JAVA TÉMÁK AZ ALAPOKON TÚL

Vállalati Információs Rendszerek

© 2017-2026, Dr. Hegedűs Péter

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

JAVA REFLECTION

- Egy Java technika
- Lehetővé teszi az osztályok, interfészek, mezők és metódusok futás időben történő vizsgálatát
  - Nem kell tudnunk az adott elem nevét fordítási időben
  - Példányosíthatunk objektumot, meghívhatunk metódust, lekérdezhetünk/beállíthatunk mező értéket
    - Mindezt akár úgy, hogy az adott elem neve egy dinamikusan összeállított string-ből jön
- Hasznos többek között például
  - JavaBean-ek feldolgozásához
  - Annotációk kezeléséhez
    - Pl. JUnit tesztek

CLASS OBJECT

- Létezik egy speciális típus, Class
  - Minden Java objektum egy megfelelő class-hoz tartozik
  - `getClass()` metódus adja vissza (Object-ben definiált)
    - Vagy az osztály `class` adattagja (pl. `Foo.class`)
- Adott objektum osztályáról tartalmaz meta-adatokat
  - Név
  - Csomag
  - Metódusok
  - Mezők
  - Konstruktorok
  - Annotációk
- A reflection API részét képezi

```
public void process(Object obj) {
    if (obj.getClass().isAnnotationPresent(Processable.class)) {
        // process somehow;
    }
}
```

JAVA REFLECTION API

- A `java.lang.reflect` csomag tartalmazza
- Főbb inetrfészek
  - Member
  - Type
- Főbb osztályok
  - Method
  - Field
  - Constructor
- Részletes dokumentáció
  - https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html

JAVA REFLECTION API PÉLDA

```
import java.lang.reflect.*;

public class DumpMethods {
   public static void main(String args[])
   {
      try {
         Class c = Class.forName(args[0]);
         Method m[] = c.getDeclaredMethods();
         for (int i = 0; i < m.length; i++)
         System.out.println(m[i].toString());
      }
      catch (Throwable e) {
         System.err.println(e);
      }
   }
}
```

JAVA REFLECTION ELŐNYÖK/HÁTRÁNYOK

- Előnyei, tipikus felhasználási területei
  - Bővíthetőség
    - Külső komponensek név alapján történő betöltése
  - Class browser és vizuális fejlesztői környezetek
  - Debugger és tesztelő eszközök
- Amennyiben valami reflection nélkül is megoldható, az a preferált!!!
- Reflection hátrányai
  - Teljesítmény csökkenés
  - Biztonsági kockázatok
    - Runtime permission
  - Belső adatok felfedése
    - Pl. private adattagok

ANNOTÁCIÓK

- Meta-adatok megadásának egy formája
- Olyan adatokat szolgáltat egy programhoz, ami alapvetően nem képezi magának a programnak a részét
- Nincs közvetlen hatásuk az annotált kód működésére
- A legegyszerűbb formájában egy annotáció így néz ki
  - `@Entity`
  - A @ jelzi a fordító számára, hogy egy annotáció következik
  - Alkalmazható osztályokra, metódusokra, mezőkre
- Tartalmazhatnak egyéb elemeket
  - `@Author(name = "Jane Doe")`
- Egy deklarációt több annotációval is elláthatunk

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

ANNOTÁCIÓK FELHASZNÁLÁSA

- Annotációk tipikus felhasználása
  - Fordító számára információ megadása
    - Pl. hibák detektálásához, vagy figyelmeztetések kihagyásához
  - Fordítás idejű feldolgozás
    - Pl. kódgenerálás, XML fájl létrehozás, stb.
  - Futás idejű feldolgozás
    - Néhány annotáció futásidőben is elérhető
- Annotációk használhatók egy típus felhasználásakor is (nem csak a deklarációnál)
  - `new @Interned MyObject();`
  - `myString = (@NonNull String) str;`
  - `class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }`
  - `void monitorTemperature() throws @Critical TemperatureException { ... }`

ELŐRE DEFINIÁLT ANNOTÁCIÓK

- Java nyelv által használt annotációk
  - **@Deprecated**
  - **@Override**
  - **@SuppressWarnings**
  - **@SafeVarargs**
  - **@FunctionalInterface**
- Annotációkra alkalmazható annotációk
  - **@Retention**
  - **@Documented**
  - **@Target**
  - **@Inherited**
  - **@Repeatable**

SAJÁT ANNOTÁCIÓ KÉSZÍTÉSE

- Deklaráció

```java
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{
    int studentAge() default 18;
    String studentName();
    String stuAddress();
    String stuStream() default "CSE";
}
```

- Felhasználás

```java
@MyCustomAnnotation(
    studentName="Chaitanya",
    stuAddress="Agra, India"
)
public class MyClass {
...
}
```

RETENTION POLICY

- A @**Retention** annotációval lehet megadni
- Meghatározza, hogy az annotációk mikor érhetők el
  - RetentionPolicy.SOURCE
    - Csak forráskód szintjén érhetők el, nincs nyomuk a lefordított byte-kódban
  - RetentionPolicy.CLASS
    - Belekerül a byte-kódba, de a class betöltése után elveszik, azaz futási időben nincsen jelen az annotáció
    - Alapértelmezett érték
  - RetentionPolicy.RUNTIME
    - Futásidőben is elérhető annotáció (reflection-nel lekérdezhető)

SAJÁT ANNOTÁCIÓ FELDOLGOZÁSA

- Class objektum és reflection segítségével

```
1  Class aClass = MyClass.class;
2  Annotation[] annotations = aClass.getAnnotations();
3
4  for(Annotation annotation : annotations){
5      if(annotation instanceof MyCustomAnnotation){
6          MyCustomAnnotation myAnnotation = (MyCustomAnnotation) annotation;
7          System.out.println("name: " + myAnnotation.studentName());
8          System.out.println("address: " + myAnnotation.stuAddress());
9      }
10 }
```

JAVA GENERICS

- Generics = típusok használata paraméterként osztályok, interfészek vagy metódusok deklarálásakor
- Kód újra felhasználást segítik
- Hagyományos paraméterekkel ellentétben a típus paraméterek értékei típusok lesznek
  - A normál paraméterek értékei konkrét értékek
- Növelik a fordító által felfedhető hibák arányát
  - Erősebb ellenőrzés fordítás során
  - Cast műveletek kiiktatása
- Általános, típusfüggetlen algoritmusok írhatók

GENERIKUS OSZTÁLY, METÓDUS

- Nem generikus példa

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

- Generikus megvalósítás

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

GENERIKUS OSZTÁLY, METÓDUS HÍVÁSA

- Generikus osztály használata
  - vagy

```
Box<Integer> integerBox = new Box<Integer>();
```

```
Box<Integer> integerBox = new Box<>();
```

- Generikus metódusok

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

  - vagy akár

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

JAVA 7 ÉS 8 HASZNOS ÚJDONSÁGAI

- Java7
  - Try-with-resources
  - Try multiple catch
- Java 8
  - Lambda kifejezések
  - Collection API
    - Stream API
    - forEach
  - Default methods
  - Security
- Java 9
  - Modul rendszer (Jigsaw)
  - Linkelés
- Java 10 (11+) ☺
  - var // type inference

TRY-WITH-RESOURCES

- Egy speciális try utasítás, ami erőforrásokat deklarál
  - Az erőforrás egy olyan objektum, amit a használat végeztével le kell zárni
  - A konstrukció biztosítja minden ilyen erőforrás lezárását az utasítás végén
  - Bármi, ami implementálja a `java.lang.AutoCloseable` interfészt, használható erőforrásként
  - Az erőforrás deklaráció a `try` kulcsszó után zárójelek közt jelenik meg
  - A lezárás normális és kivételes viselkedés esetén is megtörténik
  - Több erőforrást is definiálhatunk

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

TRY MULTIPLE CATCH

- Egy `catch` blokk több különböző hiba típust tud egyszerre kezelni
  - Kód duplikáció kiküszöbölése
  - Túl általános hibák elkapásának megelőzése

```
catch (IOException ex) {
     logger.log(ex);
     throw ex;
catch (SQLException ex) {
     logger.log(ex);
     throw ex;
}
```



```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

LAMBDA KIFEJEZÉSEK

- Név nélküli függvények
- Az egy metódust tartalmazó anonymous belső osztályok kiváltására
  - Functional interface
  - Olyan interfész, aminek egyetlen absztrakt metódusa van (pl. Runnable, ActionListener, stb.)
- Funkcionalitás mint metódus argumentum, azaz kód, mint adat
- Felépítésük
  - Argumentum lista -> törzs
  - `(int x, int y) -> x + y`
- Funkcionális programozási képességek Java-ban

FUNCTIONAL INTERFACE

- Interfész egyetlen absztrakt metódussal
- `@FunctionalInterface` annotáció
- Előre definiált interfészek
  - Function
  - Supplier
  - Consumer
  - Predicate
  - Operator

```
1  Function<Integer, String> intToString = Object::toString;
2  Function<String, String> quote = s -> "'" + s + "'";
3
4  Function<Integer, String> quoteIntToString = quote.compose(intToString);
5
6  assertEquals("'5'", quoteIntToString.apply(5));
```

LAMBDA KIFEJEZÉS PÉLDÁK

- Interfész megadása

```
 6  public class RunnableTest {
 7    public static void main(String[] args) {
 8
 9      System.out.println("=== RunnableTest ===");
10
11      // Anonymous Runnable
12      Runnable r1 = new Runnable(){
13
14        @Override
15        public void run(){
16          System.out.println("Hello world one!");
17        }
18      };
19
20      // Lambda Runnable
21      Runnable r2 = () -> System.out.println("Hello world two!");
22
23      // Run em!
24      r1.run();
25      r2.run();
26
27    }
28  }
```

- Rendezéshez

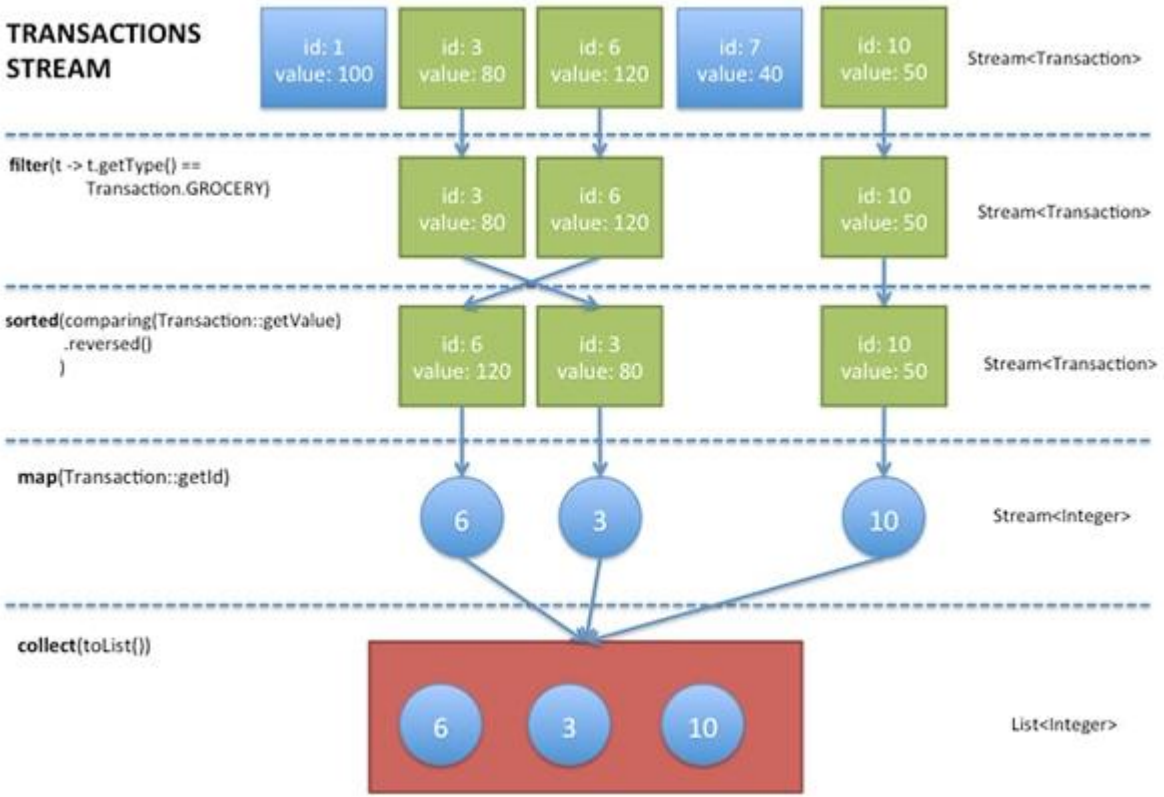
```
// Print Desc
System.out.println("=== Sorted Desc SurName ===");
Collections.sort(personList, (p1,  p2) -> p2.getSurName().compareTo(p1.getSurName()));
```

STREAM API

- Kollekciók (soros) folyammá (stream) alakítása
- Becsomagolják a kollekciókat, és plusz műveleteket adnak hozzájuk
  - A mit-re lehet fókuszálni a hogyan helyett
- Egy sor klasszikus feladat kompakt módon, deklaratívan megadható
  - Szűrés
  - Csoportosítás
  - Map-reduce
- Erősen épít a lambda kifejezésekre
- A legtöbb stream művelet stream-et ad vissza, így a műveletek láncolhatók
- Tipikus használat
  1. Stream létrehozása
  2. Néhány köztes művelet végrehajtása (újabb stream-ek jönnek létre)
  3. Befejező művelet végrehajtása (nem hoz létre újabb stream-et és/vagy mellékhatása van)

STREAM PÉLDA

```
List<Integer> transactionsIds =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```



FOREACH

- Kollekciók kompaktabb bejárásához
- `Consumer` funkcionális interfész paraméter
  - Azaz lambda kifejezéssel megadható

```
1  List<String> names = new ArrayList<>();
2  names.add("Larry");
3  names.add("Steve");
4  names.add("James");
5  names.add("Conan");
6  names.add("Ellen");
7
8  names.forEach(new Consumer<String>() {
9      public void accept(String name) {
10         System.out.println(name);
11     }
12 });
```

  - vagy

```
1  names.forEach(name -> System.out.println(name));
```

  - vagy

```
1  names.forEach(System.out::println);
```

COLLECTION API

- Map közvetlen rendezése
  - Stream-ek segítségével
- Iteráció `forEach` segítségével
- `getOrDefult`
  - Nincs szükség if-else null ellenőrzésre
- `putIfAbsent`
  - Map kulcsok felülírása kivédhető
- Értékek közvetlen feldolgozása
  - Nincs szükség az érték lekérésére, módosítására, visszarakására
- Merge map-hez

DEFAULT METHODS

- Interface metódusok tartalmazhatnak alapértelmezett törzset
- Lehetőséget ad új metódusok deklarálására meglévő interfészekben anélkül, hogy elrontaná a meglévő implementációt

```
@FunctionalInterface
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
                action.accept(t);
        }
    }
}
```

SECURITY

- TLS 1.1 és TLS 1.2 alapértelmezetten engedélyezett
- Korlátozott `doPrivileged`
- Erősebb jelszó titkosító algoritmusok
- KeyStore fejlesztések
- SHA-224
- Nagy entrópiájú véletlen szám generátor
- Megbízhatóbb tanúsítvány ellenőrzés
- Stb.

JAVA MODULOK

- Újabb absztrakciós szint
  - Csomag szintű függőségek
  - A jar fájloktól finomabb granularitás
- Moduláris jar fájlok
- Requires/export
- Core osztályok

```
module kocsi {
    exports hu.lamborghini.kocsi;

    requires motor;
}
```

JAVA 9 LINKELÉS

- Jlink
- Kompakt profilok
- Megtakarított tárhely

JAVA 10 LOKÁLIS VÁLTOZÓ TÍPUS KIKÖVETKEZTETÉS

- A var kulcsszó bevezetése típus automatikus kikövetkeztetéséhez
  - Fejlesztői élmény növelése
  - „Syntactic sugar”
- Változó deklarálása a típus explicit megnevezése nélkül

```
List<String> list = new ArrayList<String>();
Stream<String> stream = getStream();
```



```
var list = new ArrayList<String>();
var stream = getStream();
```

JAVA 11-21

- 11: run source files directly (java MyScript.java), var for lambda parameters
- 12: Unicode 11 support
- 13: switch expression (preview), multiline strings (""")
- 14: switch expression, records (preview), enhanced NPE message, pattern matching for instanceOf, new GC
- 15: sealed classes with permit (preview)
- 16: records and pattern matching
- 17: pattern matching for switch, sealed class
- 18: simple web server (jwebserver)
- 19: lot of preview features (virtual threads, memory API, vector API, structured concurrency, etc.)
- 20: iterations of preview features
- 21: iterations of preview features

KÖSZÖNÖM A FIGYELMET!

„A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.”

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE