Objektno usmerjeno programiranje

Izrazoslovje OOP

“Razred” pomeni kategorijo stvari

- Ime razreda lahko v Javi uporabimo kot tip polja ali lokalne spremenljivke ali kot povratni tip funkcije (metode)

“Objekt” pomeni konkretno stvar, ki pripada nekemu razredu

- Temu pravimo tudi “instanca”

Primer: poglejmo naslednjo vrstico:

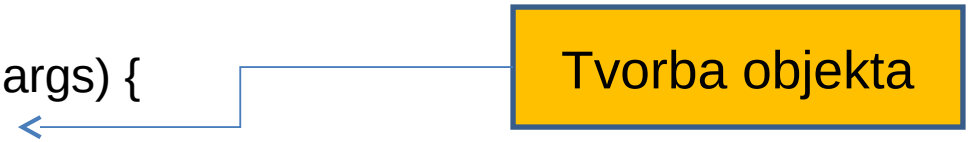
```
String  s1 = “Pozdrav";
```

Tu je String razred, spremenljivka s1 in vrednost “Pozdrav" sta objekta (ali “instanci iz razreda String”)

Spremenljivke instance (podatkovni člani objekta)

```
public class Ship1 {
    public double x, y, speed, direction;
    public String name;
}

public class Test1 {
    public static void main(String[ ] args) {
        Ship1 s1 = new Ship1();
         s1.x = 0.0;
         s1.y = 0.0;
         s1.speed = 1.0;
         s1.direction = 0.0; // East
         s1.name = "Ship1";
        Ship1 s2 = new Ship1();
         s2.x = 0.0;
         s2.y = 0.0;
         s2.speed = 2.0;
         s2.direction = 135.0; // Northwest
         s2.name = "Ship2";
...  // glej nadaljevanje
```



nadaljevanje

```
...
        s1.x = s1.x + s1.speed * Math.cos(s1.direction * Math.PI / 180.0);
        s1.y = s1.y + s1.speed * Math.sin(s1.direction * Math.PI / 180.0);
        s2.x = s2.x + s2.speed * Math.cos(s2.direction * Math.PI / 180.0);
        s2.y = s2.y + s2.speed * Math.sin(s2.direction * Math.PI / 180.0);
        System.out.println(s1.name + " is at (“ + s1.x + "," + s1.y + ").");
        System.out.println(s2.name + " is at ("+ s2.x + "," + s2.y + ").");
    }
}
```

Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).

Objekti in reference

Potem, ko definiramo razred, lahko zlahka deklariramo spremenljivko (referenco na objekt) za ta razred

```
Ship s1, s2;
Point start;
Color blue;
```

Referenče objektov so v začetku “null”
Vrednost “null” ne smemo enačiti z vrednostjo “nič”
Primitivnih podatkovnih tipov ne moremo “kastati” na nek objekt

Za eksplicitno tvorbo objekta, ki je naslovljen, potrebujemo operator “new”

```
ClassName variableName = new ClassName();
```

Metode

```
public class Ship2 {
      public double x=0.0, y=0.0, speed=1.0,
direction=0.0;
      public String name = "UnnamedShip";

      private double degreesToRadians(double degrees)
{
            return(degrees * Math.PI / 180.0);
       }
     public void move() {
          double angle = degreesToRadians(direction);
          x = x + speed * Math.cos(angle);
          y = y + speed * Math.sin(angle);
     }
     public void printLocation() {
          System.out.println(name + " is at (“ + x + "," + y +
```

Metode - nadaljevanje

```
public class Test2 {
    public static void main(String[] args) {
        Ship2 s1 = new Ship2();
        s1.name = "Ship1";
        Ship2 s2 = new Ship2();
        s2.direction = 135.0; // Northwest
        s2.speed = 2.0;
        s2.name = "Ship2";
        s1.move();
        s2.move();
        s1.printLocation();
        s2.printLocation();
    }
}
```

Izpis

Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).

Statične metode

Statične metode kličemo preko imena razreda:

```
ClassName.functionName(arguments);
```

Primer: razred Math ima statično metodo cos, ki pričakuje kot argument podatek tipa double. Tako lahko pokličemo

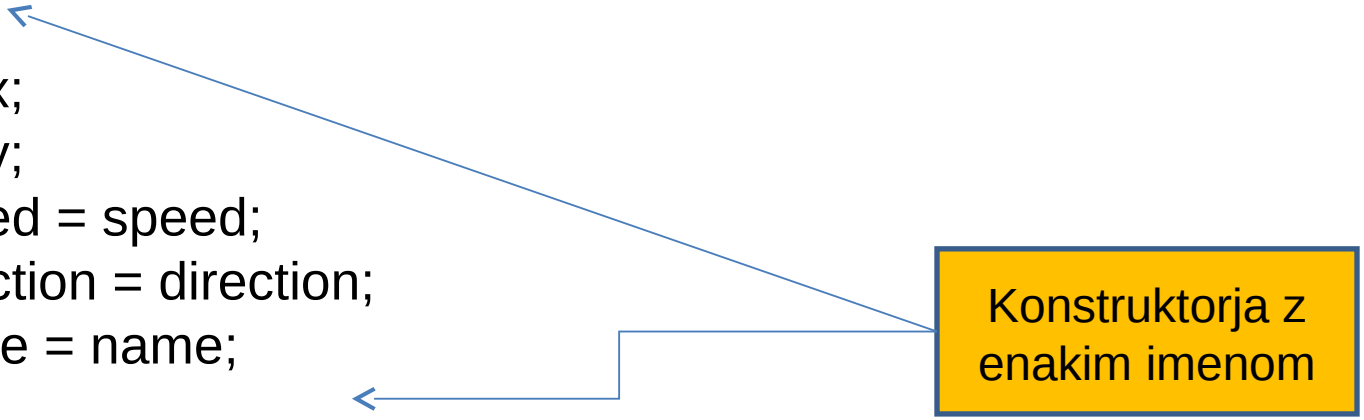
```
Math.cos(3.5)
```

ne da bi imeli kakšen objekt (instanco) iz razreda Math.

Opomba za metodo “main”
Sistem pokliče metodo main, ne da bi prej tvoril objekt. Zato lahko ta metoda direktno (brez predhodne tvorbe objekta) kliče le statične metode

Večkratno definiranje metod (overloading)

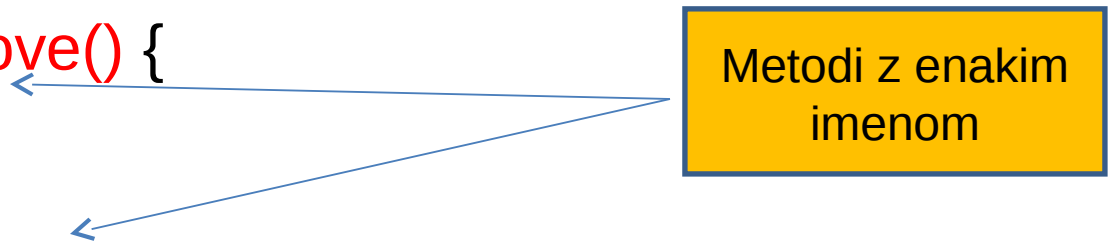
```
public class Ship4 {
    public double x=0.0, y=0.0, speed=1.0, direction=0.0;
    public String name;
    public Ship4(double x, double y,  double speed, double direction, String
name) {
            this.x = x;
            this.y = y;
            this.speed = speed;
            this.direction = direction;
            this.name = name;
    }
    public Ship4(String name) {
          this.name = name;
    }
    private double degreesToRadians(double degrees) {
        return(degrees * Math.PI / 180.0);
    }
```



(se nadaljuje)

Večkratno definiranje (nadaljevanje)

```
...
    public void move() {
        move(1);
    }
    public void move(int steps) {
        double angle = degreesToRadians(direction);
        x = x + (double)steps * speed * Math.cos(angle);
        y = y + (double)steps * speed * Math.sin(angle);
    }
    public void printLocation() {
         System.out.println(name + " is at (“ + x + "," + y +
").");
    }
}
```



Večkratno prekrivanje metod (uporaba)

```
public class Test4 {
    public static void main(String[] args) {
        Ship4 s1 = new Ship4("Ship1");
        Ship4 s2 = new Ship4(0.0, 0.0, 2.0, 135.0,
"Ship2");
        s1.move();
        s2.move(3);
        s1.printLocation();
        s2.printLocation();
    }
}
```

Ship1 is at (1.0,0.0).
Ship2 is at (-4.24264...,4.24264...).

“getters – setters”

```
public class Ship {
    private double x=0.0, y=0.0, speed=1.0, direction=0.0;
    private String name;
    …
    /** Get current X location. */
    public double getX() {
        return(x);
    }
    /** Set current X location. */
     public void setX(double x) {
       this.x = x;
   }
```

getter je metoda, ki vrne neko lastnost.

setter je metoda, ki postavi neko lastnost na podano vrednost.

Dedovanje

```
public class Speedboat extends Ship {
    private String color = "red";
    public Speedboat(String name) {
        super(name);
        setSpeed(20);
    }
    public Speedboat(double x, double y, double speed, double
direction,
            String name, String color) {
        super(x, y, speed, direction, name);
        setColor(color);
    }
    public void printLocation() {
        System.out.print(getColor().toUpperCase() + " ");
        super.printLocation();
    }
...
```

Uporaba izpeljanega in dedovanega razreda

```
public class SpeedboatTest {
    public static void main(String[] args) {
        Speedboat s1 = new Speedboat("Speedboat1");
        Speedboat s2 = new Speedboat(0.0, 0.0, 2.0, 135.0,
              "Speedboat2", "blue");
         Ship s3 = new Ship (0.0, 0.0, 2.0, 135.0, "Ship1");
        s1.move();
        s2.move();
        s3.move();
        s1.printLocation();
        s2.printLocation();
        s3.printLocation();
    }
}
```

```
RED Speedboat1 is at (20,0).
BLUE Speedboat2 is at (-1.41421,1.41421).
Ship1 is at (-1.41421,1.41421).
```

Kaj kaže prejšnji primer?

- Kako definiramo podrazrede (izpeljane razrede)
- Kako uporabljamo dedovane metode
- Uporaba super(…) za podedovane konstruktorje (le, če konstruktor btez argumentov ni uporaben)
- Uporaba super.nekaMetoda(…) z podedovane metode (le, če je kon

Dedovani konstruktorji in super

Ko tvorimo instanco (objekt) nekega podrazredam bo sistem avtomatično najprej poklical konstruktor nadrazreda

- Privzeto bo poklical nadrazredov konstruktor brez argumentov
- Če hočemo poklicati kakšen drug starševski konstruktor, ga pokličemo s super(*args)*
- Če v konstruktorju podrazreda kličemo super(…) , mora to biti prvi stavek v konstruktorju

Redeklarirane metode in super.metoda()

Če razred definira metodo z enakim imenom, tipom povratka in argumentov kot jo ima nadrazred, tedaj ta metoda prekrije metodo nadrazreda

Prekrijemo (redeklaracija) lahko le metode, ki niso statične

Če imamo lokalno definirano metodo in podedovano metodo z istim imenom in argumenti, lahko pokličemo podedovano metodo tako:

super.imeMetode(...)

Zaporedna uporaba predpon super (super.super.imemetode) ni dovoljena.

Abstraktni razredi

Za abstraktne razrede ne moremo tvoriti instanc (ne moremo klicati “new”).

Abstraktni razredi dovolijo deklaracijo razredov, ki definirajo le del implementacije, podrobnosti pa morajo podati podrazredi

Razred je abstrakten, če vsaj eni metodi razreda manjka implementacija

Abstraktna metoda nima implementacije (pri C++ to imenujemo čista virtualna funkcija)

Vsak razred, ki ima vsaj eno abstraktno metodo, moramo deklarirati kot abstrakten

Če podrazred prekrije vse abstraktne metode nadrazreda, lahko tvorimo instance tega podrazreda

Abstrakten razred ima lahko spremenljivke in polno implementirane metode– Any subclass can override an inherited concrete method

Abstraktne razrede lahko naslavljamo, čeprav ne moremo tvoriti

Abstraktni razredi: primer

```
public abstract class ThreeDShape {
    public abstract void drawShape(Graphics g);
    public abstract void resize(double scale);
}
ThreeDShape s1 = new Sphere(...);
ThreeDShape[] shapes = new ThreeDShape[20];
shapes[0] = new Cube(...);
```

Polje tipa ThreeDShape je precej uporabno

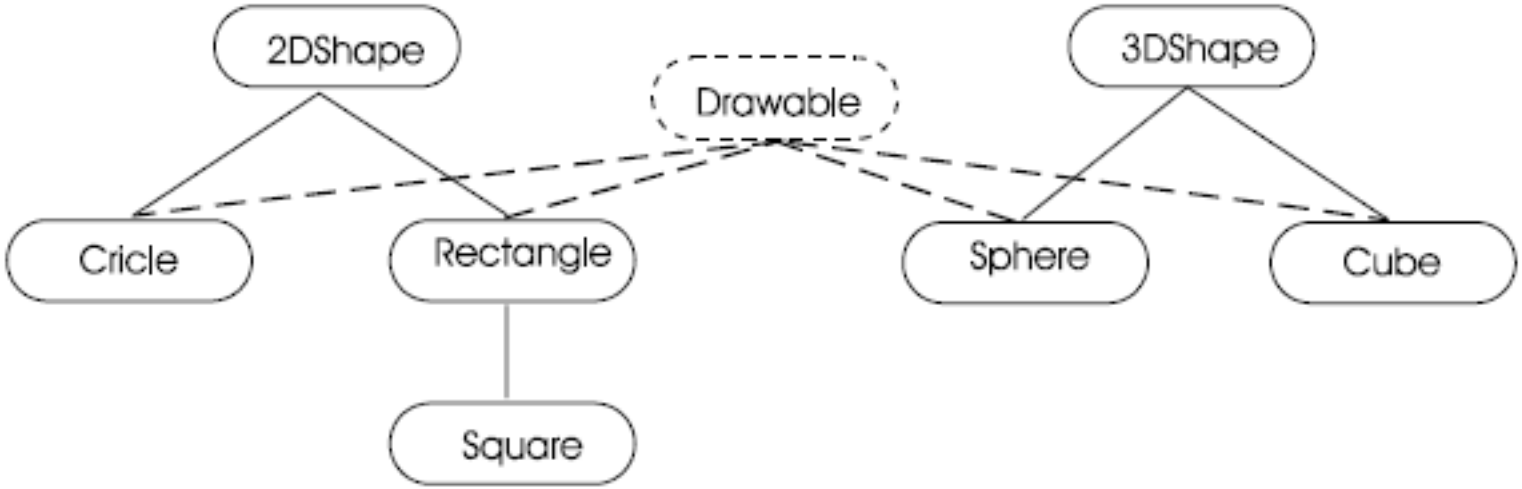
Polje v resnici vsebuje le instance iz podrazredov ThreeDShape

V zanki lahko preletavamo polje in kličemo metodi drawShape inresize

Garantiramo, da imajo podrazred Cube (in podobni) omenjene metode

Vmesniki (interfaces)

Vmesniki definirajo javanski tip, ki vsebuje le konstante in abstraktne metode
Vmesnik ne implementira nobene metode, pač pa vsili to vsakemu razredu, ki uporablja tak vmesnik
Razred, ki uporabi (implements) vmesnik, mora podati definicije vseh metod, ali pa mora tudi sam biti deklariran kot abstrakten



Vmesniki

- All methods in an interface are implicitly abstract and the keyword abstract is not required in a method declaration
- Data fields in an interface are implicitly static final (constants)
- All data fields and methods in an interface are implicitly public

```
public interface Interface1 {
    DataType CONSTANT1 = value1;
    DataType CONSTANT2 = value2;
    ReturnType1 method1(ArgType1 arg);
    ReturnType2 method2(ArgType2 arg);
}
```

Uporaba več vmesnikov

– Vmesnike lahko izpeljemo (extends) tudi iz drugih vmesnikov (dedovanje), kar pripelje do pojmov podvmesniki in nadvmesniki.
– V razliko od razredov lahko vmesnike izpeljemo (iz več kot enega vmesnika

```
public interface Displayable extends Drawable,
Printable {
    // dodatne konstante in abstraktne metode
    ...
}
```

Uporaba več vmesnikov v razredih: – Interfaces provide a form of multiple inheritance because a class can implement more than one interface at a time

```
public class Circle extends TwoDShape
implements Drawable, Printable {
...
}
```

Polimorfizem

“Polymorphic” dobesedno pomeni “ima več oblik” , pri objektno usmerjenem programiranju pa to pomeni “ima več obnašanj”

Polimorfična metoda izvaja različne akcije, odvisno od objekta, ki je naslovljen

To poznamo tudi kot povezovanje v času izvajanja (*late binding, run-time binding)*

V praksi uporabljamo polimorfizem skupaj z naslavljanjem polj pri preletavanju kolekcij objektov in dostopu do polimorfičnih metod posameznih objektov

Polimorfizem: primer

```
public class PolymorphismTest {
    public static void main(String[] args) {
        Ship[] ships = new Ship[3];
        ships[0] = new Ship(0.0, 0.0, 2.0, 135.0, "Ship1");
        ships[1] = new Speedboat("Speedboat1");
        ships[2] = new Speedboat(0.0, 0.0, 2.0, 135.0,
"Speedboat2",
             "blue");
        for(int i=0; i<ships.length ; i++) {
            ships[i].move();
            ships[i].printLocation();
        }
    }
}
```

RED Speedboat1 is at (20,0).
BLUE Speedboat2 is at (-1.41421,1.41421).
Ship1 is at (-1.41421,1.41421).