Objektno orijentirano programiranje u Javi

Uvod u objektno-orijentirano programiranje

- Razlika između proceduralnog (linearnog) i objektno-orijentiranog programiranja svodi se na način pisanja i organiziranja programskog koda, te način izvođenja naredbi unutar programa
- Kod proceduralnog programiranja naredbe se izvode jedna po jedna, od početka do kraja datoteke koja sadrži izvorni kôd
- Kod objektno-orijentiranog programiranja programski kod je često podijeljen u više dijelova (klasa), pri čemu svaki dio ima posebnu namjenu
- Objektno-orijentirano programiranje se temelji na apstraktnijem pristupu pisanja programa, jer se koriste principi ponovnog iskorištavanja programskog koda (engl. *reuseability*) i korištenje obrazaca dizajniranja (engl. *design patterns*)

Četiri ključne značajke OOP-a

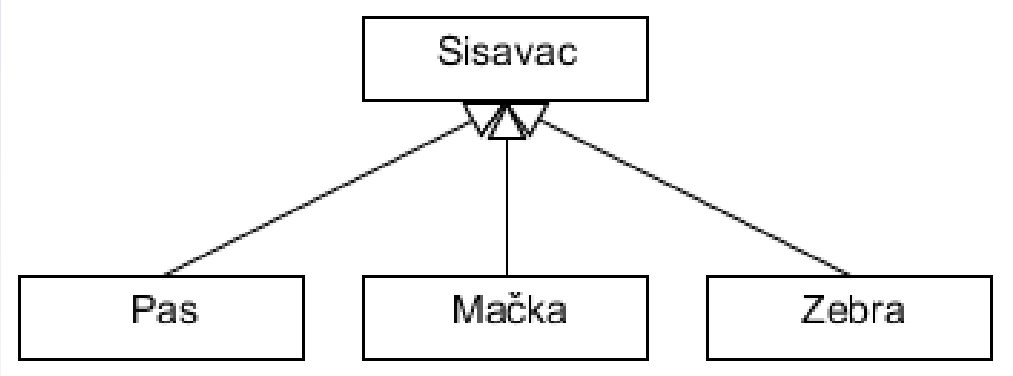
- Objektno-orijentirano programiranje temelji se na četiri ključne značajke:
  - **Generalizacija**
  - **Nasljeđivanje**
  - **Enkapsulacija**
  - **Polimorfizam**
- Generalizacija se temelji na određivanju zajedničkih svojstava objekata koji spadaju u zajedničku kategoriju
- Nasljeđivanje se odnosi na prenošenje svojstava iz jedne (nad)klase u ostale (pod)klase
- Enkapsulacija podrazumijeva "skrivanje" podataka o konkretnoj implementaciji ponašanja objekata klase
- Polimorfizam se temelji na dinamičkom pozivanju metoda klasa koje imaju zajedničku nadklasu ili sučelje

Generalizacija (1/2)

- Primjer: potrebno je dizajnirati model podataka koji će simulirati ponašanje različitih životinja
- Svaka životinja mora biti predstavljena zasebnom klasom, te tako omogućiti kreiranje objekata svake od klasa koji će predstavljati konkretne primjerke svake životinje
- Na primjer, potrebno je kreirati klasu za psa, mačku, zebru, papigu itd.
- Sve životinje imaju dosta zajedničkih svojstava, ali i različitosti
- Na primjer, psi, mačke i zebre spadaju u skupinu sisavaca, što je jedna posebna kategorija životinja
- U tom slučaju potrebno je kreirati klasu koja će sadržavati zajednička svojstva svih sisavaca i koju će nasljeđivati klase koje predstavljaju životinje koje su sisavci

Generalizacija (2/2)

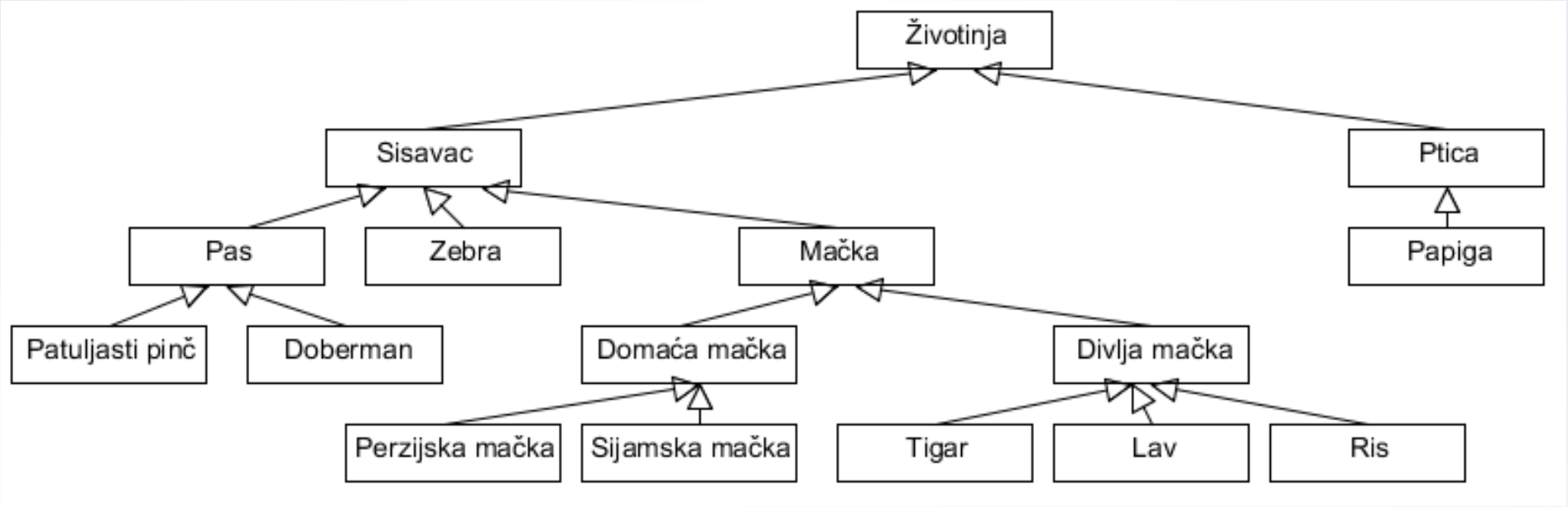
- UML dijagram klasa za tu hijerarhiju klasa izgleda ovako:



- Zajednička klasa "Sisavac" sadrži svojstva koja nasljeđuju sve ostale klase "Pas", "Mačka" i "Zebra"
- Klasa "Sisavac" se naziva **nadklasa** (engl. *Superclass*), a klase "Pas", "Mačka" i "Zebra" nazivaju se **podklase** (engl. *Subclass*)
- Klasa "Sisavac" je nadklasa klasi "Pas", a klasa "Pas" je podklasa klase "Sisavac"

Nasljeđivanje (1/4)

- Svojstvo preuzimanja podklasa zajedničkih svojstava nadklase naziva se nasljeđivanje
- Na primjer, moguće je UML dijagram klasa proširiti na sljedeći način i time dodatno specijalizirati klase i podvrste životinja:

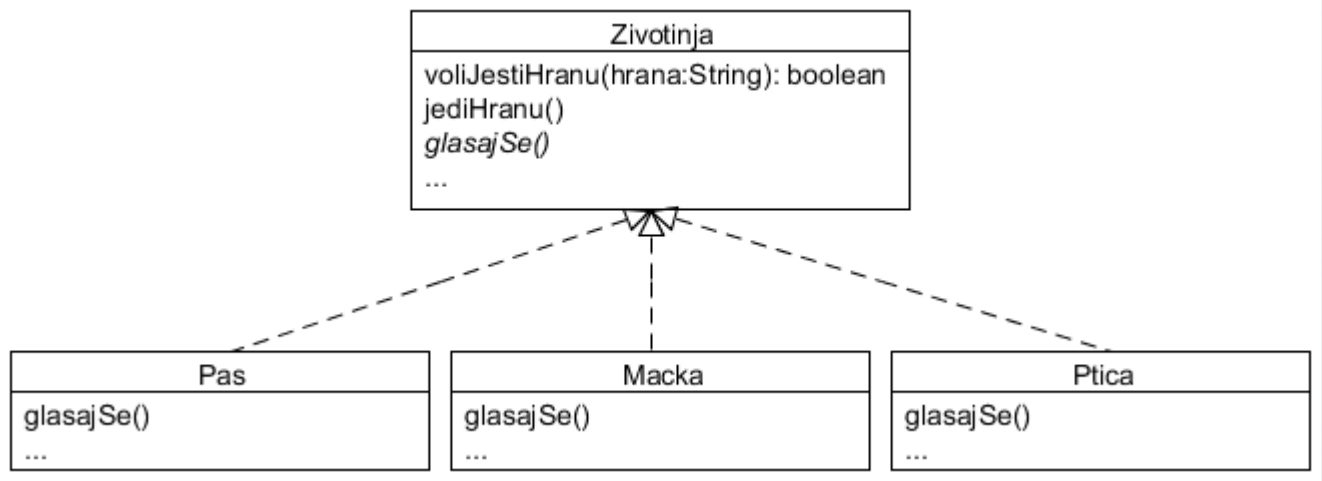


Nasljeđivanje (2/4)

- Uvođenjem novih klasa u model podataka maksimalno su iskorištene postojeće klase i na taj način je potrebno samo naslijediti neku od klasa i implementirati specifičnosti za novu klasu
- Također, uvođenjem novih svojstava (varijabli, metoda...) postojećim nadklasama, sve podklase također nasljeđuju i ta nova svojstva
- Svaka podklasa može naslijediti samo jednu nadklasu, međutim, može implementirati više sučelja
- Klase mogu biti i apstraktne, što znači da su **predviđene isključivo za nasljeđivanje**, te iz njih **nije moguće kreirati objekte**
- Osim toga klase mogu biti i finalizirane, što znači da **nisu predviđene za daljnja nasljeđivanja**

Nasljeđivanje (3/4)

- Primjer nasljeđivanja klasa:



- Klasa "Zivotinja" sadrži zajedničke metode koje nasljeđuju sve ostale klase, npr.

```
public class Zivotinja {
    public boolean voliJestiHranu(String hrana) {return false;}
    public void jediHranu() {this.gladna = false;}
    public void glasajSe() {System.out.println("???");}
}
```

Nasljeđivanje (4/4)

- Na primjer, klasa koja nasljeđuje klasu "Zivotinja" može "redefinirati" metode kako bi odgovarale njenom ponašanju, npr.:

```
public class Pas extends Zivotinja {
  public boolean voliJestHranu(String hrana) {
    return ("kosti".equals(hrana) == true);
  }
  public void glasajSe() {System.out.println("VAU-VAU!");}
}
```

- Pomoću ključne riječi "**extends**" označava se nasljeđivanje klasa, klasa "Pas" nasljeđuje klasu "Zivotinja"
- Tehnika nadjačavanja ("redefiniranja") metoda u podklasama naziva se *method overriding*

Apstraktne klase

- Apstraktne klase su predviđene isključivo za nasljeđivanje i iz njih nije moguće kreirati objekte
- Mogu sadržavati implementacije metoda, međutim, mogu sadržavati i apstraktne metode koje ne sadrže implementaciju
- Apstraktne metode sadrže samo "specifikaciju" metode: naziv, te ulazne i izlazne parametre, bez implementacije:

  npr. **public abstract void glasajSe();**
- Ako klasa **sadrži apstraktnu metodu**, **mora biti apstraktna**
- Apstraktna klasa može se proglasiti apstraktnom i u slučaju kad ne sadrži apstraktnu metodu
- Sve klase koje nasljeđuju apstraktnu nadklasu moraju implementirati sve apstraktne metode iz nadklase, inače će i one biti apstraktne

Sučelja

- Svaka podklasa može naslijediti samo jednu nadklasu, ali može implementirati više sučelja (engl. *interface*)
- Sučelja nisu klase, već označavaju "specifikaciju" koju moraju ispuniti klase koje ih implementiraju
- Ta specifikacija sastoji se od popisa metoda koje trebaju biti implementirane unutar klasa, na primjer:

```
public interface Mesojed {
   public void uhvatiPlijen(Zivotinja plijen);
}
```

- Klasa koja implementira sučelje izgledala bi ovako:

```
public class Macka implements Mesojed {
   ...
   public void uhvatiPlijen(Mis mis) {this.hrana=mis;}
   ...
}
```

Ključna riječ "final"

- Ukoliko je potrebno spriječiti daljnje nasljeđivanje neke klase ili daljnje redefiniranje neke metode iz nadklase u podklasama, potrebno je kod deklaracije koristiti ključnu riječ "final"
- Ako bi postojala, npr. klasa "PerzijskaMacka" i autor klase bi htio spriječiti daljnje nasljeđivanje te klase, njena deklaracija bi izgledala ovako:

  public final class PerzijskaMacka {...}
- Isto tako je moguće definirati i "final" metode koje se ne mogu nadjačati u podklasama, npr.

```
public class DomacaMacka extends Macka{
   ...
   public final void predi() {System.out.println("Mrrrrr!");}
   ...
}
```

Enkapsulacija

- Često je potrebno varijable unutar objekata "sakriti" i time onemogućiti izravan pristup prema njima
- Kako bi ipak bilo moguće koristiti podatke unutar objekata, potrebno je napisati "javne" metode za korištenje tih podataka
- Te metode često se nazivaju "metode instance"
- Autor klase definira način na koji su ti podaci dostupni "vanjskom svijetu", kreirajući "getter" i "setter" metode ili neke druge metode koje neizravno vraćaju vrijednosti varijabli
- Na taj način je definirano "javno sučelje" prema stanju objekata opisanom pomoću njegovih "privatnih varijabli"

Metode instance

- Pomoću metoda instance moguće je pristupati svim privatnim varijablama objekta (varijablama instance) ili pozivati neke od metoda koje se nalaze unutar klase
- Metoda instance (objekta) može se pozvati iz svakog objekta, te se obavlja nad podacima koji se nalaze u tom objektu
- Opći oblik metode instance izgleda ovako:

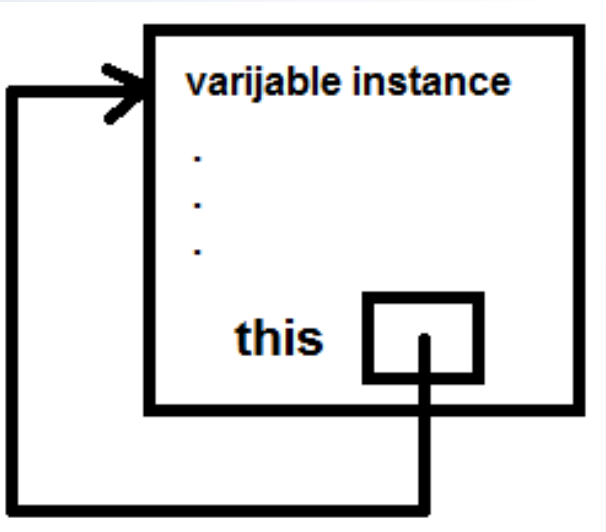
```
public "povratniTip" "imeMetode" ("listaParametara")
{
   //programski kod metode
}
```

- Metode instance pozivaju se tako da se navede naziv objekta, operator "." i naziv metode, npr.:

```
nazivObjekta.pozivMetode(ulazniParametar);
```

Ključna riječ "this"

- Ključna riječ "this" služi za referenciranje sastavnih dijelova klase kao što su varijable instance i metode, ali i samog objekta:



- Često se koristi kod "setter" metoda (name je privatna varijabla klase, a name naziv ulaznog parametra metode):

```
public void setName(String name) {
    this.name = name;
}
```

Ključna riječ "super"

- Ukoliko je potrebno unutar podklasa pristupiti nekoj metodi ili varijabli iz nadklase, potrebno je koristiti ključnu riječ "super"
- Slično kao što ključna riječ "this" služi za dohvaćanje elemenata iz same klase, "super" dohvaća elemente nadklase, npr.

```
public class Zivotinja {...
   public void jediHranu() {this.gladna = false;}...
}
public class Pas extends Zivotinja {...
   if (this.gladna == true && hrana != null)
      super.jediHranu();
}
```

Metode i varijable klase (1/2)

- Osim metoda i varijabla instance, moguće je napisati metode i varijable klase, koje su zajedničke za sve objekte klase
- Takve metode i klase još se nazivaju i "statičke", jer se kod deklaracije koristi ključna riječ "static"
- Varijable klase najčešće su neke konstante vezane uz klasu ili neka varijabla zajednička za sve objekte (npr. brojač kreiranih objekata):

```
public static final int BROJ_ECTS_BODOVA = 30;
public static int brojacObjekata = 0;
```

Metode i varijable klase (2/2)

- Metode klase najčešće sadrže zajedničku logiku koju koriste svi objekti klase, npr.

```
public static void resetBrojacObjekata() {
        brojacObjekata = 0;
}
```

- Statičke metode pozivaju se bez potrebe za instanciranjem objekata (kao što je to slučaj kod metoda instance), već se mogu izravno pozivati iz klase na sljedeći način (pretpostavlja se da se metoda "resetBrojacObjekata" nalazi unutar klase "Student"):

```
Student.resetBrojacObjekata();
```

- "Student" predstavlja samu klasu, a ne objekt klase

Polimorfizam (1/2)

- Svojstvo akcije ili metode da se "ponaša" drugačije, ovisno o tipu objekta nad kojim se izvršava
- Postoje tri vrste manifestiranja polimorfizma: *overloading*, *overriding* i *dynamic method binding*
- Dinamičko povezivanje metoda temelji se na nasljeđivanju, a metoda koja se poziva određuje se tek u fazi izvođenja programa
- Na primjer, definiranjem sljedećih objekata:

```
Zivotinja pas = new Pas();
Zivotinja macka = new Macka();
Zivotinja ptica = new Ptica();
```

- definiraju se "konkretni" objekti, ali njihovo "sučelje" je zajedničko i definirano unutar klase "Zivotinja"

Polimorfizam (2/2)

- Ako se ti objekti spreme u jedno polje koje sadrži sve objekte klasa kojima je zajedničko nasljeđivanje klase "Zivotinja", u trenutku prevođenja programa nije moguće znati o kojem tipu objekta se radi:

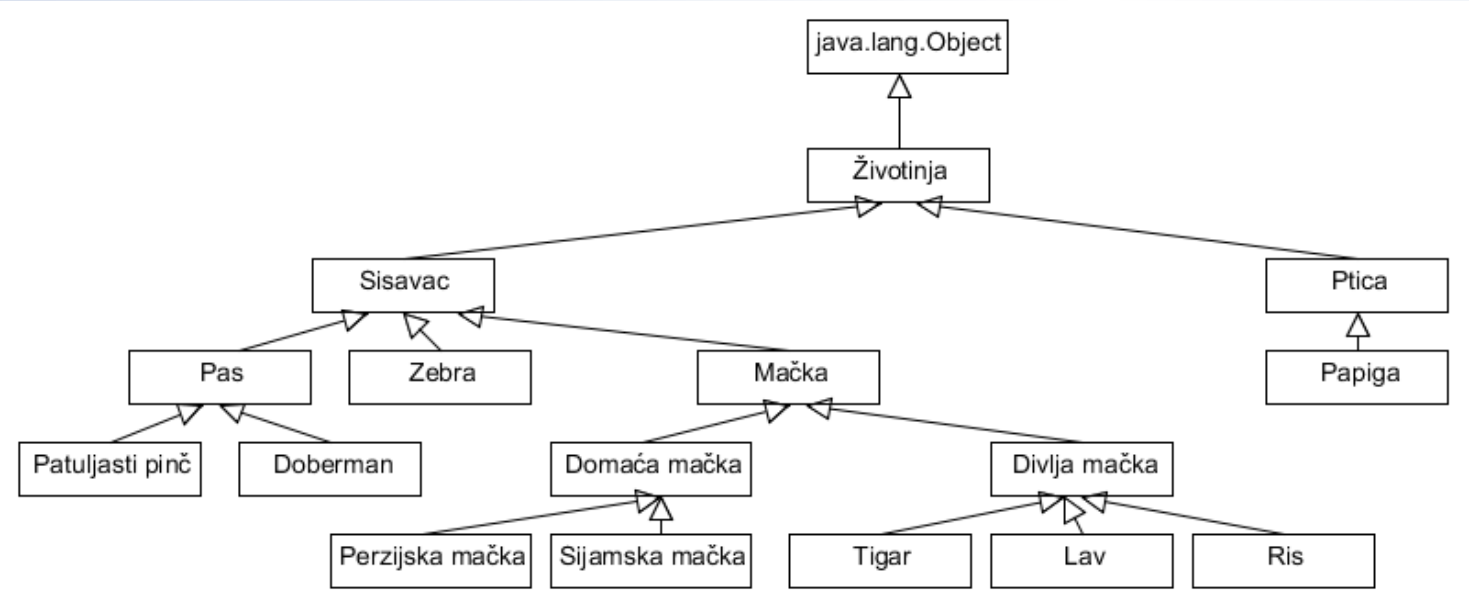
```
Zivotinja[] polje = new Zivotinja[3];
polje[0] = pas; polje[1] = macka; polje[2] = ptica;
```

- Kako je metoda "glasajSe()" apstraktna unutar apstraktne klase "Zivotinja", nije ju moguće pozvati, jer ni nema implementacije
- Međutim, dinamičkim povezivanjem u trenutku izvođenja programa pozvati će se ispravna metoda "glasajSe()" za pojedini objekt:

```
for (Zivotinja zivotinja : polje) {
    zivotinja.glasajSe();
}
```

Klasa "java.lang.Object"

- Postoji jedna klasa koju izravno ili neizravno nasljeđuju sve ostale klase u Javi: java.lang.Object



- Ta klasa sadrži osnovne metode koje su zajedničke za sve klase u Javi, kao što su "toString()" za pretvorbu objekta u String, "equals(Object drugiObjekt)" za usporedbu objekata itd.

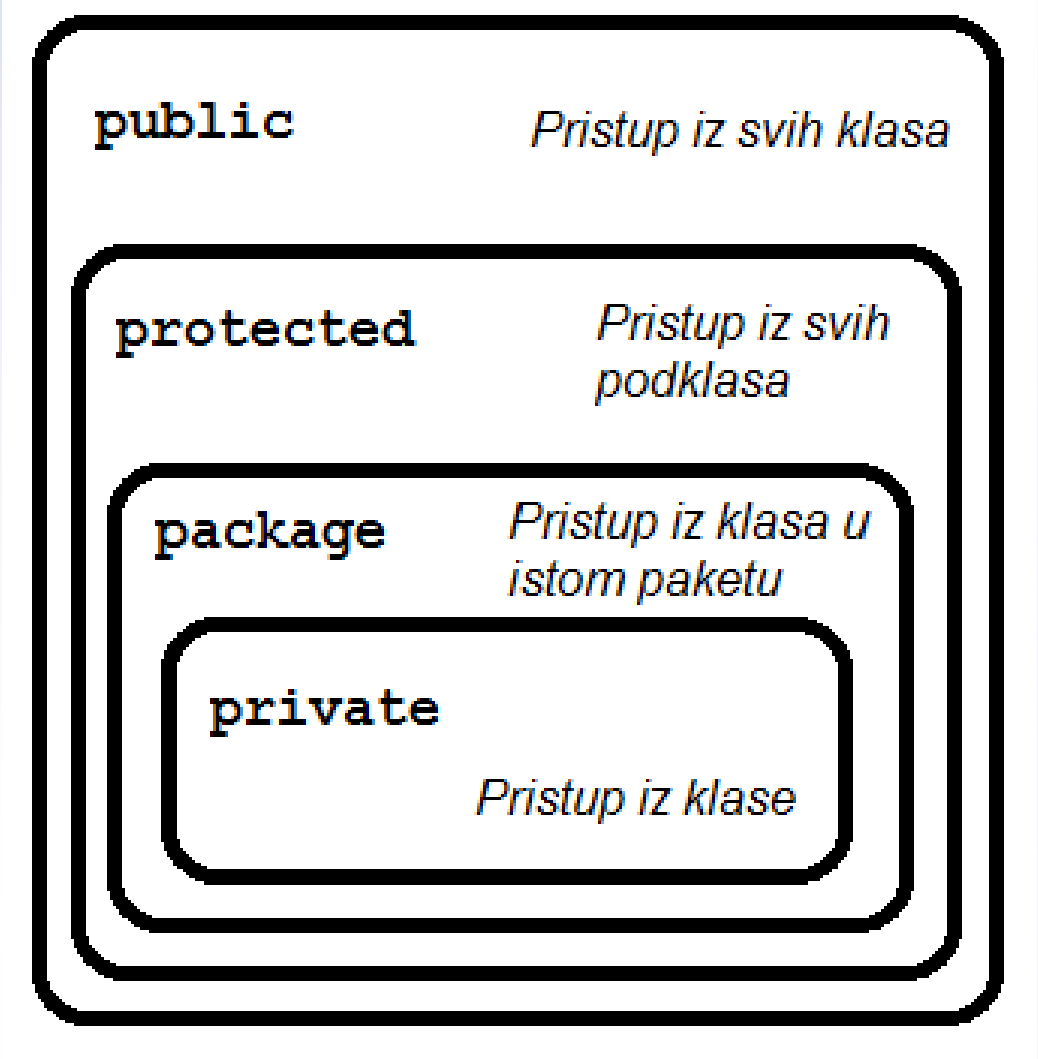
Klase i paketi

- Paketi služe za organiziranje klasa u cjeline
- Definiraju modul aplikacije i sastavni su dijelovi imena klase, po kojima se one razlikuju
- Ako se dvije klase jednako nazivaju, a nalaze se u različitim paketima, one se tretiraju kao različite klase
- Korištenjem određenih modifikatora koji definiraju prava pristupa sadržaju klasa moguće je definirati da su neke klase "vidljive" jedne drugima samo ako se nalaze unutar istog paketa

Prava pristupa između komponenti (1/2)

- U Javi postoji četiri različitih modifikatora koji označavaju različite razine pristupa između klasa i komponenti (metode i varijable): **private**, **package** (bez modifikatora), **protected** i **public**
- Ukoliko je komponenta označena modifikatorom **public**, znači da joj se može pristupati iz svih ostalih klasa
- Klase koje nisu označene nijednim modifikatorom, znači da su međusobno "vidljive" unutar istog paketa
- Komponente koje su označene modifikatorom **protected**, vidljive su u podklasama u slučaju nasljeđivanja
- Komponentama koje su označene modifikatorom **private**, moguće je pristupati samo unutar klase

Prava pristupa između komponenti (2/2)



Opći oblik klase

```
deklaracija paketa;

import deklaracije;

[public] [abstract] [final] class ImeKlase
[extends ImeNadklase]
[implements ImeSučelja]
{
    [konstruktor]
    . . .
    deklaracija varijabli
    . . .
    deklaracija metoda
    . . .
}
```

Modifikatori i ključne riječi za klase

- Modifikator **public**: klasa se može koristiti u svim drugim klasama
- Bez modifikatora **public**: klasa se može koristiti samo u paketu u kojem se nalazi
- Modifikator **abstract**: klasa je predviđena isključivo za nasljeđivanje i nije moguće kreirati objekte iz takve klase
- Bez modifikatora **abstract**: klasa se može koristiti za nasljeđivanje i moguće je kreirati objekte iz takve klase
- Modifikator **final**: klasa nije predviđena za nasljeđivanje
- Bez modifikatora **final**: klasa se može nasljeđivati
- Ključna riječ **extends**: označava klasu koja se nasljeđuje
- Ključna riječ **implements**: označava jedno ili više sučelja koja implementira klasa

Modifikatori za metode (1/2)

- Modifikator **public**: metodu je moguće pozivati iz svih ostalih klasa
- Modifikator **protected**: metodu je moguće pozivati samo u klasi u kojoj se nalazi, u klasama koje se nalaze u istom paketu i u klasama koje nasljeđuju klasu s tom metodom
- Modifikator **private**: metodu je moguće koristiti samo u klasi u kojoj se nalazi
- Modifikator **abstract**: metodu je potrebno implementirati u klasama koje nasljeđuju apstraktnu klasu u kojoj se metoda nalazi
- Modifikator **final**: metoda ne može biti nadjačana (engl. *overriden*) u podklasama koje nasljeđuju klasu s metodom koja ima taj modifikator

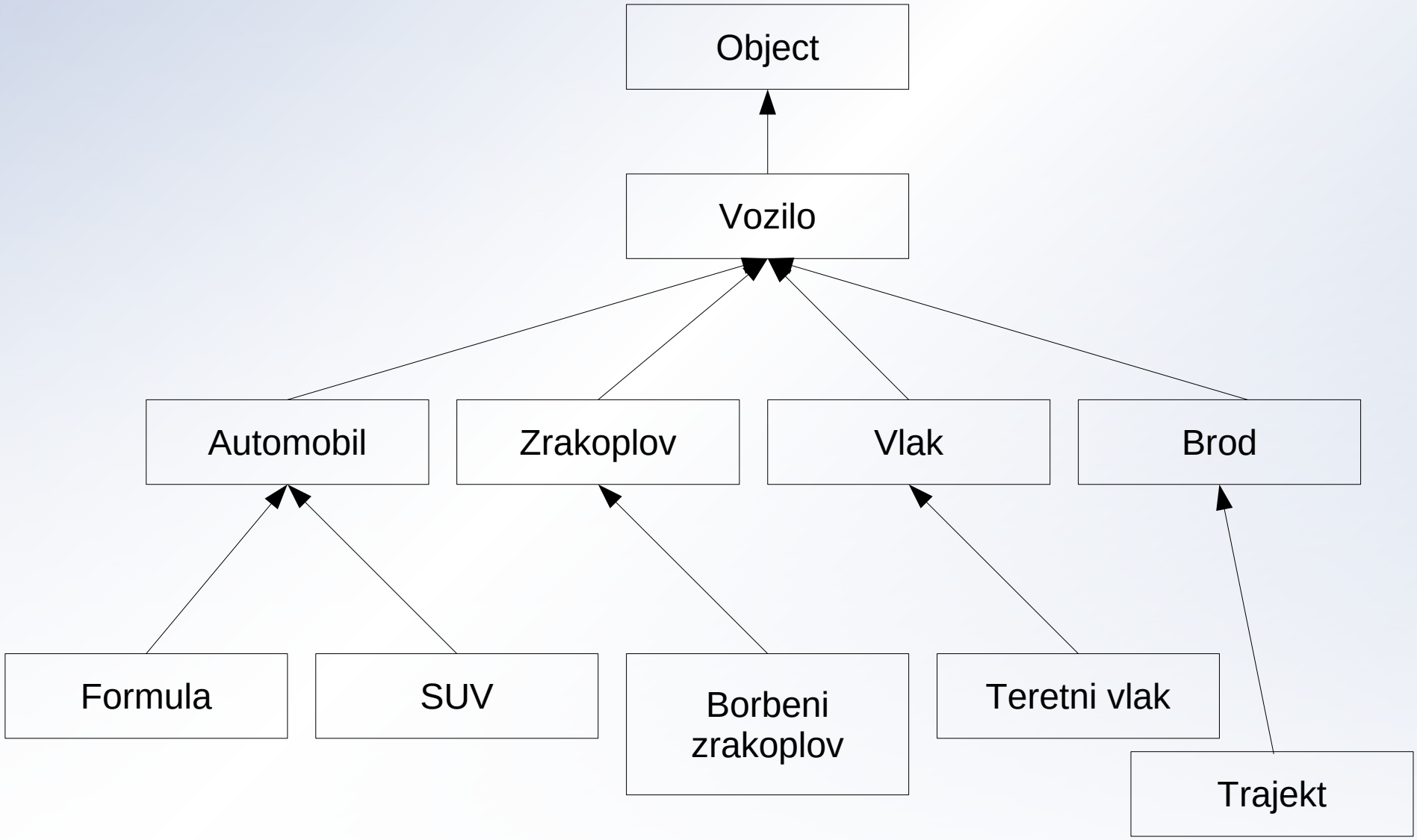
Modifikatori za metode (2/2)

- Modifikator **native**: implementacija metode je napisana u nekom drugom jeziku (npr. C-u) i predana je preko vanjskog resursa
- Modifikator **static**: metoda klase koja je zajednička za sve objekte te klase
- Modifikator **synchronized**: omogućava sigurno izvođenje metoda u višenitnim sustavima

Modifikatori za varijable

- Modifikator **public**: varijablu je moguće koristiti izravno, a ne samo preko poziva metode (npr. "settera")
- Modifikator **private**: varijabli je moguće pristupati samo iz klase u kojoj je definirana, a izvan nje joj je moguće pristupati samo uz pomoć javnih metoda ("gettera" i "settera")
- Modifikator **protected**: varijabli je moguće pristupati samo iz klase u kojoj se nalazi, iz klasa koje se nalaze u istom paketu i klasa koje nasljeđuju klasu s tom varijablom
- Modifikator **static**: označava statičku varijablu koja je zajednička za sve objekte neke klase
- Modifikator **final**: označava varijablu čija vrijednost se ne može mijenjati (konstanta)

Primjer nasljeđivanja (1/5)



Primjer nasljeđivanja (2/5)

```
package nasljedjivanje;

public class Vozilo {

    private int brojPutnika;
    private float potrosnja;

    public Vozilo(int brojPutnika, float potrosnja) {
        this.brojPutnika = brojPutnika;
        this.potrosnja = potrosnja;
    }

    public int getBrojPutnika() {
        return brojPutnika;
    }
    public void setBrojPutnika(int brojPutnika) {
        this.brojPutnika = brojPutnika;
    }
    ...
}
```

Primjer nasljeđivanja (3/5)

```
package nasljedjivanje;

public class Automobil extends Vozilo {

    public static final String VRSTA_MOTORA_BENZINSKI =
                                                    "VRSTA_MOTORA_BENZINSKI";
    public static final String VRSTA_MOTORA_DIZEL = "VRSTA_MOTORA_DIZEL";
    public static final String VRSTA_MOTORA_HIBRIDNI = "VRSTA_MOTORA_HIBRIDNI";

    private String vrstaMotora;
    private int brojVrata;

    public Automobil(String vrstaMotora, int brojVrata, int brojPutnika, float
                     potrosnja) {
            super(brojPutnika, potrosnja);
            this.vrstaMotora = vrstaMotora;
            this.brojVrata = brojVrata;
    }

    public String getVrstaMotora() {
        return vrstaMotora;
    }
    public void setVrstaMotora(String vrstaMotora) {
        this.vrstaMotora = vrstaMotora;
    }
    ...
}
```

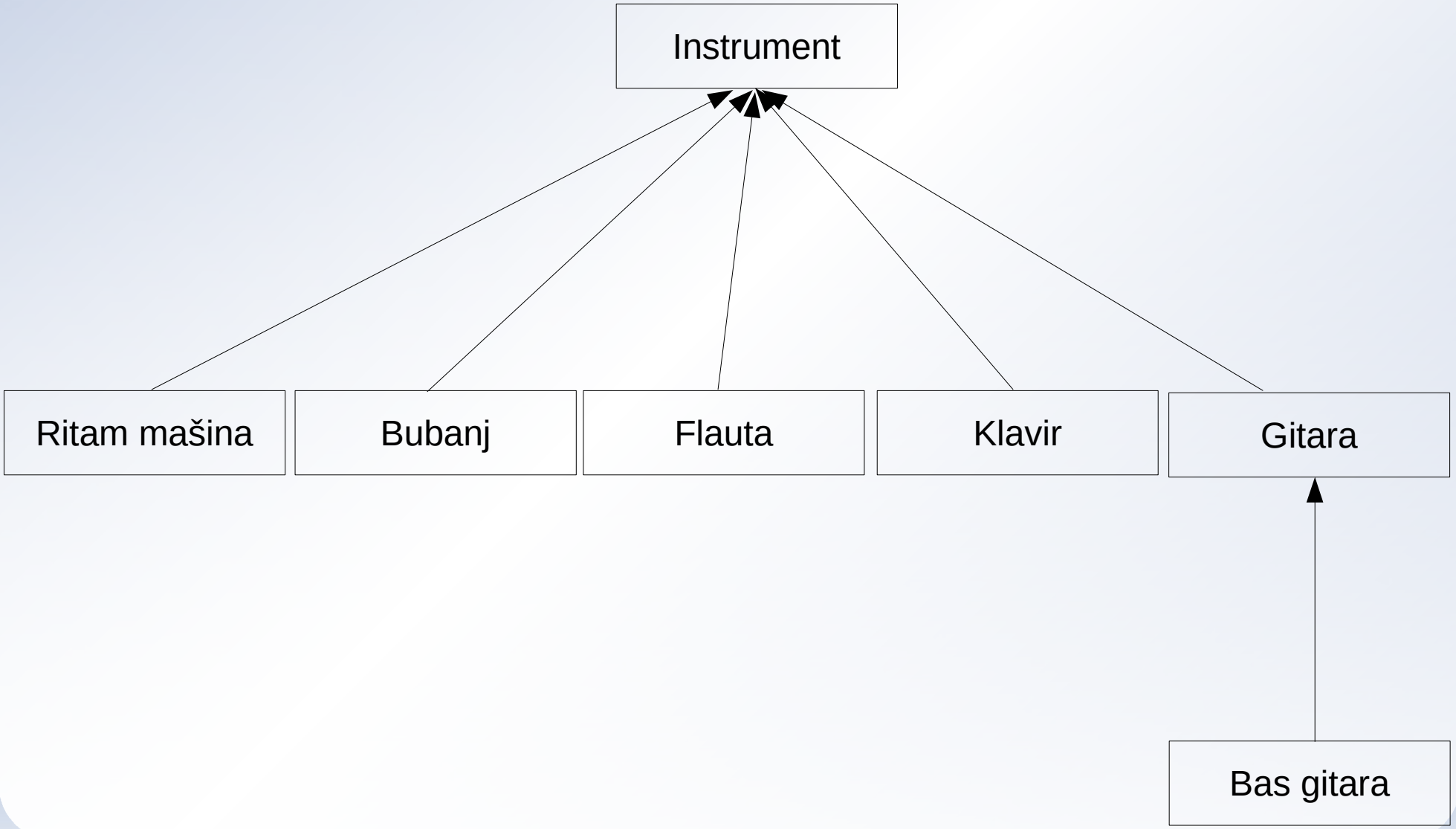
Primjer nasljeđivanja (4/5)

```
package nasljedjivanje;

public class Zrakoplov extends Vozilo {

    private float rasponKrila;
    private float sirinaTrupa;

    public Zrakoplov(int brojPutnika, float potrosnja,
                     float rasponKrila,  float sirinaTrupa)
    {
              super(brojPutnika, potrosnja);
              this.rasponKrila = rasponKrila;
              this.sirinaTrupa = sirinaTrupa;
    }
    public float getRasponKrila() {
        return rasponKrila;
    }
    public void setRasponKrila(float rasponKrila) {
        this.rasponKrila = rasponKrila;
    }
    ...
}
```

Primjer nasljeđivanja (5/5)

```
package nasljedjivanje;

public class Test {

    /**
    * @param args
    */
    public static void main(String[] args) {
        Automobil auto = new
            Automobil(Automobil.VRSTA_MOTORA_BENZINSKI, 5, 5, (float) 8.9);

        Zrakoplov avion = new
            Zrakoplov(100, 200, (float)25.9, (float) 5.3);

        if (auto.getPotrosnja() > avion.getPotrosnja()) {
            System.out.println("Auto troši više od aviona!");
        }
        else {
            System.out.println("Avion troši više od auta!");
        }
    }
}
```

Primjer polimorfizma (1/10)



Primjer polimorfizma (2/10)

```
package polimorfizam;

public abstract class Instrument {

    public static final int VRSTA_INSTRUMENTA_PUHACKI = 1;
    public static final int VRSTA_INSTRUMENTA_UDARALJKE = 2;
    public static final int VRSTA_INSTRUMENTA_ZICANI = 3;
    public static final int VRSTA_INSTRUMENTA_ELEKTRONICKI = 4;
    public static final int VRSTA_INSTRUMENTA_KLAVIJATURE = 5;

    private int vrstaInstrumenta;

    public int getVrstaInstrumenta() {
        return vrstaInstrumenta;
    }

    public void setVrstaInstrumenta(int vrstaInstrumenta) {
        this.vrstaInstrumenta = vrstaInstrumenta;
    }

    public abstract void odsvirajNotu();

}
```

Primjer polimorfizma (3/10)

```
package polimorfizam;

public class Bubanj extends Instrument {

    public Bubanj() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_UDARALJKE);
    }

    @Override
    public void odsvirajNotu() {
        System.out.println("Nota odsvirana bubnjem!");
    }

}
```

Primjer polimorfizma (4/10)

```
package polimorfizam;

public class Flauta extends Instrument {

    public Flauta() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_PUHACKI);
    }

    @Override
    public void odsvirajNotu() {
        System.out.println("Nota odsvirana flautom!");
    }

}
```

Primjer polimorfizma (5/10)

```
package polimorfizam;

public class Gitara extends Instrument {

    public Gitara() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_ZICANI);
    }

    @Override
    public void odsvirajNotu() {
        System.out.println("Nota odsvirana gitarom");
    }

}
```

Primjer polimorfizma (6/10)

```
package polimorfizam;

public class Klavir extends Instrument {

    public Klavir() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_KLAVIJATURE);
    }

    @Override
    public void odsvirajNotu() {
        System.out.println("Nota odsvirana klavirom!");
    }

}
```

Primjer polimorfizma (7/10)

```
package polimorfizam;

public class RitamMasina extends Instrument {

    public RitamMasina() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_ELEKTRONICKI);
    }

    @Override
    public void odsvirajNotu() {
        System.out.println("Ritam odsviran ritam-mašinom!");
    }

}
```

Primjer polimorfizma (8/10)

```
package polimorfizam;

import java.util.Random;

public class PolimorfizamTest {

    public static final int BROJ_INSTRUMENATA = 10;

    /**
    * @param args
    */
    public static void main(String[] args) {

        Instrument[] instrumenti = new
                                    Instrument[BROJ_INSTRUMENATA];

        instrumenti = popuniPoljeRandomInstrumentima(instrumenti);

        for (int i = 0; i < BROJ_INSTRUMENATA; i++) {
            instrumenti[i].odsvirajNotu();
        }
}
```

Primjer polimorfizma (9/10)

```
private static Instrument[] popuniPoljeRandomInstrumentima(
                                        Instrument[] instrumenti) {

    Random random = new Random();

    for (int i = 0; i < BROJ_INSTRUMENATA; i++) {
        switch(random.nextInt(5)) {
            case 0 :
                instrumenti[i] = new Flauta();break;
            case 1 :
                instrumenti[i] = new Bubanj();break;
            case 2 :
                instrumenti[i] = new Gitara();break;
            case 3 :
                instrumenti[i] = new Klavir();break;
            case 4 :
                instrumenti[i] = new RitamMasina();break;
            default:
                instrumenti[i] = new RitamMasina();break;
            }
        }

    return instrumenti;

    }
}
```

Primjer polimorfizma (10/10)

- Ispis:

```
Nota odsvirana flautom!
Nota odsvirana gitarom!
Nota odsvirana ritam mašinom!
Nota odsvirana klavirom!
Nota odsvirana bubnjem!
Nota odsvirana gitarom!
Nota odsvirana bubnjem!
Nota odsvirana gitarom!
Nota odsvirana flautom!
Nota odsvirana klavirom!
```

Primjer sučelja

- Pomoću sučelja "Elektricno" instrumentima je moguće dodati svojstvo uključivanja i isključivanja, koje je potrebno implementirati unutar metoda sučelja:

```
package polimorfizam;

public interface Elektricno {

   public void ukljuci();
   public void iskljuci();

}
```

Primjer implementacije sučelja

```
package polimorfizam;

public class RitamMasina extends Instrument implements Elektricno {

    private boolean ukljuceno;

    public RitamMasina() {
        setVrstaInstrumenta(VRSTA_INSTRUMENTA_ELEKTRONICKI);
        ukljuceno = false;
    }

    @Override
    public void odsvirajNotu() {

        if (ukljuceno == true) {
            System.out.println("Nota odsvirana ritam-mašinom!");
        }
        else {
            System.out.println("Ritam-mašina je isključena!");
        }
    }

    public void ukljuci() {
        ukljuceno = true;
    }

    public void iskljuci() {
        ukljuceno = false;
    }
}
```

Cast operacija (1/2)

- Često je potrebno nekoj varijabli dodijeliti neki drugi tip podatka kako bi se izvršila zadana operacija, npr.

```
int suma = 100;
int broj = 20;
float prosjek = (float) suma / broj;
```

- Operacija promjene tipa podatka korištenjem novog tipa u zagradi ispred varijable naziva se "*cast*" operacija
- Slično je moguće napraviti i s objektima, na primjer:

```
Instrument ritamMasina = new RitamMasina();
((RitamMasina) ritamMasina).ukljuci();
```

ili

```
Zrakoplov drugiAvion = new Zrakoplov(...);
Zrakoplov treciAvion = (Zrakoplov) drugiAvion.clone();
```

Cast operacija (2/2)

- Ukoliko se ne postave tipovi objekata na ispravan način, *compiler* često javlja pogrešku:

  "**Type mismatch: cannot convert from Object to Zrakoplov**"

  (u slučaju kad bi se pozvala sljedeća naredba:

  Zrakoplov treciAvion = drugiAvion.clone();)
- U slučaju da se tijekom izvođenja dogodi da se pokuša *cast*-ati objekt tipa koji nije kompatibilan s tipom u koji objekt treba pretvoriti, događa se iznimka "**ClassCastException**".

Operator *instanceof*

- Pomoću operatora *instanceof* moguće je provjeriti je li neki objekt instanca neke određene klase
- Na primjer, ako je potrebno provjeriti je li objekt "mojObjekt" instanca klase "Gitara" to je moguće napraviti na sljedeći način:

```
if (mojObjekt instanceof Gitara) {
   //objekt "mojObjekt" je instanca klase "Gitara"
}
```

- Operator *instanceof* se često koristi kako bi se izbjeglo generiranje iznimke "ClassCastException"