

Programozás III

OOP ÖRÖKLŐDÉS,
INTERFÉSZ

OBJEKTUM-ORIENTÁLT PARADIGMA – ALAPELVEK



OBJEKTUM-ORIENTÁLT PARADIGMA – ALAPELVEK

Az OOP alapelvei:

- egységbezárás;
- öröklődés;
- polimorfizmus (sokalakúság)

- újrahasznosíthatóság

OBJEKTUM-ORIENTÁLT PARADIGMA – ÖRÖKLŐDÉS

Öröklés (inheritance):

Az objektum-osztályok továbbfejlesztésének lehetősége. Ennek során a származtatott osztály öröklí ősétől annak attribútumait és metódusait, de ezeket bizonyos szabályok mellett újakkal egészítheti ki, és meg is változtathatja.

Azt a folyamatot, amikor egy már meglévő osztály felhasználásával, kiterjesztésével hozunk létre egy osztályt, öröklődésnek vagy kiterjesztésnek nevezzük

Az eredeti osztályt ősosztálynak nevezzük az új, továbbfejlesztett osztályt származtatott osztálynak. (szülő – superclass; gyerek – subclass).

OBJEKTUM-ORIENTÁLT PARADIGMA – ÖRÖKLŐDÉS

Alapgondolat: a gyerekek öröklik őseik metódusait és változóit.

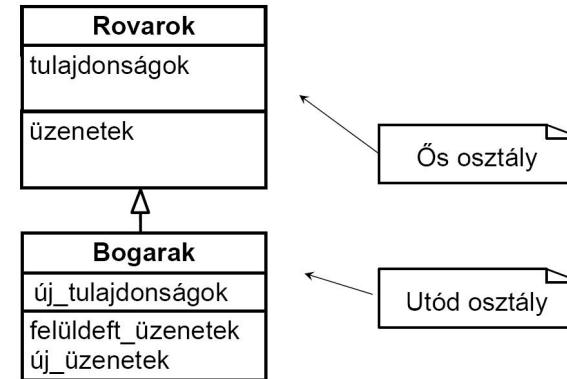
Az ősosztály minden metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz. (Feltéve, ha az ős megengedi.)

A gyerek minden új művelete vagy adata egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz.

Minden metódus, amelyet átdefiniálunk a gyerekben, a hierarchiában felülbírálja az örökölt metódust.

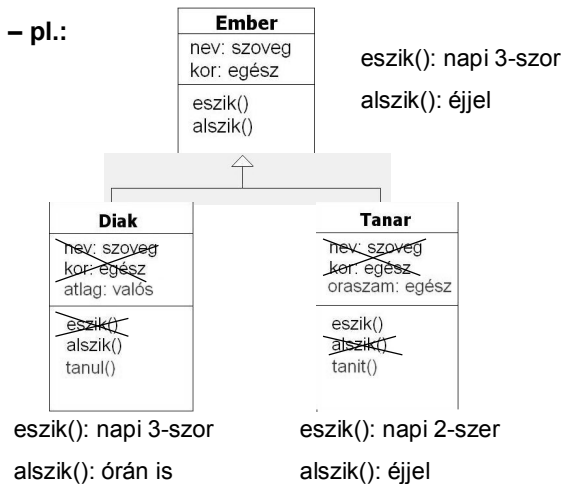
OOP – ÖRÖKLŐDÉS UML JELÖLÉSE

Öröklődés



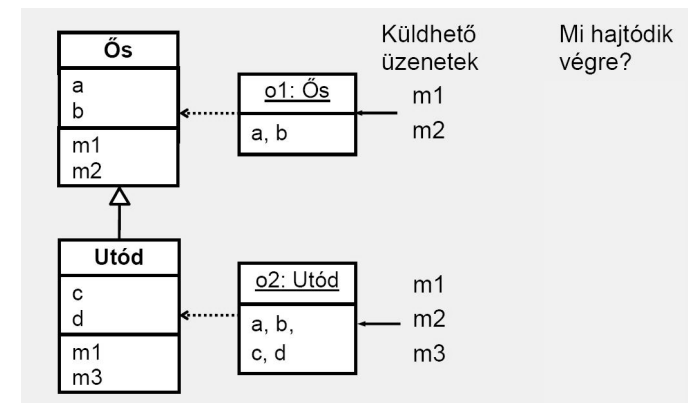
OBJEKTUM-ORIENTÁLT PARADIGMA – ÖRÖKLŐDÉS

Öröklődés – pl.:



OBJEKTUM-ORIENTÁLT PARADIGMA – ÖRÖKLŐDÉS

Utód adatai, küldhető üzenetek:



OBJEKTUM-ORIENTÁLT PARADIGMA – ÖRÖKLŐDÉS

Egy ősből több leszármaztatott osztályt is készíthetünk. Az öröklődési hierarchia mélysége tetszőleges.

Egy származtatott osztálynak

- legfeljebb egy szülője lehet (pl.: Pascal, Java, C#) (öröklődési fa)
- több szülője is lehet (pl.: C++) – (öröklődési gráf)

Metódusok törzsét megváltoztathatjuk, visszatérési típusát nem. A változók nevét, típusát sem változtatjuk. Új adatokkal és metódusokkal kiegészíthetjük az osztályt.

Statikus adattag/metódus öröklését nagyon végig kell gondolni.

AZ ÖRÖKLŐDÉS HÁTTERE

Az osztályokhoz tartozó VMT-t a fordítóprogram hozza létre. Minden osztálynak van saját VMT-je, amelyben annyi bejegyzés van, ahány virtuális metódusa van az osztálynak.

Amikor példányosítunk, a kiválasztott konstruktor dönti el, hogy melyik osztály VMT tábláját csatolja a példányhoz.

Amikor a program fut, és eléri azt a pontot, ahol a késői kötés szerepel, akkor a kód egyszerűen kiolvassa az aktuális példány VMT mezőjét, és az ott szereplő táblázatban „megkeresi” a kívánt bejegyzést, és kiolvassa, hogy melyik metódus-változatot kell meghívni.

AZ ÖRÖKLŐDÉS HÁTTERE

Az öröklést a virtuális metódusok használata teszi lehetővé.

Az ilyen metódusok meghívásakor a hívásban végrehajtásra kerülő implementációt az adott objektumpéldány típusa határozza meg, függetlenül a felhasznált referencia típusától.

Vagyis: csak futás közben dől el, hogy a virtuális metódus meghívása konkrétan melyik metódus legyen. (késői kötés)

A virtuális metódusok címét minden objektum a hozzá kapcsolódó VMT-ben (Virtuális metódus tábla) tárolja.

ÖRÖKLŐDÉS A JAVA-BAN

A Java minden metódusa virtuális. ⇒ Minden kötés késői.

Előny: könnyebb az öröklődés, nehezebb hibát véteni.

Hátrány: lassúbb, nagyobb memóriaigény.

ÖRÖKLŐDÉS A JAVA-BAN

Öröklődéshez az **extends** kulcsszót használjuk:

```
[módosító] class UtodOszaly extends OsOszaly
{
    //törzs
}
```

ÖRÖKLŐDÉS A JAVA-BAN

Az őszosztály konstruktorának meghívása:

- (1) Ha van közvetlen konstruktorhívás (super kulcsszó), akkor a paraméterlistának megfelelő konstruktor hívódik meg az őszosztályban.
- (2) Ha nincs közvetlen konstruktorhívás, de van az őszosztályban paraméterek nélküli konstruktor (pl.: default konstruktor), akkor az hívódik meg.
- (3) Ha nincs közvetlen konstruktorhívás, és az őszosztályban nincs paraméterek nélküli konstruktor, fordítási hiba keletkezik.

ÖRÖKLŐDÉS A JAVA-BAN

this objektumreferencia

- az aktuális objektumra hivatkozik

super objektumreferencia

- a közvetlen őszosztályra hivatkozik
- változók elérése : **super.getVáltozónév()**
- metódusok elérése : **super.metódusnév()**

Konstruktorok kérdése: az utódok **nem öröklík** a konstruktorokat, azok megírásáról magunknak kell gondoskodni.

Az ősz osztály konstruktorának meghívása:

super (paraméterek)

(a super() hivatkozás mindig a konstruktor legelső utasítása!)

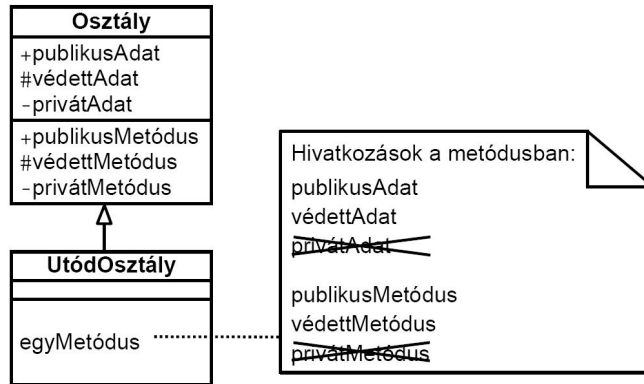
ÖRÖKLŐDÉS A JAVA-BAN

Ha egy leszármazott osztály metódusának ugyanaz a szignatúrája (azaz neve, paramétereinek száma és típusa), valamint visszatérési típusa, mint az őszosztály metódusának, akkor a leszármazott osztály felülírja az őszosztály metódusát.

Ha egy leszármazott osztály egy osztálymetódust ugyanazzal az aláírással (szignatúrával) definiál, mint a felsőbb osztálybeli metódus, akkor a leszármazott osztály metódusa elrejti (elfedi) a szülőosztálybelit.

OOB – BIZTONSÁGI KÉRDÉSEK

Osztály védelme



ÖRÖKLŐDÉS A JAVA-BAN – TOVÁBBI MÓDOSÍTÓK

– final:

ebből az osztályból semmi sem származtatható
(előny: sebesség)
a módosító használható változó és metódusnév előtt is.

– abstract:

„kötelező” származtatni belőle
magyarul: nem példányosítható.

Az absztrakt (azaz törzs nélküli) metódust a származtatott osztályban felül kell írni (ott mondjuk meg, hogy mit is csináljon), egyébként a származtatott osztály is absztrakt lesz.

FONTOS: Adattagok továbbra is **private** elérésűek legyenek – biztonsági okok.

ÖRÖKLŐDÉS A JAVA-BAN – LÁTHATÓSÁG

Láthatóság (hozzáférési mód, védelem):

Az utód csak azt láthatja, amit az ős megenged neki (amit nem tilt le).

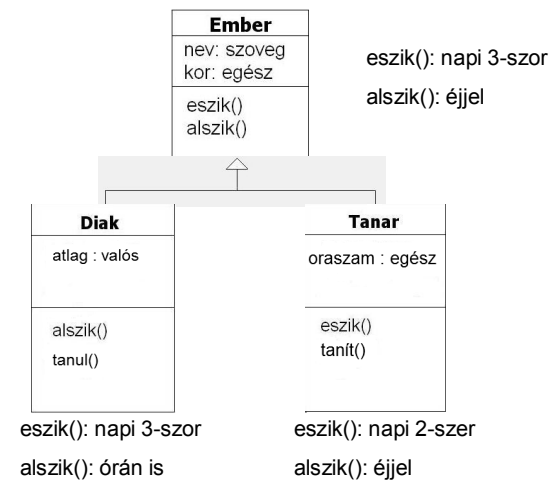
– Nyilvános (**public**): mindenki láthatja – UML jelölés: +

– Privát (**private**): csak az osztály saját metódusai férhetnek hozzá – UML jelölés: -

– Védeett (**protected**): csak az osztályból, leszármazottjaiból és a csomagból lehet hozzáférni. – UML jelölés: #

Ha nincs módosító, akkor csak az aktuális csomag osztályai látják egymást. A láthatóság nem szűkülhet az öröklődéskor.

ÖRÖKLŐDÉS – PÉLDA



A PÉLDA JAVA MEGVALÓSÍTÁSA

```
public class Ember{

    private String nev;
    private int kor;
    public Ember(String nev, int kor){
        this.nev = nev;
        this.kor = kor; Helyesen: születési év + életkor() metódus
    }

    public void eszik(){
        System.out.println("Napi 3-szor eszik.");
    }

    public void alszik(){
        System.out.println("Ejszaka jot alszik");
    }

    public String getNev(){
        return nev;
    }
}
```

A PÉLDA JAVA MEGVALÓSÍTÁSA

```
public class Foprogram{

    public static void main(String args[]){

        Ember egyik = new Ember("Janos", 35);
        Diak masik = new Diak("Jani", 21, (float)4.2);

        System.out.println("Az ember neve: " + egyik.getNev());
        System.out.println("viselkedese: ");
        egyik.eszik();
        egyik.alszik();

        System.out.println();

        System.out.println("A diak neve: " + masik.getNev());
        System.out.println("viselkedese: ");
        masik.eszik();
        masik.alszik();
        masik.tanul();
    }
}
```

A PÉLDA JAVA MEGVALÓSÍTÁSA

```
public class Diak extends Ember{

    private float atlag;

    public Diak(String nev, int kor, float atlag){
        super(nev, kor);
        this.atlag = atlag;
    }

    public void alszik(){
        System.out.println("Oran neha elalszik.");
    }

    public void tanul(){
        System.out.println
            ("De ha megszoritjak, akkor tanul.");
    }
}
```

A PÉLDA JAVA MEGVALÓSÍTÁSA

```
Az ember neve: Janos
viselkedese:
Napi 3-szor eszik.
Ejszaka jot alszik

A diak neve: Jani
viselkedese:
Napi 3-szor eszik.
Oran neha elalszik.
De ha megszoritjak, akkor tanul.

Process completed.
```

OBJEKTUM-ORIENTÁLT PARADIGMA – ALAPELVEK

3. alapelv:

Sokalakúság (polimorfizmus – polymorphism):

Ugyanarra a kérelemre a különböző objektumok különbözőképpen reagálnak. (Pl. különböző módon „beszél()nek” az állatok – az Allat osztályban definiált beszél() metódust másképp valósítják meg.

A származtatás során az ős osztályok metódusai újrainrás nélkül is képesek legyenek az új, átdefiniált metódusok használatára.



OOP ALAPELVEK – ÖSSZEFOGLALVA

Objektumok: egy egységben az adatok és a rajtuk végzett műveletek. (Egyszerre a modularitás és a struktúra elemei.)

Egységbezárás (az információ elrejtése): Az objektumokat nem lehet kívülről nem várt módon manipulálni. Csak meghatározott metódusokon keresztül módosítható az állapot.

Öröklés: Létrehozhatók már létező típusok specializációi, melyek használhatják (és bővíthetik) a létező típusok műveleteit anélkül, hogy újra meg kellene valósítani őket.

Polimorfizmus: A referenciák különböző típusú objektumokra hivatkozhatnak, és a hivatkozott objektumoktól függő viselkedést produkálhatnak.

POLIMORFIZMUS – PÉLDA

```
public abstract class Allat {
    public abstract String beszed();
}

public class Macska extends Allat{
    @Override
    public String beszed() {
        return "miau";
    }
}

public class Kutya extends Allat{
    @Override
    public String beszed() {
        return "vau";
    }
}

public static void main(String[] args) {
    List<Allat> allatok = new ArrayList<>();
    allatok.add(new Kutya());
    allatok.add(new Macska());
    System.out.println("Állathangok: ");
    for (Allat allat : allatok) {
        System.out.println(allat.beszed());
    }
}

run:
Állathangok:
vau
miau
```

ÖRÖKLŐDÉS A JAVA-BAN



ÖRÖKLŐDÉS A JAVA-BAN

JAVA-ban csak egyszeres öröklődés van – pedig néha kellene többszörös is. (Pl. lakókocsi)

DE megvalósítható a többszörös öröklődés is.

Mi az oka a többszörös öröklődéstől való idegenkedésnek?

- a többszörös öröklődés problémát jelenthet a fordító számára (pl.: káró típusú öröklési lánc)
- biztonsági okok
(a hackerek könnyen kihasználhatják ezt a problémát)

TÖBBSZÖRÖS ÖRÖKLÉS – MEGOLDÁS: INTERFÉSZ

Az interfész konstans értékek és absztrakt metódusok deklarációjának az összessége.

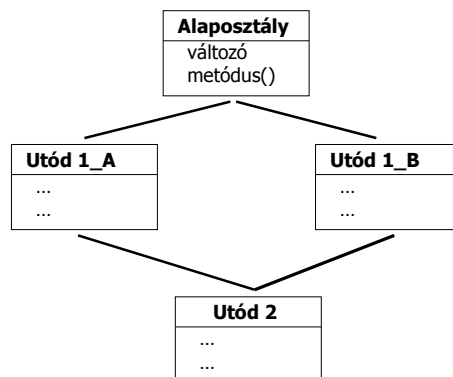
vagyis:

a metódusok törzs nélkül vannak deklarálnva

önmagában nem használhatók, implementálni kell őket

az implementációt egy-egy osztály végzi

KÁRÓ TÍPUSÚ ÖRÖKLÉSI LÁNC



Probléma: az Utód2 osztály kétszeresen örökölné az alaposztály tulajdonságait, ez pedig nem lehetséges.

TÖBBSZÖRÖS ÖRÖKLÉS – MEGOLDÁS: INTERFÉSZ

Interfész létrehozása:

```
[módosítók] interface InterfeszNev [extends...]  
{  
    //absztrakt metódusok  
}
```

- az interfész módosítója csak **publikus** lehet
- az absztrakt metódusok csak publikusak lehetnek
- az interfész alapértelmezésben absztrakt
(nem lehet példányosítani)

TÖBBSZÖRÖS ÖRÖKLÉS – MEGOLDÁS: INTERFÉSZ

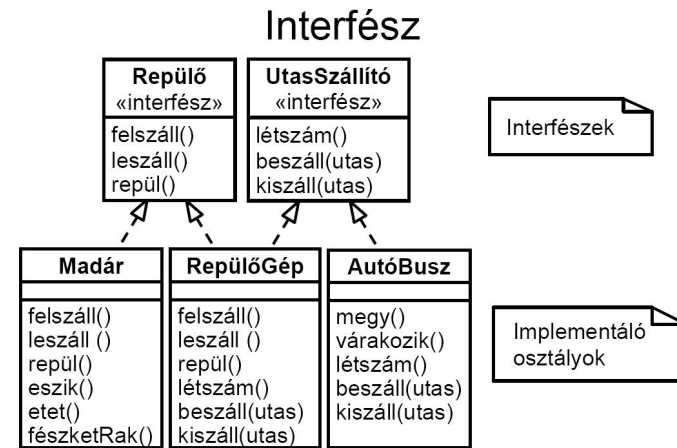
Interfész implementálása:

```
[módosítók] class OsztályNév implements InterfészNév{  
    //absztrakt metódusok  
}
```

Több interfész implementálása esetén vesszővel felsoroljuk az implements kulcsszó után az összes interfész nevét

KÖTELEZŐ AZ ÖSSZES ABSZTRAKT METÓDUS IMPLEMENTÁLÁSA!!!

INTERFÉSZ – UML (PÉLDA)



TÖBBSZÖRÖS ÖRÖKLÉS – MEGOLDÁS: INTERFÉSZ

Egy osztály több interfészt is implementálhat:

```
public class Osztaly implements Egyik, Masik {
```

Interfészek között is létezik öröklődés – jelölése szintén az **extends** kulcsszó.



Ezért alkalmas a többszörös öröklés megvalósítására!!!

Sőt, interfészek között többszörös öröklődés is lehet:

```
public interface Utod extends Egyik, Masik {
```

INTERFÉSZ – PÉLDA

```
interface Ital {  
    void koccint();  
}  
  
interface Alkohol extends Ital {  
    boolean megart(int pohar);  
}  
  
class Sor implements Alkohol {  
  
    public boolean megart (int pohar) {  
        if (pohar < 2) return false; else return true;  
    }  
  
    public void koccint() {  
        System.out.println("Nem koccintunk.");  
    }  
}  
  
public class Pelda {  
  
    public static void main(String args[]) {  
        Sor b = new Sor();  
        int i=3;  
        if (b.megart(i)) System.out.println("Vigyázz, megárt!");  
        b.koccint();  
    }  
}
```

```

interface Ital {
    void koccint();
}

interface Alkohol extends Ital{
    boolean megart(int pohar);
}

class Sor implements Alkohol {

    public boolean megart (int pohar){
        if (pohar < 2) return false; else return true;
    }

    public void koccint() {
        System.out.println("Nem koccintunk.");
    }
}

```

INTERFÉSZ – PÉLDA

Fordításkor létrejött fájlok:

```

Ital.class
Alkohol.class
Sor.class
Pelda.class

```

Eredmény:

```

Vigyázz, megárt!
Nem koccintunk.

Process completed.

```

INTERFÉSZ – PÉLDA

```

public class Pelda {

    public static void main(String args[]){
        Sor b = new Sor();
        int i=3;
        if (b.megart(i)){
            System.out.println("Vigyázz, megárt!");
        }
        b.koccint();
    }
}

```

Lehet-e egy forrásfájlban az összes?

INTERFÉSZ ALKALMAZÁSOK

Interfész „saját” alkalmazása:

Ha nagy a projekt, és többen dolgoznak rajta, akkor először célszerű felületeket tervezni interface alapon, azután jöhet az osztálygyártás. – Az interfész egy vázat ad (egy tervrajz).

A Java sok „kész” interfészt használ:

A Java fejlődése: Egy új igény esetén születik egy jsr (Java Service Request) felület-definíció (interface), majd a gyártók elkészítik a saját megvalósításait.

Ilyen pl. jdbc. A jdbc egy interface gyűjtemény (felület), amelyet minden adatbázisgyártó implementál.

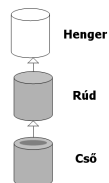
A grafikus csomagok is sok interfészt használnak.

ELEGÁNS PROJECT-SZERKEZET

Javasolt felépítés:

- interface (esetleg extends Comparable)
- az interface-t implementáló abstract osztály
- az abstract osztályt kiterjesztő további osztályok

```
Pl.      public interface AlakzatInterface {  
public abstract class AbsztraktAlakzat implements AlakzatInterface {
```

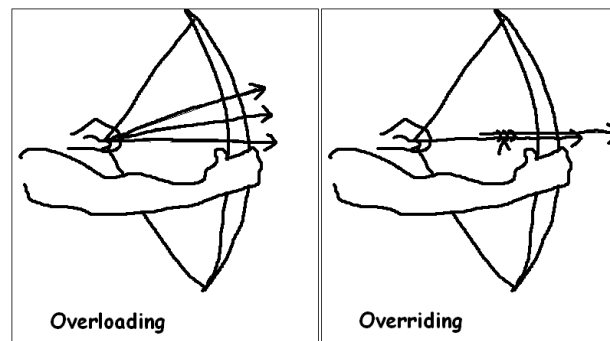


```
public class Henger extends AbsztraktAlakzat{
```

```
public class Rúd extends Henger {
```

```
public class Cső extends Rúd {
```

OVERLOAD vs OVERRIDE



KÉRDÉSEK

1. Min múlik, hogy interfész vagy absztrakt osztály?
2. Mi a különbség az overload és override között?

OVERLOAD, OVERRIDE

Kiválasztás fordítási időben.
(statikus kötés)

private, static, final túlterhelhető.

Azonos név, eltérő szignatúra. (szignatúra: név + paraméterek típusa, száma)

Azonos osztályban vannak.

Kiválasztás futási időben.
(dinamikus kötés)

private, static, final nem definiálható felül.

Név, szignatúra, visszatérési típus azonos.

Különböző osztályokban vannak.