V. Csomagok, kivételkezelés

I. Csomagolás

1. Csomagolási alapfogalmak

„Egy **csomag** (*package*) valamilyen szempontból összetartozó osztályok és interfészek csoportja. Csomagok használatával a program áttekinthetővé válik, egyszerűbb lesz a deklaráció azonosítása és védelme."

Egy Java program csomagok halmaza, csomagokban van a Java fejlesztőkörnyezet kódja és a programozó által írt alkalmazás is. A JDK szabványos osztálygyűjteménye például az *rt.jar* csomagrendszerben található.

A Java csomagszerkezete hierarchikus:

- a csomagok egymásba ágyazhatók
- tetszőleges mélységű csomagstruktúra építhető fel
- egy szintre akárhány osztály, interfész illetve csomag tehető
- a csomagnevekben az egyes szinteket pont választja el egymástól
- egy csomagba a logikailag összetartozó elemek csoportosítandók
    - pl.: a java csomag alatt lévő java.io csomagban az I/O műveletekkel kapcsolatos
    - míg a java.math csomagban a matematikai funkciójú osztályok kaptak helyet
- Vigyázat!!! A csomagokra való hivatkozás, noha az elnevezések ezt sugallják, nem rekurzív, vagyis a csomagok nem tartalmazzák egymást.
    - a java csomag például nem tartalmazza sem a java.io sem pedig a java.math csomagokat, vagy az azokban lévő osztályokat. A csomagok esetében tehát csak a nevek hierarchikusak, a csomagok maguk nem!
- Előfordulhat olyan csomag, amelyben nincsenek osztályok vagy interfészek. Például a java csomagban nincsenek osztályok, de alcsomagjai vannak.

A csomaghierarchiának egy egyértelmű könyvtárstruktúra feleltethető meg a tárolóeszközön. A csomagnevek és a könyvtárnevek párhuzamba állíthatók egymással, és a lefordított bájtkódok tárolásának is ezt a könyvtárstruktúrát kell követnie. Tehát a csomag fogalma logikai és fizikai szinten is értelmezhető.

2. Nevek

A Java megengedi, hogy két külön csomagban legyen két azonos nevű osztály. Emiatt a külön csomagban lévő osztályokat valamilyen módon meg kell tudni különböztetni. Az azonos nevű osztályok megkülönböztetésére szolgálnak az úgynevezett **minősített nevek** (qualified name). A minősített nevek az osztályt tartalmazó csomag teljes nevéből, és az osztály **egyszerű nevéből** (simple name) állnak össze, a csomagnevet az osztálynévtől **pont**tal választjuk el.

Csomag	Osztály egyszerű neve	Osztály minősített neve
`java.lang`	`System`	`java.lang.System`

Néhány fontos dolog, amire még figyelnünk kell a nevek kapcsán:

- **Vigyázat! Nincsenek relatív minősített nevek**, mert nincs aktuális, aktív csomag, vagy csomag érvényességi kör (scope)! A lang.System név tehát csak akkor értelmes, ha létezik egy lang teljes nevű csomag, amelyben van egy System nevű osztály. A java.lang.System osztályt ez a név semmiképpen nem jelölheti!

- Mivel a csomaghierarchia szintjeit egymástól és a minősített nevekben az osztálynévtől egyaránt a pont választja el, néhány megszorítást kell tennünk:
  - ha például egy prog.graph csomagnak van egy window alcsomagja, a prog.graph csomagban nem lehet window nevű osztály, vagy interfész, mert a prog.graph.window név kétértelmű lenne
  - ha ezt nem tartjuk be, fordítási idejű hibát eredményez
- A csomagok elnevezése tetszőleges, azzal a megkötéssel, hogy a **java** és **javax** kezdetű csomagnevek **fent vannak tartva** a Java API-nak.

3. Csomagok deklarációja

Egy Java forrásállomány összes típusdeklarációja (osztálya és interfésze) szükségképpen ugyanabba a csomagba kerül. Egy osztály vagy interfész a **package** kulcsszó segítségével helyezhetünk el egy csomagba:

```
package csomag;

public class A {}
```

- egy forráskódban **csak egy** package deklaráció lehet, a forráskód legelején
- ha nincs package deklaráció, akkor az adott osztály vagy interfész egy névtelen csomagba kerül
- a csomagot teljes útvonalával azonosítjuk a csomag gyökérkönyvtárától számítva

- a forráskód package deklarációjában szereplő csomagnév és a forráskód könyvtára meg kell hogy feleljenek egymásnak
- a csomagrendszert (könyvtárat) általában becsomagoljuk egy *JAR-állományba*

4. Hivatkozás osztályokra

Egy osztályra a **definíciójával megegyező csomagban** az egyszerű nevével hivatkozhatunk.

csomag/A.java
```
package csomag;

public class A {}
```

csomag/B.java
```
package csomag;

public class B {
   public void metodus () {
      new A ();                          // Azonos csomagban vagyunk
   }
}
```

Egy osztályra a **definíciójától eltérő csomagban** többféleképpen hivatkozhatunk. Az egyszerű név most nem jó, mert másik csomagban lévő osztályra akarunk hivatkozni.

csomag1/A.java
```
package csomag1;

public class A {}
```

csomag2/B.java
```
package csomag2;

public class B {
   public void metodus () {
      new A ();                          // Ez csomag2.A lenne!
   }
}
```

Módszerek a probléma megoldására:

- **minősített nevek** használatával

  Az osztály minden említésekor ki kell írni annak minősített nevét → rendkívül kényelmetlen lenne

csomag2/B.java
```
package csomag2;

import csomag1.*;                      // Igény szerinti importálás

public class B {
   public void metodus () {
      new A ();                        // Ez most csomag1.A
   }
}
```

- **egyenkénti import deklaráció**

  Ha egy csomagból csak egy-két osztályra van szükségünk, akkor az import deklarációk egyik fajtáját, az egyenkénti import deklarációt használhatjuk. Ilyenkor természetesen az összes használni kívánt osztályhoz egy-egy külön import deklaráció kell.

  `import csomagnév.[csomagnév [.csomagnév]…].Osztálynév;`
  `import csomagnév.[csomagnév [.csomagnév]…].Interfésznév;`

```
package csomag2;                                          csomag2/B.java

import csomag1.A;                                  // Egyenkénti importálás

public class B {
   public void metodus () {
      new A ();                                          // Ez most csomag1.A
   }
}
```

- **igény szerinti importálás**

  *Az import deklarációk ezen fajtáját akkor célszerű használni, ha egy csomagból több osztályt is használni szeretnénk. Ilyenkor elég az egész csomaghoz egy import deklaráció. A * jóker karakter szolgál annak jelzésére, hogy az adott csomag összes osztályát és interfészét importáljuk.*

  `import csomagnév.[csomagnév [.csomagnév]…].*;`

```
package csomag2;                                          csomag2/B.java

import csomag1.*;                                 // Igény szerinti importálás

public class B {
   public void metodus () {
      new A ();                                          // Ez most csomag1.A
   }
}
```

Fontos megjegyzések:

- Létezik az import deklarációknak egy különleges típusa, az automatikus importálás. Ez azt jelenti, hogy a `java.lang` csomag implicite mindig importált, vagyis olyan, mintha miden forráskód elején szerepelne az **`import java.lang.*;`** sor.
- Az importálás NEM rekurzív! Az import `p.*;` sor nem importálja a `p` csomag alcsomagjait és azok osztályait, csupán a p-ben lévő osztályokat és interfészeket.

- Névtelen csomagban lévő osztályra nem lehet minősített névvel hivatkozni, és az ott lévő osztályok import deklarációk segítségével sem elérhetők.

```
public class A {                                                    A.java
}                                  // Az A osztály a névtelen csomagban van
```

```
package csomag;                                                 csomag/B.java

import ???                      // Nem tudok a névtelen csomagra hivatkozni!

public class B {
   public B () {
      new A ();                  // Fordítási hiba: ez a csomag.A-t jelenti!
      new ???.A ();              // Nem tudok a névtelen csomagra hivatkozni!
   }
}
```

5. Osztályok láthatósága

A láthatóságról már volt szó a korábbi fejezetekben, de a csomagok ismeretében bővíthetjük a láthatósággal kapcsolatos ismereteinket.

Az osztályok illetve interfészek definíciójakor a *class* illetve *interface* kulcsszavak előtt hozzáférés-módosítókat (láthatósági módosítókat) adtunk meg:

- **public** : az osztály tetszőleges csomagban lévő osztály számára elérhető
- **nincs módosító (package private)** : az osztály csak a saját csomagja számára elérhető
- **private** : csak a deklarációt tartalmazó osztály hivatkozhat rá
- **protected** : az őt tartalmazó csomagban bárki hivatkozhat rá, más csomagokból csak az utódosztályok

```
package csomag1;                                              csomag1/A.java

public class A {}                                        // Publikus osztály
class B {}                                          // Package private osztály
private class C {}    // Fordítási hiba, itt nem használható a private
```

```
package csomag1;                                              csomag1/D.java

public class D {
   public D () { new B (); }              // OK, csomag1-ben látszik a B
}
```

```
package csomag2;                                              csomag2/E.java

public class E {
   public E () {
      new csomag1.A ();       // OK, az A minden csomag számára látható
      new csomag1.B ();        // Fordítási hiba, a itt nem látszik a B
   }
}
```

6. JAR állományok

Miután elkészítettünk egy Java programot vagy egy segédkönyvtárat, át kell adnunk azt a felhasználónak. Meglehetősen kényelmetlen dolog lenne, ha a felhasználónak osztálykönyvtárak sokaságát adnánk át, mert így a telepítés és a környezeti beállítás is nehézkes lenne. Sokkal kényelmesebb megoldás, ha a futtatáshoz szükséges állományokat összecsomagoljuk, és egyetlen állományként adjuk át a megrendelőnek.

Erre biztosít lehetőséget a **JAR** (***J***ava ***AR***chive) állományok használata, amelyek szabványos ZIP formátumú tömörített állományok (*zip* helyett *jar* kiterjesztéssel).

Egy JAR állomány tartalmazhat:

- **bájtkódokat** (class állományokat) amelyek osztályokat, interfészeket tartalmaznak
- **könyvtárakat** (amelyek fizikailag valósítják meg a csomagok hierarchiáját)
- **erőforrásokat** (képeket, dokumentumokat, hangokat…stb)

Bizonyos JAR állományok futtathatók, mások pedig nem. A futtatható JAR állományoknak kell hogy legyen egy belépési pontja, vagyis egy statikus main metódust tartalmazó fő-osztálya, amelyről a JAR aláírásállománya (manifest file) ad információt.

a) JAR készítése

JAR állományok összeállításához a JDK jar.exe programja használható:

Common JAR file operations

Operation	Command
To create a JAR file	`jar cf jar-file input-file(s)`
To view the contents of a JAR file	`jar tf jar-file`
To extract the contents of a JAR file	`jar xf jar-file`
To extract specific files from a JAR file	`jar xf jar-file archived-file(s)`
To run an application packaged as a JAR file (requires the `Main-class` manifest header)	`java -jar app.jar`
To invoke an applet packaged as a JAR file	`<applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height> </applet>`

(A JRE lib könyvtárában található **rt.jar** állomány tartalmazza a Java beépített osztálykönyvtárait)

II. Kivételkezelés

1. A kivétel fogalma

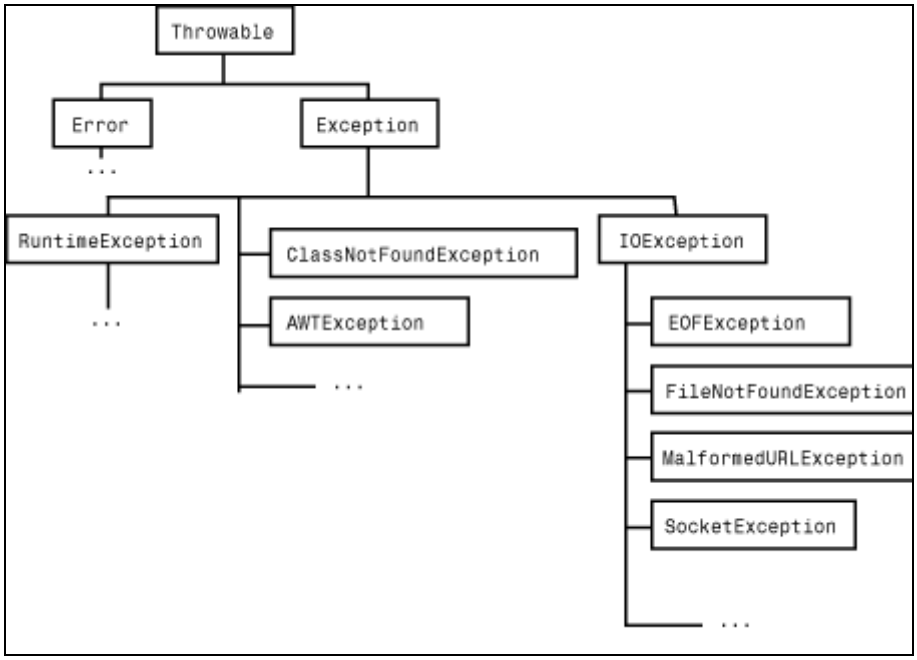
Sajnos a számítógépprogramok sohasem teljesen hibátlanok. A programozási nyelvek fejlesztését a hibátlan programokra való törekvés határozta és határozza meg. Egy ebben az irányban tett lépés a fordítók továbbfejlesztése, hogy azok a hibák jelentős részét már fordításkor felfedezzék. Az olyan modern programozási nyelvek, mint a Java, a programvégrehajtás fázisában is több támogatást nyújtanak.

Gondoljunk például a tömbökre. Ha a programvégrehajtás során a tömb egy nem létező elemére hivatkozunk, a Java értelmező egy *ArrayOutOfBoundsException* üzenetet és a hibás programsor számát kijelezve leállítja a program végrehajtását.

Amint a későbbiekben látni fogjuk, a Java nyelvi szinten támogatja a hibakezelést és ezt nevezzük kivételkezelésnek.

2. Mik azok a kivételek?

A hibakezelés úgynevezett kivétel (exception) objektumok segítségével történik, amelyek a kivételosztályok példányai. A kivételosztályok a Throwable osztályból öröklődnek. A különböző hibatípusok mindegyikéhez saját kivételosztály tartozik, ilyen például az ArrayIndexOutOfBoundsException , amely a tömbök indexelt hozzáféréséhez tartozik.



(Az *Error (rendszerhiba)* osztály példányai a Java futtató környezet belső hibái.)

3. Kivételek elkapása, kezelése

Szóhasználat:

- Egy kivétel keletkezik (*occurs*), dobódik (*is thrown*), vagy explicite dobjuk (*throw*)
- A kivétel kezelése (handling) úgy történik, hogy a dobott kivételt elkapjuk (*catch*)

try – catch szerkezet

A kivételek elkaphatók és kezelhetők, erre a **try – catch** szerkezetet használjuk, melynek általános formája a következő:

```
try {                                                    // try blokk
   ... utasítások ...
}
catch (KivételOsztály1 k) {                          // 1. catch blokk
   ... 1. típusú kivétel kezelése, k az elkapott kivétel objektum ...
}
catch (KivételOsztály2 k) {                          // 2. catch blokk
   ... 2. típusú kivétel kezelése, k az elkapott kivétel objektum ...
}
```

Felépítése:

- **try blokk**

  A try blokk tartalmazza a program normális logikáját tükröző utasításokat. Általában a try blokk futása során keletkeznek azok a kivételek, amelyeket el kell fognunk

- **catch blokk**

  Minden catch blokk egy-egy kivételkezelőt definiál. A blokk fejében paraméterként pontosan egy formális kivételobjektum van megadva. A catch blokk fogja kezelni az érkező kivételobjektumot.

A **try – catch** blokk végrehajtása az alábbiak szerint történik:

- Ha a try blokk utasításainak végrehajtása során **nem dobódott kivétel**, a try-catch blokk végrehajtása **normálisan befejeződik**.
- Ha **kivétel dobódott**, sorban megpróbáljuk a keletkezett kivétel objektumot **illeszteni a catch blokkok fejében szereplő osztályokra**. Illeszkedés: a kivétel osztály megegyezik a catch blokk fejében adott osztállyal, vagy annak leszármazottja (instanceof reláció).
- Ha **volt illeszkedés**, az adott **catch blokkot végrehajtjuk**, és a try-catch blokk normálisan befejeződik, illetve ha a catch blokkban újabb kivétel dobódik, kivétellel fejeződik be.
- Ha **nincs illeszkedés**, a try-catch blokk **kivétellel fejeződik be**.

***try – catch - finally* szerkezet**

Szükségünk lehet arra, hogy a try-catch blokk lefutásának eredményétől függetlenül végrehajtsunk egy kódrészletet. (A try-catch blokk normálisan is végetérhet, de kivétel is dobódhat belőle.) Erre szolgál a **finally blokk**. A catch blokkok után opcionálisan következhet egy darab finally blokk.

Az ilyen konstrukció neve **try-catch-finally blokk**:

```
try {                                                    // try blokk
   ... utasítások ...
} catch (KivételOsztály k) {                            // catch blokk
   ... adott típusú kivétel kezelése ...
} finally {                                             // finally blokk
   ... utasítások ...
}
```

Ha van finally blokk, akkor az minden körülmények között végrehajtódik, mindegy, hogy a végrehajtás normális volt-e, kiugrottunk-e a try blokkból, illetve hogy kezeltük-e a keletkezett hibát vagy sem. A finally blokkban szokás elvégezni a mindenképpen szükséges befejező tevékenységeket.