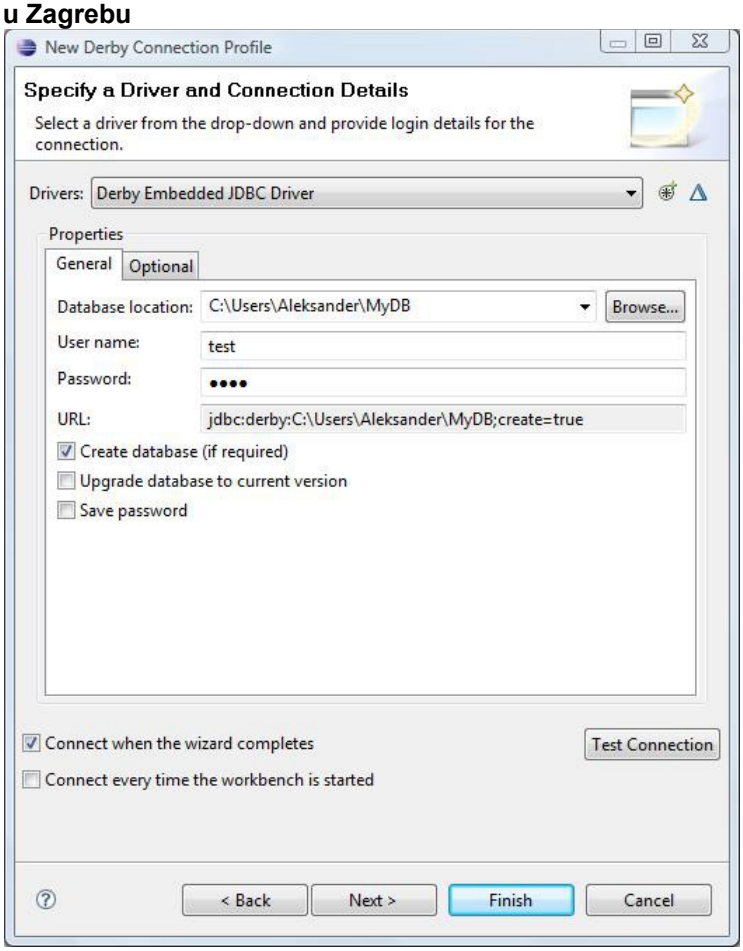
Slika 10.6. Dio "Database Development" perspektive koja sadrži "Data Source Explorer" pogled

Kreiranje nove veze s bazom podataka i kreiranje nove baze podataka moguće je obaviti pritiskom desne tipke miša nad mapom "Database Connections" unutar pogleda "Data Source Explorer" i odabirom opcije "New...". Nakon toga prikazuje se sljedeći prozor (slika 10.7.):



Slika 10.7. Ekran za odabir vrste baze podataka

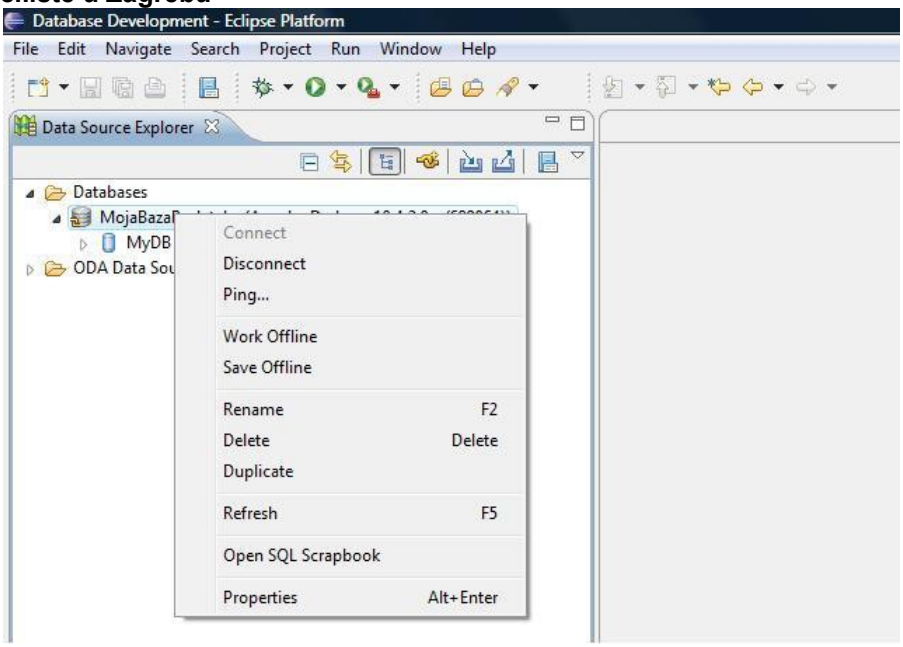
Odabirom opcije "Derby" i odabirom imena za bazu podataka definirani su osnovni podaci o bazi podataka. Pritiskom na tipku "Next" otvara se prozor (slika 10.8.) na kojem se definiraju dodatni detalji za spajanje bazu podataka, kao što su lokacija baze podataka na tvrdom disku, korisničko ime i lozinka, te URL:



Slika 10.8. Ekran za definiranje dodatnih detalja o vezi s bazom podataka

Korisničko ime, lozinka i URL će se koristiti u programskom kodu koji služi za kreiranje veze s bazom podataka. Osim toga je moguće iskoristiti opcije kao što su automatsko kreiranje baze podataka ukoliko ona ne postoji, automatsko spajanje na bazu podataka ili testiranje veze s bazom podataka, kao što je prikazano na slici 10.8.

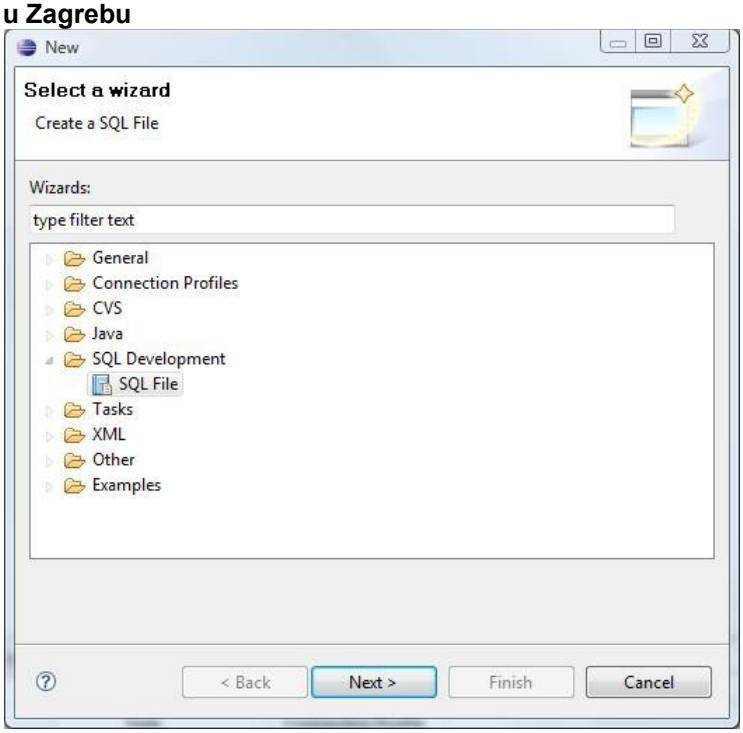
Pritiskom na tipku "Finish" završava kreiranje veze s bazom podataka nakon čega je moguće i koristiti tu novokreiranu bazu. Ako je na ekranu prikazanom na slici 10.8. odabrana opcija da se nakon završetka konfiguriranja baze podataka veza s bazom automatski uspostavlja, u tom slučaju nije potrebno napraviti ništa drugo jer je veza već ostvarena. Međutim, ponovnim pokretanjem Eclipse razvojnog okruženja, tu vezu je često potrebno nanovo uspostaviti ili ponekad i prekinuti kako bi se mogla koristiti iz drugih programa. Opcije za uspostavljanje veze s bazom podataka ("Connect") ili prekidanje veze s bazom podataka ("Disconnect") moguće je pozvati iz izbornika koji je moguće dobiti pritiskom desne tipke miša nad ikonom baze podataka kao što je prikazano na sljedećoj slici:



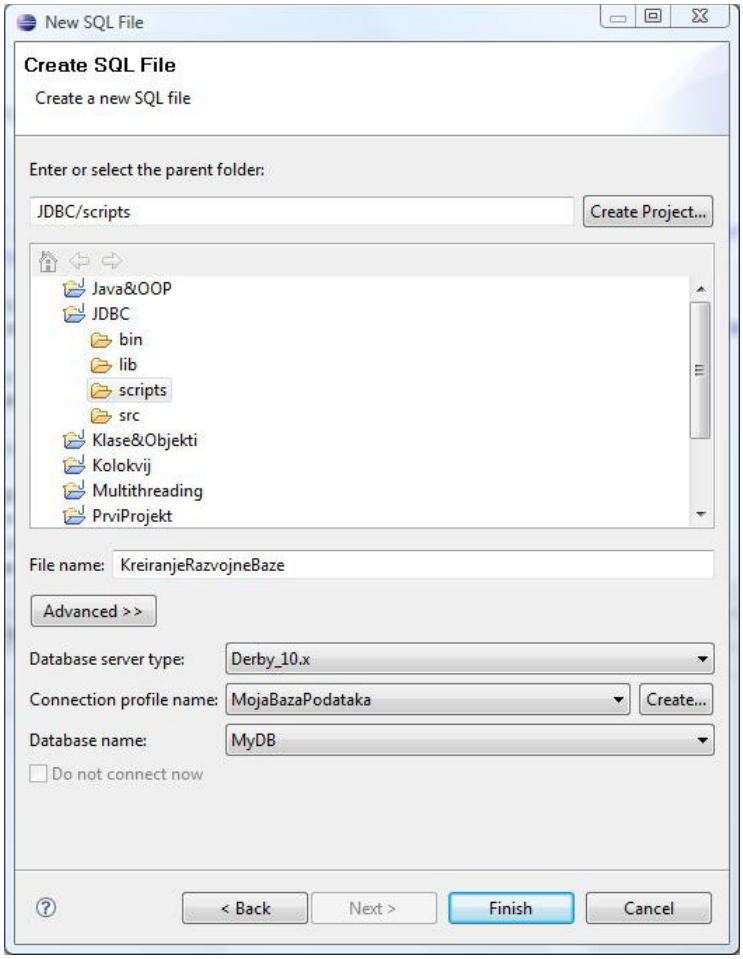
10.9. Izbornik za ostvarivanje ili prekidanje veze s bazom podataka

10.3.3. Primjer kreiranja baze podataka pomoću dodatka DTP

Dodatak DTP osim ostvarivanja veze na bazu podataka nudi mogućnost i izvođenja upita i ostalih naredbi nad bazom podataka kao što je ažuriranje redaka ili dodavanje novih redaka. U sklopu dodatka DTP dodan je i novi tip datoteke koji služi za pisanje SQL naredbi za bazu podataka. Datoteku te vrste moguće je u Java projekt dodati odabirom sljedeće opcije: File->New->Other...->SQL Development->SQL File, kao što je prikazano na slici 10.10.



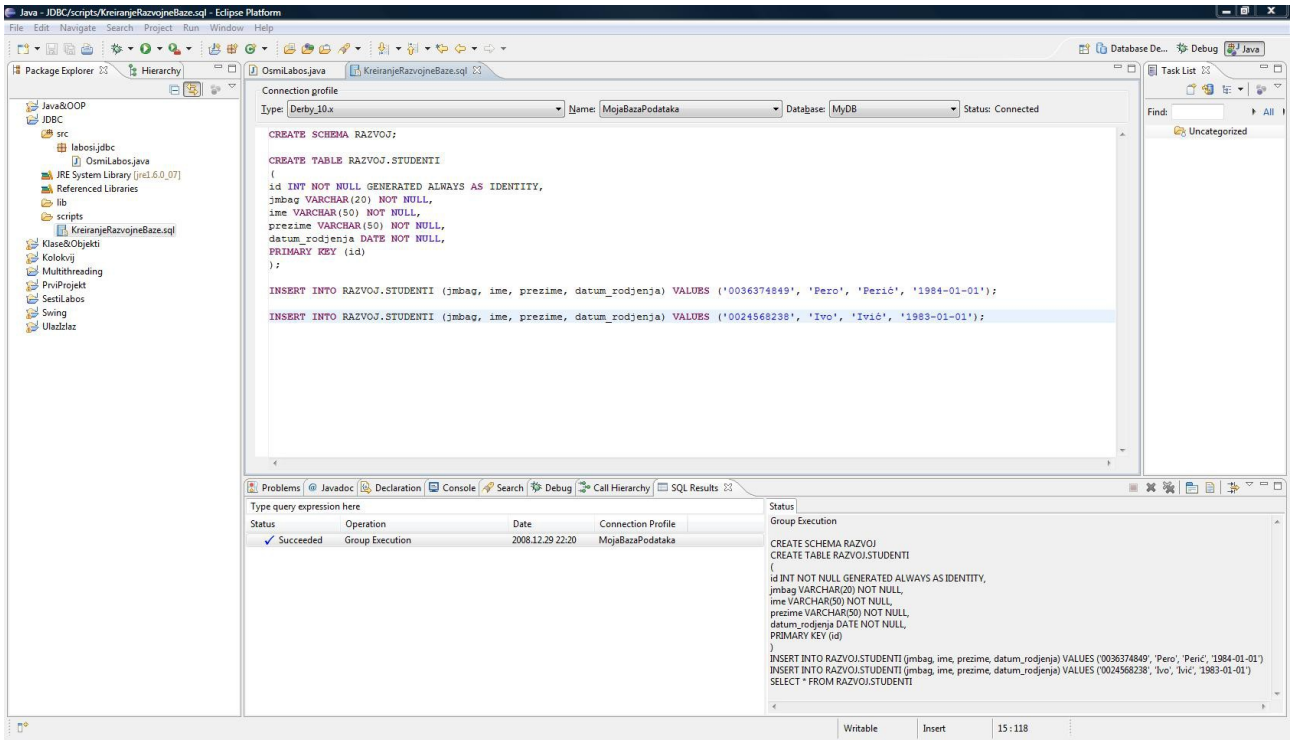
Slika 10.10. Ekran za dodavanje datoteke za pisanje i izvršavanje SQL narebi nad bazom podataka



Slika 10.11. Ekran za definiranje detalja SQL datoteke vezanih za spajanje na bazu podataka

Pritiskom na tipku "Next" otvara se prozor u koji je potrebno upisati sve detalje vezane za bazu podataka nad kojom se žele izvršavati SQL naredbe, na primjer, kao što je prikazano na slici 10.12. Datoteka pod imenom "KreiranjeRazvojneBaze" kreirati će se u mapi

"scripts" unutar Java projekta "JDBC". SQL naredbe napisane unutar te datoteke izvoditi će se nad bazom podataka tipa Derby inačice 10, koristiti će se profil za spajanje na bazu nazvan "MojaBazaPodataka", a naziv baze podataka je "MyDB". Pritiskom na tipku "Finish" kreira se datoteka u kojoj je moguće pisati SQL naredbe koje će se izvoditi nad bazom podataka.



Slika 10.12. Ekran s razvojnim okruženjem Eclipse nakon izvođenja SQL naredbi

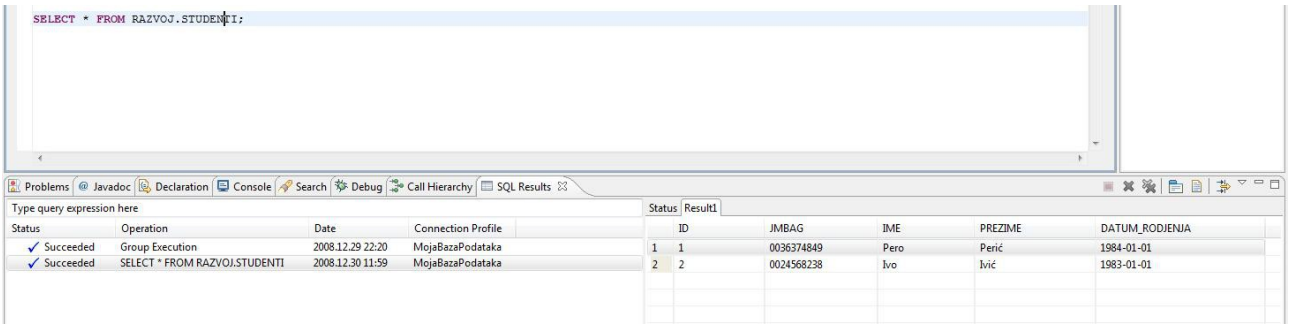
Ako je potrebno izvesti sljedeće SQL naredbe kako bi se kreirala razvojna baza podataka s tablicom STUDENTI koja se sadržavati dva zapisa o studentima, potrebno je ubaciti sljedeće naredbe u SQL datoteku:

```
CREATE SCHEMA RAZVOJ;

CREATE TABLE RAZVOJ.STUDENTI
(
id INT NOT NULL GENERATED ALWAYS AS IDENTITY,
jmbag VARCHAR(20) NOT NULL,
ime VARCHAR(50) NOT NULL,
prezime VARCHAR(50) NOT NULL,
datum_rodjenja DATE NOT NULL,
PRIMARY KEY (id)
);

INSERT INTO RAZVOJ.STUDENTI (jmbag, ime, prezime, datum_rodjenja)
VALUES ('0036374849', 'Pero', 'Perić', '1984-01-01');
INSERT INTO RAZVOJ.STUDENTI (jmbag, ime, prezime, datum_rodjenja)
VALUES ('0024568238', 'Ivo', 'Ivić', '1983-01-01');
```

Nakon ubacivanja navedenih SQL naredbi potrebno je izvesti navedene naredbe. To je moguće postići tako da se pritisne desna tipka miša na površini prozora sa SQL naredbama i odabere opcija "Execute All". Na slici 10.12. prikazan je ekran nakon izvođenja

navedenih SQL naredbi. Osim toga je moguće izvoditi naredbu po naredbu ako se označi navedena naredba pomoću miša i nakon pritiska desne tipke miša odabere opcija "Execute Selected Text".

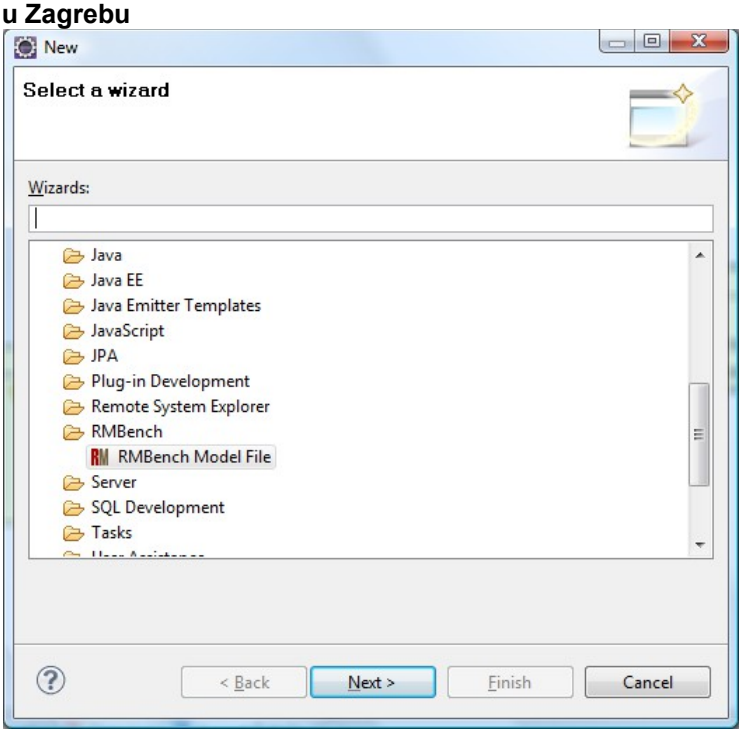
Na taj način moguće je izvoditi i upite nad bazom podataka i pregledavati rezultate izvršenja tih upita. Na slici 10.13. prikazan je dio ekrana na kojem je moguće vidjeti SQL naredbu koja je izvedena i s desne strane i rezultate same SQL naredbe "SELECT * FROM RAZVOJ.STUDENTI". Rezultat izvođenja upita su dva retka baze podataka koja su ubačena u bazu kako je opisano prethodnom dijelu vježbe.



Slika 10.13. Ekran s rezultatima izvođenja SQL naredbi

10.3.4. Dodatak za vizualno kreiranje baze podataka – RMBench

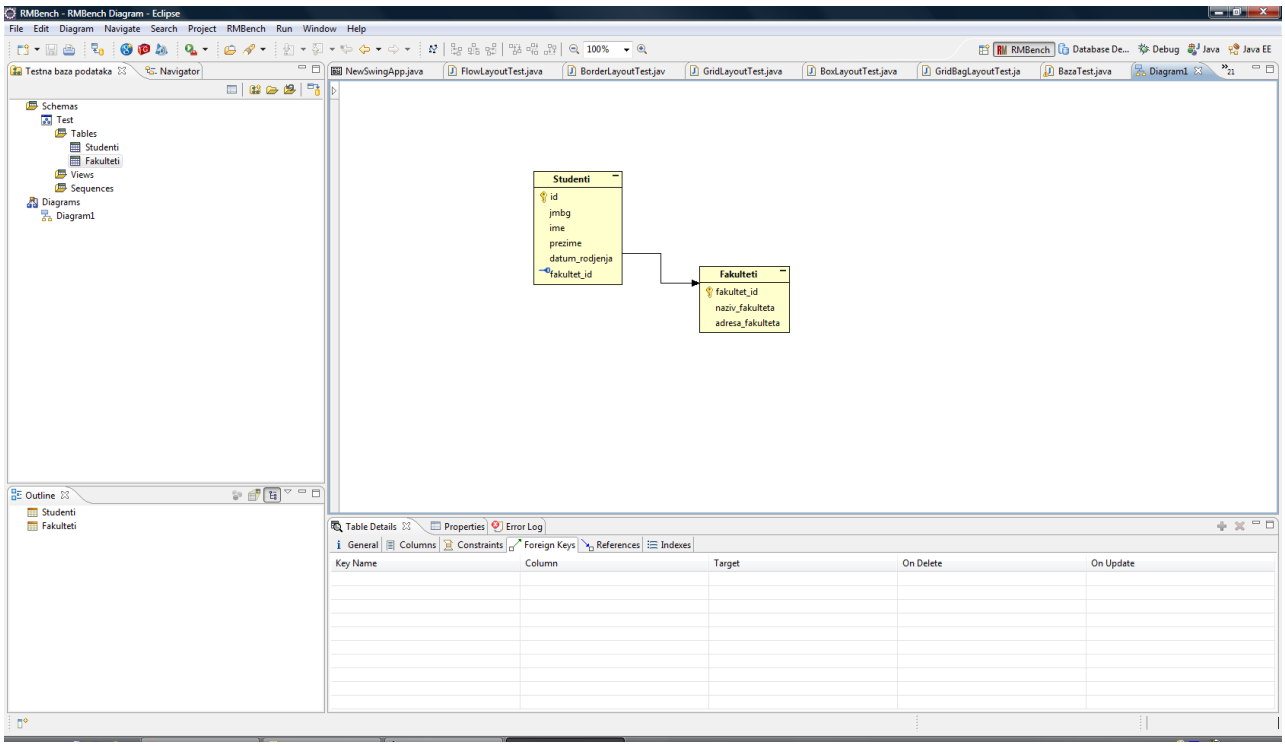
Osim "ručnog" kreiranja baze podataka pisanjem SQL koda, bazu je moguće modelirati i pomoću dodatka koji se naziva ***RMBench Relational Modeler***. Iako je besplatna inačica ograničena po pitanju generiranja SQL koda iz vizualnog dijagrama, moguće ga je iskoristiti za generiranje vizualnog dijagrama na osnovi gotove baze podataka. Dodatak je moguće instalirati u razvojno okruženje Eclipse pomoću update site-a: http://rmbench.com/update (detalji instalacijske procedure opisani su u poglavlju 2.4.). Više informacija o samom dodatku moguće je pronaći na stranici http://code.google.com/p/rmbench/.



Nakon uspješne instalacije moguće je kreirati posebnu vrstu datoteke *RMBench Model File* koja omogućava vizualno kreiranje baze podataka (slika 10.14.), te otvoriti pripadajuću perspektivu pod nazivom *RMBench*.

Slika 10.14. Vrsta datoteke za vizualno kreiranje baze podataka

Unutar RMBench perspektive moguće je vizualno kreirati tablice, ali zbog ograničenja licence nije moguće izravno kreirati SQL kod. Međutim, odabirom opcije *RMBench->Reverse Engineer* moguće je iz postojeće baze postupkom obrnutog inženjeringa dobiti SQL dijagram.



Slika 10.15. RMBench perspektiva

11. DODACI

11.1. Klasa java.util.Calendar

U javadoc dokumentaciji za klasu java.util.Date puno je metoda i konstruktora označeno zastarjelim (engl. deprecated). To znači da se ne preporuča korištenje tih metoda i konstruktora, te da postoji bolja i naprednija inačica klase koja obavlja istu funkcionalnost.

Za potrebe kreiranja određenog datuma (što se mora koristiti kod rješavanja zadatka za ovu laboratorijsku vježbu), postoji klasa **`Calendar`**. Osim kreiranja trenutnog i određenog datuma, **`Calendar`** se koristi za niz drugih operacija koje su detaljnije opisane u javadocu.

Da bi se te funkcionalnosti mogle iskoristiti, prvo je potrebno instancirati klasu **`Calendar`**:

```
Calendar cal = Calendar.getInstance();
```

Može se primijeniti da se za instanciranje ne koristi operator **`new`**, već da se nad klasom **`Calendar`** poziva statička metoda **`getInstance`**.

Nakon instanciranja, moguće je pozvati metode nad instancom **`cal`** iz gornjeg primjera:

```
cal.getTime(); // dohvaćanje trenutnog datuma

// postavljanje i ispis sutrašnjeg datuma
cal.set(Calendar.DATE, cal.get(Calendar.DATE + 1);
System.out.println(cal.getTime());

//postavljanje i ispis datuma 01.02.2018.
cal.set(Calendar.DATE, 1);
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
cal.set(Calendar.YEAR, 2018);
System.out.println(cal.getTime());
```

Klasa **`Calendar`** funkcionira na principu postavljanja i dohvaćanja podataka prema njihovim oznakama. Na primjer, prema gornjem primjeru može se primijetiti da je kod postavljanja nove vrijednosti dana u mjesecu u metodi set potrebno definirati podatak koji se postavlja (**`Calendar.DATE`**), te vrijednost na koju se podatak postavlja (1). Slično vrijedi i za postavljanje ostalih podataka kao što su mjesec i godina. Kod korištenja klase **`Calendar`** preporuča se maksimalno korištenje konstanti koje definira sama klasa. Npr. kod postavljanja mjeseca u naredbi

```
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
```

koristi se konstanta **`Calendar.FEBRUARY`** zbog toga što bi postavljanje mjeseca za vrijednost '2' zapravo značilo postavljanje na mjesec ožujak (**`Calendar.MARCH`**) zbog toga što brojanje mjeseci počinje od 0. Informacije je ostalim konstantama mogu se pronaći u javadocu.

12. LITERATURA

1.Bruce Eckel – Thinking in Java, 4th edition, Prentice Hall, 2007.
2.Christian Ullenboom - Java ist auch eine Insel, 8. Auflage, Galileo Computing, 2009.
3.Berthold Daum - Professional Eclipse 3 for Java Developers, Wiley, 2005.
4.Java for Programmers: Deitel Developer Series, Prentice Hall, veljača, 2009.
5.A Programmer's Guide to Java SCJP Certification: A Comprehensive Primer 3rd Edition, 2009.
6.Java Concurrency in Practice, Addion Wesley, svibanj, 2006.
7.Head First Java, 2nd edition, O'Reilly, veljača, 2005.
8.Java – The Good Parts, O'Reilly, svibanj, 2010.
9.Eclipse IDE Pocket Guide, O'Reilly, kolovoz, 2005.
10.Effective Java, 2nd edition, Prentice Hall, svibanj, 2008.
11.Robust Java: Exception Handling, Testing, and Debugging, Prentice Hall, rujan, 2004.
12.Java Collections, Apress, travanj, 2001.
13.Java I/O, O'Reilly, ožujak 1999.
14.Swing: A Beginners Guide, McGraw-Hill, rujan, 2006.
15.Swing, Second Edition, Manning, veljača, 2003.
16.Filthy Rich Clients, Prentice Hall, kolovoz, 2007