Programozási nyelvek – Java
Csomagok

Kozsik Tamás

Csomag

- Program tagolása
- Összetartozó osztályok összefogása
- Programkönyvtárak
  - Szabványos programkönyvtár

A triviálisnál nagyobb méretű programoknál általában már használunk csomagokat, amelyek segítségével a programot alkotó osztálydefiníciókat (illetve általánosabban: típusdefiníciókat) logikai rendbe szervezhetjük. A szabványos programkönyvtár esetén is jól megfigyelhető, hogy a logikailag összetartozó típusdefiníciókat egy csomagba sorolják, illetve a csomagok között is hierarchikus rendet vezetnek be (csomagok, alcsomagok).

A csomagok lehetőséget adnak arra, hogy az alapvető OOP egységeket, az osztályokat nagyobb logikai egységbe szervezve áttekinthetőbbé tegyük a kódbázist. Az előző előadások és gyakorlatok megmutatták azt, hogy csomagok készítése nélkül is megírhatjuk Java programunkat.

A `package` utasítás

```
package geometry;

class Point {
    int x, y;
    void move( int dx, int dy ){
        x += dx;
        y += dy;
    }
}
```

- Osztály (teljes) neve: `geometry.Point`
- Osztály rövid neve: `Point`

A forrásfájl elejére írt `package` utasítás jelzi azt, hogy az ebben a fordítási egységben található típusdefiníciók (jelen esetben egyetlen osztálydefiníció) a `geometry` nevű csomagba tartozzon. A csomagba tartozás azt is jelenti, hogy az osztály „igazi'', azaz *teljes* neve a csomagnévvel minősített `geometry.Point` legyen. A `class` kulcsszó után olvasható az osztály *rövid* neve: `Point`. A csomagon belül az osztályra a rövid nevével hivatkozhatunk, egyéb esetben a teljes nevet kell használnunk.

A Java csomag fogalma meglehetősen hasonlít a C++ névtér (namespace) fogalmához.

Hierarchikus névtér

```
package geometry.basics;

class Point {    // geometry.basics.Point
    int x, y;
    void move( int dx, int dy ){
        x += dx;
```

```
        y += dy;
    }
}
```

- Szabványos programkönyvtár, pl. `java.net.ServerSocket`
- `hu.elte.kto.teaching.javabsc.geometry.basics.Point`

Már a csomagnév is lehet összetett név, abban az értelemben, hogy névkomponensek ponttal elválasztott sorozata lehet. Ez egy hierarchikus elnevezési rendszert (névteret, angolul namespace-t) vezet be a programunkba. A példánkban a `Point` osztályt a `geometry.basics` csomagba írtuk, azaz a `geometry` csomag `basics` alcsomagjába. Így az osztály teljes neve `geometry.basics.Point` lesz.

A hierarchikus névtér segítségével egész nagy programokat is szépen tudunk strukturálni. Ezt a szemléletet tükrözi a szabványos programkönyvtár is. A standard API (Application Programming Interface, azaz alkalmazásprogramozói felület) nagyrészt a `java` csomagban van, bár más csomagokat is találhatunk benne. A `java` csomagon belül olyan alcsomagokat láthatunk mint például a hálózatos alkalmazások fejlesztését támogató `net` alcsomag, vagy az input-output kezelésére való `io` alcsomag. A csomagok logikai egymásba ágyazása még mélyebb is lehet, például az `io` alcsomagon belül találjuk a `zip` alcsomagot (azaz `java.io.zip`), mely ZIP-tömörítésű állományok kezeléséhez használható.

A csomagoknak van még egy szerepük a Java programozói társadalomban. Ha több helyről szedjük össze a programunkat alkotó osztálydefiníciókat, például különböző open-source könyvtárakat is felhasználunk a programunk elkészítéséhez, akkor el kell kerülnünk a névütközéseket. Ökölszabályként az mondható, hogy egy adott programban minden típus nevének egyedinek kell lennie. (Pontosabban: minden, ugyanazzal az osztálybetöltővel betöltött típus nevének.) Ezt a programozók gyakran úgy érik el, hogy az általuk írt, mások általi felhasználásra tervezett könyvtárakat olyan csomagba teszik, amelynek a nevébe belekódolnak egy globálisan egyedi azonosítót, illeszkedve az Interneten használt domain nevekhez. Például én, ha az itteni `Point` osztály a nagyvilág szíves felhasználására készíteném, akkor a `hu.elte.kto` előtaggal garantálnám, hogy a mások által fejlesztett típusoktól eltérő, egyedi névvel rendelkezzen, és a saját magam által fejlesztett egyéb típusoktól is megkülönböztetném azzal, hogy az adott projekt nevét is belekódolnám a csomagnévbe. Ez az az idióma, amely miatt gyakran találkozunk igen hosszú nevű csomagokkal.

Fordítás, futtatás

- Munkakönyvtár
  (working directory)
- Hierarchikus csomagszerkezet → könyvtárszerkezet
- Fordítás a munkakönyvtárból
  - Fájlnév teljes elérési úttal
- Futtatás a munkakönyvtárból
  - Teljes osztálynév

```
$ ls -R
.:
geometry

./geometry:
basics

./geometry/basics:
Main.java  Point.java
$ javac geometry/basics/*.java
$ ls geometry/basics
Main.class  Main.java
Point.class  Point.java
$ java geometry.basics.Main
$
```

A csomagokba szervezett típusdefiníciók a forrásfájlok elhelyezésére is megkötést jelentenek, valamint a fordító és a virtuális gép meghívására is kihatnak. A típusdefiníciókat egy, a teljes nevüket tükröző, a megfelelő alkönyvtárban elhelyezett forrásfájlokba kell írnunk. Ha a projektünk *aktuális munkakönyvtárát* tekintjük kiindulási alapként (lehet ez pl. Linuxon a `/home/kto/projects/bscjava` vagy Windowson a `C:\Users\kto\projects\bscjava`), akkor ezen könyvtáron belül ki kell építenünk azon alkönyvtárak hierarchikus rendszerét, amelyek a projektben használt hierarchikus csomagneveket tükrözik, és ebben az alkönyvtárrendszerben kell dolgoznunk. Induljunk ki abból a példából, hogy a `geometry.basics.Point` osztályra van szükségünk. A munkakönyvtárunk tartalmazni fog egy `geometry` alkönyvtárat, az tartalmazni fog egy `basics` alkönyvtárat, és abban lesz a `Point.java` fájl.

Fordítani a projekt munkakönyvtárából fogunk, és nem *cd*-zünk be a forrásfájlt tartalmazó alkönyvtárba! A fordító meghívásánál a fájl relatív elérési útját adjuk meg a munkakönyvtárhoz képest. Ez a relatív elérési út a csomagnévvel minősített, teljes osztálynévnek felel meg. A fordító (ha másképp nem rendelkezünk) a lefordított állományt a „helyére'' teszi, azaz ugyanabba az alkönyvtárba, ahol a forrásfájl is található.

Csomagunk nem csak a `Point`, hanem a `Main` osztályt is tartalmazza, melyet szintén lefordítottunk. Feltéve, hogy ez utóbbbi tartalmazza a megfelelő `main` metódust, le is futtathatjuk a programunkat. A futtatás ismét a munkakönyvtárból kell történjen. A virtuális gépnek a teljes osztálynevet fogjuk megadni.

Látható tehát, hogy a fordító és a virtuális gép a számára szükséges típusdefiníciókat tartalmazó bájtkódállományokat a típus teljes neve alapján keresi meg. A csomagnév elárulja, hogy milyen alkönyvtárban, a rövid név pedig azt, hogy milyen nevű (`.class` kiterjesztésű) fájlban van az a kód, amit keres.

Fordítás: Java és C

```
$ ls geometry/basics
Main.java  Point.java
$ javac geometry/basics/Point.java
$ ls geometry/basics
Main.java  Point.class  Point.java
$ javac geometry/basics/Main.java
$ ls geometry/basics
Main.class  Main.java  Point.class  Point.java
$ java geometry.basics.Main
$
```

Fontos itt kiemelni, hogy szemben a C-vel, itt az egyes fordítási egységek fordítása nem független egymástól. A Java nyelv alaposabban ellenőrzi az egyes egységek közötti kapcsolatokat. Egy fordítási egység lefordításánál az abban használt típusdefiníciók helyes használatát is ellenőrzi. A C nyelvben a fordítási egységek közötti függőségeket nem ellenőrzi a fordító. A fejállományok ügyes használatával a programozók ki tudnak kényszeríteni egy ellenőrzést (ugyanazt fejállományt, mely tartalmazza egy egység külső interfészét, include-oljuk az egység definíciójába is és a felhasználó kódba is). Ez egy workaround, ami segít minket abban, hogy fordítási időben megtaláljuk az inkonzisztenciákat. A C nyelv fordítási elvét *independent compilationnek* szokás nevezni. A fordítási egységek közötti inkonzisztenciák fordítási idő helyett inkább csak szerkesztési (linking) időben kerülnek kimutatásra.

A Java nyelvben, mint láttuk, más mechanizmus van: ezt *separate compilationnek* hívják. Egy egység fordítása során a használt fordítási egységek kódját (egész pontosan a használt fordítási egység kódjából kiolvasható interfészinformációkat) összeveti a fordító a használat módjával, így már fordítási időben kiderül az inkonzisztens használat. Ez fontos is, hiszen a Java esetében alapvetően nincs is statikus szerkesztés.

Az, hogy a class-fájlok nevei és a típusnevek szinkronban vannak egymással (beleértve az alkönyvtárak és csomagok neveit), garantálja azt, hogy fordítás közben (és majd később, futtatás közben is) a `javac` fordító (illetve a `java` parancs) megtalálja majd azt a class-fájlt, amire szüksége van, ami tartalmazza a használni kívánt osztály kódját.

Rekurzív fordítás

```
$ ls geometry/basics
Main.java  Point.java
$ javac geometry/basics/Main.java
$ ls geometry/basics
Main.class  Main.java  Point.class  Point.java
$ java geometry.basics.Main
$
```

A `javac` fordító még egy technikával igyekszik kényelmesebbé tenni a programozó munkáját. Ha egy fordítási egység fordításánál észreveszi, hogy az egység használ egy olyan osztályt, amely még nincs lefordítva (így nem tudja azonnal ellenőrizni, hogy a használat helyes-e), akkor rekurzívan azt a használt osztályt is megpróbálja lefordítani. Ehhez az kell, hogy az osztályt tartalmazó forrásfájlt megtalálja. Ebben segíthet az, hogy ha a forrásfájl neve is megegyezik a tartalmazott osztály nevével.

Nemsokára látni fogjuk, hogy bizonyos esetekben nem kell, hogy a fájlnév megegyezzen a tartalmazott típus nevével. Ha élünk ezzel a lehetőséggel, akkor egyben elveszítjük azt a kényelmi funkciót, amit a rekurzív fordítás kínál – a fordító nem lesz képes megtalálni azt a forrásfájlt, amit le kellene fordítani, hogy előálljon az a class-fájl, amivel az eredetileg lefordítani kívánt egységben történt használatot össze lehet vetni.

Névtelen csomag

Default/anonymous package

- Ha nem írunk `package` utasítást
- Forrásfájl közvetlenül a munkakönyvtárba
- Kis kódbázis esetén rendben van

Ha egy forrásfájl elején nincs `package` utasítás, a forrásfájlban lévő osztály (illetve általában: osztályok, sőt típusok) egy speciális csomagba kerül(-nek). Kisebb méretű programoknál ez egy jól alkalmazható technika, a gyakorlatban azonban a Java programozók szinte mindig használnak explicit módon csomagokat.

A névtelen csomag használata esetén a forrásfájl (és a lefordított bájtkód) a munkakönyvtárba kerül.

Láthatósági kategóriák

- `private` (privát, rejtett)
  - csak az osztálydefiníción belül
- semmi (félnyilvános, package-private)
  - csak az ugyanabban a csomagban lévő osztálydefiníciókban
- `public` (publikus, nyilvános)
  - osztály is
  - tagok, konstruktor is

Csomagokat használó típusdefiníciók esetén az eddig megismert szabályokat a láthatósággal kapcsolatban ki kell bővítenünk. (Egy későbbi előadáson pedig majd még tovább bővítjük.) Meg kell különböztetnünk három láthatósági kategóriát. Ezek közül a `public` kulcsszóval jelölt kategória a csomagokon átívelő nyilvánosságot biztosítja. Ha egy típust (pl. osztályt) félnyilvános láthatósággal definiálunk, akkor az csak az ugyanabban a csomagban definiált egyéb típusokban (osztályokban) hivatkozhatunk.

Ugyanez igaz egy félnyilvános láthatóságú konstruktorra, mezőre vagy metódusra is.

Nyilvános és rejtett tagokat tartalmazó nyilvános osztály

```
package hu.elte.kto.javabsc.eloadas;

public class Time {
```

```
    private int hour;                 // 0 <= hour < 24
    private int minute;               // 0 <= minute < 60
    public Time( int hour, int minute ){ ... }
    public int getHour(){ return hour; }
    public int getMinute(){ return minute; }
    public void setHour( int hour ){ ... }
    public void setMinute( int minute ){ ... }
    public void aMinutePassed(){ ... }
}
```

Egy köznapi osztálydefiníció tehát úgy szokott kinézni, hogy valamilyen csomagban kerül elhelyezésre. A belső állapotot leíró adattagok jellemzően `private`-ok (megfelelő nylvános setterrel és getterrel), valamint nyilvános konstruktorral és metódusokkal van felszerelve. Emellett lehetnek benne privát (vagy akár félnyilvános) segédműveletek. A gyakorlatban a félnyilvános konstruktorokat és tagokat nem nagyon gyakran használják, félnyilvános osztályokkal viszont gyakrabban találkozhatunk.

Több csomagból álló program

hu/elte/kto/javabsc/eloadas/Time.java

```
package hu.elte.kto.javabsc.eloadas;

public class Time {
    ...
}
```

Main.java

```
// a névtelen csomagban

class Main {
    public static void main( String[] args ){
        hu.elte.kto.javabsc.eloadas.Time morning = new Time(6,10);
                                  // fordítási hiba --^
    }
}
```

Ha a programunkat alkotó osztálydefiníciók különböző csomagokban vannak, akkor több dologra is oda kell figyelnünk. Egyrészt a láthatóságra, hiszen a különböző csomagokban lévő osztályok csak a publikus osztályokat, konstruktorokat, tagokat láthatják egymásból. Másrészt a típusnevekre, hiszen alapvetően a teljes nevet (a csomagnévvel minősített nevet) kell használni. Ebből a szempontból a konstruktor neve is típusnévnek számít, ezért a fenti kód fordítási hibát tartalmaz. A konstruktorhívás helyesen: `new hu.elte.kto.javabsc.eloadas.Time(6,10)`.

Egy csomagon belül

hu/elte/kto/javabsc/eloadas/Time.java

```
package hu.elte.kto.javabsc.eloadas;

class Time {
    ...
}
```

hu/elte/kto/javabsc/eloadas/Main.java

```
package hu.elte.kto.javabsc.eloadas;
```

```
class Main {
    public static void main( String[] args ){
        Time morning = new Time(6,10);
        ...
    }
}
```

Ha a típusdefiníciók ugyanabban a csomagban találhatók, akkor természetesen a félnyilvános definíciók is láthatók egymásból, és a teljes név használatától is eltekinthetünk.

Egy forrásfájlban több típusdefiníció

hu/elte/kto/javabsc/eloadas/Time.java

```
package hu.elte.kto.javabsc.eloadas;

public class Time {
    ...
}

class Main {
    public static void main( String[] args ){
        Time morning = new Time(6,10);
        ...
    }
}
```

A forrásfájlokba mindezidáig egy-egy típusdefiníciót írtunk, de már utaltunk arra, hogy ez nem feltétlenül kell, hogy így legyen. Ebben a példában azt látjuk, hogy ugyanabban a forrásfájlban két osztálydefiníció került. A `package` utasítás a forrásfájl elején mindkettőre vonatkozik, egy forrásfájlon belül minden típus ugyanabba a csomagba kerül.

A forrásfájl neve meg kell hogy egyezzen a forrásfájlban esetlegesen megtalálható publikus típus rövid nevével. Ebből az is következik, hogy egy forrásfájlban legfeljebb egy publikus típust definiálhatunk. Ha a forrásfájlban nincs publikus típus, akkor persze mindegy, hogy milyen fájlnevet választunk.

A gyakorlatban általában csak egy típusdefiníciót írunk egy fájlba, és a nevét akkor is egyeztetjük a fájlnévvel, ha nem publikus.

Az `import` utasítás

hu/elte/kto/javabsc/eloadas/Time.java

```
package hu.elte.kto.javabsc.eloadas;

public class Time {
    ...
}
```

Main.java

```
import hu.elte.kto.javabsc.eloadas.Time;

class Main {
    public static void main( String[] args ){
        Time morning = new Time(6,10);
        ...
    }
}
```

A teljes név használata – főleg, ha extrém hosszú a csomagnév, mint ebben a példában – időnként kényelmetlen. Az `import` utasítás megadja a lehetőséget arra, hogy egy teljes név helyett a rövid nevet használhassuk.

Minősített név feloldása

- Osztály teljes neve helyett a rövid neve
- `import hu.elte.kto.javabsc.eloadas.*;`
- Nem tranzitív
- A `java.lang` csomag típusait nem kell
- Névütközés: teljes név kell
  - java.util.List
  - java.awt.List

Az `import` utasítással lehetőség nyílik arra, hogy egy adott típust, illetve a csillagos `import` utasítást használva: egy adott csomag minden típusát rövid névvel használjuk a teljes név helyett. A csillag nem tranzitív, abban az értelemben, hogy a fenti példában az `import` utasítás nem teszi a `hu.elte.kto.javabsc.eloadas` csomag *alcsomagjainak* típusait rövid néven elérhetővé.

Arra sem használható az import utasítás, hogy megszabaduljunk a teljes név első néhány névkomponensétől. Például az

```
import hu.elte.kto.*;
```

utasítás után sem érhetjük el „félig-meddig rövid'' névvel, `javabsc.eloadas.Time` névvel a `hu.elte.kto.javabsc.eloadas.Time` osztályt. Egy osztályt vagy teljes névvel vagy rövid névvel hivatkozhatunk, köztes megoldás nincs.

Ha több `import` utasítąst használunk, előfordulhat, hogy különböző csomagokból különböző, de azonos rövid nevű típusok válnak rövid névvel elérhetővé. Ez önmagában nem baj. Viszont a rövid név nyilván nem használható ebben az esetben valamelyik típus megnevezésére: a rövid név nem egyértelmű. A névütközésben részt vevő típusokat természetesen továbbra is egyértelműen azonosíthatjuk a teljes nevükkel.

És általában is igaz az, hogy az `import` utasítás lehetőséget ad arra, hogy egyes típusokat rövid névvel haszáljunk, de ezzel a lehetőséggel nem kötelező élni: a teljes név használata bármikor megengedett.

Fordítási egység szerkezete

- opcionális `package` utasítás
- 0, 1 vagy több `import` utasítás
- 1 vagy több típusdefiníció

Egy fordítási egység elején szerepelhet legfeljebb egy `package` utasítás. Ezt követheti akárhány `import` utasítás. Ezután jönnek a típusdefiníciók. A `package` és `import` utasítások a fordítási egységben (forrásfájlban) szereplő összes típusra vonatkoznak.

Jó lenne, ha az `import` utasítást ennél rugalmasabban lehetne használni, például egy blokkban kiadott `import` utasítás csak arra a blokkra engedélyezné a rövid név használatát a teljes név helyett.

javac kapcsolók

```
-d <directory>     Specify where to place generated class files


--source-path <path>, -sourcepath <path>
                   Specify where to find input source files


--class-path <path>, -classpath <path>, -cp <path>
                   Specify where to find user class files...
```

A fordító legegyszerűbb használata az, amikor a munkakönyvtárban vannak (a csomagoknak megfelelő alkönyvtárakba szervezett) forrásfájlok, és melléjük helyeztetjük el a bájtód-állományokat is. Ennél szofisztikáltabb megoldást tesznek lehetővé a fordítóprogram kapcsolói. A `-d` kapcsoló után megadhatunk egy könyvtárnevet: ide teszi (a megfelelő könyvtárszerkezetet létrehozva) a bájtkód-állományokat a fordító. A `-sourcepath` kapcsoló után könyvtárnevek egy sorozatát (ún. útvonalkifejezést) adhatunk meg. Ott keresi a forrásszövegetek a fordító.

A legizgalmasabb a három fenti kapcsoló közül a `-classpath`, mely nem csak a fordító, hanem a virtuális gép esetén is használható.

classpath

```
javac -classpath .:/usr/lib/java:/opt/java/myfiles.jar \\
      geometry/basics/Point.java

java -classpath .:/usr/lib/java:/opt/java/myfiles.jar \\
      geometry.basics.Main
```

- Ha kell a `colors.RGB` osztály:
  - `./colors/RGB.class`
  - `/usr/lib/java/colors/RGB.class`
  - `/opt/java/myfiles.jar`-ban `colors/RGB.class`
- Windows alatt: `.;C:\Users\kto\mylib;D:\myfiles.jar`
- `CLASSPATH`

Mind a fordító, mind a virtuális gép ott keresi a class-fájlokat, ahol a classpath mutatja. Ha a `geometry.basics.Point` osztály lefordításához szükség van a `colors.RGB` osztály kódjára, akkor a fent látható módon keresi a fordító a kódot tartalmazó class-fájlt. Ugyanezt lehet mondani arról az esetről, amikor a `geometry.basics.Main` osztály futtatása során a virtuális gépnek szüksége van a `colors.RGB` osztályra.

A classpath kifejezésben nem csak könyvtárnevek, hanem `.jar` vagy `.zip` állományok is megadhatók. Abban is tud keresni mind a `javac`, mind a `java`.

Windows alatt az útvonalkifejezésekben az egyes könyvtár- és fájlneveket nem kettősponttal, hanem pontosvesszővel választjuk el, és az elérési utakban a névkomponenseket nem per-, hanem rep-jel (backslash) választja el.

Ha a `javac`, illetve `java` parancsok meghívása során nem használjuk a `-classpath` kapcsolót, akkor (de csak akkor!) ez a két parancs a `CLASSPATH` környezeti változó tartalma alapján keresi a class-fájlokat. Ha ez a környezeti változó nincs beállítva, akkor a classpath az aktuális munkakönyvtárat (`.`) fogja csak tartalmazni.

jar fájlok

- Java Archive
- ZIP-tömörítésű fájl
- `jar` parancs az SDK-ban

A jar-fájlok olyan tömörített állományok, melyek könyvtárszerkezetben elhelyezett fájlokat tartalmaznak. Jellemzően ilyen állományokban szokás egy programkönyvtárat nyilvánosságra hozni a Java világban.