An Introduction to Functional Programming in Java

In this lecture we explain how the functional programing paradigm is implemented in Java. We begin by reviewing the concepts of lambda expression and functional interface, outline features such as method reference, and conclude with examples of using functional idioms.

1. Functional Programming

Functional programming is a style of programming where the model of computation evolves around evaluating function. While in the object-oriented programming the basic element of programming is an object, in the functional programming the fundamental unit of programming is a function.

A function takes zero or more arguments and produces a single value. It maps a set of values called arguments to a single result value. For example, if you want to determine the minimum of two integers you could define a function that takes as arguments two integers and returns a integer:

```
int minimum(int x, int y);
```

Likewise, you are asked to determine if a string has all characters in uppercase:

```
boolean isUppercase(String str);
```

One can easily generalize the function for a generic object T. The function tests if the object T has a specific property and has the signature:

```
boolean test(T t);
```

Thus, the `test` function maps a type T to a `boolean` type and can be formally represented as:

```
T -> boolean
```

However, the question remains: could such a function be defined in Java? The answer is yes. Furthermore, what is its syntax and how could one use it? In order to answer these questions we need to define a concept called *`lambda expression`*.

2. Lambda Calculus

Lambda expression is part of a system called *`lambda calculus`*. Alfonso Church created the formal system labeled lambda calculus or λ-calculus in 1936 (λ - lambda is a greek letter). The fundamental operation of the λ-calculus is an *`application`*.

If we consider `F` as a function (algorithm) and `I` as data (input) then an `application` in lambda calculus can be expressed as:

```
F.I
```

Another basic operation in lambda calculus is *abstraction*. If E[x] is an expression containing (depending on) x, then the abstraction:

λx.E[x]

denotes the function that maps x to E[x]. The function can be represented **x -> E[x]**.

Abstraction and application work together nicely. For example, if the abstraction λx.x+1 is applied to an argument y then the expression (λx.x+1)y can be reduced to the value y+1, by replacing the formal parameter x with the actual parameter y in the body of the function which is x+1.

The expression (λx.x+1)3 denotes the function x->x+1 applied to the argument 3, giving the value 3+1 which is 4. Therefore, a lambda expression could be used for defining a function without a name (a nameless or anonymous function).

3. Lambda expression

A *Lambda expression* represents an anonymous function by defining its arguments and its body. It consists of a set of parameters, followed by the *Lambda operator* (**->**) and body. The general format of a lambda expression is:

```
( parameters ) -> { body }
```

A lambda expression can receive zero or parameters enclosed in parentheses, separated by commas, whose type could be inferred from the context. The body of the lambda expression can contain an expression or a block statement.

For instance, lambda expression x->x+1 states that given a number it increments it.

When there is only one parameter, whose type could be inferred, the parentheses can be excluded. If the body of the lambda expression has only one statement the return type of the anonymous function is the same as that of the expression. Furthermore, the curly brackets are not required.

However, where there is more than one statements the curly brackets are mandatory and the return type is void if nothing is returned, or the type of the value returned from the block statement.

```
(int x, int y)  ->  {  int min = x < y  ?  x : y; return min; }
```

4. Example of lambda expressions

`() -> { };`	No parameters, empty block statement
`() -> {System.out.println("Lambda");};`	No parameters, block statement
`() -> 24`	No params, returns the integer 24

`(int x, int y) -> x + y;`	Two integers returns the sum
`(Object x) -> x`	Given an object, it returns it
`(Person p) ->  p.age > 25 && p.salary < 2000`	Given reference to Person returns boolean
`n -> n % 2 == 0;`	Given a number returns boolean
`(String s1, String s2) ->` `  s1.length() + s2.length();`	Given two strings returns an integer

The syntax of a lambda expression is clear. However, how could we introduce a lambda expression in Java where everything is an object. The answer is related to a concept called: `Functional Interface`.

5. Functional Interface

A *functional interface* is a Java interface that has *a single abstract method.*. To explicitly mark an interface as being functional interface there is an annotation called `@FunctionalInterface.` (The interface could have more than one method if they are static or default methods.)

Let us suppose that given two integer we would like to define a special operation of summing these two numbers if and only if the sum of them is an even number. Therefore we create an interface with only one method that takes two integers and returns a boolean value.

```
boolean evenSum(int x, int y);
```

This could be used to create a functional interface. Here is how it looks like:

```
/**************************************************************************
 *  Compilation:  javac Summable.java
 *
 *  Summable - Interface with a single abstract method
 *  Summable is a functional interface
 **************************************************************************/

@FunctionalInterface
public interface Summable {

    /**
     * Returns true only if the sum of params is even
     *
     * @param x the integer operand
     * @param y the integer operand*
```

```
     * @return  true if the sum of x and y is an even number
     */
    boolean evenSum(int x, int y);
}
```

6. From Interface to Lambda Expression

Starting from `Summable` interface, let us see how we can use it in our program:

6.1 The first way of using an interface in Java was introduced in JDK 1.0. One can create a class that implements the `Summable` interface. Here is a simple implementation:

```
/**************************************************************************
 *  Compilation:  javac FirstWay.java
 *  Execution:    java FirstWay
 *
 *  Classical example of using an interface by developing a class
 *  that implements it.
 *
 *  Output
 *  Is sum even? false
 *
 * @author Jordan Anastasiade
 * @version 1.0, 12 Aug 2017
 *************************************************************************/
public class FirstWay implements Summable{

    /**
     * The implementation of evenSum
     * defined in Summable interface
     *
     * @param x the integer operand
     * @param y the integer operand
     * @return true if the sum of x and y is an even number
     */
    @Override
    public boolean evenSum(int x, int y) {
        return (x + y) % 2 == 0;
    }
```

```
    public static void main(String[] args) {

        //create the obj of type Summable
        Summable obj = new FirstWay();

        //invoke method eventSum and print the result
        System.out.println("Is sum even? " + obj.evenSum(1, 2));
    }
}
```

6.2 The second way to create an object of interface type is at least convoluted as far as the code readability. It takes advantage of anonymous class (class without name). Here is an example of working with the `Summable` interface by creating an anonymous class.

```
/**************************************************************************
 *  Compilation:  javac SecondWay.java
 *  Execution:    java SecondWay
 *
 *  Example of using an interface by implementing it with
 *  an anonymous class
 *
 *  Output
 *  Is sum even? false
 *
 * @author Jordan Anastasiade
 * @version 1.0, 13 Aug 2017
 *************************************************************************/
public class SecondWay {

    public static void main(String[] args) {

        //anonymous class
        //create the object of type Summable and invoke eventSum on it

        System.out.println("Is sum even? " + new Summable() {
            @Override
            public boolean evenSum(int x, int y) {
                return (x + y) % 2 == 0;
            }
        }.evenSum(1, 2));
    }
```

```
}
```

6.2 The third way to create an object of interface type, introduced in Java 8, is based on the lambda expressions. It is simple and elegant, easy to implement and understand.
A lambda expression for the method:

```
boolean evenSum(int x, int y);
```

Is

```
(x, y) -> { return (x + y) % 2 == 0; };
```

Here is an example of working with the `Summable` interface by employing a lambda expression:

```
/**************************************************************************
 *  Compilation:  javac ThirdWay.java
 *  Execution:    java ThirdWay
 *
 *  Example of using an interface by defining a lambda expression
 *
 *  Output
 *  Is sum even? false
 *
 * @author Jordan Anastasiade
 * @version 1.0, 13 Aug 2017
 *************************************************************************/
public class ThirdWay {

    public static void main(String[] args) {

        //create an obj of type Summable using a lambda expression:
        //(x, y) -> { return (x + y) % 2 == 0; };
        Summable obj = (x, y) -> { return (x + y) % 2 == 0; };

        System.out.println("Is sum even? " + obj.evenSum(1, 2));
    }
}
```

One of the obvious advantage of using lambda expressions in Java is based on code simplicity. It eliminates a lot of boilerplate code and lets us concentrate on the core functionality. More importantly, it encapsulates the behaviour as block of code, that can be used as method params or can be the return value of a method.

After this short introduction, we will study further the lambda expressions in the context of functional programming in Java.