JAVA FOR THE STUDY OF EVOLUTION

VOLUME XVI

FUNCTIONAL JAVA, EASY VS. COMPLEX PROBLEMS

José del Carmen Rodríguez Santamaría
The EvolJava Community

Contents

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than calculus alone.

This is the 16th volume: What do human beings do to develop software? They use a sort of enriched GP (Genetic Programming), an important part of which is the very programming language. So, lambda expressions were introduced in Java 8(2015) to ease the task of designing evolvable software that shall be easy to adapt to new functions and new data structures. This is what lambdas is all about. We review them together with the style of programming that they generate. The new style is illustrated with some selected programs that show that problems can be easy else difficult for Evolution.

Introduction

It is clearly known how to program computers to autonomously develop software with the help of evolution. The discipline is known as GP (Genetic Programming) but the crude reality is that GP cannot compete with human developers. The amusing thing is these use an enriched GP as the main strategy to develop computer programs. In fact, most people, the Author included, base their work on the copy&paste operations, retrieving pieces of code from suitable web sites. We all know what those pieces do but in general we do not know neither how they work nor what is behind its design. Nevertheless, we can modify them to serve local purposes. So, what we do is human genetic programming, a human application of evolution to software development. But it is not Darwinian because every developer has a powerful intuition and a wide background with whose help reuse becomes successful. This means that after fixing many bugs whose amendment creates more bugs, one finally gets what one wants.

So much suffering with human GP has led to the belief that programming languages play an important role in the quality and quantity of produced software by human beings. This is obvious if one compares Java with Assembler, but for modern high level languages that belief is not that clear because modernity has come accompanied with high quality in all fronts due to a long evolutionary race. Anyway, all extant languages continue to evolve to the better and all seem to strive to do their best against a terrible law: Evolve or perish. Evolve to be the best, the most loved, otherwise you are in risk of declining to then disappear. At the same time many new languages are continuously proposed. And some claim that they represent the natural, needed and futurist development of the Java evolutionary branch. Renown examples: .NET, Kotlin, Scala, Groovy. Also: Java runs over a virtual computer known as the JVM (Java Virtual Machine) and, as a real computer, it can support whatever language. So, the JVM hostes many languages: Clojure (a version of Lisp, functional), Groovy(Functional), Scala (Extremely functional), JRuby (an implementation of Ruby), Jython (an implementation of Python).

So, in regard with Java a major and successful evolutionary breakthrough occurred in Java 8 in which lambda expressions were introduced. We already have

employed them as simplifiers in JavaFX listeners and lists but there are many more things to them. In particular, Lambda expressions enclose a new style of programming that is referred to as **functional programming**. In this paradigm the *what to do* is visible while the *how to do it* is hidden under suitable interfaces. By contrast, ordinary Java code is always overload with detailed instructions to do things. Example:

Suppose we want to remove prime numbers from a list of numbers. The following code does the task:

```
Iterator<Integer> iterator = numbers.iterator();

        while (iterator.hasNext()) {

            Integer next = iterator.next();

            if (!isPrime(next)) {

                iterator.remove();

            }

        }
```

What is the problem? It is too lengthy because when a number is removed, a void is created that is filled in by shifting elements one place. Now, Java 8 invites to do the task in the following simple way:

```
numbers.removeIf(integer -> !isPrime(integer));
```

What is the virtue? The code is clear and without technical garbage, which is left to the implementation. This, in its turn, has been done by experts that have produced very efficient code (Rahman, [14] 2016). In general, implementations are hidden so, they can be perfected every day and the developer is not overloaded with technicalities.

As this example, we present here some ready to use code that can be used to learn and explore the fundamentals of lambdas and related features including its accompanying style.

In regard with concepts, selected programs remind us that there are easy and complex problems for evolution.

Most exercises are addressed to invite the Reader to learn from Internet. Java is extremely vast and the new flavor that we are exploring here, functional programming, is also a giant world. That is why permanent study is the only choice in our community.

No program is included in this text. Instead an accompanying NetBeans project can be downloaded, imported and printed. To list the content of all Java files of a NetBeans project use program `Q65 PrintSourceFiles`. Java files can be printed to the console of NetBeans from where it can be copied&pasted into any document. Alternatively, files can be saved to a .txt file that can be opened with any word processor.

Knowledge with context and code about Java 8 can be found in (Venkat, [23] 2014)

JavaFX is assumed.

Bogotá, Colombia, *José Rodríguez*
Nov 2016

Chapter 1

Overture

`Highly functional evolvable code`

1 ***Introduction and purpose.*** *Lambda expressions are introduced as a majestic solution to write simple code with a lot of functionality. They generate a new style of programming and one of its ingredients is unveiled: the strict separation of data from functions together with their mutual adequacy and fair concordance with the purpose in mind. Evolvability of produced software is expected to be propelled.*

1.1 Liners

Liners or **one-liners** are pieces of code one line long that may create an object, process and consume it. In Java, a line ends with a semicolon but for sake of readability it could be displayed in various rows. Liners became possible in Java after the introduction of **Lambda expressions** that are distinguished by an arrow $->$. It is almost immediate to understand how they function.

2 ***Example. The following is a prototype of liner. It is a program that instantiates a list with 3 elements of type String and then commands to write then, one by one, to the console. Let us notice that the created list has no name, it is anonymous so, it cannot be reused, it is consumed right after creation and processing.***

```
Arrays.asList( "a", "b", "d" )
      .forEach( e -> System.out.println( e ) );
```

3 ***Example. Program*** `Q3 Liners` ***shows some elementary liners.***

4 ***Exercise.*** *Run the previous program and play with the code. Play enough to disentangle the function of each liner. Give to each liner a title in accordance with its function. Annotate it in front of the number of each liner. You can compare you answer with ours in program* `Q4 Liners2`*.*

5 ***n-liners.*** *For sake of cosiness, one might prefer to divide a liner in two or more lines. Some examples are in program* `Q5 Liners3`

6 ***Exercise.*** *Run the previous program and play with the code.*

7 ***Exercise.*** *The previous code works for integers. Adapt it to strings. Agree else disagree with what everyone preaches: lambda expressions are evolvable, easy to adapt to new situations. You can see our answer in program* `Q7 Liners4`*.*

1.2 Functions

Functions implement associations of individual elements in a set to individual elements in another set.

8 ***Example.*** *The following snippet implements a function that multiplies listed numbers by two:*

```
Arrays.asList( 1, 11, 7, 4)
               .stream()
               .map(x -> 2*x)
               .forEach(y -> System.out.println(y));
```

Indeed, this liner contains two functions, the first is contained in the method map and is the function properly and the second commands to print the output of the doubling function. Let us observe that neither of these functions has a declared name. This is so because they are intended to never be used again so, they are anonymous.

9 ***Anonymous functions and classes.***

Let us consider the following liner

```
Arrays.asList( 1,2,3 )
            .forEach( (Integer e ) -> System.out.println( e ) );
```

This liner assigns a printing action to each integer in the domain. So, it is a function. But it has no name: it is an **anonymous function**. This contrasts with the methods in Java which also encode functions but generally have a name. Named functions and methods are used for reuse. Here, in **functional programming**, functions are created just in need and immediately are consumed after being used. They exist no more. So, anonymous functions are volatile.

Here we have a name that can be called on for reuse, very popular in ordinary Java:

```
 Cosumer<String c = new Consumer<String>()
```

By contrast, we have here an **Anonymous class** that has no name:

```
 friends.forEach(new Consumer<String>() {
public void accept(final String name) {
System.out.println(name);
}
});
```

If the instantiation of a class is not for reuse but for just instantaneous use and further disappearing, it needs no name, it is anonymous, it is an **anonymous class**.

10 *The trap behind duplications*

Anonymous functions must be cloned if they are needed in another part of the program. This leads to duplication of code. Because that situation results from the natural unfolding of the programming activity, one does not worry to much about it. Nevertheless, duplication leads to non evolvable software. This means that when one wants to make changes, if one piece is touched, then all cloned pieces must be modified accordingly. The price to adapt the code to new situation is multiplied by the number of copies that each piece has. Which is the remedy?

11 *Functions for reuse.*

In Functional Java a **Function** may be very simple. The following function adds five to an input number. It is called with the method apply():

```
Function <Integer, Integer> addFive = i -> i + 5;
System.out.println("addFive.apply(7) = " + addFive.apply(7));
```

In this example, Function is a reserved word of Java that implies the use of a Java interface that is not visible. If one does not use the inbuilt interface function, one must define an interface as follows:

```
//Input = Integer,
//output = Integer,
//name = square,
//calculation: (a) -> a * a


Operation<Integer, Integer> square = (a) -> a * a;
```

Nevertheless, to be executed, Java needs a Functional Interface, which a special type of method that abstracts the notion of function: the correspondence between two elements, the input and the output, created by an inner calculation:

```
//A functional interface accepts an input I
//makes a calculation
//and returns an output.
//The calculation that produces the output
//is not implemented, only declared.
//It is implemented in the declaration of each function
@FunctionalInterface
public interface Operation<I, O>
{
    //An output is produced after
    //making a calculus over an input
    O calculate(I input);
}
```

The functional interface enables the Java compiler to understand a calling on the function name:

```
//Execution
int squareOfThree = square.calculate(3);
```

The interface seems to be redundant. Nevertheless, it is the very same for all functions that work on the same scheme. And the number of schemes is infinite so, one must define exactly the needed one.

12 ***Example.*** *An implementation of various callable functions with lambdas can be seen in program* `Q12 FunctionalInterfaces`*. All functions work on the same interface because they all have the very same scheme.*

13 ***Exercise.*** *Run the previous program and play with the code.*

14 ***Exercise.*** *Study some generalizations of functions presented by Ufimtsev ([21], 2015) and use them to make a demo. Our answer can be seen in program* ***Q14 FunctionalInterfaces2****.*

15 ***Exercise.*** *Build an example of a not void method that is made into a lambda expression. Since it is not void so, it encodes for a function. Our answer is in program* ***Q15 FunctionalInterfaces3****.*

16 ***Exercise.*** *Study the code by tutorialspoint ([19], 2016) on lambda expressions. Make your own demo. Our answer is in program* ***Q16 Java8Tester****.*

One can define a Java map to declare an assigning through a table. Maps can be eventually combined with lambda expressions.

17 ***Example. How to employ lambda expressions to deal with maps is illustrated in the program*** **`Q17 MapExample`***.*

18 ***Composition and currying of functions*** *. Chaining of functions is called* ***composition****. When functions have various arguments, the process of creating new functions by keeping some argument constant is called* ***currying****.*

19 ***Example.*** *If we have $f(x) = 3x+5$ and $g(x) = 7x$ then $(f \circ g)(x) = f(g(x)) = 3(7x)+5 = 21x+5$. This is composition. On the other hand, if we have $h(x,y) = x+2y$, one can consider what happens if one keeps $x = 11$. The result is $j(y) = 11+2y$. If $x = 23$, we get $l(y) = 23+2y$. In general, we have the function T that to each t associates the function $T_t(y) = t+2y$. This is currying.*

20 ***Exercise.*** *Study the code written by (timyates, [18] 2013) that compares currying and composition. Make you own demo. Our answer is in program* ***Q20 Currying****.*

21 ***Multi-arguments and currying.***

If one gets a function on three variables, as the following
$(a,b,c) -> d = something$
one always can replaced it by currying:
$a -> b -> c -> d = something$

22 ***Exercise.*** *Study the code written by (Fusco, [5] 2016) that compares currying and composition. Make a demo. Our answer is in program* ***Q22 CurryingInterface****.*

23 ***Challenge.*** *In the industry higher order functions (functions that associate a function to a set of functions) may come accompanied with threads. Study the theme following (Peralta, [13] 2016) and make your own demo.*

1.3 The package `Function` in Java 8

We have used the interface `Function` of the package `Function` and in the next section we will learn about the interface `Predicates` in the same package. As always, we are making good use of some 5% of the possibilities of Java. For instance and in regard with the aforementioned package, Java has all the following ready to use interfaces (Oracle, [12] 2016):

```
BiConsumer<T,U>
Represents an operation that accepts two input arguments
and returns no result.

BiFunction<T,U,R>
Represents a function that accepts two arguments
and produces a result.

BinaryOperator<T>
Represents an operation upon two operands of the same type,
producing a result of the same type as
the operands.

BiPredicate<T,U>
Represents a predicate (boolean-valued function) of two arguments.

BooleanSupplier
```

```
Represents a supplier of boolean-valued results.

Consumer<T>
Represents an operation that accepts a single
input argument and returns no result.

DoubleBinaryOperator
Represents an operation upon two double-valued operands
and producing a double-valued result.

DoubleConsumer
Represents an operation that accepts a single double-valued
argument and returns no result.

DoubleFunction<R>
Represents a function that accepts a double-valued argument
and produces a result.

DoublePredicate
Represents a predicate (boolean-valued function) of one
double-valued argument.

DoubleSupplier
Represents a supplier of double-valued results.

DoubleToIntFunction
Represents a function that accepts a double-valued argument
and produces an int-valued result.

DoubleToLongFunction
Represents a function that accepts a double-valued argument
and produces a long-valued result.

DoubleUnaryOperator
Represents an operation on a single double-valued operand
that produces a double-valued result.

Function<T,R>
Represents a function that accepts one argument and produces a result.
```

```
IntBinaryOperator
Represents an operation upon two int-valued operands
and producing an int-valued result.

IntConsumer
Represents an operation that accepts a single int-valued
argument and returns no result.

IntFunction<R>
Represents a function that accepts an int-valued argument
and produces a result.

IntPredicate
Represents a predicate (boolean-valued function) of
one int-valued argument.

IntSupplier
Represents a supplier of int-valued results.

IntToDoubleFunction
Represents a function that accepts an int-valued argument
produces a double-valued result.

IntToLongFunction
Represents a function that accepts an int-valued argument
and produces a long-valued result.

IntUnaryOperator
Represents an operation on a single int-valued operand
that produces an int-valued result.

LongBinaryOperator
Represents an operation upon two long-valued operands
and producing a long-valued result.

LongConsumer
Represents an operation that accepts a single long-valued
argument and returns no result.

LongFunction<R>
```

```
Represents a function that accepts a long-valued argument
and produces a result.

LongPredicate
Represents a predicate (boolean-valued function) of
one long-valued argument.

LongSupplier
Represents a supplier of long-valued results.

LongToDoubleFunction
Represents a function that accepts a long-valued argument
and produces a double-valued result.

LongToIntFunction
Represents a function that accepts a long-valued argument
and produces an int-valued result.

LongUnaryOperator
Represents an operation on a single long-valued operand that
produces a long-valued result.

ObjDoubleConsumer<T>
Represents an operation that accepts an object-valued and a
double-valued argument, and returns no
result.

ObjIntConsumer<T>
Represents an operation that accepts an object-valued and a
int-valued argument, and returns no
result.

ObjLongConsumer<T>
Represents an operation that accepts an object-valued and a
long-valued argument, and returns no
result.

Predicate<T>
Represents a predicate (boolean-valued function) of one argument.
```

```
Supplier<T>
Represents a supplier of results.

ToDoubleBiFunction<T,U>
Represents a function that accepts two arguments
and produces a double-valued result.

ToDoubleFunction<T>
Represents a function that produces a double-valued result.

ToIntBiFunction<T,U>
Represents a function that accepts two arguments
and produces an int-valued result.

ToIntFunction<T>
Represents a function that produces an int-valued result.

ToLongBiFunction<T,U>
Represents a function that accepts two arguments
and produces a long-valued result.

ToLongFunction<T>
Represents a function that produces a long-valued result.

UnaryOperator<T>
Represents an operation on a single operand
that produces a result of the same type as its operand.
```

1.4 Predicates

We saw that functions are used to associate elements in a list to elements in another list. Let us see now predicates that associate sublists to lists.

24 ***Where predicates come from.*** *When we filter a set by a property or* ***predicate****, some elements pass while others not. Those that pass can be collected into a set that results to be a subset of the former one. So, predicates associates subsets to sets or, more formally, sublists to lists.*

25 ***Example.*** *In the following liner we encode the predicate* ***p is less than 8*** *and then filter the set with elements 7, 5 and 9. The output is the set composed by the*

elements 7 and 5.

```
Arrays.asList(7,5,9)
                .stream()
                .filter(p -> p < 8)
                .forEach(System.out::println);
```

26 ***Example.*** *We can define our own predicates to achieve generality. Say, to define a general predicate of the sort* `p is less than k`*, where* `k` *can be any Integer we proceed as shown in* `Program Q26 MyPredicate.`

27 ***Clean predicates***

Your code is a window to the joy of programming. This is what everyone desires. But reality is otherwise: programming is extremely difficult and so, one is always correcting bugs that create more bugs and in devising new adaptations that create also more bugs. The net result is a mesh. That is why software seems to be not evolvable by its very nature. Now, this realistic view is technically poor. Instead, there is software that is more evolvable than other and discriminators have been unveiled, although the list may vary from person to person. For Kevin Chabot ([1], 2013), the properties that discriminate evolvable from not evolvable software are:

- DRY (don't repeat yourself): writing code more than once is not a good fit for a lazy developer. It also makes your software more difficult to maintain because it becomes harder to make your business logic consistent.
- Readability: following clean-code best practices, 80% of writing code is reading the code that already exists. Having complicated lambda expressions is still a bit hard to read compared to a simple one-line statement.
- Testability: your business logic needs to be well-tested. It is advised to unit-test your complex predicates. And that is just much easier to do when you separate your business predicate from your operational code.

We can see what he prefers by studying the code that he provides:

28 ***Exercise.*** *Study the guidelines given by the cited document and develop a demo. You can choose to write a single program with various inner classes or three files for three distinct classes. You can compare you work with ours as follows: a file with everything can be found in program* `Lambdas Q28a`

`CleanPredicates`. The solution in three files (as experts would prefer) contains the programs `Q28b Employee`, `Q28c EmployeePredicates` and `Q28d EmployeePredicatesTest`, a class with the main() method.

29 ***Exercise and challenge.*** *Run your demo and and play with the code. Study it and decide whether or not this setting complies with the conditions of being dry, readable, and testable. Do you consider that this style is evolvable? Does it admit easy modifications? Think of adding a new employee, or defining a new field, or adding a new predicate, or devising a new test.*

1.5 The dot technology

Let us learn a very elegant way of chaining methods that modify the same object. It is the **dot technology**, which means that to add a call to a method one must print a dot and then insert the call.

30 ***Example.*** *We have various methods that mutate a string. All of them can be chained one after another in whatever order. Program `Q30 DotTechnology` shows how to do this along the dot technology:*

31 ***Exercise.*** *Run the previous program and play with the code. Verify that one can chain as many calls to whatever method in whatever order.*

32 ***Research.*** *NetBeans is an intelligent IDE (Integrated Development Environment). Its intelligence covers the dot technology because when one prints the dot with the intention of inserting a call to a method, NetBeans displays a pop up list of all methods that can be called, some from Java itself and the rest from own programming. To inquire how this is made possible, begin with* ***Java reflection****.*

33 ***Security issue.***

Intelligent persons solve problems, wise ones prevent their appearing. This is truth for complex affairs and henceforth for software development, too. Modern Java actively accepts this by offering the possibility of **functional programming**. Its visible head is conformed by lambda expressions while the blood that permeates the whole body is security. That is why lambda expressions accepts only final variables and everything inside shall be volatile and pop up when needed to then disappear. So, Lambda expressions are consumers by excellence. The theme is referred to as **immutability** (Oracle, [11], 2016).

34 ***Example.*** *The following two snippets by (Davis, [2] 2016) contrast two forms of solving the same problem, the first involves a mutable variable that is updated on every step, the second is written in declarative form and is immutable because nothing is changed by the developer.*

The following piece of code is mutable because it changes the value of `myCount[0]`:

```
int[] myCount = new int[1];
list.forEach(dragon -> {
     if (dragon.isGreen()) myCount[0]++;
 }
```

The following piece of code is declarative, immutable:

```
list.stream().filter(Dragon::isGreen).count();
```

Now, in the declarative code a what to do is proposed but not the how, which is left to well tested methods that are not under the scope of the developer and that cause no side effects, no type of interference. So, the developer has his or her mind free to focus in the overall plan and style.

The issue of mutability touches everything. To illustrate this, let us discuss an innocent detail of the previous program:

35 ***Example.*** *The previous program contains the class DotTechnology. It is instantiated in the main with the instruction* `new DotTechnology()`*. This instantiation can eventually modify the whole world, or can accept an overriding and so, it can create a mesh. It violates the design conception of immutability. The program* `Q35 DotTechnology2` *shows how this trouble is solved.*

36 ***Exercise.*** *Modify previous program to include arguments in the methods. Our answer is in program* `Q36 DotTechnology3`*.*

37 ***Challenge.*** *In previous programs we can found a global variable. It was made global to be modifiable by all members of a set of methods. So, by its very construction, it is against the directive of immutability. Invent something and/or make a research to see what alternatives are to global variables. This will become necessary if one pursues very complex tasks with multi-threading, web services and the like.*

1.6 References

In modern Java, after version 8, there is no difference among data, functions or methods in regard with referencing. So, anyone of them can be passed as argument in a method. The formalism is, nevertheless, verbose and convolute. So, pay attention.

38 ***Exercise: Reference To An Instance Method Of A Particular Object.*** *Study the code by (IDR Solutions [7], 2015) that shows how to use a Reference to an instance method of Of A Particular Object. Make a demo. Our answer is in program* `Q38 ReferenceToInstanceMethodOAPO.`

39 ***Exercise: a call to an Instance Method Of An Arbitrary Object Of A Particular Type.*** *Study the code by (IDR Solutions l.c., 2015) that shows how to use a Reference to an instance method of a particular class. Make a demo. Our answer is in program* `Q39 ReferenceToInstanceMethodAOPT.`

40 ***Exercise: a call to a list of methods.*** *Study the code by (user2572526, [22] 2015) that shows how to use a Reference to a list of methods. Make a demo. Our answer is in program* `Q40 ReferenceToListOfMethods.`

41 ***Exercise: a reference to a predicate.*** *Study the code by (IDR Solutions l.c., 2015) that shows how to use a Reference to a predicate. Make a demo. Our answer is in program* `Q41 ReferenceToPredicate.`

42 ***Exercise: a call to a function.*** *Study the code by (IDR Solutions l.c., 2015) that shows how to use a reference to a function. Make a demo. Our answer is in program* `Q42 ReferenceToFunction.`

43 ***Exercise: composing function.*** *Study the code by (StudyTrails, [17] 2014) that shows how to compose functions. Make a demo. Our answer is in program* `Program Q43 ReferenceAndFunctionComposition.`

44 ***Exercise: a class as argument.*** *Study the code by (Maroof, [9] 2016) that shows how to compose functions. Make a demo. Our answer is in program* `Q44 ReferenceToAClass.`

45 ***Exercise: ReferenceToSuperClass.*** *Study the code by (Friesen, [4] 2015) that shows a reference to a method belonging in a superclass. Make a demo. Our answer is in program* `Q45 ReferenceToSuperClass.`

46 ***Exercise: an interface demo.*** *Study the code by (Friesen, l.c., 2015) that shows an example of an interface in his Drivable Interface. Make a demo. Our answer is in program* `Program Q46 InterfaceDemo.`

47 ***Exercise: an interface with default method.*** *Study the code by (Friesen, l.c., 2015) that shows that an interface can have a default method. Make a demo. Our answer is in program* `Q47 InterfaceWithDefaultMethod.`

48 ***Exercise: an interface with a default method that is overridden.*** *Study the code by (Friesen, l.c., 2015) that shows that the default method of an interface can be overridden. Make a demo. Our answer is in program* `Q48 InterfaceWithOverride.`

49 ***Exercise: an interface with a default and helper methods.*** *Study the code by (tutorialspoint, [20] 2016) that shows that an interface can have helper methods apart from the default one. Make a demo. Our answer is in program* `Q49 InterfacWithHelperMethods.`

50 ***Exercise: an interface as argument.*** *Study the code by (Friesen, l.c., 2015) that shows how to pass an interface as argument of a method. Make a demo. Our answer is in program* `Q50 ReferenceToInterface.`

1.7 Tail-call recursion

Recursive functions are a source of eternal problems.

51 ***Definition.*** *A* ***function is recursive*** *when it uses self-reference in its definition.*

52 ***Example.*** *The factorial is recursive:* $n! = n(n-1)!$ *and* $0! = 1!$. *So,* $4! = 4 \times 3! = 4 \times 3 \times 2! = 4 \times 3 \times 2 \times 1! = 4 \times 3 \times 2 \times 1 \times 0! = 4 \times 3 \times 2 \times 1 \times 1 = 24$

Java 8 offers the possibility to attack recursive problems with a new tool that is called **tail-call recursion** or **tail-call Optimization**.

53 ***Exercise.*** *Develop a program to calculate factorials. Various answers are in program* `Q53 Factorials.`

54 ***Exercise.*** *Study the code by (Davis l.c., [2] section 10.4, 2016) that shows how to use the* ***tail-call recursion*** *technology. Make a demo. Our answer is in program* `Q54 Factorials2.` *Play enough with the new tool to agree else disagree with the Author: there is no improvement in performance over all simple methods.*

55 ***Challenge: a new try.*** *Study the code by (Saumont, [16] 2014) that shows the sign of expertise: he tries this and that to see what does it better to deal with a recursive function. Make a demo and test it.*

1.8 Optionals

Let us see the modern way of facing possible null pointer exceptions, which are exceptions caused by a call to something that does not exists, say, a path that leads to the wrong file.

56 ***Exercise: optionals.*** *Study the code by (Gioiosa, [6] 2016) that shows how to deal with calls that possible cause null pointer exceptions. Make a demo. Our answer is in program* `Q56 OptionalDemo`*.*

1.9 Listeners

Listeners were used to be complicated. With lambda expressions they became really simple and extremely popular. De facto they are used everywhere since its inception in Java 8 in 2015.

57 ***Example of listeners with lambdas.*** *Our application* `Finch`*, which can be found in the Art area of our site, has many examples of using lambda expressions to implement listeners.*

1.10 Streams

We have used streams everywhere but rather blindly. A bit of context may help us.

58 ***Exercise.*** *Study the tutorial on streams by (Benjamin Winterberg, [24] 2014) and make a demo. Our answer is in program* `Q58 StreamsDemo`*.*

1.11 Lambda expressions with objects

How do lambdas react to the complexity of real life? To see it, let us confront lambda expressions with a class that represents a complex object.

59 ***Example. Program*** `Q59 People` ***shows lambda expressions in action to manage complex data structures.***

60 ***Exercise.*** *Run the previous program and play with the code. Study the code enough to agree else disagree with the claim of the Author: lambda expressions obey a clear pattern of design:* `[lambda expressions allow] a clear and elegant distinction between the what [should be done] and how [should it be done]`*. Is that important?*

61 ***Imperative vs. declarative styles of programming***

Imperative means: do this according to this procedure and then that according to that other. Declarative means: do this and then than (such as you know).

1.12 Conclusion

Lambda expressions were added to Java as a magnificent idea to continue in the first rankings in the battle for the supremacy among programming languages. We have reviewed some material to kill curiosity about them. We find them nice, interesting, compact, evolvable but sometimes convolute. Lambda expressions conform the vertebral spine of Functional Programming along a declarative style. A code that shows how to do things is called **imperative**. A code that shows what to do, what must be do is called **declarative**. Java 8 opens the possibility to use declarative programming inside the usual Object Oriented Programming. Everyone is very glad with this new option. OOP encapsulates into a single unity a group of variables that describes a real object together with their interrelation and methods to process such data. On the other hand, the **functional programming** as it is also called the declarative implementation, spins more around changes and workflows than on states of the treated objects.

Chapter 2

Applications

`Old problems in new tuxedo`

62 ***Purpose.*** *Some model problems that we have already solved with ordinary Java will be reformulated in terms of streams, lambda expressions and the like.*

2.1 Numerical integration

We know how to make a numeric approximation to an integral using ordinary Java. Functional programming add a more natural flavor to this methodology.

63 ***Exercise: Simpson's method.*** *The basic idea is to replace the area under a curve by a suitable set of rectangles or trapeziums. Study the code by (Moore, [10]) and make a demo. You can see our answer in program* `Q63 SimpsonMethodJava8`*. The important thing is that in Functional Java the natural mathematical setting can be transparently copied into a program. So, the ensemble looks very nice.*

2.2 Printing the Java files of a NetBeans project

Java evolves. This means that one develops a perfect program today but tomorrow a new version of the Java compiler declares that it is obsolete in some regard. So, software must maintained, updated to go with the developments of Java. Not to mention the modernization of style. To avoid fatiguing work, we continue to put programs in a separated file with the corresponding NetBeans project but we do not list programs here, in the text for the Reader. Nevertheless, it would be good to have the possibility to read programs outside NetBeans, in a text processor. Let us see how we can help ourselves.

64 ***Exercise.*** *Scan the web for ways of using Java 8 to list the names of files in a directory. Our answer can be seen in program* `Q64 ListFiles`.

65 ***Exercise.*** *Develop a Java 8 program to print the content of all Java files in the source directory of current NetBeans project into a file to be read with a word processor. Our answer can be seen in program* `Q65 PrintSourceFiles`.

2.3 A professional example

It is advisory to give a look at professional software that definitively is much more than good, modern code.

66 ***Challenge.*** *Prompt the web with the word* `Jenetics`*, and find a site that hosts an application that contains a genetic algorithm in Java 8. The User Guide is the right place to begin with. Then, one would like to see the application running. If you can do that over NetBeans, please, share your knowledge with us (jose@evoljava.com).*

There is no doubt that evolution is almighty in a universe without limitations of time or resources. But in our universe, it is must be remarked that each problem has a cost in development, time and memory. But we usually pay attention just to one indicator: the number of individuals checked for the appointed function. Let us see that problems can be classified according to their cost: there are easy problems that are immediately solved by evolution while others may deserve more patience and still there are others that demand more time than provided.

2.4 An easy problem

Evolution is today an ordinary tool to solve optimization problems. So, let us commit ourselves to the development in Functional Java of a genetic algorithm to solve a simple problem, one that is easy for evolution because it immediately outputs a perfect solution. The Dawkins Era was dominated by the tacit belief that all problems were easy for evolution . So, let us present in this section the epitome of easiness, a problem of retro-engineering, a classic by Dawkins ([3], 1986).

67 ***The problem.*** *A given phrase or a string must be matched. For each try you are given a distance to the target measured by the number of mismatches in relation with the target. Example: target = evolution; try = msrldftkv. Distance = 8 because with the exception of the char l all others do not match.*

We have too many options to incorporate functional methods into GA (Genetic algorithm). We will solve the problem in various steps. Let us begin with the dot technology to chain methods one after the another. It is very nice. A genetic algorithm is dressed in this new tuxedo.

68 ***Exercise.*** *Program* `Program D107 Shakespeare` *of Vol IV has been copied to program* `Q68a Shakespeare` *in this package. It contains some minor modifications and the fixing of a bug (the program erroneously reported the best individual per generation). This program uses evolution to match a phrase, a string. It shows that there are problems that are very easy for evolution. The program uses Swing and ordinary Java. In a first step, delete employed Swing threads and recast the UI into JavaFX. You can see our answer in program* `Q68b Shakespeare2`*. Imagine now that somebody proposes an extremely long random input string: provide the possibility to cancel the job using the JavaFX Service class. You can see our answer in program* `Q68c Shakespeare3`*. To end, try a phrase with symbols such as $, &, (,... and verify that the the program cannot match it because the program uses a restricted alphabet. Make the corresponding amendment to make the code stable against all possible symbols. Warning: if you include all possible symbols in your alphabet, most possibly the time for matching simple phrases will run away. So, be smart. Our answer is in program* `Q68d Shakespeare4.`

69 ***Exercise.*** *Rewrite the code along the dot technology. Warning: the* ***dot technology*** *demands key methods to be static and this is impossible inside an inner class. So, you must export the inner class to its own file. Many answers could exist. Ours can be seen in program* `Q69 Shakespeare5` *and its accompanying class* `Q69b EvolutionFunctional.`

Let us incorporate into our genetic algorithm the stream technology in reference to arrays. Let us do it step by step beginning with mutation.

70 ***Example. Mutation with streams can be found in program*** **`Q70 MutationWithStreams.`**

71 ***Exercise.*** *Run the previous program and play with the code.*

72 ***Exercise.*** *Following the example of mutation, do the same with recombination and reproduction. Our answers can be found in programs* `Q72a RecombinationWithStreams` *and* `Q72b ReproductionWithStreams`*.*

73 ***Exercise.*** *Use developed methods to compose the simplest GA to guess a phrase. Our answer can be found in program* `Q73 GAWithStreams`*.*

We already have seen how functional programming is incorporated into GAs to deal with arrays or lists with the help of streams. Let us get acquainted now with other facilities to deal with strings. The idea is that strings can be tokenized onto lists of chars. This eventually would allow us to simulate mutation and recombination of strings with the help of the stream technology.

74 ***Streaming strings.*** *Program* `Q74 FunctionalStrings` *presents some ideas of functional programming to deal with strings.*

75 ***Exercise.*** *Run the previous program and play with the code.*

76 ***Example. Mutation over a list of chars is shown in program*** **`Q76 MutationAsBifunction`.**

77 ***Exercise.*** *Run the previous program and play with the code.*

78 ***Exercise.*** *The previous program implements mutation over lists of chars but it lacks recombination. Develop it. Formulate the correct questions and help yourself with Internet.* Answer

79 ***Exercise.*** *Modify program* `Q73 GAWithStreams` *to incorporate mutation as a bifunction and recombination as a trifunction. Our answer can be seen in program* `Q79 GAWithStreams2`

80 ***Exercise.*** *Using streams remake program* `Q69 Shakespeare5` *together with its accompanying class* `Q68b EvolutionFunctional`. *Our answer can be seen in program* `Q80 Shakespeare6` *and its accompanying class* `Q80b EvolutionFunctional2`.

81 ***Exercise.*** *Use developed programs to test the belief of the Author: programs that divide problems in subproblems run faster than those that not. Therefore, program* `Q69 Shakespeare5` *that divides the goal of matching a phrase in a succession of problems of matching composing chars shall be faster than program* `Q80 Shakespeare6`.

82 ***Exercise.*** *Verify that when the evolutionary rate is low, circa 0.1, program* `Q80 Shakespeare6` *runs fast. But its performance gets lower if the mutation rate increases to, say, 0.5. Explain. Discuss the following argument: our results imply that the theory of evolution is congruent with the observed evolution of the fidelity of the biochemical machinery of DNA replication and DNA repairing: poorly implemented for bacteria, extremely delicate for higher organisms.* Answer

83 ***Conundrum.*** *A problem that can be divided in subproblems is in general easy for evolution. Now, how does evolution know that a problem is easy and can be divided in subgoals? The fact is that this anthropomorphism is ill formulated because evolution has no knowledge. Instead, a problem is easy for evolution when a high recombination rate allows the program to rapidly converge to the goal. Recombination is the natural tool of evolution to join together short partial solution to conform an enlarged solution that can be grown further to be a full solution. In such a case, achieved results do not cause stagnation. This happens when the solution to a subgoal does not negatively interferes with the solutions to other subgoals. In that way, subgoals can be achieved separately and in parallel and this is spontaneously done by recombination. That is why evolution is powerful. In hindsight, the solution can be grown as gathering of small change. This implies that a difficult problem for evolution is one in which as recombination as mutation shall be set to low values in order to not destroy achieved results. ¿Are there difficult problems for evolution?*

84 ***Research.*** *Functional languages are loved because of their elegance, security and evolvability. Nevertheless, they have been blamed for being too slow. Learn how to measure time in Java 8 to make appropriate measurements to determine whether or not this is certain for the functional Java vs. ordinary OOP Java. To that aim, you can compare the performance of program* `Q67d Shakespeare4` *with that of* `Q68 Shakespeare6` *under a varying regime of the length of random strings. Warning: make sure that both program implement the same algorithm else make the corresponding leveling. Compare your results with others persons. Results might depend on the numbers of available cores and on the operational system because Linux has the fame of being excellent at parallel computing.*

85 ***Exercise.*** *Develop an application to be used as a lab to study the following questions:*

1. *For a given alphabet, the complexity of a task is the the length of the input phrase. The cost of finding a solution is the size of the population times the number of generations. We study how the cost of a solution depends on the complexity of the problem under various settings:*
2. *How does the cost vary when randomness is replaced by evolution?*
3. *Is evolution better than randomness?*
4. *How does the cost vary when the goal is divided in a ladder of subgoals?*
5. *Is favorable to divide a great goal into a ladder of subgoals?*

6. *How does the cost vary with respect to mutation? to recombination? to selection or type of reproduction? to population size?*
7. *Which is the optimal constant value of mutation? of recombination? of selection?*
8. *How can one mutate but defending what one has achieved?*
9. *Are aforementioned factors independent? Which is the global optimal setting?*
10. *How good is evolution to find mediocre solutions (with errors diminished to a half)? How does it do to find perfect solutions (with no errors at all)?*
11. *How is related the cost of finding a perfect solution to that of finding a mediocre one or an almost perfect one (one error in 28 chars, 96% of the whole task, 4% off-perfection)?*
12. *Which is the cost of finding a solution epsilon-off perfection?*

You can see our answer in program `Q85a Shakespeare7` *and its accompanying class* `Q85b EvolutionFunctional3`.

86 ***Exercise and Challenge.*** *Use your application to verify that evolution is extremely good when compared with randomness. Discuss the following implication: That is why evolution is a cosmological necessity.*

87 ***Challenge.*** *Add to your developed application the possibility to explore questions of the sort: Is a varying mutational rate favorable? In this regard, test the belief of the Author: the mutational rate can be high at the very beginning but must fade away as perfection increases.*

88 ***Challenge.*** *Add to your application a simulation of fossilization: Were a fossil study carried out, how would the quality of found fossils vary along strata in regard with perfection?*

We have put forward the class Shakespeare to illustrate how an easy problem looks when it is solved by evolution. Now, the adjective easy is rather subjective. How can we remedy this?

89 ***Challenge: the polynomial and exponential classes.*** *In reference to the problem of matching a phrase let us consider the function*

cost = population size $\times$ *number of generations to achieve the goal*

Our interest is to study the dependency of this function on the length of the input phrase. If the associated regression function can be bounded above by a polynomial, then we say that the corresponding algorithm belongs in the (evolutionary) polynomial class of complexity. So, proof else reject the belief of the Author: All our implementations of Shakespeare belong in this class. What is then a complex problem? It is a problem whose regression function cannot be bounded above by a polynomial but instead by an exponential function. What is an hyper-difficult problem? It is one that cannot be bounded above by an exponential function but by the exponential of an exponential. Which is the merit of evolution? The merit of evolution is that for randomness this problem is in the exponential class (verify!) but in the polynomial class for evolution. Let us define a problem to be easy when it is in the polynomial class and complex when it is in the exponential class. Then we can say : the problem of matching a phrase is difficult for randomness but easy for evolution.

90 ***Challenge: Functional Java is verbose.*** *It is a reality that a language can be highly economic. To see this, give a look at a genetic algorithm in JavaScript by (jsfiddle, [8] 2016?) that does the same thing as the previous program. You will also see that the program runs directly in the Internet browser because JavaScript comes with every browser. Decide whether the Author is right: Functional Java is less verbose than OOP Java but Functional Java is anyway too verbose as compared with JavaScript. Now, verbosity is not a defect: one always come again to old software so, verbosity gives context and facilitates reading. But it shall be well engineered.*

2.5 A complex problem

Everyone knows how difficult is to do something functional. As we know from `Vol III versión 2, where are the bugs` that difficulty is captured by SAT. So, this SAT is a mine of gold that deserves to be carefully and exhaustively exploited. We formally extend a heartwarming invitation to face that difficult enterprise by reformulating a pertinent program into our new flavor of functional programming.

We will encode our boolean values in BitSets so, let us see how to implement mutation and recombination.

91 ***Mutation with BitSets and streams can be seen in Program*** **`Q91 MutationWithBitSets`**.

92 ***Run the previous program and play with the code***.

93 ***Exercise: implement recombination with BitSets and streams:*** *You can see our answer in program* `Q93 RecombinationWithBitSets`.

We can now present the code to study the battle of Evolution against **MAX-K-SAT=Maximum K-satisfiability** that is the problem whose goal is to find an assignment that maximizes the number of satisfied clauses for a given conjunctive normal formula over N variables, where each clause contains at most K variables and each chosen variable can be negated.

94 ***KSAT in functional programming*** *The code is in program* `Q94 MAXKSAT`

95 ***Exercise.*** *Run previous program and play with the code.*

96 ***Exercise.*** *The notion of function is at the core of evolutionary biology and SAT, MAXKSAT and variants capture the associated complexity of its implementation. So, make your best to test the previous program to guarantee that it correctly does what it promises: to built propositions and to test over it diverse truth assignments for clause satisfaction.*

2.6 A very complex problem

With the use of grammars we devised in `Volume 2 version 2 Evolution as a software developer` a three hierarchical approach to have evolution developing the code for a binary adder. The conveyed idea is that GP and the evolution of grammars are synonyms. On the other hand, the genetic code is a grammar for it consists of a set of replacement rules of codons by amino acids. Thus, the problem of the evolution of the genetic code is naturally framed in the field of the evolution of grammars. That is why GP is so important for us. So, let us see how our three hierarchical approach looks from the perspective of functional programming.

97 ***Exercise.*** *The OOP program* `Q97 ThreeLevelsHier` *is a slightly corrected version of program of Vol 2 version 2* `ThreeLevelsHier, B129`. *It pretends to have evolution developing the code for a binary adder. Run the program and play with the code.*

98 ***Exercise.*** *Translate into functional programming the previous program . Warning: The Author saw how the program* `Q97 ThreeLevelsHier` *synthesized the code for a binary adder of* `1+1` *but was unable to see a binary adder for*

`11 + 11`. *Thus, we failed and therefore we do not know for certain which the cause could be: lack of patience? a bug? a misconception? So, make your best to check your code for fairness, consistency and correctness. Our answer is in program* `Q98 BinaryAdder3LH`. *Our program can develop the code for a binary adder of 1+1 but a binary added of 11+11 was not synthesized even 50 million trials were checked.*

99 ***Enabling continuity of populations + exercise.*** *The program developed by the Author for the previous exercise destroys populations every time a new round of evolution of combinatorial basis is done. What could happen if we enable continuity of populations? To study this question we developed program* `Q99 BinaryAdder3LH2`. *Run the program and play with the code. It seems to the Author that this program is faster than the previous one to develop a binary adder of 1+1 but, nevertheless, the synthesis of a binary adder of 11+11 and related was not seen even though the number of trials surpassed 66 millions.*

100 ***Exercise.*** *While the previous program allows the continuity of populations, anyway it destroys the codons of individuals for the sake of the evolution of the combinatorial subbases attached to them. Make you best to reuse the codons of extant individuals to build the new ones. You can see our answer in program* `Q100 BinaryAdder3LH3`. *This last version runs faster than previous ones but anyway and adder of 11+11 was not by the Author even a run up to 100 millions trials allowed.*

101 ***Challenge*** *Transform the program developed in the last exercise into a laboratory of evolution, whose purpose might be to teach that perfection demands an extremely high cost that may surpasses current technical limitations while bad and mediocre solutions are perfectly realizable.*

2.7 A strange problem

A very simple problem in regard with the style of software design was found in a previous volume to be extremely difficult for evolution. It is so intriguing that we have decided to reformulate it in functional programming to see whether or not we can catch an error of design of something else that could deceive us.

102 ***The bisection problem*** *In* `Vol 3 version 2 Where are te bugs` *we considered the* ***minimum bisection problem*** *for a graph* (V,E) *with an even number of vertexes, which consists in cutting* V *into two parts of equal size, such that the number of edges between them is to be minimized. We consider in our*

simulation graphs that are populated with links at random. The purpose is to study the number of operations of an algorithm that are needed to solve the problem depending on the numbers of vertexes and links. See program `Q102 Bisection`.

103 ***Exercise.*** *Run the program and play with the code. Verify that the program do not care of the restriction that a mutant bisection must be a bisection and that the recombination of two bisections must be a bisection and so it produces the obvious output for the unrestricted problem: all vertexes must be at one side and no vertex at the other. Develop the corresponding correction: there are infinitely many forms of doing that: one of ours is presented in program* `Q103 Bisection2`.

104 ***Exercise.*** *Develop a correction to the previous program: ours is presented in program* `Q104 Bisection2`. *Agree el reject t he conclusion of the Author: randomness does almost everything through the initial population and evolution adds very little and sometimes worsens the result.*

105 ***Graduation.*** *All along present volume we have implemented elitist and extremely elitist selection and besides we have allowed to turn off mutation and recombination so we also have implemented no selection, i.e. reproduction at random. This setting has allowed us to demean evolution by saying that sometimes it is worse than randomness. This assertion is abusive: there are infinitely many ways of implementing selection and we have missed one very appropriate to combine the power of randomness and that of selection: probabilistic cloning of each individual in proportion to fitness. This setting most probably will reproduce the fittest members but also will reproduce some unfit individuals that will induce the exploration of new alleys of evolution. In that way, the population might become cured from stagnation at local optima. Try this out.*

2.8 Conclusion

We have verified that evolution in the form of GA (Genetic Algorithms) or GP (Genetic Programming) is a numerical methodology to solve optimization problems. Being real, evolution must have verifiable properties. The one we have underscored here was that some problems are very easy for evolution (guess a phrase), there are others that demand more resources (find a satisfying assignment of a logical proposition), and there are others that break evolution (very simple problems in GP). Because the class of simple problems is not void, one must expect natural evolution to be responsible for some changes along lineages and no fossil record witnessing to this must be demanded. On the other hand the class of breaking problems is also non void and its epitome is GP, the ordinary world of evolutionary

genomics because the genome is software. This implies that we have a mandatory prediction: if evolution is the cause of the origin of the species, the fossil record must be filled in malformations and extant populations in malfunctions. Because the international evolutionary literature lacks that proof, that literature is deprived from scientific value.

Chapter 3

Conclusions

Challenge.

Answers

78, page 22. Following the implementation of mutation, recombination over lists of chars has the structure of a trifunction: it takes two chars and a probability. If the probability is less than the recombination rate, it returns the first char else the second. So, we scanned the web for trifunctions and made a demo in program Q78a TrifunctionDemo. Then, we implemented recombination in the program Q78b RecombinationAsTrifunction. The mathematical analysis of this setting is in (Rodríguez et al, [15] 2009.

82, page 22. Low mutation creates variability but high mutation creates chaos. So, when a problem can be divided in subproblems, a low mutation rate implements the strategy of divide, conquer and defend. When the mutation rate is high achieved partial results are not defended and are destroyed, here is no defense and with difficulty one can gather something. One might think that this rhythm is congruent with evolution: DNA repairing machine was coarse at the dawn of life and its performance augmented to become what it is today, a tremendous keeper of fidelity. This is wrong: there is no such thing as life with low performance biochemical machinery.

Bibliography

[1] KEVIN CHABOT (2013) Writing Clean Predicates with Java 8, Java Zone
https://dzone.com/articles/writing-clean-predicates-java
Verified 1/IX/2016 11

[2] ADAM L. DAVIS (2016) What's New in Java 8, section 10.2 Immutability
https://leanpub.com/whatsnewinjava8/read
Verified 1/IX/2016 13, 15

[3] DAWKINS R (1986) The Blind Watchmaker. W.W.Norton & Company, NY, London.
http://terebess.hu/keletkultinfo/The_Blind_Watchmaker.pdf
Revised 14/X/2016 20

[4] JEFF FRIESEN (2015)
Listing 5. ReferenceToSuperClass version 5: A functional interface Java 101.
http://www.javaworld.com/article/2946534/learn-java/java-101-the-essential
Verified 1/IX/2016 14

[5] MARIO FUSCO (2016) Higher-order Functions, Functions Composition, and Currying in Java 8 DZone Integration Zone
https://dzone.com/articles/higher-order-functions
Verified 1/IX/2016 6

[6] MARIO PIO GIOIOSA. "I LIKE CODE TO BE SIMPLE, DIRECT AND EFFICIENT" Java 8 Optional—Replace Your Get() Calls
https://dzone.com/articles/java-8-optional-replace-your-get-calls
Verified 2/IX/2016 16

[7] IDR SOLUTIONS (2015) REFERENCE TO AN INSTANCE METHOD OF A PARTICULAR OBJECT JAVA 8 METHOD REFERENCES EXPLAINED IN 5 MINUTES
https://blog.idrsolutions.com/2015/02/java-8-method-references-explained-5
Verified 1/IX/2016 14

[8] JSFIDDLE.NET (2016) Javascript Genetic Algorithm
http://jsfiddle.net/BBxc6/669/
Verified 2/IX/2016 25

[9] SAR MAROOF (2106) Java Quiz: Passing Objects to Methods The latest intermediate-level Java quiz from DZone's resident quizmaster Sar Maroof!
https://dzone.com/articles/java-quiz-passing-objects-to-methods
Verified 1/IX/2016 14

[10] JOHN I. MOORE, JR. (2014) Java programming with lambda expressions. A mathematical example demonstrates the power of lambdas in Java 8
http://www.javaworld.com/article/2092260/java-se/java-programming-w
Verified 2/IX/2016 19

[11] ORACLE (2015) Immutable Objects
https://docs.oracle.com/javase/tutorial/essential/concurrency/immut
Verified 1/IX/2016 12

[12] ORACLE (2016) Package java.util.function
https://docs.oracle.com/javase/8/docs/api/java/util/function/packag
Verified 1/IX/2016 6

[13] FEDERICO PERALTA SCHAFFNER (2016) An Informal Approach to Higher-Order Runnables In this detailed tutorial, Federico Peralta Schaffner details the mechanics of higher-order functions, including the higher-order runnable in Java 8. DZone, Java Zone
https://dzone.com/articles/higher-order-runnables
Verified 1/IX/2016 6

[14] BAZLUR RAHMAN (2016) Removing Items From ArrayLists in Java 8
https://dzone.com/articles/removing-items-from-arraylist-in-javanb
Verified 10/X/2016 vi

rahman16

[15] RODRIGUEZ J, F.B. CHRISTIANSEN, H.F. HOENIGSBERG (2016) Recombination and bitsets
http://arxiv.org/abs/0902.2904
Verified 5/IX/2016 33

[16] PIERRE-YVES SAUMONT (2014) Fibonacci Tutorial with Java 8 Examples: recursive and corecursive DZone
https://dzone.com/articles/do-it-java-8-recursive-and
Verified 2/IX/2016 15

[17] STUDYTRAILS (2015) Java 8 - Lambdas, Method References and Composition © Copyright StudyTrails 2012-2014
http://www.studytrails.com/java/java8/Java8_Lambdas_FunctionalProgramming.
Verified 1/IX/2016 14

[18] TIMYATES (2013) (2012) Currying.java
https://gist.github.com/timyates/7674005
Verified 1/IX/2016 5

[19] TUTORIALSPOINT (2016) Java 8 - Lambda Expressions
http://www.tutorialspoint.com/java8/java8_lambda_expressions.htm
Verified 31/VIII/2016 5

[20] TUTORIALSPOINT (2016) Default Method Example
http://www.tutorialspoint.com/java8/java8_default_methods.htm
Verified 1/IX/2016 15

[21] STACKOVERFLOW, LEO UFIMTSEV (2016) 'Can a java lambda have more than 1 parameter?
http://stackoverflow.com/questions/27872387/can-a-java-lambda-have-more-th
Verified 31/VIII/2016 5

[22] USER2572526 (2015) Java8 pass a method as parameter using lambdas, StackOverFlaw
http://stackoverflow.com/questions/32075813/java8-pass-a-method-as-paramet
Verified 1/IX/2016 14

[23] VENKAT SUBRAMANIAM (2014) vii

Functional Programming in Java, Harnessing the Power of Java 8 - Lambda Expressions, The Pragmatic Bookshelf, Dallas, Texas • Raleigh, North Carolina, 2014
Try this url:

```
https://www.google.com/search?client=ubuntu&channel
=fs&q=Functional+Programming+in+Java+Harnessing
+the+Power+of+Java+8+Lambda+Expressions
+Venkat+Subramaniam&ie=utf-8&oe=utf-8
```

This link is live in program `Q64 ListFiles` in the accompanying NetBeans project.

Else prompt Google with
Functional Programming in Java - Tistory pdf
Verified 1/IX/2016

[24] BENJAMIN WINTERBERG (2014) Java 8 Stream Tutorial
http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples
Verified 2/IX/2016 16