

Návrh softwarových systémů - Java 8

Jiří Šebek *Návrh softwarových systémů*
(B6B36NSS)



```
public final void onSensorChanged(SensorEvent event)
{
    m_flightIntensity = event.values[0];
    m_etAmblight.setText("" + m_flightIntensity + " lx");
}

... resume()
... light, ... (NORMAL);
```

Co je nového ?

- Default interface methods
- **Lambda expressions**
- Method references and repeatable annotations
- Streams
- Functional interfaces
- Map extensions
- The new Date API
- Více..



Předchozí verze

- Java 5 (září 2004)
 - Generika
 - Anotace
 - Enum
- Java 6 (prosinec 2006)
 - skriptování – API pro skriptovací jazyky: Scripting Language Support (JSR 223) a vestavěná implementace JavaScriptu
 - Java Compiler API (JSR 199) – možnost volat kompilátor z programu

Předchozí verze

- Java 7 (červenec 2011), OpenJDK je referenční implementací Javy SE 7.
 - invokedynamic – podpora dynamických jazyků v JVM, obecně se z JVM stává platforma pro běh různých jazyků, ne jen Javy
 - switch podle Stringů
 - AutoCloseable – zdroje implementující toto rozhraní může automaticky zavírat try (...) { ... } blok a nemusí se o jejich zavírání explicitně starat programátor
 - tzv. diamantový operátor – příklad :new ArrayList<T>()
 - catch-multiple – zachytávání více typů výjimek v jednom catch
 - rozšířené atributy – API pro práci s rozšířenými atributy souborů a symbolickými odkazy

Default interface methods

- non-abstract method implementations to interfaces by utilizing the default **keyword**.

```
interface Formula {  
    double calculate(int a);  
  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

Lambda expressions

- Zjednodušení vývoje a přináší funkcionální programování do Javy
- Definice :

parameter -> expression body

Lambda expressions

```
//pomocí anonymní třídy  
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
```

```
Collections.sort(names, new Comparator<String>() {  
    @Override  
    public int compare(String a, String b) {  
        return b.compareTo(a);  
    }  
});
```

```
//lambda 1  
Collections.sort(names, (String a, String b) -> {  
    return b.compareTo(a);  
});
```

```
//lambda 2  
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

Lambda expressions

```
public class Java8Tester {
    public static void main(String args[]){
        Java8Tester tester = new Java8Tester();

        //with type declaration
        MathOperation addition = (int a, int b) -> a + b;

        //with out type declaration
        MathOperation subtraction = (a, b) -> a - b;

        //with return statement along with curly braces
        MathOperation multiplication = (int a, int b) -> { return a * b; };

        //without return statement and without curly braces
        MathOperation division = (int a, int b) -> a / b;

        System.out.println("10 + 5 = " + tester.operate(10, 5, addition));
        System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));
        System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));
        System.out.println("10 / 5 = " + tester.operate(10, 5, division));
    }
}
```

```
interface MathOperation {
    int operation(int a, int b);
}

private int operate(int a, int b,
    MathOperation mathOperation){
    return mathOperation.operation(a, b);
}
```


Lambda scope

- Na co si dát pozor !

```
int num = 1;  
Converter<Integer, String> stringConverter =  
    (from) -> String.valueOf(from + num);  
//ok přeloží se až sem  
//num nemusí být final od anonymní třídy
```

```
num = 3;  
//zapisovat do num mimo scope lambdy je  
zakázáno  
//v době překladu totiž num musí být už final
```

Functional Interfaces

- rozhraní (interface), která mají právě jednu abstraktní metodu.
- Kontrola typovosti pro lambda výrazy

```
@FunctionalInterface  
interface Converter<F, T> {  
    T convert(F from);  
}
```

```
Converter<String, Integer> converter =  
    (from) -> Integer.valueOf(from);  
Integer converted = converter.convert("123");  
System.out.println(converted); // 123
```

Method references

- Pomocí klíčového znaku `::` můžeme předávat odkazy na funkce a konstruktory

```
Converter<String, Integer> converter =  
    (from) -> Integer.valueOf(from);  
Integer converted = converter.convert("123");  
System.out.println(converted); // 123
```

```
Converter<String, Integer> converter =  
    Integer::valueOf;  
Integer converted = converter.convert("123");  
System.out.println(converted); // 123
```

Method references

- Další ukázka ale s instanční metodou

```
class Something {  
    String startsWith(String s) {  
        return String.valueOf(s.charAt(0));  
    }  
}
```

```
Something something = new Something();  
Converter<String, String> converter =  
    Something::startsWith;  
String converted = converter.convert("Java");  
System.out.println(converted); // "J"
```

Method references

- Další ukázka s konstruktorem

```
interface PersonFactory<P extends Person> {  
    P create(String firstName, String lastName);  
}
```

```
class Person {  
    String firstName;  
    String lastName;  
  
    Person() {}
```

```
    PersonFactory<Person> personFactory =  
        Person::new;  
    Person person =  
        personFactory.create("Peter", "Parker");
```

```
    Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

Predikáty

- Funkce které vrací true nebo false

```
Predicate<String> predicate = (s) -> s.length() > 0;
```

```
predicate.test("foo");           // true  
predicate.negate().test("foo");  // false
```

```
Predicate<Boolean> nonNull = Objects::nonNull;  
Predicate<Boolean> isNull = Objects::isNull;
```

```
Predicate<String> isEmpty = String::isEmpty;  
Predicate<String> isEmpty = isEmpty.negate();
```

Streams

- Reprezentují sekvenci elementů (vytváříme z listů, map apod..)
- Typy :
 - intermediate (vrací typ)
 - Terminal (vrací stream)
- Ulehčují práci s kolekcemi
- Mohou pracovat sekvenčně i paralelně
- Možnost generovat spousty statistických informací (lazy initialization → jinak by to zpomalovalo výkon)

Streams příklady

```
//kolekce  
List<String> stringCollection = new ArrayList<>();  
stringCollection.add("string");  
..
```

```
//filter example  
StringCollection  
    .stream(). //intermediate  
    filter((s) -> s.startsWith("s"))//intermediate  
    .forEach(System.out::println);//terminal (void)
```

```
//sorted  
stringCollection  
    .stream()  
    .sorted()  
    .filter((s) -> s.startsWith("s"))  
    .forEach(System.out::println);
```


Streams příklady

```
//kolekce  
List<String> stringCollection = new ArrayList<>();  
stringCollection.add("string");  
..
```

```
//filter example  
StringCollection  
    .stream(). //intermediate  
    filter((s) -> s.startsWith("a"))//intermediate  
    .forEach(System.out::println);//terminal (void)
```

```
//sorted  
stringCollection  
    .stream()  
    .sorted()  
    .filter((s) -> s.startsWith("a"))  
    .forEach(System.out::println);
```

Sorting

- Java 7 vs Java 8 přístup

```
Array.sort(myArray);  
//využívá jen jedno jádro počítače
```

```
Arrays.parallelSort(myArray);  
//využívá paralelní přístup  
//rozdělý kolekci na několik částí, paralelně setřídí a vrátí  
zpět  
//rychlejší, výhodnější
```

**otázka : bude algoritmus vždy 2x rychlejší když poběží na 2 procesorech ?
(paralelismus se dá zřídit na dvou úrovních vlákn a procesory)**

Sequential Sort

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));

// sequential sort took: 899 ms
```

Parallel Sort

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms
```

Děkuji za pozornost

Jiří Šebek

