Lambda Expressions

”A lambda expression is generally - in mathematics and computing - a function.” Lambda expressions

- enable to represent anonymous methods with a more compact syntax (methods without a name).
- introduce the idea of functions into the Java language (functional programming).
- allow to create and use single method classes.

A Lambda expression λ

- must have a functional interface (FI). A FI is an interface that has one abstract method and zero or more default methods.
- is an instance of a functional interface, which is itself a subtype of `Object`.
- typically includes a parameter list, a return type, and a body:
  `(parameter list) -> { statements; }`
  or
  `(parameter list) -> expression`

The Hello Lambda World Example

```
1 public class HelloLambdaWorld {
2    public static void main(String[] args) {
3       String[] arr = {"Hello ", "Lambda ", "World!"};
4       /*1*/ Arrays.asList(arr).forEach(s->System.out.print(s));
5    }
6 }
```

The method forEach expects a Consumer-Object as parameter.

```
1 void Iterable.forEach(Consumer<? super T> action) { ... }
```

The Consumer-Interface looks like as follows.

```
1 @FunctionalInterface
2 public interface Consumer<T> {
3    void accept(T t);
4 }
```

The lambda expression provides code for the accept()-method. This could be done by an anonymous class as well:

```
1 Arrays.asList(arr).forEach(new Consumer<String>() {
2    public void accept(String t) {
3       System.out.print(t);
4    }
5 });
```

When dealing with lambda expressions the compiler has to know the following facts:

- The definition of the forEach()-method informs the compiler that the method expects a Consumer-Object.
- The Consumer-Interface is a functional interface. So, the compiler knows that the lambda expression delivers the code of the accept()-method.
- The compiler knows the datatype of the parameter s of the lambda expression because the forEach()-method has been used for a list with String-Objects.

Java and Lambda Expressions λ

When javac (the Java compiler) encounters a LE, it interprets it as **the body of a method with a specific signature**. To identify this method, it looks at the surrounding code. To be legal Java code, the LE must satisfy the following:

1. The LE must appear where an instance of an interface type is expected.
2. The expected interface type should have exactly one mandatory method.
3. The expected interface method should have a signature that exactly matches that of the LE.

Some more Examples

1. take two integers and return their sum

```
(int x, int y) -> x + y
```

2. take two integers and return their difference

```
(x, y) -> x - y
```

3. takes no values and returns 50

```
() -> 50
```

4. takes a string, prints its value, and returns nothing

```
(String s) -> System.out.println(s);
```

5. takes a number and returns the result of doubling it

```
x -> 2*x
```

6. takes a collection, clears utm and returns its previous size

```
c -> {int s = c.size(); c.clear(); return s; }
```

   Parameter types may be

   (a) explicitly declared (1,4) or
   (b) implicitly inferred (2,5,6)

   They may not be mixed in a single lambda expression.
   The body may be

   (a) a block (6) (can return a value or nothing) or
   (b) an expression (1-5) (may return a value or nothing)

   Parantheses may be avoided for a single inferred-type parameter (5,6).

7. Simple JavaFX GUI

```
Button btn = new Button("Click Me!");
/* Anonymous Inner Class */
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        btn.setText("You clicked Me!");
    }
});
```

   To refactor this anonymous inner class into a lambda expression:

```
Button btn = new Button("Click Me!");
/* Lambda Expression */
btn.setOnAction((ActionEvent event) -> {
        btn.setText("You clicked Me!");
});
```

or even just

```
Button btn = new Button("Click Me!");
/* Lambda Expression */
btn.setOnAction(event -> { btn.setText("You clicked Me!"); });
```

8. Calculator - Lambda expression that takes more than one formal parameter

```
package lambdas;
public class Calculator {
   interface IntegerMath {
      int operation (int a, int b);
   }

   public int operate(int a, int b, IntegerMath op) {
      return op.operation(a,b);
   }

   public static void main(String[] args) {
      Calculator myCalc = new Calculator();
      IntegerMath addition = (a, b) -> a + b;
      IntegerMath subtraction = (a, b) -> a - b;
      System.out.println("40 + 2 = " + myCalc.operate(40, 2, addition));
      System.out.println("20 - 10 = " + myCalc.operate(20, 10, subtraction));
      System.out.println("5 * 10 = " + myCalc.operate(5, 10, (a, b) -> a * b));
   }
}
```

Method References

For some lambda expression there may be a method having exactly fitting parameter list and return type. In this case the lambda expression can be replaced with a so-called method reference:

object reference :: instance method Example. Print the elements of a list.

```
List<Double> ld = new ArrayList<>();
ld.add(1.0);
ld.add(2.0);
ld.add(3.0;
ld.forEach(System.out::println);
```

or with help of the λ-function:

```
ld.forEach(d->System.out.println(d));
```

class name :: instance method Example. Contact with prettyPrint.

```
Arrays.asList(contacts).forEach(Contact::prettyPrint);
```

or with help of the λ-function:

```
Arrays.asList(contacts).forEach(k->k.prettyPrint());
```

class name :: class method Example. Get the minimum element of a list. (The list is executed as stream. reduce() compares the first two numbers, determines the minimum of these two numbers and takes the result for comparing with the next number in the list.)

```
List<Integer> data = Arrays.asList(7,2,5,4);
Optional<Integer> minimum = data.stream().reduce(Math::min);
System.out.println(minimum);
```

or with help of the λ-function:

```
minimum = data.stream().reduce(i1,i2)->Math.min(i1,i2);
```

An instance method can either be used for a concrete object reference or a class. A static method has to be used in connection with a class name. The class_name/object_reference is followed by the ::-operator, followed by the method_name without parameter list.

Do It

1. *List of Strings.*

   Given is the following list of Strings:

```
String[] msg = {"Java", "makes", "us", "HAPPY!"}
List<String> lst = Arrays.asList(msg);
```

   Formulate a lambda expression for lst.forEach to print the strings in small letters.

2. *List of Appointments.*

   Given is the following class for managing appointments:

```
public class Appointment {
   public String description;
   public String city;
   public java.time.LocalDateTime time;

   public Appointment(String descr, String where, LocalDateTime time) {
      description = descr;
      city = where;
      this.time = time;
   }
}
```

   You store several Appointment-Objects in an array and would like to sort them according to time. Use Arrays.sort and deliver as second parameter a lambda expression for implementing the compare()-method of the Comparator-interface.

3. *List of Integers.*

   Given is the following code fragment:

```
Integer[] data = {1,2,3};
List<Integer> lst = Arrays.asList(data);
lst.forEach(System.out::println);
```

   Present an alternative implementation by using a lambda expression.