Functional Interfaces and Lambda Expressions

Objectives

- Understanding Functional Interfaces
- Anonymous inner classes and Functional Interfaces (Legacy code)
- Lambda Expressions:
  - Relation to Functional Interfaces
  - Structure and syntax

Functional Interfaces

- A Functional Interface is any Interface that…
  - Has only ONE ABSTRACT METHOD
  - Can have any number of default and static methods as long as it only have ONE ABSTRACT METHOD
- A functional interface can be marked with a special annotation to check that it is a functional interface.
- @FunctionalInterface

Functional Interface Example

A Functional Interface contains…

```
@FunctionalInterface
public interface DoStringStuff {

    //This Implementation could be ANYTHING that take two Strings and return a String
    String operate(String s1, String s2);

}
```

…only one abstract method

Implement Functional Interface

We could implement our interface and define an implementation

```
public class StringOperator implements DoStringStuff{

    @Override
    public String operate(String s1, String s2) {
        return s1.concat(s2);
    }

}
```

We could implement this interface in a lot of classes and come up with lots of useful implementations…

Lets be a bit more flexible…

And we can instantiate our class and call our implemented method

```
public static void main(String[] args) {
    StringOperator concatter = new StringOperator();
    String firstName = "Erik", lastName = "Svensson";
    System.out.println(concatter.operate(firstName, " " + lastName));
}
```

Anonymous Inner Classes

- It´s a local inner class that does not have a name.
- Declared and instantiated in one statement using the new keyword
- Can use it to extend existing class or implement an Interface.
- Like creating implementation on the fly.

Anonymous Inner Class example

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

    //Creating our Anonymous inner class
    DoStringStuff getBiggestString = new DoStringStuff() {

        @Override
        public String operate(String s1, String s2) {
            return s1.length() >= s2.length() ? s1 : s2;
        }

    }; //Notice semicolon. We are actually creating a local variable AND its implementation here.

    System.out.println(getBiggestString.operate("Erik", "Svensson"));

}
```

Moving Anonymous Inner classes to fields

```java
//We could break out our anonymous inner class and make a field
static DoStringStuff getBiggestString = new DoStringStuff() {
    @Override
    public String operate(String s1, String s2) {
        return s1.length() >= s2.length() ? s1 : s2;
    }
};

static DoStringStuff concatenate = new DoStringStuff() {
    @Override
    public String operate(String s1, String s2) {
        return s1 + s2;
    }
};
```

This means that we can store behaviour and use it in a method…

Functional Interfaces in method parameters

```
public static String doStringStuff(String s1, String s2, DoStringStuff operation) {
    return operation.operate(s1, s2);
}


public static void main(String[] args) {

    System.out.println(doStringStuff("Erik ","Svensson", concatenate));
    System.out.println(doStringStuff("ABCD","EFGHIJK", getBiggestString));

    //Making an implementation inline
    String result = doStringStuff("Hello", "World", new DoStringStuff() {

        @Override
        public String operate(String s1, String s2) {
            StringBuilder sb = new StringBuilder();
            sb.append(s1 + " " + s2).reverse();
            return sb.toString();
        }

    });

    System.out.println(result);
}
```

Using Functional Interfaces in method parameters allow us to send our implementation(”behaviour”) as an argument.

We can also define our behaviour when our method is invoked.

This method can do anything that takes two Strings and returns a String

Anonymous Inner Classes drawbacks.

- Code gets harder to read.
- Its not concise enough.

Solution using Lambda expressions

```java
public static void main(String[] args) {
    DoStringStuff getBiggestString = (String s1, String s2) -> s1.length() >= s2.length() ? s1 : s2;
    DoStringStuff concatenate = (String s1, String s2) -> s1 + s2;

    String biggest = doStringStuff("ABC", "ABCD", getBiggestString);
    String combined = doStringStuff("Erik ", "Svensson", concatenate);
    String reversed = doStringStuff("Erik", "Svensson", (s1,s2) -> new StringBuilder(s1 + " " + s2).reverse().toString());
}

public static String doStringStuff(String s1, String s2, DoStringStuff operation) {
    return operation.operate(s1, s2);
}
```

Lambda Expression Defined

Argument list Arrow Body

```java
(String str1, String str2) -> str1 + str2;
```

```java
public interface DoStringStuff {
    String operate(String s1, String s2);
}
```

All lambda expressions have consists of arguments, arrow token and body

We want to match the arguments to the abstract method in our interface.
The String names doesn't need to match just the signature (Order and Type)

The arrow token tells the JVM that on the right side comes the actual implementation

Since the implementation is only one statement, no return statement is needed.

Argument Lists and shorter Lambdas

DoStringStuff functional interface has ONE abstract method that takes two Strings as arguments

```
public interface DoStringStuff {
    String operate(String s1, String s2);
}
```

$\downarrow$

```
    String operate(String, String)
```

Method has this signature

$\downarrow$

```
DoStringStuff concatenate = (str1, str2) -> str1 + str2;
```

Shortened Lambda Expression

Since functional interfaces only have **one abstract method**, JVM infers that whatever arguments we write **MUST match** the signature.

Hence we dont need to specify the type of our arguments when we write lambda expressions

Single argument Lambda Expressions

```java
public interface Printer {
    void print(String s);
}
```

```java
public static void main(String[] args) {
    Printer printer = (String s) -> System.out.println(s);
    printer.print("Hello World"); //Hello World
}
```

```java
public interface Conditional {
    boolean testInt(int t);
}
```

```java
public static void main(String[] args) {
    Conditional positive = num -> num > 0;
    System.out.println(positive.testInt(5)); // TRUE
}
```

When argument list contains only one argument we don’t need to wrap it into parenthesis ()

No argument(s) Lambda Expression

```java
public interface IntRandomGen{
    int generate();
}

public static void main(String[] args) {
    //Need to be final or effectively final
    int upper = 10;
    //When having an empty argument list the empty () is required
    IntRandomGen rng = () -> new Random().nextInt(upper + 1);

    System.out.println(rng.generate()); //Any number between 0 - 10
}
```

When you have a method to implement that takes no arguments, you need to include empty parenthesis

Multi Statement Lambda Expressions

```java
public interface Calculator {
    double calculate(double number1, double number2, String operator);
}

public static void main(String[] args) {
    //MultiLine Lambda Expressions need to have a body
    Calculator calculator = (num1, num2, op) -> {
        switch(op) {
        case "+":
            return num1 + num2;
        case "-":
            return num1 - num2;
        case "*":
            return num1 * num2;
        case "/":
            return num1 / num2;
        default:
            return 0;
        }
    }; //Don't forget this semicolon

    double result = calculator.calculate(10.4, 9.6, "+"); //20.0
}
```

Sometimes you need to execute multiple statements in a lambda expression.

Important that you wrap your statements inside a scope {} with a semicolon ; after the scope.

Multi Statement Lambda Expressions

```
public interface DoStringStuff {
    String operate(String s1, String s2);
}

public static void main(String[] args) {

    //Compile error
    DoStringStuff concat = (s1, s2) -> return s1 + s2;
}
```

If you have a single line statement there is a implied return, thus if you specify your own return you will get a syntax error.

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

    //Works fine
    DoStringStuff concat = (s1, s2) -> {return s1 + s2;};
}
```

If you want to keep the return statement you need to give the Lambda Expression a scope and treat it like a Multi Statement Lambda Expression

Practice:

Create these two Interfaces

```java
public interface Action {
    void execute(Product p);
}

public interface Conditional {
    boolean test(Product p);
}
```

Create this class

```
Product
-----------------------------
private String productName
private double price
private int stock
-----------------------------
//Getters and setters
public String toString()
```

Your task is to make a method that takes a List of Products, a Conditional and an Action as arguments.

Method should iterate though the List and apply a filter using the Conditional. On each Product passing the filter you apply the Action.

You will have to make Lambda Implementations to help you accomplish the following scenarios.

- Print out all Products that have stock value of 0.
- Print out the productName of all the Products that starts with B.
- Print out all Products that have price above 100 AND lower than 150
- Increase the price of all Products that have a stock value of less than 10 AND above 0 by 50%

Questions?