Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Reflection

Bevezetés

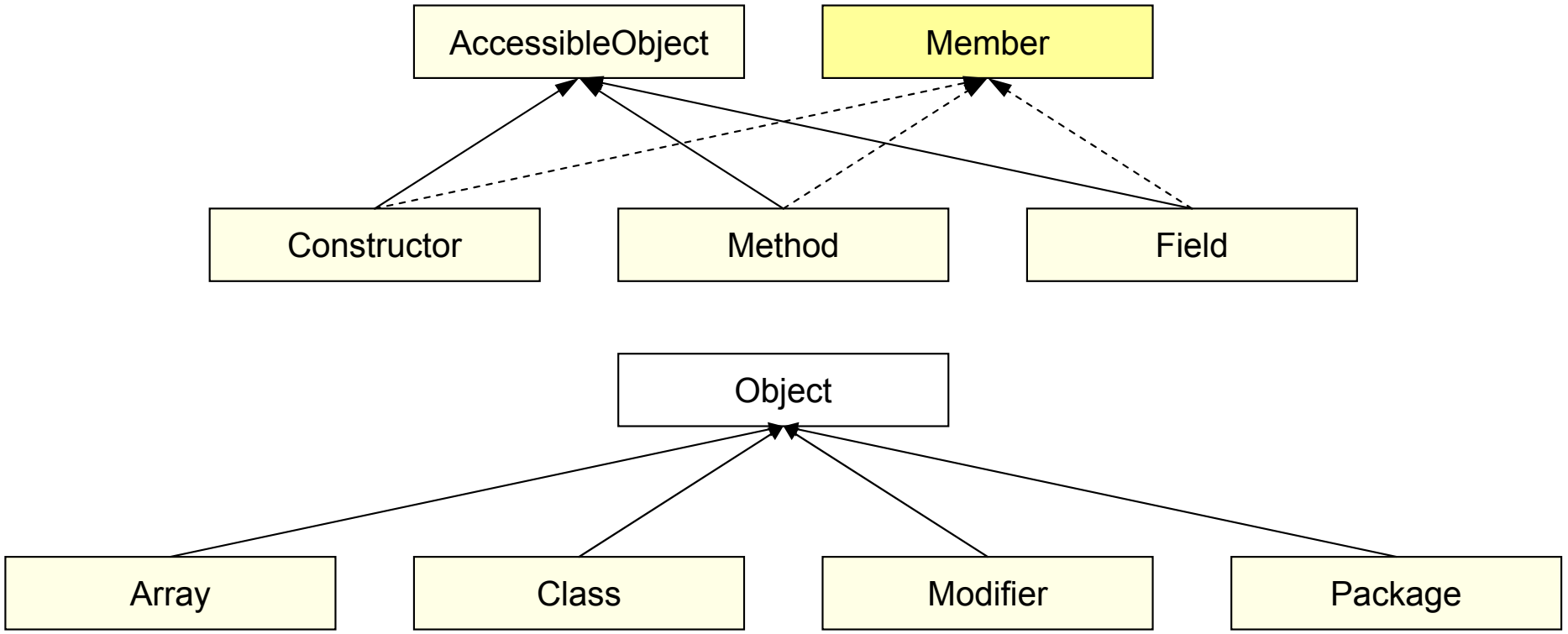
- A programozási nyelvek zömében nincs lehetőség egy program **metaszintű elemeihez programból hozzáférni**.
- **Metaszintű elemek** alatt itt az osztályokat, azok konstruktorait, metódusait, mezőit, illetve ezek jellemzőit értjük.
- A Java volt az első olyan nyelv, amely ezt lehetővé tette, az úgynevezett **Reflection API**-n keresztül.
- A reflection szónak ebben a kontextusban nincs elfogadott magyar megfelelője (a tükrözés nem túl szerencsés), így a továbbiakban is reflection-nek fogjuk hívni.

A Reflection API elemei

- A Reflection API zömmel a **java.lang.reflect** csomag osztályaiból és interfészeiből áll, valamint ide tartoznak a **java.lang.Class**, illetve a **java.lang.Package** osztályok is.
- A Reflection API elemei két fő részre oszthatók:
  - A **program elemeit és azok tulajdonságait** reprezentáló osztályok és interfészek:
    - Member, AccessibleObject, Array, Class, Constructor, Field, Method, Modifier, Package
  - **Segédosztályok** és interfészek:
    - InvocationHandler, Proxy, ReflectPermission
- Bár nyelvi elem, a Reflection API elemei közé sorolhatjuk a **class literál**okat is.

Programok elemei a Reflection API-ban I.

- Az alábbi ábra a Reflection API osztályainak viszonyát mutatja.



Programok elemei a Reflection API-ban II.

- Hogyan férhetünk hozzá a fenti elemekhez?
- A Java program alapeleme az osztály, így ebből kell kiindulnunk.
- Az osztályt a Reflection API-ban a Class osztály példányai reprezentálják.
- A világ kezdete: **szert tenni egy**, a vizsgálandó osztályt vagy objektumot leíró **Class példányra**.
- Ez alapvetően három módon történhet. Az egyik módszer **fordításidőben** állítja elő a Class objektumot, a másik kettő **futásidőben**.

Class objektumok előállítása I.

- Fordításidőben egy osztályhoz, interfészhez, tömbhöz, vagy primitív típushoz tartozó Class objektumot egy úgynevezett **class literál** segítségével állíthatunk elő (§15.8.2). Ez az osztály, interfész, tömb, vagy primitív típus nevéből, egy pontból ("."), és a class szóból áll.
- Példák:

```
String.class                // OK, osztály
Comparable.class            // OK, interfész
boolean[][].class           // OK, tömb
ArrayList[].class           // OK, tömb
int.class                   // OK, primitív típus
void.class                  // OK, primitív típus (!)

int i = 12; ... i.class     // hiba, az i nem típusnév
"alma".class                // hiba, az "alma" nem típusnév
```

Class objektumok előállítása II.

- Futásidőben egy objektum típusának megfelelő Class objektumot az Object osztály **getClass ()** metódusa segítségével kérhetjük le.
- Mivel minden osztály implicite az Object-ből származik, ezt minden objektummal megtehetjük (természetesen tömbökkel is!).
- Ugyanezen okból primitív típusokhoz tartozó Class objektumot így nem kaphatunk, ám erre a class literálok miatt amúgy sincs szükség.
- Példák:

```
new ArrayList ().getClass ();                              // OK, objektum
"alma".getClass ();                                      // OK, ez is objektum
int[] a = new int [12]; ... a.getClass ();               // OK, ez is objektum
this.getClass ();                                      // OK, tipikus használat

String.getClass ();          // hiba, a getClass () nem statikus metódus
Object o = null; o.getClass ();            // hiba, NullPointerException
```

Class objektumok előállítása III.

- A harmadik módszer Class objektum előállítására a Class osztály statikus **forName metódus**ának használata.
- A forName metódus segítségével az osztály neve alapján kaphatjuk meg az osztályt leíró Class objektumot.
- Ez a művelet az osztály betöltésével jár, ha az még nem történt volna meg, így a forName metódus ClassNotFoundException-t (is) dobhat.

```
try {
   Class cls = Class.forName ("java.lang.Object");
} catch (Exception e) {
   System.out.println ("Hiba az osztály betöltése során.");
}
```

- A forName metódusnak mindig az osztály **minősített nevét** kell megadni.

A Class osztály I.

- Class osztály példányai segítségével férhetünk hozzá az osztály konstruktoraihoz, metódusaihoz, mezőihez, és egyéb jellemzőihez.
- A következőkben a Class osztály fontosabb metódusait tekintjük át.
  - Class forName (String name)
    Már volt róla szó.
  - String getName ()
    Ha a Class objektum nem tömb típusú objektumhoz tartozik, a megfelelő osztály **nev**ét, egyébként a típus **kódolt alak**ját kapjuk (lásd később).
  - Példa nem tömb típusú objektumra:

```
String.class.getName () -> java.lang.String
```

Kitérő: Java típusok kódolt alakja I.

- A Java a típusokat belül egy speciális kódolt alakban tárolja.
- A kódokat az alábbi táblázat foglalja össze:

Java típus	Kód
`byte`	`B`
`char`	`C`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`

Java típus	Kód
`short`	`S`
`boolean`	`Z`
`void`	`V`
`osztály vagy interfész`	`L`*`osztálynév`*`;`
`tömb`	`[`*`elemtípus`*

- A tömbök dimenziószámát a tömb típusoknál a [-ek száma jelzi.

Kitérő: Java típusok kódolt alakja II.

- Példák

Java típus	Kód
`String`	`Ljava.lang.String;`
`int[]`	`[I`
`boolean[][]`	`[[Z`
`Object[]`	`[Ljava.lang.Object;`

- Ezzel a kódolással **metódusok szignatúrája** is leírható:

```
(paraméterek típusai)visszatérési érték típusa
```

```
String someMethod (int a,String[] b) ->
   (ILjava.lang.String;)Ljava.lang.String

void otherMethod (ArrayList l) ->
   (Ljava.util.ArrayList;)V
```

A Class osztály II.

- Class osztály további metódusai:
  - ClassLoader getClassLoader ()
    Az osztályt **betöltő ClassLoader**-t adja meg.
  - Class getComponentType ()
    Ha ez egy tömb típus, **a tömb elemtípusá**t adja meg.
  - Class getDeclaringClass ()
    Ha ez egy beágyazott osztály, **a beágyazó osztály**t adja meg.
  - Class[] getInterfaces ()
    Az osztály által **megvalósított interfészek**et adja meg.
  - int getModifiers ()
    Az **osztály módosítói**t adja meg.

A Class osztály III.

- A Class osztály további metódusai:
  - Package getPackage ()
    Az osztályt tartalmazó csomagot leíró **Package** objektumot adja meg.
  - Class getSuperclass ()
    Az **osztály ősé**t adja meg.
- Az alábbi metódusok az osztály publikus tagjainak lekérdezésére szolgálnak:
  - Class[] getClasses ()
    Az osztály **publikus belső osztályai**t, interfészeit adja meg.
  - Constructor getConstructors (Class[] parameterTypes)
    A megadott paraméterlistájú **publikus konstruktor**t adja meg.

A Class osztály IV.

- A Class osztály további metódusai:
  - Constructor[] getConstructors ()
    Az osztály **publikus konstruktorai**t adja meg.
  - Field getField (String name)
    A megadott nevű **publikus mező**t adja meg.
  - Method getMethod (String name,Class[] parameterTypes)
    A megadott nevű és paraméterű **publikus metódus**t adja meg.
  - Method[] getMethods ()
    Az osztály **publikus metódusai**t adja meg.
- A fenti lekérdező metódusoknak létezik a **Declared** szóval kiegészített változata is (pl. getDeclaredConstructors), amelyek nem a publikus, hanem az **osztályban deklarált tagok**at adják meg.

A Class osztály V.

- Példa

```
public class Test implements java.io.Serializable {

   public static void main (String[] args) {
      System.out.println (int[].class.getComponentType ());
      System.out.println (Test2.class.getDeclaringClass ());
      System.out.println (Test.class.getInterfaces () [0]);
      System.out.println (String.class.getPackage ());
      System.out.println (Test.class.getSuperclass ());
      // kiiratjuk a Test.class.getMethods () elemeit
      // kiiratjuk a Test.class.getDeclaredMethods () elemeit
   }

   class Test2 {}
}
```

A Class osztály VI.

- Eredmény:

```
int
class Test
interface java.io.Serializable
int
package java.lang, Java Platform API Specification, version 1.4
class java.lang.Object

public static void Test.main(java.lang.String[])
public native int java.lang.Object.hashCode()
...
public final native void java.lang.Object.notifyAll()

public static void Test.main(java.lang.String[])
static java.lang.Class Test.class$(java.lang.String)
```

A Class osztály VII.

- Az alábbi metódusok a Class objektum által reprezentált elem tulajdonságaira kérdeznek rá:
  - isArray: true, ha ez egy **tömb típus**.
  - isInterface: true, ha ez egy **interfész**.
  - isPrimitive: true, ha ez egy **primitív típus** (illetve void!).
- Az alábbi két metódus leszármazási viszonyokat ellenőriz:
  - isAssignableFrom (Class cls): true, ha ez a típus **megegyezik a cls paraméterben megadott típussal**, vagy annak **ősosztálya**, illetve **ősinterfésze**.
  - isInstance (Object obj): true, ha ennek a típusnak **értékül adható a megadott objektum**. Ez az instanceof operátor dinamikus megfelelője.

A Package osztály

- A Package osztály egy **csomag**ot reprezentál.
- Adatainak egy része csak akkor van kitöltve, ha a csomag egy **JAR file**-ból töltődött be, és a **manifest file**-ban a megfelelő adatok szerepelnek.
- A Package legfontosabb szolgáltatásai a következők:
  - getImplementationTitle: a megvalósítás **címe**,
  - getImplementationVendor: a megvalósítás **készítője**,
  - getImplementationVersion: a megvalósítás **verziója**,
  - ugyanezek Specification-nel: a **specifikáció** adatai,
  - isCompatibleWith: true, ha **kompatíbilis egy megadott verzióval**,
  - isSealed: true, ha a csomag **sealed**.

A Member interfész

- Az osztályok tagjait reprezentáló Constructor, Method és Field osztályok implementálják a **Member** interfészt.
- A Member interfész az alábbi általános szolgáltatásokat nyújtja:
  - Elkérhetjük a tagot deklaráló osztály **Class objektumá**t.
  - Megkaphatjuk a tag **módosítói**t.
  - Lekérdezhetjük a tag **nev**ét.
- Fontos a deklaráló osztály fogalma, mert noha egy osztályban szerepelhet egy tag öröklődés folytán is, **az adott tag deklaráló osztálya az az osztály, amelyikben a deklarációja ténylegesen szerepel**.

Az AccessibleObject osztály

- Az osztályok tagjait reprezentáló Constructor, Method és Field osztályok az **AccessibleObject** osztályból származnak.
- Egy osztály vagy objektum tagjaihoz való hozzáférést a **private** illetve **public** módosítók szabályozzák.
- A Reflection API segítségével lehetővé válik az, hogy **az amúgy nem látható** (például private) **tagokhoz is hozzáférhessünk**. Ez sok esetben szükséges lehet, például a Serialization-hoz hasonló szolgáltatások számára.
- Az AccessibleObject-től örökölt metódusok segítségével a taghoz való korlátlan hozzáférést engedélyezhetjük, vagy tilthatjuk.
- Ha a korlátlan hozzáférés le van tiltva, a Java-ban szokásos hozzáférési szabályok érvényesek.

A Modifier osztály

- A Member interfész getModifier metódusa egy integer érték formájában adja meg a módosítókat, ezt a számot a **Modifier** osztály segítségével tudjuk értelmezni.
- A következő lekérdező metódusok állnak rendelkezésünkre: isAbstract, isFinal, isInterface, isNative, isPrivate, isProtected, isPublic, isStatic, isStrict, isSynchronized, isTransient, isVolatile.

```
public class Test {
   public static void main (String[] args) {
      try {
         Class c = Test.class;
         Method m;
         m = c.getMethod ("main",new Class[] { String[].class } );
         System.out.println (Modifier.isStatic (m.getModifiers ()));
      catch (Exception e) {}
   }
}
```

A Constructor osztály

- Egy konstruktort reprezentáló osztály.
- Legfontosabb szolgáltatásai:
  - Lekérdezhetjük a konstruktor által **dobható kivételek**et.
  - Lekérdezhetjük a konstruktor **paramétereinek típusá**t.
  - **Létrehozhatunk** a konstruktort tartalmazó osztályból **egy példányt**, a megadott paraméterekkel. **A primitív típusú paraméterértékeket be kell csomagolni**.

```
public class Test {
   public Test (String a,int b) { ... }
}
...
Constructor c = Test.class.getConstructor (
   new Class[] { String.class,int.class });
Test t = (Test) c.newInstance (new Object[] { "a",new Integer (1) });
```

A Method osztály

- Egy metódust reprezentáló osztály.
- Legfontosabb szolgáltatásai:
  - Lekérdezhetjük a metódus által **dobható kivételek**et.
  - Lekérdezhetjük a metódus **paramétereinek típusá**t.
  - Lekérdezhetjük a metódus **visszatérési értékének típusá**t.
  - **Végrehajthatjuk** a metódust, a **megadott paraméterekkel**.
- Ha a metódus nem osztálymetódus, egy objektumot is meg kell adni.
- A **visszatérési értéket** egy Object referenciaként kapjuk vissza, a primitív típusú visszatérési értékek be vannak csomagolva.
- A metódus által **dobott kivételeket** InvocationTargetException-ba csomagolva kapjuk meg.

A Field osztály

- Egy mezőt reprezentáló osztály.
- Legfontosabb szolgáltatásai:
  - Lekérdezhetjük a mező értékét.
  - Új értéket adhatunk a mezőnek.
- Ha a mező nem osztálymező, a műveleteknél egy objektumot is meg kell adni.
- Minden primitív típushoz van külön setter / getter metódus, illetve van egy általános Object paraméterekkel operáló setter / getter is.

```
public class Test { private static int i = 1; }
...
Field f = Test.class.getDeclaredField ("i");
f.setAccessible (true);
f.setInt (null,12);
```

Az Array osztály

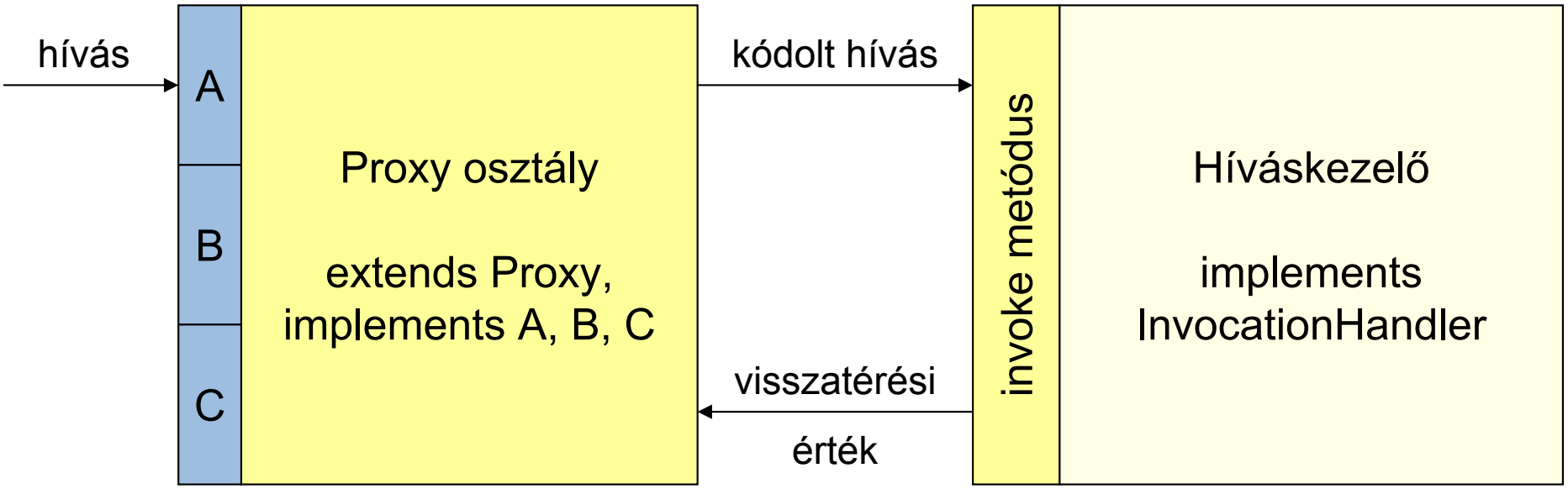
- **Tömbök dinamikus létrehozásá**ra szolgáló osztály.
- Szolgáltatásai:
  - Egy **elem értékének lekérdezése illetve beállítása** (primitív elemtípusokhoz és objektum típusokhoz is vannak setter / getter metódusok).
  - Tetszőleges elemtípusú és dimenziószámú **tömb létrehozása**.

```
public class Test {
   public static void main (String[] args) {
      String[][] s = (String[][]) Array.newInstance (
         String.class,new int[] { 2,3 });
      Array.set (Array.get (s,0),1,"alma");
      System.out.println (s [0][1]);
   }
}
```

Proxy osztályok I.

- A **proxy osztály**ok (dynamic proxy classes, vagy proxy classes) olyan dinamikusan létrejött osztályok, amelyek **futásidőben meghatározható interfészeket** (proxy interface) **implementálnak**.
- Az interfészek metódusaira érkező hívásokat a proxy osztály egy **híváskezelő** (invocation handler) osztályhoz irányítja.
- Proxy osztályt a java.lang.reflect.Proxy osztály segítségével hozhatunk létre.
- Ez a funkcionalitás Java-ban nem valósítható meg, még a Reflection API segítségével sem, mert az, hogy egy osztály milyen interfészeket implementál, nem változtatható futásidőben.
- A Proxy osztály úgy hoz létre futásidőben osztályokat, hogy dinamikusan legenerálja a bytekódjukat, és azt betölti egy ClassLoader segítségével.

Proxy osztályok II.



- Az interfészekre érkező hívásokat a proxy osztály továbbítja az InvocationHandler-t megvalósító híváskezelő **invoke** metódusának.
- Az invoke metódus megkapja a **proxy objektum**ot, a meghívott metódushoz tartozó **Method objektum**ot és a **paraméterek listáj**át. A **visszatérési érték** a proxy-n keresztül visszakerül a hívóhoz.

Proxy osztályok tulajdonságai

- A proxy osztályok legfontosabb tulajdonságai az alábbiak:
  - **public**, **final** és **nem abstract**,
  - az **egyszerű neve nem meghatározott**, de a $Proxy-val kezdődő nevek fenntartottak a proxy osztályok számára,
  - ha implementál egy nem-publikus interfészt, akkor **az adott interfész csomagjába kerül**, ha nem, **a csomag nem meghatározott**,
  - a **java.lang.reflect.Proxy** osztályból származik,
  - **egy konstruktora van**, amely egy InvocationHandler implementációt vesz át.

Proxy példányok tulajdonságai

- A proxy példányok legfontosabb tulajdonságai:
  - ha a p proxy példány implementál egy Ize interfészt, az (Ize) p cast művelet sikerül, és p instanceof Ize értéke true,
  - az Object-ben deklarált **hashCode**, **equals**, és **toString** metódusok hívásait a proxy **továbbítja a híváskezelőnek**, az Object többi metódusát nem, ezek úgy viselkednek, mint egy Object példány esetében.
- Ha több interfészben is szerepel ugyanaz a metódus, a legelsőként megadott, a kérdéses metódust deklaráló interfész metódusaként továbbítódik a hívás a híváskezelő felé, **az interfészek sorrendje** tehát **számít**.

Proxy osztályok létrehozása

- Proxy osztályokat a Proxy osztály **getProxyClass**, illetve **newProxyInstance** metódusaival hozhatunk létre:

```
Class pClass = Proxy.getProxyClass (
   ActionListener.class.getClassLoader (),
   new Class[] { ActionListener.class,MouseListener.class }
);

Proxy p = Proxy.newProxyInstance (
   ActionListener.class.getClassLoader (),
   new Class[] { ActionListener.class,MouseListener.class },
   new InvocationHandler () {
      public Object invoke (Object pxy,Method mtd,Object[] args) {
         ...
      }
   }
);
```

Proxy osztályok alkalmazása I.

```
public class Test extends JFrame {
   public Test () {
      JTextField field = new JTextField ();
      DocumentListener dl=(DocumentListener) Proxy.newProxyInstance (
         DocumentListener.class.getClassLoader (),
         new Class[] { DocumentListener.class },
         new InvocationHandler () {
            public Object invoke (Object p,Method m,Object[] a) {
               System.out.println ("Ne nyulka-piszka!");
               return null;
            }
         });
      field.getDocument ().addDocumentListener (dl);
      getContentPane ().add (field);
      pack ();
      setVisible (true);
   }
   public static void main (String[] args) { new Test (); }
}
```

Proxy osztályok alkalmazása II.

- Ugyanaz a hatás, EventHander segítségével:

```
public class Test extends JFrame {
   public Test () {
      JTextField field = new JTextField ();
      DocumentListener dl = (DocumentListener) EventHandler.create (
         DocumentListener.class,this,"documentEvent");
      field.getDocument ().addDocumentListener (dl);
      getContentPane ().add (field);
      pack ();
      setVisible (true);
   }

   public void documentEvent () {
      System.out.println ("Ne nyulka-piszka!");
   }

   public static void main (String[] args) { new Test (); }
}
```