Lambdas & Streams - úvod

This is … lambda!!! :-)

- Od Java SE 8 je možné toto:

```
button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        layout.addComponent(new Label("Thank you for clicking"));
    }
});
```

- Zapsat takto:

```
button.addClickListener(e->{
    layout.addComponent(new Label("Thank you for clicking"));
});
```

- Poznámka: Příklad je z Vaadin frameworku.
- Poznámka: Lambda je anonymní funkce.

Rozbor Lambda funkce I.

- Funkce se skládá z:
  1) Název
  2) Vstupní argumenty
  3) Tělo
  4) Návratový typ
- Pouze 2) a 3) je důležité, funkce může být anonymní a návratový typ lze získat z výsledku funkce. Lambda tedy obsahuje pouze vstupní argumenty a samotné tělo funkce.
- Vzhledem k tomu, že Lambda je implementačně kompatibilní s vnitřní anonymní třídou, lze lehce integrovat se stávajícím kódem.
  - http://programmers.stackexchange.com/questions/195081/is-a-lambda-expression-something-more-than-an-anonymous-inner-class-with-a-singl

Rozbor Lambda funkce II.

- Argumenty:
  - Ani jeden argument:

    ```
    ()->
    ```

  - Právě jeden argument:

    ```
    (e)->    nebo:    e->
    ```

  - Více argumentů:

    ```
    (e1, e2)->
    ```

Rozbor Lambda funkce III.

- Když se tělo Lambda funkce skládá z jediného výrazu:

```
int[] pole = new int[] { 1, 2, 3, 4, 5 };

Stream.of(pole).forEach(i -> System.out.println(i));

Nebo:

Stream.of(pole).forEach(i -> {System.out.println(i);});
```

POZOR!!! **http://blog.agiledeveloper.com/2015/06/lambdas-are-glue-code.html**

Venkat Subramaniam
@venkat_s

Follow

Lambda expressions should be glue code.
Two lines may be too many.

Další příklady

```
Runnable runnable = () -> {
   while(true) {
      System.out.println("this is run() method " + new Date());
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
};

new Thread(runnable).start();
```

Mějme tuto třídu:

```
public class Person {

   private String name;

   public String getName() {

      return name;

   }

   public void setName(String name) {

      this.name = name;

   }

   public Person() { }

   public Person(String name) {

      this.name = name;

   }

}
```

List forEach()

- Lambda výraz je možné použít pro jednoduché procházení listu:

```
ArrayList<Person> persons = new ArrayList<>();

persons.add(new Person("Jirka"));

persons.add(new Person("Michal"));

persons.add(new Person("Ivan"));


persons.forEach(n -> System.out.println(n.getName()));
```

Method references I.

- Pokud lambda výraz nedělá nic jiného, než že volá stávající metodu, pak můžete použít method references:
  - http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

```
strings.forEach(System.out::println);
```

Když ještě přidáte k třídě Person metodu toString(), pak to vypíše smysluplné texty.

Method references II.

- Lepší použití Method references:
- Dříve bylo nutné pro řazení listu provést například toto:

```
public class PersonComparator implements Comparator<Person> {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getName().compareTo(o2.getName());
   }
}
```

- Utřídění listu:

```
Collections.sort(persons, new PersonComparator());
```

Method references III.

- Nyní je možné toto:
- Přidejte do třídy Person tuto metodu:

```
public static int compareByName(Person o1, Person o2) {
   return o1.getName().compareTo(o2.getName());
}
```

- Utřídění listu:

```
Collections.sort(persons, Person::compareByName);
```

Poznámka:
Od Java 8 obsahuje
java.util.List metodu sort(),
která udělá to samé

Lambdas … again

- Nebo toto:

```
Collections.sort(persons, (p1, p2) -> {
    return p1.getName().compareTo(p2.getName());
});
```

Method references IV.

- Pozor! U method references záleží na pořadí parametrů.

```
Integer[] pole = new Integer[] { 1, 2, 3, 4, 5 };

int sum1 = Stream.of(pole).reduce(0, (total, e) -> Integer.sum(total, e));

int sum2 = Stream.of(pole).reduce(0, Integer::sum);

System.out.println(sum1);

System.out.println(sum2);
```

Toto je ekvivalentní

Průšvih by nastal kdyby záleželo na pořadí vstupních parametrů funkce
a získavaly by se v opačném pořadí
než v jakém by je funkce vyžadovala!!!
(v takovém případě se použije lambda)

Method references V.

- Problém!
- Snažím se překonvertovat stream objektů typu Integer na stream objektů typu String:

```
Integer[] pole = new Integer[] { 1, 2, 3, 4, 5 };

Stream<String> stream = Stream.of(pole).map(Integer::toString);
```

- Chyba:
  - Ambiguous method reference: both toString() and toString(int) from the type Integer are eligible
- Řešení: Lambda, nebo vlastní funkce.

```
Stream<String> stream = Stream.of(pole).map(e -> Integer.toString(e));
```

Lambda & scope

- Lambda výraz nedefinuje nový scope, pokračuje se v předcházejícím. Co to znamená? Toto není možné:

```
String tmp = "test";

Collections.sort(persons, (p1, p2) -> {

   String tmp = "test 2";

   return p1.getName().compareTo(p2.getName());

});
```

- Zase na druhou stranu … když jste u vnitřní anonymní třídy chtěli použít proměnnou z nadřazené metody, pak jste ji museli označit klíčovým slovem `final`. To s lambda výrazy není nutné.
  - To ale neznamená, že je možné tuto proměnnou změnit. Pokud se o to pokusíte, pak dostanete tuto chybu: `Local variable tmp defined in an enclosing scope must be final or effectively final`

Effectively final znamená, že proměnná nemusí být označena klíčovým slovem final, ale že se k ní tak musíte chovat.

Lambda & nové metody

- Interface jako List, Map apod. od Java 8 mají užitečné default metody, které jsou navržené pro lehkou spolupráci s lambda výrazy a díky kterým již není nutné vytvářet util třídy a metody jako Collections.sort().
- java.util.List: sort(), forEach(), removeIf(), replaceAll()
- java.util.Map:forEach(), replaceAll(), compute(), computeIfAbsent(), computeIfPresent(), merge()
- java.util.Collection: removeIf()

```
persons.removeIf(p -> {

   if(p.getName().equals("Jirka")) { return true; }

   return false;

});
```

Streams

Důvod pro streamy

- Stream je náhrada kolekcí … a také důvod pro default metody (aby bylo možné do stávajících kolekcí přidat tuto novou funkcionalitu).
- Streamy vznikly pro zjednodušení tvorby paralelního zpracování metod. Toho standardní kolekce nejsou schopny.
- Externí iterace:

```
for (Person person : persons) {
   person.setName(person.getName().trim());
}
```

for-each cyklus vnitřně zavolá persons.iterator() a sekvenčně projde tento list.

- Interní iterace:

```
persons.forEach(person -> {
   person.setName(person.getName().trim());
});
```

foreach průchod listem persons neprovádíme my, ale knihovna (Java)

Příklad

- Vypíše všechny lidi, kteří mají jméno „Jirka“:

```
persons.stream().filter(person -> {
    return person.getName().equals("Jirka");
}).forEach(System.out::println);
```

- Metody streamů jsou rozdělené do dvou skupin:
  - Intermediate: například filter. Nepovinné a volají se před terminal metodou.
  - Terminal: například forEach. Volání streamu vždy končí terminal metodou.
- Jakmile se zavolá terminal metoda, je stream ukončen:

```
Stream<Person> stream = persons.stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println);
```

```
Exception in thread "main"
java.lang.IllegalStateException:
stream has already been operated
upon or closed
```

parallelStream

- Ve výchozím nastavení pracují streamy s daty sériově. Můžete ale zvolit paralelní vykonávání:

```
long count = persons.parallelStream().filter(person -> {
   return person.getName().startsWith("M");
}).count();
System.out.println(count);
```

- Pozor na použití parallel streamů ve webových aplikacích:
  - http://zeroturnaround.com/rebellabs/java-parallel-streams-are-bad-for-your-health/

parallelStream II.

- Parallel stream používá fork-join-pool. Ve výchozím nastavení je počet vláken v něm: počet procesorů – 1
- Jak to změnit?
  - http://stackoverflow.com/questions/21163108/custom-thread-pool-in-java-8-parallel-stream