Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Objektum-orientáltság a Java-ban

Bevezetés I.

- A Java **objektum-orientált** nyelv, megtalálható benne az objektum-orientált programozási paradigma legtöbb eleme.
- Az alábbi táblázat egy másik ismert objektum-orientált nyelv, a **C++** és a Java legfontosabb különbségeit foglalja össze (OOP szempontból).

C++	Java
Osztályok, absztrakt osztályok.	Osztályok, absztrakt osztályok, interfészek.
Függvények, változók osztályon kívül is lehetnek.	Metódusok, mezők csak osztályon belül lehetnek.
Explicite deklarálható virtuális metódusok.	Minden metódus virtuális.
Osztály- és metódusdeklaráció elkülönülhet.	Az osztályhoz tartozó összes kód az osztály deklarációján belül van.
Többszörös öröklődés.	Egyszeres öröklődés + interfészek + …
Egy file-ban tipikusan több osztály definíciója is van.	Egy file-ban általában csak egy osztály definíciója van.

Bevezetés II.

- Szóhasználat:
  - **osztály** (class),
  - az osztály **példány**ai (instance) az **objektum**ok (object),
  - az objektumok az osztályokból **példányosítás**sal (instantiation) jönnek létre,
  - az osztály állapotát a **mező**k (field) hordozzák,
  - az osztály műveletei a **metódus**ok (method).
- A mezők **NEM** “változók”, mert a “változó” kifejezés mást jelent!
- A metódusok **NEM** “függvények”, mert a Java-ban nincsenek függvények!

Osztályok I.

- Egy Java **osztály**t így definiálhatunk:

```
módosítók class osztálynév
   [ extends ősosztálynév ]
   [ implements interfésznév [ ,interfésznév ] ... ] {
   tagok
}
```

- A **módosítók**kal (modifier) az osztály hozzáférhetőségét, leszármaztathatóságát, és néhány egyéb jellemzőjét befolyásolhatjuk.
- A módosítók az alábbiak lehetnek, jelentésüket lásd később:

```
public protected private final static strictfp
```

- Az **osztálynév** tetszőleges érvényes Java azonosító lehet.
- Az extends és az implements jelentését lásd később.

Osztályok II.

```
class Alma {}
```

- A fenti példa egy minimális osztálydefiníció.

```
public class Jonatán extends Alma {
   double d;

   public boolean beleharap (int aNumber,int bNumber) {
      ...
   }
}
```

- A második példában az osztály definíciója összetettebb, és tagokat is tartalmaz.

Tagok

- Az osztály **tag**jai (member) a következők lehetnek:
  - konstruktorok,
  - inicializátorok,
  - metódusok,
  - mezők,
  - osztályok, interfészek.
- A konstruktorok kivételével minden tag lehet:
  - **példányhoz tartozó** (példány-…), vagy
  - **osztályhoz tartozó** (osztály-…, statikus …).
- A konstruktorok mindig az osztályhoz tartoznak.

Konstruktorok I.

- A **konstruktor**ok az osztályok példányosításakor futnak le, feladatuk az új objektum állapotának inicializálása.
- A konstruktorok neve megegyezik az osztály nevével.

```
[ módosítók ] osztálynév (paraméterlista) [ throws kivételtípus ] {
   ...
}
```

- A módosítók a konstruktor hozzáférhetőségét határozzák meg, jelentésüket lásd később:

```
public protected private
```

Konstruktorok II.

- A paraméterek megadása így történik:

```
[ final ] típus1 azonosító1 [ ,[ final ] típus2 azonosító2 ] ...
```

- Egy osztályban **több konstruktor is lehet**, ha paraméterlistájuk különbözik.
- Két konstruktor paraméterlistája akkor különbözik, ha
  - a paraméterek száma különböző, vagy
  - a paraméterek típusa különböző.
- A paraméterek típusa tetszőleges Java típus lehet.
- A `final` és a `throws` jelentését lásd később.

Konstruktorok III.

```
Kutya (int fogakSzáma,boolean harapós,double magasság) { ... }   // OK
Kutya (int fogakSzáma,boolean harapós,KutyaSzín szín) { ... }    // OK
Kutya (int fogakSzáma,boolean harapós,double súly) { ... }   // Nem OK
```

- A fenti példa egy osztály három konstruktorát mutatja.
- Az első két konstruktor “megfér” egy osztályban, mert a paraméterlistájuk különböző: az utolsó paraméter típusa eltérő.
- A harmadik konstruktor “ütközik” az elsővel, mert paraméterlistájuk megegyezik. A paraméter neve természetesen nem számít, hiszen annak csak a konstruktoron belül van szerepe.

Konstruktorok IV.

- A konstruktorhívásoknak három típusa létezik:
  - **implicit konstruktorhívás** példányosítás során,
  - **explicit konstruktorhívás**,
  - **ősosztály konstruktorának hívása**.
- Implicit konstruktorhívás:

```
new osztálynév (paraméterlista)
```

- A fenti kifejezés létrehozza az *osztálynév* nevű osztály egy példányát, a megadott paramétereknek megfelelő szignatúrájú konstruktor segítségével.
- Példa:

```
new Kutya (120,true,60);
```

Konstruktorok V.

- Explicit konstruktorhívás akkor történik, ha egy osztály egy konstruktorából egy másik konstruktort hívunk meg:

```
this (paraméterlista);
```

- A fenti konstrukció tipikus használata, hogy egy általános paraméterezésű konstruktort hívunk meg más, specifikusabb paraméterezésű konstruktorokból:

```
Osztaly () {
   this (1);
}

Osztaly (int a) {
   this (a,0);
}

Osztaly (int a,int b) { ... }
```

Konstruktorok VI.

- Ősosztály konstruktorának hívása a **super** kulcsszóval történik (pontos szabályokat lásd később):

```
super (paraméterek);
```

- **Az explicit konstruktorhívás és az ősosztály konstruktorának hívása csak a konstruktor első utasítása lehet.**
- **Megjegyzés**: az explicit konstruktorhívásból adódó rekurziót a fordító észleli, és fordítási hibát jelez.
- Ha az osztályban nem definiálunk konstruktort, a fordító automatikusan generál egy paraméterek nélküli, úgynevezett **default konstruktor**t.

Kitérő: referenciák I.

- A Java-ban a típusok két nagy csoportja a primitív típusok és a **referencia típus**ok.
- A primitív típusú változók közvetlenül a megfelelő típusú értékeket tárolják.
- A referencia típusú változóknak két alapvető állapota van:
  - egy objektumra mutatnak rá, vagy
  - nem mutatnak semmire, ekkor értékük a **null** literállal jelölt érték.
- A referencia fogalma nagyjából megfelel a C pointer fogalmának, vannak azonban fontos különbségek.

Kitérő: referenciák II.

- Az alábbi táblázat a Java referenciát hasonlítja össze a C és a C++ pointer, illetve referencia fogalmával.

Java	C illetve C++ pointer	C++ referencia
Primitív típusokra mutató referenciák nem léteznek.	int a; int *b = &a;	int a; int &b = a;
Objektumokra csak referenciákkal hivatkozhatunk.	Osztaly o ();	Osztaly o ();
Osztaly p = new Osztaly ();	Osztaly o = Osztaly (); Osztaly *p = &o;	Osztaly o = Osztaly (); Osztaly &r = o;
Osztaly o = null;	Osztaly *p = NULL;	Inicializálatlan (illetve null) referencia nem létezik.

- Következmény: a C-ben és C++-ban használt címe (**&**) és pointer indirekció (*****) operátorokra a Java-ban nincs szükség.

Példányhoz tartozó tagok I.

- A példányhoz tartozó tagokhoz egy, az adott objektumra mutató referencián keresztül férhetünk hozzá.
- A referencián keresztül a "**.**" operátorral férhetünk a tagokhoz:

```
Osztaly o = new Osztaly ();          // o egy Osztaly típusú referencia
o.tag                                  // hivatkozás o tag nevű tagjára
```

- **Megjegyzés**: az előbbiekből az is következik, hogy a C-ben és a C++-ban használatos nyíl (**->**) operátor sem létezik a Java-ban, illetve megfelel a "**.**" operátornak.

Példányhoz tartozó tagok II.

- A **példány-inicializátor**ok a konstruktor előtt futnak le.
- Használatuk akkor célszerű, ha az osztálynak több konstruktora van, és ezek közös kódrészleteket tartalmaznak.
- Egy osztályban egy, vagy több példány-inicializátor lehet, definíciójuk:

```
{
   ...
}
```

Példányhoz tartozó tagok III.

- A példánymetódusok így definiálhatók:

```
[ módosítók ] visszatérésiértéktípus azonosító (paraméterlista)
   [ throws kivételtípus ] {
   ...
}
```

- A példánymetódusok módosítói a metódus hozzáférhetőségét és egyes implementációs tulajdonságait határozzák meg és az alábbiak lehetnek (jelentésüket lásd később):

```
public protected private abstract final native strictfp synchronized
```

- A metódusnév tetszőleges érvényes Java azonosító lehet.
- A **throws** jelentését lásd később.

Példányhoz tartozó tagok IV.

- A metódus visszatérési értékének típusa helyén tetszőleges Java típus állhat, vagy a **void** kulcsszó, amely azt jelzi, hogy a metódusnak nincs visszatérési értéke.
- **Megjegyzés**: konstruktornak soha nem lehet visszatérési értéke, ezért a void-ot nem kell és nem is szabad kiírni.
- A metódusok paraméterlistájának szerkezete megegyezik a konstruktorokéval.
- Egy osztályban lehet több azonos nevű, de eltérő paraméterlistájú metódus is. Ezt a metódusnév **túlterhelés**ének (overloading) nevezzük.

Példányhoz tartozó tagok V.

```
boolean vajonHarap (int éhség,boolean postás) {
   return éhség > 100 || postás;
}

boolean vajonHarap (int éhség,boolean postás,boolean veszett) {
   return veszett || vajonHarap (éhség,postás);
}
```

- A fenti példa két metódust, illetve a metódusnév túlterhelést mutatja be.
- A második metódus az elsőnek általánosított változata.

Példányhoz tartozó tagok VI.

- Egy példánymetódust az objektumra mutató referencián keresztül hívhatunk meg, a **()** (metódushívás) operátor segítségével:

```
Osztaly o = new Osztaly ();
o.metodus (parameter);
```

- A fenti példában az o.metodus (parameter) kifejezés típusa a metódus visszatérési értékének típusa, értéke pedig a visszatérési érték lesz.
- Ha a metódusnak nincs visszatérési értéke, a fenti kifejezés csak utasításként használható.

```
if (vajonHarap (milyenÉhes (hányÓraVan (),mikorKapott ())),false)) {
   futokNagyon ();
}
```

Példányhoz tartozó tagok VII.

- Példánymezők definíciója:

```
[ módosítók ] típus azonosító [ = érték ] [ ,azonosító [ = érték ] ];
```

- A módosítók a mező hozzáférhetőségét és egyéb tulajdonságait határozzák meg, jelentésüket lásd később:

```
public protected private final transient volatile
```

- A mező típusa tetszőleges Java típus, azonosítója pedig tetszőleges érvényes Java azonosító lehet.
- A deklaráción belül a mezőt inicializálhatjuk, ha értéket is megadunk. Az érték lehet konstans, vagy az osztály egy másik mezője.
- Egy deklaráción belül több, azonos típusú mezőt deklarálhatunk, illetve inicializálhatunk.

Példányhoz tartozó tagok VIII.

- Példánymezőhöz egy, az objektumra mutató referencián keresztül férhetünk hozzá:

```
Osztaly o = new Osztaly ();
o.mezo
```

- A fenti példában az o.mezo kifejezés típusa a mező típusa, értéke pedig a mező értéke lesz.
- Példák meződefiníciókra:

```
int a;
boolean eleje = true,vége = false;
```

Statikus tagok I.

- A statikus inicializátorok az osztály betöltődésekor (≈ első használatakor) futnak le.
- Definíciójuk a **static** kulcsszóból, és egy blokkból áll.

```
static {
   ...
}
```

- Statikus inicializátort akkor érdemes használni, ha az osztály statikus mezőinek inicializálása egy egyszerű kifejezésnél bonyolultabb kódot igényel.

Statikus tagok II.

- A statikus metódusok deklarációja hasonló a példánymetódusokéhoz, a static módosító hozzátételével.

```
[ módosítók ] static visszatérésiértéktípus azonosító
   (paraméterlista) [ throws kivételtípus ] {
   ...
}
```

- A statikus metódusok lehetséges módosítói:

```
public protected private final native strictfp synchronized
```

- A metódus többi jellemzője megegyezik a példánymetódusoknál leírtakkal.

Statikus tagok III.

- Egy statikus metódus meghívása az osztályra, és a metódus nevére való hivatkozással történik, a **()** (metódushívás) operátor segítségével:

```
Osztaly.metodus (parameter);
```

- A fenti példában az Osztaly.metodus (parameter) kifejezés típusa a metódus visszatérési értékének típusa, értéke pedig a visszatérési érték lesz.
- Ha a metódusnak nincs visszatérési értéke, a fenti kifejezés csak utasításként használható.

Statikus tagok III.

- Statikus mezők definíciója:

```
[ módosítók ] static típus azonosító [ = érték ]
   [ ,azonosító [ = érték ] ];
```

- A módosítók a mező hozzáférhetőségét és egyéb tulajdonságait határozzák meg, jelentésüket lásd később:

```
public protected private final transient volatile
```

- A mezők többi jellemzője megegyezik a példánymezőknél leírtakkal.

Példányhoz tartozó tagok VII.

- Statikus mezőkhöz az oszályra és a mező nevére történő hivatkozással férhetünk hozzá:

```
Osztaly.mezo
```

- A fenti példában az Osztaly.mezo kifejezés típusa a mező típusa, értéke pedig a mező értéke lesz

Példány- és statikus tagok I.

- A példány- és statikus tagok közötti különbség a példány- és statikus mezők közötti különbségből adódik.
- A **példánymezők objektumokhoz kötődnek**, vagyis egy osztály egy-egy példánya a mező egy-egy külön példányát tartalmazza.
- A **statikus mezők** ezzel szemben **osztályokhoz kötődnek**, vagyis egy osztály összes példányához a mező egyetlen példánya tartozik.
- Ebből az következik, hogy szükség van a példány- és statikus metódusok megkülönböztetésére is, mert azok a metódusok, amelyek csak statikus mezőkre hivatkoznak, értelemszerűen nem kötődnek az egyes példányokhoz, így meghívásuk módja is más lehet.

Példány- és statikus tagok II.

- Az előbbiekből az alábbi szabályok következnek:
  - A példánymetódusok hozzáférhetnek az osztály összes mezőjéhez, a példány- és a statikus mezőkhöz egyaránt.
  - A statikus metódusok csak az osztály statikus mezőihez férhetnek hozzá.
  - Következmény: **a statikus metódusok csak az osztály statikus metódusait hívhatják meg**.
- A szabályok alkalmazását megkönnyítő fogalom a **statikus kontextus**.
- Általános szabály: **az osztály példányhoz kötődő tagjai statikus kontextusból nem hozzáférhetők**.
- Statikus kontextusnak minősül például (később lesz még több is): statikus inicializátorok, statikus metódusok.

Példány- és statikus tagok III.

```
class Osztaly {
   void metodus1 () {
      a = 3;                                          // OK, példánymetódus
      b = 5;     // OK, statikus mezőhöz bármilyen metódus hozzáférhet
   }

   static void metodus2 () {
      a = 4;          // fordítási hiba, statikus kontextusban vagyunk
      b = 6;                                          // OK, statikus mező
   }

   int a;
   static int b;
}
```

```
Osztaly o = new Osztaly ();                        // példány létrehozása
o.metodus1 ();                                                      // OK
o.metodus2 ();                                                      // OK
Osztaly.metodus1 ();  // fordítási hiba, statikus kontextusban vagyunk
Osztaly.metodus2 ();                              // OK, statikus metódus
```

Kitérő: a this referencia

- A **this** kulcsszó mindig a this-t tartalmazó kód meghívásakor használt referencia értékét jelenti, vagyis az osztály "aktuális" példányát.

```
class Osztaly {
   void m () {
      ... this ...
   }
}
```

```
Osztaly o = new Osztaly ();
o.m ();              // a this értéke ekkor az o értékével egyezik meg
```

- A fentiekből következik, hogy a this referencia statikus kontextusban nem létezik

Kitérő: a final módosító szerepe mezők esetén

- A **final** módosítóval ellátott mezők értéke az első értékadást követően nem változtatható meg.
- A final példánymezők értéket kell kapjanak az osztály minden konstruktorának végéig, különben fordítási hibát kapunk.
- A final statikus mezők értéket kell kapjanak a statikus inicializátorok végéig, különben fordítási hibát kapunk.
- A final statikus mezőket **fordításidejű konstans**oknak (compile-time constant) nevezzük.
- Az "értéket kell kapjanak" kifejezés a nyelvdefinícióban leírt definite assignment-et jelenti. Ennek tárgyalására itt nem térünk ki, részletesen lásd a JLS 16. fejezetét.

Öröklődés I.

- Az **öröklődés** célja az, hogy egy osztály tulajdonságait megváltoztatva új osztályt hozzunk létre.
- Az öröklődés során rendszerint az osztály bővítése történik, bár elképzelhető szűkítés is.
- A Java-ban az öröklődés új tagok behozását, illetve a meglévők módosítását jelenti.
- Az eredeti osztályt **ősosztály**nak (superclass), az öröklődéssel, vagy **leszármaztatás**sal (subclassing) létrehozott új osztályt pedig **leszármazott osztály**nak (subclass) nevezzük.
- A leszármazott osztályból öröklődéssel újabb leszármazottakat hozhatunk létre, így **osztályok hierarchiája** alakul ki.

Öröklődés II.

- Egy osztály **közvetlen ős**e (immediate superclass) az az osztály, amelyből leszármaztattuk.
- Egy osztálynak minden olyan osztály őse, amely vagy a közvetlen őse, vagy annak őse.
- A leszármazás tényét a leszármazott osztály deklarációjakor kell jelölni, az **extends** kulcsszóval:

```
class A {
}

class B extends A {
}
```

- Ha az extends deklarációt elhagyjuk, az osztály implicite az **Object** nevű osztályból származik, amely a Java API része

Öröklődés III.

- Az öröklődés hatással van a referencia típusok használatára is:

```
class A { ... }

class B { ... }                                  // nem származik az A-ból!
```

```
A o = new A ();                                                    // OK
o = new B ();                  // fordítási hiba, eltérő referencia típus
B o2 = new B ();                                                   // OK
o2 = new A ();                 // fordítási hiba, eltérő referencia típus
```

```
class A { ... }

class B extends A { ... }                              // A-ból származik
```

```
A o = new A ();                                                    // OK
o = new B ();                                          // OK, A a B őse
B o2 = new B ();                                                   // OK
o2 = new A ();                          // fordítási hiba, B nem őse A-nak
```

Öröklődés IV.

- A referencia típusok konverziójának szabályai:
  - Minden referencia típusú érték **automatikusan konvertálható ősosztály típusú referenciára**.
  - Minden referencia típusú érték **explicit konverzió**val (typecast) konvertálható leszármazott típusú referenciára. Ha a konvertálandó érték nem konvertálható a megadott leszármazott típusra, futásidejű hiba keletkezik.

```
class A { ... }
class B extends A { ... }
```

```
A o = new B ();                              // OK, automatikus konverzió
B o2 = (B) o;                                   // OK, explicit konverzió
A o3 = new A ();
o2 = (B) o3;              // futásidejű hiba, a konverzió nem lehetséges
```

Öröklődés V.

- A futásidejű hibák számának csökkentése érdekében az eleve lehetetlen konverziókat a fordító érzékeli, és fordítási hiba keletkezik:

```
class A { ... }
class B { ... }                                    // nem származik A-ból!
```

```
A o = new B ();
                                  // fordítási hiba, B-nek semmi köze A-hoz
B o2 = (B) o;
            // fordítási hiba, o nem tartalmazhat B típusú referenciát
```

Öröklődés VI.

- A futásidejű hibák kiküszöbölésére alkalmas a referenciák által mutatott objektumok típusának futásidejű vizsgálatára (a **referenciák futásidejű típusá**nak vizsgálatára) szolgáló **instanceof** operátor is:

```
class A { ... }
class B extends A { ... }
```

```
A o = new B ();
A o2 = new A ();
o instanceof B                                    // true
o instanceof A                                    // true
o2 instanceof B                                   // false
o2 instanceof A                                   // true
```

- Az `instanceof` operátor első operandusa egy referencia, a második pedig egy típusnév. A kifejezés értéke `true`, ha a referencia által mutatott objektum értékül adható egy, a megadott típusú referenciának, egyébként `false`.

Öröklődés VII.

- A fentiekből következik, hogy egy Object típusú referenciának bármilyen típusú objektumra mutató referencia értékül adható:

```
class A { ... }
class B extends A { ... }
```

```
Object o = new A ();
o = new B ();
```

- Egy `Object` típusú referencia tehát hasonlóan viselkedik, mint a C-ben egy void típusú pointer.
- Rendkívül rossz programozói gyakorlat, ha minden referenciát Object típusúnak deklarálunk, mert így a fordításidejű típusellenőrzés szinte minden előnyét elveszítjük, ráadásul a sok explicit típuskonverzió miatt programunk áttekinthetetlenné válik.

Kitérő: final módosító szerepe osztályok esetén

- A `final` módosítóval ellátott osztályokból **nem hozhatunk létre leszármazott osztályt**.
- Erre általában akkor van szükség, ha nem megengedhető, hogy az adott osztály helyett egy leszármazottját használjuk.
- Mivel bármely típusú referenciának értékül adható egy leszármazott típusú referencia, ennek megakadályozása csak úgy lehetséges, ha eleve kizárjuk a leszármazottak létrejöttét.
- Ez akkor lehet fontos, ha az osztály viselkedésének (esetleg rosszindulatú) megváltoztatása veszélyeztetné a program stabilitását, vagy biztonságát.

Tagok öröklődése I.

- Az öröklődés során az osztály tagjainak egy része öröklődik, vagyis a leszármazott osztály szintén rendelkezni fog az adott taggal.
- A leszármazott osztályban az öröklődéssel "kapott" tagokat megváltoztathatjuk.
- Az alábbiakban áttekintjük a tagok öröklődésének szabályait.

Tagok öröklődése II.

- **A konstruktorok nem öröklődnek.**
- Ennek oka az, hogy a konstruktor feladata a konkrét osztály inicializálása, az ős inicializálását az ős konstruktora végzi el.

```
class A {
   A () { a = 1; }            // az A mezőinek inicializálását végzi el

   int a;
}

class B extends A {
   B () { b = 2; }         // csak a B mezőinek inicializálását végzi el

   int b;
}
```

Tagok öröklődése III.

- A fenti példában az ősosztály konstruktora automatikusan meghívódik.
- Az ősosztály konstruktorainak meghívása az alábbi szabályok szerint történik:
  - Ha **van explicit konstruktorhívás** (super kulcsszó), a paraméterlistának megfelelő konstruktor hívódik meg az ősosztályban.
  - Ha **nincs explicit konstruktorhívás**, és van az ősosztályban paraméterek nélküli konstruktor (például default konstruktor), akkor az hívódik meg.
  - Ha **nincs sem explicit konstruktorhívás**, és **az ősosztályban nincs paraméterek nélküli konstruktor**, fordítási hiba keletkezik.

Tagok öröklődése IV.

- Példa konstruktorhívásokra:

```
class A {
   A () { ... }                  // van paraméterek nélküli konstruktor
   A (int a) { ... }
}

class B extends A {
   B () { }                   // OK, van paraméterek nélküli konstruktor
   B (int a) { super (a); }            // OK, explicit konstruktorhívás
}
```

```
class A {
   A (int a) { ... }                      // nincs default konstruktor!
}

class B extends A {
   B () { }   // fordítási hiba, nincs paraméterek nélküli konstruktor
   B (int a) { super (a); }            // OK, explicit konstruktorhívás
}
```

Tagok öröklődése V.

- A **példány- és statikus inicializátorok** a konstruktorokhoz hasonló okokból szintén **nem öröklődnek**, de az ősosztály példányosítása, illetve inicializálása során végrehajtódnak.
- Az alábbi példa a végrehajtási sorrendet mutatja a B osztály példányosítása során:

```
class A {
   static { ... }                          // 1, az A osztály betöltődésekor
   { ... }                                                              // 3
   A () {}                                                              // 4
}

class B extends A {
   static { ... }                           // 2, a B osztály betöltődésekor
   { ... }                                                              // 5
   B () {}                                                              // 6
}
```

Tagok öröklődése VI.

- Mind a példány-, mind a statikus metódusok öröklődnek:

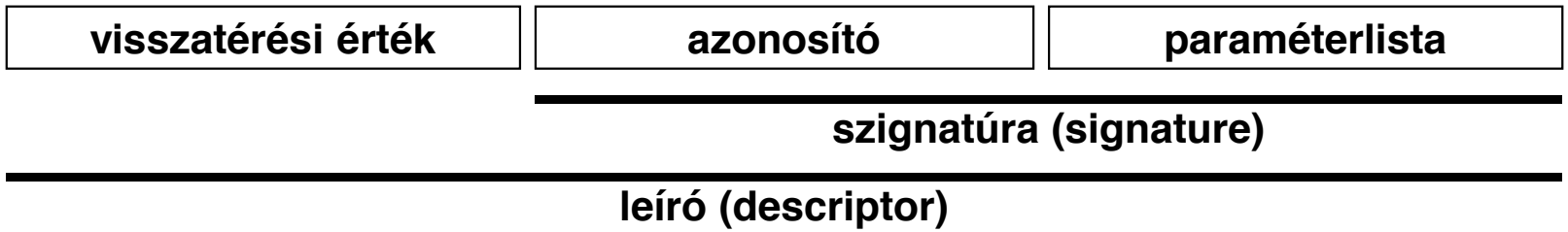
```
class A {
   void m1 () { ... }
   static void m2 () { ... }
}

class B extends A {}
```

```
A a = new A ();
B b = new B ();
...
a.m1 ();                                                    // OK
b.m1 ();                                                    // OK
a.m2 ();                                                    // OK
b.m2 ();                                                    // OK
A.m2 ();                                                    // OK
B.m2 ();                                                    // OK
```

Kitérő: metódusok szignatúrája és leírója

- Egy osztályban lehet több, **azonos nevű**, de **eltérő paraméterlistájú** metódus is.
- Két paraméterlista akkor különböző, ha vagy a paraméterek száma, vagy azok típusa különbözik.
- Egy metódus **szignatúrá**ját (signature) az azonosítója (neve) és paramétereinek típusa alkotja.
- Egy metódus **leíró**ját (descriptor) visszatérési értékének típusa, és szignatúrája alkotja.



Tagok öröklődése VII.

- Gyakori eset, hogy egy metódus viselkedését a leszármazott osztályban meg akarjuk változtatni.
- A leszármazott osztályban lévő metódus akkor "változtat meg" egy ősosztályban lévő metódust, ha az **ős- és a leszármazott osztályban lévő metódusok szignatúrája megegyezik**.
- Fontos szabály, hogy ha a leszármazottban lévő metódus szignatúrája megegyezik az ősben lévő metóduséval, tehát az leszármazottban lévő metódus "megváltoztatja" az ősben lévő metódust, akkor **primitív visszatérési érték esetén az alosztályban is ugyanaz kell legyen a visszatérési érték, referencia típus esetén azonban a felülíró metódus visszatérési értéke lehet alosztálya a felülírtének**. (Java5 óta - JLS 8.4.5, *covariant returns*)

Tagok öröklődése VIII.

- A következő példa a fenti szabályokat illusztrálja:

```
class A {
   void m1 (int i) { ... }
   int m2 (char c) { ... }
   int m3 (boolean b) { ... }
   int m4 (boolean b) { ... }
   A m5() { ... }
}

class B extends A {
   void m1 (int i) { ... }              // OK, "megváltoztatja" m1-et
   void m2 (char c) { ... }
                    // fordítási hiba, eltérő visszatérési érték típus
   int m3 (int i) { ... }           // OK, semmi köze az A-beli m3-hoz
   void m4 () { ... }               // OK, semmi köze az A-beli m4-hez
   B m5() { ... }                     // OK, B az A-nak leszármazottja
}
```

Tagok öröklődése IX.

- A példány- és statikus metódusokra vonatkozó további szabályok eltérőek.
- Ha egy példánymetódus "megváltoztat" egy, az ősosztályban lévő példánymetódust, azt mondjuk, hogy a leszármazottban lévő metódus **felüldefiniál**ja (override) az ősosztályban lévő metódust.
- **Példánymetódus nem definiálhat felül statikus metódust.**
- Azt, hogy egy metódushívás esetén az ősosztály, vagy a leszármazott osztály megfelelő szignatúrájú metódusa hívódik meg, **a referencia által mutatott objektum típusa** (a referencia futásidejű típusa) határozza meg.
- Az előzőekben leírt viselkedést **polimorfizmus**nak, a metódust pedig **virtuális metódus**nak nevezzük.

Tagok öröklődése X.

- A következő példa az előbbi szabályokat illusztrálja:

```
class A {
   int m1 () { return 1; }
}

class B extends A {
   int m1 () { return 2; }          // felüldefiniálja az A-beli m1-et
}
```

```
A r1 = new A ();
B r2 = new B ();
A r3 = r2;
...
r1.m1 ()                                          // a kifejezés értéke 1
r2.m1 ()                                          // a kifejezés értéke 2
r3.m1 ()        // a kifejezés értéke 2, bár az r3 A típusú referencia
```

Tagok öröklődése XI.

- Ha egy statikus metódus "megváltoztat" egy, az ősosztályban lévő statikus metódust, azt mondjuk, hogy a leszármazottban lévő metódus **elrejt**i (hide) az ősben lévő metódust.
- **Statikus metódus nem rejthet el példánymetódust.**
- Azt, hogy egy metódushívás esetén az ős-, vagy a leszármazott osztály megfelelő szignatúrájú metódusa hívódik meg, **a metódushíváskor használt referencia típusa**, illetve a megadott osztálynév határozza meg.

Tagok öröklődése XII.

- A következő példa az előbbi szabályokat illusztrálja:

```
class A {
   static int m1 () { return 1; }
}

class B extends A {
   static int m1 () { return 2; }          // elrejti az A-beli m1-et
}
```

```
A r1 = new A ();
B r2 = new B ();
A r3 = r2;
...
r1.m1 ()                                   // a kifejezés értéke 1
r2.m1 ()                                   // a kifejezés értéke 2
r3.m1 ()   // a kifejezés értéke 1, bár az r3 a B egy példányára mutat
((A) r2).m1 ()             // a kifejezés értéke 1, a typecast is elég
A.m1 ()                                    // a kifejezés értéke 1
B.m1 ()                                    // a kifejezés értéke 2
```

Tagok öröklődése XIII.

- A felüldefiniált, illetve elrejtett metódusokat a super kulcsszóval hívhatjuk meg:

```
class A {
   int m1 () { ... }
   static int m2 () { ... }
}

class B extends A {
   int m1 () { ... }
   static int m2 () { ... }
   void m3 () {
      m1 ();                                    // a B-beli m1
      m2 ();                                    // a B-beli m2
      super.m1 ();                              // az A-beli m1
      super.m2 ();                              // az A-beli m2
      A.m2 ();                                  // az A-beli m2
   }
}
```

Kitérő: final módosító szerepe metódusoknál

- A `final` módosítóval ellátott metódusok leszármaztatással nem definiálhatók felül, illetve nem rejthetők el.
- Minden metódus, amelyet egy final osztályban definiáltunk, implicite final.
- A final metódusok használatának leggyakoribb célja az, hogy az osztály leszármaztatásával ne lehessen valamely, az osztály működését kritikusan befolyásoló kódrészletet megváltoztatni, ugyanakkor lehessen leszármazottakat létrehozni.

Tagok öröklődése XIV.

- Mind a példány-, mind a statikus mezők öröklődnek.

```
class A {
   int i = 1;
   static int j = 2;
}

class B extends A {}
```

```
A a = new A ();
B b = new B ();
...
a.i                                                    // OK
a.j                                                    // OK
b.i                                                    // OK
b.j                                                    // OK
A.j                                                    // OK
B.j                                                    // OK
```

Tagok öröklődése XV.

- Mint a metódusok esetében, a mezők esetében is előfordulhat, hogy egy mezőt az öröklődés során meg akarunk változtatni.
- A mezőknek mind a típusát, mind a kezdeti értékét (ha van) megváltoztathatjuk.
- A leszármazott osztályban lévő mező akkor "változtat meg" egy ősosztálybeli mezőt, ha **azonosítójuk** (nevük) **megegyezik**.

Tagok öröklődése XVI.

```
class A {
   int i = 1;
   static int j = 2;
}

class B extends A {
   double i = 3.14;
   static boolean j = true;
}
```

```
A r1 = new A ();
B r2 = new B ();
A r3 = r2;
...
r1.i                                    // a kifejezés értéke 1
r1.j                                    // a kifejezés értéke 2
r2.i                                 // a kifejezés értéke 3.14
r2.j                                 // a kifejezés értéke true
r3.i                                    // a kifejezés értéke 1
r3.j                                    // a kifejezés értéke 2
```

Tagok öröklődése XVII.

- Az előbbi példából az látszik, hogy mind a példány-, mind a statikus mezők esetében **a hivatkozásban használt referencia típusa, illetve a megadott osztálynév** határozza meg azt, hogy az ősosztály, vagy a leszármazott osztály mezőjére hivatkozunk.
- A leszármazott osztály mezője mindkét esetben **elrejt**i (hide) az ősosztály azonos nevű mezőjét.
- A metódusoknál leírtaktól eltérően **példány- és statikus mezők egymást is elrejthetik**.

Tagok öröklődése XVIII.

- Az elrejtett mezőhöz a super kulcsszóval férhetünk hozzá:

```
class A {
   int i = 1;
   static int j = 2;
}

class B extends A {
   void m () {
      i = 0;                                          // a B-beli i
      j = false;                                      // a B-beli j
      super.i = 0;                                   // az A-beli i
      super.j = 0;                                   // az A-beli j
   }

   double i = 3.14;
   static boolean j = true;
}
```

Árnyékolás I.

- Előfordulhat, hogy egy metóduson belül egy lokális változó (amely lehet a blokkon belül deklarált lokális változó, vagy egy metódus paraméter) neve megegyezik a metódust tartalmazó osztály egyik mezőjének nevével.
- A lokális változó ekkor **leárnyékol**ja (shadow) a mezőt.
- A mezőhöz ekkor példánymetódus esetén a this referencia segítségével, statikus metódus esetén pedig az osztály nevének kiírásával férhetünk hozzá.

Árnyékolás II.

```
class A {
   void m () {
      int a;                  // a lokális változó leárnyékolja a mezőt
      a = 1;                                          // a lokális változó
      this.a = 2;                                                // a mező
   }

   static void n (boolean b) {     // a paraméter leárnyékolja a mezőt
      b = true;                                                // a paraméter
      A.b = 3;                                                   // a mező
   }

   int a;
   static int b;
}
```

Árnyékolás III.

- Jellegzetes példa árnyékolásra, és a this referencia használatára:

```
class A {
   void setNumber (int number) {
      this.number = number;              // a mező értékének beállítása
   }

   int number;
}
```

- A fenti megoldással nagyon gyakran találkozhatunk.
- Látszólagos bőbeszédűsége ellenére nagyon hasznos, mert nem kell egy új nevet bevezetni a paraméternek, így a kód áttekinthetőbb lesz.
- **Hátrány:** ha a fenti kódból a this-t kifelejtjük, a paraméternek adunk értéket (a saját értékét). Mivel ez a hiba fordítási hibát nem okoz, nagyon nehezen felderíthető.

Hozzáférés-vezérlés I.

- A **hozzáférés-vezérlés** célja, hogy a program egyes elemei csak azon más elemek számára legyenek közvetlenül elérhetők, amelyeknek erre valóban szükségük van.
- A hozzáférhetőségek helyes beállításával megvalósítható az **information hiding** elve, vagyis az egyes osztályok belső felépítése és működése elrejthető a külvilág elől.
- Az information hiding egyrészt véd a programozási hibák népes családja ellen, másrészt a program felépítésének átgondolására késztet.
- A beállított hozzáférhetőségeket a fordító minden esetben betartatja, de ezek a korlátok futásidőben áthághatók (a Reflection API segítségével, lásd később).

Hozzáférés-vezérlés II.

- A Java-ban az osztályok, illetve ezek tagjainak hozzáférhetősége a **hozzáférés-módosítók** segítségével állítható be.
- A módosítók, és az általuk jelölt hozzáférési szintek az alábbiak:
  - **public** módosító: **publikus** (public), az adott elem minden más elem számára hozzáférhető,
  - nincs módosító: **csomagon belüli** (package private), az adott elem az elemet tartalmazó csomagon (lásd később) belül hozzáférhető,
  - **protected** módosító: **védett** (protected), az adott elem csak az elemet tartalmazó osztály leszármazottai számára hozzáférhető,
  - **private** módosító: **privát** (private), az adott elem csak az elemet tartalmazó osztály számára hozzáférhető.

Osztályok hozzáférhetősége

- Osztályok esetén a hozzáférési szintek jelentése:
  - **public**: az osztály tetszőleges kód számára hozzáférhető,
  - **package private**: az osztály csak a definiáló csomag (lásd később) többi osztálya számára hozzáférhető,
  - **protected**: az osztály csak a definiáló osztály leszármazottai számára hozzáférhető (csak más osztályok belsejében definiált osztályok esetében használható),
  - **private**: az osztály csak a definiáló osztály számára hozzáférhető (csak más osztályok belsejében definiált osztályok esetén használható).

Tagok hozzáférhetősége

- Tagok esetén a hozzáférési szintek jelentése:
  - **public**: a tag tetszőleges kód számára hozzáférhető,
  - **package private**: a tag csak a tagot definiáló osztályt definiáló csomag (lásd később) többi osztálya számára hozzáférhető,
  - **protected**: a tag csak a definiáló osztály leszármazottai számára hozzáférhető,
  - **private**: a tag csak a definiáló osztály számára hozzáférhető.
- Metódusok esetében a leszármazott osztályban lévő, az ős egyik metódusát felüldefiniáló, vagy elrejtő metódus hozzáférhetősége **nem lehet szigorúbb, mint az ősben lévő metódusé**.
- Egy publikus metódust például nem definiálhatunk felül egy privát metódussal.

Beágyazott osztályok I.

- A **beágyazott** (nested) osztályok olyan osztályok, amelyeket egy másik osztályon belül definiálunk.
- A beágyazott osztályok az alábbiak szerint csoportosíthatók:
  - **statikus tagosztály**ok (static member class),
  - **belső osztály**ok (inner class), amelyek tovább csoportosíthatók:
    - **példány tagosztály**ok (non-static member class),
    - **lokális osztály**ok (local class),
    - **névtelen osztály**ok (anonymous inner class).

Beágyazott osztályok II.

- A **statikus tagosztály**ok a **befoglaló osztály** (enclosing class) statikus tagjaként vannak definiálva:

```
class A {
   static class B {
   }
}
```

- A statikus tagosztályok nem férhetnek hozzá a befoglaló osztály példány tagjaihoz, vagyis statikus kontextusnak minősülnek.
- A statikus tagosztályokhoz, mint a többi statikus taghoz, hozzáférhetünk a befoglaló osztály példányaitól függetlenül (az előző példa deklarációit alapul véve):

```
A.B b = new A.B ();
```

Beágyazott osztályok III.

- A **példány tagosztály**ok a statikus tagosztályokhoz hasonlóak, de nem statikus tagként vannak definiálva:

```
class A {
   class B {
   }
}
```

- A példány tagosztályok hozzáférhetnek a befoglaló osztály példány tagjaihoz, tehát nem minősülnek statikus kontextusnak.
- A példány tagosztályokhoz csak a befoglaló osztály egy példányán keresztül férhetünk hozzá, ezt **befoglaló példány**nak (enclosing instance) nevezzük.
- A példány tagosztály példányai mindig tartalmaznak egy referenciát a befoglaló példányra.

Beágyazott osztályok IV.

- A példány tagosztályok (és általában **a belső osztályok**) **nem definiálhatnak semmilyen statikus tagot, kivéve fordításidejű konstansokat**, mert ezek szerepét a befoglaló osztály statikus tagjai töltik be.
- Példány tagosztályok példányosítása így történik:

```
class A {
   class B {
   }
}
```

```
A a = new A ();
A.B b = a.new B ();
```

Beágyazott osztályok V.

- **Lokális osztály**t a lokális változókhoz hasonló módon, egy blokkon belül tetszőleges helyen definiálhatunk:

```
class A {
   void m () {
      class L { ... }
   }
}
```

- A lokális osztályok tulajdonképpen belső osztályok, de csak a befoglaló blokkon belül léteznek.
- Lokális osztályok definiálásakor semmilyen hozzáférés-módosító nem használható.

Beágyazott osztályok VI.

- A **névtelen osztály**ok olyan belső osztályok, amelyeknek csak egyetlen példánya jön létre.

```
class A { ... }

class B {
   void m () {
      A a = new A () {
         void n () { ... }
      }
   }
}
```

- A fenti példában a névtelen osztály az A osztály leszármazottja, és egy új metódust (n) definiál.

Beágyazott osztályok VII.

- Mivel a névtelen osztályoknak nincs nevük, explicit konstruktoruk sem lehet, azonban a fordító automatikusan generál egy olyan konstruktort, amely egy super hívás segítségével a létrehozáskor megadott paramétereket továbbítja az ősosztály konstruktorának.

```
class A {
   A (int a,double b) { ... }
}

class B {
   void m () {
      A a = new A (12,3.14) {
         void n () { ... }
      }
   }
}
```

Beágyazott osztályok VIII.

- Névtelen osztályok használatának akkor van értelme, ha egy osztály valamely jellemzőjét egyetlen használat erejéig szeretnénk megváltoztatni, vagy kiegészíteni.
- Névtelen osztályokra további példákat lásd később.

Interfészek I.

- Az **interfész**ek az osztályokhoz hasonló nyelvi elemek, azonban a bennük definiált metódusok nincsenek megvalósítva.
- Az interfészek egy jól definiált felületet írnak le, amelyen keresztül egy osztály valamely funkcionalitása elérhető.
- Az interfészeket az osztályok **megvalósít**ják (implement).
- Az interfész legfontosabb tulajdonsága, hogy **független a megvalósító osztály implementációjától**.
- **Egy osztály több interfészt is megvalósíthat**, így egy osztály több, akár független funkciót is tartalmazhat anélkül, hogy az ezeket használó kliensek tudnának arról, hogy ezek egy osztályban vannak megvalósítva.
- **Egy interfészt több osztály is megvalósíthat**, a megvalósítás tehát szabadon cserélhető, és a kliensek számára rejtett.

Interfészek II.

- Azt, hogy egy osztály implementál egy interfészt, a fordító számára az osztály definiálásakor az **implements** kulcsszó segítségével jelezni kell.
- Ahhoz, hogy az osztály valóban implementálja az interfészt, szükséges az, hogy az osztály az interfészben definiált összes metódust megvalósítsa.
- Az osztály csak akkor implementálja az interfészt, ha az implements után az interfésznév fel van tüntetve, a metódusok megvalósítása önmagában nem elég.

Interfészek III.

```
interface I1 {
   void m1 ();
}

interface I2 {
   void m2 ();
}

class C implements I1,I2 {
   public void m1 () { ... }
   public void m2 () { ... }
}
```

- Az interfészben a metódusok törzse nincs, és nem is lehet megadva.
- Az interfészekben **minden tag implicite publikus**, így az interfészt megvalósító osztályban mindig ki kell tenni a public módosítót.

Interfészek IV.

- Ha egy osztály több interfészt implementál, előfordulhat, hogy van olyan metódus, amely egyszerre több interfészben is szerepel.
- A metódust ebben az esetben természetesen csak egyszer kell (lehet) megvalósítani.
- Ha a két metódus csak a visszatérési érték típusában különbözik, fordítási hibát kapunk.

```
interface I1 { void m (); }

interface I2 { int m (); }

class C implements I1,I2 {                          // fordítási hiba
   ...
}
```

Interfészek V.

- Interfészekben definiálhatunk **mező**ket is, ezek **implicite public static final-ek** (vagyis fordításidejű konstansok).
- Az interfészt implementáló osztályban ezek a mezők az osztályban definiált mezőkhöz hasonlóan használhatók.
- Ha az osztályban definiált, vagy más interfészekből származó mezőkkel névütközés lép fel, a mezőre csak az interfésznévvel együtt lehet hivatkozni.

```
interface I1 { int CONST = 1; }

interface I2 { int CONST = 2; }

class C implements I1,I2 {              // I1.CONST, I2.CONST, C.CONST
   public static final int CONST = 3;
}
```

Interfészek VI.

- Interfészekben **osztály**okat és **interfész**eket is definiálhatunk, ezek a többi taghoz hasonlóan **implicite publikusak**.
- Az interfészben definiált osztályokhoz és interfészekhez az implementáló osztályokban, és azokon kívül is hozzáférhetünk.

```
interface I {
   interface J { ... }
   class C { ... }
}
```

```
class A implements I.J { ... }
```

```
I.C c = new I.C ();
```

Interfészek VII.

- Az interfészek **referenciatípusként is használhatók**.
- Ez jó programozói gyakorlat, mert hangsúlyozza, hogy a kód többi része független az interfészt megvalósító osztály implementációjától.
- Ha ezt a megoldást használjuk, a programkódot csak egy helyen kell javítani, ha másik implementációt választunk.

```
interface I { ... }

class C implements I { ... }
```

```
I i = new C ();
```

Interfészek VIII.

- Interfészekből az osztályokhoz hasonlóan lehet **leszármazott interfész**eket létrehozni az extends kulcsszó használatával.
- A leszármazott interfész tartalmazza az ős összes metódusát, illetve a leszármazottban definiált metódusokat.
- Ha a leszármazottban az ős egyik metódusával azonos szignatúrájú, de eltérő visszatérési értékű metódust definiálunk, fordítási hibát kapunk.

Absztrakt osztályok I.

- Az **absztrakt osztály**ok olyan osztályok, amelyeket arra tervezünk, hogy használat előtt leszármaztatással új osztályokat hozzunk létre belőlük.
- Az osztály definiálásakor az **abstract** módosító használatával jelezzük, hogy az osztály absztrakt.
- **Az absztrakt osztályok nem példányosíthatók.**
- Absztrakt osztály **nem lehet final**, mert akkor soha nem lehetne példányosítani.

```
abstract class A { ... }
```

```
A a = new A ();                    // fordítási hiba, absztrakt osztály
```

Absztrakt osztályok II.

- Az absztrakt osztályok leszármazottai, hacsak maguk is nem absztraktak, példányosíthatók

```
abstract class A {
   ...
}

class B extends A {
   ...
}
```

```
A a = new B ();                                    // OK, B nem absztrakt
```

Absztrakt osztályok III.

- Absztrakt osztályban definiálhatunk **absztrakt metódus**okat is, amelyeknek, az interfészekben definiált metódusokhoz hasonlóan, nincs törzsük.
- Ha egy osztályban van absztrakt metódus, az osztályt absztraktnak kell definiálni.
- Ha egy absztrakt osztály leszármazottja nem valósítja meg az absztrakt metódust, a leszármazott is absztrakt lesz.

```
abstract class A {
   abstract void m ();
}

class B extends A {          // OK, B megvalósítja m-et, nem absztrakt
   void m () { ... }
}
```

Absztrakt osztályok IV.

- **Absztrakt metódus nem lehet final**, mert akkor soha nem lehetne megvalósítani.
- **Absztrakt metódus nem lehet statikus**, mert a statikus metódusok meghívásához nincs szükség példányosításra, így lehetséges lenne meghívni egy absztrakt metódust.

Absztrakt osztályok V.

- Ha egy osztály implementál egy interfészt, de nem valósítja meg minden metódusát, absztraktnak kell definiálni.

```
interface I {
   void m1 ();
   void m2 ();
}

abstract class C implements I {   // absztrakt, nem valósítja meg m2-t
   public void m1 () { ... }
}

class D extends C {                // nem absztrakt, megvalósítja m2-t
   public void m2 () { ... }
}
```

```
I i = new C ();                          // fordítási hiba, C absztrakt
I j = new D ();                                 // OK, D nem absztrakt
```

Absztrakt osztályok VI.

- Az előbbi példához hasonló megoldást gyakran alkalmazunk API-k tervezésekor.
- Az interfész az, amit az API felhasználójának meg kell valósítania.
- Előfordulhat, hogy az interfész valószínű megvalósításainak zömében néhány metódust ugyanúgy kell megvalósítani, ekkor egy absztrakt osztályban ezt a néhány metódust előre megvalósíthatjuk, ez a **részleges megvalósítás** (skeletal implementation).
- A részleges megvalósítást tartalmazó osztály neve általában az Abstract szóval kezdődik.

```
interface Valami { ... }
abstract class AbstractValami implements Valami { ... }

class EgyValami extends AbstractValami { ... }
```