Programare avansata pe obiecte - laborator 8 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 8

Functional programming:

Since Java 8 => support for functional programming via the lambda expression and Stream API.

A lambda expression is characterized by the following syntax:

```
(A list of parameters separated by commas) -> {expression body which
contains one or more statements}
```

A lambda expression can be shortened in two ways because JDK compiler supports type inference.

- ➢ Can omit the declaration of the parameter’s type. The compiler can infer it from the parameter’s value.
- ➢ Can omit the return keyword if the expression body has a single expression.

A lambda expression can be simplified with the following conditions:

- ➢ Can omit the parenthesis for a single parameter.
- ➢ Can omit the curly brackets if the expression body only contains a single statement.

Java Functional Interfaces:

- ➢ The term Java functional interface was introduced in Java 8.
- ➢ A functional interface in Java is an interface that contains only a single abstract (unimplemented) method.
- ➢ A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

Pure function: is a function that takes an input and returns an output. It has a single purpose and doesn’t mutate any state; It always produces the same output for the same input.

Java 8 provides 40+ common **predefined functional interfaces**. All of them except the Consumer FI are pure functions.

1. **Function:**

    a. A **Function FI accepts one argument and returns one result**. Its abstract method is called **apply(Object).**

    b. Java 8 provides several convenient FIs for the primitive data types: IntFunction, DoubleFunction, IntToDoubleFunction, IntToLongFunction, DoubleToIntFunction, DoubleToLongFunction, LongToDoubleFunction, and LongToIntFunction.

    c. A **BiFunction** FI accepts two arguments and produces a result. Its abstract method is called **apply(Object, Object).**

    d. Java 8 also provides ToDoubleBiFunction, ToIntBiFunction, and ToLongBiFunction that accepts two arguments and produces a double-valued, int-valued, and long-valued result.

```
package com.paolabs.lab8.ex1;

import java.util.Arrays;
import java.util.List;
import java.util.function.*;
import java.util.stream.Collectors;

public class FunctionFIDemo {

   public static String concatStringsBiFunction(String s1, String s2) {
       BiFunction<String, String, String> concat = (a, b) -> a + b;
       String combinedStr = concat.apply(s1, s2);
       return combinedStr;
   }

   public static int multiplyTwoIntsBiFunction(int i1, int i2) {
       BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;
       Integer product = multiply.apply(i1, i2);
       return product;
   }

   public static String convertDoubleToStringDoubleFunction(double d) {
       DoubleFunction<String> doubleToString = num -> Double.toString(num);
       return doubleToString.apply(d);
   }

   public static int convertDoubleToIntDoubleToIntFunction(double d) {
       DoubleToIntFunction doubleToInt = num -> (int) num;
```

```java
        return doubleToInt.applyAsInt(d);
    }

    public static long convertDoubleToLongDoubleToLongFunction(double d) {
        DoubleToLongFunction doubleToLongFunc = num -> (long) num;
        return doubleToLongFunc.applyAsLong(d);
    }

    public static void convertStringToIntegerFunction() {
        Function<String, Integer> convertToWordCount = String::length;
        List<String> words = Arrays.asList("The", "That", "John", "Thanks");

        List<Integer> wordsCounts =
words.stream().map(convertToWordCount).collect(Collectors.toList());

        for (int n : wordsCounts) {
            System.out.println(n);
        }
    }

    public static String convertIntegerToStringIntFunction(int number) {
        IntFunction<String> intToString = num -> Integer.toString(num);

        return intToString.apply(number);
    }

    public static double convertIntToDoubleIntToDoubleFunction(int number) {
        IntToDoubleFunction intToDoubleFunc = num -> (double) num;

        return intToDoubleFunc.applyAsDouble(number);
    }

    public static double powerTwoIntToDoubleBiFunction(int i1, int i2) {
        ToDoubleBiFunction<Integer, Integer> concat = (a, b) -> Math.pow(a,
b);
        double powerRet = concat.applyAsDouble(i1, i2);
        return powerRet;
    }
}
```

2. **Predicate:**

    a. A **Predicate FI** accepts **one argument and returns a Boolean value**. Its abstract method is **test(Object).**

    b. A BiPredicate FI accepts two arguments and returns a Boolean value. Java 8 also provides IntPredicate, LongPredicate, and DoublePredicate for the primitive data types.

```java
package com.paolabs.lab8.ex1;

import java.util.function.*;
import java.util.stream.Stream;

public class PredicateDemo {

   public boolean whichIsBiggerBiPredicate(int n1, int n2) {
       BiPredicate<Integer, Integer> isBigger = (x, y) -> x > y;
       return isBigger.test(n1, n2);
   }

   public boolean isPositiveDoublePredicate(double n) {
       DoublePredicate isPositive = x -> x > 0;
       return isPositive.test(n);
   }

   public boolean isNegativeIntPredicate(int n1) {
       IntPredicate isNegative = x -> x < 0;
       return isNegative.test(n1);
   }

   public boolean isDivisibleByThreeLongPredicate(int nr) {
       LongPredicate isDivisibleBy3 = x -> x % 3 == 0;
       return isDivisibleBy3.test(12);
   }

   public boolean isEvenPredicate(int nr) {
       Predicate<Integer> isEven = s -> s % 2 == 0;
       return isEven.test(nr);
   }

   public void streamFilter(String[] fruits) {
       Stream.of(fruits)
```

```
               .filter(fruit -> fruit.startsWith("A"))
               .forEach(fruit -> System.out.println("Started with A:" +
fruit));
   }
}
```

3. Supplier

a. A **Supplier FI accepts no argument** and **returns a result**. Its abstract method is **get()**.

b. Java 8 provides convenient interfaces for the primitive data types: IntSupplier, DoubleSupplier, BooleanSupplier, and LongSupplier.

```
package com.paolabs.lab8.ex1;

import java.util.function.*;

public class SupplierDemo {

   public boolean getAsBoolean() {
       BooleanSupplier booleanSupplier = () -> true;
       return booleanSupplier.getAsBoolean();
   }

   public double getAsDouble() {
       DoubleSupplier pi = () -> Math.PI;
       return pi.getAsDouble();
   }

   public int getAsInt() {
       IntSupplier maxInteger = () -> Integer.MAX_VALUE;
       return maxInteger.getAsInt();
   }

   public long getAsLong() {
       LongSupplier maxLongValue = () -> Long.MAX_VALUE;
       return maxLongValue.getAsLong();
   }

   public String asString() {
       Supplier<String> message = () -> "Mary is fun";
```

```
        return message.get();
    }
}
```

4. **Consumer**

   a. A **Consumer FI accepts a single argument and returns no result**. Its abstract method is **accept(Object).**

   b. Java 8 also provides convenient interfaces for the primitive data types: IntConsumer, LongConsumer, DoubleConsumer, BiConsumer, ObjtIntConsumer, ObjLongConsumer, and ObjDoubleconsumer.

```
package com.paolabs.lab8.ex1;

import java.util.Arrays;
import java.util.function.*;

public class ConsumerDemo {

   public void printBiConsumer() {
       BiConsumer<String, String> echo = (x, y) -> {
           System.out.println(x);
           System.out.println(y);
       };
       echo.accept("This is first line.", "Here is another line");
   }

   public void convertToLowercase() {
       Consumer<String> convertToLowercase = s ->
System.out.println(s.toLowerCase());
       convertToLowercase.accept("convert to ALL lowercase");
   }

   public void printPrefix() {
       Consumer<String> sayHello = name -> System.out.println("Hello, " +
name);
       for (String name : Arrays.asList("Silvia", "John", "Doe")) {
           sayHello.accept(name);
       }
   }
```

```
    public void printDoubleConsumer() {
        DoubleConsumer echo = System.out::println;
        echo.accept(3.3);
    }

    public void printIntConsumer() {
        IntConsumer echo = System.out::println;
        echo.accept(3);
    }

    public void printLongConsumer() {
        LongConsumer echo = System.out::println;
        echo.accept(34L);
    }
}
```

5. UnaryOperator

a. A **UnaryOperator FI is a specialization of Function whose operand and result are the same type.** Its abstract method is **apply(Object).**

b. Java 8 provides separated classes for the primitive data types: IntUnaryOperator, DoubleUnaryOperator, and LongUnaryOperator.

```
package com.paolabs.lab8.ex1;

import java.util.function.DoubleUnaryOperator;
import java.util.function.IntUnaryOperator;
import java.util.function.LongUnaryOperator;
import java.util.function.UnaryOperator;

public class UnaryOperatorDemo {

    public void convertToUppercase() {
        UnaryOperator<String> convertToUppercase = String::toUpperCase;

        String uppercase = convertToUppercase.apply("this will be all
uppercase");
        System.out.println(uppercase);
    }

    public void doubleIt(int d) {
        IntUnaryOperator doubledIt = x -> x * 2;
        System.out.println(doubledIt.applyAsInt(d));
    }

    public void squareItLongUnaryOperator() {
        LongUnaryOperator squareIt = x -> x * x;
        System.out.println(squareIt.applyAsLong(12));
    }

    public void squareItDoubleUnaryOperator() {
        DoubleUnaryOperator squareIt = x -> x * x;
        System.out.println(squareIt.applyAsDouble(12));
    }

}
```

6. **BinaryOperator**

    a. A **BinaryOperator FI is a specialization of BiFunction whose operands and result are the same type**. Its abstract method is **apply(Object)**.

    b. Java 8 provides separated classes for the int, long, and double data type as IntBinaryOperator, LongBinaryOperator, and DoubleBinaryOperator.

```
package com.paolabs.lab8.ex1;

import java.util.function.BinaryOperator;
import java.util.function.DoubleBinaryOperator;
import java.util.function.IntBinaryOperator;
import java.util.function.LongBinaryOperator;

public class BinaryOperatorDemo {

    public void add() {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        Integer sum = add.apply(10, 12);

        System.out.println(sum.intValue());
    }

    public void addNumbers() {
        IntBinaryOperator add2 = (a, b) -> a + b;
```

```
       int sum = add2.applyAsInt(10, 12);
       System.out.println(sum);
   }

   public void multiplyNumbers() {
       LongBinaryOperator add2 = (a, b) -> a * b;
       long product = add2.applyAsLong(10, 12);
       System.out.println(product);
   }

   public void powerToNumber() {
       DoubleBinaryOperator add2 = (a, b) -> Math.pow(a, b);

       double powerRet = add2.applyAsDouble(10, 2);
       System.out.println(powerRet);
   }

}
```

7. Customized Functional Interfaces

a. @FunctionalInterface - marks an interface as a FI. J

b. Java compiler will throw an error when an interface marked with @FunctionalInterface has more than one abstract methods.

```
package com.paolabs.lab8.ex1;

@FunctionalInterface
public interface CustomFI {

   void hello(String hello);
}
```

```
package com.paolabs.lab8;

import com.paolabs.lab8.ex1.CustomFI;
```

```
public class Main {

   public static void main(String[] args) {
       CustomFI customFI = msg -> System.out.println(msg);
       customFI.hello("Hello everyone!");
   }
}
```