Functional Programming in Java

Fritz Solms
August 24, 2015

1 Introduction

1.1 Introduction to functional programming

Since late 1980’s we have experienced a steady growth of functional programming languages. Lisp was widely used in the artificial intelligence boom and resulted in the introduction of a range of other functional programming languages including Haskell, Clojure, and Erlang. Although these languages have seen a growing adoption, the use of pure functional languages is still relatively uncommon.

However, the strengths of functional programming languages has resulted in the introduction of hybrid languages like Scala (supporting object-oriented and functional paradigms) as well as traditional object-oriented languages incorporating more and more functional concepts (e.g. Java).

The theoretical (mathematical) framework for functional programing is λ-calculus. This formal logic framework was developed by Alonzo Church and Stephen Kleene in the 1930s. Two core simplifications made the formalism more tractable and made it practical for implementation of functional programming languages. These are the concept of anonymous functions which are specified directly as a mapping

\[(x, y) \mapsto x^2 - y\]

and the observation that multiple-parameter functions can be mapped onto single parameter functions which return another function

\[f(x = 2, y = 3) \mapsto (x^2 - 3)|_{x=2} = 1\]

The latter is commonly known as currying.

Interestingly, functional programming languages have been with us from the start. IPL was developed in 1956 (one year prior to FORTRAN) and Lisp in 1958 (just one year after FORTRAN). This was followed by a range of functional programming language leading to the development for an open standard for functional programming, Haskell, which was released in 1987.

Nevertheless, none of the pure functional programming languages have thus far achieved a level of adoption which is comparable with the common programming languages like Java, C, C++, JavaScript, Python, . . . . Instead, many of these widely adopted languages have introduced better support for functional programming within the context of imperative programming.

1.1.1 Functional programming concepts

Functional programming introduces a range of concepts which are different from imperative programming approaches like procedural or object-oriented programming. Being based on a
solid mathematical basis, functional programming languages are generally “more pure” than imperative languages.

The core concepts of functional programming include

- first class and higher order functions,
- pure functions,
- that variables aren’t, i.e. they are immutable
- that multi-parameter functions can be mapped onto a chain of single parameter functions using currying,
- that recursion is used instead of iteration,
- lambda expressions,
- closures,
- referential transparency,
- lazy evaluation, and
- monads.

In functional programming languages functions are treated as first class language elements, i.e. they can appear in similar places to where variables can appear. Functions are thus declared, passed as parameters and returned as return values.

Furthermore, functions can operate on other functions resulting in higher-order functions.

Functions in a functional programming language are pure in that they produce no “side effects” – they receive an input and, based on that input, calculate some result. The result is not influenced by any external state. Neither is the function able to change any external state.

This characteristic implies that functions are repeatable. Given the same input, one will always get the same result. A function with its inputs can thus, at any stage, be replaced by its value without affecting the behaviour of the system. This property is called referential transparency. The advantages of referential transparency include

- that reasoning about the code is often substantially simplified,
- that the code can be more easily optimized, and
- that partial evaluations can be done by the compiler (often even at compile-time).

In pure functional programming there is no mutable programme state. It is a more mathematical approach which is declarative instead of imperative. There are no variables or objects whose state can be modified during the course of the programme execution. In fact, time itself plays a much smaller factor in functional programming than what it does in imperative programming. In other words, you cannot say $x = 7$ and later on contradict yourself and say $x = 9$.

However, in practice one often does need to change the state of the environment (e.g. change records in a database of change what id displayed on a screen). In functional programming such effects are usually isolated in what is commonly called monads. These monads behave pure functional from a user’s perspective with any mutable state only visible within the monad.

In functional programming there is no iteration (e.g. for or while loops). There would be no point to iteration since one cannot change the state of any variables or objects anyway.

Instead one uses recursion, i.e. define a function in terms of itself. Below is an example of a recursive implementation of a factorial function:
factorial \( n \) = if \( n == 0 \) then 1 else \( n \times \) factorial \( n - 1 \)

Lambdas are anonymous functions, i.e. functions which do not have a name and which are defined “in-line”. They are also called function literals.

For a \( \lambda \)-function one specifies the mapping of the function parameters onto a function body. For example

\[ x \rightarrow 2x^2 - 3x + 4 \]

\( \lambda \)-functions are a form of nested function. The “nested” \( \lambda \)-function has access to the variables of function providing the containing scope.

A closure is a function together with an enclosing scope, i.e. a function which is defined within an enclosing scope and which has access to the variables from the enclosing scope.

In Java a closure can be created using an inner class. The inner class does have access to the variables defined in the enclosing scope. For example

```java
public class MyLinkedList
{
    private Iterator iterator()
    {
        return new Iterator()
        {
            public void reset()
            {
                current = head;
            }
        };
    }
    private Node current;
    private Node head;
}
```

Map-Reduce is used to process large amounts of data in parallel with the computation typically being distributed across machines within a cluster or grid. The processing is typically done using

- **filters** to filter all irrelevant data,
- **maps** to perform some processing on individual data elements, and
- **reduce** to aggregate the individual mapping results into a result for the process as a whole.

For example, one could process a large client list, use filter to retain only those clients who have been with the company for less than 2 years, map out their ages and reduce the collection of customer ages into a single number representing the average age of those customers.

Functional programming languages have a particularly suited to map-reduce processing due to

- the suitability of pure functions for massive parallelization, and
- extensive built in filtering, mapping and aggregation functionality.

Functional Programming languages (e.g. Haskell) typically use lazy evaluation, i.e. the valuation of expressions is deferred until the value is required. For example, if you provide a lambda expression as argument to a function, that expression is not evaluated when the parameter is passed. Instead it is only evaluated when its value is required (which may be never).

There are a number of benefits around lazy evaluation including:

- Avoids unnecessary evaluations
  - some of which could yield errors
  - \( \rightarrow \) reduces errors
• Avoids repeated evaluation
  – value simply obtained from lookup table
• Also, since functions pure
  – valuation stored in lookup table
  – re-used instead of re-calculated
  – can create infinite collections

The following table highlights some key differences between functional and imperative programming:

<table>
<thead>
<tr>
<th>Functional</th>
<th>Imperative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate expressions</td>
<td>Statements which change state</td>
</tr>
<tr>
<td>Recursion</td>
<td>Iteration</td>
</tr>
<tr>
<td>Variables can be assigned once</td>
<td>Variables can change state</td>
</tr>
<tr>
<td>Referential integrity</td>
<td>Code behaviour depends on state of environment</td>
</tr>
</tbody>
</table>

### 1.1.2 Strengths and weaknesses of functional programming

The strengths of functional programming include that it is a simpler programming paradigm for concurrency, reuse and testing, that code is typically more compact. However, many problems are naturally modeled in a stateful way. Functional programming is not the most natural paradigm for such a domain. Furthermore, there is generally lower skills availability for functional programming and imperative programmers who would like to move to functional programming will need to make a significant paradigm shift.

Finally, functional programming is generally more difficult to trace and performance bottlenecks are generally more difficult to analyze.

### 1.2 Java support for functional programming

Java 8 onwards does provide a number of language features supporting functional programming including

• Lambda expressions
• Passing methods and static functions as parameters
• Functional interfaces
• For convenience, a set of standard interfaces and classes including
  – standard functional interfaces,
  – common operators
  – a predicate interface
  – standard interfaces for consumers and suppliers
• Filter-Map-Reduce functionality
• Concurrent, parallel & lazy processing via Streams API
Table 1: Java support for standard functional programming concepts.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>First class functions</td>
<td>yes</td>
</tr>
<tr>
<td>Pure functions without side effects</td>
<td>not guaranteed</td>
</tr>
<tr>
<td>Lambdas</td>
<td>yes</td>
</tr>
<tr>
<td>Higher order functions</td>
<td>yes</td>
</tr>
<tr>
<td>Lazy evaluation</td>
<td>partially via Streams API</td>
</tr>
<tr>
<td>Closures</td>
<td>yes</td>
</tr>
<tr>
<td>Referential integrity</td>
<td>no</td>
</tr>
<tr>
<td>Recursion</td>
<td>yes</td>
</tr>
</tbody>
</table>

1.3 Benefits of functional programming support

The benefits of the functional programming support in Java include

- More compact/cleaner syntax.
- Encourages functional way of thinking.
- More intuitive for many problems.
- Ability to easily/cleanly pass functions.
- Can cleanly pass references to methods.
- Simpler concurrency/parallelization

2 Functional Interfaces

You have most likely often used Single Abstract Method (SAM) interfaces like Runnable, Callable, etc. You commonly implement them with either a named or an anonymous class, instantiate them and pass the instance as a parameter to a constructor or method. The purpose of these interfaces is to provide a mechanism to pass a some block of code – a function. In Java 8, these interfaces have been annotated as FunctionalInterfaces.

Functional interfaces are thus interfaces with a single method representing a function. They can be used to

- create Lambda expressions, and
- for providing method parameters or methods as return values.

In addition to the single functional method, one can add

- any method from java.lang.Object in order to specify that these should be overridden, and
- default methods which are used to provide some limited support for multiple inheritance in Java.
2.1 Syntax

Functional interfaces are also called Single Abstract Method (SAM) interfaces. They are created by defining an interface with a single “function method” and preferably annotating that interface as a @FunctionalInterface. The function method may have any number of parameters including none and any return type including void:

```java
@FunctionalInterface
public interface Function {
    public double value(double x);
}
```

Adding any other method → compiler error, except if it is a method from java.lang.Object or a default method.

3 Lambda expressions

You will have used a range of Single Abstract Method (SAM) interfaces like, for example, Runnable and ActionListener. You would have provided implementations as either named or anonymous inner classes. The anonymous inner classes form closures with the instance members of the enclosing class. If you decided to use an anonymous inner class, you will have done so either to prevent reuse or to reduce the code bulk or both.

Lambda expressions are anonymous inner functions specified through a mapping of function parameters onto a function body. For example, the following lambda expression can be used to evaluate the length of a two-dimensional vector:

```java
(x, y) -> x*x + y*y
```

Lambda expressions have access to the instance member of the enclosing class, but not to the local variables of the enclosing method.

3.1 Syntax for Lambda expressions

Lambda expressions are anonymous functions which directly specify a mapping of its parameters onto a block of code representing the function body. The parameters are specified within round brackets followed by an arrow pointing to the function body. For example

```java
(type1 arg1, type2 arg2...) -> { body }
```

is a lambda expression on two parameters and a function body specified within curly brackets.

If parameters types can be inferred, they need not be specified:

```java
(arg1, arg2...) -> { body }
```

If the lambda expression has more than one parameter, the parameters are delimited by commas. If the lambda expression does not take a parameter, one uses empty round brackets:

```java
new Thread(() -> f()).start();
```

The body of lambda expression can contain zero or more statements. If there are no or multiple statements, you need to enclose the body within curly brackets.
As a further simplification, one can omit the round brackets for single parameter lambda expressions and the curly brackets if single statement functions:

\[ x \to \text{return } x \times x \]

### 3.2 Code reduction through Lambda expressions

Lambda expressions are typically very compact. Hence, using lambda expressions can result in a significant reduction in code bulk.

Consider, for example, the following code snippet:

```java
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        doSomething();
    }
});
```

The amount of code can be significantly reduced using a lambda expression:

```java
button.addActionListener(event -> x.doSomething());
```

### 3.3 Variable capture

Lambda expressions have access to the instance members of enclosing class as well as to final and effectively-final local variables of enclosing method, but not to true local variables of the enclosing method. For example, the following unit test

```java
public class VariableCaptureTest {
    @Test
    public void test1()
    {
        JButton b = new JButton();
        double nonFinalLocalVariable = 13;
        //++nonFinalLocalVariable;
        b.addActionListener(event -> System.out.println(instanceMember + nonFinalLocalVariable));
        b.doClick();
        }
    private double instanceMember = 3;
}
```

will compile and run if and only if the line containing `++nonFinalLocalVariable` is commented out. Otherwise the variable `nonFinalLocalVariable` is neither final nor effectively final and we will receive a compiler error.

### 3.4 Differences between lambda expressions and anonymous inner classes

On first sight lambda-expressions may look simply like syntactic sugar around anonymous inner classes. This is, however, not the case. There are some important differences between lambda expressions and anonymous inner classes.
Table 2: Some important differences between anonymous inner classes and lambda expressions.

<table>
<thead>
<tr>
<th>Annonymous Inner Class</th>
<th>Lambda Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>this</code> reference refers to instance of inner class.</td>
<td>The <code>this</code> reference refers to instance of enclosing class.</td>
</tr>
<tr>
<td>Compiled to separate class</td>
<td>Compiled to private method of outer class</td>
</tr>
<tr>
<td>Can have instance members</td>
<td>Cannot have instance members</td>
</tr>
<tr>
<td>Invoked through method table</td>
<td>Invoked via <code>invokedynamic</code></td>
</tr>
</tbody>
</table>

3.5 Some typical examples

- Convenient way of performing function on each element of a collection:

  ```java
collections.forEach(a -> System.out.println(a.getAccountId() + " + " + a.getBalance()));
```

- Specifying a comparator for a sorted collection

  ```java
  SortedSet<Person> persons = new TreeSet<Person>((p1, p2) -> p1.getDateOfBirth().compareTo(p2.getDateOfBirth()));
  ```

- Mathematical problems which have function parameters
  - Finding roots/solutions, minimum/maximum, integration/differentiation, ...

- Flexible processing pipelines

- Task definitions
  - Using lambda expressions to specify `Runnable` and `Callable` functions.

- Event handlers

- Query specifications
  - e.g. JPQL queries

4 Example: Simpson integration

This section shows an example which uses the Simpson integration method for calculating the definite integral of a provided function over a provided interval. The section demonstrates the use of functional interfaces, lambda expressions, and passing both method and static function references to other functions.

4.0.1 The Maven pom
4.0.2 The functional Function interface

package za.co.solms.training.java.lambdaExpressions.simpson;

/**
 * A functional interface for a simple 1-dimensional function R→R.
 */
```java
/**
 * Calculates and returns the function value at x.
 * @param x the value for which the function value is queried
 * @return the favue of f(x)
 */
public double value(double x);

public String toString(); // can overwrite methods of java.lang.Object
}

4.0.3 An integrator functional interface

```java
package za.co.solms.training.java.lambdaExpressions.simpson;

/**
 * A functional interface for an integrator which calculates
 * the definite integral accurate to an error bound of eps.
 * @author fritz@solms.co.za
 */
@FunctionalInterface
public interface Integrator
{
    /**
     * Calculates the definite integral of f from lower to upper bound.
     * @param f the function to be integrated
     * @param lowerBound the lower bound for the definite integral
     * @param upperBound the upper bound for the definite integral
     * @return the value of the definite integral
     */
    public double integrate(Function f, double lowerBound, double upperBound);
}

4.0.4 A Simpson integrator receiving a function as parameter

```java
package za.co.solms.training.java.lambdaExpressions.simpson;

/**
 * An implementation of the functional {Integrator} interface which uses the
 * Simpson method for calculating the approximate definite integral.
 * @author fritz@solms.co.za
 */
public class SimpsonIntegrator implements Integrator
{
    @Override
    public double integrate(Function f, double lowerBound, double upperBound)
    {
        double h = (upperBound - lowerBound) / N; // step size
        double sum = f.value(lowerBound) + f.value(upperBound);

        // Note: Should have rather been
        // double sum = f(lowerBound) + f(upperBound);

        for (int i=1; i<=N; i+=2)
            sum += 4 * f.value(lowerBound + i*h);

        for (int i=2; i<=N-1; i+=2)
            sum += 2 * f.value(lowerBound + i*h);

        return sum * h;
    }
}
```
4.0.5 Unit test providing function as parameter

```java
package za.co.solms.training.java.lambdaExpressions.simpson;
import static org.junit.Assert.*;
import org.junit.Test;
public class SimpsonIntegratorTest {
    /**
     * Integral for 3’rd degree polynomial must be exact
     */
    @Test
    public void testTraditional() {
        Function f = new Function() {
            public double value(double x) { return x*x + x; }
        };
        double integral = new SimpsonIntegrator().integrate(f, 0, 2);
        assertEquals(14.0/3.0, integral, 1e-12);
    }
    @Test
    public void testLambdaExpressions() {
        Function sine = x -> Math.sin(x);
        Function f1 = x -> sine.value(x*x) - x*x+x;
        double integral = new SimpsonIntegrator().integrate(
           x -> sine.value(x*x) - x*x+x, 1.5, 3.5);
        assertEquals(-8.45159789, integral, 1e-6);
    }
    class Parabola {
        public Parabola(double a, double b, double c) {
            this.a = a; this.b = b; this.c = c; }
        public double value(double x) { return a*x*x + b*x + c; }
        private double a, b, c;
    }
    @Test
    public void testMethodReference() {
        Parabola myPara = new Parabola(1,2,3);
        double integral = new SimpsonIntegrator().integrate(myPara::value, -2, 2);
        assertEquals(17.333333333333333,integral,1e-12);
    }
    @Test
    public void testStaticMethodReference() {
        double integral = new SimpsonIntegrator().integrate(Math::sin, 0, 2*Math.PI);
    }
}
```
5 Provided functional interfaces

The java.util.function package provides a range of functional interfaces removing the need to specify standard interfaces.

A functional interface should be implemented by a pure function, i.e. a block of code which receives an input and calculates as output, but which does not alter the environment (have any side-effects).

- One parameter functions: Function<T,R>
  - Functional method: apply(T arg)
- 2-parameter functions: BiFunction<T,U,R>
  - Functional Method: apply(T arg1, U arg2)
- Both have also default methods for function composition:
  - processing pipeline from this function followed by another function:
    ```java
    default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
    ```
  - processing pipeline from another function followed by this function followed
    ```java
    default <V> Function<V,R> compose(Function<? super V,? extends T> before)
    ```
- Also range of specializations, e.g. LongToDoubleFunction
- All can be used for Lambda expressions

6 Consumers

Even though Consumer is annotated as a FunctionalInterface, it should really be regarded as a pseudo-functional interface. The reason for this is that it represents a sink which receives a single input and has no return value. Since it does not return anything, its purpose can thus be only in the side-effects — it is thus not a pure function.

The Consumer interface does provide support for the chaining of consumers, i.e. to assemble a consumption pipeline. This is particularly useful as consumers are generally used in the context of constructing processing pipelines — often these processing pipelines are stream-processing pipelines. To this end the the interface provides a default andThen method:
```java
/** Performs this operation on the given argument. */
void accept(T t);

/** Returns a composed Consumer that performs, in sequence, this
 * operation followed by the after operation. */
default Consumer<T> andThen(Consumer<? super T> after)
{
    Objects.requireNonNull(after);
    return (T t) -> { accept(t); after.accept(t); }
}
```

The java.util.function package also provides a number of convenience consumers like BiConsumer which consumes two parameters, DoubleConsumer, LongConsumer, and so on.

## 7 Suppliers

Supplier is a pseudo-functional interface. It has no input, but produces an output, i.e. data objects. Suppliers thus represent a data sources or data providers.

A Supplier may or may not have side-effects. For example, suppliers which provide data from databases, files or lists commonly do not have any side effects. On the other hand, suppliers which source objects from queues or streams generally remove objects from those streams. These do have side-effects.

## 8 Operators

Operators are special types of functions which receive the operands as inputs, perform an operation on the operands to yield a result and return the result. Often, though not necessarily, the operands and the result are of the same type. Functions for which the parameters and return value are of the same type are called endomorphisms in functional programming.

Operators should be pure functions which do not alter the inputs or the environment and which solely yield a calculated result.

The java.util.function package provides UnaryOperator binaryOperator as well as some convenience specializations like, for example, DoubleUnaryOperator and DoubleBinaryOperator. UnaryOperator is a specialization of R Function<T> and R BiFunction<T,U> respectively. The operand function is inherited from the Function and BiFunction interfaces:

```
R apply(T, U)
```

or often

```
T apply(T, T)
```

## 9 Predicate

Predicates are boolean-valued functions which receive a single parameter. They are used to assess whether some expression is true. Since Predicate is annotated as a FunctionalInterface, predicates can be used to specify lambda expressions method or static function references. The
Predicate's function method is boolean test(T t) which is used to test whether parameter evaluates to true. The following example illustrates the usefulness of predicates:

```java
class Example {
    public int sumConditional(List<Integer> numbers, Predicate<Integer> p) {
        int sum = 0;
        numbers.forEach(value -> {
            if (p.test(value)) {
                sum += value;
            }
        });
        return sum;
    }

    sumConditional(numbers, n -> true); // summing all elements
    sumConditional(numbers, n -> n % 2 == 0); // summing even numbers
    sumAll(numbers, n -> n > 3); // summing all elements with value > 3
}
```

Composite predicates representing higher-level logic assembled from lower level logical components can be assembled using default methods for logical operands provided by Predicate:

- Logical AND:

  ```java
  default Predicate<T> and(Predicate<? super T> other)
  ```

- Logical OR:

  ```java
  default Predicate<T> or(Predicate<? super T> other)
  ```

- Logical NOT:

  ```java
  default Predicate<T> negate()
  ```

- Logical ==:

  ```java
  static <T> Predicate<T> isEqual(Object targetRef)
  ```

In this example we use predicates, mapping functions and consumers to construct a generic processing pipeline and use that processing pipeline to process a collection of marks in order to send emails to those students who are requested to write a re-exam.

## 10 The generic processing pipeline

In this example we build a generic processing pipeline which received a

1. a source collection,
2. predicate which is used to filter out selected source elements from a collection,
3. a mapping function which maps source elements onto elements which should be processed, and
4. and a processing consumer which is a processing sink used to process the elements to be processed.
The predicate, mapping function and processing consumer are all typically provided as Lambda expressions. The generic processing pipeline is provided within the `SourceFilterMapProcessPipeline` class:

```java
public class SourceFilterMapProcessPipeline {
    public <SourceType, ConsumableType> void process (Iterable<SourceType> source, Predicate<SourceType> tester, Function<SourceType, ConsumableType> mapper, Consumer<ConsumableType> consumer) {
        for (SourceType sourceElement: source) {
            if (tester.test(sourceElement)) {
                ConsumableType consumableElement = mapper.apply(sourceElement);
                consumer.accept(consumableElement);
            }
        }
    }
}
```

11 Sending re-exam notification emails

We illustrate the generic processing pipeline using student marks. To this end we introduce a simple `Person` and a `Mark` class as illustrated in Figure 1.
11.0.6 Person.java

```java
package za.co.solms.training.java.functionalProgramming.marks;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;

/**
 * A simple person class
 * @author fritz@solms.co.za
 */
public class Person {
    public Person(String name, String emailAddress){
        this.name = name;
        this.emailAddress = emailAddress;
    }
    public String getNome() {
        return name;
    }
    public void setNome(String name) {
        this.name = name;
    }
    private String name;
    private String emailAddress;
}
```

11.0.7 Mark.java

```java
/**
 * A student's mark
 * @author fritz@solms.co.za
 */
public class Mark {
    public Mark(int mark, Person student) {
        this.mark = mark;
        this.student = student;
    }
    public Person getStudent() {return student;}
    public void setStudent(Person student) {this.student = student;}
    public int getMark() {return mark;}
    public void setMark(int mark) {this.mark = mark;}
    private int mark;
    private Person student;
}
```

11.1 MarksUtilities.java

Now we introduce a utility class which uses the generic processing pipeline to
1. filter all marks within the range: \([40, 50]\).

2. find for those marks the email address of the associated student, and

3. dispatch an email to the student to notify the student that he/she needs to attend the re-exam.

```java
/**
 * A utilities class making use of the generic (@link SourceFilterMapProcessPipeline) for a variety of services
 * @author fritz@solms.co.za
 */
public class MarksUtilities {
    public static void sendReExamNotificationEmails(List<Mark> marks, List<String> names) {
        new SourceFilterMapProcessPipeline().process(
            marks, // the source
            x -> x.getMark() < 50 && x.getMark() >= 40, // a predicate
            m -> m.getStudent().getEmailAddress(), // a mapper
            email -> sendReExamNotificationEmail(email)); // a consumer
    }
}
```

Finally we use the `MarksUtilities` class in a unit test with some dummy data:

```java
/**
 * @author fritz@solms.co.za
 */
public class MarksUtilitiesTest {
    /**
     * @throws java.lang.Exception
     */
    @Before
    /**
     * @throws java.lang.Exception
     */
    public void sendReExamNotificationEmail(String email) {
        System.out.println("Busy sending re-exam notification email to " + email);
    }
}
```
public void setUp() throws Exception {
    int numStudents = 100;
    for (int i=0; i<numStudents; ++i)
        marks.add(new Mark(randomNumberGenerator.nextInt(100),
            new Person("Lucia" + i, "lucia"+i + "@mailProvider.org")));}

/**
 * @throws java.lang.Exception
 */
@After
tearDown() throws Exception {
    marks.clear();
}

@Test
testSendReExamNotificationEmails() {
    MarksUtilities.sendReExamNotificationEmails(marks);
}

private List<Mark> marks = new LinkedList<Mark>();

private static Random randomNumberGenerator = new Random();

12 Best practices

- Lambda expression should be a pure function
- Should not be for global reuse (prevent reuse)

13 Functional Programming Exercises

1. The root of a function is the value of \(x\) for which \(f(x) = 0\). In this project you should develop an infrastructure for 1-dimensional root solvers and an implementation of a concrete root solver which uses the bisection method for finding a root for a function. The method receives a range over which \(f\) changes sign (containing an odd number of roots) and the required accuracy, \(\varepsilon\), as inputs. The interval is halved, and the sub-interval over which the \(f\) changes sign is selected as new interval. This is repeated until the function value at either of the boundaries is less than the required accuracy, \(\varepsilon\). In particular

   a) Write a functional interface for a 1-dimensional function.
   b) Write a functional interface for a RootSolver which receives a function, an interval over which the function changes sign and the required accuracy, \(\varepsilon\), as inputs.
   c) Write an implementation of RootSolver which uses the bisection method to solve for the root of a given function.
   d) Write a unit test which uses lambda expressions for functions and verifies that the BiSectionRootSolver correctly calculates the roots of those functions. Use

   (i) \(f(x) = 2x - 1\), range = \([0, 3]\), solution: \(x = \frac{1}{2}\)
   (ii) \(f(x) = x^2 + 2x - 3\), range = \([-4, -1]\), solution: \(x = -3\)