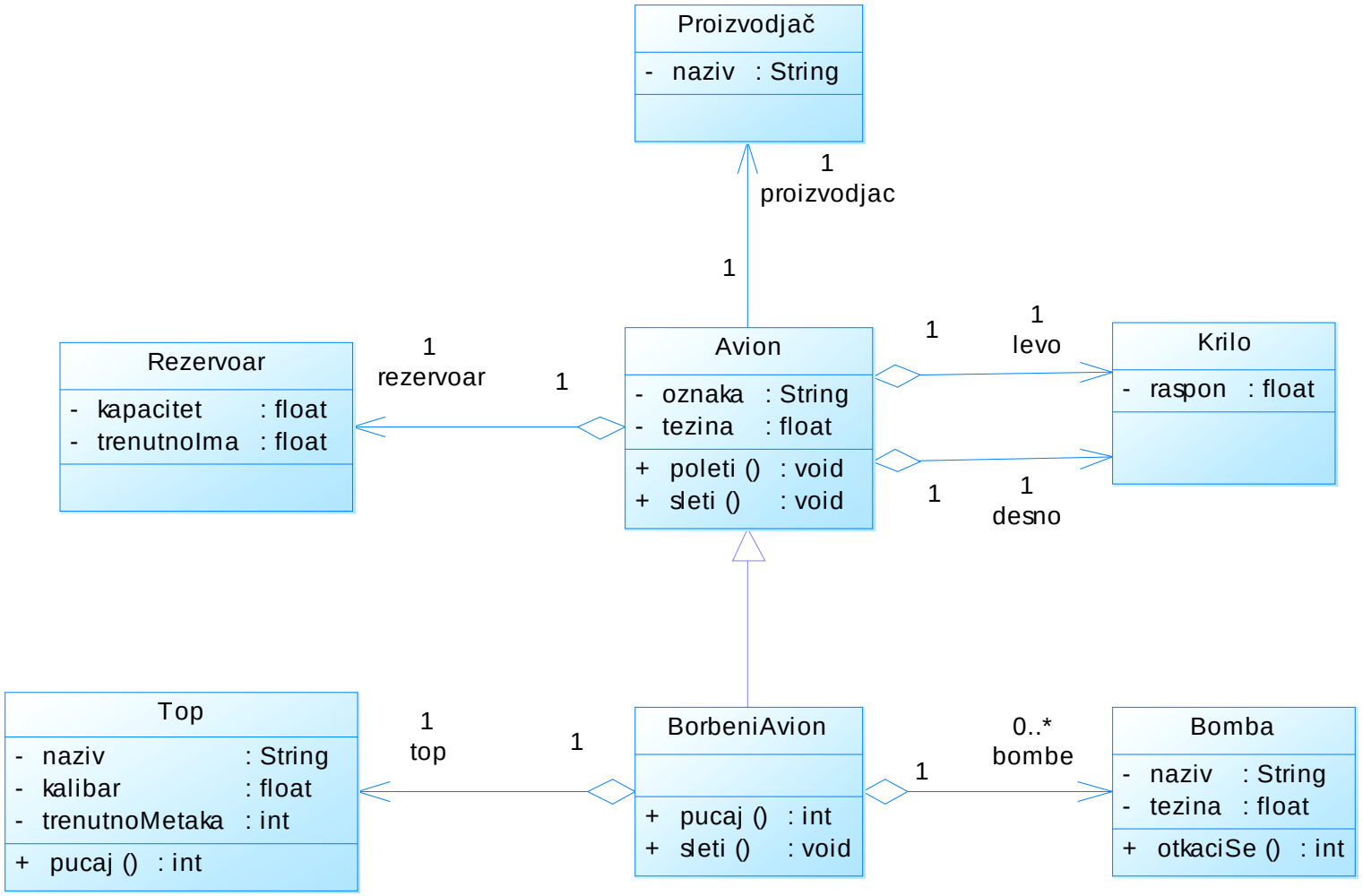
Objektno Orijentisano Programiranje 1

Nasleđivanje, apstraktne klase, polimorfizam, izuzeci

Nasleđivanje

- Preuzimanje atributa i metoda iz roditeljske klase
- Terminologija: klasa naslednica nasleđuje roditeljsku klasu
- Jednu roditeljsku klasu može da nasledi više klasa naslednica
- Obrnuto ne važi – jedna klasa može da nasledi samo jednu roditeljsku klasu
  - jednostruko nasleđivanje

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;
  Bomba[] bombe;
  void sleti() { ... }
  int pucaj() { ... }
}
```

- Postoji samo jednostruko nasleđivanje

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 (direktorijuma, foldera)

Modifikator pristupa klase

- Klasa može da ima *public* modifikator ispred definicije
- To znači da je vidljiva iz svih drugih klasa, bez obzira gde su definisane (bez obzira na paket-folder)
- Ako u jednom fajlu imamo više definicija klasa, samo ona čije ime se poklapa sa imenom datoteke mora da bude *public*
  - preporuka: jedna public klasa u jednom fajlu

Modifikator pristupa ispred konstruktora

- Ako konstruktor ima *public* modifikator, to znači da ta klasa može da se kreira iz bilo koje druge klase (bez obzira u kom paketu-folderu se nalazi)
- Ako stavimo *private* modifikator ispred konstruktora, niko ne može da kreira instance te klase
  - uvod u Singleton šablon

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 {
   void metoda1() {
     System.out.println("metoda1 klase A");
   }
   void metoda2() {
     System.out.println("metoda2 klase A");
   }
}
class B extends A {
   @Override
   void 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() { ... }
}
```

Ključna 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(); // svakako bi se izvršilo
   System.out.println("Konstruktor borbenog
   aviona.");
   top = new Top();
   bombe = new Bomba[] { new Bomba(),
            new Bomba()};
}
```

Ključna reč ***super*** u konstruktoru

- Konstruktor u klasi naslednici će svakako pozvati default konstruktor u roditeljskoj klasi, čak i kada ga ne napravimo
  - uz pomoć reči ***super***, možemo da pozovemo neki drugi konstruktor
- Preporuka:
  - kada god se pravi klasa, napraviti i default konstruktor

Ključna reč ***super*** u konstruktoru

- Ako napravimo konstruktor sa parametrom i u roditeljskoj i u klasi naslednici, bez poziva ***super(parametar)***, u klasi naslednici, roditeljski konstruktor sa parametrom se ne bi pozvao:

```
public class Automobil extends Vozilo {
  public Automobil() {
    super(); // ovo ne moramo
    System.out.println("Konstruktor Automobila");
  }
  public Automobil(String s) {
    // ovo moramo, ako želimo da se pozove konstruktor
    // sa parametrom u roditeljskoj klasi
    super(s);
    System.out.println("Konstruktor Automobila sa parametrom:
   " + s);
  }
}
```

Apstraktne klase

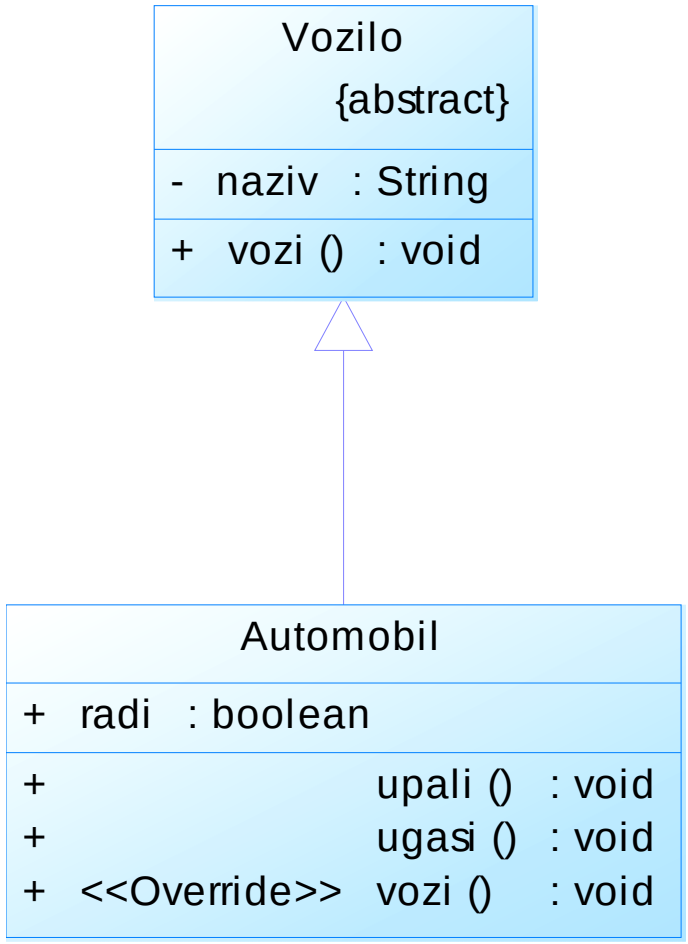
- Klase koje ne mogu imati svoje objekte, već samo njene klase naslednice mogu da imaju objekte (ako i one nisu apstraktne)

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

class B extends A {
  @Override
   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

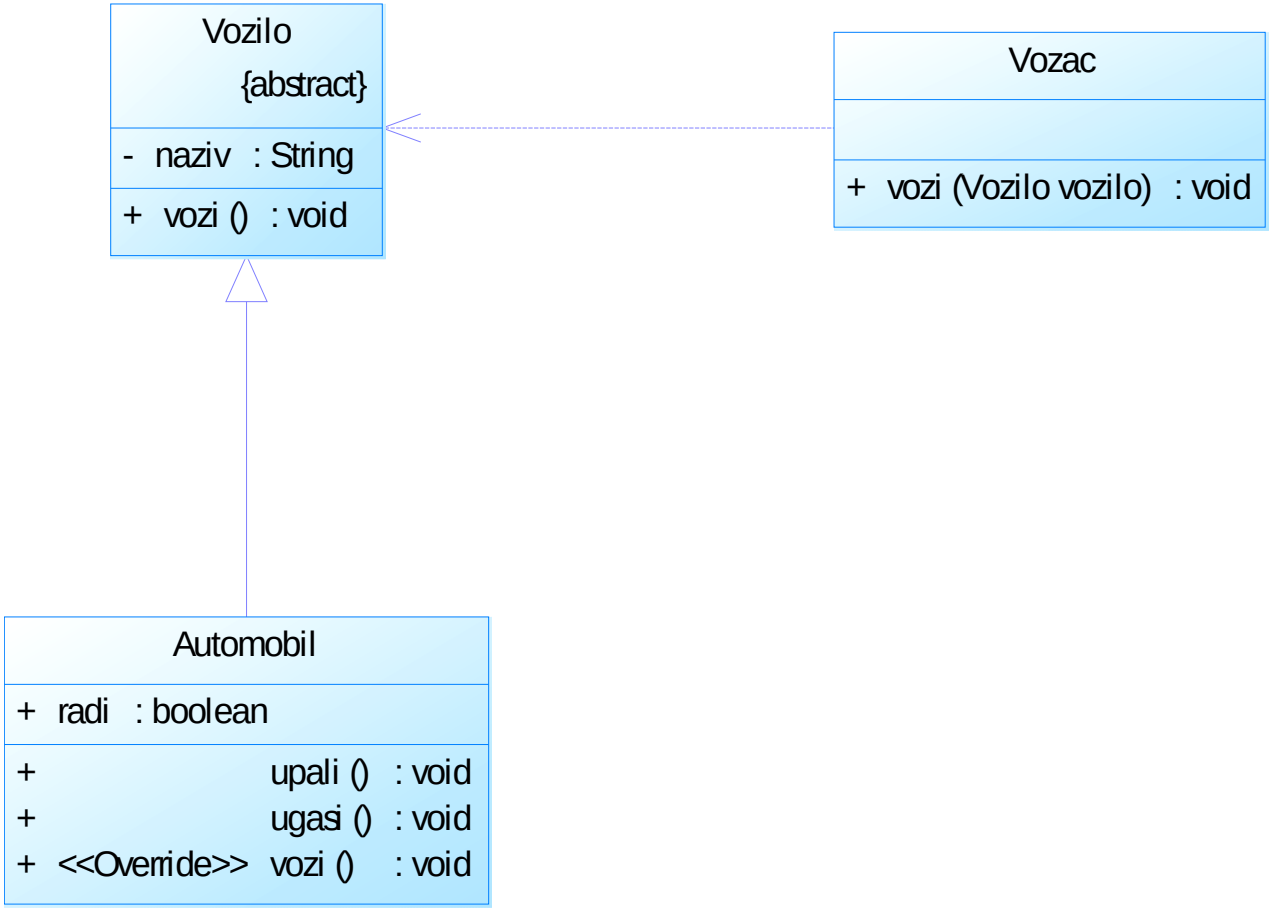
```
public abstract class Vozilo {
  private String naziv;
  public abstract void vozi();
}

public class Automobil extends Vozilo {
  public boolean radi;
  public void upali() {
    radi = true;
  }
  public void ugasi() {
    radi = false;
  }
  @Override
  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 {
  @Override
  void vozi() { ... }
}
class Kamion extends Vozilo {
  @Override
  void vozi() { ... }
}
class Vozac {
  void vozi(Vozilo v) {
    v.vozi();
  }
}
...
Vozac v = new Vozac();
v.vozi(new Automobil());
```

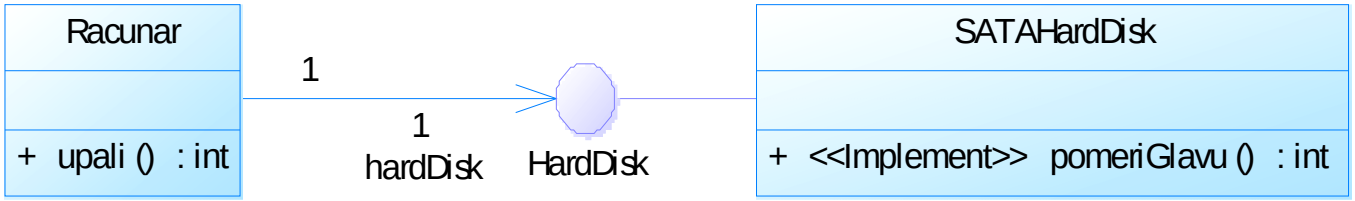
Polimorfizam

```
abstract class Vozilo {
  abstract double platiPutarinu();
}
class Automobil extends Vozilo {
  @Override
  double platiPutarinu() { return 240; }
}
class Kamion extends Vozilo {
  @Override
  double platiPutarinu() { return 1000; }
}
class NaplatnaRampa{
  double naplatiPutarinu(Vozilo v) {
    return v.platiPutarinu();
  }
}
...
NaplatnaRampa nr = new NaplatnaRampa();
double iznos = nr.naplatiPutarinu(new Automobil());
iznos += nr.naplatiPutarinu(new Kamion());
```

Interfejsi

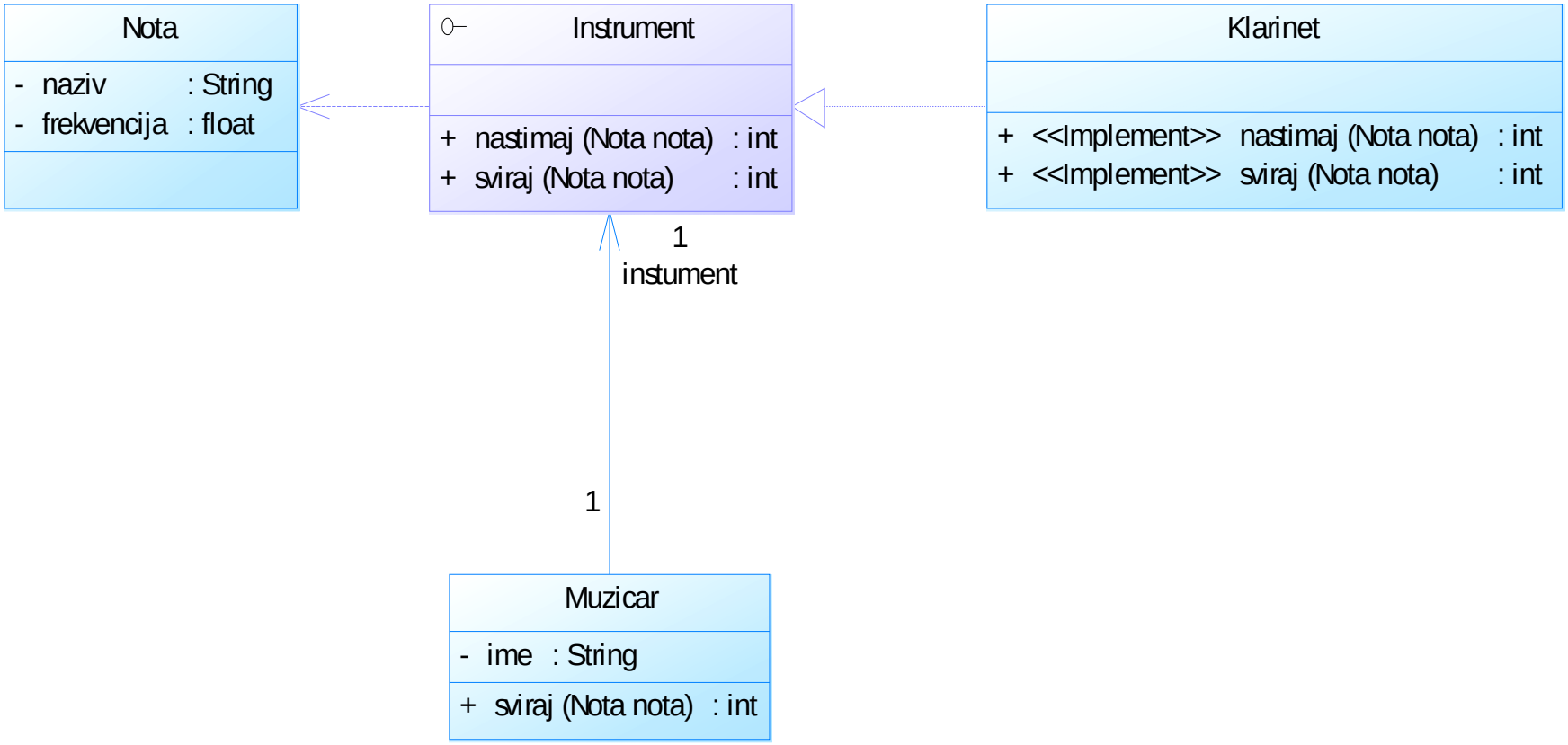
- Omogućavaju definisanje samo apstraktnih metoda, konstanti i statičkih atributa
- Interfejs nije klasa! On je spisak metoda i atributa koje klasa koja implementira interfejs mora da poseduje.
- Sve metode su implicitno public, a svi atributi su implicitno public static final.
- Interfejsi se ne nasleđuju, 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 {
  @Override
  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 {
  @Override
  public int sviraj(Nota nota) { ... }
  @Override
  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() { ... }
}
```

Unutrašnje klase

- Klase definisane bilo gde unutar neke druge klase
  - ta druga klasa se zove spoljašnja klasa

Inner classes (unutrašnje klase)

```
class Spoljasnja {
  Spoljasnja() { ... }
  void metoda() {
    new Unutrasnja(); // ovo nije problem
  }

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

Inner classes (unutrašnje klase)

- Unutrašnju klasu kreiramo iz spoljašnje klase bez dodatnih intervencija
- Konstrukcija objekta unutrašnje klase izvan spoljašnje klase:

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

Statičke unutrašnje klase

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

Anonimne unutrašnje klase

- Definišu se bilo gde unutar druge klase
  - može čak i unutar poziva metode!
- Nemaju ime, a definišu se na specifičan način
  - ime class fajla se sastoji iz rednog broja, separatora '$' i klase u kojoj se nalaze

Anonimne unutrašnje klase

```
JButton b = new JButton("Pritisni me");
ActionListener al = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ...
  }
};
b.addActionListener(al);

JButton b = new JButton("Pritisni me");
b.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
          ...
  }
});
```

Izuzeci

- Mehanizam prijavljivanja greške
- Greška se signalizira "bacanjem" izuzetka
- Metoda koja poziva potencijalno "grešnu" metodu "hvata" izuzetak
- Hijerarhija:
  - Throwable – roditeljska klasa
    - Error – ozbiljne sistemske greške
    - Exception – bazna klasa za sve standardne izuzetke
      - unchecked: RuntimeException i njene naslednice – ne moraju da se obuhvate try/catch blokom
      - checked: Ostale klase koje nasleđuju Exception klasu i koje moraju da se obuhvate try/catch blokom

Izuzeci

- Checked (Exception i njene naslednice) – moraju da se uhvate
  - **`IOException`**
  - **`SQLException`**
  - **`...`**
- Unchecked (RuntimeException i njene naslednice) – ne moraju da se uhvate, jer mogu da se programski spreče
  - **`NullPointerException`**
  - **`ArrayIndexOutOfBoundsException`**
  - **`ArithmeticException`**
  - **`...`**

Izuzeci

```
try {
  // kod koji može da izazove
  // izuzetak
} catch (java.io.IOException ex) {
  System.out.println("Problem sa datotekom!");
} catch (IndexOutOfBoundsException 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
  - ne moramo da obuhvatimo try-catch blokom, već da deklarišemo da i pozivajuća metoda takođe baca izuzetak
    - tako možemo da prebacujemo odgovornost hvatanja izuzetka na gore

Konvencije davanja imena

- Nazivi klasa pišu se malim slovima, ali početnim velikim slovom (npr. **`Automobil`**, **`ArrayList`**).
- Ukoliko se naziv klase sastoji iz više reči, reči se spajaju i svaka od njih počinje velikim slovom (npr. **`HashMap`**, **`ArrayIndexOutOfBoundsException`**).
- Nazivi metoda i atributa pišu se malim slovima (npr. **`size`**, **`width`**). Ako se sastoje od više reči, one se spajaju, pri čemu sve reči počevši od druge počinju velikim slovom (npr. **`setSize`**, **`handleMessage`**).
- Nazivi paketa pišu se isključivo malim slovima. Ukoliko se sastoje iz više reči, reči se spajaju (npr. **`mojpaket`**, **`velikipaket.malipaket`**).
- Detaljan opis konvencija nalazi se na adresi http://www.oracle.com/technetwork/java/codeconv-138413.html.

Konvencije davanja imena

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