

Lambdas expressions

Jean-Luc Delarbre

Introduction

- **Function**
 - Unit of code that defines a processing with some input and output
- **Before Java 8, it was impossible to**
 - define them outside a class
 - pass them as function parameter



A useful programming technique

- **Easily modify code behavior only by changing its arguments**

```
double myMethod(KindOfProcessing definableProcessing, Other parameters...) {  
    //...  
    double someResult = myProcessing.apply(a, b);  
    //...  
    return finalResult;  
}
```

```
double val1 = myMethod((a, b) -> (a+b)/2, Other parameters);  
double val2 = myMethod((a, b) -> Math.abs(a-b), Other parameters);
```

- **Lambdas**

- Function hold by reference (thus passable as argument)
- Anonymous function



Functional interface why

- **Lambdas: Function (Unit of processing)**
 - Signature
 - Inputs
 - Output
 - Name
 - Some code
- **Functional interface defines lambdas signature**



Functional interface how

- **Java interface**
 - Only one abstract method
 - Optionally some default/static methods
- **Lambda signature = functional interface method signature**

```
@FunctionalInterface  
Interface MyFunctionalInterface {  
    ReturnType doProcess(ParameterType param, ...);  
}
```



Functional Interface annotation

- `@FunctionalInterface`
- **Communicate the intent**
- **Allow compiler check**
 - Usable with lambda



Still plain old Java

```
@FunctionalInterface
interface StatisticInfo {
    double compute(double a, double b);
}

class Mean extends StatisticInfo {
    double compute(double a, double b) {
        return (a+b)/2;
    }
}

static void main(String[] args) {
    double a = Double.parseDouble(args[1]);
    double b = Double.parseDouble(args[2]);

    StatisticInfo mean = new Mean();
    displayStatisticInfo(a, b, mean);
}

void displayStatisticInfo(double a, double b, StatisticInfo statisticInfo) {
    double val = statisticInfo.compute(a, b);
    System.out.println(val);
}
```



Anonymous class

```
@FunctionalInterface
interface StatisticInfo {
    double compute(double a, double b);
}

static void main(String[] args) {
    double a = Double.parseDouble(args[1]);
    double b = Double.parseDouble(args[2]);

    displayStatisticInfo(a, b, new StatisticInfo() {
        double compute(double a, double b) {
            return (a+b)/2;
        }
    });
}

void displayStatisticInfo(double a, double b, StatisticInfo statisticInfo) {
    double val = statisticInfo.compute(a, b);
    System.out.println(val);
}
```



Lambdas

```
@FunctionalInterface
interface StatisticInfo {
    double compute(double a, double b);
}

static void main(String[] args) {
    double a = Double.parseDouble(args[1]);
    double b = Double.parseDouble(args[2]);

    displayStatisticInfo(a, b, (a, b) -> (a+b)/2);
}

void displayStatisticInfo(double a, double b, StatisticInfo
statisticInfo) {
    double val = statisticInfo.compute(a, b);
    System.out.println(val);
}
```



Lambdas advantages

- **Enhance readability**

- More focus on intent
 - Less boilerplate code
- Facilitate functional programming style
- Syntactic shortcut (sugar) for anonymous class

- **Lambdas**

- No more than few lines
- self explanatory



Lambdas syntax

- **Parameters**

- 0, 1 or many
- Almost always inferred by compiler according to functional interface definition (same for return value)

- **(parameters) -> expression;**

```
() -> ;
```

```
() -> System.out.println("Hello");
```

```
a -> 2*a; (a) -> 2*a; (int a) -> 2*a;
```

```
(a, b) -> Math.abs(a-b);
```

- **(parameters) -> {instructions;}**

```
(a, b) -> {
```

```
    double mean = (a+b)/2;
```

```
    double span = Math.abs(a-b);
```

```
    return Math.max(mean, span);
```

```
};
```



Variables scope

- **Like anonymous class, lambdas could create closure (capture of variable)**

```
// Must be effectively final  
(final) int someInt = someInstruction;
```

```
Consumer<Integer> c = a -> a+someInt;
```

- **Usable variables are:**
 - Lambda parameters
 - Variable defined in lambda body
 - (Effectively) final variables in enclosing context



Method reference

- **Instead of defining lambda, allow to call existing**
 - Static method
 - Method of instance
 - Method of arbitrary object
 - Constructor



Method reference sample

```
@FunctionalInterface
interface StatisticInfo {
    double compute(double a, double b);
}

Class SomeClass {
static void main(String[] args) {
    double a = Double.parseDouble(args[1]);
    double b = Double.parseDouble(args[2]);

    displayStatisticInfo(a, b, SomeClass::meanS);
    displayStatisticInfo(a, b, new SomeClass()::mean); // this::mean possible from
                                                        // an instance of the class
}

void displayStatisticInfo(double a, double b, StatisticInfo statisticInfo) {
    double val = statisticInfo.compute(a, b);
    System.out.println(val);
}

static double meanS(double a, double b) {
    return (a+b)/2;
}

double mean(double a, double b) {
    return (a+b)/2;
}
}
```



JDK predefined Functional interface

- **Consumer<T>**
 - DoubleConsumer, IntConsumer ...
- **Function<T,R>**
 - BiFunction<T,U,R>, DoubleToIntFunction ...
- **Predicate<T>**
 - DoublePredicate
- **Supplier<T>**
 - BooleanSupplier, ...



JDK API usage

```
public static void main(String[] args) {  
    Function<Integer,Long> doubler = (i) -> (long) i * 2;  
    System.out.println(doubler.apply(2));  
}
```

```
public static void main(String[] args) {  
    Predicate<Integer> pair = i -> i%2;  
    Predicate<Integer> multiple3 = i -> i%3;  
  
    boolean multiple2and3 = pair.and(multiple3).test(6);  
    System.out.println(multiple2and3);  
}
```

