

# Programozási nyelvek II.: JAVA, 11. gyakorlat

2017. november 27-december 1.

# A 11. gyakorlat tematikája

- Öröklődés
  - Osztályhierarchia
  - Az Object osztály
  - Láthatósági módosítók
  - A super pszeudováltozó
  - Felüldefiniálás
- Kompozíció
- Aggregáció

- Interfészek
  - Interfészek definíciója
  - Öröklődés
  - Interfész megvalósítása
  - Az interfész mint típus
    - változódeklarációkban
    - formális paraméterek specifikációjában
  - Generikus interfészek

- Osztályok származtatása és kiterjesztése (formailag: `extends`, pl. `class Circle extends Shape`, is-a reláció)
- Osztály kiegészítése új tagokkal (példányváltozókkal, metódusokkal)
- Szülőosztály, gyermekosztály
- Ős: a szülő reflexív, tranzitív lezártja
- Leszármazott: a gyermek reflexív, tranzitív lezártja
- A gyermek a szülő tagjaival is rendelkezik (az öröklés révén) + ki is terjesztheti azokat

Előnyei:

- Kód-újrafelhasználás (a kód redundanciája csökken)
- Olvashatóság + karbantarthatóság megkönnyítése
- Öröklődés (mint tervezésszintű fogalom, l. később a kompozíció és az aggregáció fogalmát): kódmegosztás, altípusképzés

mint altípusképzés (egy parciális rendezés)

- A gyermek típusa a szülő típusának egy altípusa, mert
  - a gyermek rendelkezik a szülő összes attribútumával
  - a gyermek minden eseményre reagálni tud, amelyre a szülője is
- Következmény: minden olyan helyzetben, amikor a szülőt használhatjuk, használhatjuk a gyermeket is

- Az öröklődési reláció gráfként megadva
- Egyszeres öröklődés esetén ez egy (irányított) erdő
- Javában az `Object` minden osztály közös őse (*univerzális őosztály*)  
⇒ az öröklődési gráf egy fa
- Ha nem adunk meg `extends`-et, akkor implicit `extends` van

- Predefinit, a `java.lang`-ban van definiálva
- Az `Object` osztály olyan metódusok definícióit tartalmazza, amelyekkel minden objektumnak rendelkeznie kell, pl.
  - `boolean equals(Object obj)`,
  - `String toString()`,
  - `int hashCode()`

(<http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>)



# Láthatósági módosítók (emlékeztető)

Minden adattagra és minden metódusra pontosan egy hozzáférési kategória vonatkozhat, ezért az alább megadott módosítószavak közül pontosan egyet lehet használni minden egyes tag és metódus esetében. A hozzáférési kategóriák (módosítószavak) a következők:

- Félnyilvános: ha nem írunk semmit
  - azonos csomagban definiált osztályok (objektumai)
- Nyilvános: `public`
  - különböző csomagokban definiált osztályok is elérik
  - pl. a főprogramnak is ilyennek kell lennie, hogy futtatható legyen:  
`public static void main(String[] args)`
- Privát: `private`
  - csak az osztálydefiníción belül érhető el
  - az osztály minden objektuma
- Védett: `protected`
  - a félnyilvános kategória kiterjesztése: azonos csomagban lévő, plusz a leszármazottak

# Láthatósági reláció (emlékeztető)

`public`  $\supseteq$  `protected`  $\supseteq$  `package-private`  $\supseteq$  `private`

| Módosító                               | osztály | csomag | leszármazott | mindenki |
|--|---------|--------|--------------|----------|
| <code>public</code>                    | igen    | igen   | igen         | igen     |
| <code>protected</code>                 | igen    | igen   | igen         | nem      |
| nincs ( <code>package private</code> ) | igen    | igen   | nem          | nem      |
| <code>private</code>                   | igen    | nem    | nem          | nem      |

# A super pszeudováltozó

- A konstruktor nem örökölhető, de meghívható `super` névvel (a szülőosztálybeli konstruktor a legelső sorban)
- Ha nem hívunk meg egy konstruktorban egy másikat, akkor implicit módon egy paraméter nélküli `super()`; hívás történik
- Ha a szülőnek nincs paraméter nélküli konstruktora, akkor fordítási hibát kapunk
- A `super` megelőzi az osztálydefinícióban szereplő példányváltozó inicializálásokat
- Egy `protected` konstruktort `new`-val csak a csomagon belül hívhatunk meg, `super`-ként pedig csak a csomagon kívül

- A gyermek osztályban bizonyos eseményekre másképp kell / lehet reagálni, mint a szülőosztályban (ősosztályban)  $\implies$  felüldefiniálás
- Örökölt metódushoz új definíciót rendelünk
- A felüldefiniált metódus elérés: `super.xxx()`

# A felüldefiniálás szabályai

- A szignatúra megegyezik (ha a szignatúra különböző: túlterhelés)
- A visszatérési típus megegyezik
- A hozzáférési kategória: nem szűkíthető ( $\text{private} \subseteq \text{félnyilvános} \subseteq \text{protected} \subseteq \text{public}$ )
- Specifikált kiváltható kivételek: nem bővíthetők
- Ha a szignatúra ugyanaz, de az előbbi három feltétel nem teljesül, akkor fordítási hibát kapunk

# Az öröklődés szemléltetése egy példán

...az Alakzatok osztálya...

(Point.java, Circle.java, Rectangle.java, Shape.java, ShapeTest.java)

# Pontok osztálya (Point.java)

```
class Point {  
  
    public double x;  
    public double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    ...  
}
```

# Pontok osztálya (Point.java)

```
...  
public void setX(double x) {  
    this.x = x;  
}  
  
public void setY(double y) {  
    this.y = y;  
}  
  
public void move(double dx, double dy) {  
    x += dx;  
    y += dy;  
}  
...
```



# Pontok osztálya (Point.java)

```
...  
public double distance(Point p) {  
    return distance(this, p);  
}  
  
public static double distance(Point p1, Point p2) {  
    return Math.sqrt(Math.pow(p1.x - p2.x, 2) +  
        Math.pow(p1.y - p2.y, 2));  
}  
  
public String toString() {  
    return "(" + x + "," + y + ")";  
}  
  
}
```

# Alakzatok osztálya (Shape.java)

```
class Shape {  
  
    private Point center;  
    protected double area;  
    protected double circumference;  
  
    public Shape(double x, double y) {  
        center = new Point(0.0,0.0);  
        center.x = x;  
        center.y = y;  
    }  
  
    public Shape() {  
        center = new Point(0.0,0.0);  
    }  
    ...  
}
```

# Alakzatok osztálya (Shape.java)

```
...  
public Point getCenter() {  
    Point result = new Point(0.0,0.0);  
    result.x = this.center.x;  
    result.y = this.center.y;  
    return result;  
}  
  
public void setCenter(Point center) {  
    this.center.x = center.x;  
    this.center.y = center.y;  
}  
public double getArea() {  
    return area;  
}  
...
```

# Alakzatok osztálya (Shape.java)

```
...
public double getCircumference() {
    return circumference;
}

public void move(double dx, double dy) {
    center.move(dx, dy);
}

public void toCenterPoint(Point p) {
    center = p;
}

public String toString() {
    return "kozeppont=" + center.x
        + ", " + center.y + ";";
}
}
```

# Körök osztálya (Circle.java)

```
class Circle extends Shape{

    private double radius;

    public Circle(Point p, double radius) {
        super(p.x,p.y);
        this.radius = radius;
    }

    public Circle(double x, double y) {
        super(x,y);
    }

    public Circle() {
        super();
    }

    ...
}
```

# Körök osztálya (Circle.java)

```
...  
public double getRadius() {  
    return radius;  
}  
  
public void setRadius(double r) {  
    if (r < 0.0) r = 0.0;  
    radius = r;  
}  
  
public void enlarge(double factor) {  
    radius *= factor;  
}  
...
```

# Körök osztálya (Circle.java)

```
...  
public double getArea() {  
    super.area = Math.PI*radius*radius;  
    return super.getArea();  
}  
public double getCircumference() {  
    super.circumference = 2*Math.PI*radius;  
    return super.getCircumference();  
}  
  
public boolean liesWithin(Point p, double delta) {  
    return Math.abs(super.getCenter().distance(p)  
        - radius) < delta;  
}  
...
```

# Körök osztálya (Circle.java)

```
...
public boolean isInside(Point p) {
    return (Math.pow(p.x - super.getCenter().getX(),2)
        +Math.pow(p.y - super.getCenter().getY(),2)
        <= Math.pow(radius,2));
}

public String toString() {
    return super.toString() + ", sugar=" + radius;
}
}
```



# Téglalapok osztálya (Rectangle.java)

```
class Rectangle extends Shape{

    private double length;
    private double width;

    public Rectangle(Point p, double length, double width) {
        super(p.x,p.y);
        this.length = length;
        this.width = width;
    }

    public Rectangle(double x, double y) {
        super(x,y);
    }

    public Rectangle() {
        super();
    }

    ...
}
```

# Téglalapok osztálya (Rectangle.java)

```
...  
public double getLength() {  
    return length;  
}  
  
public void setLength(double l) {  
    if (l < 0) {  
        l = -l;  
    }  
    length = l;  
}  
  
public double getWidth() {  
    return width;  
}  
...
```

# Téglalapok osztálya (Rectangle.java)

```
...  
public void setWidth(double w) {  
    if (w < 0) {  
        w = -w;  
    }  
    width = w;  
}  
  
public double getArea() {  
    super.area = width * length;  
    return super.getArea();  
}  
...
```

# Téglalapok osztálya (Rectangle.java)

```
...
public double getCircumference() {
    super.circumference = 2*(width + length);
    return super.getCircumference();
}

public String toString() {
    return super.toString() + ",_hosszusag_=" +
        + length + ",_szelesseg_=" + width;
}
}
```

# A főprogram (ShapeTest.java)

```
class ShapeTest {  
  
    public static void main(String[] args) {  
        ...  
    }  
}
```

- Az objektumtípusok közötti kapcsolat (asszociáció) egyik fajtája
- A két osztály között nem öröklődési kapcsolat van, hanem csupán az egyik osztály felhasználja a másik osztályt a definíciójához, annak szerves része lesz, anélkül nem létezhet

```
class Person {  
  
    private String name; /* String "is part of" Person */  
    private Integer age; /* Integer "is part of" Person */  
  
    public static Person make(String name, Integer age) {  
        return ((age > 0 && !name.isEmpty()) ?  
            new Person(name, age) : null);  
    }  
  
    private Person(String name, Integer age) {  
        this.name = new String(name);  
        this.age = new Integer(age);  
    }  
  
    ...  
}
```

```
...  
public String getName() {  
    return new String(name);  
}  
  
public Integer getAge() {  
    return new Integer(age);  
}  
  
public String toString() {  
    return String.format  
        ("Person { name = %s, age = %d }", name, age);  
}  
}
```



# Aggregáció

```
class Person {
    private String name;
    /* String "is part of" Person */
    private Integer age;
    /* Integer "is part of" Person */
    private ArrayList<Person> friends;
    /* Person "has" Persons */

    public static Person make(String name, Integer age) {
        return ((age > 0 && !name.isEmpty()) ?
            new Person(name, age) : null);
    }

    private Person(String name, Integer age) {
        this.name = new String(name);
        this.age = new Integer(age);
        this.friends = new ArrayList<Person>();
    }

    ...
}
```

# Aggregáció

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

public Integer getAge() {
    return new Integer(age);
}

public void addFriend(Person friend) {
    friends.add(friend);
}

public String toString() {
    return String.format
        ("Person { name = %s, age = %d, friends = %s }",
         name, age, friends.toString());
}
}
```

- referenciatípus
- ~ absztrakt osztályok
- ~ típusspecifikáció
- többszörös öröklődés (altípus reláció)
- Elnevezési konvenció: nevük gyakran a -ható, -hető képzővel végződik (azaz -able), pl. Comparable, Runnable (Futtatható)

- metódusdeklarációk (absztract metódusok)
- `public final static` adattagok
- tagosztályok (taginterfészek)
- default metódusok (Java 8-tól)
- statikus metódusok (Java 8-tól)

```
interface Bicycle {  
    void changeCadence(int newValue);  
    void changeGear(int newValue);  
    void speedUp(int increment);  
    void applyBrakes(int decrement);  
}
```

# Az interfészt megvalósító osztály – példa

```
class BMXBicycle implements Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    ...  
}
```

## Az interfészt megvalósító osztály – példa

```
...
void applyBrakes(int decrement) {
    speed = speed - decrement;
}

void printStates() {
    System.out.println("cadence:" +
        cadence + "┘speed:" +
        speed + "┘gear:" + gear);
}
}
```

- Az interfészek nem példányosíthatók, előbb meg kell őket valósítani
- Az osztálydeklarációban szerepel az `implements` kulcsszó, amely után több interfész felsorolható, amelyeket az osztály megvalósít
- A metódusok megvalósítása `public` kell, hogy legyen
- A konstansok specifikációját nem kell megismételni



- Interfészek is kiterjeszthetik egymást: `extends` után azoknak az interfészeknek a listája, amelyektől az adott interfész örököl
- Többszörös öröklődés lehetséges
- Az osztályok megvalósíthatnak interfészeket
  - ugyanazt az interfészt többen is
  - ugyanaz az osztály többet is

- Öröklődés osztályok között
  - fa (egyszeres öröklődés, közös gyökér)
  - kódöröklés
- Öröklődés interfészek között
  - körmentes gráf (többszörös öröklődés, nincs közös gyökér)
  - specifikáció öröklése
- Megvalósítás osztályok és interfészek között
  - kapcsolat a két gráf között, továbbra is körmentes
  - specifikáció öröklése

- Pl. ha az I egy interfész, J az I egyik őse, az A osztály megvalósítja I-t, B leszármazottja az A-nak, akkor B megvalósítja J-t.

# Az interfész mint típus

- használható változódeklarációkban
- használható formális paraméterek specifikációjában

- egy interfész típusú változó: referencia, amely olyan objektumra mutathat, amely osztálya (közvetlenül vagy közvetve) megvalósítja az interfészt

```
I v = new A();
```

```
J w = new B();
```

# Az interfész mint típus – formális paraméterek specifikációjában

- egy interfész típusú formális paraméter: megadható egy olyan aktuális paraméter, amely egy objektum, és amely osztálya (közvetlenül vagy közvetve) megvalósítja az interfészt

```
void m(I p){...} m(new A());  
void n(J p){...} n(new B());
```

# Az interfész mint típus – formális paraméterek specifikációjában

- Ha egy változó (vagy formális paraméter) deklarált típusa (azaz statikus típusa) egy interfész, akkor
  - dinamikus típusa egy azt megvalósító osztály
  - a változóra olyan műveleteket használhatunk, amelyek az interfészben (közvetlenül vagy közvetve) definiálva vannak

Típussal paraméterezhető interfészek

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}
```



# Generikus interfészek

```
public class OrderedPair<K, V> implements Pair<K, V> {  
  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey()    { return key; }  
    public V getValue() { return value; }  
}
```

```
Pair<String, Integer> p1
    = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2
    = new OrderedPair<String, String>("hello", "world");
```

# Feladatok



## SzínesPont és SzínesKör osztály (ColouredPoint.java, ColouredCircle.java, ColouredShapeTest.java)

Készítsük el a SzínesPont osztályt a Pont osztály leszármazottjaként. Új tulajdonság: szín. Új műveletek: szín beállítása és lekérdezése. A szín attribútum legyen privát.

Készítsük el a SzínesKör osztályt a Kör osztály leszármazottjaként. Új műveletei: a szín beállítása és lekérdezése. A színes körök színét a középpontjuk színe határozza meg, amely nem közönséges pont, hanem színes pont.

Készítsünk főprogramot e két osztály tesztelésére! Oldjuk meg a feladatot csomagok nélkül és csomagokkal is (rakjuk a Kör és az Alakzat osztályt a `geo`, a SzínesKör osztályt `geo.coloured`, a Pont osztályt a `utils.basics`, a SzínesPont osztályt a `utils.basics.coloured`, a főprogramot pedig a `main` csomagba)!

## Stack<T> generikus interfész (Stack.java)

Készítsünk egy Stack<T> generikus interfészt, amely egy verem műveleteit definiálja (elem elhelyezése a verem tetején, elem kivétele a verem tetejéről, legfelső elem lekérdezése, iterátor létrehozása, méret lekérdezése, kivételek dobása hibák esetén)!

## ArrayStack<T>, LinkedStack<T> (ArrayStack.java, LinkedStack.java, StackTest.java)

Valósítsuk meg az ArrayStack<T> és a LinkedStack<T> osztályt, amelyek a Stack<T> interfész két különböző implementációját adják meg! Az ArrayStack<T> osztály egy tömbbel, a LinkedStack<T> osztály pedig egy láncolt listával ábrázolja a vermet. Készítsünk főprogramot, amely teszteli az egyes osztályokban definiált műveleteket!