



Programozási nyelvek 2

Öröklődés

Dr. Tiba Attila

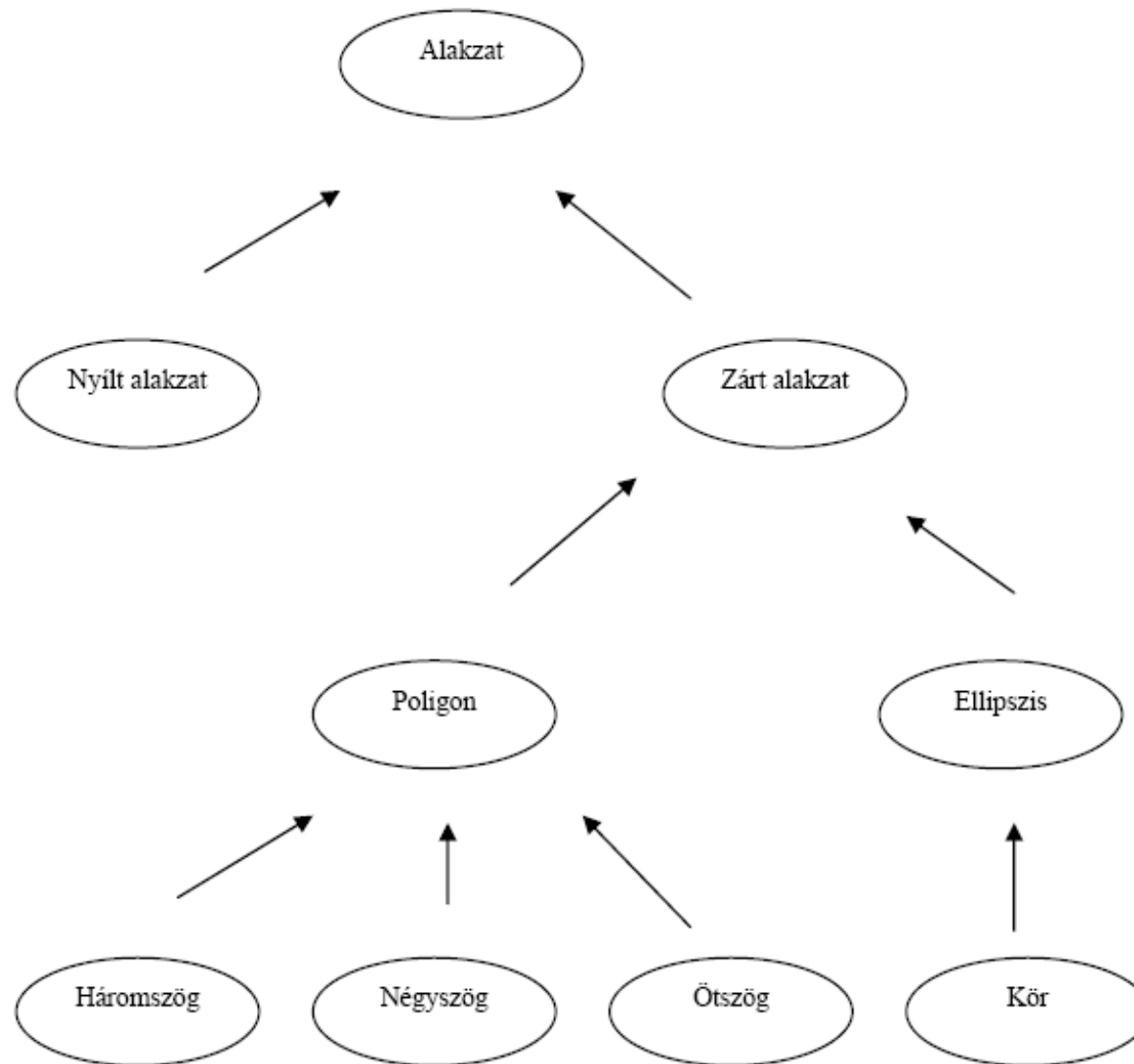
Objektum, öröklődés

- Az OO nyelvek az osztályok között egy aszimmetrikus kapcsolatot értelmeznek, melynek neve öröklődés.
- Az öröklődés az újrafelhasználhatóság eszköze.
- Az öröklődési viszonynál egy már létező szuperosztályhoz (szülő osztályhoz, alaposztályhoz) hozunk létre egy új osztályt, melynek elnevezése alosztály (gyermek osztály, származtatott osztály).
- Az öröklődés lényege, hogy az alosztály
 - átveszi (örökli) szuperosztályának minden (a bezárás által megengedett) attribútumát és módszerét,
 - ezeket azonnal fel is tudja használni,
 - új attribútumokat és módszereket definiálhat,
 - az átvett eszközöket átnevezheti,
 - az átvett neveket újradeklarálhatja,
 - megváltoztathatja a láthatósági viszonyokat,
 - a módszereket újraimplementálhatja.

Objektum, osztályhierarchia

- Egy alosztály lehet egy másik osztály szuperosztálya – így egy osztályhierarchia jön létre.
- Az osztályhierarchia egyszeres öröklődés esetén fa, többszörös öröklődés esetén aciklikus gráf.
- A többszörös öröklődési nyelvek osztályhierarchiájában is van azonban egy kitüntetett „gyökér” osztály, amelynek nincs szuperosztálya, és léteznek olyan „levél” osztályok, amelyeknek nincsenek alosztályai.
- A többszörös öröklődésnél gondot okozhat a különböző szuperosztályokban használt azonos nevek ütközése.

Objektum, öröklődési fa (példa)



Szülőosztály konstruktorai öröklődnek?

Nem, nem öröklődnek.

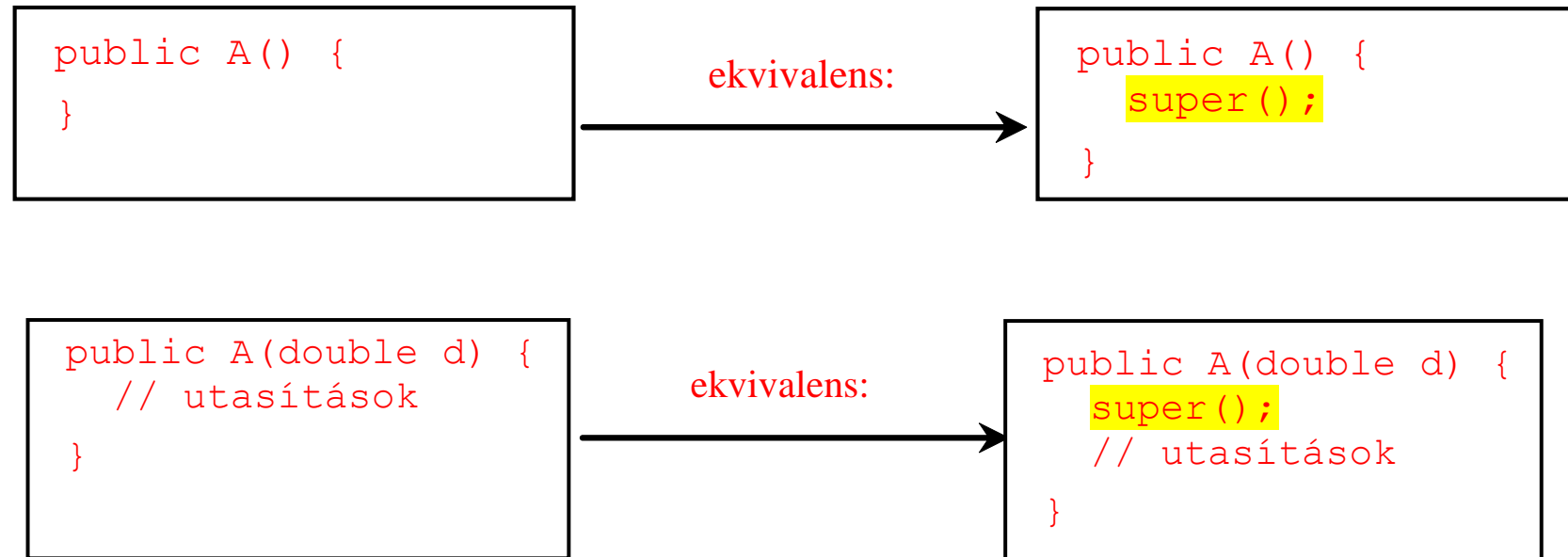
Explicit vagy implicit módon hívhatók meg.

Explicit meghívás a super kulcsszóval történik.

A konstruktort az osztály egy példányának elkészítéséhez használjuk. A tulajdonságoktól és metódusoktól eltérően a szülőosztály konstruktorai nem öröklődnek. Ezek csak a gyermekosztályok konstruktoraiból hívhatók a super kulcsszó segítségével. *Ha nem használjuk explicit módon a super kulcsszót, akkor a szülőosztály no-arg konstruktora kerül automatikusan meghívásra.*

Szülőosztály konstruktora mindig hívásra kerül

Egy konstruktor meghívhat egy túlterhelt konstruktort vagy a szülőosztályának konstruktorát. Ha egyik sincs explicit módon hívva, a fordító super() kódot illeszt első utasításként a konstruktorba. Például,



A `super` kulcsszó használata

A `super` kulcsszó annak az osztálynak a szülőosztályára hivatkozik, amelyikben a `super` kulcsszó megjelenik. A kulcsszó kétféleképpen használható:

- Szülőosztály konstruktorának meghívására
- Szülőosztály metódusának meghívására

FIGYELMEZTETÉS

A szülőosztály konstruktorának meghívásához használnunk kell a super kulcsszót.

A Java megköveteli, hogy a super kulcsszót használó utasítás elsőként jelenjen meg a konstruktorban.

Konstruktorok láncolása

Egy osztály példányának konstruálása az összes szülőosztály konstruktorát meghívja az öröklődési láncon keresztül. Ezt nevezzük a *konstruktorok láncolásának*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Végrehajtás követése

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

1. Indítás a main
metódustól

Végrehajtás követése

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Faculty
konstruktor hívása

Végrehajtás követése

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

3. Employee no-arg
konstruktorának hívása

Végrehajtás követése

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

4. Employee(String)
konstruktor hívása

Végrehajtás követése

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

5. Person() konstruktor hívása

Végrehajtás követése

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. println végrehajtása

Végrehajtás követése

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

7. println végrehajtása

Végrehajtás követése

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

8. println végrehajtása

Végrehajtás követése

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

9. println végrehajtása

Gyermekosztály deklarációja

A gyermekosztály kiterjeszti a szülőosztályból származó tulajdonságokat és metódusokat.

Továbbá:

- Felvehet új tulajdonságokat
- Felvehet új metódusokat
- Újrimplementálhatja a szülőosztály metódusait

Szülőosztály metódusainak hívása

Átírhatjuk a printCircle() metódust a Circle osztályban az alábbi módon:

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

Szülőosztály metódusainak felülírása

Egy gyermekosztály megörökli a szülő metódusait. Időnként a gyermekosztálynak felül kell írnia a szülőosztályból örökölt metódus implementációját.

```
public class Circle extends GeometricObject {  
    // Más metódusok figyelmen kívül hagyva  
  
    /** A GeometricObject osztály toString metódusának felülírása*/  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

MEGJEGYZÉS

Egy példányszintű metódus csak akkor írható felül, ha elérhető. Vagyis private metódus nem írható felül, mivel az osztályán kívül nem látható. Ha egy gyermekosztályban definiált metódus private a szülőosztályában, a két metódus teljesen független.

MEGJEGYZÉS

A példányszintű metódushoz hasonlóan egy statikus metódus is örökölhető. Egy statikus metódus azonban nem írható felül. Ha a szülőosztályban definiált statikus metódus újra lett definiálva egy gyermekosztályban, a szülőosztályban definiált metódus rejtett.

Objektum, túlterhelés

- Az OO nyelvek általában megengedik a módszernevek túlterhelését, azaz egy osztályon belül azonos nevű és természetesen eltérő implementációjú módszereket tudunk létrehozni.
- Ekkor természetesen a hivatkozások feloldásához a specifikációknak különbözniük kell a paraméterek számában, vagy azok típusában.

Felülírás vs. túlterhelés

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // Ez a metódus felülírja a B metódusát  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // Ez a metódus túlterheli a B metódusát  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

Objektum, kötés

- Egy alosztály az örökölt módszereket újrainplementálhatja.
- Ez azt jelenti, hogy különböző osztályokban azonos módszerspecifikációhoz különböző implementáció tartozik (polimorfizmus).
- Ezek után a kérdés az, hogy ha meghívunk egy objektumra egy ilyen módszert, akkor melyik implementáció fog lefutni.
- A választ egy nyelvi mechanizmus, a kötés adja meg.

Objektum, kötés

- Az OO nyelvek két fajta kötést ismernek:
 - Statikus kötés: már fordításkor eldől a kérdés. Ekkor a helyettesíthetőség nem játszik szerepet. A forrásszövegben megadott objektum deklaráció osztályának módszere fog lefutni minden esetben.
 - Dinamikus kötés: a kérdés csak futási időben dől el, a megoldás a helyettesíthetőségen alapul. Annak az objektumnak a példányosító osztályában megadott (vagy ha nem írta fölül, akkor az öröklött) implementáció fog lefutni, amelyik ténylegesen kezelésre kerül.

Objektum, kötés

- Tehát ha a `Polygon` osztályban van egy `Kerület` módszer, amelyet újraimplementálunk a `Négyszög` osztályban, akkor statikus kötés esetén, ha egy `poligon` objektumra meghívjuk, akkor a `Polygon` implementáció fut le akkor is, ha a futás közben egy `négyszög` objektum `kerületét` határozzuk meg.
- Dinamikus kötés esetén viszont az utóbbi esetben a `Négyszög` implementációja fog futni.
- Egyes OO nyelvek dinamikus kötést használnak, másokban mindkettő jelen van, az egyik alapértelmezett, a másikat a programozónak explicit módon kell beállítania.

Objektum, helyettesíthetőség

- Az öröklődésen alapul és az újrafelhasználhatóságnak egy jellegzetes megnyilvánulása a helyettesíthetőség.
- Egy leszármazott osztály példánya a program szövegében minden olyan helyen megjelenhet, ahol az előd osztály egy példánya (az utóbbi helyettesíthető az előbbivel).
- Egy konkrét `Kör` objektum egyben `Ellipszis` és `Zárt alakzat` is, tehát ezeknek az osztályoknak is objektuma.
- Viszont csak a `Kör` osztálynak a példánya.

Az Object osztály és metódusai

A Java minden osztálya a java.lang.Object osztályból származik. Ha egy osztály definiálásakor nincs öröklődés megadva, az osztály szülőosztálya az Object osztály lesz.

```
public class Circle {  
    ...  
}
```

Ekvivalens

```
public class Circle extends Object {  
    ...  
}
```

Object osztály toString() metódusa

A toString() metódus visszaadja az objektum egy szöveges reprezentációját. Az alapértelmezett implementáció a következőket tartalmazó sztringet adja vissza: annak az osztálynak a neve, aminek az objektum példánya; az @ szimbólum; az objektumot reprezentáló szám.

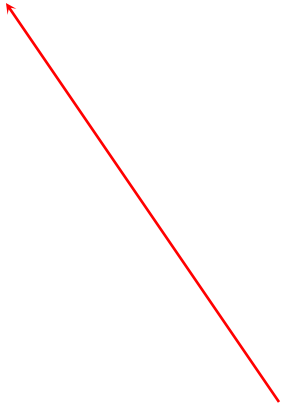
```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

A kód a következő sztringet eredményezi: Loan@15037e5. Ez az üzenet nem túl informatív. Emiatt célszerű felülírni a toString metódust, ami egy jobb leírását adja az egyes objektumoknak.

Objektumok átalakítása (casting)

Az átalakító (cast) operátort már használtuk egyszerű adattípusok konvertálására. Az átalakítás használható egy osztály objektumának más osztálytípusúvá alakítására is az öröklődési hierarchián belül.

```
Object o = new Student(); // Implicit átalakítás
```



Az Object o = new Student() utasítás (implicit átalakítás) szabályos, mivel a Student egy példánya az Object egy példánya is.

Miért szükséges az átalakítás?

Tegyük fel, hogy egy o objektumreferenciát akarunk egy Student típusú változóhoz a következő utasítással:

```
Student b = o;
```

Fordítási hiba történik. Miért működik az **Object o = new Student()** utasítás, és miért nem a **Student b = o**? Azért, mert egy Student objektum mindig példánya az Object osztálynak, de egy Object nem szükségszerűen példánya a Student-nek. Bár mi látjuk, hogy az o valóban egy Student objektum, a fordító ezt nem látja. Ahhoz, hogy a fordítónak megadjuk, hogy az o egy Student objektum, explicit átalakítást kell használnunk. A szintakszis hasonló az egyszerű adattípusok átalakításához. A cél objektumtípust zárójelben kell megadnunk az átalakítandó objektum neve előtt:

```
Student b = (Student)o; // Explicit átalakítás
```

Átalakítás szülőosztályról gyermekosztályra

Explicit átalakítás szükséges, ha szülőosztály objektumát akarjuk gyermekosztályra konvertálni.

```
Apple x = (Apple) fruit;
```

```
Orange x = (Orange) fruit;
```

Az instanceof operátor

Használjuk az instanceof operátort annak ellenőrzésére, hogy egy objektum egy osztály példánya-e:

```
Object myObject = new Circle();  
... // Forráskód  
/** Átalakítás, ha a myObject a Circle egy  
    példánya*/  
if (myObject instanceof Circle) {  
    System.out.println("The circle diameter is " +  
        ((Circle)myObject).getDiameter());  
    ...  
}
```

Az equals metódot

Az `equals()` metódot két objektum tartalmát hasonlítja össze. A metódot `Object` osztályban lévő alapértelmezett implementációja a következő:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Az `equals`
metódot például
a `Circle`
osztályban felül
van írva.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```

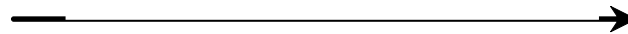
MEGJEGYZÉS

A `==` operátort használjuk két egyszerű típusú adat összehasonlításához, vagy annak eldöntésére, hogy két objektum hivatkozása megegyezik-e. Az `equals` metódus két objektum tartalmának egyezőségét vizsgálja, feltéve, hogy a metódus módosult az objektumok definiálóosztályában. A `==` operátor erősebb az `equals` metódusnál abban, hogy a `==` ellenőrzi, hogy két referenciaváltozó ugyanarra az objektumra hivatkozik-e.

A `protected` módosító

- A `protected` módosító az osztály adatalemeire és metódusaira alkalmazható. Egy publikus osztály védett (`protected`) adata vagy metódusa a csomag minden más osztályából vagy azok (akár más csomagban lévő) gyermekosztályaiból érhetők el.
- `private`, `default`, `protected`, `public`

Növekvő láthatóság

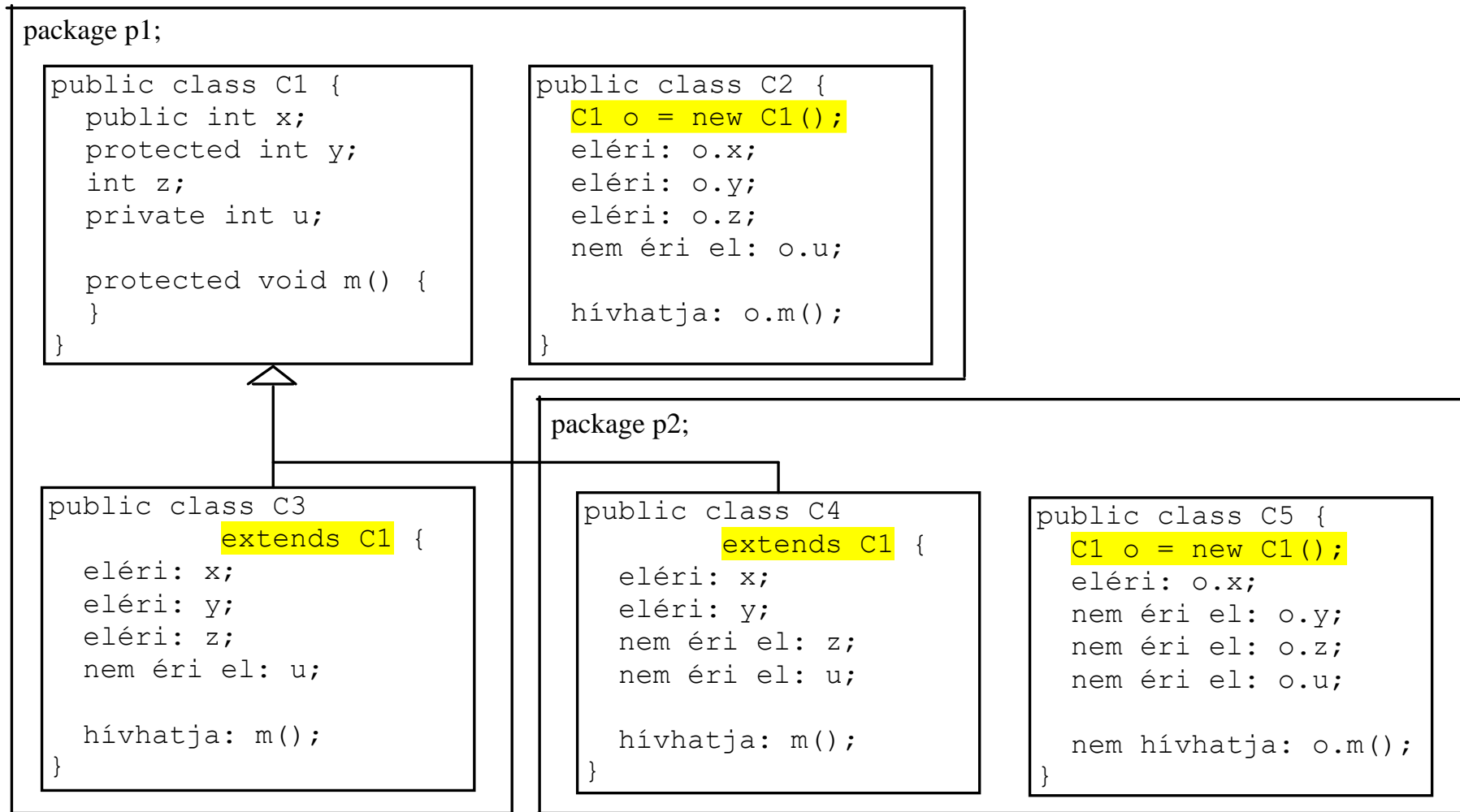


`private`, `semmi` (ha nincs módosító), `protected`, `public`

Elérés összegzése

Osztály tagjainak módosítója	Elérés ugyanazon osztályból	Elérés ugyanazon csomagból	Elérés gyermek- osztályból	Elérés másik csomagból
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Láthatósági módosítók



Gyermekosztály nem gyengítheti az elérést

Egy gyermekosztály felülírhatja a szülőosztályának egy védett metódusát, és publikussá teheti a láthatóságát. Egy gyermekosztály azonban nem szűkítheti a szülőosztályban definiált metódus láthatóságát. Így például a szülőosztályban publikusként metódust publikusként kell definiálni a gyermekosztályban is.

A final módosító

- A `final` osztály nem terjeszthető ki:

```
final class Math {  
  
    ...  
  
}
```

- A `final` változó egy konstans:

```
final static double PI = 3.14159;
```

- A `final` metódus nem írható felül a gyermekosztályokban.

Objektum, absztrakt osztály

- Az OO nyelvek általában ismerik az absztrakt osztály fogalmát.
- Az absztrakt osztály egy olyan eszköz, amellyel viselkedésmintákat adhatunk meg, amelyeket aztán valamely leszármazott osztály majd konkretizál.

Objektum, absztrakt osztály

- Egy absztrakt osztályban általában vannak absztrakt módszerek, ezeknek csak a specifikációja létezik, implementációjuk nem.
- Egy absztrakt osztályból származtatható absztrakt és konkrét osztály.
- A konkrét osztály minden módszeréhez kötelező az implementáció, egy osztály viszont mindaddig absztrakt marad, amíg legalább egy módszere absztrakt.
- Az absztrakt osztályok nem példányosíthatók, csak örököltethetők.

Objektum, absztrakt osztály

- Egyes OO nyelvekben létrehozhatók olyan osztályok, amelyekből nem lehet alosztályokat származtatni (ezek az öröklődési hierarchia „levelei” lesznek).
- Ezek természetesen nem lehetnek absztrakt osztályok, hiszen akkor soha nem lehetne őket konkretizálni.



Programozási nyelvek 2

Interfészek

Interfész/felület

- Az interfész olyan viselkedéseket definiál, amelyet az osztályhierarchia tetszőleges osztályával megvalósíthatunk.
- Az interfész (`interface`) metódusfejeket definiál abból a célból, hogy valamely osztály azt a későbbiekben implementálja, megvalósítsa.
- Definiálhat konstansokat (`public static final`)
- Az objektum elérésének, használatának egy lehetséges módját határozza meg.
- Egy interfészből nem lehet példányt létrehozni
- Az interfészek örökíthetők
- Egy konkrét osztály megvalósítja az interfészt, ha az összes metódusát megvalósítja (UML szaggatott nyíl, java kulcsszó: `implements`)
- Konvenció: az `implements` záradék az `extends` záradékot követi, ha mindkettő van.

Interfész/felület

- Egy osztály több interfészt is implementálhat
- Az értékadás kompatibilitás hasonló, mint az öröklődés esetén
- Mivel az interfész a megvalósítás nélküli, vagyis absztrakt metódusok listája, alig különbözik az absztrakt osztálytól. A különbségek:
 - Az interfész egyetlen metódust sem implementálhat, az absztrakt osztály igen.
 - Az osztály megvalósíthat több interfészt, de csak egy ősosztálya lehet.
 - Az interfész nem része az osztályhierarchiának. Egymástól "független" osztályok is megvalósíthatják ugyanazt az interfészt.
- Amikor egy osztály megvalósít egy interfészt, akkor alapvetően aláír egy szerződést. Az osztálynak implementálni kell az interfészben és szülőinterfészeiben deklarált összes metódust, vagy az osztályt absztraktként kell deklarálni.

Interfész/felület

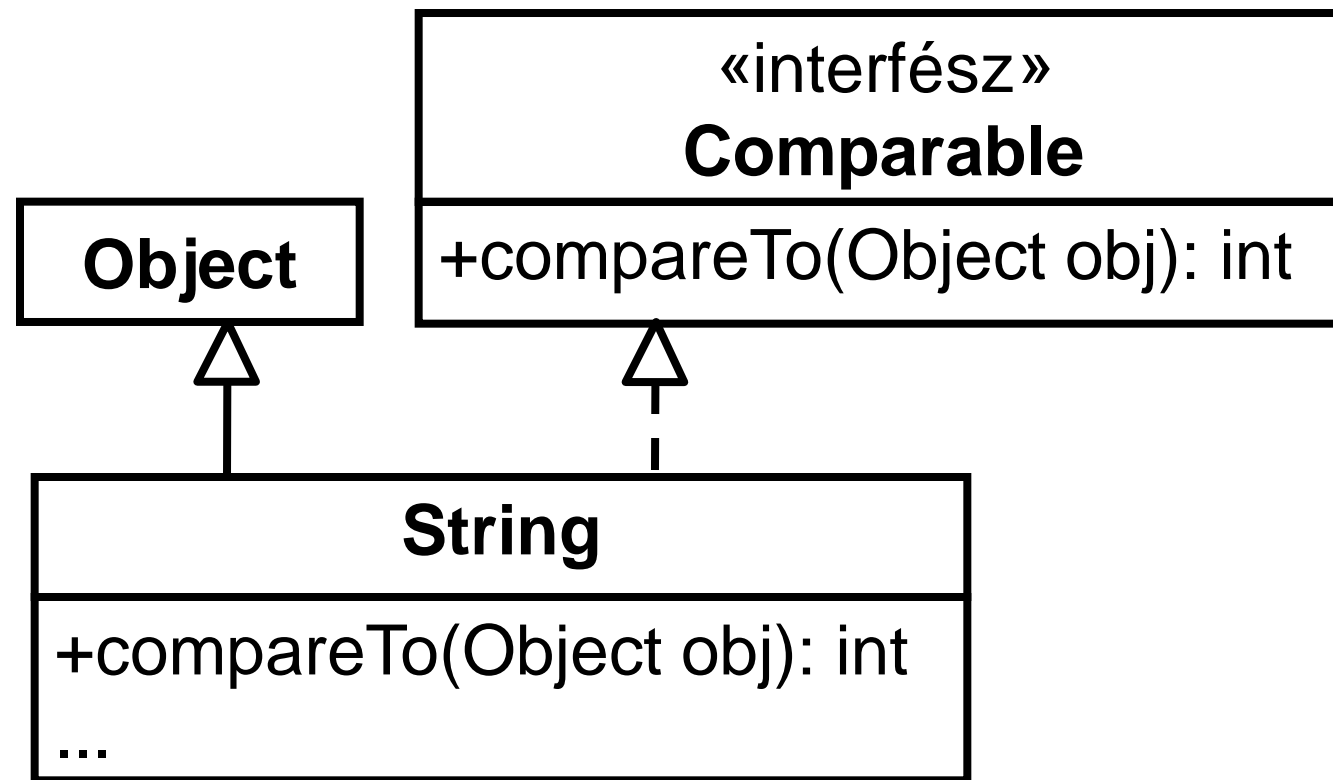
- Az interfészek hasznosak a következő esetekben:
 - Hasonlóságok megfogalmazása anélkül, hogy mesterként osztályhierarchiát építenénk fel (például az Ember és Papagáj is tud fütyülni, mégsem a Fütyül osztályból, hanem pl. az Emlős osztályból származtatjuk és csak azt kérjük, hogy implementálja a Fütyül osztályt is.)
 - Olyan metódusok definiálása, amelyeket több osztályban meg kell valósítani
 - Többszörös öröklődés modellezése

Példa interfészre

```
public interface Predator {  
    public boolean chasePrey(Prey p);  
    public void eatPrey(Prey p);  
}  
  
public class Cat implements Predator {  
    public boolean chasePrey(Prey p) {  
        // p préda üldözésének implementálása  
    }  
  
    public void eatPrey (Prey p) {  
        // p préda elfogyasztásának implementálása  
    }  
}
```

A Comparable interfész

- Az elemek rendezhetővé válnak, ha implementálják a Comparable interfészt (compareTo metódusát).
- Például, a String hasonlítható:



Példa: Comparable interfész

```
class Henger implements Comparable<Henger>{  
    private double sugar, magassag;  
  
    ...  
  
    public int compareTo(Henger obj) {  
        if(terfogat()<obj.terfogat())  
            return -1;  
        if (terfogat()>obj.terfogat())  
            return 1;  
        return 0;  
    }  
}
```

```
import java.util.*;

public class HengerProgram {
    public static void main(String[] args) {
        Henger[] hengerek=new Henger[4];
        ...
        //Max térfogat
        int max=0;
        for(int i=0;i<hengerek.length;i++)
            if(hengerek[i].compareTo(hengerek[max])>0)    max=i;
        System.out.println("A legnagyobb térfogat: "
                           +hengerek[max].terfogat());

        //Rendezés
        Arrays.sort(hengerek);
        //Lista
        System.out.println("\nRendezve:");
        for(int i=0;i<hengerek.length;i++)
            System.out.println(hengerek[i]);
    }
}
```