# 5. Inheritance and Polymorphism

INHERITANCE - ključni koncept OOP-a, dozvoljava da se nove klase grade na osnovu postojećih klasa
⟹ nadograđuju se klase sa novima

Kada neka klasa nasljeđuje od postojeće klase, ona reuses (nasljeđuje) metode i atribute bazne klase i dodaje nove metode i polja za primjenu klase na specifičnu situaciju => izvedena klasa extends baznu (početnu) klasu

↳ Nešto što je zajedničko za više klasa ne treba pisati više puta, već jednom i onda nasljeđivati.
↳ Te klase nasljeđe zajedničko i dodaju nove metode i polja da bi se koristile nešto specifično.

→ iste klase i metode drugo znacenje u ov. kontekstu

**POLYMORPHISM** je pojam usko vezan za inheritance i on daje snagu OOP-u. Značenje je => varijable početne klase se mogu odnositi na bilo koji objekat izvedenog tipa, dok metode pozvane na početni (base) tip se šalju na izvedeni tip.

↳ U base classi definišemo objekat i metode, a zatim objekti izvedenog tipa se mogu odnositi na te osnovne objekte.

npr: Ako je base object Shape, a novi → izvedeni objekat je Circle, on predstavlja Shape s nekim parametrima. Sve metode koje se koriste na Shape mogu i na Circle

Inheritance daje formalan mehanizam za reuse koda (reciklirање, ponovnu upotrebu). Dakle, korištenjem inheritence, gradimo nove klase koje dodaju atribute i metode postojećim klasama.

PRIMJER: Zaposlenici u kompaniji.

- Svaki zaposlenik u firmi ima ime.
  - Postoje i specijalne vrste zaposlenika (npr. programer koji koristi specifičan programski jezik ili menadžer prodaje zadužen za određen broj klijenata)

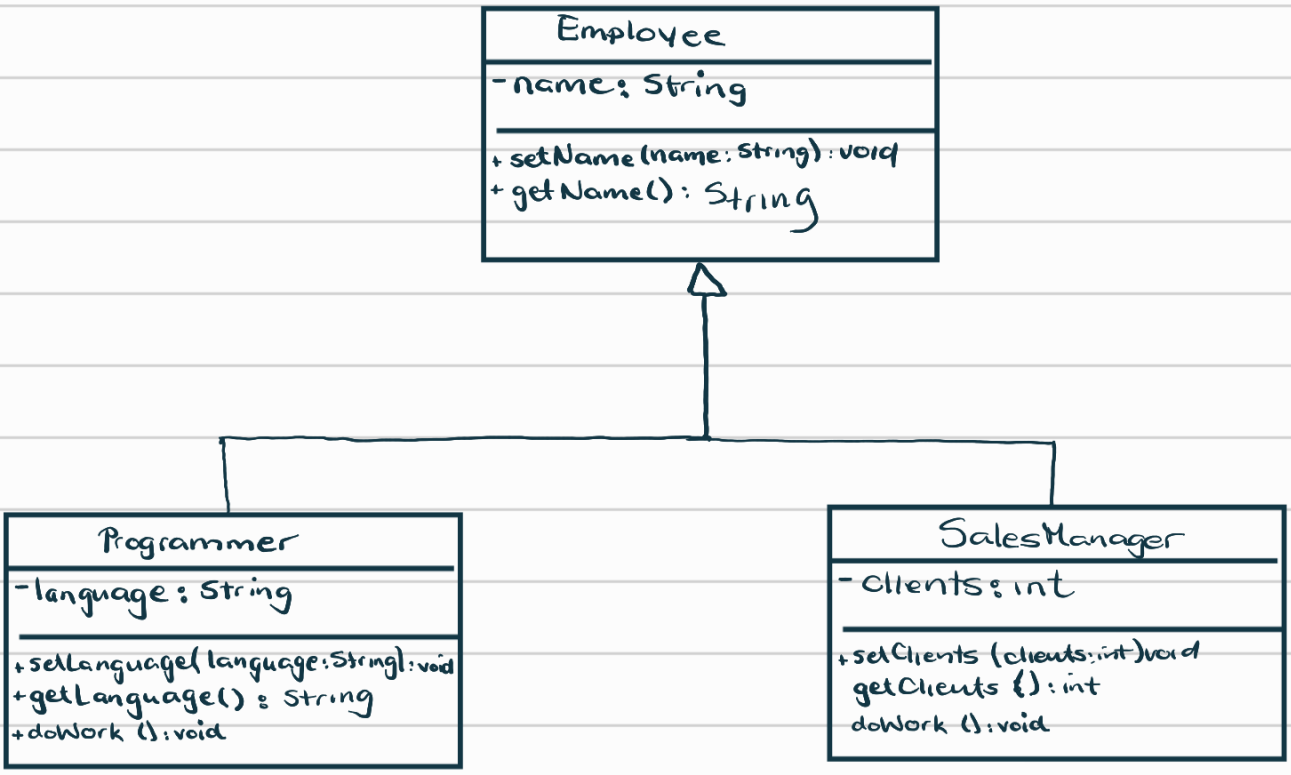
Da bi uspostavili model klase za ovu situaciju, prvo definišemo prikladne klase i zatim veze između njih.

↦ Klase koje koristimo su Employee, Programmer, SalesManager.

Kako su Programmer i SalesManager posebne vrste zaposlenika, proširuju pojam zaposlenika, te dvije klase nasljeđuju od klase Employee.

Metoda doWork ispisuje šta programeri i menadžeri rade tokom radnog vremena

* Inheritance se označava trouglom usmjerenim ka osnovnoj bazi




```java
public class Employee {

  private String name;

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

  public String getName() {
    return name;
  }
}
```

Definisanje „set" i „get" metode za svaki atribut klase za pristup i modifikaciju atributa objekta je česta praksa Java programming.

```java
public class Programmer extends Employee {

  private String language;

  public void setLanguage(String language) {
    this.language = language;
  }

  public void doWork() {
```

Ključna riječ extends indicira da ova klasa dodaje nešto – proširava Employee klasu

↳ Programmer klasa nasljeđuje sve od Employee waiting

↳ Unutar klase se implementiraju dodatne karakteristike



```java
    System.out.println("Programs in " + language + ".");
    System.out.println("Drinks a cup of coffee.");
  }
}
```

```java
public class SalesManager extends Employee {

  private int clients;

  public void setClients(int clients) {
    this.clients = clients;
  }

  public void doWork() {
    System.out.println("Makes a phone call to one of his " + clients
        + " clients.");
    System.out.println("Checks the sales of the last month.");
  }
}
```

```java
public class EmployeeTest {

  public static void main(String[] args) {

    Programmer p1 = new Programmer();
    p1.setName("Valery Miller");
    p1.setLanguage("FORTRAN");

    SalesManager sm1 = new SalesManager();
    sm1.setName("John Wood");
    sm1.setClients(16);

    // the programmer
    System.out.println(p1.getName() + " is working:");
    p1.doWork();

    // the sales manager
    System.out.println(sm1.getName() + " is working:");
    sm1.doWork();
  }
}
```

nova varijabla tipa Programmer i konstruiše novi Programmer objekat

dajemo ime zaposleniku

možemo setName metodu primijeniti na Programmer objektu iako nije implementirana u class Programmer, zato što ona nasljeđuje od Employee klase koja ima metodu setName

konstruiše se objekat SalesManager

Printamo šta rade

## Korištenje base classe – tip base klase se može koristiti za pohranjivanje objekta izvedene klase

```java
Programmer p1 = new Programmer();
p1.setName("Valery Miller");
p1.setLanguage("FORTRAN");

Employee e1 = p1;
System.out.println(e1.getName());
```

Employee → stores → Programmer

moguće zato što je svaki programer i zaposlenik

Ovo je ESENCIJALNO za efikasno OOP

# 5.2. Abstract class

Ako bi iznad definisane klase htjeli koristiti za predstavljanje zaposlenike velike kompanije. => kompanija može imati veliki broj programera i menadžera prodaje

⇓

Bilo bi veoma pogodno da se zaposlenicima upravlja na jedinstven način umjesto da se tretiraju na različit način kao iznad

Npr. možemo zaposlenike pohranjivat u niz i onda koristimo for petlju za printanje onog što zaposlenici rade.

```
1  public class EmployeeTest {
2
3    public static void main(String[] args) {
4
5      Programmer p1 = new Programmer();
6      p1.setName("Valery Miller");
7      p1.setLanguage("FORTRAN");
8
9      SalesManager sm1 = new SalesManager();
10     sm1.setName("John Wood");
11     sm1.setClients(16);
12
13     Employee[] employees = new Employee[2];
14     employees[0] = p1;
15     employees[1] = sm1;
16
17     for (int i = 0; i < employees.length; i++) {
18       System.out.println(employees[i].getName() + " is working:");
19       employees[i].doWork();
20       System.out.println();
21     }
```

pravimo niz zaposlenika

} dodajemo u niz

process the loop

okay jer je getName definisana u Employee klasu

Complaint → Employee doesn't have a doWork method

→ Pristup ima smisla → svi zaposlenici rade, metode su već implementirane.

Moramo da implementiramo doWork metodu u Employee klasi !! Kako, ne možemo u superklasi (početnoj klasi) znati šta rade članovi u subklasama (zaposlenici)?

Rješenje ⟹ uvođenje doWork metode u Employee klasi i deklarisati je kao abstract.

„Abstract" metoda samo specifiše da objekat ima metodu, ali je ne implementira.

↳ Implementacija metode se delegira izvedenim klasama, tj. metode se implementiraju u subklasama!

Klasa sa abstract metodom se također zove abstract class.

↳ NE MOGU SE KREIRATI OBJEKTI ABSTRACT KLASA

U class dyagramu, abstract klasu metodu možemo označiti ili italic slovima ili s {abstract}

| Employee {abstract} |
| --- |
| - name : String |
| + setName(name : String) : void<br>+ getName() : String<br>+ doWork() : void {abstract} |

Nema promjena u subklasama jer one već imaju doWork

U Java-i -> keyword je abstract

```java
public abstract class Employee {

  private String name;

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

  public String getName() {
    return name;
  }

  public abstract void doWork();
}
```

Izlaz ovog programa da su nam jasno govori o Java object variables polymorphic (isti objekat/metoda 9 drugo značenje)

Razmotrimo sada polymorphism.

```
1  Employee[] employees = new Employee[2];
2  employees[0] = p1;   ──→ } pohranjujemo Programmer i SalesManager objekte
3  employees[1] = sm1;  ──→ } u niz tipa Employee
4
5  for (int i = 0; i < employees.length; i++) {
6    System.out.println(employees[i].getName() + " is working:");
7    employees[i].doWork();
8    System.out.println();
9  }
```

Varijabla tipa Employee se može odnositi na objekat tipa Employee ili na bilo koju subklasu tipa Employee (kao Programmer ili SalesManager)

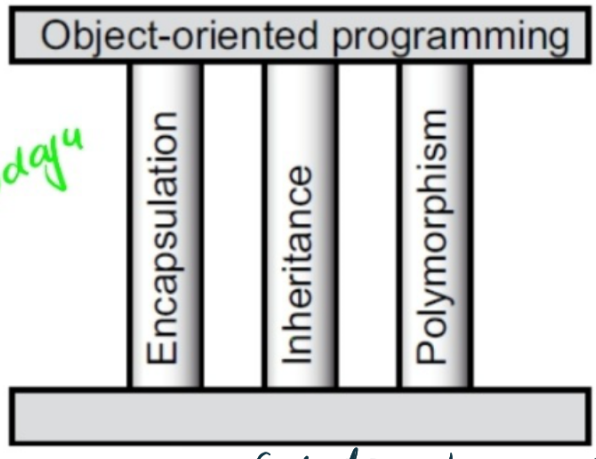
Vidimo da u 7 liniji se metoda doWork poziva na objekat tipa Employee.

↳ Ako posmatramo izlaz, vidimo da se pravilna metoda implementirala u izvedenim klasama
↳ i.e. metoda ima isti naziv, ali za različit objekat ima različitu strukturu/značenje

⇓

Ta osobina predstavlja POLYMORPHISM.

# 5.2.2. Tri stuba OOP-a




klase proširuju druge klase i dodaju im neke opcije

→ bundling of data, along with methods that operate on that data into a single unit

**Encapsulation** Objects hide internal representation

**Inheritance** Classes can extend other classes and add functionality

**Polymorphism** Same types or methods can have different meanings in different contexts

# 5.2.3. Abstract classes in Finite Element Programming

**Material models:** A material model calculates for a given strain tensor the corresponding stress tensor. Conceptually, this computation is the same for a simple isotropic elastic material model as for a complex plastic material model. By introducing an abstract class `MaterialModel` all specific material models could be used in the same manner. Polymorphism delegates the stress calculation to the concrete class (e.g. `IsotropicElasticMM`) as needed.

**Element formulations:** An abstract class `Element` declares abstract methods to compute the element stiffness matrix $\mathbf{k}^e$ and the element load vector $\mathbf{r}^e$. The concrete subclasses (e.g. `Truss3D`) are then responsible for these computations.

**Cross sections:** One dimensional elements need to know the properties of the cross section such as the area or the moments of inertia. An abstract class `CrossSection` declares the corresponding methods which are implemented in the concrete subclasses like `RectangularCS`.

Mehanizam POLIMORFIZMA čini program extensible.

[redacted]. For instance, in the context of Finite Element programming you could introduce a new material model without having to change any code inside the element classes that use a material model. The element only has to know that the material model computes stresses. It does not have to know anything about how this is done.

# PRAKTIČAN PRIMJER: Sections (presjeci)

In order to establish a model of a structure consisting of one-dimensional members, we need classes that represent cross sections. For our purposes, a cross section should be able to compute the area and it should provide the vertexes of its outline for visualization. The outline is defined in two-dimensional space.

The functionality described above should be declared in an abstract class `CrossSection`.

(↳ površina, tjemena okvira radi vizualizacije outline-a definisanog u 2D.

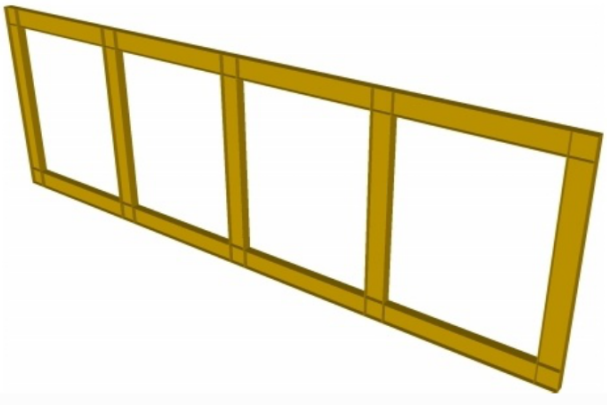
## Zadaci:

- Sketch a class diagram containing at least four different types of cross sections. Identify the relevant parameters for those classes (e.g. width and height for a rectangular cross section).
- Implement in Java the abstract base class `CrossSection` and at least one concrete class in addition to `RectangularCS` that extends `CrossSection`. Hint: a circular outline can be approximated by a polygon.
- Create another class to test your classes. Create different sections and compute the area.
- Use the `Extrusion` class from the `inf.v3d` package to visualize a simple structure. Look at the Javadoc of the `Extrusion` class to find out how to use it. Note that you can use the outline of one section for several `Extrusion` objects:
- Use the `Extrusion` class from the `inf.v3d` package to visualize a simple structure. Look at the Javadoc of the `Extrusion` class to find out how to use it. Note that you can use the outline of one section for several `Extrusion` objects:

```
1  RectangularCS cs1 = new RectangularCS(0.1, 0.1);
2
3  Extrusion e1 = new Extrusion(); e1.setOutline(cs1.getOutline());
4  e1.setPoint1(0,0,0); e1.setPoint2(0,1,0);
5
6  Extrusion e2 = new Extrusion(); e2.setOutline(cs1.getOutline());
7  e2.setPoint1(3,0,0); e2.setPoint2(3,1,0);
```

(Also, don't forget to add statements like `viewer.addObject3D(e1);` to show your 3D objects.)

The figure shows an example that uses I-shaped cross sections.



**CrossSection {abstract}**

- name: String

+ getName(): String
+ setName(name: String): void
+ getArea(): double {abstract}
+ getOutline(): double[][] {abstract}

common for all CrossSections

**RectangleCS**

- w: double
- h: double

+ RectangleCS(w: double, h: height)
+ getArea(): double
+ getOutline(): double[][]
+ getHeight(): double
+ getWidth(): double

**CircleCS**

- r: radius

+ CircleCS(r: radius)
+ getArea(): double
+ getOutline(): double[][]
+ getRadius(): double

**HProfileCS**

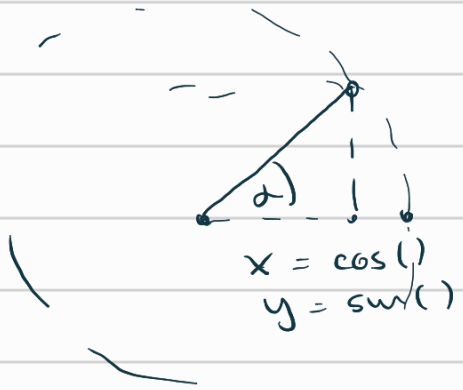
- h: double
- w: double
- t: double

+ HProfileCS(h: double, w: double, t: double)
+ getArea(): double
+ getOutline(): double[][]
+ getHeight(): double
+ getWidth(): double
+ getThickness(): double

**UProfileCS**

- h: double
- w: double
- t: double

+ UProfileCS(h: double, w: double, t: double)
+ getArea(): double
+ getOutline(): double[][]
+ getHeight(): double
+ getWidth(): double
+ getThickness(): double



My code:

```java
package chapter6;

public abstract class CrossSection {

    private String name;

    public CrossSection () { }

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

    public String getName () {
        return this.name;
    }
    public abstract double  getArea();

    public abstract double [][] getOutline();

}
```

```java
package chapter6;

public class RectangularCS extends CrossSection {

    private double w;
    private double h;

    public RectangularCS ( double width, double height) {
        this.w=width;
        this.h=height;}

    public double getArea () {
        return w*h;}

    public double [] [] getOutline () {
        double [] [] outline = new double [4] [2];
        outline [0][0] = - w / 2.0;
        outline [0][1] = -h/2.0;
        outline [1][0] =  w / 2.0;
        outline [1][1] = -h/2.0;
        outline [2][0] =  w / 2.0;
        outline [2][1] =  h/2.0;
        outline [3][0] = - w / 2.0;
        outline [3][1] =  h / 2.0;
        return outline;
    }

    public double getHeight () {
        return h;
    }

    public double getWidth() {
        return w;
    }
}
```

```java
package chapter6;

public class CircleCS extends CrossSection {

    private double r;
    private static int N=20;

    public CircleCS (double r) {
        this.r=r;
    }
    public double getArea() {
        return Math.pow(r, 2)* Math.PI/4;
    }

    public double[][] getOutline() {

        double [] [] outline = new double [N][2];
        double dphi = 2*Math.PI / (N-1);

        for (int i=0; i<N; i++) {
                    outline[i][0] = r * Math.cos(i*dphi);
                    outline [i][1] = r* Math.sin(i*dphi);
        }

        return outline;
    }

    public double getRadius () {
        return r;
    }
}
```

```java
package chapter6;

import inf.v3d.obj.Extrusion;
import inf.v3d.view.*;

public class CrossSectionDemoProgram {

    public static void main (String [] args) {

        RectangularCS r = new RectangularCS (1,1);
        CrossSection r1 = r;

        double [] [] outline = r1.getOutline();

        Extrusion e1 = new Extrusion (); e1.setOutline(outline);
        e1.setPoint1(0, 0, 0);
        e1.setPoint2(0, 1, 0);

        CircleCS c = new CircleCS (0.2);
        CrossSection c1 = c;
        double [] [] outline1 = c1.getOutline();

        Extrusion e2 = new Extrusion (); e2.setOutline(outline1);
        e2.setColor("blue");
        e2.setPoint1(0,0,0);
        e2.setPoint2(0,1,0);

        Viewer v = new Viewer ();
        v.addObject3D(e1);
        v.addObject3D(e2);
        v.setVisible(true);

        System.out.println("Area of figure 1 is " + c1.getArea());
        System.out.println("Area of figure 2 is " + r1.getArea());
    }
```

# 5.4. Interfaces

U Java-i, **interface** daju način za specificiranje skupa metoda bez davanja ikakve implementacije.

↓

Klasa koja obezbjeđuje metode navedene u interface-u kaže se da **implementira** taj interface.

Dakle, možemo reći da su interface-i slični klasama koje imaju samo apstraktne metode. → Razlika je samo što klasa može imati samo jednu superklasu, ali može implementirati bilo koji broj interface-a.

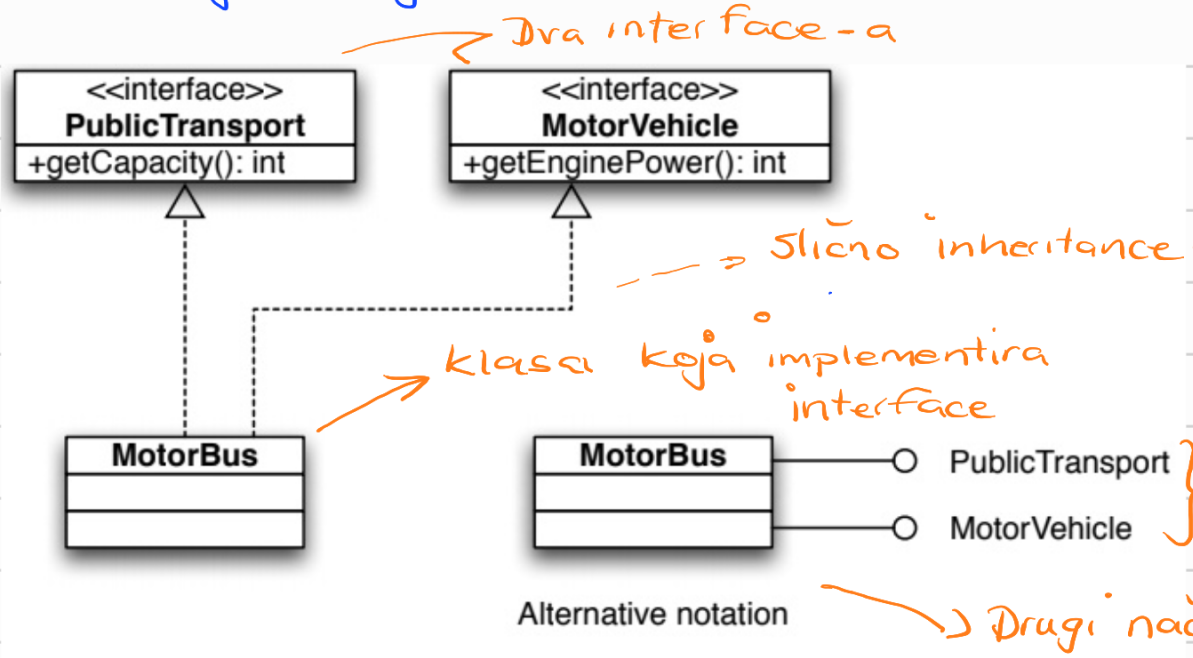



**Figure 5.4:** Some sorts of vehicles

```
1 public interface PublicTransport {
2   public int getCapacity();
3 }
```

Interface se deklariše s ključnom riječju "interface"

```
1 public interface MotorVehicle {
2   public double getEnginePower();
3 }
```

samo se deklarišu metode (bez koda)

```java
public class MotorBus implements PublicTransport, MotorVehicle {

  public int getCapacity() {
    return 30;
  }

  public double getEnginePower() {
    return 154.2;
  }
}
```

Da bi naglasili da klasa implementira interface koristimo ključnu riječ implements

Važan problem kod interface-a je da svaki interface definiše tip podataka koji se može koristiti za deklarisanje varijabli.

```java
  public void printCapacity(PublicTransport p) {
    System.out.println(p.getCapacity());
  }

  public void printEnginePower(MotorVehicle m) {
    System.out.println(m.getEnginePower());
  }
```

Definisano unutar implementirajuće klase

The most difficult thing about interfaces is to explain why we need them. For me, the most important issue is that interfaces introduce a higher level of abstraction to a software project that improves software quality. In some sense, interfaces isolate areas of high complexity and thus help to split a software in smaller independent units.

izoliraju područja velike kompleksnosti i na taj način omogućavaju podjelu softvera na manje nezavisne jedinice