# 5 előadás

## Osztályszintű elérés (statikusság)

Egy osztályban definiált attribútumok és műveletek alapvetően az osztályból példányosított objektumok sajátjai lesznek, csak az objektumokon keresztül érhetőek el

Kivéve azokat az attribútumokat és műveleteket, amelyeknek osztályszintű elérést adunk meg, azaz amelyeket statikussá teszünk

A statikus attribútumok és műveletek függetlenek az illető osztályból példányosított objektumoktól, helyettük csak az osztályhoz tartoznak, az osztályon keresztül érhetőek el

## Osztályszintű attribútumok

Olyan attribútumok, melyek az objektumok helyett az osztályhoz tartoznak, objektumok nélkül is elérhetőek

Az osztályukhoz tartozó egyetlen példányuk létezik, az összes objektum ugyanezt az egy értéket használja

Szokásos használatuk:

* Konstans értékek
* Jelzőértékek (flag-ek)
* Az adott osztályba tartozó objektumok közösen használható értékei

## Osztályszintű műveletek

Olyan műveletek, melyek az objektumok helyett az osztályhoz tartoznak, objektumok nélkül is elérhetőek

Meghívható csupán az osztályra hivatkozva is

Az osztályszintű műveletek kizárólag osztályszintű, azaz statikus attribútumokat és műveleteket használhatnak

Szokásos használatuk:

* Segédfüggvények
* Matematikai függvények

## Segédosztályok

A segédosztályok (utility class) kizárólag statikus attribútumokat és műveleteket tartalmaznak, általában nem szokás objektumokat példányosítani belőlük



Nem lehet példányosítani, mert konstruktora rejtett: private Math**(){}**

Konstansok: public static final double PI**=**3.14**;**

Segédfüggvények: public static double sqrt**(**double a**){**...**}**

## Osztályszintű elérés példa

\Tech\stb\diak\prt1ea5\_EtrPersons.zip

Statikus adattag: private static char yearSign**;**

Statikus függvény: private static String generateEtr**(**EtrPerson aThis**)** **{**

A nem statikus attribútumok eléréséhez objektum kell: sb**.**append**(**aThis**.**firstName**.**toUpperCase**().**charAt**(**1**));**

A statikusok eléréséhez objektum nem kell: counters**.**put**(**nameFract**,** count**);**

A $Map<String, Integer>$ az asszociatív tömb. Ugyan úgy interfész, mint a $List<>$, ezért inicializálni kell.
Elhelyezés vagy felülírás: counters**.**put**(**nameFract**,** count**);**

(Konstans változókat csupa nagybetűvel szokás elnevezni.)

### Statikus konstruktor

static **{**

 counters **=** **new** TreeMap**<>();**

 yearSign **=** 'T'**;**

 INSTITUTE\_CODE **=** "ELTE"**;**

**}**

### Teszt

List**<**EtrPerson**>** etrPersons **=** **new** ArrayList**<>();**

etrPersons**.**add**(new** EtrPerson**(**"Teszt"**,** "Elek1"**,** "Informatikai Kar"**));**

etrPersons**.**add**(new** EtrPerson**(**"Teszt"**,** "Elek2"**,** "Informatikai Kar"**));**

etrPersons**.**add**(new** EtrPerson**(**"Teszt"**,** "Elek3"**,** "Bölcsészet Tudományi Kar"**));**

etrPersons**.**add**(new** EtrPerson**(**"Szabó"**,** "Teréz"**,** "Informatikai Kar"**));**

etrPersons**.**add**(new** EtrPerson**(**"Szabó"**,** "Tóbiás"**,** "Informatikai Kar"**));**

etrPersons**.**add**(new** EtrPerson**(**"Székely"**,** "Tihamér"**,** "Informatikai Kar"**));**

**for** **(**EtrPerson etrPerson **:** etrPersons**)** **{**

 System**.**out**.**println**(**etrPerson**);**

**}**

## Obfuscation

Bájtkódok: Egy osztály lefordításának eredménye, bináris formátumban. Minden információ megvan benne, tehát vissza lehet fejteni.

Ha hibát ír ki a program, ne lássa a felhasználó, hol történt a hiba. Ne lehessen visszafejteni (decompile) a programot.

## Kivételkezelés

Kivétel (exception): olyan hiba (kivételes helyzet), melynek oka nem a futó programban keresendő, hanem külső körülményre vezethető vissza

Kivételkezelés (exception handling): egy esetlegesen fellépő kivétel futás közbeni megoldása – akár a hiba jelzése formájában, akár a hiba kijavítása formájában

Például: a felhasználó rossz jelszót adott meg – a program jelezze számára a hibát és kérdezze meg újra a jelszót

Java-ban a kivételkezelés kivétel (Exception, Runtime​Exception) típusú objektumok használatával valósul meg

Ha a program futása közben kivétel lép fel, de a program az adott helyen nem képes kezelni a kivételt, akkor eldob egy kivétel típusú objektumot

A kivétel objektum a függvényhívásokon keresztül visszafelé vándorol, egészen addig, amíg egy kivételkezelő blokk el nem kapja, és nem kezeli

Ha a kivétel kijut egészen a virtuális gépig, akkor a program futása leáll

Minden Exception típusú kivételt muszáj vagy kezelni, vagy jelezni a továbbadását

A Runtime​Exception (RTE) típusú kivételeket nem kötelező sem elkapni, sem kezelni, de egy jól működő program ezeket is kezeli

### Saját kivételtípus definiálása

Nem muszáj, de javasolt.

class EtrException **extends** Exception **{**

 public EtrException**(**String message**){super(**message**);}**

**}**

### Kivétel eldobása

Ha AA-ZZ minden azonosító elfogyott.

**throw** **new** EtrException**(**"Namespace full"**);**

Ha helyben tudjuk kezelni, nem kell kivételt dobni.

### Kivétel továbbengedésének jelzése

A kiváltó függvényben jelezni kell, hogy ilyet dobhat.

private static String generateEtr**(**EtrPerson aThis**)** **throws** EtrException**{**

### Kivétel kezelése

**try {**

 etrPersons**.**add**(new** EtrPerson**(**"Teszt"**,** "Elek"**,** "IK"**));**

**} catch (**EtrException e**) {**

 System**.**err**.**println**(**e**.**getMessage**());**

**}**

Ilyet alkalmazunk RTE esetén is.

A $try$ blokkon belül mindig a lehető legkevesebb kód legyen.

Az összes lehetséges előforduló kivételt kezeljük külön-külön.

Ne $try$ blokkon belül deklaráljuk azt a változót, amit azon kívül is szeretnénk használni. Deklaráljuk előtte, definiáljuk benne. Minden végrehajtási ágon legyen értéke a változónak. Ha a syntax checker mégis panaszkodik (pl. System**.**exit**(**1**);** után nem látja, hogy meg fog állni az összes végrehajtási ág), akkor trükközni kell. Vagy $null$-t adunk kezdőértéknek, vagy $return$ az $exit$ után.

### Finally blokk

Mindig lefut. Akkor is, ha nincs kivétel, akkor is, ha van. Kódismétlést spórolunk meg vele.

**try{**...**}** **catch(**...**){**...**}** **finally** **{**/\*mindig lefut\*/**}**

Például oos**.**close**();** mindig le kell, fusson, ezért a $finally$ blokkba rakjuk.

## Fájlkezelés

Emlékeztető: mit jelent a fájlkezelés? Mikor és mire használjuk?

Most a lényeg: milyen Java osztályok segítségével és hogyan működik?

A fájlkezelés absztrahálása:

* $java.io.InputStream:$ byte-csatornák olvasásához
* $java.io.OutputStream:$ byte-csatornák írásához
* $java.io.Reader:$ karaktercsatornák olvasásához
* $java.io.Writer:$ karaktercsatornák írásához

Az előbbi absztrakt osztályokból származnak a konkrét csatornákhoz használatos osztályok, így pl. fájlkezelésre:

* $java.io.FileInputStream:$ byte-ok olvasása egy fájlból
* $java.io.FileOutputStream:$ byte-ok írása egy fájlba
* $java.io.FileReader:$ karakterek olvasása egy fájlból
* $java.io.FileWriter:$ karakterek írása egy fájlba

Ezek mindössze a megnyitott fájl bájtjainak/karaktereinek egyenkénti olvasására/írására használhatóak.

## Az előbbieket érdemes összetettebb adatkezelésre alkalmas objektumokkal kombinálni

### Bináris

Ez 8 bájt lesz:



DataOutputStream dos**;**

dos**.**writeUTF**(null);**

Ne felejtsük el bezárni.

### Szöveg

Szöveget FileWriter-re épített PrintWriter segítségével lehet kiírni:



Ha egy kombinált csatornát bezárunk, az az alatta levő csatornákat is bezárja.

Ez egy pufferesen kezelt csatorna. Van **.**append**()**.

Mi a különbség a kettő között? DOS binárisan, PW szövegesen kezeli az adatokat.

DOS binárisan írja ki a $20$-at (8 byte), PW meg kiírja, hogy "20".

### ZIP

Tömörített $zip$ fájlok beolvasására van a ZipInputStream:



Bejegyzések között a **.**getNextEntry**();** lépked.

Felhasználásnál nem az $entry$-t kell átadni, hanem a $ZipInputStream$-et. Miért?

Erre még építhetünk harmadikat, hogy ne csak bájtonként dolgozzunk.

### Tokenized data beolvasás

Nem származik se az input​stream-ből, se a reader-ből.

Rá lehet építeni bármelyik stream-re.

Token – elem.

A default elválasztó a whitespace (szóköz, tabulátor, enter).



Most az sc**.**hasNext**()** csak annyit néz, hogy elértünk-e a végére. Cikluson belül pedig ellenőrzi, hogy várt típusú adat jön-e.

A **.**Next**()** függvénynek is vannak változatai, aminek segítségével különböző típusokat tudunk beolvasni.

### Kivételkezelés

Mindig figyeljünk oda. Fájlkezelésnél mindig akarunk kivételeket dobni:

IOException

Ezt kötelező kezelni.

## Serialization (Sorosítás)

Lehetőség van teljes Java objektumok fájlba való kiírására

Ehhez a virtuális gép sorosítja az adott objektumot, ezért az objektum osztályának meg kell valósítania a $Serializable$ interfészt. Ez rekurzívan érvényes az objektum összes adattagjára is

Rekurzív adatszerkezetek, (pl. gráf) sorosítása végtelen ciklusba fullad, ha nem jól csináljuk.

Amit szeretnénk később sorosítani, azt meg kell jelölni:

public class EtrPerson **implements** Serializable**{**

Ez nem implementál semmit, csak jelzi a virtuális gépnek, hogy sorosítani szeretnénk.

A sorosított adatot általában Object​Output​Stream-mel szoktuk kiíratni:



Rendelkezik minden művelettel, mint a Data​Output​Stream, de ezen felül kap egy $writeObject$-et, amit használtuk.

Primitív típusok mindig Serializable.

Végig kell nézni a teljes adatszerkezetünket, hogy minden eleme Serializable-e.

## Deserialization (Sorosított adat visszafejtése)

A sorosított adatok természetesen a fájlokból beolvashatóak, visszafejthetőek objektumokká

Ehhez teljesülni kell a következő feltételeknek:

A beolvasott objektum tényleg olyan típusú legyen, mint amire számítunk

Legyen betöltve a beolvasott objektum osztálya a virtuális gépbe – pontosan az az osztály legyen az. (Különben ClassNotFoundException kivételt dob.)

ObjectOutputStream párja: ObjectInputStream, visszatérési értéke $Object$.

Ezt castolni érdemes/kell megfelelő típussá.



Ha az osztályt később megváltoztatjuk, pl. bevezetünk egy új adattagot, akkor meghal (InvalidClassException). Mert megváltozott az $UID$. Új függvény bevezetése általában nem okoz gondot.

Programfejlesztés során ez felmerülhet, a korábban sorosított adatok elvesznek.

Előadás vége.