# Programski jezik Java

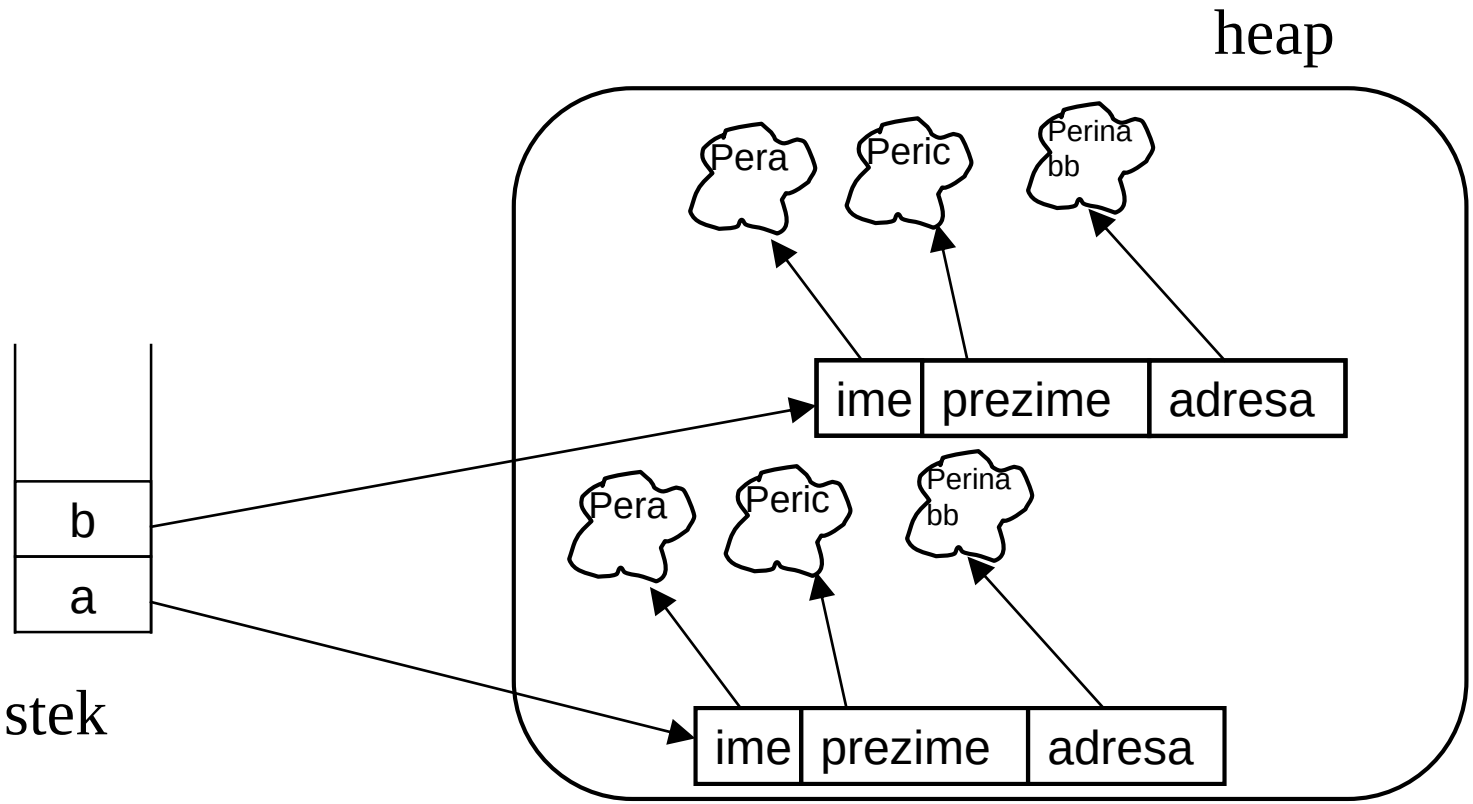
Objektno orijentisano programiranje

# Deep vs. shallow copy

- Ako operatorom '=' zapravo prenosimo vrenost reference, kako onda da napravimo kopiju nekog objekta?
  - ako napravimo metodu:
    **`Automobil deepCopy() {...}`**

  koja će praviti kopiju objekta, onda moramo da za svaki atribut uradimo deep copy
  - operator '=' kod primitivnih tipova radi deep copy
    - kod referenci ne radi deep copy objekta, već kopira reference

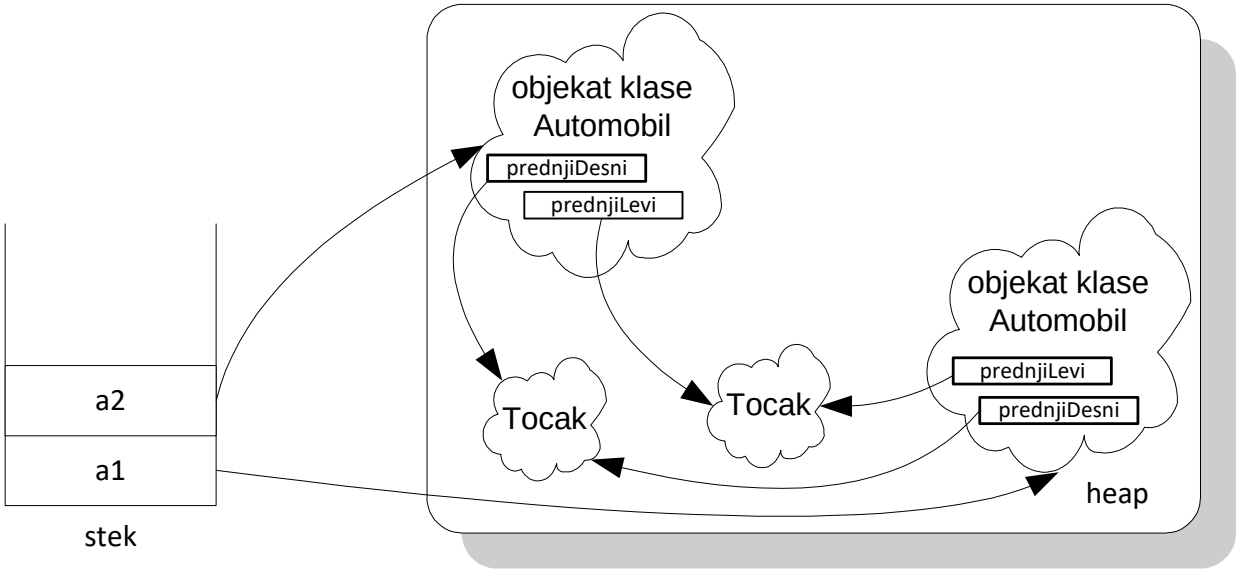
# Deep vs. shallow copy




```
Osoba a = new Osoba("Pera", "Peric", "Perina bb");
Osoba b = a.deepCopy();
```

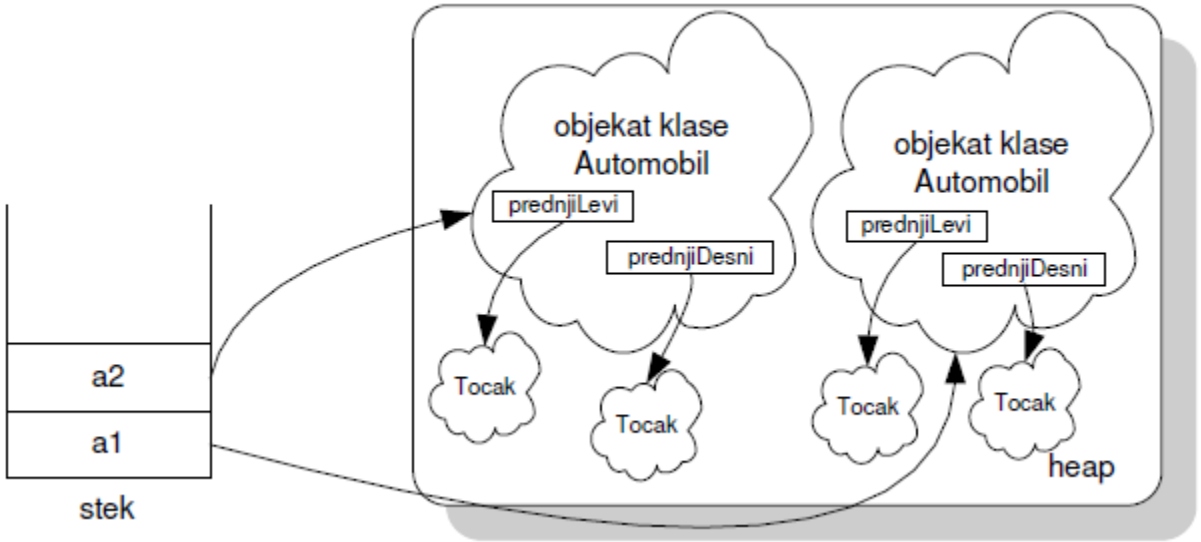
# Plitka kopija

```
Automobil shallowCopy () {
   Automobil ret = new Automobil();
   ret.prednjiLevi = this.prednjiLevi;
   ret.prednjiDesni = this.prednjiDesni;
   ret.zadnjiLevi = this.zadnjiLevi;
   ret.zadnjiDesni = this.zadnjiDesni;
}
```

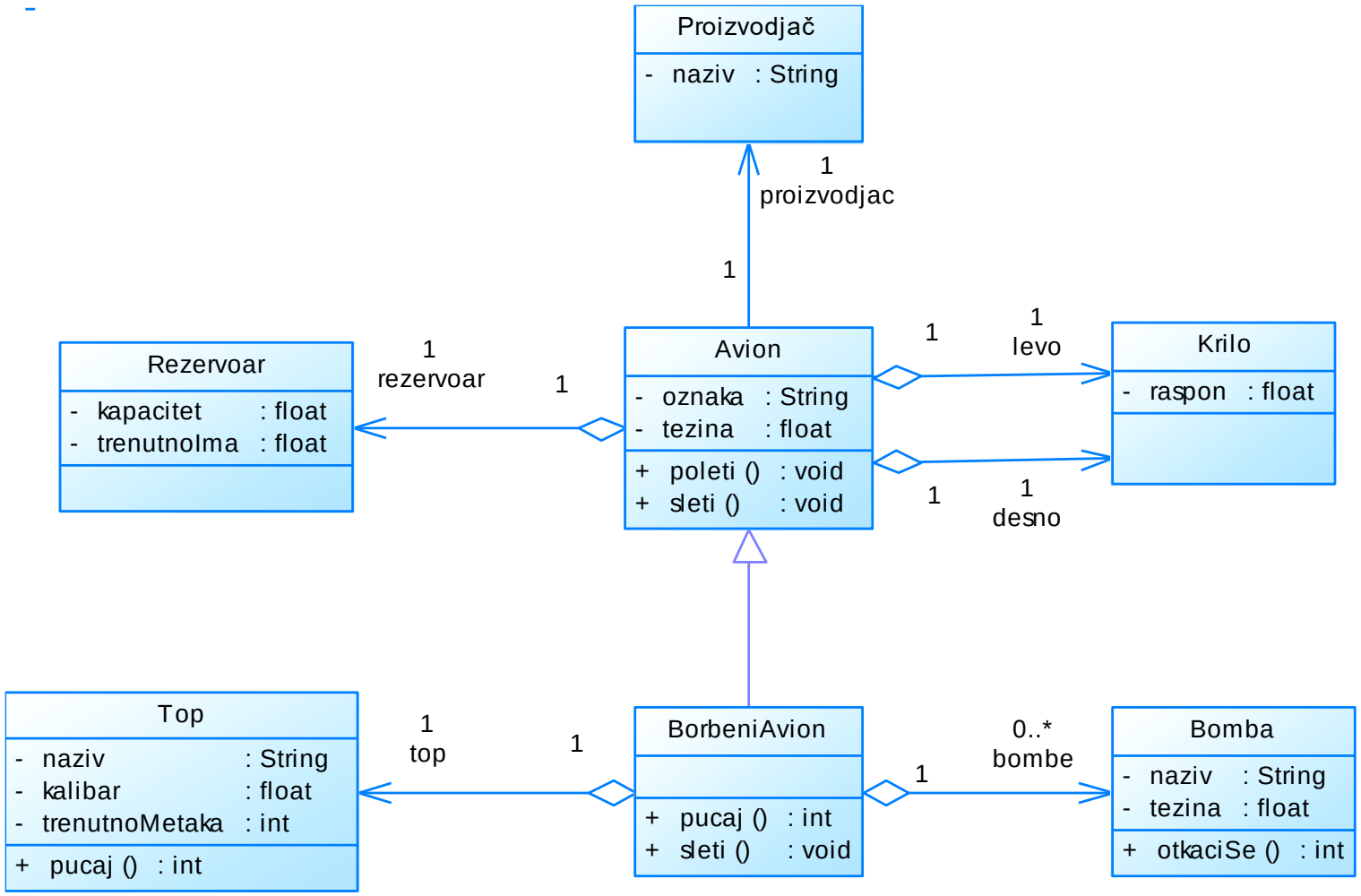


# Duboka kopija

```
Automobil deepCopy () {
  Automobil ret = new Automobil();
  ret.prednjiLevi = this.prednjiLevi.deepCopy();
  ret.prednjiDesni = this.prednjiDesni.deepCopy();
  ret.zadnjiLevi = this.zadnjiLevi.deepCopy();
  ret.zadnjiDesni = this.zadnjiDesni.deepCopy();
}
```



# Nasleđivanje



# Nasleđivanje

```
class Avion {
  String oznaka;
  float tezina;
  Rezervoar rezervoar;
  Krilo levo, desno;
  Proizvodjac proizvodjac;
  void poleti() { ... }
  void sleti() { ... }
}

class BorbeniAvion extends Avion {
  Top top;
  ArrayList<Bomba> bombe;
  void sleti() { ... }
  int pucaj() { ... }
}
```

- postoji samo jednostruko nasleđivanje

# Nasleđivanje

```
class item{
   int base;
     public item()
   {
      base = 10;
   }
}
```

# Nasleđivanje

```
public class sword extends item{
  int modifier;
  public sword(){
     modifier = 3;
  }
  public String toString(){
     Integer z = modifier+base;
     return z.toString();
  }
}
```

# Nasleđivanje

```
public class test
{
    public static void main(String [] args)
    {
        sword cSword = new sword();
        System.out.print(cSword);
    }
}
```

# Calculation

```
class Calculation{
   int z;

   public void addition(int x, int y){
      z = x+y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void subtraction(int x,int y){
      z = x-y;
      System.out.println("The difference between the given numbers:"+z);
   }

}
```

# My_Calculation

```
public class My_Calculation extends Calculation{

   public void multiplication(int x, int y){
      z = x*y;
      System.out.println("The product of the given numbers:"+z);
   }

   public static void main(String args[]){
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.subtraction(a, b);
      demo.multiplication(a, b);
   }

}
```

# Veze tipa asocijacije i agregacije (UML i Java)

- Veza tipa asocijacije je za atribute koji nisu isključivi deo glavne klase (klasa Proizvođač je u vezi tipa asocijacije sa klasom Avion), već mogu da postoje i nezavisno od glavne klase
- Veza tipa agregacije je za atribute koji su deo celine (Rezervoar ← Avion, Krilo ← Avion) i nema smisla da postoje nezavisno od glavne klase
- Kardinalnost veze određuje da li će atribut biti promenljiva ili kolekcija (niz, ArrayLista i sl.)

# Modifikatori pristupa

- **`public`** – vidljiv za sve klase
- **`protected`** – vidljiv samo za klase naslednice i klase iz istog paketa
- **`private`** – vidljiv samo unutar svoje klase
- nespecificiran (*friendly*) – vidljiv samo za klase iz istog paketa

# getters & setters

- Ponekad je potrebno obezbediti kontrolisan pristup atributima, kako za čitanje, tako i za pisanje.
- To se postiže posanjem odgovarajućih metoda kroz koje se pristupa atributima:

```
public class Student {
  private String ime;
  public String getIme() {
    return ime;
  }
  public void setIme(String ime) {
    this.ime = ime;
  }
}
```

# getters & setters

- Ova kombinacija atributa i njegovog getter-a i setter-a se još zove i svojstvo (*property*).
- Ovim je omogućeno da se čitanje vrednosti svojstva sprovodi kroz njegov getter, a izmena kroz setter.
- Ako izostavimo setter, dobijamo *read only* svojstvo.

# Method overriding

- Pojava da u klasi naslednici postoji metoda istog imena i parametara kao i u baznoj klasi
- Anotacija @Override
- Primer:
  - klasa A ima metodu **metoda1()**
  - klasa B nasleđuje klasu A i takođe ima metodu **metoda1()**

# Method overriding

```
class A {
   int metoda1() {
     System.out.println("metoda1 klase A");
   }
   int metoda2() {
     System.out.println("metoda2 klase A");
   }
}
class B extends A {
   int metoda1() {
     System.out.println("metoda1 klase B");
   }
}
...
A varA = new A();
B varB = new B();
varA.metoda1();
varB.metoda1();
varA.metoda2();
varB.metoda2();
```

# Method overriding

- na konzoli će pisati

```
metoda1 klase A
metoda1 klase B
metoda2 klase A
metoda2 klase A
```

# Ključna reč ***super***

- Ključna reč ***super*** označava roditeljsku klasu. Ona se može koristiti i u metodama i u konstruktorima:

```
class BorbeniAvion extends Avion {
  Top top;
  Bomba[] bombe;
  @Override
  void sleti() {
    System.out.println("BorbeniAvion odbacuje bombe.");
    System.out.println("BorbeniAvion slece.");
    super.sleti();
  }
  void pucaj() { ... }
}
```

# Klučna reč ***super*** u konstruktoru

- Ključna reč super u konstruktoru označava da pozivamo konstruktor roditeljske klase i tada se mora napisati na samom početku konstruktora klase naslednice:

```
public BorbeniAvion() {
   super();
   System.out.println("Konstruktor borbenog
aviona.");
   top = new Top();
   bombe = new Bomba[] { new Bomba(),
                         new Bomba()};
}
```

# Apstraktne klase

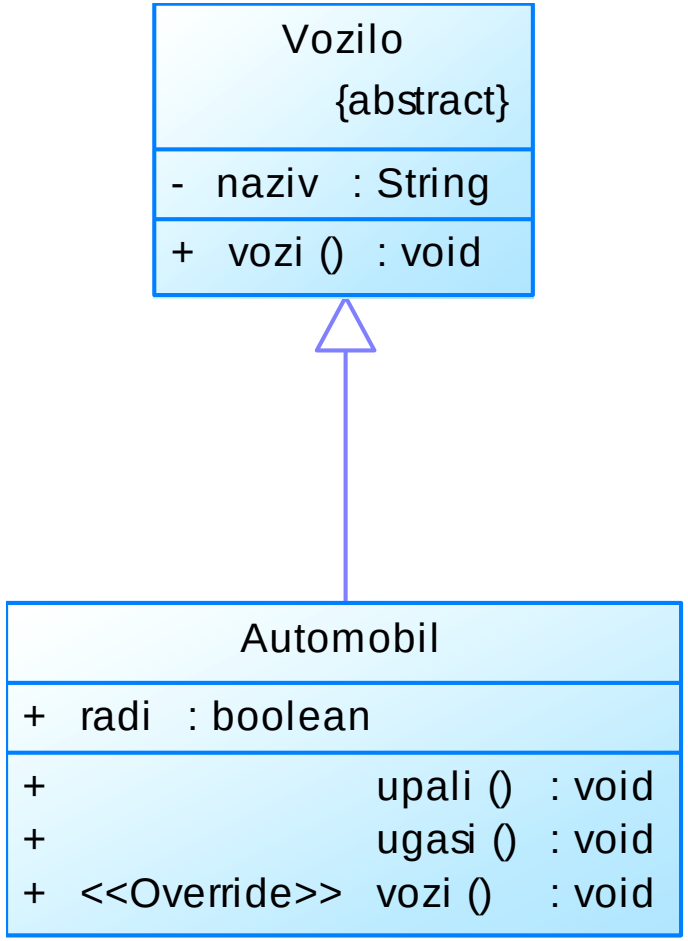
- klase koje ne mogu imati svoje objekte, već samo njene klase naslednice mogu da imaju objekte (ako i one nisu apstraktne)

```
abstract class A {
  int i;
  public void metoda1() { ... }
  public abstract void metoda2();
  ...
}

class B extends A {
  public void metoda2() { ... }
}
```

- Ako klasa ima makar jednu apstraktnu metodu, mora da se deklariše kao apstraktna.
- Apstraktna klasa ne mora da ima apstraktne metode!

# Apstraktne klase

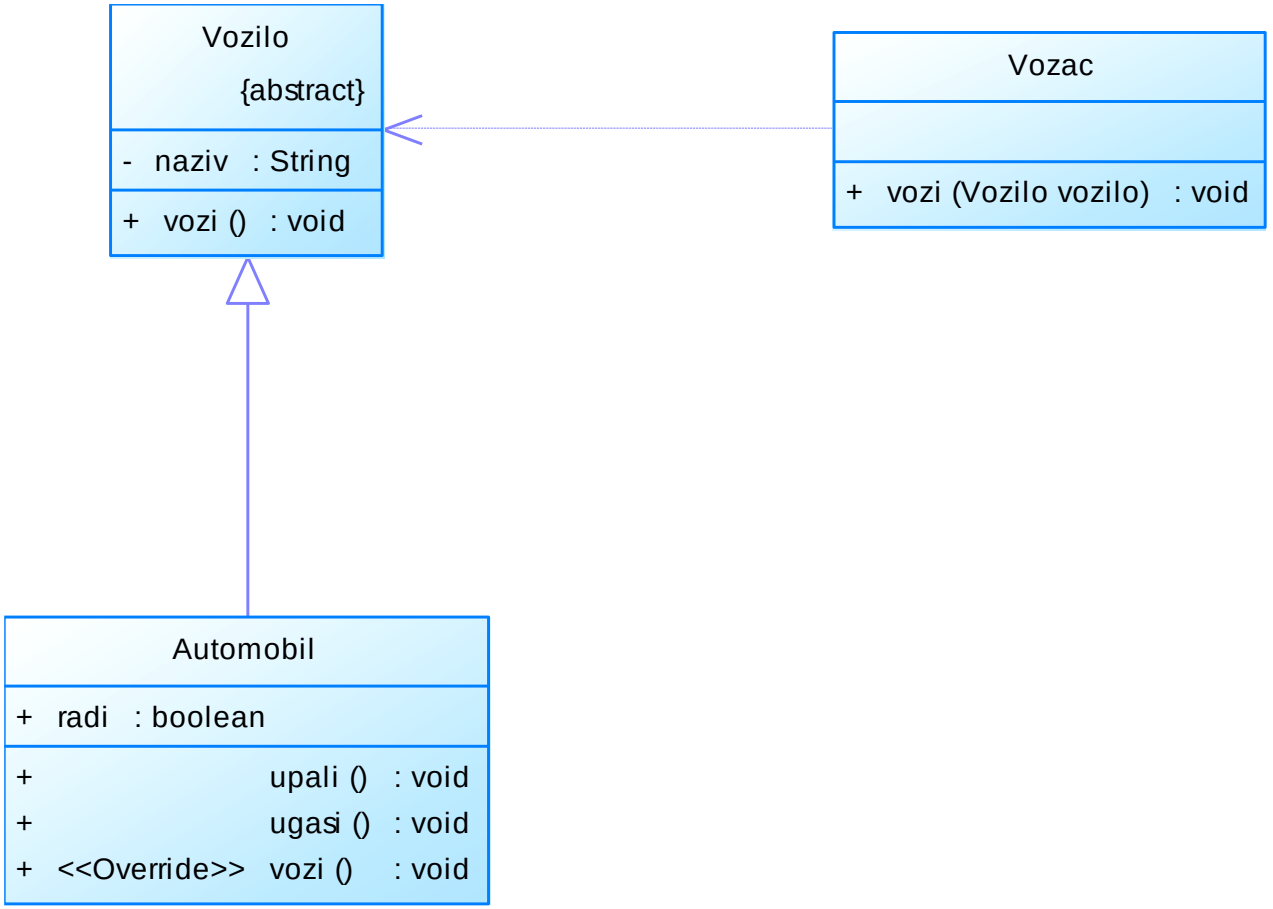
```
public abstract class Vozilo {
  private String naziv;
  public abstract void vozi();
}

public class Automobil extends Vozilo {
  public boolean radi;
  public void upali() {
    radi = true;
  }
  public void ugasi() {
    radi = false;
  }
  public void vozi() {
    ...
  }
}
```



# Polimorfizam

- Situacija kada se poziva metoda nekog objekta, a ne zna se unapred kakav je to konkretan objekat
  - ono što se zna je koja mu je bazna klasa
- Tada je moguće u programu pozivati metode bazne klase, a da se zapravo pozivaju metode konkretne klase koja nasleđuje baznu klasu

# Polimorfizam



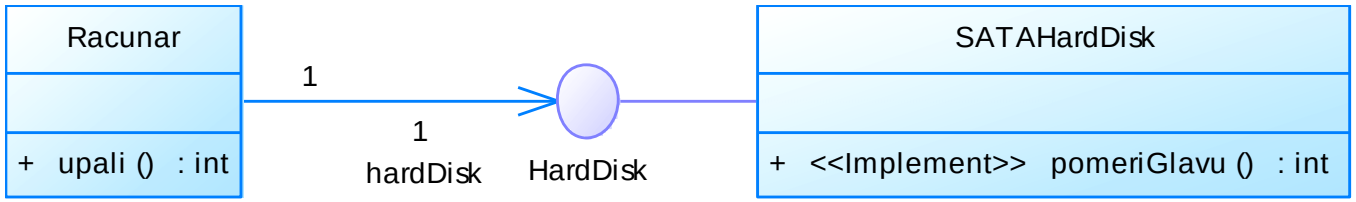
# Polimorfizam

```
abstract class Vozilo {
  abstract void vozi();
}
class Automobil extends Vozilo {
  void vozi() { ... }
}
class Vozac {
  void vozi(Vozilo v) {
    v.vozi();
  }
}
...
Vozac v = new Vozac();
v.vozi(new Automobil());
```

# Interfejsi

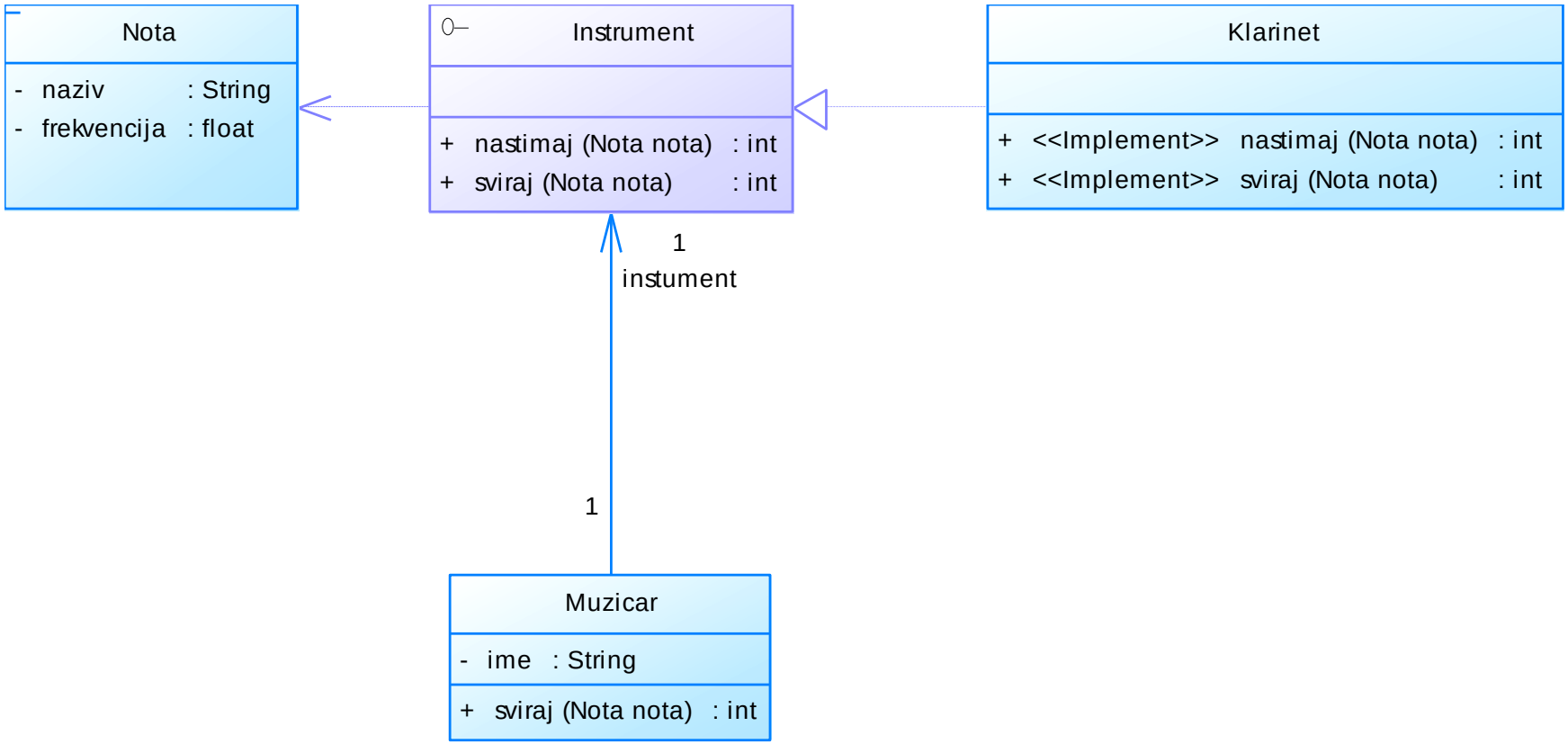
- Omogućavaju definisanje samo apstraktnih metoda, konstanti i statičkih atributa
- Interfejs nije klasa! On je spisak metoda i atributa koje klasa koja implementira interfejs mora da poseduje.
- Sve metode su implicitno public, a svi atributi su implicitno public static final.
- Interfejsi se ne nasleđuju, već implementiraju
- Da bi klasa implementirala interfejs, mora da redefiniše sve njegove metode
- Jedan interfejs može da nasledi drugog
- **Jedna klasa može da implementira jedan ili više interfejsa**

# Interfejsi




```
public class Racunar {
  public HardDisk hardDisk;
  public int upali() {
  }
}
public interface HardDisk {
  int pomeriGlavu();
}
public class SATAHardDisk implements HardDisk {
  public int pomeriGlavu() {
    ...
  }
}
```

# Interfejsi


Nota
- naziv : String
- frekvencija : float
Instrument
+ nastimaj (Nota nota) : int
+ sviraj (Nota nota) : int
Klarinet
+ <<Implement>> nastimaj (Nota nota) : int
+ <<Implement>> sviraj (Nota nota) : int
1
instument
1
Muzicar
- ime : String
+ sviraj (Nota nota) : int

# Interfejsi

```
interface Instrument {
   int sviraj(Nota nota);
   int nastimaj(Nota nota);
}
class Klarinet implements Instrument {
   public int sviraj(Nota nota) { ... }
   public int nastimaj(Nota nota) { ... }
}
class Muzicar {
  Instrument instrument;
  int sviraj(Nota nota) {
    return instrument.sviraj(nota);
  }
}
...
Muzicar m = new Muzicar();
m.instrument = new Klarinet();
m.sviraj(nota);
```

# Interfejsi

```
interface USB {
   void init();
   byte[] getData();
}
interface Camera {
   void init();
   Picture getPicture();
}
```

# Interfejsi

```
class USBKeyboard implements USB {
   @Override
   void init() { ... }
   @Override
   byte[] getData() { ... }
}
```

# Interfejsi

```
class WebCam implements USB, Camera {
  @Override
  void init() { ... }
  @Override
  byte[] getData() { ... }
  @Override
  Picture getPicture() { ... }
}
```

# Inner classes (unutrašnje klase)

```
class Spoljasnja {
  Spoljasnja() { ... }
  void metoda() { ... }

  class Unutrasnja {
    void metoda() { ... }
  }
}
```

# Inner classes (unutrašnje klase)

- Konstrukcija objekta unutrašnje klase izvan spoljašnje klase

```
Spoljasnja sp = new Spoljasnja();
Spoljasnja.Unutrasnja un = sp.new Unutrasnja();
```

# Statičke unutrašnje klase

```
class Spoljasnja {
  void metoda() { ... }
  static class UnutrasnjaStatic {
    int metoda2() { ... }
  }
}
...
Spoljasnja.UnutrasnjaStatic u =
new Spoljasnja.UnutrasnjaStatic();
```

# Izuzeci

- Mehanizam prijavljivanja greške
- Greška se signalizira "bacanjem" izuzetka
- Metoda koja poziva potencijalno "grešnu" metodu "hvata" izuzetak

# Izuzeci

- dve vrste izuzetaka:
  - checked (Exception) – moraju da se uhvate
    - **`EOFException`**
    - **`SQLException`**
    - `...`
  - unchecked (RuntimeException) – ne moraju da se uhvate
    - **`NullPointerException`**
    - **`ArrayIndexOutOfBoundsException`**
    - **`ArithmeticException`**
    - `...`

# Izuzeci

```
try {
  // kod koji može da izazove
  // izuzetak
}
catch (java.io.EOFException ex) {
  System.out.println("Kraj datoteke pre vremena!");
}
catch (ArrayIndexOutOfBoundsException ex) {
  System.out.println("Pristup van granica niza");
}
catch (Exception ex) {
  System.out.println("Svi ostali izuzeci");
}
finally {
  // kod koji se izvršava u svakom slučaju
}
```

# Izuzeci

- programsko izazivanje izuzetka

```
throw new Exception("Ovo je jedan izuzetak");
```

- korisnički definisani izuzeci

```
class MojException extends Exception {
  MojException(String s) {
    super(s);
  }
}
```

# Izuzeci

- ključna reč **`throws`**

```
void f(int i) throws MojException { ... }
```

- propagacija izuzetaka

# Paketi

- način za hijerarhijsko organizovanje programa u module
- implicitni paket
- upotreba

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

# Paketi

- kreiranje paketa

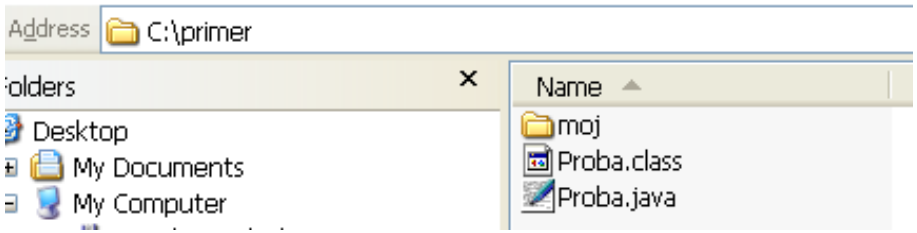
```
package imePaketa;
...
public class MojaKlasa { ... }
```

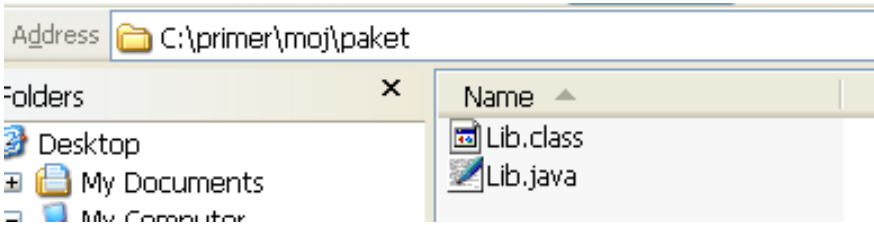
- korišćenje paketa

```
import imePaketa.MojaKlasa;
...
MojaKlasa m = new MojaKlasa();

imePaketa.MojaKlasa m = new imePaketa.MojaKlasa();
```

# Paketi

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

# 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: biblioteka.jar 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%\doc

# Odabrane klase

# Klasa `Object`

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

# Ključna reč *instanceof*

- Vraća *true* ako je levi operand zadatog tipa (desni operand)
- Primer:

```
if (s instanceof Automobil)
  System.out.println("s je automobil");
```

# 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)

# 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()`**

# 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**

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

# 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.print(v.get(i) + " ");
  }
}
```

# 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

# Klasa **HashMap**

```
import java.util.HashMap;
public class HashMapTest {
  public static void main(String args[]) {
    HashMap ht = new HashMap();
    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 " +
(String)ht.get(indeks));
    indeks = "E10093";
    System.out.println("Student sa brojem indeksa " +
                       indeks + " je " +
(String)ht.get(indeks));
}
```

# 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));
}
```

# Wrapper klase i autoboxing

- Za sve primitivne tipove postoje odgovarajuće klase:
  - int → Integer
  - long → Long
  - boolean → Boolean
- Imaju korisnu statičku metodu Xxxx.parseXxxx()
  - Integer.parseInt(“10”)
  - Long.parseLong(“10”)
- 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

# 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);
```

# 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}");
```

# Klasa `StringTokenizer`

- Radi sličan posao kao i metoda split() klase String – "cepa" zadati string po delimiteru  (ili delimiterima)
- Ne radi sa regularnim izrazima
- Reziltat cepanja je kolekcija stringovva kroz koju se iterira metodama hasMoreTokens() i nextToken()

# Klasa **StringTokenizer**

```
import java.util.*;
class TokenizerTest {
  public static void main(String args[]) {
    String text = "Ovo je probni tekst";
    StringTokenizer st = new StringTokenizer(text, " ");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
  }
}
```

# 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);
```

# Konvencije davanja imena

- nazivi klasa (**`MojaKlasa`**)
- nazivi metoda (**`mojaMetoda`**)
- nazivi atributa (**`mojAtribut`**)
- nazivi paketa (**`mojpaket.drugipaket`**)
- set/get metode (**`setAtribut/getAtribut`**)
- konstante (MAX_INTEGER)