PROGRAMIRANJE I PROGRAMSKI JEZICI
OBJEKTNO ORJENTISANO PROGRAMIRANJE

Matematička gimnazija
2019

https://www.jetbrains.com/idea/download

Version: 2018.3.4
Build: 183.5429.30
Released: January 29, 2019

https://www.eclipse.org/downloads/

Literatura

- Oracle Java dokumentacija (http://docs.oracle.com/javase/tutorial), 2016
- Core Java Volume I ± Fundamentals (10th edition), Cay S. Horstmann-Gary Cornell, 2016
- Java (SE 7) Tom I - Osnove, prevod devetog izdanja, Cay S. Horstmann, Gary Cornell, 2013
- Objektno-orijentisano programiranje i programski jezik Java, Mirjana Ivanović, Zoran Budimac, Miloš Radovanović, Dejan Mitrović, 2016
- Programski jezik JAVA sa rešenim zadacima, Laslo Kraus, 2013

Pregled tema

- Istorijat
- Okruženje Java programa i uvodni primeri
- Osnovni elementi jezika Java
- Klase i objekti
- Nasleđivanje
- Interfejsi
- Izuzeci
- Generici
- Kolekcije

Istorijat

Nastanak i razvoj Jave

- Kako i zašto je nastala?
- Šta je čini tako važnom?

Nastanak i razvoj Jave

- Programski jezici se usavršavaju i razvijaju iz dva osnovna razloga
  - da bi se prilagodili izmenjenom okruženju
  - da bi se u njih ugradila poboljšanja i novosti iz oblasti programiranja

Poreklo Jave

- Java je srodnik jezika C++, koji je direktan potomak jezika C
- Veći deo svojih osobina Java je nasledila od ova dva jezika

  - Iz jezika C je preuzela sintaksu

  - Mnoge objektno orjentisane osobine Jave nastale su pod uticajem C++
- Nastaje u procesu prilagođavanja programskih jezika tako da reše neki od problema koji nije bio rešiv

Početak savremenog programiranja

- Programski jezici sa favorizovanim skupom osobima
- FORTRAN

  + pisanje efikasnih programa za primenu u nauci

  - nije pogodan za pisanje sistemskih programa
- BASIC

  + lako se uči

  - nije struktuiran tako da nije pogodan za pisanje dugačkih programa. GOTO glavna programska struktura, dugački programi jako teško razumljivi.
- Asembler

  + veoma efikasni programi

  - teško se uči i efikasno koristi. Teško se ispravljaju greške

Početak savremenog programiranja

- Pascal

  \+ struktuiran

  \- efikasnost nije bila osnovni cilj

- C

  Napisan 1972. godine od strane Denisa Ričija. Potreba za struktuiranim, efikasnim jezikom visokog nivoa koji može da zameni asemblerski kod za pisanje sistemskih programa.

  \+ moćan, efikasan, struktuiran, srazmerno se lako uči. Zamislili su ga, izgradili i razvijali, programeri za svoje potrebe po svom ukusu

  \- kod velikih projekata, složenost prevazilazi mogućnosti programera

Objektno orjentisana metodologija

- Krajem sedamdesetih, početkom osamdesetih godina prošlog veka, C postaje dominantan programski jezik
- Struktuirani jezik omogućava lako pisanje umereno složenih programa
- Kada projekat dostigne određenu veličinu, njegova složenost prevazilazi mogućnost programera
- Osmišljen je nov način programiranja, objektno orjentisano programiranje (OOP), koje je približnije ljudskom načinu razmišljanja i rešavanja problema
- Sastoji se od identifikovanja objekata, njihovih osobina i funkcija, postavljanje objekata u odgovarajuću sekvencu za rešavanje određenog problema

Sledeći korak: C++

- Osmislio ga je Bjarne Stroustrup 1979. godine
- Prvobitno ime "C sa klasama", zatim C++
- Stvoren kao poboljšanje jezika C objektno orjentisanom metodologijom
- Krajem osamdesetih, početkom devedesetih godina, OOP na jeziku C++ je uzelo maha
- Javlja se potreba za jezikom nezavisnim od računarske platforme
- Pravljenje softvera za različite elektronske uređaje u domaćinstvu, mikrotalasne pećnice, daljinske upravljače
- C i C++ se pri prevođenju moraju usmeriti na određeni procesor. Potreban je potpun prevodilac namenjen konkretom procesoru

Pojava Jave

- Koncipirana 1991. godine, James Gosling sa saradnicima iz korporacije Sun Microsystems, kasnije preuzima Oracle
- Nastaje za potrebe interaktivne televizije, ali tada infrakstruktura nije bila dovoljno uznapredovala za takav poduhvat
- Zvanično se pojavila 1995. godine kao univerzalan programski jezik
- Razvojem interneta, zbog raznolikosti arhitektura računara koji su povezani na internet, potreba za prenosivim programima značajno raste

Osobine Jave

- Izvršava se na specijalnoj virtuelnoj mašini
- Izvršni kod se ne mora kompilirati zasebno za pojedinačne arhitekture
- Izvršni kod se može koristiti na bilo kom računaru koji ima podršku za Javu
- Zapravo, Java prevodilac ne generiše izvršni kod, već tzv. bajt kod
- Bajt kod je visoko optimizovani skup instrukcija koji u trenutku izvršavanja tumači (interpretira) Java virtuelna masina (JVM)
- Za različite platforme je potrebno napraviti različite JVM
- Sve one razumeju isti Javin bajt kod

Osobine Jave 2

- Jednostavnost

  programeri mogu lako da je nauče i efikasno koriste.
- Robusnost

  Otpornost na greške. Izbegavaju se greške pri upravljanju memorijom. Obrada izuzetaka.
- Platformska nezavisnost

  Java ima sopstvenu softversku platfomu koja se prilagođava hardverskoj platformi.
- Bezbednost

  Zlonameran kod može da prouzrukuje štetu ako neovlašćeno pristupa sistemskim resursima.

  Program se ograničava na Java izvršno okruženje i ne dozvoljava pristup drugim delovima računara.

Osobine Jave 3

- Nezavisnost od arhitekture

  Nema karakteristika koje zavise od arhitekture, na primer, veličina primitivnih tipova je utvrđena.
- Prenosivost

  Javin bajtkod se može preneti na bilo koju platformu.
- Distibuiranost

  Pristupanje resursima pomoću URL se ne razlikuje od pristupanja datoteci

  Podržava i daljinsko pozivanje metoda.
- Višenitnost

  Moguće je pisati Java programe koji rade na mnogim zadacima odjednom tako što se definišu višestruke niti, koje dele istu memoriju.

Okruženje Java programa i uvodni primeri

Preuzimanje alata

1. Preuzeti Java Development Kit 8 (JDK 8) sa adrese
http://www.oracle.com/technetwork/java/javase/downloads

- Sadrži u sebi Java Runtime Environment (JRE), odnosno virtuelnu mašinu.

Izaberite x86 za 32-bitnu verziju, ili x64 za 64-bitnu verziju

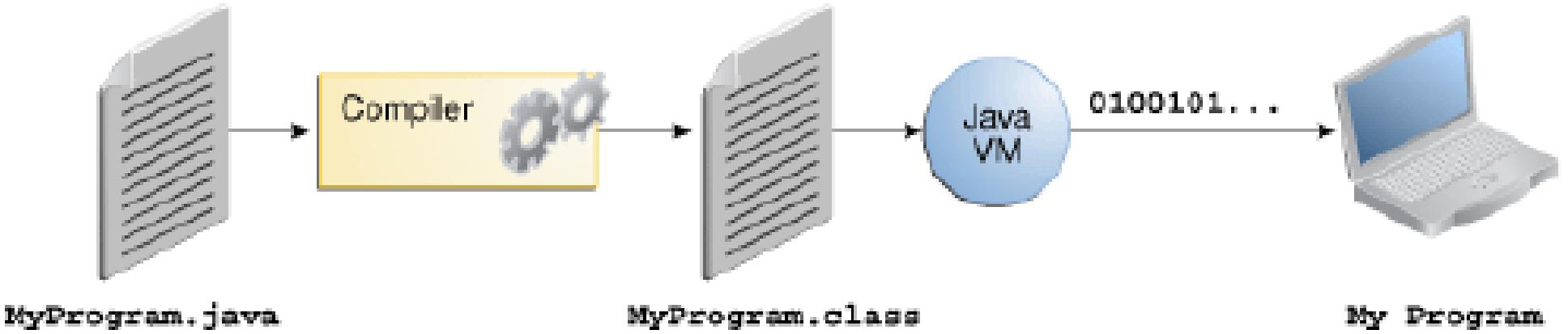
2. Preuzeti Eclipse razvojno okruženje sa adrese
https://www.eclipse.org/downloads

- Izabrati Eclipse IDE for Java Developers

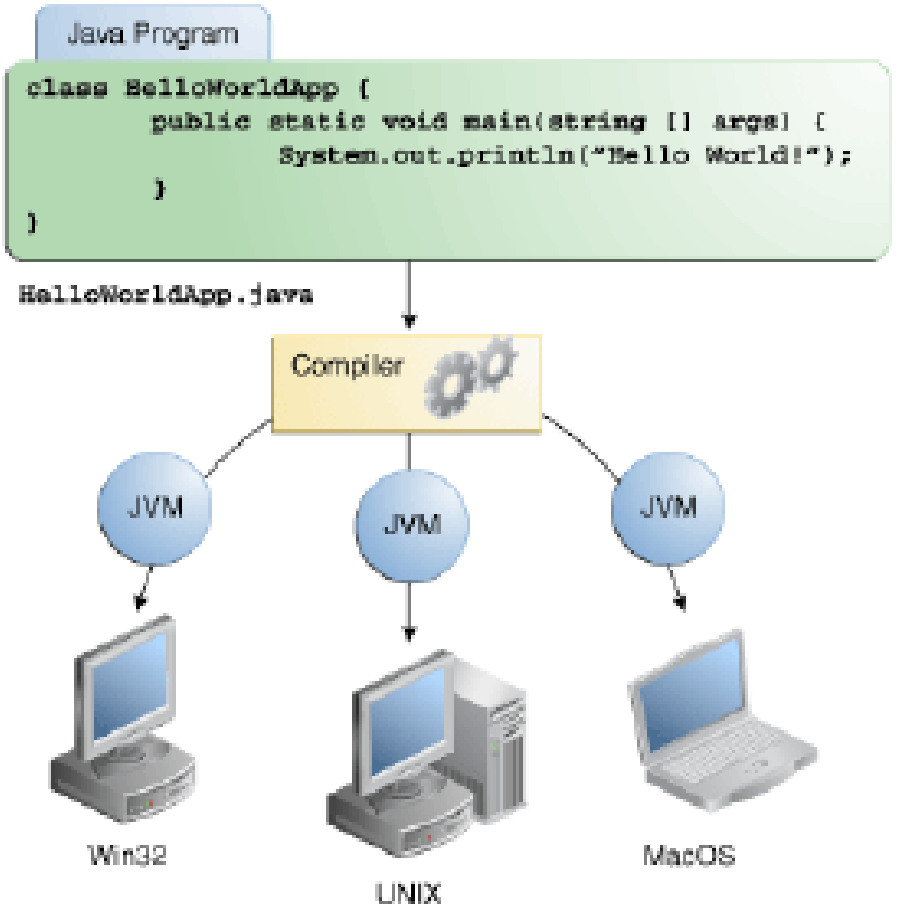
Nakon preuzimanja prvo instalirati JDK, pa onda otpakovati Eclipse.

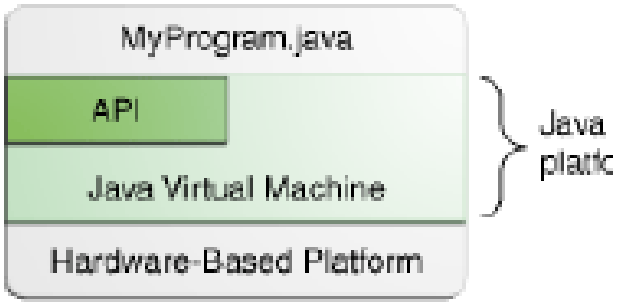
Datoteke .java i .class

Fajl sa ekstenzijom **.java** je kod sa instrukcijama, takozvani **izvorni kod**, dok je fajl sa ekstenzijom **.class bajt kod**. Mi pišemo izvorni kod, koji kada kompiliramo dobijemo bajt kod, koji zatim Java virtuelna mašina interpretira i pokreće



Java virtuelna mašina



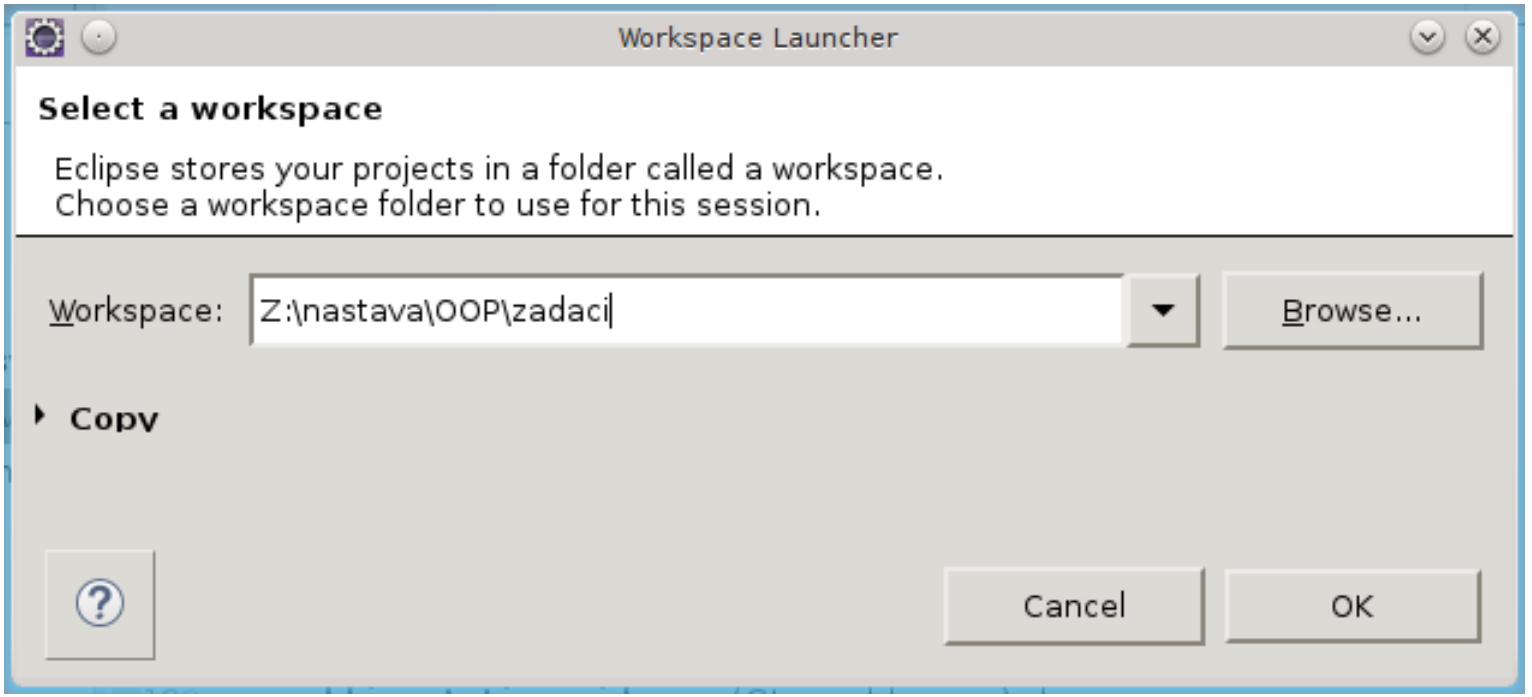


Pokretanje razvojnog okruženja

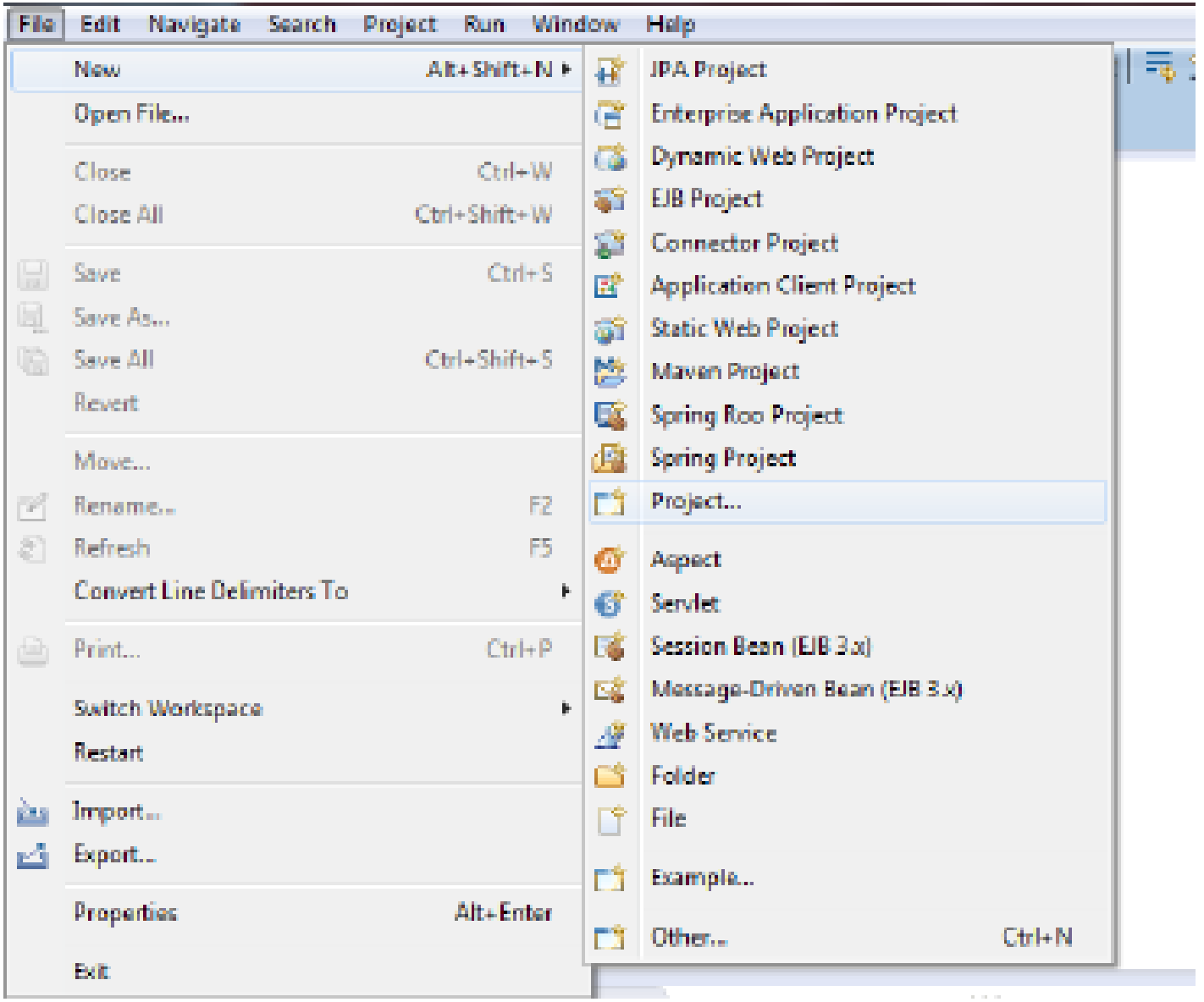
Eclipse se pokreće klikom na program Eclipse

Potom je potrebno podesiti putanju do direktorijuma gde će se čuvati projekti.
File→ Switch Workspace →Other

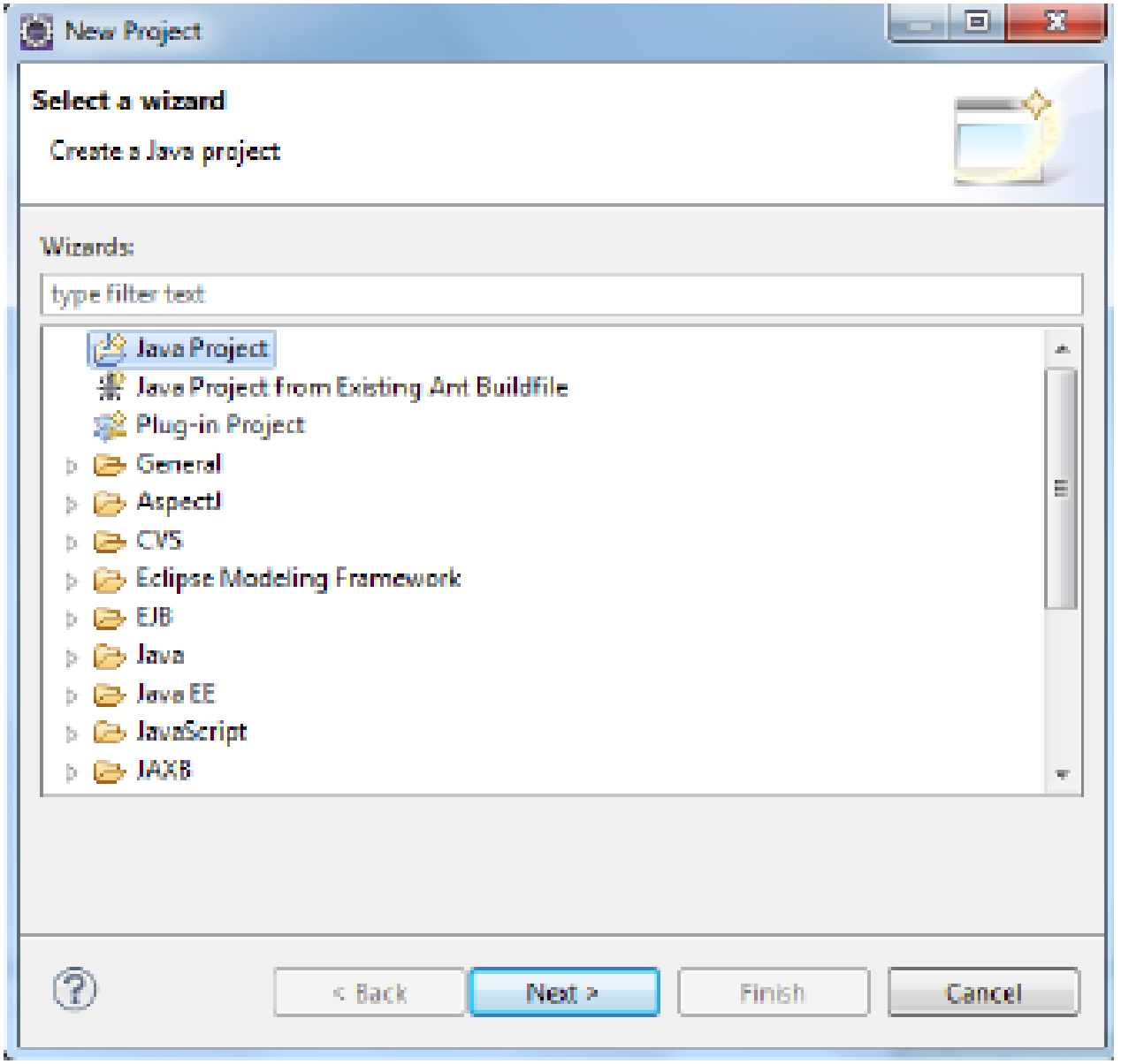
Napraviti neki svoj direktorijum gde će vam biti svi projekti



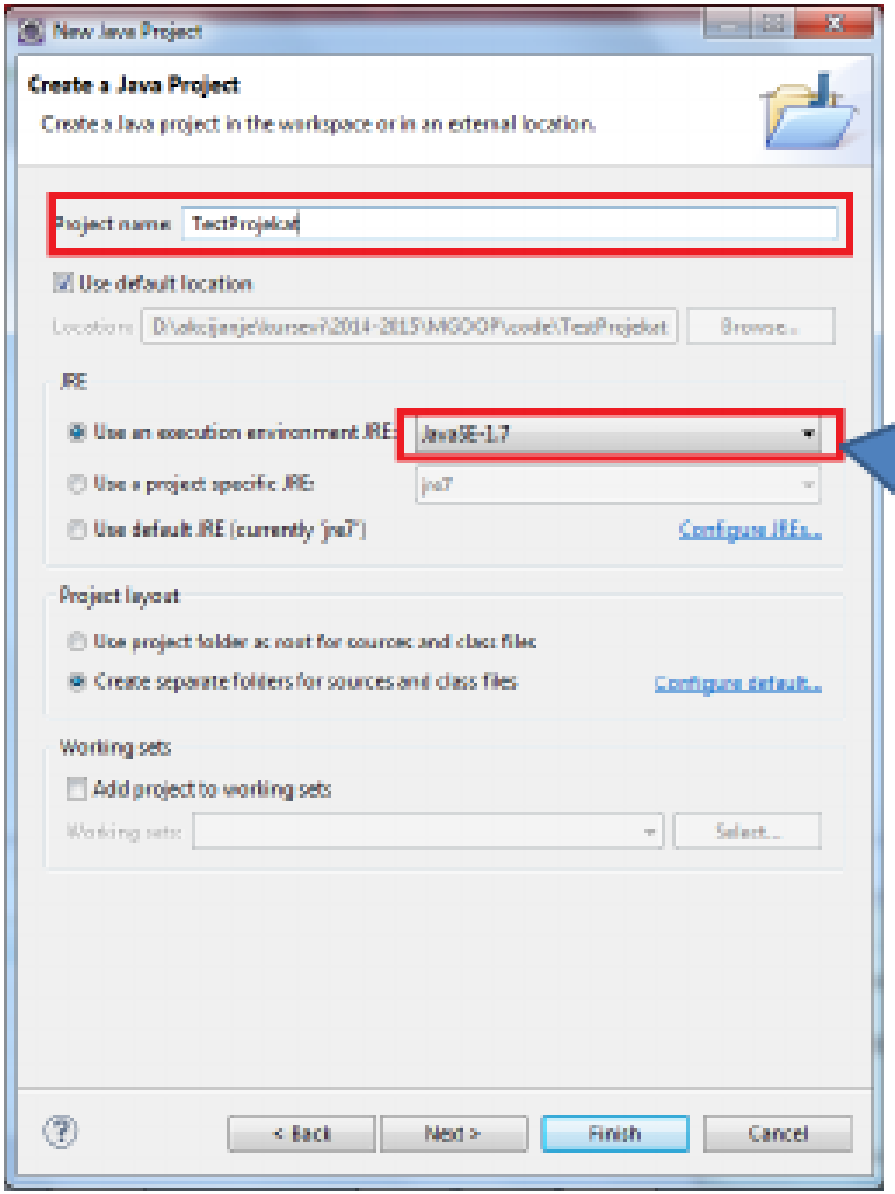
Kreiranje Java projekta



Kreiranje Java projekta



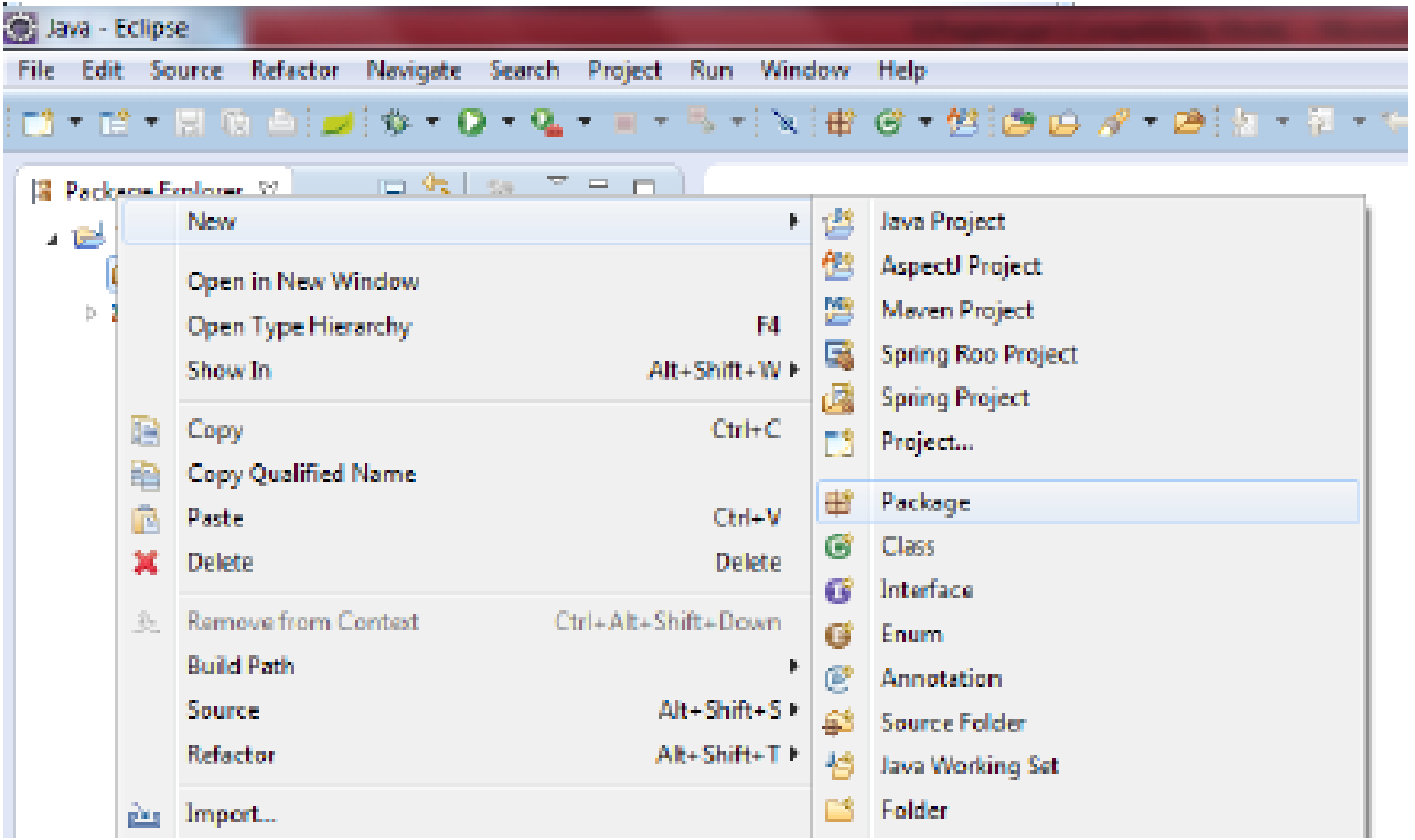
Kreiranje Java projekta



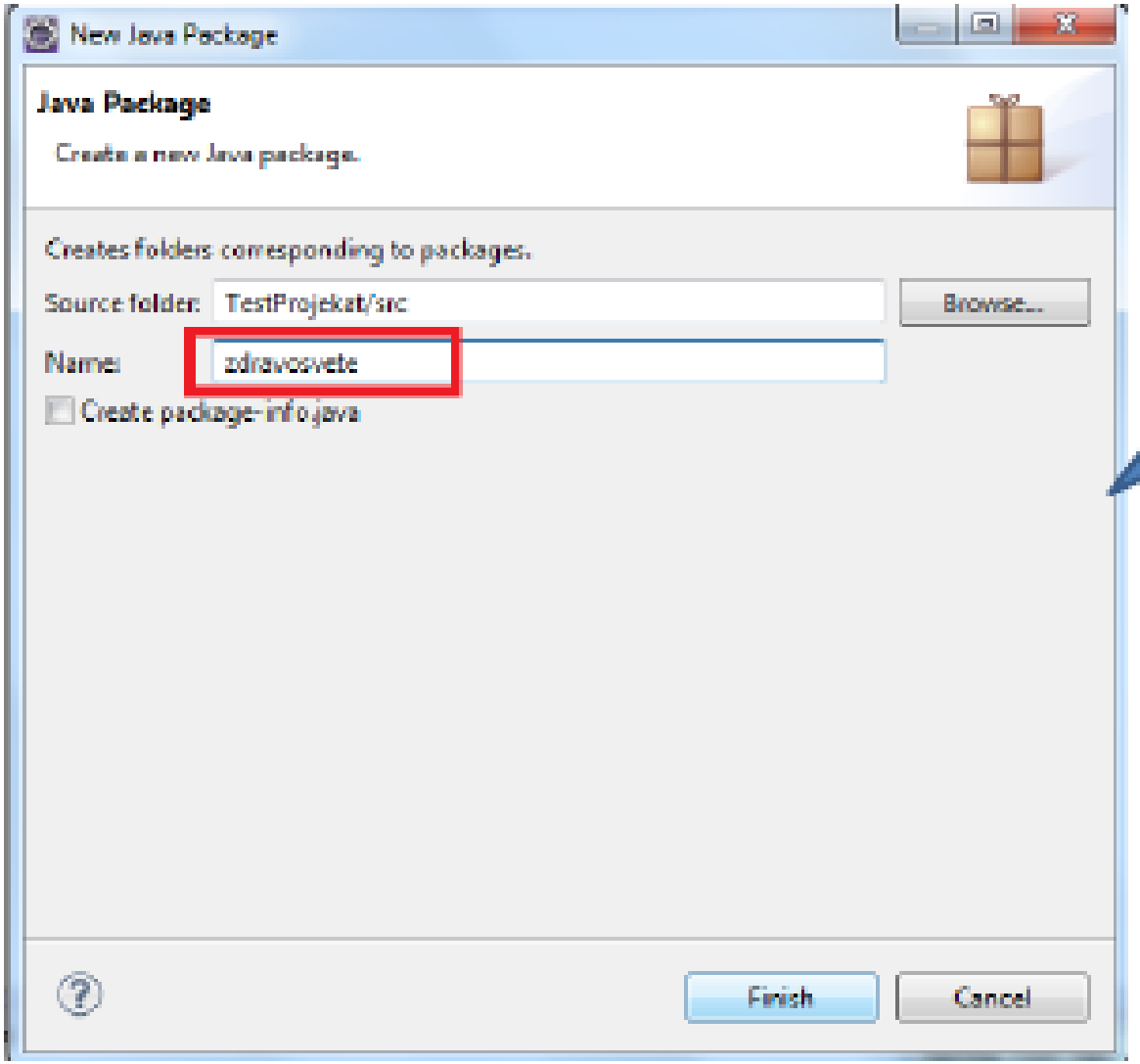
Može se desiti da JDK (JRE) nije povezan sa Eclipse-om. Takav scenario razrešiti sa nastavnikom.

Pravljenje novog paketa

Desni klik na src → New → Package → . . .



Pravljenje novog paketa

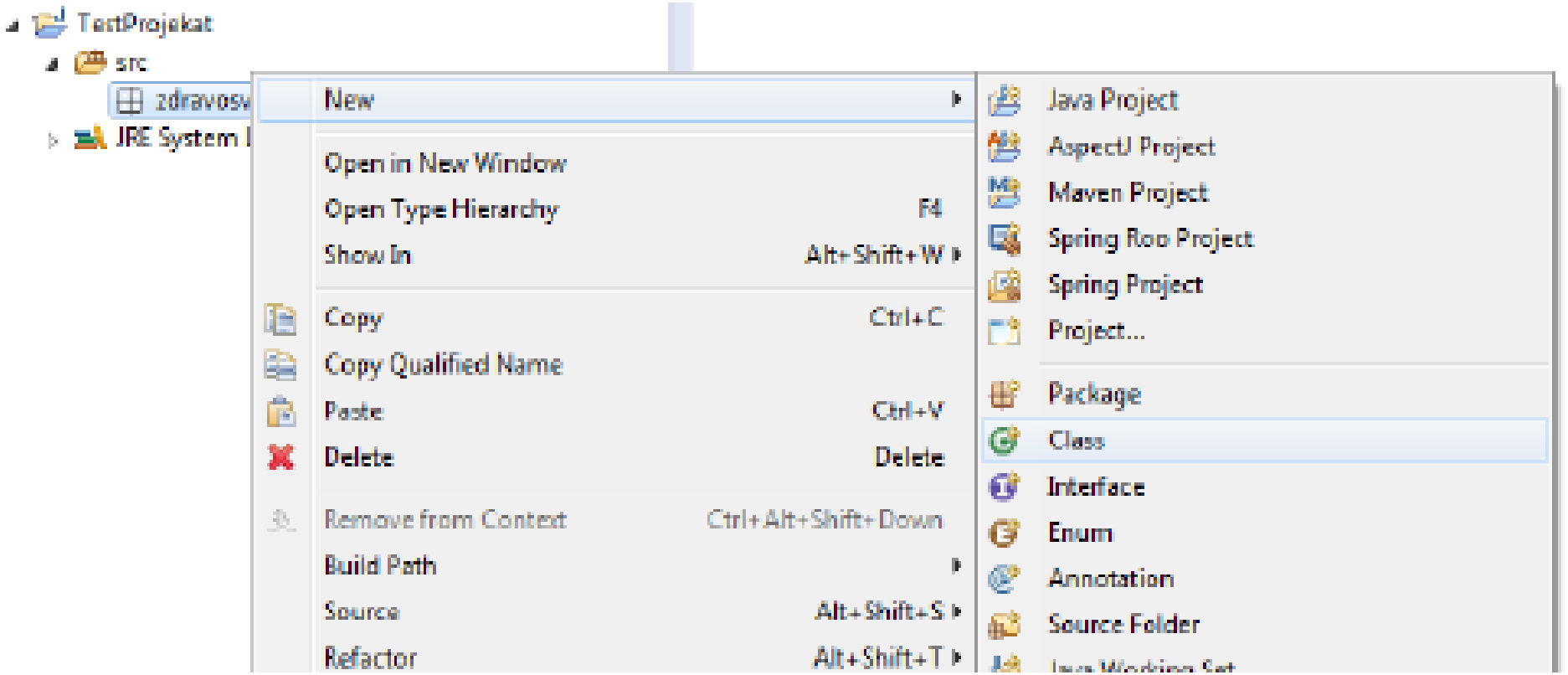


Paketi su Java direktorijumi u kojima držimo srodan kod.

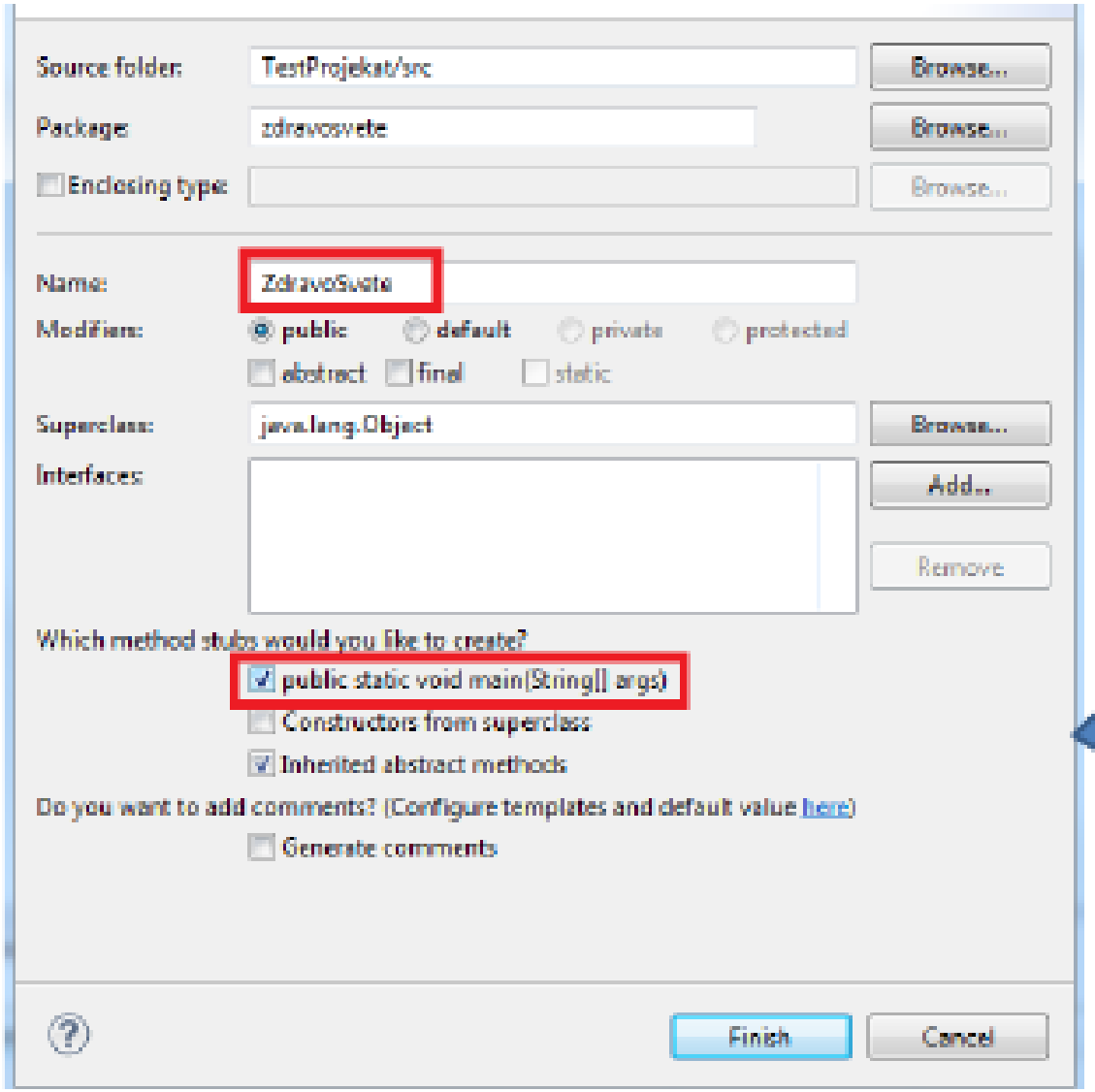
Pravljenje Java datoteke

Pravimo novu klasu u kojoj će se nalaziti kod

Desni klik na paket → New → Class . . .



Pravljenje Java datoteke - klase



- Klasa je osnovna jedinica u kojoj se nalazi java kod (datoteka).
- Više srodnih klasa grupišemo u pakete (direktorijume)

- Unesite ime klase.
- Čekirajte opciju **main** kako bi klasa imala mogućnost pokretanja.
- Ne mora svaka klasa da ima ovu mogućnost!

Segmenti koda u klasi

ZdravoSvete.java

```java
// Ovo je linijski komentar

/* Ovo je više-
   linijski komentar
*/

/* Ovo je ime paketa i ono mora da se poklapa sa lokacijom klase.
 * Npr. ako bi neko ovu klasu prebacio u drugi paket, kompajler bi javio grešku
 */
package zdravosvete;

/* Ime klase mora da odgovara nazivu .java datoteke */
public class ZdravoSvete {

    /* main metod (funkcija) kao u C-u.
     * Klase koje nemaju main metod nisu izvršne.
     */
    public static void main(String[] args) {
        // Trenutno klasa ne radi ništa jer je main metod prazan
    }

}
```

Zdravo svete primer

ZdravoSvete.java | Primer1.java

```java
/* Ovo je više-
   linijski komentar
*/

/* Ovo je ime paketa i ono mora da se poklapa sa lokacijom klase.
 * Npr. ako bi neko ovu klasu prebacio u drugi paket, kompajler bi javio grešku
 */
package zdravosvete;

/* Ime klase mora da odgovara nazivu .java datoteke */
public class ZdravoSvete {

    /* main metod (funkcija) kao u C-u.
     * Klase koje nemaju main metod nisu izvršne.
     */
    public static void main(String[] args) {
        //Ispisuje na konzoli uneti tekst i stavlja novi red na kraju
        System.out.println("Zdravo svete!");
        //System.out.print (bez ln) radi to isto samo ne stavlja novi red na kraju
    }

}
```

Komentari

```
/**
* Klasa ZdravoSvete implementira aplikaciju koja ispisuje
* string "Zdravo svete" na standardni izlaz.
*/

class ZdravoSvete {
   public static void main(String[] args) {
      System.out.println("Zdravo svete!");          // Ispisuje string.
   }
}
```

/* neki tekst */

- kompajler ignorise sve izmedju /* i */

/** dokumentacija */

- kompajler takodje ignorise I ovakve delove teksta,
  ali njih cita javadoc alat pri generisanju dokumentacije

// neki tekst

- kompajler ignorise sve od // do kraja linije

Definicija klase

```
class ZdravoSvete {
   public static void main(String[] args) {
      System.out.println("Zdravo svete!"); // Ispisuje string.
   }
}
```

```
class ImeKlase {
. . .
}
```

Počinje ključnom reči class, za kojom sledi naziv klase. Kod koji predstavlja telo klase se navodi unutar vitičastih zagrada { i }.

Metoda main

```
class ZdravoSvete {
   public static void main(String[] args) {
      System.out.println("Zdravo svete!"); // Ispisuje string.
   }
}
```

Svaka aplikacija mora da sadrži main metodu, čiji je prototip

public static void main(String[] args)

Modifikator *public* označava da je metoda javna i da se može pozvati i izvan klase, dok modifikator *static* označava da se metoda može pozvati i bez postojanja objekta nad kojim se poziva (o tome će biti više reči kasnije). Za njima sledi tip povratne vrednosti, koji je *void*, čime se označava da metod nema povratnu vrednost.

Metoda prihvata jedan argument, a to je niz elemenata *String* tipa.

Ispis na standardni izlaz

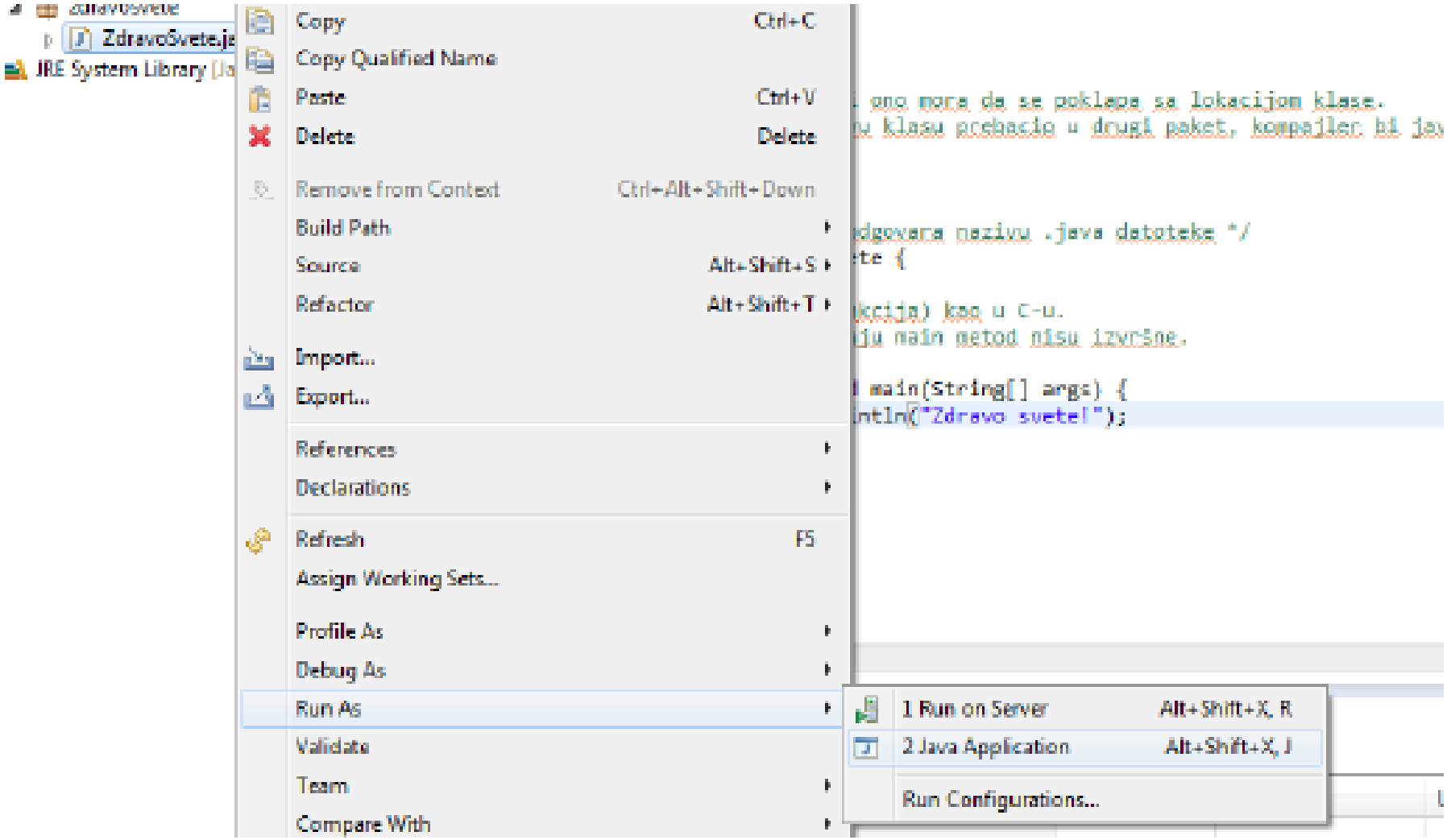
```
class ZdravoSvete {
   public static void main(String[] args) {
      System.out.println("Zdravo svete!"); // Ispisuje string.
   }
}
```

System je ugrađena klasa iz *java.lang* paketa. Ona sadrži člana *out,* koji je tipa klase *PrintStream*, a koja služi za ispis podataka. Neke od metoda klase *PrintStream* su *print, println, printf* i druge.

Metoda print *println* ispisuje i znak za novi red nakon ispisa stringa.

Pokretanje klase

Desni klik na klasu → Run as → Java Application



Prikaz rezultata u konzoli

```
 */
package zdravosvete;

/* Ime klase mora da odgovara nazivu .java datoteke */
public class ZdravoSvete {

    /* main metod (funkcija) kao u C-u.
     * Klase koje nemaju main metod nisu izvršne.
     */
    public static void main(String[] args) {
        System.out.println("Zdravo svete!");
    }

}
```

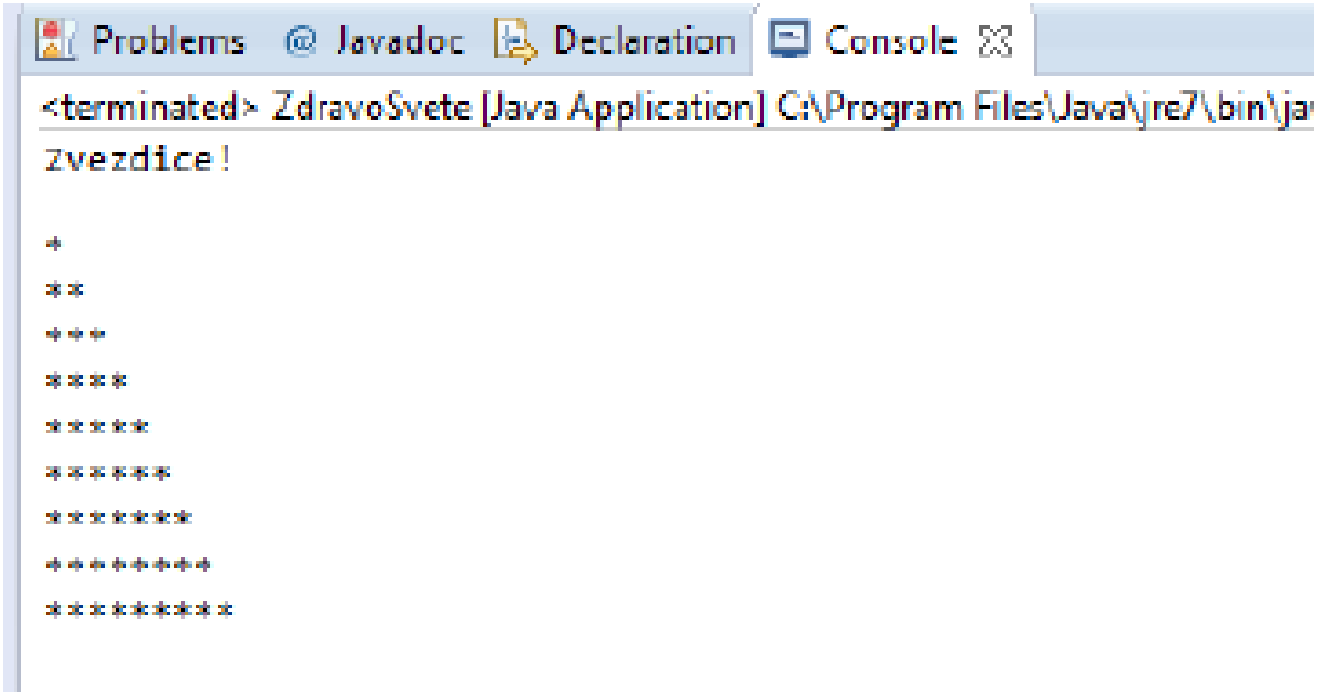
Problems @ Javadoc Declaration Console

<terminated> ZdravoSvete [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (12.09.2014. 10.26.00)

```
Zdravo svete!
```

Zadaci

1. Napraviti projekat pod nazivom Zvezdice
2. U okviru njega napraviti paket pod nazivom lako
3. U okviru paketa napraviti klasu Primer1
4. Korišćenje petlji i promenljivih je isto kao u C-u



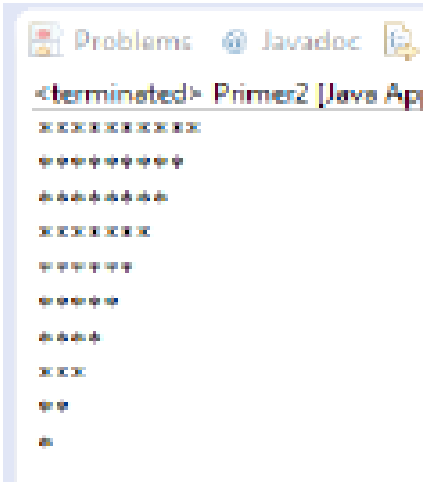
Rešenje

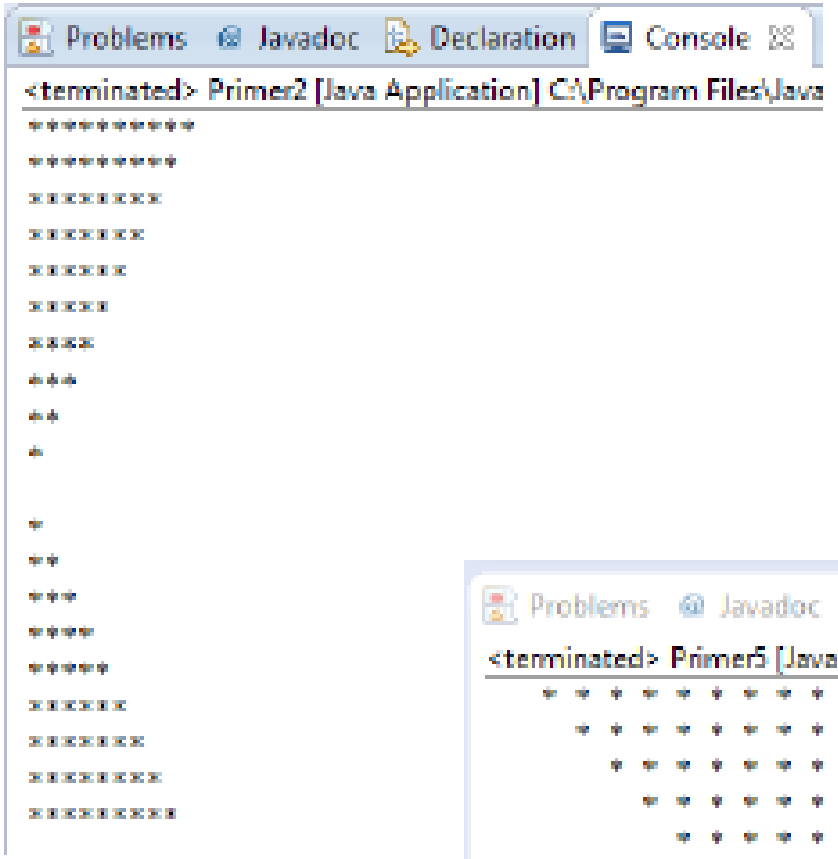
```
class Primer1 {
   public static void main(String[] args) {
      int i;
      int j;

      for(i=1; i<10; i++){
        for(j=1; j<=i; j++)
           System.out.print("*");
        System.out.println();
      }
   }
}
```

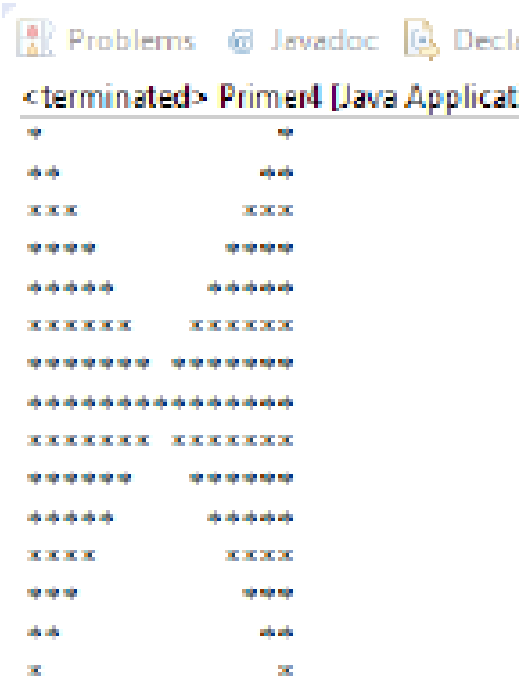
Zadaci za kod kuće.

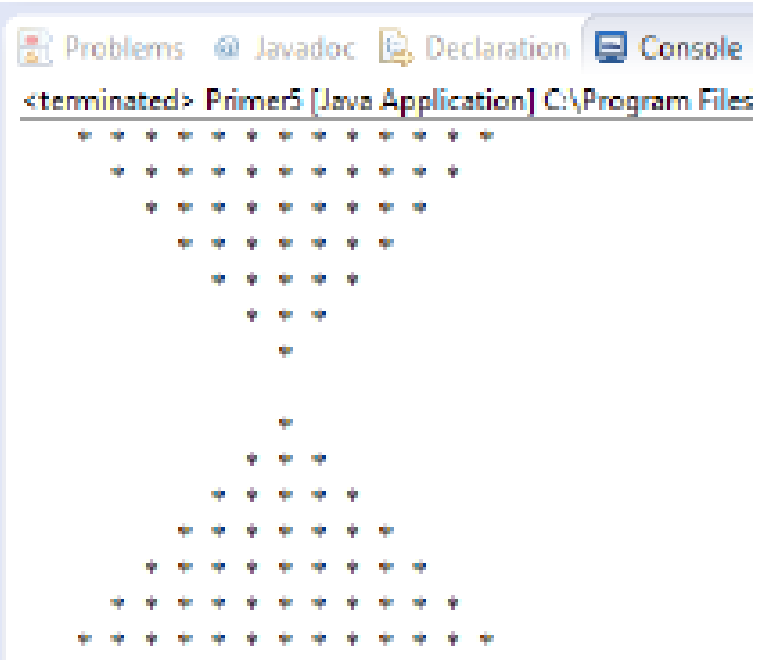
Napomena:
Sve uraditi korišćenjem dve ugnježdene petlje.

Ne praviti novi projekat, već sve staviti u projekat Zvezdice paket lako.









Pravljenje i pokretanje aplikacije u konzoli (bez razvojnog okruženja)

Programi se mogu pisati i čitati u razvojnom okruženju, na primer *Eclipse*. Razvojno okruženje veoma olakšava proces pisanja i rada sa programima. Takođe, programi se mogu napraviti, i izmeniti, i u bilo kom tekstualnom editoru, na primer *Notepad*. Zatim, kompilirati i pokretati iz konzole (zove se i komandna linija).

Prvi način smo videli ranije, drugi način će biti predstavljen sada. Demonstriraćemo proces za operativni sistem *Windows*.

Potrebno je da imate

1. Instaliran JDK 8

2. Neki tekst editor.

Mi ćemo ovde koristiti tekst editor *Notepad*. Mada se ove instrukcije mogu primeniti i na bilo koji drugi

Pravljenje aplikacije

Napravićemo aplikaciju koja ispisuje "Zdravo svete" na standardni izlaz

Potrebno je da

1. **Kreiramo izvorni kod i sačuvamo u .java datoteci**

Izvorni kod sadrži instrukcije pisane, u našim primerima, u programskom jeziku Java

2. **Kompiliramo izvorni kod u bajt kod, odnosno .class datoteku**

Java kompilator *javac* prevodi te instrukcije u instrukcije koje razume Java virtuelna mašina

3. **Pokrenemo program**

Alat *java* koristi virtuelnu mašinu da pokrene našu aplikaciju

Kreiranje datoteke sa izvornim kodom

Kreirajte novu datoteku

Start → Programs → Accessories → Notepad

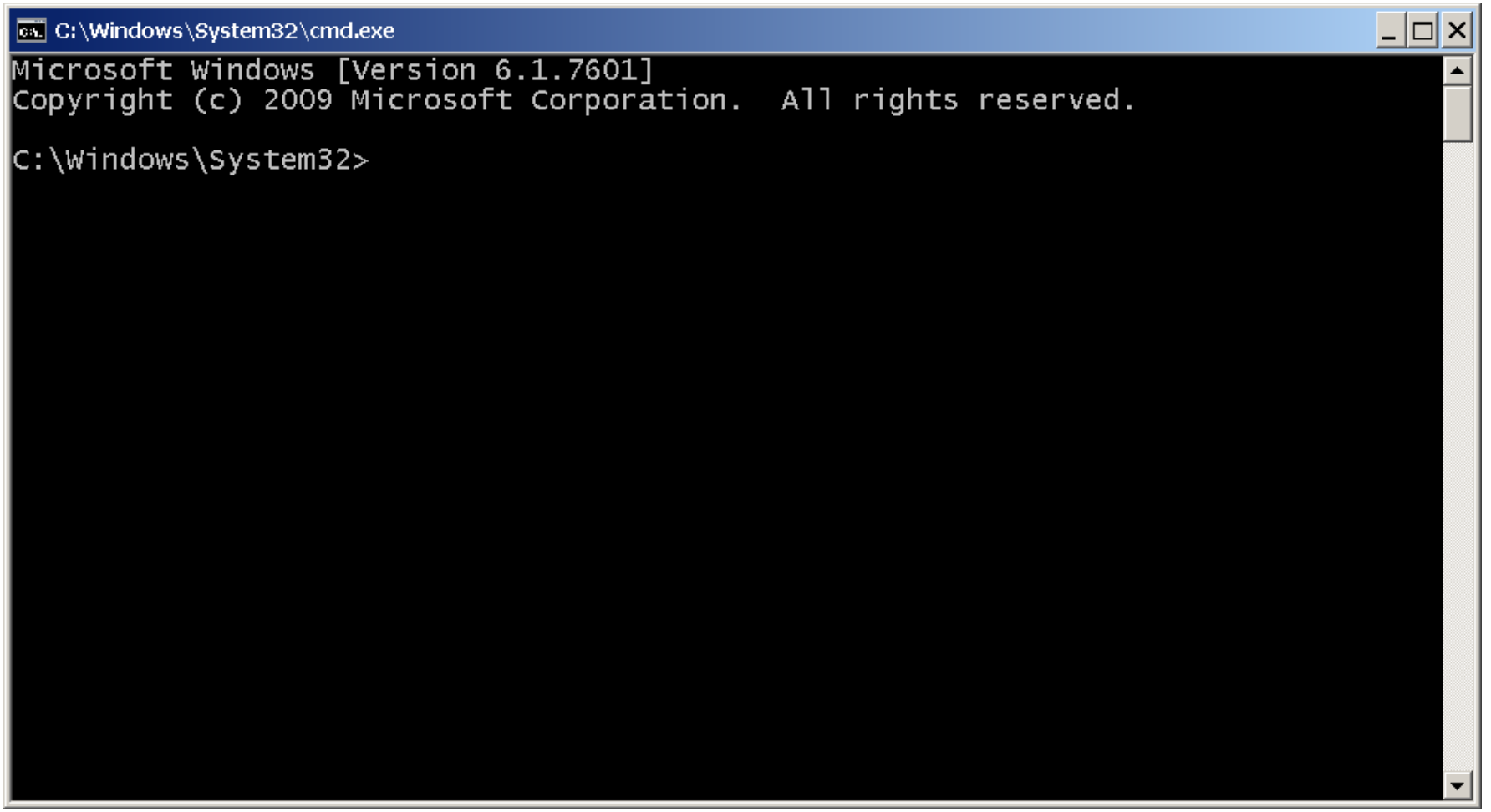
U tekst editoru otkucajte program koji ispisuje "Zdravo svete!", sa prethodnih strana (bez navođenja paketa kao u prvom primeru). Vodite računa o malim i velikim slovima, ZdravoSvete nije isto što i zdravosvete.

Sačuvajte datoteku File → Save As

1. U **Save in** prozoru izaberite direktorijum gde želite da sačuvate fajl. Na primer Z:\nastava\OOP\primeri
2. U polju **File name** unesite "ZdravoSvete.java"
3. **Save as type** izaberite Text documents (*.txt)
4. Za **Encoding** izaberite ANSI
5. Kliknite na **Save**

Kompiliranje izvorne datoteke u .class datoteku

Otvorite komandni prozor, Start → Run, ukucajte cmd



Kompiliranje izvorne datoteke u .class datoteku

Komand prompt prikazuje trenutni direktorijum. Promenite trenutni direktorijum na direktorijum u kome se nalazi datoteka sa izvornim kodom (Z:\nastava\OOP\primeri)

C:\Windows\System64> Z:

Z:\> cd nastava\OOP\primeri

Sada bi prompt trebalo da se promeni na Z:\nastava\OOP\primeri>

Ukoliko ukucate komandu dir, trebalo bi da se izlista i vaša datoteka.

Kompiliranje izvorne datoteke u .class datoteku

Kompilirajte program

Z:\nastava\OOP\primeri> javac ZdravoSvete.java

Ukoliko ukucate komandu dir, trebalo bi da se izlista i ZdravoSvete.class

Pokrenite program

Z:\nastava\OOP\primeri> java ZdravoSvete
Zdravo svete!

Česte greške

- **javac is not recognized as an internal or external command, operable program or batch file**

  Rešenje: pokrenite javac sa punom putanjom

  Z:\nastava\OOP\primeri>C:\jdk1.8.0\bin\javac ZdravoSvete.java
- **Could not find or load main class ZdravoSvete**

  Ne može da pronadje bajt kod. Proverite da li je napravljena .class datoteka
- **Could not find or load main class ZdravoSvete.class**

  Navedite samo ime klase, bez nastavka .class
- **Exception in thread "main" java.lang.NoSuchMethodError: main**

  java zahteva da klasa ima main metod

Zadaci

1. Pronađite java datoteku koju ste napravili uz pomoć *Eclipse* i otvorite je u *Notepad*-u. Izmenite poruku na "Pozdrav svima" i sačuvajte datoteku.

2. Pozicionirajte prompt tako da se nalazi u direktorijumu u kome je paket u kome se nalazi .java datoteka. To je direktorijum oblika
Z:\workspace\projekat\src\paket

Z:\> cd nastava\OOP\TestProjekat\src

3. Kompilirajte i pokrenite program

Z:\nastava\OOP\TestProjekat\src> javac zdravosvete\ZdravoSvete.java

Z:\nastava\OOP\TestProjekat\src> java zdravosvete.ZdravoSvete

Da li ispisuje očekivanu poruku?

4. Pokrenite *Eclipse* i otvorite taj isti program. Da li se izmena vidi i tu?

Dakle, to je jedna ista datoteka, kojoj možemo pristupati ili iz nekog tekstualnog editora ili iz razvojnog okruženja. Slično, možemo je kompilirati i pokretati ili iz komandne linije ili iz razvojnog okruženja.

Osnovni elementi jezika Java

Struktura Java programa

Java program se uvek sastoji od jedne ili više klasa.

Uobičajeno je da se svaka klasa stavlja u posebnu datoteku koja mora da ima isto ime kao i klasa sadržana u njemu.

Java izvorna datoteka mora da ima ekstenziju .java.

Svaka Java aplikacija sadrži klasu koja definiše metodu main. Ime ove klase je ime koje dajemo kao argument Java interpreteru.

Kolekcija već postojećih klasa se naziva biblioteka.

Klase su grupisane u odgovarajuće skupove koji se nazivaju **paketi**.

Korisnik može sam da pravi pakete koji će sadržati odgovarajuće klase.

Paketi

Java raspolaže velikim brojem **standardnih paketa**.

Najčešće se koriste

- **java.lang** - osnovne karakteristike jezika, rad sa stringovima i nizovima. Klase ovog paketa su automatski uključene u program

- **java.io** - klase za ulazno/izlazne operacije

- **java.util** - klase Vector, Stack, Scanner . . .

Da bismo koristili klase iz nekog paketa, koristimo **import** deklaraciju sa imenom paketa ili klase:

import java.util.Vector;

import java.util.*; - uključuju se sve klase iz paketa

Jedino je java.lang uvek uključen i nije ga potrebno uključivati

Može se koristiti klasa iz nekog paketa i bez import deklaracije, ali se mora navesti puno ime klase

java.util.Vector a;

Tipovi podataka

Celobrojni tipovi

int	4 bajta	~ -2×10^9 do ~ 2×10^9
short	2 bajta	-32,768 do 32,767
long	8 bajtova	~ -9×10^{18} do ~ 9×10^{18}
byte	1 bajt	-128 do 127

Tipovi podataka u pokretnom zarezu

float	4 bajta	~ $+-3.40282347 \times 10^{38}$
double	8 bajtova	~ $+-1.7969313486231570 \times 10^{308}$

Karakterski tip

char	2 bajta	. . .

Logički tip

boolean	zavisi od VM	false, true

Tipovi podataka

Literali su eksplicitne vrednosti podataka koje se javljaju u programu.

Na primer, 15 je celobrojni literal tipa int

Literali tipa long imaju sufiks **L** ili **l**. Za heksadekadnu osnosu koristi se **0x** ili **0X**, dok se za binarnu osnovu koristi **0b** ili **0B**.

Literali u pokretnom zarezu su podrazumevano tipa double. Kada hoćemo da naglasimo da je vrednost float dodamo sufiks **f** ili **F**.

Promenljive se deklarišu na sledeći način

```
double plata;
int brojRadnihDana;
long populacijaZemlje
boolean kraj;
```

Inicijalizacija promenljivih

```
int dani, meseci;
dani = 6;
int godine = 17;
```

Konstante se definišu ključnom reči final

```
final double pi = 3.14;
```

Operatori

Aritmetički operatori su +, -, *, /

Ukoliko su oba argumenta celi brojevi, / označava celobrojno deljenje, u suprotnom je razlomljeno deljenje

Ostatak se računa pomoću operatora %

Neke od matematičkih funkcija i konstanti su definisane u klasi Math: sqrt, pow, round, rand, sin, cos, tan, atan, exp, log, log10, PI, E.

Koriste se sa prefiksom Math

```
double a = Math.sqrt(Math.pow(b, 2) + Math.pow(c,2));
```

Ili bez prefiksa ako se u zaglavlju navede

```
import static java.lang.Math.*;
. . .
double a = sqrt(pow(b, 2) + pow(c,2));
```

Operatori

Konverzija izmedju tipova se izvodi na sledeći način

Ako je jedan operand double, drugi se konvertuju u double.
Inace, ako je jedan operand float, drugi se konvertuju u float.
Inace, ako je jedan operand long, drugi se konvertuju u long
Inace, oba se konvertuju se u int.

Ne postoji automatska konverzija boolean u int i obratno već se mora porediti eksplicitno if(a!=0) . . .

Konverzija int u double se može vršiti i ovako: ako su a i b promenljive celobojnog tipa, izraz a/b je tipa int, ali izraz a/(b+0.0) ili (a*1.0)/b je tipa double.

Kastovanje se vrši navođenjem tipa u zagradi ispred promenljive

```
int a = 3, b = 5;

double c = 1.5 + b/a;       // c je 2.5

c = 1.5 + (double)b/a;      // c je 3.17

double x = 9.997;
int nx = (int) x;           // nx je 9
int nx = (int) Math.round(x);      // nx je 10
```

Operatori

Kombinovanje dodele i operatora

x += 4; isto što i x = x + 4;

Ukoliko je dodela različitog tipa, konvertuje se u tip sa leve strane. Ako je x int, izraz x += 3.5; isto što i x += (int)(x + 3.5)

Inkrement i dekrement

```
int n = 12;
n++;      // n ima vrednost 13
--n;      // n je bilo 13, sada ima vrednost 12
```

```
int n = 7, m = 7;
int a = 2 * ++m;      // a ima vrednost 16, m ima vrednost 8
int b = 2 * n++;      // b ima vrednost 14, n ima vrednost 8
```

Operatori

Relacioni operatori - poređenje na jednakost ==, na nejednakost !=, relacije <, >, <=, >=.

Logički operatori - konjunkcija (i) &&, disjunkcija (ili) ||, negacija !. Izvodi se lenjo izračunavanje, to znaci da se drugi argument ne izračunava ukoliko je prvim već određena vrednost izraza.

```
x != 0  && 1/x > x + y
```

Ternarni operator ?: - Izraz e1 ? e2 : e3, vraca e2 ukoliko je e1 tačno, u suprotnom vraća e3.

Primer1, računanje minimuma

```
double min = x < y ? x : y
```

Primer2, računanje int vrednosti na osnovu boolean vrednosti

```
boolean b = false;
int c = b ? 1 : 0;
```

Operatori

Bitovski operatori - konjunkcija &, disjunkcija |, negacija ~, ekskluzivna disjunkcija (xor) ^. Može se koristiti za pristup bitovima.

```
int cetvrtiBitOdKraja = (n & 0b1000) / 0b1000;
```

Oba argumenta se izračunavaju, ne primenjuje se lenjo izračunavanje.

Operatori siftovanja u levo i desno <<, >>.

Logičko šiftovanje u desno >>> (na upražnjena mesta dovode se nule).

```
int poslPetBitova = n & 0b11111;
int naPocetak = poslPetBitova <<27;          // 32 - 5
int vratiNaKraj = naPocetak >>>27;
int cetvrtiBitOdKraja = (n & (1 << 3)) >> 3;
```

Hijerarhija operatora

Ukoliko se ne koriste zagrade, operatori se izvršavaju po hijerarhiji prioriteta (tabela). Ukoliko su istog prioriteta, izvršavaju se po asocijativnosti (navedeno u tabeli)

[] . () (pozivanje metode)	S leva na desno
! ~ ++ - +(unarni) -(unarni) () (kastovanje) new	S desna na levo
* / %	S leva na desno
+ -	S leva na desno
<< >> >>>	S leva na desno
< <= > >= instanceof	S leva na desno
== !=	S leva na desno
&	S leva na desno
^	S leva na desno
|	S leva na desno
&&	S leva na desno
||	S leva na desno
?:	S desna na levo
= += -= *= /+ %= &= |= ^= <<= >>= >>>=	S desna na levo

Primer

Program koji za data dva cela broja računa i ispisuje njihove zbir, razliku, proizvod, celobrojni količnik, ostatak pri deljenju prvog broja drugim brojem, realno deljenje.

```
public class CeliBrojevi{
 public static void main(String[] args){
   int a = 17, b = 9;
   int z = a+b, r = a-b;
   System.out.printf("Zbir = %d, razlika = %d\n", z, r);
   System.out.printf("a %% b = %d,  a/b = %d, a * b = %d\n", a%b, a/b, a*b);
   System.out.printf("%d / %d = %f\n", a, b, (a+0.0)/b);
 }
}
```

Zadaci

1. a) Napisati program koji za data dva realna broja dvostruke tačnosti računa i ispisuje njihove: zbir, razliku proizvod i količnik.

b) Kastuje brojeve u brojeve jednostruke tačnosti i ispisati isto kao i prvoj tački.

2. Napisati klasu **Min3** koja za data tri realna broja računa i ispisuje na izlaz najmanji od ova tri broja. Koristiti operator ?:

3. Napisati program koji za dati trocifreni broj ispisuje na izlaz dekadne cifre tog broja.

4. Napisati program koji za dati poluprečnik kruga ispisuje na izlaz obim kruga i njegov poluprečnik.

Čitanje sa ulaza

Scanner je klasa koja implementira jednostavni skener koji parsira primitivne tipove i stringove.

Ulaz se deli na tokene, a kao delimiter se podrazumevano koristi prazan karakter (space).

Rezultujući tokeni se mogu dalje konvertovati u vrednosti različitih tipova koristeći metode

next() - ucitava sledeći token

nextDouble() - ucitava sledeći token kao double

nextInt() - ucitava sledeći token kao int

nextLine() - ucitava sledeću liniju

Za svaki od metoda postoje i odgovarajuće metode kao što su

hasNextInt() - vraća true ako se sledećim pozivom nextInt() moze učitati int vrednost.

Primer

```
import java.util.Scanner;
public class Primer {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.printf("Kako se zoves? ");
    String ime = in.nextLine();
    System.out.printf("Koliko imas godina? ");
    int godine = in.nextInt();
    System.out.printf("Koliko je 5/2?");
    double rez = in.nextDouble();
    System.out.printf("%s\n", rez==5/2 ? "Tacno" : "Netacno");
    System.out.printf("Zdravo, %s. ",  ime);
    System.out.printf("Sledece godine ces imati %d godina.", godine+1);
    in.close();    // kada se zavrsi sa koriscenjem, skener treba zatvoriti

}
```

Pisanje na izlaz

Metod System.out.print ispisuje na izlaz string koji je dat kao argument.

```
System.out.print("Unet je broj " + rez + "\n");
```

Metod System.out.println ispisuje na izlaz string koji je dat kao argument, a zatim ispisuje i znak za novi red.

```
System.out.println("Unet je broj " + rez);
```

Metod System.out.printf se koristi za formatiranje izlaza koji funkcioniše isto kao i funkcija *printf* u C-u.

```
System.out.printf("Unet je broj %4.2f.", rez);
System.out.printf("Zdravo, %s. Imas %d godina.", ime, godine);
```

Neki od specifikatora konverzije su d (ceo broj), f (broj u pokretnom zarezu), s (string), b(boolean vrednost), %(procenat simbol).

Kontrola toka

Blok

```
int n;
{
  int k;   // k postoji samo u ovom bloku
}
```

Selekcija

```
if (uslov) naredba
```

Ukoliko ima više naredbi koristi se blok {naredba1; naredba2; . . .}

```
if (uslov) {lista naredbi} else {lista naredbi}
```

Else se grupiše sa najblizim if. Može se koristiti i else if

```
if(uslov) naredba else if(uslov) naredba else naredba
```

Kontrola toka

Petlje

while (uslov) naredba ili
while (uslov) {lista naredbi}

do naredba while(uslov) ili
do {lista naredbi} while (uslov)

for(inicijalizacija; uslov; promena) naredba
for(inicijalizacija; uslov; promena) {lista naredbi}

```
int s = 0, f = 1;
for (int i=1; i<5; i++)
{
  s += i; f *= i;
}
```

s = ? f = ?

```
int s=0; f=1; i=1;
while(i<5)
{
  s+=i; f*=i;
  i++;
}
```

s = ? f = ?

```
int s=0; f=1; i=1;
do
{
  s+=i; f*=i;
}while(++i<5)
```

s = ? f = ?

Naredba višestrukog izbora

switch (izbor) {case 1: . . . break; case 2: . . . break; default: . . . break;}

Nakon oznake case mogu se naći konstante tipa char, byte, short, int, enumerisane konstante i stringovski literali.

```
int ulaz = . . .;
switch(ulaz)
{
  case 1: . . . break;
  case 2: . . . break;
  case 3: . . . break;
  . . .
  default: . . .
}
```

```
String ulaz = . . .;
switch(ulaz.toLowerCase())
{
  case "da": . . .  break;
  . . .
}
```

Kontrola toka

Break naredba se koristiti da se izađe iz petlje. Break naredbom se izlazi samo iz najbliže petlje.

```
while(godine <= 100)
{
  suma += godine * osnova;
  if(suma > cilj) break;
  godine++;
}
```

```
Scanner in = new Scanner(System.in)
while(true)
{
  System.out.print("Unesi broj, -1 za kraj:  ");
  n = in.nextInt();
  if(n == -1) break;
}
```

Kontrola toka

Continue naredba takodje menja regularni tok izvršavanja. Continue prenosi izvršavanje na početak najbliže petlje.

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

while(suma<cilj)
{
  System.out.print("Unesi broj: ");
  n = in.nextInt();
  if(n<0) continue;
  suma += n;
}
```

Ukoliko se continue koristi u for petlji, prelazi se na deo ažuriranja.

```
for(int i=0; i<10; i++)
{
  System.out.print("Unesi broj: ");
  n = in.nextInt();
  if(n<0) continue;
  sum += n;
}
```

Zadaci

1. Napisati program koji učitava sa ulaza tri cela broja i ispisuje na izlaz brojeve u rastućem poretku

2. Napisati program koji učitava sa ulaza dva cela broja n i m

- Ispisuje 0 1 2 . . . n

- Ispisuje sve cele brojeve izmedju n i m, bez n i m, u opadajućem poretku

- Ispisuje sve brojeve izmedju n i m sa korakom 0.5, uključujući n i m, u opadajućem poretku

- Ispisuje sve brojeve deljive sa 3 koji su između n i m, uključujući n i m

- Računa i ispisuje s = 1 + 2 + . . . + n

- Računa i ispisuje n! = 1 * 2 * . . . * n

- Računa i ispisuje s = 1! + 2! + . . . + n!

3. Napisati program koji učitava sa ulaza dan, mesec i godinu, tako da bira odgovarajući tip za smestanje podataka, a zatim računa i ispisuje na izlaz datum sledećeg dana.

Stringovi

Java nema ugrađeni tip string, umesto toga standardna bibilioteka sadrži klasu *String* koja čuva string kao niz Unicode karaktera.

```
String s = "";
String pozdrav = "Zdravo";
```

Metode za rad sa stringovima

length() - vraća duzinu stringa

charAt(int) - vraca element na datoj poziciji

. . .

Može se izdvojiti **podstring** iz većeg stringa metodom substring klase *String*.

```
String pozdrav = "Zdravo";
String s = pozdrav.substring(0, 3);
```

Pravi se string počevši od pozicije 0, dužine 3, osnosno s je "Zdr". Dužina podstringa s.substring(a, b) je b-a.

Stringovi

Operator + se koristi za **nadovezivanje** stringova

```
String p = "Jedno";
String r = "drugo";
String m = p+r; // Promenljiva m je "Jednodrugo".
```

Ukoliko se na string nadoveže promenljiva koja nije string, ona se konvertuje u string.

```
int godine = 18;
String odgovor = "N" + godine; // Promenljiva ocena ima vrednost "N18";
```

Ovo se najčešće koristi pri ispisu

```
System.out.println("Odgovor je " + odgovor);
```

Ukoliko treba **spojiti više stringova razdvojenih delimiterom**, koristite metodu join klase String

```
String svi = String.join("/", "S", "M", "L", "XL");  // Rezultat je "S/M/L/XL"
```

Stringovi

Za testiranje da li su **dva string jednaka** koristi se equals metoda, koja vraća true ako su s i t jednaki, u suprotnom vraća false.

```
if(s.equals(t)). . .      može i    "Zdravo".equals(pozdrav)
```

Može se koristiti i metod compareTo

```
if (pozdrav.compareTo("Zdravo")  ==  0) . . .
```

Za testiranje da li su **dva stringa jednaka do na mala i velika slova**, koristi se metoda equalsIgnoreCase klase *String*

```
"Zdravo".equalsIgnoreCase("zdravo")
```

Stringovi su nepromenljivi - klasa String nema metod kojim bi se mogao izmeniti string. Ukoliko želite da izmenite string, potrebno je da napravite novi string. Nije moguće izmeniti slovo `Z' na `z' u stringu "Zdravo". Ali je moguće napraviti novi string od starog

```
pozdrav = `z' + pozdrav.substring(1, 6);
```

Stringovi

Prazan string "" je string dužine nula. Može se proveriti da li je string prazan pomoću

```
String str = "";
if (str.length == 0) . . .    ili
if (str.equals("")) . . .
```

Osim toga, String promenljiva može imati posebnu vrednost, takozvanu **null vrednost**, koja označava da nijedan objekat trenutno nije pridružen toj promenljvoj.

Da bi se testiralo da li je string null koristiti

```
if (str == null) . . .
```

Nekada je potrebno testirati na obe vrednosti

```
If (str != null && str.length != 0) . . .
```

Greška je pozvati metod nad promenljivom koja ima vrednost null.

Stringovi

Formatirani string se pravi metodom format klase *String*. Koristi se slično kao prinf metoda, s tom razlikom što se rezultat piše u string, umesto na izlaz.

```
String s1 = "Zdravo",  s2 = "svima";

String s = String.format("%s, %s\n", s1, s2);
```

Klasa *String* ima više od 50 metoda. Neke od metoda su

```
int charAt(int index)
int compareTo(String str)
boolean equals(Object obj)
boolean equalsIgnoreCase(String obj)
boolean startWith(String prefiks)
boolean endsWith(String sufiks)
int indexOf(String str)
int length()
String substring(int beginIndex)
String substring(int beginIndex, endIndex)
String toLowerCase()
String toUpperCase()
Striing join(charSequence delimiter, CharSequence ¼ elements)
```

Čitanje iz stringa

Skener se može napraviti i nad stringom:

```
import java.util.Scanner;
. . .
String str = "Danas je 1 oktobar 2016";
Scanner s = new Scanner(str);
System.out.println(s.next()+ "*" + s.next() + ":");
System.out.println(s.nextInt() + "." + s.next() + " " + s.nextInt() + "." +
" " + "god.");
```

Jedna od metoda za promenu delimitera je useDelimiter(String pattern).

```
import java.util.Scanner;
. . .
Scanner scanner =
new Scanner("1,2,3,4,5,6").useDelimiter(",");
```

Primer

```
import java.util.Scanner;
public class CitanjeIzStringa
{
  public static void main(String[] args)
  {
    Scanner scanner = new Scanner("1,2,3,4,5,6").useDelimiter(",");
    while(scanner.nasNextInt())
    {
      int num = scanner.nextInt();
      if(num%2 == 0) System.out.printf("%d\n", num);
      else System.out.printf("Suma je neparna\n");
    }
  }
}
```

Primer

```
import java.util.Scanner;
public class CitanjeIzStringa
{
  public static void main(String[] args)
  {
    Scanner scanner = new Scanner("1,2,3,4,5,6").useDelimiter(",");
    while(scanner.nasNextInt())
    {
      int num = scanner.nextInt();
      if(num%2 == 0) System.out.printf("%d\n", num);
      else System.out.printf("Suma je neparna\n");
    }
  }
}
```

Zadaci

1. Napisati program koji za dati string dužine šest ispisuje sve elemente na parnim pozicijama.

2. Napisati program koji iz datog stringa, dužine veće od dva, izostavlja karakter koji se nalazi na sredini stringa, ili dva karaktera na sredini ukoliko je string parne dužine.

3. Napisati program koji spaja tri data stringa, tako da između njih postavlja " * " i ispisuje dobijeni string na izlaz.

Nizovi

Nizovi su strukture podataka koje sadrže kolekciju vrednosti istog tipa.

Svakoj od ovih vrednosti se može pristupiti pomoću njegovog indeksa u nizu. Na primer, ako je a niz, tada je a[i] i-ti član niza.

Niz se deklariše navođenjem uglastih zagrada nakon tipa članova niza.

```
int[] a;   // deklaracija niza a, a ima vrednost null
```

Niz se može inicijalizovati operatorom new.

```
int[] a = new int[100];     // niz od 100 promenljvih tipa int
```

Popunjava niz uzastopnim brojevima od 0 do 99

```
for(int i=0; i<100; i++)
  a[i] = i;
```

Prilikom kreiranja niza brojeva koji su prostog tipa, sve vrednosti članova se postavljaju na 0. Članovi niza boolean tipa se postavljaju na vrednost false. Članovi referencijalnog tipa se postavljaju na null.

Nizovi

Kreira se niz od 100 objekata klase String koji imaju vrednosti null

```
String[] imena = new String[100];
```

Ukoliko se žele prazni stringovi, mogu se inicijalizovati na ovaj način

```
for(int i; i<100; i++)
  imena[i] = "";
```

Broj elemenata niza se dobija metodom *length*

```
for(int i=0; i<imena.length; i++)
  System.out.println(imena[i]);
```

Nizovi

Ukoliko je potrebno proći kroz ceo niz, može se koristiti sledeća petlja

```
for(promenljiva : kolekcija) naredba;
```

Ovim izrazom se postavlja promenljiva na svaki član niza redom i za te vrednosti se izvršava naredba. Kolekcija mora biti niz ili neki objekat klase koja implementira *Iterable* interfejs

```
int[] a = {3, 4, 5, 6, 7};
for(int element : a)
  System.out.println(element);
```

Isto je što i

```
for(int i=0; i<a.length;  i++)
  System.out.println(a[i]);
```

Nizovi

U Javi se može kreirati i inicijalizovati niz u isto vreme.

```
int[] malibrojevi = {1, 2, 3, 4, 5};
//nije potrebno koristiti new operator.
```

Može se kopirati jedan niz u drugi, ali tada nizovi dele iste članove, dakle ne prave se kopije članova implicitno

```
int[] srecnibrojevi = malibrojevi;
srecnibrojevi[3] = 14;
// malibrojevi[3] takodje ce imati vrednost 14
```

Ukoliko je potrebno dobiti sopstvenu kopiju svih članova, koristi se copyOf metoda klase *Arrays*

```
int[] kopija = Arrays.copyOf(malibrojevi, malibrojevi.length);
```

Često se ova metoda koristi da se poveća ili promeni dužina niza

```
int[] kopija = Arrays.copyOf(malibrojevi, 2 * malibrojevi.length);
```

Nizovi

Svaki Java program ima glavnu metodu (main metodu) sa argumentom *String[] args*. Ovaj argument ukazuje na to da glavna metoda prihvata niz stringova, kao takozvane **argumente komandne linije**.

```
public class Poruka
{
  public static void main(String[] args)
  {
    if(args.length == 0 || args[0].equals("-z"))
      System.out.println("Zdravo");
    else if(args[0].equals("-d")){
      System.out.println("Dovidjenja,");
      for(int i=1; i<args.length; i++)
        System.out.print(" " + args[i]);
    }
    System.out.println("!");
  }
}
```

Pokrenite program sa java Poruka -d toplo leto. Ispisaće se

Dovidjenja, toplo leto!

Sortiranje niza

Za sortiranje niza brojeva može se koristiti metod sort klase *Arrays*,

```
int[] a = new int[1000];
Arrays.sort(a);
for(int e : a)
  System.out.println(e);
```

Ovaj metod implementira quicksort algoritam. Postoje i druge metode koje implementiraju druge algoritme.

Nizovi

Višedimenizonalni nizovi imaju više od jednog indeksa za pristup svojim elementima. Koriste se za predstavljanje tabela, ili nekih složenije organizovanih podataka.

```
int[][] niz = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
```

```
int[][] obican = new int[5][10];
// promenljiva niz je inicijalizovana
// clanovi su postavljeni na 0
```

Članovima se pristupa sa obican[i][j].

```
for(int i=1; i<obican.length; i++)
  for(int j=1; j<obican[i].length; j++)
    obican[i][j] = + obican[i-1][j-1] + 1;
```

Izlaz?

Nizovi

Primer: Transponovanje matrice

```
Scanner in = new Scanner(System.in);
int n = 5, m = 3; float[][] a = new float[n][m], b = new float[m][n];

for (int i = 0; i < n; i++) {
  System.out.print("Vrsta " + (i + 1) + "? ");
  for (int j = 0; j < m; j++)
    a[i][j] = in.nextFloat();

}

for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++)
    b[j][i] = a[i][j];
}

for (int i = 0; i < m; i++) {
  for (int j = 0; j < m; j++)
    System.out.print(b[i][j] + " ");
  System.out.println();
}
```

Nizovi

Java zapravo nema dvodimenzionalne nizove, već ima **jednodimenzionalne nizove nizova**. Svaki od članova niza predstavlja niz za sebe, tako da se sa njime može raditi kao sa običnim nizom. Na primer, mogu se zameniti članovi (koji su nizovi).

```
int[] temp = magic[i];
magic[i] = magic[i+1];
magic[i+1] = temp;
```

Jednostavno je napraviti i niz nizova različitih dužina. Prvo treba alocirati niz koji čuva vrste.

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

Zatim, alocirati nizove koji predstavljaju vrste.

```
for(int i=0; i<10; i++)
  trougaoni[i] = new int[i+1];
```

Nizovi

Elementima se može pristupiti na prethodno opisani način

```
for (int i = 0; i < trougaoni.length; i++) {
  trougaoni[i][i] = trougaoni[i][0] = 1;
  for (int j = 1; j < i; j++) {
    trougaoni[i][j] = trougaoni[i - 1][j] +
          trougaoni[i - 1][j - 1];
  }
}
```

```
for (int i=0; i<n; i++) {
  for (int j=0; j<i; j++)
   System.out.print(" " + trougaoni[i][j]);
  System.out.println();
}
```

Izlaz?

Zadaci

1. Napisati program koja učitava sa ulaza broj elemenata niza, a zatim i elemente niza realnih brojeva. Ispisuje na izlaz sve elemente koji su manji od srednje vrednosi niza.

2. Napisati program koji učitava sa ulaza broj elemenata niza, a zatim i elemente niza celih brojeva. Obrće niz i ispisuje niz na izlaz.

3.Napisati program koji učitava sa ulaza broj elemenata niza, a zatim i elemente niza celih brojeva. Učitava sa ulaza i broj koji treba izbaciti iz niza. Izbacuje iz niza sva pojavljivanja unete vrednosti i ispisuje dobijeni niz na izlaz.

4. Napisati program koji učitava sa ulaza dimenzije matrice m i n, kao i elemente matrice celih brojeva, a zatim ispisuje na izlaz zbirove po vrstama i po kolonama.

Klase i objekti

Objekti

Posmatrajmo nastavni proces u školi. Nastava se održava u učionicama sa oznakama u1, u2 i u3. Za svaku od njih imamo broj mesta, informaciju da li je kabinet sa računarima ili ne. Ako je kabinet sa računarima, imamo i broj računara.

Učenici i nastavnici su deo nastavnog procesa, koji se izvodi u raspoloživim učionicama.

Učenici Pera i Ana, pripadaju odeljenju IIIc, dok Marko pripada odeljenju IIIa. Njima se postavljaju neki od zadataka z1, z2, z3, z4.

Postoje lekcije l1, l2, koje mogu predavati nastavnici Mira i Dušan trećoj godini, odeljenjima a, b i c. Nastavnici mogu da ispituju učenike određene zadatke.

Možemo uočiti različite **objekte** ovog nastavnog procesa

učionice u1, u2, u3; učenike Peru, Anu, Marka; nastavnike Miru, Dušana

zadatke z1, z2, z3, z4; lekcije l1, l2; razred: III; odeljenja: a, b, c

Klase, apstrakcija

Uočavamo da neki objekti imaju slične uloge i zajednička svojstva.

Nastavnici Mira i Dušan koordiniraju nastavnim procesom, predaju lekcije, vrše proveru znanja učenika. Oni imaju zajednička svojstva: ime, prezime, adresa, predmet koji predaju, godine radnog staža. Oni pripadaju jednoj **klasi** učesnika u nastavnom procesu, odnosno, klasi *Nastavnici*.

Na sličan način uočavamo da Pera, Ana i Marko pripadaju klasi *Učenici*. Oni imaju svojstva od važnosti ime, prezime, adresa, datum rođenja, razred, odeljenje. Imaju i svojstva koja su od manje važnosti za nastavni proces kao što su boja očiju, boja kose i slično.

Osnovni nivo **apstrakcije** je uočavanje šablona za objekte sa više zajedničkih osobina.

Atributi, metode

Svojstva od važnosti za proces koji pratimo nazivamo **atributima**.

Učenicima možemo pridružiti i akcije koje su značajne za proces: učenik radi zadatak i dobija ocenu, učenik prelazi u sledeći razred ili menja odeljenje.

Akcije u okviru klase nazivamo **metodama**. Njima se opisuje funkcionalnost objekata te klase.

Klasa *Učenik* je uopštenje, ili **apstrakcija**, pojedinačnih učenika. Svaki pojedinačni učenik je primerak, ili **instanca** klase *Učenik*.

Marko je objekat klase *Učenik*, a vrednost njegovog atributa *ime* je "Marko", vrednost atributa *razred* je "III".

Atributi, metode

Nastavnik

atributi: ime, prezime, adresa, radni_staž, predmet

metode: predajeLekciju(lekcija, razred, odeljenje)

ispituje(ucenik, zadatak)

pomeniAdresu(nova_adresa)

Učenik

atributi: ime, prezime, adresa, razred, odeljenje

metode: radiZadatak(zadatak)

prelaziUSledećiRazred()

pomeniAdresu(nova_adresa)

Enkapsulacija

Svaki objekat u objektno orjentisanom programiranju predstavlja samostalnu, zaokruženu celinu. Svaki objekat ima svoja unutrašnja stanja (atribute), kao i skup metoda (ili akcija) koje može da izvrši.

Enkapsulacija je postupak objedinjavanja stanja i ponašanja objekta u jednu celinu. Može se odnositi i na zaštitu podataka i kontrolu pristupa, takozvano učaurivanje.

Ovakav pristup omogućava lakše organizovanje rada sa većim skupom podataka. Za upotrebu objekata dovoljno je znati načine komunikacije sa njima (metode koje nude), bez poznavanja njihove realizacije.

Dozvoliti pristup poljima samo preko metoda te klase, a svim ostalim metodama zabraniti pristup. Programi treba da rade sa poljima klase samo preko metoda te klase. Enkapsulacijom se objekat pretvara u crnu kutiju koja ima neke osobine i nudi određene servise.

Nasleđivanje

Uočavamo da klase *Učenik* i *Nastavnik* imaju zajedničke atribute ime, prezime i adresu, i neke zajedničke metode.

Možemo napraviti generalizaciju, tako što pravimo novu klasu *Osoba*. Tada se klasa *Osoba* **specijalizuje** u klase *Učenik* ili *Nastavnik*, dok se klase *Učenik* ili *Nastavnik* **generalizuju** klasom *Osoba*.

Klase koje su specijalizacija neke druge, osnovne klase, nasleđuju sve njene atribute i metode. Na primer, metoda *promeniAdresu* klase *Osoba* se može primeniti i na klasu *Učenik* i na klasu *Nastavnik*.

Izvedene klase, osim nasleđenih osobina i funkcionalnosti, mogu posedovati i specifične osobine i funcionalnosti.

Izvedenu klasu *Učenik* opisujemo dodatnim atributima: *razred*, *odeljenje*, kao i dodatnim funkcionalnostima: *radiZadatak*, *prelaziUSledećiRazred*.

Nasleđivanje obezbeđuje smanjenje i lakše održavanje koda. Zajedničke osobine i funkcionalnosti se pišu samo jednom, dok se njihove eventualne izmene vrše samo u osnovnoj klasi.

Nasleđivanje

- *Osoba*

  atributi: ime, prezime, adresa

  metode: pomeniAdresu(nova_adresa)

  ispisiOsnovneInformacije()

- *Nastavnik* nasleđuje klasu *Osoba*

  atributi: (nasleđeni +)

  radni_staž, predmet

  metode: (nasleđene +)

  predajeLekciju(lekcija, razred, odeljenje)

  ispituje(ucenik, zadatak)

- *Učenik* nasleđuje klasu *Osoba*

  atributi: (nasleđeni +)

  razred, odeljenje

  metode: (nasleđene +)

  radiZadatak(zadatak)

  prelaziUSledećiRazred()

Polimorfizam

Objekti različitih izvedenih klasa mogu da se ponašaju različito izvršavajući iste funkcionalnosti osnovne klase.

U izvedenim klasama možemo predefinisati neke od nasleđenih funkcionalnosti, ukoliko izvedena klasa ima svoje specifičnosti.

Osobina da se isti metod osnovne klase izvršava drugačije u zavisnosti kojoj izvedenoj klasi objekat koji ga poziva pripada, naziva se **polimorfizam**.

Na primer, neka osnovna klasa *Osoba* ima metod *ispisiOsnovneInformacije*. Objekat klase *Učenik* može da ispisuje ime, prezime, razred i odeljenje, dok objekat klase *Nastavnik* može da ispisuje ime, prezime i predmet koji nastavnik predaje.

Struktura objektno orjentisanog programa

Tradicionalno strukturno programiranje se sastoji od dizajniranja skupa procedura ili algoritama kojima se rešava problem. Jednom kada se definišu procedure, potrebno je naći odgovarajuće strukture za smeštanje podataka.

Objektno orjentisano programiranje ima drugačiji redosled. Prvo se smeštaju podaci, a zatim se pronalazi odgovarajući način da se rukuje podacima.

Objektno orjentisani program je sačinjen od objekata. Svaki objekat ima svoju funkcionalnost, koja je izložena korisnicima, i skrivenu implementaciju.

Za male probleme svođenje na definisanje pocedura funkcioniše dobro. Ali za veće probleme objekti su primereniji. Na primer, veb pretrazivač ima oko 2000 procedura. OOP stilom bi postojalo 100 klasa, sa oko 20 metoda.

Jednostavnije je raditi sa manjim objektima, a zatim spajati te objekte radi ostvarenja cilja, umesto da se odjednom rukuje svim procedurama.

Objektno orjentisana paradigma

Zasniva se na pravljenju i korišćenju tipova (klasa) koji omogućavaju grupisanje podataka, tako i delova programa koji ove podatke koriste. Vrednosti (instance) ovih tipova se nazivaju objektima.

Sav kod koji se piše mora se nalaziti u nekoj klasi. Standardna Java biblioteka sadrzi oko nekoliko hiljada klasa za rešavanje problema iz raznih domena. I dalje je potrebno definisati sopstvene klase koje rešavaju specificni problem. Tipovi pomoću kojih se u Javi prave objekti se nazivaju referencijalni tipovi. U Javi postoji pet vrsti referencijalnih tipova

- Klase
- Interfejsi
- Nizovi
- Nabrojivi tipovi
- Lambda izrazi

Referencijalni tipovi su složeni tipovi podataka. Zovu se jos i struktuirani tipovi.

Objektno orjentisana paradigma

Da bi se izvodilo OOP potrebno je **identifikovati sledeće**

- **Ponašanje objekta**, šta se može uraditi sa objektom, koje metode treba da ima.
- **Stanje objekta**, kako se objekat ponaša kada se pozovu ove metode.
- **Identitet objekta**, kako se taj objekat prepoznaje među drugim objektima koji mogu imati isto ponašanje i stanje.

Svi objekti iste klase imaju isto **ponašanje**. Svaki objekat čuva informaciju o tome sta sadrži, a to se naziva **stanje objekta**.

Objektno orjentisana paradigma

Najčešće **relacije medju klasama** su

- **zavisnost** (koristi)

  Klasa zavisi od druge klase kada koristi metode koje ta klasa ima.

  Porudžbina treba da ima pristup računu, pa stoga klasa Porudžbina koristi klasu Račun.

  Treba nastojati da se smanji broj klasa koje zavise jedne od drugih. Kada klasa A ne zavisi od klase B, tada izmena klase B ne može prouzrokovati greske u klasi A koje nastaju korišćenjem klase B.

- **spajanje** (ima)

  Porudžbina sadrži (ima) robu.

- **nasleđivanje** (je)

  Pas i Mačka su klase koje se mogu generalizovati u jednu klasu, Životinja. Pas je Životinja.

Definisanje novih klasa

Na samom početku smo napravili klasu koja ima samo main metodu. Sada je vreme da napravimo klase koje su potrebne za sofisticiranije aplikacije. Klase uglavnom nemaju main metodu. Umesto toga, one imaju svoja polja i metode. Da bi se napravio potpun program treba kombinovati više klasa, od kojih samo jedna ima main metodu.

Najjednostavniji oblik definicije klase u Javi je

```
class ImeKlase
{
  polje1; polje2;
  ...
  konstruktor1; konstruktor2;
  ...
  metoda1; metoda2;
  ...
}
```

Definisanje novih klasa

Primer jednostavne klase kojom se može opisati tačka

```
class Tacka
{
  float x, y;                          /* polja klase */
     Tacka(float X, float Y){          /* konstruktor */
     x = X; y = Y;                     /* dodeljuje se vrednost poljima klase */
  }

  void transliraj(float pomerajX, float pomerajY){        /* metod klase */
    x = x + pomerajX; y = y + pomerajY;
  }

  public String toString(){            /* metod klase */
    return "Tacka (" + x + "," + y+ ")";
  }
}
```

Definisanje novih klasa

Primer korišćenja klase Tacka

```
class TackaTest {
  public static void main(String[] args){
    Tacka A;                    /* A ne pokazuje jos uvek ni na jedan objekat */
    A = new Tacka(2, 3);                /* A je referenca na novi objekat */
    System.out.println(A);      /* poziva se metoda toString klase Tacka */
    A.transliraj(1, 2);                 /* menja se objekat  putem metode*/
    System.out.println(A);
    A.x = 0;                    /* menja se objekat direktnim pristupom */
    A.y = 0;
    System.out.println(A);
  }
}
```

Definisanje novih klasa

Primer: dve promenljive i jedan objekat. A i B su reference na isti objekat.

```
class TackaTest2
{
  public static void main(String[] args){
    Tacka A = new Tacka(2, 3);
    System.out.println(A);

    Tacka B = A;          /* ne vrsi se kopiranje  objekta,
                             već kopiranje reference */
    B.transliraj(1, 2);    /* dakle, menja se
                           objekat na koji pokazuje i A */
    System.out.println(A);
  }
}
```

Definisanje novih klasa

Primer: Operatori za ispisivanje jednakosti. Jednakost se odnosi na reference, a ne na stanje objekta.

```
class TackaTest3 {
  public static void main(String[] args){
    Tacka A = new Tacka(2, 3);     /* kreira se novi       objekat */
    Tacka B = new Tacka(2, 3);   /* kreira se novi objekat, sa istim stanjem */
    Tacka C = A;                          /* kreira se nova referenca na objekat */
    Tacka D = null;                       /* kreiraju se reference koja ne */
    Tacka E = null                        /* pokazuju ni na jedan objekat */
    System.out.println(A == B? "A == B" : "A != B");
    System.out.println(A == C? "A == C" : "A != C");
    System.out.println(B == C? "B == C" : "B != C");
    System.out.println(D == E? "D == E" : "D != E");
  }
}
```

Zadatak

Napisati klasu Tacka, kao u prethodnim primerima. Napisati zatim klase TackaTest, TackaTest2, TackaTest3 i pokrenuti ih.

Preopterećivanje (overloading)

Preopterećivanje je korišćenje istog imena ili simbola za označavanje više različitih konstrukcija u jeziku.

U Javi se ovaj pojam odnosi na mogućnost deklarisanja više istoimenih metoda u jednoj klasi.

Ove metode se moraju međusobno razlikovati po broju ili tipu svojih formalnih parametara. Koji će se metod izvršiti zavisi od tipa stvarnih parametara sa kojima će se metod pozvati.

Kompajler bira koju će metodu upotrebiti. Projavljuje grešku ukoliko ne postoji odgovarajuća metoda, ili postoji više metoda za koje se ne zna koja je adekvatniji izbor.

Isto važi i za konstruktore, kojih takođe može biti više u istoj klasi.

Potpis metode se sastoji od naziva i tipova parametara koje ona prihvata. Ne možete imati dve metode sa istim potpisom.

Povratna vrednosti nije deo potpisa, tako da ne mogu postojati dve metode sa istip potpisom čak iako imaju različitu povratnu vrednost.

Preopterećivanje

Primer: više metoda sa istim imenom

```
float rastojanje(){   /* rastojanje od  koordinatnog pocetka */
  return Math.sqrt(x*x + y*y);
}

float rastojanje(Tacka t){   /* rastojanje od neke tacke */
  return Math.sqrt(Math.pow(x-t.x, 2) + Math.pow(y-t.y, 2));
}

class TackaTest4 {
  public static void main(String[] args){
    Tacka A = new Tacka(2, 3);
    System.out.println(A.rastojanje());
    System.out.println(A.rastojanje(new  Tacka(5, 6)));
  }
}
```

Preopterećivanje

Primer: više konstruktora

```
Tacka(){
  x = 1; y = 1;  /* konstruktor bez argumenata, stanje polja
                    se postavlja na unapred definisano stanje */
}

Tacka(float xa, float ya){
  x = xa; y = ya;  /* polja se postavljaju na
                      prosledjene vrednosti */
}


Tacka(Tacka t){
  x = t.x; y = t.y;  /* kopiraju se vrednosti x i y
                        prosledjene tacke */
}


Tacka(float xa){
  x = xa; y = 0;   /* postavlja se samo jedna
                      koordinata */
}
```

Preopterećivanje

Primer: test više konstruktora

```
class TackaTest5 {
  public static void main(String[] args){
    Tacka A = new Tacka();
    Tacka B = new Tacka(2, 3);
    Tacka C = new Tacka(A);
    Tacka D = new Tacka(2);

    System.out.println(A.rastojanje(B));
    System.out.println(C.rastojanje(A));
  }
}
```

Parametri metoda

Razmotrimo temine koji se koriste u računarstvu za opisivanje načina kako se parametri mogu preneti u metodu. Termin prenos po vrednosti, označava da metoda uzima vrednost koja joj je prosleđena. U suprotnom, prenos po referenci označava da metoda dobija lokaciju promenljive. Stoga, metoda može promeniti vrednost promenljive čija joj je lokacija prosleđena. To nije slučaj pri prenosu po vrednosti. Pri prenosu po vrednosti dobija se samo kopija promenljive.

U Java programskom jeziku **uvek se vrši prenos po vrednosti**. To znači da metod dobija kopije svih vrednosti parametara. Metoda ne može promeniti vrednosti promenljivih koje su korišćene kao parametri. Ali, ako je prosleđena lokacija objekta, to znači da metoda dobija kopiju te lokacije, pa samim tim može promeniti objekat koji se nalazi na toj lokaciji. Ali ne i samu lokaciju.

Postoje dve vrste parametara

- primitivni tipovi (brojevi, boolean vrednosti)
- reference objekata

Parametri metoda

Primer sa prostim tipom double

```
public static void utrostruci(double x) // ne radi
{
   x = x * 3;
}
```

```
double procenat = 10;
utrostruci(procenat);
```

Nakon poziva procenat ima vrednost 10. Ovo ne radi, jer je prosleđena kopija parametra, koja je pomnožena sa 3.

Parametri metoda

Situacija je drugačija sa referencama objekata

```
public static void utrostruciPlatu(Osoba x) // radi
{
  x.plata *= 3;
}
```

```
ana = new Osoba(. . .);
utrostruciPlatu(ana);
```

Kada pozovemo metodu, plata će biti povećana, pošto se prosleđuje kopija reference objekta, pa se objekat može menjati.

Parametri metoda

Razmotrimo sledeći slučaj. Pošto se prenosi kopija reference, to znači da se sama ta referenca ne može menjati

```
public static void zameni(Osoba a, Osoba b) // ne radi
{
  Osoba pom = a;
  a = b;
  b = pom;
}
```

```
Osoba a = new Osoba(. . .);
Osoba b = new Osoba(. . .);
zameni(a, b);
```

Zamena neće biti izvršena, pošto smo zamenili samo kopije referenci, ali ne i same reference.

Parametri metoda

Primer:

```
class Osoba {
    String ime, prezime;
    double plata;

    Osoba(String i, String p){
        ime = i; prezime = p; plata = 100;
     }

    String toString() {
        return ime + " " + prezime;
    }
}
```

Parametri metoda

```
class Sef {
    String odeljenje;
    String datum;

    Sef(String o) {
        odeljenje = o; datum = new Date();
    }

    void utrostruciPlatu(Osoba x){
        x.plata *= 3;
    }
    void zameni(Osoba a, Osoba b){
       Osoba t = a; a = b; b = t;
    }
    String toString() {
        return "Sef na odeljenju " + odeljenje + " od dana " + datum;
    }
}
```

Parametri metoda

```
class OsobaTest{
    public static void main(String[] args){
      Osoba[] osobe = new Osoba[3];
      osobe[0] = new Osoba("Ana", "Ilic");
      osobe[1] = new Osoba("Pera", "Peric");
      osobe[2] = new Osobe("Mika", "Lazic");

      Sef sef = new Sef("Prodaja");
      System.out.println(sef);
      sef.utrostruciPlatu(osobe[0]);
      System.out.println(osobe[0] + ", plata: " + osobe[0].plata);

      sef.zameni(osobe[0], osobe[2]);
      System.out.println(osobe[2]);     // nije zamenjeno

      Osobe t = osobe[0]; osobe[0] = osobe[2]; osobe[2] = t;
      System.out.println(osobe[2]); // jeste zamenjeno
    }
}
```

Parametri metoda

Svaka metoda ima jedan **implicitni** parametar (objekat nad kojim je pozvana) i nula ili vise **eksplicitnih** parametara (koji su navedeni u potpisu metode).

Implicitni parametar se referiše ključnom rečju this.

```
Tacka(float xa, float ya){
  x = xa; y = ya;
}
```

Isto je što i

```
Tacka(float xa, float ya){
  this.x = xa; this.y = ya;
}
```

Zadaci

Enkapsulacija

Kod svake klase se mogu uočiti dva nivoa, nivo dizajna klase i nivo implementacije klase.

Pod dizajnom podrazumevamo načine na koje će se objektima te klase pristupati, odnosno kako klasa izgleda spolja. Kojim poljima se može pristupati izvan tela klase, kojim metodima, i šta ti metodi treba da urade.

Pod implementacijom klase podrazumevamo konkretnu realizaciju svega onoga što smo prilikom dizajniranja klase zamislili.

Svaki Java program se sastoji od klasa. Da bi se program lakše pravio, odžavao, modifikovao, kao i da bi se omogućio timski rad na pravljenju programa, veoma je pogodno da implementacija svake klase ostane skrivena koliko god je to moguće. Kako je neka klasa implementirana to treba da zna samo programer koji je tu klasu napravio. Kreatori drugih klasa ne moraju da znaju ništa o implementaciji te klase kako bi mogli da je koriste.

Skrivanje detalja implementacije nazivamo **enkapsulacijom**. U Javi je omogućeno maksimalno korišćenje enkapsulacije prilikom pisanja programa. Iako se programi mogu pisati i bez enkapsulacije, taj stil se ne preporučuje.

Enkapsulacija

Šta se dobija enkapsulacijom?

Posmatrajmo našu klasu Tacka. U njoj nema skrivanja detalja implementacije, zbog čega druge klase mogu da pristupaju poljima klase.

```
class TackaTest{
  public static void main(String[] args){
    Tacka A = new Tacka(2, 3);
    A.transliraj(12f, 13f);
    System.out.println("Nove koordinate (" + A.x  + "," + A.y + ")");
                                                          /* kriticno.*/
  }
}
```

Navedeni program direktno pristupa poljima objekta A. Ukoliko je polje klase vidljivo svima, to znači da svi programi koji ga direktno koriste zavise od njegove implementacije.

Pretpostavimo da je nakon pisanja gore navedenog programa došlo do potrebe da se implementacije klase promeni. Na primer, potrebno je dodati metod za rotaciju tačke oko koordinatnog početka, koji će se znatno više koristiti. Da bi implementacija klase bila što efikasnija, potrebno je čuvati polarne koordinate umesto Dekartovih koordinata.

Enkapsulacija

Modifikacija prvobitne klase Tacka

```
class Tacka{
  float ugao, duzina;
  Tacka(float x, float y){
    duzina = nadjiDuzinu(x, y);
    ugao = nadjiUgao(x, y);
  }
  float nadjiDuzinu(float x, float y) { ¼ }
  float nadjiUgao(float x, float y) { ¼ }
  float nadjiX(float ugao, float duzina) { ¼ }
  float nadjiY(float ugao, float duzina) { ¼ }
  void rotiraj(float ugao) { . . . }
  void transliraj(float pX, float pY) {
      float noviX = nadjiX(ugao, duzina) + pX;
      float noviY = nadjiY(ugao, duzina) + pY;
      duzina = nadjiDuzinu(noviX, noviY);
      ugao = nadjiUgao(noviX, noviY);
  }
  String toString() {  System.out.println("Tacka ("+
      nadjiX() + "," + nadjiY() + ")");  }
}
```

Enkapsulacija

Klasa je izgubila polja x i y, a dobila ugao i duzinu. Konstruktor i metode transliraj i toString() su pretrpele promene zbog prelaska na polarne koordinate. Dodate su metode rotiraj, nadjiX, nadjiY, nadjiUgao i nadjiDuzinu.

Nakon ovakve modifikacije, program koji je pristupao poljima x i y više neće raditi. Kao ni svi ostali takvi programi. U našem primeru nije teško uvesti izmenu. Međutim, situacija bi bila znatno gora ako bi se starim poljima klase Tacka pristupalo na nekoliko desetina mesta u programu sa nekoliko hiljada redova programskom koda.

Enkapsulacija

Prvobitna klasa Tacka uz korišćenje enkapsulacije

```
class Tacka
{
  private float x, y;     /* skrivanje polja postize se modifikatorom private */
  Tacka(float X, float Y){
    x = X; y = Y;
  }
  void transliraj(float pomerajX, float pomerajY){
     x = x + pomerajX; y = y + pomerajY;
  }
  public String toString(){
     return "Tacka (" + x + "," + y + ")";
  }
  float dajY() { return y; }      /* sada poljima pristupamo preko metoda */
  float dajX() { return x; }
}
```

Enkapsulacija

Izmenjeni program koji koristi klasu Tacka

```
class TackaTest{
  public static void main(String[] args){
    Tacka A = new Tacka(2, 3);
    A.transliraj(12f, 13f);
    System.out.println("Nove koordinate (" + A.dajX() + "," + A.dajY() + ")");
  }
}
```

Umesto direktnog pristupa poljima x i y, program koristi metode nadjiX i nadjiY.

Ovime je omogućeno da se implementacija klase može nesmetano menjati i prilagođavati potrebama bez uznemiravanja programa koji koriste objekte te klase.

Enkapsulacija

Modifikacija prvobitne klase Tacka korišćenjem enkapsulacije

```
class Tacka{
  private float ugao, duzina;
  Tacka(float x, float y){
    duzina = nadjiDuzinu(x, y);
    ugao = nadjiUgao(x, y);
  }
  float nadjiDuzinu(float x, float y) { . . . }
  float nadjiUgao(float x, float y) { . . . }
  float nadjiX(float ugao, float duzina) { . . . }
  float nadjiY(float ugao, float duzina) { . . . }
  void rotiraj(float ugao) { . . . }
  void transliraj(float pX, float pY) { . . . }
  float dajX() { return nadjiX(); }
  float dajY() { return nadjiY(); }
  public String toString(){
        System.out.println("Tacka ("+ dajX() + "," + dajY()+")"); }
}
```

Nakon ove izmene program koji koristi klasu ne treba menjati, jer će on i dalje dobro raditi bez obzira na krupne promene koje je pretrpela klasa Tacka.

Enkapsulacija

Modifikatori koji utiču na vidljivost klase

Svaki referencijalni tip napravljen u okviru svog paketa je vidljiv svim ostalim referencijalnim tipovima u tom paketu. Da bi referencijalni tip bio vidljiv i van svog paketa, mora se deklarisati pomocu modifikatora public.

```
public class Javna {
  /* clanovi klase */
}
```

Modifikator public je jedini modifikator koji se može koristiti da bi se uticalo na vidljivost referencijalnog tipa izvan paketa.

Enkapsulacija

Modifikatori koji utiču na vidljivost članova klase

- *bez modifikatora* - vidljiv za sve klase iz istog paketa

- **public** - može da pristupi svako ko može da pristupi klasi

- **protected** - vidljiv je za sve klase u okviru istog paketa i u svim podklasama svoje klase koje se nalaze u drugim paketima. Za konstruktor vazi izuzetak: on se sa new može pozvati samo u okviru paketa u kome je definisan, a u podklasi pomoću ključne reči super

- **private** - vidljiv samo u svojoj klasi

Opisani modifikatori određuju vidljivost na nivou klasa, a ne objekata, što znači da ukoliko polje ima modifikator private, biće vidljivo svim objektima iste klase, a ne samo objektu kome pripada.

```
Tacka(Tacka a){
  x = a.x; y = a.y;        /* x je privatno polje, ali mu se moze pristupati
                                 iz drugog objekta iste klase */
}
```

Enkapsulacija

Može se definisati polje sa oznakom **final**, u značenju da se ne može promeniti jednom kada se inicijalizuje.

Ovakvo polje se mora inicijalizovati do kraja konstrukcije objekta. Na primer, polje ime klase Osoba može da se označi sa final, pošto se ne menja nakon kreiranja objekta.

```
class Osoba{
  private final String ime;
  ¼
}
```

Enkapsulacija

Za nepromenljive klase je jasno kako funkcioniše oznaka final. Međutim, za klase koje mogu da menjaju stanje malo je zbunjujuće.

private **final** StringBuilder rez;

Inicijalizuje se pomoću konstruktora

rez = new StringBuilder();

Oznaka final znači da ce rezultati uvek pokazivati na ovaj objekat, ali to ne znači da taj objekat neće moći da se menja.

```
rez.append(LocalDate.now() + ": Gold star!\n");
```

Enkapsulacija

Ukoliko se ne navede public ili private, metodima klase se može pristupiti iz svih klasa tog paketa. Za metode je to dobro rešenje, ali za polja nije. Mora se eksplicitno navesti private da bi polje bilo privatno.

Zadaci

Statička polja, metode, konstante

Metode i polja koja imaju modifikator **static** nazivaju se statičkim.

Statički članovi klasa se mogu koristiti i bez prethodnog kreiranja objekta, odnosno instance klase. Oni se zapravo ne odnose ni na koju konkretnu instancu klase, već na samu klasu.

Svaki objekat sadrži svoju kopiju svih polja i metoda klase, osim onih polja i metoda koji su statički, jer statička polja i metode pripadaju klasi, a ne objektima.

Statičke metode ne smeju pristupati poljima koja nisu statička, niti smeju pozivati metode koje nisu statičke. One su nezavisne od bilo koje instance, pa tako ne smeju koristiti ništa sto je specifično za neku instancu, već samo ono što pripada klasi.

Statička polja, metode, konstante

Ako definišete **polje kao static**, to znači da postoji samo jedno takvo polje po klasi. U suprotnom, svaki objekat ima svoju kopiju tog polja. Na primer, hoćemo da postavimo jedinstvenu oznaku za svaku osobu.

```
class Osoba{
  private static int  nextId = 1;
  private int id;
  . . .
}
```

Svaki objekat klase Osoba ima svoj primerak polja id, ali postoji samo jedan primerak polja nextId. Čak i ako ne postoji ni jedan kreirani objekat klase Osoba, polje nextId postoji i ima svoju vrednost.

Statička polja, metode, konstante

```
public void postaviId() {
    id = nextId++;
}
```

```
Osoba ana = new Osoba(. . .);
ana.postaviId();
```

Postavlja tekuću vrednost id-a ani, i povećava vrednost na sledeći broj. Zapravo desilo se sledeće

```
ana.id = Osoba.nextId;
Osoba.nextId++;
```

Statičkim članovima se pristupa preko imena klase, za kojim sledi tačka, pa naziv člana.

Statička polja, metode, konstante

Statičke promenljive nisu tako česte. Češće se koriste **statičke konstante**. Klasa Math definiše neke od statiških konstanti.

```
public class Math{
  public static final double PI = 3.14159265358979323846;
  ...
}
```

Ovoj konstanti se iz programa može pristupiti preko **Math.PI**.

Ukoliko ova konstanta ne bi imala oznaku static, morao bi da se kreira objekat klase Math da bi mogla da se koristi. Takođe, svaki objekat te klase bi imao svoju instancu konstante PI.

Statička polja, metode, konstante

Druga statička konstanta koju smo već koristili je **System.out**. Ona je deklarisana na sledeći način

```
public class System{
  public static final PrintStream out = . . .
  . . .
}
```

Za konstante je u redu da budu javne, pošto ionako ne može da se menja njihova vrednost.

Statička polja, metode, konstante

Statičke metode su metode koje ne rade nad objektima. Na primer, računanje stepena

Math.**pow**(x, a)

Izračunava x^a. Ne koristi ni jedan objekat za izvršavanje računanja, odnosno nema implicitni parametar. Može se razmišljati o statičkim metodama kao o metodama koje nemaju parametar this. Statičke metode mogu pristupati samo statičkim poljima, jer samo ona mogu postojati bez kreiranja objekta.

```
public static int dajSledeciId(){
  return nextId;
}
```

```
int n = Osoba.dajSledeciId();
```

Statička polja, metode, konstante

Metode koje nisu static mogu se pozvati samo nad objektom. Statičke metode se mogu pozvati i nad objektom, ali to deluje zbunjujuće.

Statičke metode treba koristiti u sledećim situacijama

- Kada metod ne zahteva pristup stanju objekta, jer su svi parametri eksplicitni (Math.pow)

- Kada metod pristupa samo statičkim poljima (Osoba.dajSledeciId).

Statička polja, metode, konstante

Metode za konverziju iz Dekartovih koordinata u polarne možemo deklarisati kao statičke, zato što one ne koriste nikakve informacije o samim objektima, vec vrše konverziju nezavisno od objekata.

```
class Tacka{

  private float ugao, duzina;

  Tacka(float x, float y){
    duzina = nadjiDuzinu(x, y);
    ugao = nadjiUgao(x, y);
  }

  static float nadjiDuzinu(float x, float y) { . . . }
  static float nadjiUgao(float x, float y) { . . . }
  static float nadjiX(float ugao, float duzina) { . . . }
  static float nadjiY(float ugao, float duzina) { . . . }
  void rotiraj(float ugao) { . . . }
  void transliraj(float pX, float pY) { . . . }

  float dajX() { return nadjiX(); }
  float dajY() { return nadjiY(); }

  public String toString() {
          System.out.println("Tacka ("+ dajX() + "," + dajY()+")"); }
}
```

Statička polja, metode, konstante

Statičke metode se mogu koristiti i bez postojanja instance klase.

```
class Program {
  public static void main(String[] args){
    for(int x = -10; x <= 10; x++)
      for(int y = -10; y <= 10; y++)
        System.out.println("(" + x + "," + y + ") u Dekartovim je (" +
          Tacka.nadjiUgao(x, y) + "," + Tacka.nadjiDuzinu(x, y) + ")" +
          " u polarnim koordinatama" );
  }
}
```

Primetimo da u ovom programu nije napravljena nijedna instanca klase Tacka.

Statička polja, metode, konstante

Još jedno često korišćenje statičkih metoda je korišćenje **statičkih fabričkih metoda** za konstruisanje objekata. Klase kao sto su LocalDate i NumberFormat koriste fabričke metode da konstruisu različite objekte. Već smo videli metode LocalDate.now i LocalDate.of koje kreiraju objekte. Klasa NumberFormat takodje može da kreira objekte

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter  = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x));        // ispisuje $0.10
System.out.println(percentFormatter.format(x));         // ispisuje 10%
```

Zašto se ne koristi konstruktor u ove svrhe? Zato što se ne mogu dati različita imena konstruktorima, dok u ovom slučaju želimo dva različita objekta currency instance i percent instance.

Statička polja, metode, konstante

Main metoda je statička iz razloga što nju možemo pozivati iako ne postoji objekat.

```
public class Aplikacija{
  public static void main(String args[]){
    // konstrukcija objekata . . .
  }
}
```

Main metoda ne radi ni nad jednim objektom. Kada se program pokrene, prvo se izvrši main metoda, koja zatim konstruiše objekte koji su potrebni.

Svaka klasa može da ima main metodu, ali se main izvršava samo iz klase koja je navedena pri pozivu programa.

java Osoba

Ako je klasa Osoba deo veže aplikacije, aplikacija se pokreće sa

java Aplikacija

i main metoda klase Osoba se nikada ne izvršava.

Zadaci

Konstrukcija objekata

Predstavimo konstruktore malo detaljnije.

Telo konstruktora je slično telu bloka. Od bloka se razlikuje u tome što prva naredba u telu konstruktora može biti eksplicitni poziv drugog konstruktora.

Postoje tri vrste eksplicitnog poziva konstruktora u telu konstruktora

- pozivanje drugog konstruktora iste klase - ključna reč **this**
- pozivanje konstruktora direktne natklase - ključna reč **super**
- pozivanje konstruktora direktne natklase kada je direktna natklasa unutrašnja klasa - ključna reč **super**

Konstrukcija objekata

Eksplicitni poziv konstruktora iste klase. Klasa može da ima više konstruktora koji se razlikuju u broj u i tipu parametara. Kod velikih i kompleksnih klasa se obično pravi jedan opšti konstruktor koji se može koristiti za pravljenje svih instanci te klase. Takav konstruktor može da ima mnogo parametara, od kojih se mnogi pozivaju sa istom vrednošću.

Zbog toga je pogodno napraviti prilagođene konstruktore, koji u sebi pozivaju ovaj opšti konstruktor.

Konstruktor iste klase se poziva **na samom početku konstruktora** i to navođenjem ključne reči **this**.

Konstrukcija objekata

Primer

```
public class Ucenik {
  private String ime, adresa, razred, odeljenje;

  public Ucenik(String i, String a, String r, String o){
    ime = i; adresa = a; razred = r; odeljenje = o;
  }

  public Ucenik(String i, String a){
    this(i, a, "", "");
  }
  . . .
}
class Program{
  public static void main(String[] args){
    Ucenik A = new Ucenik("Aca", "Trg Nikole Pasica");
    Ucenik B = new Ucenik("Pera", "Cvetni Trg bb",  "treci", "a");
  }
}
```

Konstrukcija objekata

Klasa ne mora imati deklarisan ni jedan konstruktor. Takvoj klasi će Java prevodilac u toku prevođenja dodati konstruktor bez parametara.

Ukoliko klasa ima definisan bar jedan konstruktor sa jednim ili više parametara, ne može se kreirati objekat pozivom konstruktora bez parametara.

Ukoliko se želi i takav način kreiranja objekta, konstruktor bez parametara se mora eksplicitno definisati.

Konstrukcija objekata

Vrednosti se mogu postaviti i u **inicijalizacionom bloku**. Klasa može imati više inicijalizacionih blokova. Oni se izvršavaju **kad god se neki objekat te klase kreira**. Izvršavaju se redosledom kojim su navedeni u deklaraciji klase.

```
class Osoba
{
  private static int nextId; private int id;
  private String ime;
  private double plata;

  {
    id = nextId;                          /*Prvo se izvršava inicijalizacioni blok */
    nexId++;                               /*pa onda konstruktor. */
  }

  public Osoba(String ime, double plata)
    { this.ime = ime; this.plata = plata; }

  public Osoba() { ime = ""; plata = 0; }
  . . .
}
```

Konstrukcija objekata

Inicijalizacioni blokovi mogu biti i **statički**. Oni uglavnom služe za inicijalizaciju statičkih polja klase. Statički inicijalizator je programski kod koji se izvršava **prvi put kada se klasa koristi u programu**. U jednoj klasi može biti više statičkih inicijalizatora.

Statičko polje se može inicijalizovati pustivši da se dodeli
- podrazumevana vrednost public static int nextId;
- da se vrednost dodeli pri definiciji public static int nextId = 1;
- ili da se inicijalizuje u statičkom bloku, ukoliko je inicijalizacija složena

```
static
{
  Random generator = new Random();
  nextId = generator.nextInt(10000);
}
```

Na ovaj način nextId će dobiti slučajnu vrednost manju od 10000.

Konstrukcija objekata

Kao što smo videli, postoji više načina kako se polja mogu inicijalizovati. Proces konstrukcije objekta može delovati zbunjujuće. Ovo je detaljan prikaz šta se tačno dešava kada se pozove konstruktor

1. Sva polja klase se inicijalizuju na podrazumevane vrednosti (0, false ili null)

2. Sve inicijalizacije u definiciji polja se izvršavaju kao i svi inicijalizacioni blokovi

3. Ako je prva linija konstruktora drugi konstruktor, on se izvršava

4. Izvršavaju se naredbe konstruktora koji je pozvan

Konstrukcija objekata

Imena parametara mogu da zadaju glavobolju. Možete izabrati jednoslovna imena.

```
public Osoba(String i, double p)
{
  ime = i;
  plata = p;
}
```

Tada će možda biti teže razumeti kod posto ta imena nisu dovoljno opisna. Prvi predlog je da imena budu nazivi polja sa nekim prefiksom, kako bi jednostavno moglo da se zaključi značenje parametra.

```
public Osoba(String aIme, double aPlata)
{
  ime = aIme;
  plata = aPlata;
}
```

Konstrukcija objekata

Drugi predlog je da imenima parametara date isto ime kao i imenima polja. Tada će polja sa tim nazivima biti sakrivena, pa se moraju referisati pomoću this.

```
public Osoba(String ime, double plata)
{
  this.ime = ime;
  this.plata = plata;
}
```

Konstrukcija objekata

Konstruktori mogu biti pozvani samo pri kreiranju objekta u paru sa operatorom new. **Ne može se pozvati konstruktor nad već kreiranim objektom da bi se reinicijalizovale promenljive.**

```
Tacka A = new Tacka(1, 2);
A.Tacka(4, 5);   /* prijavice se greska prilikom kompilacije */
```

Nekoliko stvari treba naglasiti

- konstruktor ima isti naziv kao i klasa
- klasa može imati više konstruktora
- konstruktor može imati nula ili više argumenata
- konstruktor nema povratnu vrednost
- konstruktor se uvek poziva u kombinaciji sa operatorom new

Uklanjanje objekata

Java ima ugrađen mehanizam pod nazivom **garbage collector**, koji uklanja sve objekte koji se više ne koriste.

Međutim, nekada je potrebno eksplicitno ukloniti objekat. To se može uraditi tako što se doda metod finalize.

Ovaj metod se izvršava pre nego sto garbage collector uništi objekat, ali se ne zna tačno kada, tako da se ne treba pouzdati u njegovo izvršavanje ukoliko se u njemu izvršavaju operacije od kojih zavisi neka druga radnja.

Na primer, ukoliko je potrebno osloboditi resurse, kao što je otvorena datoteka, što pre je moguće, to treba uraditi ručno pozivom metode close za zatvaranje datoteke.

Zadaci

Paketi

Java omogućava grupisanje klasa u kolekcije. Paketi su uobičajeni način za organizaciju rada i razdvajanja našeg koda od drugih bibilioteka.

Standardna biblioteka se nalazi u paketima **java.util**, **java.lang** i drugim.

Paketi su hijerarhijska organizacija i ekvivalent su direktorijumima i poddirektorijumima.

Najveći razlog za postojanje paketa je obezbeđivanje jedinstvenosti imena klasa. Klase u različitim paketima se mogu isto zvati.

Sa gledišta kompajlera, paketi nisu povezani jedni sa drugima. Pa tako, **java.util** i **java.util.jar** nemaju zajedničkih tačaka. Svaki od njih ima svoju kolekciju klasa.

Paketi

Klasa može koristiti sve klase iz svog paketa i sve javne klase drugih paketa. Javnoj klasi nekog paketa se može pristupiti sa navođenjem punog naziva paketa ispred naziva svake klase.

java.time.LocalDate today = java.time.LocalDate.now();

Ovo je svakako nepregledno. Uobičajenije, i jednostavnije, je korišćenje **import** izraza pre definicije klase. Kada se uveze neki paket u program, svim njegovim klasama se može priistupiti bez navođenja imena paketa. Na primer, mogu se uvesti sve klase iz paketa navodjenjem *

import java.util.*;

ili samo jedna klasa

import java.time.LocalDate;

Ne može se koristiti java.*.*.

Paketi

U vecini slučajeva ovakvo uvoženje je u redu. Slučaj kada treba razmotriti šta dalje raditi je kada se pojavi konflikt imena. Na primer, oba paketa java.util i java.sql imaju klasu Date.

import java.util.*;

import.java.sql.*;

Ukoliko koristite Date klasu, dobićete grešku. Možete da dodate import klase koju koristite

import java.util.*;

import.java.sql.*;

import java.util.Date;

Ukoliko vam zaista trebaju obe klase Date, potrebno je navesti pun naziv paketa.

java.util.Date rok = new java.util.Date();

java.sql.Date danas = new java.sql.Date();

Lociranje klasa radi kompajler. Bajt kod uvek sadrži pune nazive paketa.

Paketi

Pomoću import izraza mogu se uvoziti i statičke metode i polja, ne samo klase.

```
import static java.lang.System.*;
```

Sada se mogu korititi statičke metode i polja bez navođenja prefiksa

```
out.println("Dovidjenja!");
exit(0);
```

Može se uvesti i poseban metod ili polje.

```
import static java.lang.System.out;
```

U praksi, mnogi programeri koriste pun naziv System.out i System.exit, umesto bez prefiksa. Sa druge strane, sledeći zapis je mnogo čitljiviji

```
sqrt(pow(x, 2) + pow(y, 2));
```

od zapisa

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
```

Paketi

Da bi se klasa dodala nekom paketu, potrebno je da na početku datoteke navedete ime paketa, pre koda kojim se definiše klasa.

```
package code.mg.oop;
public class Osoba{
. . .
}
```

Ukoliko se ne navede ime paketa, onda klasa pripada podrazumevanom paketu. Podrazumenvani paket nema ime.

Smestite datoteke sa kodom u poddirektorijum koji odgovara punom imenu paketa. Na primer, datoteke u paketu sa imenom code.mg.oop treba da budu u direktorijumu code/mg/oop na Linux-u. Na Windows-u je to direktorijum code\mg\oop. Kompajler smešta class fajlove na istu adresu.

Paketi

Sledeći program je smešten u dva paketa. PaketTest klasa pripada podrazumevanom paketu, pa se nalazi u početnom direktorijumu. Klasa Osoba pripada code.mg.oop paketu, stoga, datoteka Osoba.java mora da se nalazi u poddirektorijumu code/mg/oop.

(početni direktorijum):

PaketTest.java

PaketTest.class

code/mg/oop/Osoba.java

Da bi se kompilirao ovaj program potrebno je promeniti tekući direktorijum na početni direktorijum (na primer komandom cd)

javac PaketTest.java

Kompajler će automatski pronaći code/mg/oop/Osoba.java i kompilirati ga.

Paketi

Realističniji primer je

(početni direktorijum):

code/mg/oop/Osoba.java

code/mf/PaketTest.java

javac code/mf/PaketTest.java

java code.mf.PaketTest

Program može da se kompilira iako se paket ne nalazi tamo gde je navedeno, ali je potrebno da se nalazi prilikom izvršavanja programa.

Class Path

Klase su smeštene u direktorijume na fajl sistemu. Putanja do klase se mora poklapati sa imenom paketa. Class datoteke se mogu smeštati u JAR datoteke. JAR datoteka sadrži više class datoteka i poddirektorijuma u kompresovanom formatu. Kada koristite tuđe bibilioteke, uglavnom ćete dobiti jar fajl da uključite u svoj program. JDK ima brojne jar fajlove koji sadrže hiljade klasa. Kreiranje jar arhive:

jar cf nazivDatoteke.jar naziviDatotekaKojeArhiviramo

Da podelite klase između programa potrebno je da uradite sledeće

1. Smestite svoju class datoteku u direktorijum, na primer, /home/user/classdir. Ovo će biti početni direktorijum za pakete. Ukoliko dodate klasu code.mg.oop.Osoba, tada se Osoba.class mora smestiti u /home/user/classdir/code/mg/oop/Osoba.class.

2. Smestite neku JAR datoteku unutar direktorijuma, na primer /home/user/arhiva

3. Postavite class path, to je kolekcija svih lokacija koje sadrže class datoteke

Class Path

U UNIX-u su elementi u class path razdvojeni dvotačkom

/home/user/classdir:.:/home/user/arhiva/arhiva.jar

Na Windows-u su razdvojeni tačkom zarezom.

c:\classdir;.;c:\arhiva\arhiva.jar

U oba slučaja tačka označava tekući direktorijum.

Class path sadrži početni direktorijm (/home/user/classdir), tekući direktorijum (.) i JAR datoteku (/home/user/arhiva.jar). Može se koristiti džoker karakter * za direktorijum u kome se nalazi JAR datoteka.

/home/user/classdir:.:/home/user/arhiva/*

ili

c:\classdir;.;c:\arhiva*

Class Path

Sve JAR datoteke, ali ne i class datoteke, su dodate iz direktorijuma arhiva.

Class path sadrži sve direktorijume koji su početni za klase koje treba uključiti. Pretpostavimo da virtuelna mašina traži fajl code.mg.oop.Osoba.class. Ona prvo gleda u direktorijume u kome se nalaze sistemske klase jre/lib i jre/lib/ext. Ukoliko ne nađe klasu tu, okreće se ka class path. Traži sledeće datoteke

- /home/user/classdir/code/mg/oop/Osoba.class
- code/mg/oop/Osoba.class u tekućem direktorijumu
- code/mg/oop/Osoba.class u JAR arhivi /home/user/arhiva/arhiva.jar

Class Path

Kompiler ima teži zadatak. Ukoliko navedete klasu a ne navedete paket, kompiler prvo mora da nađe paket koji sadrži tu klasu. Konsultuje sve import direktive, ukoliko datoteka sa kodom sadrži uvoze

```
import java.util.*;
import code.mg.oop.*;
```

> i u kodu se koriti Osoba, kompiler tada traži java.lang.Osoba (pošto je java.lang uvek uključeno), java.util.Osoba, code.mg.oop.Osoba i Osoba u tekućem paketu. On pretražuje sve klase na svakoj od ovih putanja. Ukoliko pronađe više od jedne klase, javlja se greška.

Kompiler radi još jednu stvar, proverava da li je vreme izmene koda skorije od vremena nastanka class datoteka. Ukoliko jeste, rekompilira kod automatski.

Class Path

Najbolje je da se class path navede opcijom -classpath

java -classpath /home/user/classdir:.:/home/user/arhiva.jar MojProgram

java -classpath c:\classdir;.;c:\arhiva.jar MojProgram

Drugi način je da postavite promenljivu okruženja CLASSPATH. U terminalu sa Boure Again shell-om (bash) bi to izgledalo ovako

export CLASSPATH=/home/user/classdir:.:/home/user/arhiva.jar

ili

set CLASSPATH=c:\classdir;.;c:\arhiva.jar

Class path je postavljena dok je terminal otvoren.

Preporuke za dizajn klase

1. Uvek čuvajte podatke tako da budu privatni

2. Inicijalizujte podatke

3. Ne koristite previše osnovnih tipova u klasi. Sve njih možete zameniti novom klasom

4. Nije potrebno da svako polje ima svoj geter i seter

5. Rasparčajte klase koje imaju previše zaduženja

6. Imenujte klase i metode tako da se iz njih može lakše zaključiti čemu služe

7. Preferirajte nepromenljive klase

Nasleđivanje

Još jedan od osnovnih koncepata kod izvedenih klasa je nasleđivanje. Ideja je da se mogu praviti klase koje su zasnovane na nekim drugim, već postojećim, klasama.

Kada se vrši nasleđivanje od neke druge klase, nasleđuju se sva njena polja i metode.

Takođe, mogu se dodavati nove metode i nova polja. Ova tehnika je ključna u Java programiranju.

Klase, potklase, natklase

Neka imamo klasu Zaposleni definisanu na sledeći način

```
class Zaposleni
{
  private String ime;
  private double plata;

  Zaposleni(String ime, double plata){
    this.ime = ime;
    this.plata = plata;
  }
  public String dajIme(){ return ime; }
  public double dajPlatu() { return plata; }
}
```

Pretpostavimo da radimo u firmi gde se menadžeri tretiraju dugačije od ostalih zaposlenih.

Menadžeri su isti kao i zaposleni u mnogim aspektima.

Svi primaju platu. Od zaposlenih se očekuje da urade posao da bi primili platu, dok menadžeri dobijaju bonuse ukoliko urade predviđeni posao. Ova situacija vapi za nasleđivanjem.

Treba da se definiše nova klasa Menadzer i da joj se doda nova funkcionalnost. Mogu se pozajmiti sve osobine koje ima Zaposleni, jer menadžer i jeste zaposleni.

Ovo je relacija "pripada", ili "is-a".

Ovako se definiše Menadzer klasa koja nasleđuje klasu Zaposleni

```
public class Menadzer extends Zaposleni
{
   dodatne metode i polja
}
```

Extends ključna reč označava da se nova klasa pravi kao **proširenje** stare klase.

Stara klasa se naziva **natklasa**, **osnovna klasa**, **roditeljska klasa**. Nova klasa se naziva **potklasa**, **izvedena klasa**, **dete klasa**.

Termini potklasa i natklasa se najčešće koriste.

Klasa Zaposleni je natklasa ne zato što je nadređena svojoj natklasi, ili ima više metoda. Zapravo je obrnuto. Potklase imaju veću funkcionalnost nego natklase.

Naš menadžer ima novo polje bonus i novu metodu

```
public class Menadzer extends Zaposleni
{
  private double bonus;
  . . .
  public void postaviBonus(double bonus){
     this.bonus = bonus;}
}
```

Nema ničega posebnog u ovome. Ako imate objekat klase Menadzer možete pozvati metodu postaviBonus.

```
Menadzer sef = new Menadzer(. . .);
sef.postaviBonus(5000);
```

Naravno, ukoliko imate objekat klase Zaposleni, ne možete na njega primeniti ovu metodu.

Objekat klase Menadzer može koristiti polja klase Osoba, ime, plata, i metode dajIme(), dajPlatu(). Svi članovi nadređene klase su automatski nasleđeni u potklasi.

Kada se definiše potklasa, potrebno je samo navesti razlike, odnosno ono što je dodato u odnosu na natklasu. Pri dizajniranju klasa, najopštije funkcionalnosti se smeštaju u natklasu, dok se one specifičnije smeštaju u potklasu. Ovakva terminologija je preuzeta iz teorije skupova.

Neke metode natklase nisu pogodne za potklasu takve kakve su. Na primer, metod dajPlatu natklase vraća samo osnovnu platu, dok je za menadžera potrebno vratiti osnovnu platu plus bonus.

```
public class  Manager extends Zaposleni
{
  public double dajPlatu()  { . . . }
  . . .
}
```

Na prvi pogled reklo bi se da može samo da se predefiniše metoda

```
public double dajPlatu()
{
  return plata + bonus;   // ne radi
}
```

Ali, ovo ne radi. Prisetite se da samo metode klase Zaposleni imaju pristup privatnim poljima klase Zaposleni. To znači da dajPlatu klase Menadzer ne može direktno pristupiti polju plata. Ukoliko menadžer metod želi da pristupi ovom privatnom polju, treba da uradi isto što i druge metode, da pozove metod dajPlatu nadređene klase.

```
public double dajPlatu()
{
  double plata = super.dajPlatu();    // radi
  return plata + bonus;
}
```

Poziv dajPlatu bi pozivao sebe, tako da ako želimo da pozovemo metodu natklase koristimo ključnu reč super.

Pomoću nasleđivanja možemo dodati polja i metode, možemo ih predefinisati, ali ih se nikada ne možemo osloboditi.

Da bismo kompletirali nas primer, napravimo konstruktor.

```
public Menadzer(String ime, double plata)
{
  super(ime, plata);
  bonus = 0;
}
```

Ovde ključna reč super ima dugačije značenje. Poziva konstruktor natklase Zaposleni. Pošto metode klase Menadzer ne mogu pristupiti poljima nasleđenim od natklase Zaposleni, postrebno je pozvati konstruktor. Ukoliko ne pozovemo konstruktor natklase, pozvaće se konstruktor bez argumenata. Ukoliko natklasa nema konstruktor bez argumenata i potklasa ne poziva ni jedan drugi konstruktor javiće se greška.

```
Menadzer sef = new Menadzer("Menadzer1", 80000);
sef.postaviBonus(5000);
```

Pravimo niz od tri zaposlena

```
Zaposleni[] osoblje = new Zaposleni[3];

osoblje[0] = sef;
osoblje[1] = new Zaposleni("Hery haker", 50000);
osoblje[2] = new Zaposleni("Tony tester", 40000);
```

Ispisujemo informacije za sve zaposlene

```
for(Zaposleni z: osoblje)
System.out.println(z.dajIme() + " " + z.dajPlatu());
```

Menadzer1 85000
Hery hasker 50000
Tony tester 40000

Primetite da osoblje[1] i osoblje[2] ispisuju osnovnu platu, dok osoblje[0] ispisuje bonus plus osnovnu platu.

Šta je vredno pomena u pozivu metode

z.dajPlatu();

Ovaj poziv bira pravi metod dajPlatu. Deklarisani tip za z je Zaposleni, ali on može pokazivati i na Zaposeni i na Menadzera.

Kada se z odnosi na Zaposlenog tada z.dajPlatu poziva metod klase Zaposleni.

Kada se z odnosi na Menadzera z.dajPlatu poziva metod klase Menadzer.

Virtuelna mašina zna na koga se odnosi z, pa tako poziva pravi metod.

Činjenica da jedna promenljiva koja pokazuje na objekat (kao sto je z) može da se odnosi na više konkretnih tipova se naziva **polimorfizam**.

Automatski izbor odgovarajućeg metode se naziva **dinamičko razrešavanje**.

Klasa Zaposleni ili Menadzer mogu i na dalje da budu natklase za neke nove izvedene klase.

Zadaci

Polimorfizam

Jednostavno pravilo se može koristiti da se odluči da li je nasleđivanje pravi izbor za konkretan problem, a to je pravilo supstitucije: ukoliko je svaki objekat potklase ujedno i objekat natklase, odnosno ukoliko se može koristiti potklasa u svakoj situaciji kada se očekuje natklasa. Na primer, svaki Menadzer je Zaposleni, ali nije svaki Zaposleni Menadzer.

Zaposleni z;

z = new Zaposleni(. . .);

z = new Menadzer(. . .);

U programskom jeziku Java objektne promenljive su polimorfne.

Promenljiva klase Zaposleni može da pokazuje na Zaposleni, ili Menadzer, ili IzvrsniMenadzer, ili Programer, . . .

Primer

```
Menadzer sef  = new Menadzer(. . .);
Zaposleni[] osoblje = new Zaposleni[3];
osoblje[0] = sef;              // pokazuju na isti objekat.
```

Može se pozvati

```
sef.postaviBonus(5000);                  // ok
```

ali ne može se pozvati

```
osoblje[0].postaviBonus(5000);  // greska
```

Deklarisani tip osoblje[0] je Zaposleni, a postaviBonus nije metod klase Zaposleni.

Ne može se dodeliti referenca natklase promenljivoj potklase

```
Menadzer m = osoblje[2];                    // greska
```

Razlog je jasan, nisu svi zaposleni menadžeri. Ukoliko bi ova dodela mogla da uspe, i ukoliko je osoblje[2] objekat klase Zaposleni, tada bismo mogli da pozovemo m.postaviBonus(. . .). Ali to ne bi trebalo da možemo zato što objekat klase Zaposleni nema taj metod.

Nizovi potklasa mogu da budu konvertovani u nizove natklasa bez kastovanja

```
Menadzer[] menadzeri = new Menadzer[10];
Zaposleni[] osoblje = menadzeri;                    // ok
```

menadzeri[0] pokazuje na isto što i osoblje[0]

```
osoblje[0] = new Zaposleni(. . .);                  // nije ok
```

Ovo deluje u redu, ali onda bismo mogli da pozovemo
menadzeri[0].postaviBonus(1000);

Međutim, svaki niz pamti sa kojom je klasom prvobitno bio definisan, stoga on neće dozvoliti smeštanje nekompatibilnih referenci. U ovom slučaju ako se pokuša smeštanje zaposlenog u osoblje[0], prijaviće se greška.

Veoma je važno da znamo kako tačno se metod primenjuje na objekat. Recimo da pozovemo x.f(args), gde je x objekat klase C.

1. Kompiler gleda deklarisani tip objekta x i ime metode f. Naravno, može postojati više metoda f, ali sa različitim tipovima parametara, na primer f(String) ili f(int). Kompiler nabrojava sve metode f klase C, i sve dostupne (vidljive) metode f nadklase od C.

2. Kompiler određuje tipove parametara koji su prosleđeni pozivom metode. Ukoliko među nabrojanim metodama postoji jedinstvena metoda u kojoj se poklapaju tipovi parametara, tada se bira ona. Ovaj proces se zove razrešavanje preopterećenja.

Na primer, pozivu f.("zdravo") više odgovara f(String) nego f(int). Situacija može da se zakomplikuje zbog konverzija int u double, Menadzer u Zaposleni i slično. Ukoliko kompiler ne može da nađe ni jedan metod sa odgovarajućim parametrima, javlja se greška.

3. Ako metod ima oznaku private, static, final, ili je konstruktor, tada kompiler zna tačno koji metod da pozove. Ovo se naziva **statičko razrešavanje**. U suprotnom, metod koji će biti pozvan zavisi od stvarnog tipa implicitnog parametra i tada se koristi **dinamičko razrešavanje** u vreme izvršavanja.

4. Kada se program izvršava i koristi dinamičko razrešavanje prilikom poziva metoda, virtualna mašina ce pozvati verziju metoda koja je odgovarajuća za stvarni tip objekta na koji se odnosi x.

Na primer, ukoliko je x tipa D, gde je D potklasa klase C, tada se poziva metod f(String) klase D. Ukoliko on ne postoji poziva se metod f(String) klase C, i tako dalje.

Pogledajmo na primeru

z.dajPlatu()

Metoda nema parametara tako da ne brinemo o preopretećivanju metoda. Objekat z je klase Zaposleni. Metoda dajPlatu nije static, final, private, tako da će se vršiti dinamičko razrešavanje.

Klasa Zaposleni ima na raspolaganju sledeće metode Zaposleni.dajIme(), Zaposleni.dajPlatu(), Zaposleni.dajDatumZaposljavanja() i Zaposleni.povecajPlatu(double).

Klasa Menadzer ima na raspolaganju sledeće metode

Zaposleni.dajIme(), Menadzer.dajPlatu(), Zaposleni.dajDatumZaposljavanja(), Zaposleni.povecajPlatu(double), Menadzer.postaviBonus().

U vreme izvršavanja, prvo se za konkretan objekat gleda koje metode ima na raspolaganju, a zatim se na osnovu toga čita ko je definisao taj metod i čiji metod zapravo treba da se pozove.

Kada se predefinišu metode, metoda potklase mora biti vidljiva bar onoliko koliko i metoda natklase. Ukoliko je metoda natklase definisana kao public, i metoda potklase mora da bude definisana kao public. Primer?

Sprečavanje nasleđivanja

Nekada je potrebno da se spreči da nešto može biti nasleđeno. Klase koje ne mogu da se proširuju imaju oznaku **final** u deklaraciji.

public **final** class Izvrsni extends Menadzer { . . . }

Može se napraviti final metod u klasi koja se može naslediti. Tada se označava da se taj metod ne može prepisati (override).

```
public class  Zaposleni
{
  . . .
  public final String dajIme() { return ime; }
}
```

Ako je klasa final, svi metodi su automatski final, ali ne i polja.

Postoji samo jedan dobar razlog da klasa bude final: da se osigura da njena semantika ne može da bude promenjena potklasom.

Klasa String je na primer final da bi se bilo sigurno da ukoliko imate referencu na String, da tada ona pokazuje na String i ništa drugo osim String-a.

Svakako treba pažljivo razmišljati o final metodama i klasama kada se radi o dizajnu hijerarhije klasa.

Kastovanje

Proces prinuđene konverzuje iz jednog tipa u drugi se naziva kastovanje. Java ima sledeći zapis za kastovanje

```
double x = 3.405;
int nx = (int) x;          // kastuje x u integer.
```

Isto tako može biti potrebno da se referenca jednog objekta kastuje u referencu drugog objekta.

```
Menadzer sef = (Menadzer) osoblje[0];
```

Postoji samo jedan razlog zašto bismo želeli da napravimo kastovanje, a to je da koristimo objekat sa punim kapacitetom iako je njegov stvarni tip bio privremeno zaboravljen.

Kompiler proverava da ne obećavamo previše kada smeštamo vrednost u promenljivu. Ako dodelimo referencu potklase promenljivoj nadklase, obećavamo manje i kompiler to dopušta. Ako dodelimo referencu natklase promenljivoj podklase, obećavamo više. Stoga moramo koristiti kastovanje kako bi naše obećanje bilo provereno u toku izvršavanja.

Šta se dešava ako pokušamo da kastujemo nasleđivanje, ali lažemo u vezi toga šta objekat sadrži?

```
Menadzer sef = (Menadzer) osoblje[1];        // greska
```

Kada se program pokrene, uočava se neispunjeno obećanje i prijavljuje se greška. Ukoliko se ne obradi greška program se prekida. Stoga, dobra je praksa da se proveri da li će kastovanje uspeti operatorom instanceof

```
if(osoblje[1] instanceof Menadzer) {
        sef = (Menadzer) osoblje[1]; . . .
}
```

Na kraju, kompiler neće dozvoliti kastovanje ako je očigledno da kastovanje neće proći

```
String c = (String) osoblje[1];
                        // ovo je greska u toku kompilacije.
```

Sumirano:
1) može se kastovati samo unutar hijerarhije nasleđivanja
2) koristiti operator instanceof za proveru pre kastovanja natklase u potklasu.

U suštini, konvertovanje kastovanjem treba **izbegavati**.

Kada se pozove metoda dajPlatu dinamičko privezivanje će rešiti polimorfizam. Jedino kada je potrebno koristiti neku metodu koju potklasa ima, a natklasa nema, na primer postaviBonus, potrebno je vršiti kastovanje.

Zadaci

Apstraktne klase

Kako se penje uz hijerarhiju nasleđivanja, klase postaju sve opštije i apstraktnije. U isto vreme, klasa predak postaje toliko opšta da se o njoj više misli kao o osnovi za ostale klase, nego o klasi sa specifičnim instancama koje želimo da koristimo.

Posmatrajmo proširenje hijerarhije naše klase Zaposleni. Zaposleni je Osoba, kao što je to i Djak. Zašto se zamarati ovolikim nivoom apstrakcije? Postoje atributi koji imaju smisla za svaku osobu, kao što je atribut ime.

Recimo da postoji metod **dajPuniOpis** koji se razlikuje za ove dve potklase. Jednostavno je implementirati ovaj metod za svaku od klasa, ali kako će on izgledati za klasu Osoba? Klasa Osoba ne zna ništa o osobi osim imena. Naravno može se implementirati Osoba.dajPuniOpis da vrati prazan sting, ali postoji bolji način. Ako se koristi ključna reč **abstract**, metod se ne mora ni implementirati.

public **abstract** String dajPuniOpis();

Klasa koja ima jedan ili više apstraktnih metoda, mora i sama biti označena kao apstraktna

```
public abstract class  Osoba
{
   private String ime;


   public Person(String ime)
   {
      this.name = ime;
   }
   public abstract String dajPuniOpis();
   public String getName()
   {
      return ime;
   }
}
```

Ukoliko proširite apstraktnu klasu imate dva izbora. Možete ostaviti neki ili sve apstraktne metode nedefinisane, tada i ta klasa mora imati oznaku abstract. Ili, možete definisati sve apstraktne metode i tada klasa nije više apstraktna.

Na primer, definišimo klasu Student koja je proširenje klase Osoba i implementira metodu dajPuniOpis.

Klasa Student više ne mora da bude apstraktna, s obizom na to da više nema apstraktnih metoda.

Mada, klasa može biti označena kao apstraktna čak iako nema apstraktnih metoda.

Ne mogu se praviti instance apstraktne klase. Dakle, ne može postojati objekat apstraktne klase.

```
new Osoba("Mika")       // greska
```

Ali može se kreirati objekat potklase koja nije apstraktna i smestiti u promenljivu koja je tipa apstraktne klase.

```
Osoba p = new Student("Mika", "Fizika");
```

Konkretna klasa Student koja proširuje apstraktnu klasu Osoba

```
public class Student extends Osoba
{
   private String major;
   public Student(String ime, String major{
      super(ime); this.major = major;
   }
   public String dajPuniOpis(){
      return "student majoring in " + major;
   }
}
```

```
Osoba[] osobe = new Osoba[2];
osoba[0] = new Zaposleni(. . .);
osoba[1] = new Student(. . .);
```

Tada možemo ispisati opise

```
for(Osoba o : osobe)
  System.out.println(o.dajIme() + ", " + o.dajPuniOpis());
```

Promenljiva o uvek se odnosi na neki konkretan objekat klase koja može da bude instancijalizovana, dakle ne može pokazivati na objekat klase Osoba.

Ukoliko metod dajPuniOpis ne bi bio definisan u klasi Osoba, već samo u klasi Zaposleni i Student, onda se on ne bi mogao pozvati nad promenljivom o.

Kompiler uverava da se mogu pozvati samo metode koje su deklarisane u klasi kojoj pripada objekat. Ovo je omogućeno postojanjem apstraktnim metoda.

Apstraktne metode su veoma ceste u Java programiranju. Najčešće će se naći u interfejsima.

Protected

Kao što smo rekli, polja je najbolje označiti kao privatna, a metode kao javne. Sve što je definisano kao privatno ne može se videti van klase. To važi i za nadklase: podklasa ne može videti članove nadklase ukoliko su deklarisani kao privatni.

Kada želite da ograničite metod samo na podklase ili da dozvolite metodi da pristupi poljima superklase, koristite oznaku **protected**.

Na primer, ako natklasa Osoba deklariše datumZaposljavanja kao protected umesto private, tada Menadzer može tom polju da pristupi direktno.

Klasa Menadzer može da pristupi samo poljima Menadzer klase, ali ne i poljima objekata klase Zaposleni. Razlog je da se ne može zloupotrebiti protected mehanizam kako bi se napravilo nasleđivanje, samo da bi se pristupilo protected poljima natklase.

Protected

Koristiti **protected polja** sa pažnjom. Pretpostavimo da vaša klasa ima protected polja i da je koriste drugi programeri. Oni mogu da naslede ovu klasu i da počnu da pristupaju vašim protected poljima. U ovom slučaju, ne može se izmeniti implementacija klase bez uznemiravanja ovih programera, što nije u duhu OOP-a koji podržava enkapsulaciju.

Protected metode imaju više smisla. Klasa može deklarisati metod kao protected ukoliko se pretpostavlja da može da veruje svojim potomcima, odnosno da će na ispravan način da koriste taj metod, dok ostalim klasama ne može tako verovati.

Dobar primer je clone metoda klase Objekat. (videti kasnije)

Sumirani modifikatori pristupa kojima se kontroliše vidljivost

1. vidljivo samo klasi - **private**
2. vidljivo svima - **public**
3. vidljivo samo paketu i svim potklasama - **protected**
4. vidljivo paketu - podrazumevano, *nisu potrebne oznake*

Zadaci

Superklasa Object

Klasa Object je zajednički predak svim klasama. Kako baš svaka klasa proširuje klasu Object, to je važno upoznati se sa funkcijama koje ona nosi sa sobom. Navešćemo neke od njih.

Možete koristiti promenljivu tipa Object

Object obj = new Zaposleni(. . .);

Naravno, ovo je jedino korisno ukoliko nam treba generički smeštaj za naše promenljive. Svakako kada želimo da koristimo zgodno je kastovati

Zaposleni zap = (Zaposleni) obj;

U Javi svi nizovski tipovi, bez obzira da li su nizovi objekata ili primitivnih tipova, takođe su proširenje klase Object.

```
Zaposleni[] osoblje = new Zaposleni[10];
obj = osoblje;              // ok
obj = new int[10];          // ok
```

Metod equals

Ukoliko nije drugačije definisano, ovaj metod vraća da li su dve reference identične.

Vrlo često ovo nije informacija koja nam je potrebna. Često se pod jednakošću misli na to da objekti imaju **isto stanje**, a ne da su im reference jednake.

U sledećem primeru je predefinisan **equals** metod tako da proverava jednakost stanja objekata.

Metod **getClass** vraća klasu objekta. U našem primeru, preduslov da dva objekta budu jednaka je da pripadaju istoj klasi.

```
public class Zaposleni
{
   . . .
   public boolean equals(Object otherObject){
      if (this == otherObject) return true;

      if (otherObject == null) return false;

      if (getClass() != otherObject.getClass())
         return false;

      Zaposleni other = (Zaposleni) otherObject;

      return ime.equals(other.ime)
         && plata == other.plata
         && datumZaposljenja.equals(other.datumZaposljenja);
   }
}
```

Kada definišete metod equals u potklasi, prvo pozovite metod equals natklase, pa ako taj test ne prođe, ne mogu ni potklase da budu jednake.

```
public class Menadzer extends Zaposleni
{
   . . .
   public boolean equals(Object otherObject){
      if (!super.equals(otherObject)) return false;
      // super.equals proverava da this i
      // otherObject pripadaju istoj klasi
      Menadzer other = (Menadzer) otherObject;
      return bonus == other.bonus;
   }
}
```

Kako equals treba da se ponaša kada implicitni i eksplicitni parametri nisu iz iste klase?

Ukoliko je semantika ista za sve potklase onda se može koristiti **instanceof**. Ukoliko se semantika equals menja u zavisnosti od potklase onda se koristi **getClass**.

Ukoliko želite da se jednakost ne poredi tačno po klasi već može i po natklasi, onda se može koristiti operator instanceof. Međutim, treba voditi računa da ovaj operator nije simetričan, odnosno da **x.equals(y)** ne mora biti isto što i **y.equals(x)**.

- Ukoliko se zaposleni porede po svim poljima, onda se koristi getClass.

- Ukoliko se zaposleni porede po ID-ju, onda nije bitno da su iz iste klase, već je bitno da su iz klase Zaposleni ili neke njene potklase, onda može da se koristi instanceof.

Da li uočavate problem?

```
public class Zaposleni
{
   public boolean equals(Zaposleni other){
      return other != null
         && getClass() == other.getClass()
         && Objects.equals(ime, other.ime)
         && plata == other.plata
         && Objects.equals(datumZaposljavanja,
other.datumZaposljavanja);
   }
   . . .
}
```

Ukoliko hoćete da budete sigurni da ste predefinisali postojeći metod dodajte pre deklaracije oznaku **@Override**.

Metod hashCode

Heš kod je ceo broj koji se računa na osnovu parametara objekta. Metod **hashCode**() je metod klase **Object** koji vraća heš kod objekta. Svaki objekat ima već ugrađeno izračunavanje heš koda i to na osnovu adrese u memoriji na kojoj se nalazi.

Svaka klasa može predefinisati način na koji izračunava heš kod.

Na primer, za sledeće stringove su to vrednosti:
Ivana 71029095 Ivano 71029109 Ana65972

Algoritam po kome se računa vrednost za klasu String je:

```
int hash = 0;
for (int i = 0; i < length(); i++)
  hash = 31 * hash + charAt(i);
```

Za proste tipove se koristi statička funkcija klase Objects:

static int hashCode(int|long|short|byte|double|float|char|boolean v)

Heš kod klase Zaposleni može se računati na osnovu imena, prezimena, plate, datuma zapošljenja, ili neke kombinacije.

```
public int hashCode() {
  return 7 * Objects.hashCode(ime)
                    + 11 * Objects.hashCode(prezime)
                    + 13 * Objects.hashCode(plata)
                    + 17 * Objects.hashCode(datumZap);
}
```

Metod hash izračunava sam na osnovu ugrađenog algoritma.

```
public int hashCode() {
  return Objects.hash(ime, prezime, plata, datumZap);
}
```

Ukoliko predefinišete **equals** metod tada bi trebalo da predefinišete i **hashCode** metod.

Dva objekta za koje equals vraća true, metod hashCode bi trebalo da vrati istu vrednost. Sa druge strane, heš kodovi različitih objekata mogu biti isti.

Metod toString

Već smo se upoznali sa ovim metodom.

```
public String toString()
{
   return getClass().getName() +
      "[name=" + name +
      ",salary=" + salary +
      ",hireDay=" + hireDay + "]";
}
```

```
public class Menadzer extends Zaposleni
{
   . . .
   public String toString(){
      return super.toString() +
         "[bonus=" + bonus + "]";
   }
}
```

Sada se objekat tipa Menadzer ispisuje kao
Manager[name=. . .,salary=. . .,hireDay=. . .][bonus=. . .]

```
Menadzer p = new Menadzer(. . .);
String message = "Info: " + p;
// automatski se poziva p.toString()
```

Generička klasa ArrayList

Veličina niza se može zadati u toku izvršavanja programa

```
int n = . . .;
Zaposleni[] osoblje = new Zaposleni[n];
```

Ovaj kod ne rešava potpuno potrebu za dinamičkim prolagođavanjem veličine niza u toku izvršavanja programa. Jednom kada se postavi veličina niza, teže se naknadno menja.

U Javi se sa ovakvim situacijama može izaći na kraj uz pomoć klase ArrayList. Ova klasa je slična nizovima, s tim što ona automatski prilagođava svoju veličinu broju elemenata u nizu, bilo da se oni dodaju ili brišu iz niza.

ArrayList je generička klasa koja ima za parametar tip članova niza, koji se navodi između zagrada < i >.

```
ArrayList<Zaposleni> osoblje =
                    new ArrayList<Zaposleni>();
```

Ne mora se drugi put ni navesti tip, ukoliko on može da se zaključi iz tipa onoga gde se smešta referenca (promenljiva, parametar metoda, . . .)

```
ArrayList<Zaposleni> osoblje = new ArrayList<>();
```

Ranije nije ni postojalo navođenje tipova, već se koristila jedna klasa za sve, sada se to zove "row" tip. Takođe, ranije se koristio i tip Vector za dinamičke nizove, ali kako je sadašnja klasa ArrayList efikasnija, ne postoji više potreba za korišćenjem klase Vector.

Članovi se dodaju pomoću

osoblje.**add**(new Zaposleni(. . .));

Ukoliko se popuni prostor niz će se magično proširiti i prekopirati stare objekte na novu lokaciju.

Ovo kopiranje može i da se malo smanji ukoliko znamo koliko ćemo najmanje članova imati, tako da možemo da prosledimo tu vrednosti kako se bar do te vrednosti ne bi vršilo kopiranje.

Naravno, ako se popuni niz, svako sledeće kopiranje ne možemo da izbegnemo.

osoblje.**ensureCapacity**(100); // znamo da ce ih najmanje biti 100

```
ArrayList<Zaposleni> osoblje = new  ArrayList<>(100);  // isto to
```

Ovo nije isto što i

```
new Zaposleni[100];
```

Prvo zato što ovaj niz na početku ima sto zaposlenih, dok prvi niz na početku nema ni jednog zaposlenog sve dok ga ne dodamo sa add.

Drugo, kod prvog niza se može smestiti i više od 100 zaposlenih.

Ukoliko znate da se niz neće više povećavati možete pozvati **trimToSize** kako bi se oslobodila memorija koja nije upotrebljena.

```
osoblje.trimToSize(50);
```

Pristup članovima je malo komplikovaniji nego u običnim nizovima. Članovima se ne može pristupiti pomoću [i], već se pristupa pomoću get i set metode.

osoblje.**set**(i, hari); // a[i] = hari; za neki niz a

Nemojte pozivati list.set(i, x) ukoliko niste sigurni da je lista veća od i.

Element se dobija sa

osoblje.**get**(i); // isto sto i e = a[i];

Metode add i set kod netipizirane ArrayList prihvataju objekte bilo kog tipa, stoga treba voditi računa ako treba kastovati objekat prilikom dohvatanja. Ako se koristi ArrayList<T>, kompajler će već sam prijaviti grešku ukoliko tipovi nisu saglasni.

Ukoliko je to ono što vam treba možete iskoristiti prednosti oba, prilagodljivu veličinu i uobičajeni pristup. Na primer, napravimo listu da dodamo elemente, a zatim konvertujemo u niz i koristimo kao niz.

```
ArrayList<X> list = new ArrayList<>();
while(. . .) list.add(new X(. . .));
X[] a = new X[list.size()];
list.toArray(a);
```

Ukoliko želite da umetnete element na poziciju n koristite

```
int n = staff.size()/2;
osoblje.add(n, e);
// elementi na poziciji >=n su pomereni za jedno mesto
```

Na sličan način, može se izbrisati element

```
Zaposleni z = osoblje.remove(n);
```

Može se koristiti for-each petlja

```
for(Zaposleni z: osoblje)  radi nesto;
```

Isto je što i

```
for(int i=0; i<osoblje.size(); i++) {
   Zaposleni z = osoblje.get(i);
   radi nesto;
}
```

Obmotači

Klase obmotači imaju imena

Integer, Long, Float, Double, Short, Byte, Character, i Boolean.

Svima je nadklasa Number.

Ove klase su nepromenljive, odnosno ne može se promeniti njihova vrednosti jednom kada se naprave. Takođe, ne mogu ni da se proširuju s obzirom da su deklarisane kao final.

Ukoliko želimo listu integera, moramo da koristimo obmotače, jer lista ne prihvata proste tipove ArrayList<int>.

ArrayList<Integer> lista = new ArrayList<>();

Ovakva lista integera je prilično neefikasna u odnosu na int[].

Poziv lista.add(3) je ekvivaletno sa lista.add(Integer.valueof(3));

Ovo se naziva **autoboxing**.

Simetrično, kada se integer vrednosti dodeli Integer objekat, vrši se automatski unboxing

int n = lista.get(i) je isto sto i
int n = lista.get(i).intValue();

Integer n = 3; n++; // automatsko otp. i pak.

U većini slučajeva se ima utisak da su primitivni tipovi i obmotači jedno te isto, ali to nije slučaj, što se može videti i prilikom provere identičnosti

Integer a = 1000;
Integer b = 1000;
if(a == b) . . . ???

Specifikacija autoboxing zahteva da boolean, byte, char<=127, short, i int izmedju -128 i 127 budu zapakovani tako da iste vrednosti budu u jednom objektu. Za druge vrednosti to ne važi.

Moguće je da ima i null vrednost. Tada se ponaša uobičajeno.

```
Integer n = null;
System.out.println(2*n);            // null pointer exception
```

Ukoliko se meša Integer i Double u uslovnom izrazu, onda se Integer raspakuje i zapakuje u Double.

```
Integer n = 1;
Double x = 2.0;
System.out.println( true ? n : x);      // ispisuje 1.0
```

Konverzija stringa u integer

```
int x = Integer.parseInt(s);
```

Šta se dešava sa prosleđivanjem objekta preko metoda? Da li se može izmeniti u metodi sada pošto je objekat u pitanju?

```
public static void triple(int x)                    // ne radi
{ x = 3 * x;          // modifikuje lokalnu promenljivu }

public static void triple(Integer x)                // ne radi
{ . . . }
```

Ukoliko želite da izmenite numeričke parametre, možete koristiti klase IntHolder, BooleanHolder, i slične paketa org.omg.CORBA

```
public static void triple(IntHolder x)
{ x.value = 3 * x.value; }
```

Metode sa promenljivim brojem parametara

Moguće je napraviti metode koje mogu da se pozovu sa promenljivim brojem argumenata. Do sada ste videli jedan takav metod: **printf**.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "stvari");
```

Oba poziva pozivaju isti metod, čak iako jedan poziv ima dva parametra, a drugi tri parametra.

Metod printf je definisan na sledeći način:

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args){
     return format(fmt, args);
   }
}
```

Tri tačkice su deo Java koda. One označavaju da metod može da prihvati proizvoljan broj argumenata pored prvog argumenta fmt. Metod printf zapravo dobija dva parametra, fmt i Object[] niz koji sadrži sve ostale parametre. Ukoliko su prosleđeni primitivni tipovi, prave se obmotači i oni se prosleđuju umesto primitivnih tipova. Kompiler treba da transformiše svaki poziv printf-a tako da parametre smesta u niz, uz autoboxing ukoliko je potrebno.

```
System.out.printf("%d %s", new Object[] {new Integer(n), "stvar"});
```

Definisanje metoda sa promenljivim brojem parametara

```
public static double max(double. . . values)
{
   double largest = Double.NEGATIVE_INFINITY;
   for (double v : values)
      if (v > largest) largest = v;
   return largest;
}
```

Metod se poziva sa:

```
double m = max(3.1, 40.4, -5);
```

Kompiler prosleđuje `new double[] { 3.1, 40.4, -5 }` kao parametar funkciji max.

Enumeracije

Enumerisani tip služi za nabrojive podatke i se definiše na sledeći način

public **enum** Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};

Enumeracije su u stvari klase. Ova klasa može imati samo četiri objekta - nije moguće kreirati nove objekte. Stoga nije potrebno da se koristi equals pri poređenju, dovoljno je koristiti ==.

Sve numeracije nasleđuju klasu **Enum** koja ima brojne metode. Možda najkorisnija je **toString**, koja vraća ime enumerisane konstante.

Size.SMALL.toString() // vraca string "SMALL".

Size s = Enum.**valueOf**(Size.class, "SMALL");
// u drugom smeru postavlja s na Size.SMALL

Sve vrednosti se mogu dobiti pomoću

```
Size[] values = Size.values();
```

Ili redni broj konstante pomoću

```
int rbr = Size.MEDIUM.ordinal();
```

Ukoliko želite možete dodati i druge metode i konstruktor u svoju klasu

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
    private String skraceno;
    private Size(String skraceno) {
        this.skraceno = skraceno;
    }
    public String dajSkraceno() { return skraceno; }
}
```

Preporuke za dizajn nasleđivanja

1. Smestiti zajedničke operacije i polja u natklasu.

2. Ne koristiti protected polja.

3. Koristiti nasleđivanje u modelovanju "is-a" relacije.

4. Ne koristiti nasleđivanje ukoliko svi nasleđeni metodi i polja nemaju smisla.

5. Nemojte da menjate očekivano ponašanje kada predefinišete metod.

6. Koristiti polimorfizam umesto provere tipa.

Zadaci

Interfejsi

Tehnika kojom se opisuje šta klasa treba da radi, ali bez preciziranja kako to treba da radi, zove se **Interfejs**.

```
public interface Pokretni {
   void kreni();
   void stani();
}
```

Sve **metode** interfejsa su automatski **public**.

Sva **polja** su automatski **public static final**.

Klasa može da implementira interfejs, tako što će obezbediti da postoje implementacije metoda navedenih u definiciji interfejsa.

```
public class Vozilo implements Pokretni {
   void kreni(){
     System.out.println("Krecem se. . .");
   }
   void stani(){
     System.out.println("Zaustavio sam se.");
   }
}
```

Može se napraviti novi objekat klase koja implementira interfejs, ali ne i samog interfejsa.

```
Pokretni x;                          // ok
x = new Pokretni(. . .);             // greska
x = new Vozilo(. . .);               // ok
if(x instanceof Pokretni) . . .      // ok
```

Interfejsi mogu da se proširuju

```
public interface Brzi extends Pokretni
{
   double potrosnja();
   double OGRANICENJE_BRZINE = 100;
   // public static final konstanta
}
```

Jedna klasa može da implementira više interfejsa.

class Vozilo **implements** Pokretni, Comparable

```
public interface ZivoBice { void zivi(); }
public interface Zivotinja extends ZivoBice {
    void kreciSe();
}
public interface Biljka extends ZivoBice {
    void vrsiFotosintezu();
}
public interface Vodozemac extends Zivotinja {
    void plivaj();
}

public class Zaba implements Vodozemac {
    ...
}
```

Primer ugrađenog interfejsa - interfejs Comparable

Kada klasa implementira ovaj interfejs to znači da je uporediva.

Svaka klasa koja implementira Comparable interfejs, mora da implementira compareTo metod koji vraća int kojim će se objekti porediti prilikom sortiranja.

```
public interface Comparable
{
   int compareTo(Object other);
}
```

Na primer, **sort** metod klase **Arrays** obećava da će sortirati niz objekata ukoliko objekat implementira **Comparable** interfejs.

```
class Zaposleni implements Comparable
{
   public int compareTo(Object otherObject){
      Zaposleni other = (Zaposleni) otherObject;
      return Double.compare(plata, other.plata);
   }
}
```

Može se suziti skup objekata sa kojima je moguće porediti navođenjem tipa.

```
class Zaposleni implements Comparable<Zaposleni>
{
   public int compareTo(Zaposleni other){
      return Double.compare(plata, other.plata);
   }
}
```

Primer ugrađenog interfejsa - interfejs Comparator

Implementiranjem interfejsa Comparable može se navesti samo jedan način za poređenje i to funkcijom compareTo.

Ukoliko želimo objekte da poredimo na više načina, to se može uraditi pravljenjem više komparatora. Komparator je klasa koja impementira Comparator interfejs.

```
public interface Comparator{
  int compare(Object prvi, Object drugi);
}
```

```
public <T> interface Comparator<T>{
  int compare(T prvi, T drugi);
}
```

Na primer, za poređenje dva stringa obrnuto leksikografski ili po broju slova pravimo dva komparatora.

```
class PorediLeksOpadajuce implements Comparator<String>
{
  public int compare(String prvi, String drugi){
    return drugi.compareTo(prvi);
  }
}
class PorediPoDuzini implements Comparator<String>
{
  public int compare(String prvi, String drugi){
    return prvi.length()-drugi.length();
  }
}
```

```
Comparator<String> comp = new PorediLeksOpadajuce();
if(comp.compare("Prvi", "Drugi") > 0) . . .
```

Za poređenje nizova može se koristiti metod sort klase Arrays.

```
String[] prijatelji = {"Petar", "Maja", "Sanja"};

Arrays.sort(prijatelji, new PorediPoDuzini());
for(String s : prijatelji)
  System.out.println(s + " ");
```

Izlaz:
Maja Petar Sanja

```
Arrays.sort(prijatelji, new PorediLeksOpadajuce());
for(String s : prijatelji)
  System.out.println(s + " ");
```

Izlaz:
Sanja Petar Maja

Kloniranje objekta

Ugrađeni Interfejs **Cloneable** govori da je klasa obezbedila metod kojim se sigurno vrši kloniranje objekta.

Kada se napravi kopija objekta, kao što znamo, napravi se kopija reference na isti objekat. Ukoliko želimo da menjamo objekat, a da stari ostane nepromenjen, treba da napravimo novi objekat. Tada je potrebno da objekat kloniramo vodeći računa da i svi njegovi članovi budu klonirani ukoliko je potrebno.

```
Zaposleni zap = new Zaposleni(. . .);
Zaposleni copy = zap;
copy.povecajPlatu(10); // povecava platu i copy i zap

copy = zap.clone();
copy.povecajPlatu(10); // povecava platu samo copy
```

Međutim, nije sve tako jednostavno.

Metod clone ima modifikator protected što znači da ga može pozvati potklasa. To je zato što klasa Object ne zna kod su tipa polja potklase, pa tako možda ne uradi ispravno kloniranje.

Ukoliko su polja prosti tipovi kopiranje je u redu, pa čak i ako su nepromenljivi referentni tipovi.

Međutim, ako su u pitanju promenljivi tipovi, onda samo kopiranje reference nije dovoljno. Stoga svaki objekat treba za sebe da napravi clone metod.

Opcije koje imamo su

1. Podrazumevani **clone** metod je dovoljan

2. Podrazumevani **clone** metod se može popraviti tako što se poziva clone za polja koja su promenljive klase

3. Metod **clone** se ne upotrebljava

Ukoliko izaberete 1. ili 2. morate implementirati **Cloneable** interfejs i redefinisati **clone** metod tako da ima **public** pristup, da biste mogli na ovaj način da klonirate objekat.

Zadatak

1. a) Napraviti klasu Tacka i dodati težište klasama Kvadrat i Krug. Implementirati klase Kvadrat i Krug tako da, pored interfejsa Comparable, implementiraju i interfejs Cloneable.

b) Predefinisati metod clone tako da pravi kopiju čitavog objekta sa sopstvenim poljima, umesto da pravi samo kopiju referenci na polja.

c) Napraviti klasu za testiranje u kojoj treba kreirati jedan objekat klase Krug, a zatim ga klonirati i smestiti u drugu promenljivu tipa Krug. Ispisati težišta oba kruga.

d) Izmeniti kloniranom objektu koordinate težišta. Ispisati ponovo težišta oba kruga. Ukoliko je kloniranje ispravno izvršeno, težišta treba da se razlikuju.

Izuzeci

Izuzetak je greška koja se javlja prilikom izvršavanja programa, takozvana **runtime** greška.

Ukoliko se izuzetak ne obradi, podrazumevano se prekida program uz ispis steka na kome se nalaze funkcije koje su bile u toku izvršavanja kada se izuzetak pojavio.

Neki od izuzetaka su:

- NullPointerException
- InputMismatchException
- ArrayIndexOutOfBoundsException
- . . .

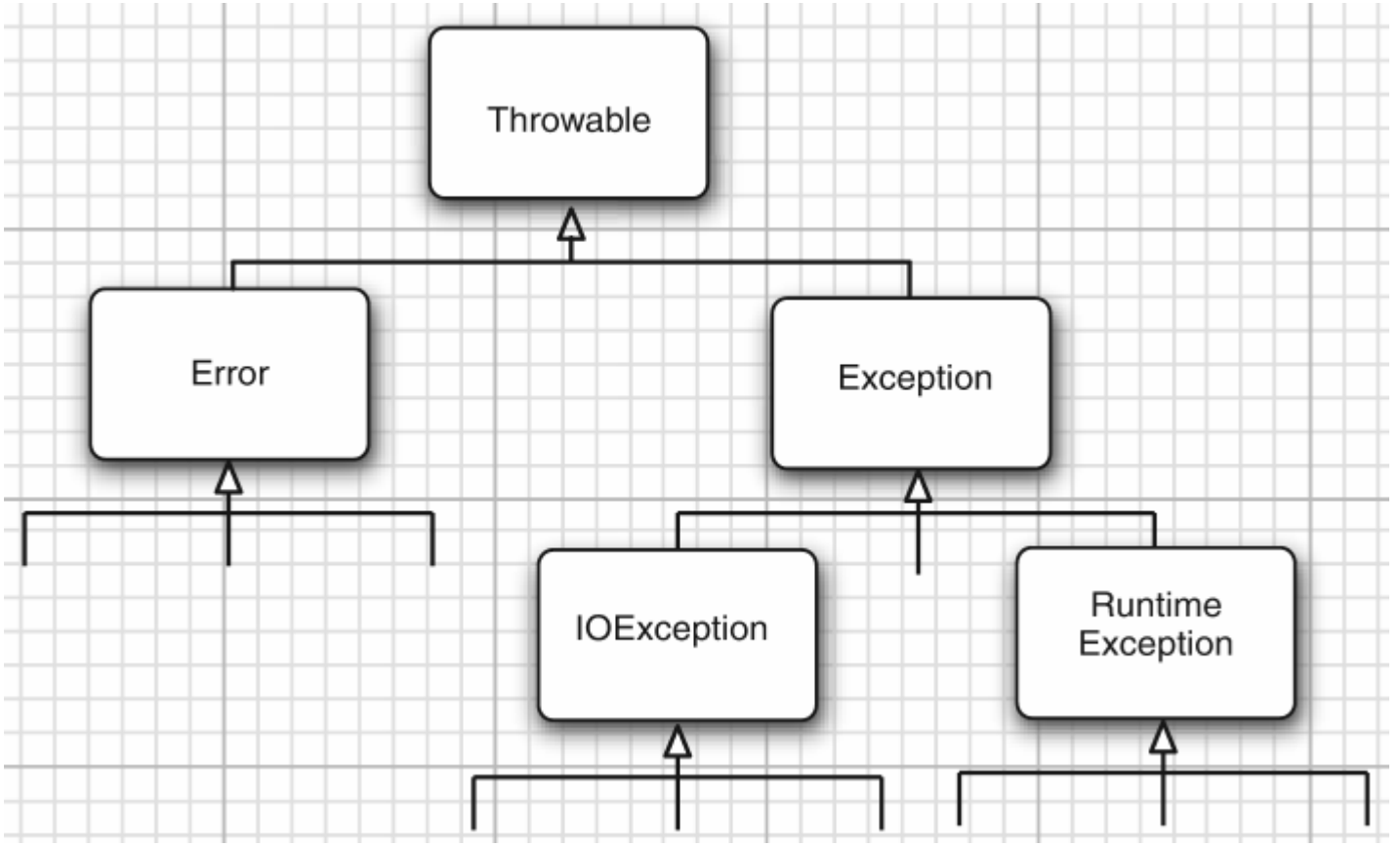
Greške koje uzrokuju izuzetke mogu biti raznolike:

- pogrešan unos (očekivali smo int a unet je string)
- nedostupan uređaj kome se pristupa (isključen je štampač)
- fizička ograničenja (nedostatak memorije prilikom upisa)
- greške u kodu (pristupili smo članu niza koji ne postoji)

Tradicionalno, ukoliko se javi neka greška u izvršavanju metoda, vraća se neka vrednost koja signalizira grešku, na primer, null ili -1, ili neka druga vrednost. Međutim, nije uvek moguće da se povratnom vrednošću signalizira greška. Na primer, ukoliko je null, ili -1, sasvim korektna povratna vrednost.

Java omogućava da se promeni tok izvršavanja metoda kada dođe do greške. Metod tada baca (throws) objekat klase **Throwable**, ili neke njene potklase, koji u sebi sadrži informacije o grešci. Metod prestaje da se izvršava regularnim tokom, već se traži način kako da se rukuje objektom koji je bačen.

Izuzetak je uvek neka klasa koja je izvedena iz natklase Throwable.



Hijerarhija **Error** se odnosi na interne greške u okviru sistema za izvršavanje Jave, virtualne mašine ili Java biblioteke. To su greške koje su retke i gde korisnici malo toga mogu da urade osim da ispišu informaciju i završe program.

Greške kojima možemo da upravljamo su u hijerarhiji **Exceptions**. Ova hijerarhija se deli na *logičke greške* koje nastaju pri izvršavanju programa (takozvane runtime greške, RuntimeException) i one koje nisu te (bez obzira što su i one nastale u toku izvršavanja).

RuntimeException nastaje zbog greške u kodu koja je mogla biti proverena i predupređena:

- pogrešno kastovanje, pristup null referenci, pristup članu niza van granica niza, . . .

Ostale greške nastaju usled loše situacije (okruženja) u kojoj se našao naš kod:

- čitanje posle kraja fajla, otvaranje fajla koji ne postoji, traženje Class objekta na osnovu stringa koji ne predstavlja ime ni jedne klase, . . .

Greške koje proizilaze iz klase Error ili klase RuntimeException nazivamo **neprovereni** (**unchecked**) izuzeci.

Ostale greške nazivamo **provereni** (**checked**) izuzeci.

Na primer, neke greške možemo pretpostaviti. Kao što je ta da fajl možda neće postojati, ili, ukoliko rukujemo fajlom, ta da je moguće da nastane neka greška sa ulazom, odnosno IOException. U zaglavlju metoda se tada navodi izuzetak koji u metodu može da nastane.

public FileInputStream(String name)
throws FileNotFoundException

Ovom deklaracijom se navodi da konstruktor kreira objekat klase FileInputStream na osnovu stringa, ali takođe može da kreira objekat klase FileNotFoundException ukoliko dođe do greške. Ukoliko nastupi drugi slučaj, objekat FileInputStream se ne kreira već se izvršavanje prenosi mehanizmu za obradu izuzetaka.

Izuzetak se pojavljuje ukoliko se desi nešto od sledećeg:

- pozvali ste metod u kome je generisan izuzetak i koji pomoću throws prosleđuje taj izuzetak

- detektovali ste grešku i sami generisali izuzetak pomoću throw

- napravili ste grešku, na primer a[-1] = 0, koja generiše izuzetak koji niste detektovali

- pojavila se interna greška u virtuelnoj mašini ili biblioteci.

Ukoliko se desi nešto od prva dva, treba dati neko obaveštenje korisniku koji poziva vaš metod da izuzetak postoji.

```
class MyAnimation
{
  . . .
  public Image loadImage(String s)  throws IOException
  { . . . }
}
```

Obaveštenje, throws

Nije dobro prosleđivati greške nasleđene od RuntimeException jer je predupreda potpuno u našoj moći, tako da je bolje proveriti u kodu nego generisati izuzetke.

```
class MyAnimation {                          // nije dobra praksa
  . . .
  void drawImage(int i) throws ArrayIndexOutOfBoundsException
  { . . . }
}
```

Nije potrebno prosleđivati interne greške (nasleđene od Error) jer su ionako izvan naše kontrole.

Obaveštenje, throws

Kada u deklaraciji metoda stoji da može baciti izuzetak neke klase, tada on može baciti izuzetak te klase ili bilo koje njegove potklase:

Na primer, ukoliko je navedena IOException, može se očekivati i FileNotFoundException.

Klasa takođe može baciti više vrsti izuzetaka:

```
class MyAnimation {
  . . .
  public Image loadImage(String s)
                                  throws EOFException, FileNotFoundException
  { . . . }
}
```

Eksplicitno generisanje izuzetka, throw

```
String ucitaj(Scanner in) throws EOFException
{
  . . .
  while(. . .){
    if(!in.hasNext())         // EOF je sledeci
      if(n < len)   throw new EOFException();
    . . .
  }
  return s;
}
```

Ukoliko postoji već izuzetak koji odgovara našem problemu, onda koristimo taj.

Pravimo naš izuzetak

ukoliko ne postoji izuzetak koji odgovara našem problemu.

```
class FileFormatException extends IOException
{
  public FileFormatException() {}
  public FileFormatException(String zalba) {
    super(zalba);
  }
  @Override
  public String getMessage(){
    return "Ovo je moja detaljna poruka o izuzetku."
  }
}
```

Generišemo ga na isti način:

```
if(n < len)   throw new FileFormatException();
```

Hvatamo i obrađujemo izuzetak, try-catch

tako što kritično mesto okružujemo try-catch blokom:

```
try{
  kod
  jos koda
} catch (ExceptionType e) {
  obrada ove vrste izuzetka
}
```

Ukoliko bilo koji deo koda u okviru try bloka baci izuzetak, preskače se ostatak try bloka i izvršava catch blok koji odgovara tom izuzetku. Ukoliko je bačen izuzetak drugog tipa, program se prekida kao da nema try bloka. Ukoliko se ne baci izuzetak, program preskače catch.

Ukoliko se pojavi izuzetak koji se ni na jednom mestu ne obradi, program se završava uz ispisivanje poruke na terminalu.

```
// Dve su mogućnosti: hvatanje i obrada izuzetka
public void read(String filename)
{
  try
  {
    InputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1) { obrada ulaza }
  } catch (IOException exception) {
    exception.printStackTrace();
  }
}
// naspram prosleđivanja izuzetka pozivajućoj metodi
public void read(String filename) throws IOException
{
  InputStream in = new FileInputStream(filename);
  int b;
  while ((b = in.read()) != -1) { obrada ulaza }
}
```

Hvatanje i obrada više vrsti izuzetaka

```
try
{
  kod
  jos koda
}
catch (FileNotFoundException | UnknownHostException e)
{
  obrada ove vrste izuzetka
}
catch (IOExceptionc e)
{
  obrada ove vrste izuzetka
}
finally
{
  izvršava se uvek, bilo da je izuzetak
  uhvaćen ili ne
}
```

Preporuke za korišćenje izuzetaka

1. Ne treba koristiti izuzetke za običnu proveru podataka.

2. Kad god je moguće treba obuhvatiti više kritičnih mesta jednim try-catch blokom umesto imati više takvih blokova sa manje naredbi.

3. Koristiti benefite hijerarhije izuzetaka, ne hvatati samo Throwable, već treba izabrati odgovarajući.

4. Ne ućutkivati izuzetke ma koliko delovali dosadno. Nije dobra praksa izbegavanje akcije pomoću catch(. . .) { // ne radi nista }.

5. Ne izbegavati izuzetke kada su potrebni. Na primer, ukoliko je stek prazan bolje je generisati izuzetak umesto vratiti null vrednost i prouzrokovati NullPointerException kasnije u kodu.

6. Propagiranje izuzetka nije znak slabosti. Zapravo, metode višeg nivoa će imati više informacija o okolnostima pod kojima je nastao izuzetak.

Zadaci

Generici

Generičko programiranje

Odnosi se na pisanje koda koji može da se koristi za objekte različitih tipova. Na primer, jedan isti kod može da se koristi i za klase koje rade sa klasom String i sa klasom Integer.

Pre generičkog programiranja klasa ArrayList je mogla da prihvati objekte bilo kog tipa bez provere.

```
public class ArrayList
{
  private Object[] elementi;
  . . .
  public Object get(int i) { . . . }
  public void add(Object o) { . . . }
}
```

Ovaj pristup ima dva problema

Prvo: prilikom dohvatanja vrednosti, neophodno je kastovanje.

```
ArrayList lista = new ArrayList();
. . .
String ime = (String) lista.get(0);
```

Drugo, može se smestiti i objekat drugog tipa, što se prevodi i pokreće bez problema, ali prilikom kastovanja u String rezultuje greškom.

```
lista.add(new Integer( . . . ));
```

Genericima, sa druge strane, mogu se navesti željeni tipovi parametara.

```
ArrayList<String> lista = new ArrayList<String>();
```

Ovime se rešavaju prethodna dva problema.

Sopstvena klasa sa generičkim tipovima

```
public class KljucVrednost<K, V>
{
  private K kljuc;
  private V vrednost;

  public KljucVrednost(K k, V v){
    kljuc = k;
    vrednost = v;
  }

  public K dajKljuc() { return kljuc; }
  . . .
}
```

Deklaracija i kreiranje objekta se izvodi navođenjem konkretnog tipa.

```
KljucVrednost<Integer, String> prom;
prom = new KljucVrednost<Integer, String>(42, "Odg.");
```

Generički metodi

Ukoliko je potrebno imati metod koji radi isto, ali za različite tipove, bez korišćenja generika bi bilo potrebno napisati po jedan metod za svaki tip.

public static void ispisi(**Integer**[] niz) { . . . }

public static void ispisi(**String**[] niz) { . . . }

public static void ispisi(**Ucenik**[] niz) { . . . }

Sa korišćenjem generičkih tipova je dovoljno napisati jedan metod

public static **<T>** void ispisi(**T**[] niz) { . . . }

Metod se poziva na sledeći način

ImeKlase.<**Ucenik**>ispisi(niz);

ImeKlase.<**String**>ispisi(niz);

Uslovljeni generički tipovi

Ukoliko je potrebno da generički tip zadovoljava još neki uslov, na primer da implementira Comparable interfejs, to se navodi na sledeći način

```
static <T extends Comparable<T>> T min3(T a, T b, T c){
  T min = a;
  if(min.compareTo(b) < 0) min = b;
  if(min.compareTo(c) < 0) min = c;
  return min;
}
```

Ukoliko je potrebno postaviti više ograničenja, razdvajaju se znakom **&**:

T extends **Comparable & Cloneable**

Sortiranje liste

```
ArrayList<String> reci = new ArrayList<String>();
. . .
Collections.sort(reci)
// sortira na podrazumevani način
//   leksikografski rastuće
```

Ako imamo **ArrayList<T>** listu, šta mora da ispunjava tip T da bismo mogli da pozovemo sortiranje?

Zadaci

1. Napisati klasu Ucenik koja ima polja broj, ime, prezime i prosečna ocena. Učitati u listu nekoliko učenika. Ispisati na konzoli listu učenika sortiranu prema prosečnoj oceni, pa po prezimenu, pa po imenu, pa po broju. Za sortiranje koristiti metod Collections.sort.

2. Doraditi prethodni zadatak na sledeći način:

Najpre, pokrenuti beskonačnu petlju u kojoj treba detektovati sledeće naredbe:

- Moguće je uneti novog učenika sa konzole unosom:

novi Broj Ime Prezime Prosek

- Moguće je obrisati učenika: obrisi Broj

Zadaci

3. Napisati klasu Skup<T> za rad sa generičkim skupovima podataka. Skup se predstavlja interno tako što ima polje ArrayList<T> vrednosti. U klasi Skup<T> predvideti sledeće metode:

```
void dodaj(T el);
void unija(Skup<T> s);
void presek(Skup<T> s);
void ispisi();
```

Primer izvršavanja:

```
Skup<Integer> s1  = new Skup<Integer>();
s1.dodaj(2);
Skup<Integer> s2 = new Skup<Integer>();
s2.dodaj(2); s2.dodaj(3); s1.unija(s2);
s1.ispisi();          // ispisuje 2 3
```

Kolekcije

Strukture podataka služe sa čuvanje nekih grupa elemenata, takozvanih kolekcija, kao i za efikasno rukovanje njima.

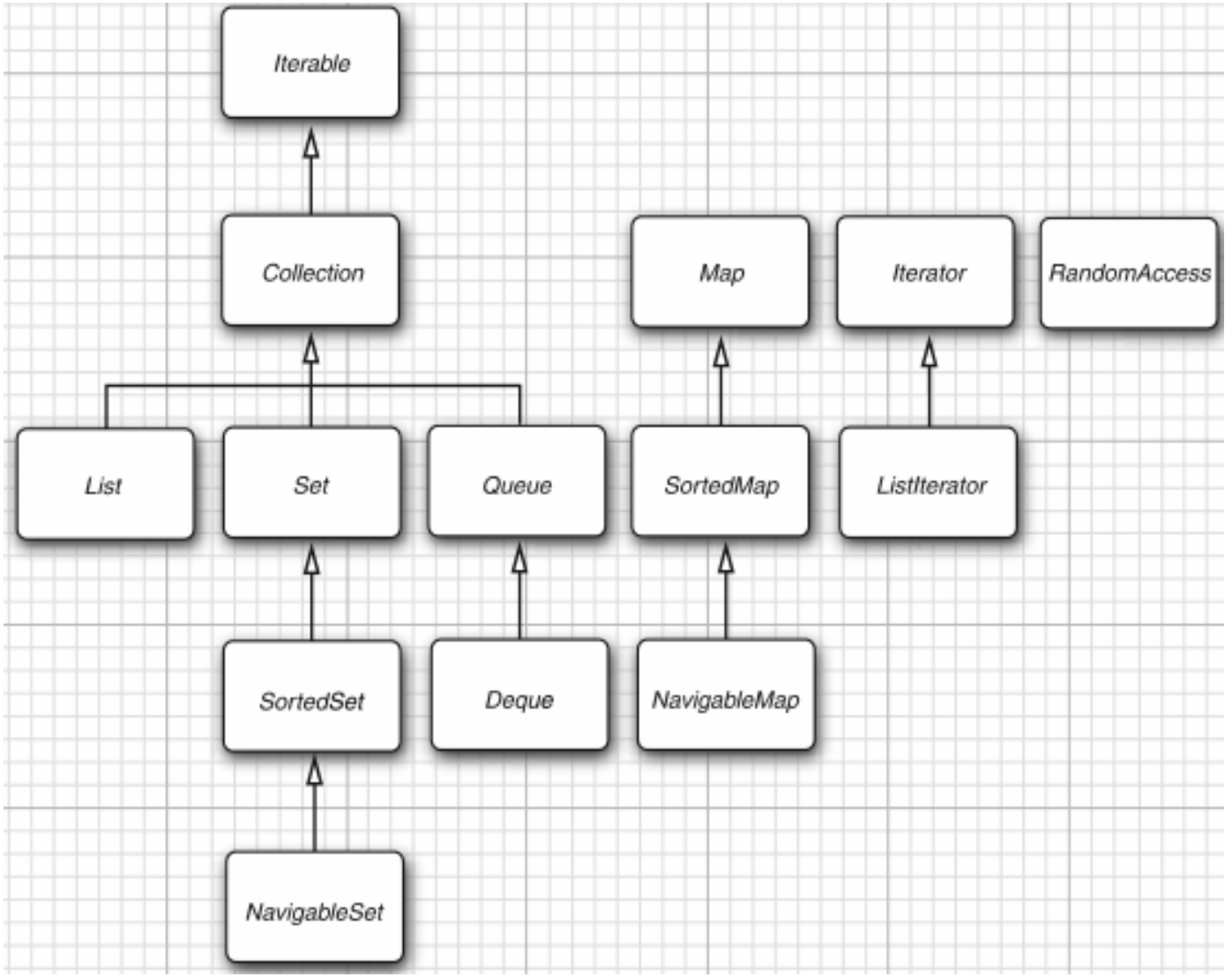
Struktura podataka koju izaberemo za rešavanje nekog problema može napraviti veliku razliku u efikasnosti izvršavanja programa kao i u lakoći odnosno težini obrade problema.

Da li je potrebno brzo pretražiti hiljade ili milione sortiranih elemenata, ili je potrebno učestalo umetati i brisati elemente? Da li je potrebno ustanoviti vezu između ključeva i vrednosti?

Java biblioteka ima ugrađene neke od struktura podataka koje možemo koristiti. Ukoliko vam nijedna struktura od ugrađenih ne odgovara, možete napisati sopstvenu strukturu koja je prilagođena problemu koji rešavate.

Oblast Strukture podataka je veoma široka i stoji ozbiljna teorija stoji iza nje. Mi ćemo ovde diskutovati samo o nekim strukturama implementiranim u Javi.

Interfejsi za rukovanje kolekcijama su:



Collection - grupa elemenata ili kolekcija

List - uređena kolekcija

Set - kolekcija bez ponavljanja elemenata, skup

SortedSet - uređeni skup

NavigableSet - skupovi u kojima se dodatno mogu pretraživati vrednosti po nekom orjentiru

Queue - red sa dodavanjem na kraj i čitanjem sa početka

Deque - red sa dodavanjem i čitanjem sa oba kraja

Map - kolekcija uređenih parova objekata, ključ-vrednost

SortedMap - kolekcija uređenih parova gde su ključevi sortirani

NavigableMap - mape u kojima se dodatno mogu pretraživati vrednosti po nekom orjentiru

Iterator - interfejs za iteraciju kroz kolekciju

ListIterator - interfejs za iteraciju kroz listu

RandomAccess - interfejs koji govori da kolekcija omogućava random pristup elementima

Collection je osnovni interfejs za kolekcije.

Neke od metoda interfejsa Collection:

int	**size**()
boolean	**isEmpty**()
boolean	**contains**(Object obj)
boolean	**containsAll**(Collection c)
boolean	**add**(Object element);
boolean	**addAll**(Collection from)
boolean	**remove**(Object obj)
boolean	**removeAll**(Collection c)
void	**clear**()
boolean	**retainAll**(Collection c)
Object[]	**toArray**()

Operacije koje imaju smisla samo u uređenim kolekcijama:

Collections.**sort**(col);
Collections.**shuffle**(col);
Collections.**reverse**(col);
Collections.**binarySearch**(col, element);

Pogledati i ostale statičke metode klase Collections.

List interfejs

Još neke od ugrađenih metoda:

void	**add**(E e)	- dodaje na kraj
void	**add**(int i, E e)	- dodaje na poziciju, ostali se transliraju
void	**addAll**(Collection c)	- dodaje elem. iz kolekcije
void	**addAll**(int i, Collection c)	- dodaje od pozicije
E	**remove**(int i)	- brise sa pozicije
E	**remove**(Object o)	- brise prvo pojavljivanje
E	**get**(int i)	- cita sa pozicije
E	**set**(int i, E e)	- postavlja na poziciju
int	**indexof**(Object o)	- indeks prvog pojavljivanja
int	**lastIndexOf**(Object o)	- indeks poslednjeg pojavljivanja
boolean	**contains**(Object o)	- da li sadrzi element
int	**size**()	- vraca broj elemenata u listi

List interfejs

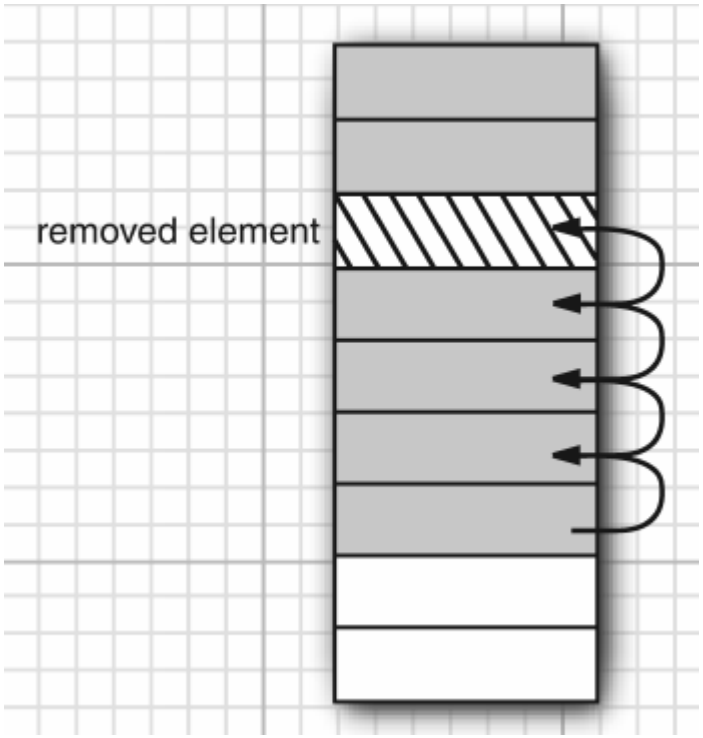
Ugrađene implementacije:

ArrayList - brža, lista ideksirana nizom koja se dinamički proširuje

LinkedList - sporija, uređena sekvenca koja omogućava efikasno umetanje i brisanje sa bilo koje lokacije

ArrayList

Brisanje ili umetanje elemenata je skupo, pošto se svi elementi nakon obrisanog ili umetnutog elementa moraju prekopirati na odgovarajuće lokacije.



ArrayList

ArrayList()

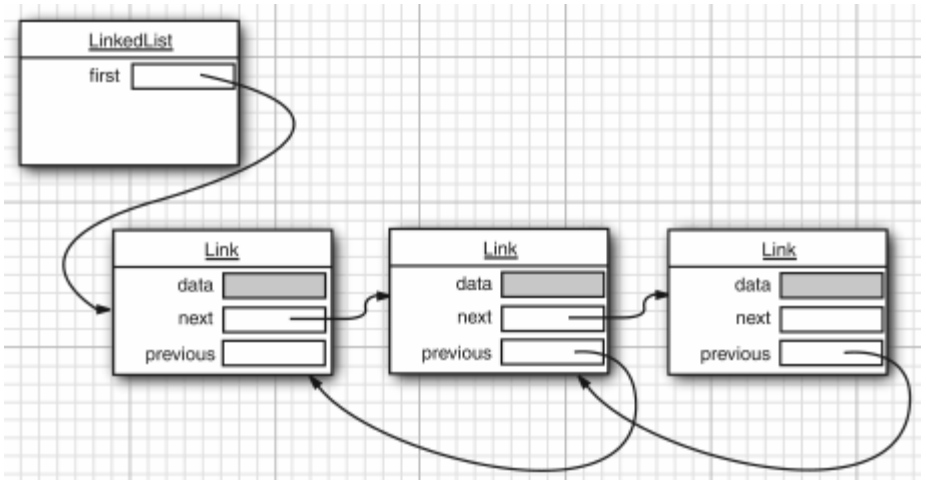
ArrayList(Collection c)

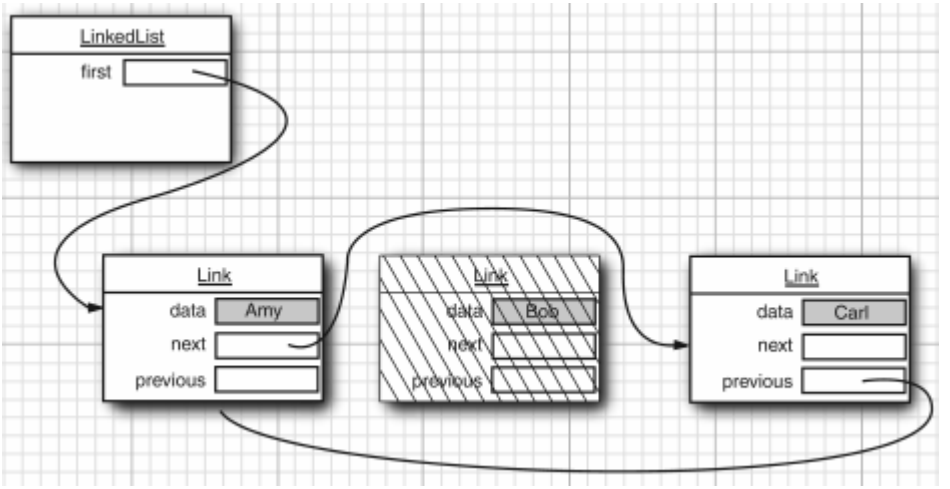
ArrayList(int minCapacity)

Jos neke od metoda klase ArrayList:

void	**ensureCapacity**(int min)	- postavlja kapacitet na min
void	**trimToSize**()	- oslobađa kapacitet koji se ne koristi

LinkedList





Rešava se problem brisanja ili umetanja elemenata.

Sve povezane liste u Javi su dvostruko povezane liste, odnosno svaki element čuva pokazivač na prethodni i na sledeći element.

LinkedList

LinkedList()

LinkedList(Collection c)

Još neke metode klase LinkedList:

void	**addFirst**(E e)	- dodaje na početak
void	**addLast**(E e)	- dodaje na kraj
E	**getFirst**()	- čita sa početka
E	**getLast**()	- čita sa kraja
E	**removeFirst**()	- čita i briše na početka
E	**removeLast**()	- čita i briše sa kraja

Set interfejs

Koristi se za implementaciju grupe elemenata bez ponavljanja, odnosno skupa elemenata.

Još neke od ugrađenih metoda:

s1.**containsAll**(s2) : Ispituje da li s1 sadrži sve iz s2

s1.**addAll**(s2) : unija, dodaje sve iz s2 u s1

s1.**retainAll**(s2) : presek, zadržava u s1 sve koji se nalaze i u s2

s1.**removeAll**(s2) : razlika, uklanja iz s1 sve koji su u s2

Kako biste napisali metodu za simetričnu razliku dva skupa korišćenjem pomenutih metoda?

Simetrična razlika dva skupa je skup koji sadrži sve one elemente koji se nalaze u jednom od skupova, ali nisu zajednički.

Set interfejs

Ugrađene implementacije:

HashSet - najbrži, neuređena kolekcija koja ne dozvoljava duplikate

LinkedHashSet - brz, skup koji pamti redosled unošenja elemenata

TreeSet - sporiji, sortiran skup, uređenje po vrednostima

EnumSet - skup enumerisanih vrednosti

Heš tabela

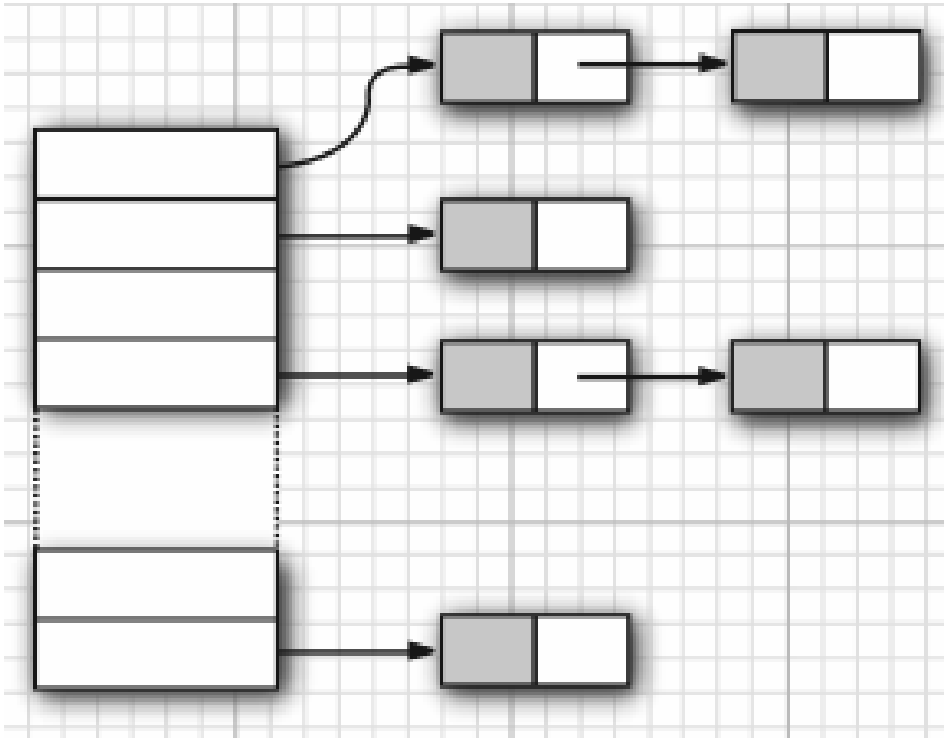
Struktura koja pronalazi elemente veoma brzo je heš tabela. Heš tabela izračunava heš vrednost pomoću **hashCode** metode klase Object, na osnovu stanja objekta. Metodu hashCode smo radili ranije. Ona se može predefinisati tako da odgovara najbolje podacima sa kojima se radi.

Na primer, heš funkcija može biti ostatak po modulu 10.

Tada su za brojeve 42, 53, 67, 23 heš vrednosti redom 2, 3, 7, 3.

Heš tabela

Heš tabela je implementirana kao niz povezanih listi. Ona smešta objekte sa istom heš vrednošću u listu koja se nalazi na poziciji sa tom heš vrednošću. Do liste se dolazi lako, uz pomoć heš vrednosti, ali se kroz listu ipak prolazi linearno. Stoga, treba pametno izabrati heš funkciju, tako da raspodela bude što ravnomernija, odnosno da ima što manje nagomilavanja kod nekih specifičnih heš vrednosti.



HashSet je implementiran heš tabelom

HashSet()

HashSet(Collection elements)

HashSet(int initialCapacity)

HashSet(int initialCapacity, float loadFactor)

Neke od metoda klase HashSet:

void	**add**(E e)	- dodaje element u skup
E	**remove**(E e)	- briše element iz skupa

Heš kod se računa metodom **hashCode**, dok se jednakost računa metodom **equals**.

LinkedHashSet je slična s tim što pamti redosled unošenja elemenata.

Primer 1

Ukoliko se ne menjaju metodi equals i hashCode u klasi Ucenik:

```
Ucenik u = new Ucenik(43, "Petar", "Petrovic", 4.3f);
Set<Ucenik> set = new HashSet<Ucenik>();
set.add(new Ucenik(11, "Marko", "Markovic", 4.0f));
set.add(u);
set.add(u);                 // nece biti unet, equals se
                                    // izracunava po referenci
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));    // bice unet
for(Ucenik i: set)
  System.out.println(i.hashCode() + ": " + i);
```

Izlaz:

```
2018699554: 52      Laza    Lazic    4.5
366712642: 11       Marko   Markovic    4.0
1829164700: 43      Petar   Petrovic     4.3
1311053135: 52      Laza    Lazic    4.5
```

```
// 1. Ispis ne garantuje redosled unošenja
```

Primer 2

Ukoliko se ne menjaju metodi equals i hashCode u klasi Ucenik:

```
Ucenik u = new Ucenik(43, "Petar", "Petrovic", 4.3f);
Set<Ucenik> set = new LinkedHashSet<Ucenik>();
set.add(new Ucenik(11, "Marko", "Markovic", 4.0f));
set.add(u);
set.add(u);                 // nece biti unet, equals se
                                    // izracunava po referenci
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));    // bice unet
for(Ucenik i: set)
  System.out.println(i.hashCode() + ": " + i);
```

Izlaz:

```
366712642: 11      Marko  Markovic   4.0
1829164700: 43     Petar  Petrovic    4.3
2018699554: 52     Laza   Lazic   4.5
1311053135: 52     Laza   Lazic   4.5
// 1. Ispis je u redosledu unosenja
```

Primer 3

Ukoliko se promene metodi equals i hashCode u klasi Ucenik

```
public boolean equals(Object o){
                                        return broj == ((Ucenik) o).broj; }
public int hashCode(){ return broj%10; }
```

```
Ucenik u = new Ucenik(43, "Petar", "Petrovic", 4.3f);
Set<Ucenik> set = new HashSet<Ucenik>();
set.add(new Ucenik(11, "Marko", "Markovic", 4.0f));
set.add(u);
set.add(u);                    // nece biti unet, equals se
                                    // izracunava po broju
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));
set.add(new Ucenik(52, "Laza", "Lazic", 4.5f));
                                  // takodje nece biti unet
for(Ucenik i: set)
  System.out.println(i.hashCode() + ": " + i);
```

Izlaz:

```
1: 11      Marko   Markovic     4.0
2: 52      Laza    Lazic    4.5
3: 43      Petar   Petrovic 4.3        // 1. Nije obavezno sotrirano
```

TreeSet je implementiran stablom

Stablo je struktura koja omogućava da se unos vrši u bilo kom redosledu elemenata, a da se vrednosti postavljaju na pozicije tako da se čitanje izvodi u sortiranom redosledu.

Dodavanje elemenata u stablo je stoga sporije od dodavanja elemenata u heš tabelu. Ali je brža provera duplikata od provere u nizu ili povezanoj listi. Ukoliko stablo sadrži n elemenata, srednja vrednost provera je $\log_2 n$.

TreeSet()

TreeSet(Collection coll)

TreeSet(Comparator comp)

Poređenje se vrši po **compareTo** metodi ili po komparatoru koji je prosleđen.

Primer 1

Ukoliko se definiše compareTo u klasi Ucenik

```
public int compareTo(Ucenik o){
                        return o.prosek.compareTo(prosek); }
```

```
Set<Ucenik> set = new TreeSet<Ucenik>();
set.add(new Ucenik(11, "Marko", "Markovic", 4.0f));
set.add(new Ucenik(43, "Petar", "Petrovic", 4.3f););
set.add(new Ucenik(52, "Laza", "Lazic", 4.0f));
                        // nece biti unet, jer
                        // se jednakost izracunava po proseku
for(Ucenik i: set) System.out.println(i);
```

Izlaz:

```
43  Petar   Petrovic    4.3
11  Marko   Markovic    4.0
```

```
// 1. U skup se ne unose duplikati proseka
// 2. Sortirano je po proseku
```

Primer 2

Ukoliko se prosledi komparator koji poredi po imenu

```
public class PorediPoImenu implements Comparator<Ucenik> {
  public int compare(Ucenik o1, Ucenik o2) {
    return o1.dajIme().compareTo(o2.dajIme());
  }
}

Set<Ucenik> set = new TreeSet<Ucenik>(new PorediPoImenu());
set.add(new Ucenik(61, "Marko", "Markovic", 4.0f));
set.add(new Ucenik(43, "Petar", "Petrovic", 4.0f));
set.add(new Ucenik(52, "Petar", "Lazic", 4.3f));
                                        // nece biti unet, jer se
                                        // poredi komparatorom po imenu
for(Ucenik i: set)
  System.out.println(i);
```

Izlaz:

```
61  Marko  Markovic   4.0
43  Petar  Petrovic   4.0
```

```
// 1. U skup se ne unose duplikati imena
// 2. Sortirano je po imenu
```

Primer 3

```
Set<String> s1 = new HashSet<String>();
s1.add("Milica");
s1.add("Petar");
s1.add("Ivana");
s1.add("Milica");

Set<String> s2 = new TreeSet<String>(s1);
Set<String> s3 = new LinkedHashSet<String>(s1);

for(String s : s1) System.out.println(s);    //šta je ispis?
for(String s : s2) System.out.println(s);    //šta je ispis?
for(String s : s3) System.out.println(s);    //šta je ispis?
```

Queue interfejs

Neke od metoda interfejsa Queue su:

boolean	**add**(E e)	- dodaje element na kraj
E	**remove**()	- skida element sa početka
E	**element**()	- čita element sa početka bez skidanja

* bacaju izuzetak ako nije uspešno izvršena akcija

Deque interfejs (izvodi se iz Queue interfejsa)

Neke od metoda interfejsa Deque su:

void	**push**(E e)	- dodaje element na početak // = addFirst
E	**pop**()	- skida element sa početka // = removeFirst
E	**element**()	- samo čita element sa početka
void	**addFirst**(E e)	- dodaje na početak
void	**addLast**(E e)	- dodaje na kraj // = add
E	**removeFirst**()	- skida element sa početka // = remove
E	**removeLast**()	- skida element sa kraja
E	**getFirst**()	- samo čita element sa početka
E	**getLast**()	- samo čita element sa kraja

* bacaju izuzetak ako nije uspešno izvršena akcija

Queue interfejs

Ugrađene implementacije:

PriorityQueue - Red koja omogućava efikasno brisanje najmanjeg elementa

Deque interfejs (a samim tim i Queue)

Ugrađene implementacije:

ArrayDeque - Red koji omogućava dodavanje i čitanje sa oba kraja

LinkedList -

PriorityQueue

Za skladištenje se koristi struktura Heap. Heap je drvolika struktura koja u opštem slučaju nije sortirana, ali se umetanje i brisanje vrši tako da je u korenom čvoru uvek smešten najmanji (ili najveći) element od svih elemenata koji se nalaze trenutno u njemu.

PriorityQueue

PriorityQueue()

PriorityQueue(int initialCapacity)

PriorityQueue(int initialCapacity, Comparator c)

Neke od metoda:

boolean	**add**(E e)	- dodaje element u heap
E	**remove**()	- čita i skida koreni element

Primer 1

```
PriorityQueue<LocalDate> pq = new PriorityQueue<>();
pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse

while (!pq.isEmpty())
      System.out.println(pq + " -> " + pq.remove());
```

Izlaz:

```
[1815-12-10, 1906-12-09, 1903-12-03, 1910-06-22] -> 1815-12-10
[1903-12-03, 1906-12-09, 1910-06-22] -> 1903-12-03
[1906-12-09, 1910-06-22] -> 1906-12-09
[1910-06-22] -> 1910-06-22
```

ArrayDeque

ArrayDeque()

ArrayDeque(Collection c)

ArrayDeque(int initialCapacity)

Neke od metoda:

Videti metode interfejsa Deque.

Map interfejs

Osnovne operacije:

V	**get**(Object key)
V	**getOrDefault**(Object key, V defaultValue)
V	**put**(K key, V value)
V	**putIfAbsent**(K key, V value)
void	**putAll**(Map<? extends K, ? extends V> entries)
V	**remove**(Object key)
boolean	**containsKey**(Object key)
boolean	**containsValue**(Object value)
Set<K>	**keySet**()
Collection<V>	**values**()

Map interfejs

Ugrađene implementacije:

HashMap - brža, struktura koja omogućava odnos ključ-vrednost

LinkedHashMap - mapa u kojoj se pamti redosled unošenja elemenata

TreeMap - sporija, mapa u kojoj su ključevi sortirani

EnumMap - mapa u kojoj ključevi pripadaju enumerisanom tipu

WeakHashMap - mapa u kojoj garbage collector može brisati elemente koji se ne koriste više

IdentityHashMap - mapa u kojoj se ključevi porede sa == umesto sa equals

Ponašanje je slično kao kod različitih implementacija skupova.

Primer 1

```
Map<String,Ucenik> registar = new HashMap<String, Ucenik>();
registar.put("nalog1", new Ucenik("Petar", "Petrovic"));
registar.put("nalog2", new Osoba("Mika", "Markovic"));
. . .
Ucenik u1 = registar.get("nepostojeci"); //vraća null
Ucenik u2 = registar.get("nalog2");        // vraća Miku

Map<String, Ucenik> registar2 = new TreeMap<String, Ucenik>();
registar2.putAll(registar);
registar.keySet();          //vraća skup ključeva
registar.values();          //vraća kolekciju vrednosti
```

Zadatak 1

Korisnik unosi sa konzole spisak imena sve dok ne unese reč KRAJ.

A. Ispisati na konzoli spisak imena sortiran leksikografski

B. Ispisati na konzoli spisak imena sortiran prema rastućem broju slova (proslediti Comparator u Collections.sort)

C. Ispisati na konzoli leksikografski sortiran spisak imena bez ponavljanja (skup)

D. Ispisati na konzoli skup imena sa brojem pojavljivanja

Ulaz: Ana Petar Mirko Petar Darko Nikola Ana Petar KRAJ

Izlaz:

A: Ana Ana Darko Mirko Nikola Petar Petar Petar

B: Ana Ana Mirko Petar Petar Petar Darko Nikola

C: Ana Darko Mirko Nikola Petar

D: Ana 2, Darko 1, Mirko 1, Nikola 1, Petar 3

Zadatak 2

Napisati program koji računa frekvencije reči u tekstu datom u datoteci tekst.txt. Pritom podrazumevati da su dve reči iste bez obzira na veličinu slova koja se koriste. Takođe, pre računanja frekvencija, eliminisati tačke, zareze itd.

Ukoliko je sadržaj datoteke tekst.txt:

Očekivani izlaz bi bio:

Iterator interfejs

koristi se za prolazak (iteriranje) unapred kroz kolekciju.

Metode:

E	**next**()	- vraća sledeći element, pomera iterator za jedno mesto u kolekciji
boolean	**hasNext**()	- da li ima još elemenata
void	**remove**()	- briše element koji stoji pre iteratora

Primer 1 (Prolazak kroz celu kolekciju)

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // radi nesto sa elementom
}
```

Ili konciznije

```
for (String element : c) {
  // radi nesto sa elementom
}
```

Primer 2 (Brisanje elemenata)

```
it.next(); it.remove(); it.remove();          // greska
it.next(); it.remove(); it.next(); it.remove();       // ok
```

Primer 3 (Dodavanje tri elementa u listu i brisanje drugog elementa)

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next(); // vraca prvi element
String second = iter.next(); // vraca drugi element
iter.remove(); // brise poslednji procitani element
```

ListIterator interfejs (nasleđuje Iterator interfejs)

koristi se za prolazak (iteriranje) u oba smera kroz kolekciju.

Metode:

E	**next**()	- vraća sledeći element, pomera iterator za jedno mesto unapred u kolekciji
E	**previous**()	- vraća prethodni element, pomera iterator za jedno mesto unazad kolekciji
boolean	**hasNext**()	- da li ima elemenata posle
boolean	**hasPrevious**()	- da li ima elemenata pre
void	**add**(E e)	- dodaje element u listu odmah ispred elementa koji bi bio vraćen sledećim pozivom next
void	**set**(E e)	- postavlja na mesto poslednjeg pročitanog elementa pomoću next ili previous
void	**remove**()	- briše element koji je poslednji vraćen preko next ili previous

Primer 4

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");

ListIterator<String> iter = staff.listIterator();
iter.next();
iter.add("Juliet");

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

Izlaz:

[Amy, Juliet, Bob, Carl]

RandomAccess interfejs

Služi da označi da je podržan brz pristup bilo kom članu kolekcije bez obzira na njegovu poziciju. Glavna svrha je da se omogući odgovarajućim algoritmima da prilagode svoje ponašanje kada se primenjuju nad listama sa slučajnim (random, direktnim) pristupom (ArrayList) i nad listama sa sekvencijalnim pristupom (LinkedList).

Na primer, najbolji algoritam koji koristi direktan pristup, može biti veoma neefikasan primenjen nad listom sa sekvencijalnom pristupom. Stoga, pre poziva algoritma može se pozvati instanceof da se proveri mogućnost pristupa.

Zadaci