PROJECT LAMBDA

Kuracja odmładzająca dla Javy.

O MNIE

Java

javablog.eu

FAKTY

JSR 335 - wyrażenia lambda dla Javy.

OpenJDK Project Lambda - prototypowa implementacja.

Część Javy 8 - GA 9 września 2013.

WYRAŻENIA LAMBDA

Anonymous functions have been a feature of programming languages since Lisp in 1958. An increasing number of modern programming languages support anonymous functions, and some notable mainstream languages have recently added support for them, the most widespread being JavaScript,[1] C#,[2] Ruby[3] and PHP[4]. Anonymous functions were added to C++ in C++11. Some object-oriented programming languages have anonymous classes, which are a similar concept, but do not support anonymous functions. Java is such a language (...)

wikipedia

C++

```
std::vector<int> some_list;
int total = 0;
for (int i=0;i<5;++i) some_list.push_back(i);
std::for_each(begin(some_list), end(some_list),
  [&total](int x) { total += x; });
```

C#

```
var values = new List<int>(){7,4,9,3};

var foo = values.ConvertAll(d => d*d);
```

SCALA

```
val list = List(1, 2, 3, 4)

list.reduceLeft((x, y) => x + y)
```

JAVA - JAK JEST TERAZ?

FUNCTIONAL INTERFACE

Jedna metoda wymagana do implementacji.

```
public interface FunctionalInterface
{
    int method(int number);
    String toString();
}
```

FUNCTIONAL INTERFACE

```
void otherMethod(FunctionalInterface attr) {...}
```

```
otherMethod(new FunctionalInterface()
{
    public int method(int number)
    {
        return number * 3;
    }
});
```

FUNCTIONAL INTERFACE

```
void otherMethod(FunctionalInterface attr) {...}
```

```
otherMethod(

                              number

                    number * 3

 );
```

KISS

```
void otherMethod(FunctionalInterface attr) {...}
```

```
otherMethod(number -> number * 3);
```

KISS

```
otherMethod(number -> number * 3);
```

```
otherMethod(new FunctionalInterface()
{
    public int method(int number)
    {
        return number * 3;
    }
});
```

KISS

```
otherMethod(number -> number * 3);
```

```
otherMethod((int n) -> n * 25);

otherMethod(n -> { return n * 25; });

otherMethod((int n) -> { return n * 25; });
```

TARGET TYPE

Skąd wiadomo, jakiemu typowi odpowiada wyrażenie?

TARGET TYPE

```
interface First
{
    int add(int number);
}
```

```
interface Second
{
    int calc(int number);
}
```

TARGET TYPE

```
interface First
{
    int add(int number);
}
```

```
First f = n -> n * 2;
```

```
interface Second
{
    int calc(int number);
}
```

```
Second s = n -> n * 2;
```

TARGET TYPE

```
interface First
{
    int add(int number);
}
```

```
First f = n -> n * 2;
```

```
interface Second
{
    int calc(int number);
}
```

```
Second s = n -> n * 2;
```

TARGET TYPE

W jakich kontekstach można użyć wyrażeń lambda?

TARGET TYPE

Zagnieżdżone wyrażenia lambda:

```
Callable<ActionListener> inst = () -> (e) -> {};
```

Doprecyzowanie typu docelowego:

```
Object inst = () -> {};
```

TARGET TYPE

Zagnieżdżone wyrażenia lambda:

```
Callable<ActionListener> inst = () -> (e) -> {};
```

Doprecyzowanie typu docelowego:

```
Object inst = () -> {};
```

TARGET TYPE

Zagnieżdżone wyrażenia lambda:

```
Callable<ActionListener> inst = () -> (e) -> {};
```

Doprecyzowanie typu docelowego:

```
Object inst = (Runnable) () -> {};
```

THIS, FINAL - LIVE DEMO

REFERENCJE METOD

```
public class StringCompareUtils
{
    public static int byLength(String a, String b)
    {
        return a.length() - b.length();
    }
}

strings.sort(StringCompareUtils::byLength);
```

REFERENCJE METOD

Metody statyczne:

```
ClassName::staticMethodName
```

Metody konkretnej instancji:

```
instanceReference::methodName
```

Metody nieokreślonej instancji:

```
ClassName::methodName
```

REFERENCJE METOD

```
                T[] a    Comparator<? super T> c
Arrays.sort(names, String::compareToIgnoreCase);
```

```
public int compare(String s1, String s2) {...}
```

```
s1.compareToIgnoreCase(s2)
```

REFERENCJE METOD

```
                T[] a    Comparator<? super T> c
Arrays.sort(names, String::compareToIgnoreCase);
```

```
public int compare(String s1, String s2) {...}
```

```
s1.compareToIgnoreCase(s2)
```

REFERENCJE METOD

Konstruktory:

`ClassName::new`

. . .

```
interface ObjectFactory
{
    Object getObject();
}

ObjectFactory factory = Thread::new;
```

INTERFEJSY

Co stanowi największy problem w rozwijaniu bibliotek?

METODY DOMYŚLNE

Dostarczają domyślną implementację metod interfejsu.

```
public interface ObjectFactory
{
    Object getObject() default
    {
        return new Object();
    }
}
```

METODY DOMYŚLNE

Dostarczają domyślną implementację metod interfejsu.

```
interface OtherFactory extends ObjectFactory
{
    Object getObject() default none;
}
```

DEFAULT + DZIEDZICZENIE

A jeśli pominiemy *default*?

```
interface OtherFactory extends ObjectFactory
{
    Object getObject();
}
```

DEFAULT + DZIEDZICZENIE

A jeśli pominiemy *default*?

```
interface OtherFactory extends ObjectFactory
{
    Object getObject() default
    {
        return new Object();
    }
}
```

DEFAULT + DZIEDZICZENIE

standardowa implementacja > domyślna implementacja

‘dokładniejsza’ implementacja domyślna wygrywaja

```
class C implements Y, Z
```

DEFAULT - KONFLIKTY

W razie konfliktu musimy zaimplementować metodę.

```
public class C implements Y, Z
{
    public void conflictingMethod()
    {
        System.out.println("My implementation");
    }
}
```

DEFAULT - KONFLIKTY

W razie konfliktu musimy zaimplementować metodę.

```
public class C implements Y, Z
{
    public void conflictingMethod()
    {
        Z.super.conflictingMethod();
    }
}
```

OK

Ale do czego można użyć tych wyrażeń?

COLLECTION API

LIVE DEMO

MATERIAŁY DODATKOWE

- JDK8 - jdk8.java.net/lambda
- openjdk.java.net/projects/lambda
    - State of the Lambda
    - State of the Lambda: Libraries Edition
- Netbeans 8 (nightly builds)
- javablog.eu

~~QUESTION TIME?~~

TALK TIME!