Prednáška 5: Výnimky, RTTI, zoskupenia objektov a generickosť v Jave

Objektovo-orientované programovanie 2012/13

Valentino Vranić

Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

20. marec 2013

Obsah prednášky

1. Výnimky
2. RTTI
3. Zoskupenia a generickosť

Výnimky

Výnimky (1)

- Programy by mali byť čím robustnejšie
- Ale pri tvorbe programu nie je možné predpovedať a ošetriť všetky *výnimočné situácie*
- Jestvujú rôzne prístupy k vysporiadaniu sa s výnimočnými situáciami
- V každom prípade ošetrenie výnimky (exception) môže znamenať:
  - prerušnie programu
  - pokus o zotavenie
- Ošetrovanie výnimočných situácií na mieste ich predpokladaného vzniku:
  - robí základný kód neprehľadným
  - často je možné len vo vyššom kontexte – napr. delenie nulou je zlé, ale prečo k nemu došlo?

Výnimky (2)

- Podpora na úrovní programovacieho jazyka:
  - vyhadzovanie výnimiek (exception throwing)
  - zachytávanie výnimiek (exception catching)
  - spracovanie výnimiek (exception handling)
- Oddelenie miesta vzniku a ošetrenia výnimočnej situácie
- Aj v iných jazykoch (C++, Delphi. . .)

Mechanizmus výnimiek v Jave

- Výnimka je objekt
  - Hierarchia tried pre výnimky začína triedou `Exception` (podtriedou triedy `Throwable`)
- Metódy, v ktorých môže dôjsť k výnimkám, to deklarujú klauzulou **throws**
- Výnimka môže vzniknúť pri chybách vo vykonávaní, ale dá sa vyhodiť aj priamo použitím príkazu **throw**
- Kód, ktorý obsahuje volania metód, v ktorých k výnimkám môže dôjsť, sa uzátvorí do bloku **try**
- Výnimky sa zachytávajú v blokoch **catch**, ktoré nasledujú po **try** bloku
- Kód, ktorý sa vykoná, nakoniec (za každých okolností) sa uvedie v bloku **finally**

Kontrola výnimiek

- Pri kontrolovaných výnimkách (checked exceptions) prekladač kontroluje, či:
  - metóda nevyhadzuje výnimky, ktoré nedeklarovala
  - v metóde jestvuje ošetrenie výnimiek (exception handler), ktoré môžu vzniknúť volaním metód, ktoré ich deklarovali
- Z kontroly sú vynechané výnimky typu RunTimeException (unchecked exceptions)
  - `NullPointerException`, `ArrayIndexOutOfBoundsException`...

Príklad: delenie

```
public class Delenie {
    public static void main(String[] args) {
      System.out.println(
         Integer.parseInt(args[0]) /
         Integer.parseInt(args[1]));
    }
}
```

- Čo hrozí?
  - Nezadanie parametrov
  - Zadanie neceločíselných hodnôt ako parametrov
  - Delenie nulou

Príklad: delenie s ošetrením výnimiek

```
public class Delenie {
    public static void main(String[] args) {
        try {
        System.out.println(
            Integer.parseInt(args[0]) /
            Integer.parseInt(args[1]));
        } catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("Nedostatocny pocet parametrov.");
        } catch (NumberFormatException e) {
          System.out.println("Nespravny format parametrov.");
        } catch (ArithmeticException e) {
          System.out.println("Chyba pri deleni.");
        }
    }
}
```

Syntax bloku try-catch-finally

- Základná syntax:

```
try {
  // blok, ktorý môže vyhadzovať výnimky tried E1, E2 atď.
  // alebo ich podtried
} catch (E1 e1) { // bloky catch sa vykonajú v zadanom poradí!
  // zachytáva triedu výnimiek E1 a jej podtriedy
} catch (E2 e2) {
  // zachytáva triedu výnimiek E2 a jej podtriedy
} finally {
  // kód, ktorý sa musí nakoniec vykonať vždy
}
```

- Java 7 umožňuje zlúčiť rovnaké bloky catch pre rozdielne typy výnimiek[1]

```
catch (X | Y | Z e) {
  ...
}
```

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html

Trieda Exception

- `Throwable`
  - `Exception`
  - `Error`
- Trieda `Exception` (dedí od triedy `Throwable`) poskytuje rôzne užtočné metódy (príklad v TiJ):
  - String getMessage() – detailná správa
  - String toString() – krátky opis
  - void printStackTrace() – výpis zásobníka vykonávania
  - Throwable fillInStackTrace() – naplnenie zásobníka vykonávania informáciami o súčasnom stave (užitočné pri opätovnom vyhadzovaní výnimky)

    ...

Vlastné výnimky

- Špecifické pre danú aplikáciu
- Dajú sa odvodiť od triedy `Exception` alebo jej podtriedy
- Ak sa odvodia od `RuntimeException`, nebudú kontrolované
- Z názvu výnimky má byť zrejmý jej význam
- Samotná trieda najčastejšie nepridáva nič
- Konvencia (tu porušená): názov končí na `Exception`

```
class NieJeTrojuholnik extends Exception {
}

class Trojuholnik {
    ...
    Trojuholnik(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
        if (...) // overenie kolinearnosti
            throw new NieJeTrojuholnik();
    }
    ...
}
```

Ošetrenie výnimky

- Ošetrenie výnimky – vo vyššom kontexte:
  - Konštruktor triedy `Trojuholnik` nevie, prečo dostal nekorektné parametre
  - Ale metóda, ktorá ho volala, by to mohla vedieť

```
class C {
   void m(Bod a, Bod b, Bod c) {
      try {
        Trojuholnik t = new Trojuholnik(a, b, c);
        ...
      } catch (NieJeTrojuholnik e) {
        ...
      }
   }
}
```

Ošetrenie výnimky (2)

- Ak metóda nevie ošetriť výnimku, môže ju preniesť do ďalšieho vyššieho kontextu jej opätovným vyhodením

```
class C {
    void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
        try {
          Trojuholnik t = new Trojuholnik(a, b, c);
          ...
        } catch (NieJeTrojuholnik e) { // výnimku sme zachytili
          throw(e); // ale nevieme čo s ňou
        }
    }
}
```

Ošetrenie výnimky (3)

- Ak metóda výnimku neošetruje, musí deklarovať, že ju vyhadzuje

```
class C {
   void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
      Trojuholnik t = new Trojuholnik(a, b, c);
      ...
   }
}
```

Ošetrenie výnimky (4)

- Prenášaním výnimka môže skončiť v metóde `main()`

```
class C {
    void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
        Trojuholnik t = new Trojuholnik(a, b, c);
        ...
    }
    public static void main(String[] args) {
        Bod[] b = { new Bod(0.0, 0.0), new Bod(1.0, 1.0), new Bod(2.0, 2.0) };
        try {
          new C().m(b[0], b[1], b[2]);
        } catch (NieJeTrojuholnik e) {
          ... // tu by sme už mohli vedieť čo robiť
        }
    }
}
```

Ošetrenie výnimky (5)

- Deklarovaním že `main()` vyhadzuje výnimku prenášame túto výnimku na výstup

```
class C {
    void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
        Trojuholnik t = new Trojuholnik(a, b, c);
        ...
    }
    public static void main(String[] args) throws NieJeTrojuholnik {
        Bod[] b = { new Bod(0.0, 0.0), new Bod(1.0, 1.0), new Bod(2.0, 2.0) };
        new M().m(b[0], b[1], b[2]);
    }
}
```

Výnimky pri prekonaných metódach (1)

- Rozsah výnimiek sa pri prekonávaní metód môže len zachovať alebo zúžiť:
  - prekonávajúca metóda nemusí deklarovať žiadne výnimky
  - prekonávajúca metóda môže deklarovať rovnaké triedy výnimiek ako prekonávaná metóda
  - prekonávajúca metóda môže deklarovať podtriedy výnimiek tried výnimiek prekonávanej metódy

Výnimky pri prekonaných metódach (2)

- Príklad:

```
class A {
    void m() {...}
}
class B extends A {
    void m() throws E {...} // prekladač by toto nepovolil
}
```

- Prečo je to problém?

```
A a;
. . . // do a sa môže dostať objekt typu A alebo B
a.m(); // Vyhadzuje toto volanie výnimku alebo nie?
       // Nedá sa kontrolovať v čase prekladu!
```

- Z hľadiska rozlíšenia preťažených metód výnimky sa neuvažujú
- V Jave 7 je možné uviesť aj viac typov výnimiek pri klauzule throws[2]
- Od verzie 7 je aj niekoľko ďalších rozšírení[3]

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html
[3] http://docs.oracle.com/javase/7/docs/technotes/guides/language/enhancements.html

RTTI

Run-Time Type Identification (RTTI)

- Mechanizmus identifikácie typu objektu v čase vykonávania programu
- Každý objekt so sebou nesie úplné informácie o svojom type
- Každá načítaná trieda je reprezentovaná objektom triedy `Class`
- Referencia objektu tejto triedy je dostupná prostredníctvom literálu **class**, napr.:

```
Trojuholnik.class
```

Operátor instanceof

- Umožňuje rozhodovanie na základe typu objektu

```
if (o instanceof Trojuholnik) {
    ...
}
```

- To isté pomocou metódy isInstance()

```
if (Trojuholnik.class.isInstance(o)) {
    ...
}
```

Metódy triedy Class

- Everything you ever wanted to know about a class...
  - `boolean isInstance(Object obj)`
  - `String getName()`
  - **static** `Class forName(String className)`
  - `Constructor[] getConstructors()`
  - `Field[] getFields()`
  - `Class getSuperclass()`
  - `Class[] getInterfaces()`
  - `Object newInstance()`

Príklad: zistenie počtu útvarov daného typu

```
class GUtvary {
    static int pocet(Utvar[] u, Class Typ) {
        int n = 0;
        for (int i = 0; i < u.length; i++)
            if (Typ.isInstance(u[i]))
                n++;
        return n;
    }
    public static void main(String args[]) {
        Utvar[] o = { new Trojuholnik(), new Kruh(), new Kruh() };
        System.out.println(pocet(o, Kruh.class));
    }
}
```

Reflexia

- Význam termínu: uvažovanie o sebe samom
- Pre jazyk reflexia znamená uvažovanie o jeho štruktúre prostredníctvom samotného jazyka
- Reflektívne API Javy: triedy `Class`, `Field`, `Method` a `Constructor`

Zoskupenia a generickosť

Zoskupenia objektov v Jave

- Podporené na úrovni jazyka – polia (arrays)
- Podporené na úrovni API – *zoskupenia* (collections)
  - Niekedy označované ako kontejnery (containers)

Polia

- Základné veci už boli povedané
  - sumarizácia v TiJ, časť *Arrays* po *The Array class*
- Každé pole je vlastne objekt
- Rýchly prístup k prvkom, ale veľkosť poľa sa nedá meniť
- Schopnosť uchovávať hodnoty primitívnych typov
- Pole má *typ*

```
class A {}
class B extends A {}
```

- Kontrola typov pri kompilácii:

```
A[] x;
x = new A[] {new B()};
x = new B[] {new B()}; // pole referencií podtypu
// x = new B[] {new A()}; // chyba pri kompilácii
```

- Viacrozmerné polia

Podpora práce s poliami

- Trieda `Array` – statické metódy pre dynamické vytváranie a prístup k poliam
- Trieda `Arrays` – rady statických preťažených metód pre prácu s poliami:
  - `equals()` – porovnávanie
  - `fill()` – naplnenie
  - `sort()` – triedenie
  - `binarySearch()` – binárne vyhľadávanie (v utriedenom poli)
- Trieda `System`
  - `arraycopy()`

Reťazce znakov

- V Jave implementované ako triedy:
  - Trieda `String` – konštantné reťazce znakov
  - Trieda `StringBuffer` – premenlivé reťazce znakov
- Rôzne metódy na podporu práce s reťazcami znakov: zakladanie, kopírovanie, spájanie, vyhľadávanie podreťazcov, nahrádzanie podreťazcov apod. (podrobnosti v dokumentácii k Java API)

Zoskupenia a generickosť

- Štruktúry údajov na uchovávanie prvkov
- Java API obsahuje rozsiahlu podporu zoskupení prostredníctvom Collections Framework[4]
- Typy zoskupení:
  - zoskupenie (collection) – v užšom zmysle:
    - zoznam (list)
    - množina (set)
  - tabuľka (map)
- Od Javy 5 všetky zoskupenia uchovávajú prvky ľubovoľného typu
- Predtým to bol len všeobecný týp Object – bolo potrebné pretypovávať

[4] http://docs.oracle.com/javase/1.4.2/docs/guide/collections/

Generickosť zoskupení (1)

- Generickosť: nová črta pridaná v Jave 5
- Zoskupenia v rámci Collections Java API sú generické
- Pri vytváraní inštancie zoskupenia treba uviesť ako parameter typ objektov, ktoré sa v ňom budú uchovávať
- Napríklad zoznam objektov typu String:

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

- Takýto typ (`List<String>`) sa označuje ako parametrizovaný
- Typ referencie `list` je `List`
  - Rozhranie `List` implementujú všetky zoskupenia tvaru zoznamu
  - Lepšie než použiť priamo referenciu typu `ArrayList`
  - Neskôr možno ľahko zmeniť presný typ použitého zoskupenia – napr. na `LinkedList`

Generickosť zoskupení (2)

- Java 7 povoľuje zjednodušený zápis, ak je parametrický typ zrejmý z kontextu:[5]

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

[5] http://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html

Generickosť zoskupení (3)

- Práca so zoznamom je jednoduchá:

```
list.add("jeden");
list.add("dva");
String s = list.get(0); // s = "jeden"
```

- Bezpečné používanie typov stráži prekladač:

```
list.add(new Integer(1)); // chyba pri preklade
```

- *Pozor!* Môže sa zdať, že by mal platiť polymorfizmus, ale nie je to tak:

```
List<Object> listObject = list; // chyba pri preklade
```

- Parametrizovaný nie je typ zoznamu, ale typ jeho prvkov

Náhradný znak pre typ

- Čo keby sme potrebovali poskytnúť metódu, ktorá akceptuje hocijaké zoskupenie
- Použije sa náhradný znak (wildcard) ?

```
import java.util.*;

public class C {
    public void m(List<?> list) {
        for(int i = 0; i < list.size(); i++) {
            Object o = list.get(i));
            ...
        }
    }
}
```

- Náhradny znak ? pri tvorbe inštancie označuje neznámy typ:

```
List<?> l = new ArrayList<String>();
l.add(new String("s")); // chyba pri preklade
l.add(null); // jedine null sa dá vložiť
```

- Takýto zoznam sa však dá čítať

Ohraničený náhradný znak (1)

- Parameter generického typu je možné obmedziť vzhľadom na dedenie
- Povedzme, že chceme poskytnúť metódu, ktorá ako parameter má zoznam hocijakých grafických útvarov odvodených od triedy `Utvar`
- Použijeme ohraničený náhradný znak (bounded wildcard)

```
public class C {
    public static void nakresliUtvary(List<? extends Utvar> list) {
        ...
    }
}
```

Ohraničený náhradný znak (2)

- Takejto metóde potom možno poskytnúť všeobecný zoznam útvarov, ale aj špeciálne zoznam kruhov, trojuholníkov apod.:

```
List<Utvar> u = new ArrayList<Utvar>();
List<Circle> k = new ArrayList<Circle>();
List<Trojuholnik> t = new ArrayList<Trojuholnik>();
...
C.nakresliUtvary(u);
C.nakresliUtvary(k);
C.nakresliUtvary(t);
```

Ohraničený náhradný znak (3)

- Ak triedy útvarov implementujú rozhranie `Kresleny`, možno urobiť aj toto:

```
public class C {
    public static void nakresliKreslene(List<? extends Kresleny> list) {
        ...
    }
}
```

- Použitie:

```
List<Kresleny> ko = new ArrayList<Kresleny>();
...
C.nakresliKresleny(ko);
```

- Výraz ? **extends** `T` zhora ohraničuje náhradný znak
  - Špecifikuje všetky typy, ktoré dedia od typu `T`
  - Typ `T` môže byť rozhranie alebo trieda
  - Vzťah dedenia môže byť aj **extends**, aj **implements**

Kontrola typov v generickosti

- Treba pamätať, že kontrola typov sa vykonáva v čase prekladu

```
public class C {
    public void m(List<? extends Kresleny> list) {
        Bod a, b, c;
        ...
        list.add(new Trojuholnik(a, b, c)); // chyba pri preklade
    }
}
```

- Keby táto situácia nebola ošetrená pri preklade, program by mohol padnúť: nevieme, či typ parametra nebude napr. `List<Kruh>`

Čisté typy

- Pri korektnom generickom kóde sa vždy uvádza typ
- Typ sa však nemusí uviesť a vtedy ide o tzv. čistý (raw) typ
- Do takého typu možno vložiť hocičo

```
List list = new ArrayList();
list.add(new String("s"));
list.add(new Integer(1));
list.add(new Object());
```

- Pri čistých typoch je kontrola typov potlačená
- Prekladač upozorní, že taký kód používa nepreverené alebo nebezpečné operácie
- Čisté typy slúžia na zabezpečenie kompatibility s predchádzajúcimi verziami Javy

Zdola ohraničený náhradný znak

- Dá sa špecifikovať aj zhora ohraničený náhradný znak
- Predpokladajme, že chceme kresliť trojuholníky a typy, od ktorých je odvodený

```
public class C {
    public static void nakresliTroj(List<? super Trojuholnik> list) {
        ...
    }
}
```

Iterátory

- Spôsob ako prechádzať zoskupením (v užšom zmysle) bez toho, aby bolo potrebné poznať jeho typ
- Každé zoskupenie dokáže poskytnúť iterátor ako objekt zavolaním metódy `iterator()`

Príklad: iterácia cez ArrayList (1)

```
public class Element {
    private int n;
    public Element(int i) {
        n = i;
    }
    public void print() {
        System.out.println("Element " + n);
    }
}
```

Príklad: iterácia cez ArrayList (2)

```
import java.util.*;

public class C {
    public static void main(String[] args) {
        List<Element> z = new ArrayList<Element>();

        for(int i = 0; i < 5; i++)
            z.add(new Element(i + 1));

        Iterator<Element> i = z.iterator();

        while(i.hasNext())
            i.next().print();
    }
}
```

Rozšírena slučka for (1)

- Nemusíme identifikovať iterátor
- Stačí použiť rozšírenú slučku **for**

```
public class C {
    public static void main(String[] args) {
        List<Element> z = new ArrayList<Element>();

        for(int i = 0; i < 5; i++)
            z.add(new Element(i + 1));

        for(Element e : z)
            e.print();
    }
}
```

- „Pre každý element e z kolekcie z“

Rozšírena slučka for (2)

- Tá istá iterácia v tvare obyčajnej slučky for:

```
for(Iterator<Element> i = z.iterator(); i.hasNext(); )
    i.next().print();
```

- Dá sa aplikovať na hocijaké zoskupenie, ktoré poskytuje iterátor (objekty typu `Iterable`)

Rozšírena slučka for a polia

- Rozšírena slučka for sa dá použiť aj na polia

```
public void nakresliKreslene(Kreslene... k) {
    for(Kreslene e : k)
        e.nakresli();
    }
}
```

- Tiež sa dá použiť aj na vymenované typy (enum) zavedené v Jave 5
  - Vymenované typy v Jave fungujú podobne ako v jazyku C
  - Sú však oveľa dômyselnejšie
  - Pozrite vysvetlenie na príkladoch na stránkach Oracle[6]

[6] http://docs.oracle.com/javase/1.5.0/docs/guide/language/enums.html

Generické metódy (1)

- Generické zoskupenia predstavujú príklady generických tried
- Metódy v generických triedach môžu byť definované v zmysle parametrov typov
  - Príkladom je metóda `add(E o)`
  - `E` je parameter typu definovaný v rozhraní `List`
- Metódy môžu tiež definovať parametre typov
- Také metódy sa označujú ako generické

Generické metódy (2)

- Predpokladajme, že potrebujeme implementovať metódu na kopírovanie z poľa do zoskupenia[7]
- Toto sa nedá urobiť:

```
static void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a)
        c.add(o); // chyba pri preklade
}
```

- Riešenie je generická metóda

```
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a)
        c.add(o);
}
```

[7] kód z G. Bracha. Generics in the Java Programming Language. July 2005. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Príklad: generický zreťazený zoznam (1)

- Povedzme, že potrebujeme implementovať jednosmerne zreťazený zoznam
- Výhodné je urobiť ho genericky, pre údaje hocijakého typu

```
public class SLLElement<T> {
    private T data;
    private SLLElement<T> next;
    public SLLElement() { }
    public SLLElement(T d) { data = d; }
    public T getData() { return data; }
    public SLLElement<T> getNext() { return next; }
    public void setData(T d) { data = d; }
    public void setNext(SLLElement<T> e) { next = e; }
    public String toString() { return data.toString(); }
}
```

Príklad: generický zreťazený zoznam (2)

```
public class SLList<T> {
    private SLLElement<T> head;
    private SLLElement<T> tail;
    public void tailInsert(T e) {
        SLLElement<T> sll_e = new SLLElement<T>(e);
        if (head == null) {
            head = sll_e;
            tail = head;
        }
        else {
            tail.setNext(sll_e);
            tail = sll_e;
        }
        tail.setNext(null);
    }
...
}
```

- Použitie:

```
SLList<Integer> l = new SLList<Integer>();

for (int i = 1; i < 10; i++)
  l.tailInsert(i);
```

Automatické balenie hodnôt primitívnych typov

- Od verzie 5.0 Java zabezpečuje automatické balenie hodnôt primitívnych typov (autoboxing)[8]
- V našom príklade:

  public void `tailInsert(T e) { . . . }`

- Vytvoríme inštanciu zoznamu pre `Integer` a použijeme automatické balenie:

```
SLList<Integer> l = new SLList<Integer>();
l.tailInsert(5);
```

- Funguje aj automatické rozbaľovanie

```
Integer o = new Integer(7);
int i = o;
```

[8] http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html

Trieda `Class` a generickosť

- Trieda `Class` je tiež generická
- Umožňuje lepšiu bezpečnosť typov
- Príklad s počítaním útvarov – na mieste argumentu `Typ` možno zadať hocijaký typ

```
public static int pocet(Utvar[] u, Class Typ) { . . . }
```

- Použitie:

```
Utvar[] o = {new Trojuholnik(), new Kruh(), new Kruh()};
System.out.println(pocet(o, Integer.class)); // OK (!)
```

- Chceme však počítať útvary, nie hocijaké objekty

```
public static int pocet(Utvar[] u, Class<? extends Utvar> Typ) {
   Utvar[] o = {new Trojuholnik(), new Kruh(), new Kruh()};
}
```

- Použitie:

```
// System.out.println(pocet(o, Integer.class)); // chyba pri preklade
System.out.println(pocet(o, Kruh.class)); // OK
```

Sumarizácia

Sumarizácia

- Mechanizmus výnimiek v Jave – kontrola a ošetrenie
- Vlastné výnimky pre robustnejší kód
- RTTI a reflexia v Jave
- Polia sú efektívne pri uchovávaní prvkov
- Väčšiu flexibilitu ponúkajú zoskupenia
- Zoskupenia sú v Jave generické: jednoducho sa dá stanoviť typ prvkov
- Generické zoskupenia predstavujú príklady generických tried
- Generické metódy definujú parametre typov
- Automatické balenie hodnôt primitívnych typov
- Generickosť triedy `Class`

Čítanie

- Dnešná prednáška:
  - OJA, kapitoly 8–10
  - Doplnková literatúra – podľa potrieb projektu:
    - kapitoly 9–11 v TiJ
    - generickosť: `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf` (môžete vynechať časti 6.2, 6.3, 7 a 10)
    - rozšírená slučka for: `http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.html`
    - automatické balenie hodnôt primitívnych typov: `http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html`
- Na ďalšiu prednášku: OJA, kapitoly 11 a 12