Objektno Orijentisano Programiranje 1

Objektno orijentisano programiranje i Java

Paketi

- Način za hijerarhijsko organizovanje programa u module
- Implicitni paket – kada ne definišemo paket
- Upotreba

```
import java.io.*;
import java.util.ArrayList;
```

Paketi

- Kreiranje paketa

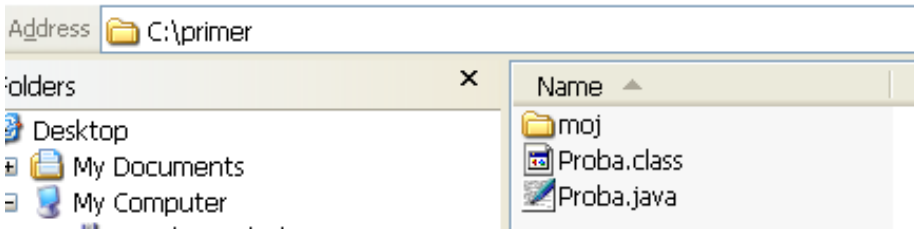
```
package imepaketa;
...
public class MojaKlasa { ... }
```

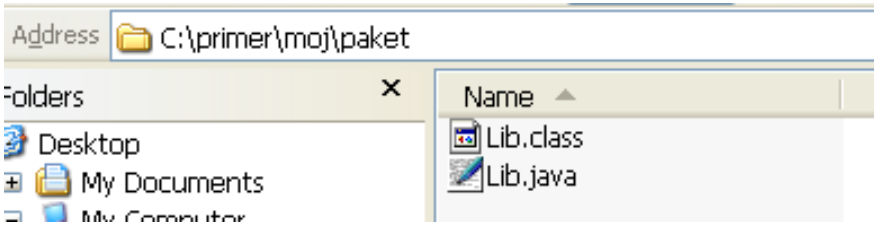
- Korišćenje paketa

```
import imepaketa.MojaKlasa;
...
MojaKlasa m = new MojaKlasa();
          ili
imepaketa.MojaKlasa m = new imepaketa.MojaKlasa();
```

Paketi

- Direktorijumi
  - hijerarhija paketa se poklapa sa hijerarhijom direktorijuma:
    - moj.paket.Lib -> moj\paket\Lib.class





TestPaketa.java

Paketi

- **`CLASSPATH`** environment varijabla:
  - predstavlja spisak foldera i JAR arhiva gde VM traži klasu koja se koristi.
  - ako **`CLASSPATH`** ne postoji, podrazumevaju se tekući direktorijum i standardne Java biblioteke.
  - ako **`CLASSPATH`** postoji, mora da sadrži i tekući direktorijum (standardne Java biblioteke se podrazumevaju). Primer:

```
CLASSPATH=.;C:\Java\lib;C:\Java\bibl.jar
```

Paketi

- JAR arhive
  - klasičan ZIP format
  - sadrži i folder **`META-INF`** u kojem je najbitnija datoteka **`manifest.mf`**
    - sadržaj **`manifest.mf`** datoteke:
      - Manifest-Version: 1.0
      - Created-By: 1.4.2_02 (Sun Microsystems Inc.)
      - Main-Class: moj.paket.Klasa
      - Class-Path: lib/biblioteka.jar lib/druga_biblioteka.jar

javadoc

- Specijalni komentari u izvornom kodu
- Automatsko generisanje programske dokumentacije
- HTML format na izlazu
- Kompletna dokumentacija Jave je generisana javadoc alatom.
  - Lokacija: %JAVA_HOME%\docs

Klasa **`Object`**

- Sve Java klase implicitno nasleđuju klasu Object
- Reprezentativne metode:
  - equals(o)
  - toString()
  - hashCode()
  - getClass()

Tacka.java

Wrapper klase

- Za sve primitivne tipove postoje odgovarajuće klase:
  - int → Integer
  - long → Long
  - boolean → Boolean
- Imaju korisnu statičku metodu Xxxx.parseXxxx()
  - int i = Integer.parseInt(“10”) ← vraća primitivni tip int
  - long l = Long.parseLong(“10”)
- Imaju korisnu statičku metodu Xxxx.valueOf()
  - Integer i = Integer.valueOf("123") ← vraća objekat klase Integer

Autoboxing i unboxing

- Autoboxing/unboxing:
  - ako metoda prima Integer kao parametar, može da se prosledi i int, odn. promenljivoj tipa Integer može da se dodeli vrednost promenljive tipa int
  - radi i u obrnutom pravcu – promenljivoj tipa int može da se dodeli vrednost promenljive tipa Integer

```
void f1(Integer i) {
  System.out.println(i);
}
void test() {
  int i = 10;
  f1(i);
}
```

Autoboxing i unboxing

- Najčešće se koristi kod kolekcija:

```
int i = 10;
ArrayList<Integer> lista = new
  ArrayList<Integer>();
lista.add(i);
int j = lista.get(0);
```

Metode sa promenljivim brojem parametara i objekti

- Ako su parametri reference na objekte neke klase, a znajući da sve klase nasleđuju klasu Object, dovoljno je definisati da metoda prima različit broj parametara tipa Object:

```
void f2(Object... params) {
  System.out.println("Parametri su:");
  for (Object p : params)
    System.out.println(p);
}
```

Metode sa promenljivim brojem parametara i objekti

- Poziv takve metode:

```
void test() {
   double x1 = 3.14;
   int x2 = 2;
   String x3 = "abc";
   f2(x1, x2, x3);
}
```

Enumeracije

- Nabrojivi tipovi podataka (celobrojni)
- Primer:

```
enum Size {SMALL, MEDIUM, LARGE,
  EXTRA_LARGE};
Size s = Size.MEDIUM;
enum Days {MON, TUE, WEN, THU, FRI,
  SAT, SUN};
Days d = Days.MON;
```

Enumeracije i objekti

- Java donosi dodatne osobine enumeracijama.
- Enumeracije više ne predstavljaju samo celobrojne nabrojive tipove, već mogu da imaju konstruktore, atribute i metode

Enumeracije

```
enum Dani {
   PON (1),
   UTO (2),
   SRE (3),
   CET (4),
   PET (5),
   SUB (6),
   NED (7);
   // atribut
   int dan;
```

Enumeracije

```
  Dani (int d) {
    dan = d;
  }
  boolean radni() {
    // if (dan == 6 || dan == 7)
    // može i ovako
    if (this == SUB || this == NED)
      return false;
    else
      return true;
  }
}
```

Enumeracije

- Korišćenje:

```
public static void main(String[] args) {
  Dani d = Dani.UTO;
  System.out.println("Dan je: " + d);
  System.out.println(d + " je radan: " +
                     d.radni());
  d = Dani.SUB;
  System.out.println("Dan je: " + d);
  System.out.println(d + " je radan: " +
                     d.radni());
}
```

Enumeracije

- Spisak vrednosti enumeracije se dobija pozivom metode ***values***, koju nije potrebno implementirati:

```
System.out.println("Dani u
  nedelji su:");
for (Dani dani : Dani.values())
   System.out.println(dani);
```

Enumeracije

- Numerička vrednost same enumeracije se dobija metodom ***ordinal***:

```
System.out.println("Celobrojna
  vrednost enumeracije Dani.SUB je:" +
      Dani.SUB.ordinal());
```

- Enumeracije imaju metodu ***name*** koja vraća naziv enumeracije:

```
System.out.println(Dani.SUB.name());
```

Enumeracije

- Enumeracije imaju statičku metodu ***valueOf*** koja vraća enumeraciju na osnovu stringa:

```
Dani d = Dani.valueOf("UTO");
System.out.println(d);
```

Klasa `String`

- Niz karaktera je podržan klasom String. String **nije** samo niz karaktera – on je klasa!
- Objekti klase String se ne mogu menjati (*immutable*)!
- Reprezentativne metode:
  - str.length()
  - str.charAt(i)
  - str.indexOf(s)
  - str.substring(a,b), str.substring(a)
  - str.equals(s), str. equalsIgnoreCase(s) – **ne koristiti ==**
  - str.startsWith(s), endsWith(s)

Stringovi.java

String literal

- Izraz pod dvostrukim navodnicima: "tekst " u Java programu predstavlja objekat klase String čija je vrednost inicijalizovana na dati tekst. Zbog toga je sasvim ispravno pisati:

```
String x = "tekst";
```

- što ima isti efekat (ali je bolje od):

```
String x = new String("tekst");
```

Klasa `String`

```
class StringTest {

  public static void main(String args[]) {
    String s1 = "Ovo je";
    String s2 = "je string";
    System.out.println(s1.substring(2));
// karakter na zadatoj poziciji
    System.out.println(s2.charAt(3));
// poređenje po jednakosti
    System.out.println(s1.equals(s2));
// pozicija zadatog podstringa
    System.out.println(s1.indexOf("je"));
// dužina stringa
    System.out.println(s2.length());
// skidanje whitespace-ova sa poč. i kraja
    System.out.println(s1.trim());
// provera da li string počinje podstringom
    System.out.println(s2.startsWith("je"));

  }
}
```

Ispis na konzoli:

```
o je
s
false
4
9
Ovo je
true
```

Redefinisan + operator sa stringovima

- Ako je jedan od operanada klase String, ceo izraz je string!

```
String a = “Vrednost i je: “ + i;
```

- Metoda **toString()**:

```
Automobil a = new Automobil();
String s = “Vrednost a:“ + a;
  // .. + a.toString();
...
JComboBox cmb = new JComboBox();
cmb.addItem(a);
```

Oprez sa operatorom + i stringovima!

- Problem:

```
String s = "";
for (int i = 0; i < 100000; i++)
  s += "a";
```

- Rešenje:

```
StringBuilder sb = new StringBuilder(100000);
sb.append("");
for (i = 0; i < 100000; i++)
  sb.append("a");
String res = sb.toString();
```

TestString.java

Implicitan poziv toString

- Metoda koja kao parametar ima referencu na objekat klase String može u svom pozivu da primi i objekat neke druge klase
- Pri tome će kompajler automatski generisati kôd koji poziva metodu toString() datog objekta, a zatim taj rezultat proslediti pozvanoj metodi.

Implicitan poziv toString

- Primer:

```
public void handleMessage(String
  message) { ... }
```

- i njen poziv:

```
handleMessage(new Automobil());
```

- Ovaj poziv se na nivou bajt-koda prevodi u:

```
handleMessage(new Automobil().toString());
```

Prenos stringa kao parametra metode

- Stringovi nisu samo nizovi karaktera, te se oni ne mogu menjati iz metoda (kao što to nizovi mogu)
- Primer:

```
public void handleMessage(String message) {
message += "xxx";
}
...
String s = "pera";
handleMessage(s);
```

Stringovi podržavaju regularne izraze

- Ovaj primer razdvaja string s na reči koje su ograničene jednim razmakom:

```
String[] rez = s.split(" ");
```

- Ako želimo da razdvojimo string s na reči koje su ograničene jednim ili više razmaka, onda koristimo sledeći regularni izraz:

```
String[] rez = s.split(" +");
```

Stringovi podržavaju regularne izraze

- Sledeći primer identifikuje dinarski iznos i razdvaja string na delove pre i posle dinarskog iznosa:

```
String s = "Obim trgovine je bio 10000,00 dinara.";
String[] rez = s.split("[0-9]+,[0-9]{2}");
```

- Detaljnije na:

https://docs.oracle.com/javase/9/docs/api//java/util/regex/Pattern.html

Metoda split() klase **`String`**

- "cepa" osnovni string na niz stringova po zadatom šablonu
  - originalni string se ne menja
  - parametar je regularni izraz
- Poziv: String[] rez = s.split(“regex”);
- Primer:

```
String s = "ja sam svetski mega car";
String[] rez = s.split(" ");
```

Metoda split() klase **`String`**

```
class SplitTest {
  public static void main(String args[]) {
    String text = "Ovo je probni tekst";
    String[] tokens = text.split(" ");
    for (int i = 0; i < tokens.length; i++)
      System.out.println(tokens[i]);
    }
  }
}
```

Metoda matches() klase **`String`**

- Vraća true ako je string u skladu sa regularnim izrazom
- Primer:

```
String s = "001-AB";
boolean rez = s.matches("\\d{3}-[A-Z]{2}");
```

Kolekcije

- Nizovi imaju jednu manu – kada se jednom naprave nije moguće promeniti veličinu.
- Kolekcije rešavaju taj problem.
- Zajedničke metode:
  - dodavanje elemenata,
  - uklanjanje elemenata,
  - iteriranje kroz kolekciju elemenata

Kolekcije

Implementacija / Koncept	Hash table	Resizable Array	Balanced Tree	Linked List	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Tipizirane kolekcije - Generics

- Tipizirane kolekcije omogućavaju smeštaj samo jednog tipa podatka u kolekciju.
- Tipizirane kolekcije se tumače kao: „kolekcija Stringova“ ili „kolekcija double brojeva“, i sl.
- Primer:

```
ArrayList<String> kolekcija1 = new ArrayList<String>();
kolekcija1.add("tekst");
String s = kolekcija1.get(0);

ArrayList<Double> kolekcija2 = new ArrayList<Double>();
kolekcija2.add(3.14);
double d = kolekcija2.get(0);
```

foreach kroz nizove i kolekcije

- Omogućuje prolazak kroz niz ili kolekciju.
- Opšta sintaksa:

```
for (varijabla : niz_ili_kolekcija) {
   ... // telo
}
```

- Primer:

```
for (int i : niz) {
  System.out.println(i);
}

ArrayList<String> kolekcija = new ArrayList<String>();
kolekcija.add("tekst1");
kolekcija.add("tekst2");
kolekcija.add("tekst3");
for (String s : kolekcija) {
  System.out.println(s.length());
}
```

Klasa **`ArrayList`**

- Predstavlja kolekciju, odn. dinamički niz
- Elementi se u ArrayList dodaju metodom add()
- Elementi se iz ArrayList uklanjaju metodom remove()
- Elementi se iz ArrayList dobijaju (ne uklanjaju se, već se samo čitaju) metodom get()

Klasa **ArrayList**

```
ArrayList<Integer> lista = new
ArrayList<Integer>();
lista.add(5);
lista.add(10);
lista.add(1, 15);
System.out.println("Velicina je: " +
lista.size());
lista.remove(0);
int broj = lista.get(0);
System.out.println(broj);
System.out.println("Velicina je: " +
lista.size());
```

Tipizirana klasa **`ArrayList`**

```
import java.util.ArrayList;
class ArrayListTest {
  public static void main(String args[]) {
    ArrayList<String> v = new ArrayList<String>();
    v.add("Ovo");
    v.add("je");
    v.add("probni");
    v.add("tekst");
    for (int i = 0; i < v.size(); i++)
      System.out.println(v.get(i));
  }
}
```

Asocijativne mape

- Memorijske strukture koje omogućuju brzu pretragu sadržaja po ključu
- Element se ubacuje u paru sa svojim ključem, koji mora da bude jedinstven

Klasa **`HashMap`**

- Predstavlja asocijativnu mapu
- U HashMap se stavljaju dva podatka:
  - ključ po kojem će se pretraživati
  - vrednost koja se skladišti u HashMap i koja se pretražuje po ključu
- Metodom put() se ključ i vrednost smeštaju u HashMap
- Metodom get() se na osnovu ključa dobavlja (samo čita) vrednost iz HashMap
  - ako se ne nađe ključ, vratiće null

Tipizirana klasa **HashMap**

```
import java.util.HashMap;
public class HashMapTest {
  public static void main(String args[]) {
    HashMap<String, String> ht =
      new HashMap<String, String>();
    ht.put("E10020", "Marko Markovic");
    ht.put("E10045", "Petar Petrovic");
    ht.put("E10093", "Jovan Jovanovic");
    String indeks = "E10045";
    System.out.println("Student sa brojem indeksa " +
                       indeks + " je " + ht.get(indeks));
    indeks = "E10093";
    System.out.println("Student sa brojem indeksa " +
                       indeks + " je " + ht.get(indeks));
}
```

Klase **`BigInteger`** i **`BigDecimal`**

- Koriste se za brojeve sa proizvoljnim brojem cifara.
- Primer:

```
BigInteger a = BigInteger.valueOf(10);
BigInteger b = BigInteger.valueOf(20);
BigInteger c = a.multiply(b);
```

Interfejs Cloneable i klasa Object

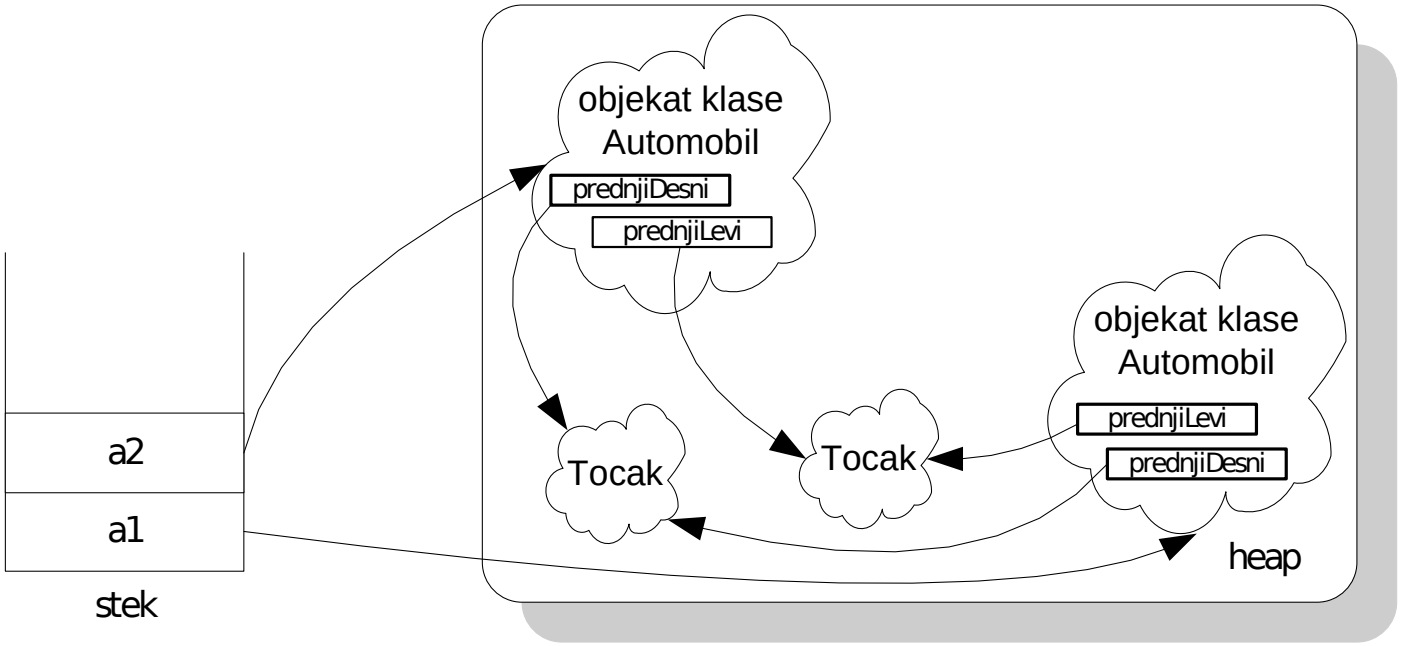
- Java podržava *shallow copy* tako što u klasi Object ima metodu ***clone***, ali je ona protected, tako da nije vidljiva iz drugih klasa
  - ova metoda pravi identičnu kopiju objekta u memoriji (bajt-po-bajt)
- Da bismo je koristili, potrebno je da implementiramo interfejs Cloneable, i redefinišemo metodu ***clone*** da bude public, ako želimo da je pozivaju druge klase.
- Pošto ova metoda pravi bajt-po-bajt kopiju originalnog objekta, njegovi primitivni atributi su iskopirani deep copy načinom
  - problem su neprimitivni atributi!

Interfejs Cloneable

```
class Automobil implements Cloneable {
Tocak prednjiLevi, prednjiDesni, zadnjiLevi, zadnjiDesni;
  Automobil() {
    prednjiLevi = new Tocak();
    prednjiDesni = new Tocak();
    zadnjiLevi = new Tocak();
    zadnjiDesni = new Tocak();
  }
  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
      throw new Error("Ovo ne bi smelo da se dogodi.");
    }
  }
```

Interfejs Cloneable

```
Automobil a1 = new Automobil();
Automobil a2 = (Automobil)a1.clone();
```



Interfejs Cloneable – deep copy

- Rešenje za deep copy upotrebom *Cloneable* interfejsa je da se u metodi *clone* svakako napravi binarna kopija originalnog objekta, ali i
  - da se zatim ručno pozove *clone* od svakog neprimitivnog atributa
    - da se za svaku klasu neprimitivnog atributa napravi *clone* metoda, koja radi isto
      - ako ima neprimitivnih atributa, da i za njih pozove *clone*, i da za njihove klase napravi *clone*

Interfejs Cloneable – deep copy

```
class Automobil implements Cloneable {
  Tocak prednjiLevi, prednjiDesni, zadnjiLevi, zadnjiDesni;
  Automobil() {
    ...
  }
  public Object clone() {
    try {
      Automobil ret = (Automobil)super.clone();
      ret.prednjiLevi = (Tocak)this.prednjiLevi.clone();
      ret.prednjiDesni = (Tocak)this.prednjiDesni.clone();
      ...
      return ret;
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
      throw new Error("Ovo ne bi smelo da se dogodi.");
    }
  }
```

Interfejs Cloneable – deep copy

```
Automobil a1 = new Automobil();
Automobil a2 = (Automobil)a1.clone();
```