Functional Programming Constructs in Java 8 and Their Integration into Lessons of Object Oriented Architecture

Rudolf Pecinovský

University of Economics, Prague, 4 Winston Churchill sq., 130 67 Prague 3
rudolf@pecinovsky.cz

Abstract. In contemporary programming the significance of parallelizability of wide range of human activities is increasing. The development of various teaching tools is so quick that it is difficult to even watch it. Concurrently a natural endeavor is arising to install such programming constructs which would enable to transfer the main burden to the used libraries and frameworks and leave only a care for the best programming of the required business logic to programmers. A significant place among these constructs belongs to functional programming, above all the lambda expressions and data streams. This paper presents their basic characteristics and how these constructs can be included into the lessons on the object oriented architecture.

1 Introduction

The performance efficiency of computers increases at present above all due to the increased number of cores as well as the number of processors in the system. Of course, this efficiency is still increasing according to the well-known Moor's Law, but the efficiency of running programs does not follow the Moor's curve for a long time. With regards to the above mentioned growth of computer effectiveness the effectiveness of programs is subjected to the Amdahl's Law which says that the *T(n)* time needed for carrying out of the algorithm at *n* processors can be derived from the *T(1)* time needed for using one processor with the help of the following formula ([1], [3]):

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

where *B* presents the proportional representation of that part of the program that cannot be parallelized. If *B=0*, i.e. when the program cannot be parallelized at all, it would run with the same speed any time independently of how many processors would be at its disposal. If *B=1*, i.e. when the program will be parallelized by 100 per cent, the program would run quicker as many times as many processors would be at its disposal.

To provide the highest parallelizability of the program is usually not simple and mostly it requires a very good knowledge of not only processed issues but also of the programming constructs through which the parallel running is carried out. That´s why we can register a strengthening pressure to transfer as much of these abilities and skills into the functionality of compilers, frameworks and libraries and to set free the programmers who then would be able to better concentrate for proposing optimal business logic. Therefore various algorithms as well as data structures are developed throughout the world which enable such transfer and make it easier ([2], [3], [4], [5]).

The side effect of these established algorithms and data structures is finding areas, the proposal of which can be significantly improved with the help of these algorithms and data structures, despite the need of its possible future parallelizability is not considered.

Several such constructs have been brought by the functional programming. The lambda expressions could be included into them as well as closures and data streams, which are gradually incorporated into programming languages proposed originally for different paradigms. Even the eighth version of Java language which is at present the most wide-spread used language not only in programming practice, but also in teaching, includes these constructs into its portfolio. Let's have a look how to include these novelties into teaching with the highest efficiency.

2 Extended concept of the `interface` construct

The new version of Java language comes with enhancing the interface usability with the possibility to define an implicit definition of certain methods together with the definition of static methods. This seemingly tiny extension has a huge impact at possibilities of future extending of the functionality of classes implementing the given interfaces.

In previous versions of the language, when you wanted to extend the possibilities of classes implementing certain interface without modifying all classes which implement the given interface, mostly you had to define the new interface, which should be implemented by those classes that would offer this extended functionality.

In case this extension would not influence the current code, it would be more advantageous to define the new class designed most often according to the design pattern *Utility Class* or *Servant*, the methods of which had instances of the extended classes among their parameters and defined the needed extension for them. However, in the newly designed interfaces it is sufficient to add methods with implicit definitions only, and the instances of all classes implementing the given interface will automatically be enriched by the functionality defined like that.

The newly added possibilities can be used also for modifying the definitions of current interfaces and for ensuing simplification of definitions of those classes which will implement them in future. For example in [8] the `Position` crate representing the position in double dimensioned space is defined. The `IMovable` interface characterizing objects that know both to get and to set their positions is then defined as follows:

```
public interface IMovable extends IPaintable
{
    public Position getPosition();

    public void setPosition(int x, int y);

    public default void setPosition(Position position)
    {
        setPosition(position.x, position.y);
    }
}
```

When the definition is set like that, the classes implementing the `IMovable` interface do not have to define all three methods because the implicit definition of `setPosition(Position)` method will suit to prevailing majority of them, and that's why they will simply take it over.

However, this making the work easier is only a side effect of the above mentioned extension of the `interface` data construct possibilities. The main advantage is the above mentioned possibility to extend easily the functionality which does not need any interventions into classes that already implement the modified interface. In the new Java version a significant part of interfaces gained the new methods with implicit definitions, whereas many of these interfaces have more implicitly defined methods than the original ones.

3 Functional Interfaces

Such interface is called functional that requires a definition of the only method from the implementing class. If it declares further methods it has to offer their implicit implementation. No other requirements are placed at functional interface. No parameters of this method are determined, neither the type of its return value.

Therefore the functional interfaces are often defined as generic types and the specification of parameter types and the type of their return value of their method is usually entered through type parameters.

There is a number of functional interfaces in the Java standard library. The older ones are mostly one-purpose – they were proposed for one specific usage (e.g. the interface `java.lang.Comparable<T>` defines the method serving for the comparison of the given object with the other one), but in the Java 8 a number of universal functional interfaces have been added. They have precisely defined only signatures of their methods, but their contract (i.e. what they will be used for) is relatively free. The main purpose of their introducing is the signature of the method declared in them according to which these interfaces mostly received their names – e.g. as follows (the overview shows full names of interfaces at the left side and the signature of declared methods in the right side):

```
java.util.function.Consumer<T>         void accept(T t)
java.util.function.BiConsumer<T, U>    void accept(T t, U u)
java.util.function.Suplier<T>          T get()
java.util.function.Function<T, R>      R apply(T t)
java.util.function.BiFunction<T, U, R> R apply(T t, U u)
java.util.function.UnaryOperator<T>    T apply(T operand)
java.util.function.BinaryOperator<T>   T apply(T left, T right)
java.util.function.Predicate<T>        boolean test(T t)
java.util.function.BiPredicate<T, U>   boolean test(T t, U u)
```

4 Lambda expressions

The Lambda expressions represent the way how to define particular code part, which we would like to use in another part of the program, as an object. These code parts are defined by the compiler as instances of certain functional interface. The lambda expression value is written down in the following form:

```
parametr -> výraz
(parametry) -> výraz
parametr -> { příkazy }
(parametry) -> { příkazy }
```

On the left from the arrow there is always a list of parameters (it may be empty) and on the right there is an action which should be carried out. The following rules are valid:

- If there is the only parameter on the left, you do not have to put it into parentheses.
- If the compiler is able to derive the type of parameter, you do not have to quote it.
- If there is no parameter on the left, you have to quote the empty parentheses.
- If you evaluate certain expression on the right, you do not have to put it into braces.

The Lambda expressions behave as instances of functional interfaces, the method of which has the corresponding parameters and returns the value of corresponding type (the compiler arranges the relevant casting). If we need to remember them, we save them as values of the given functional interface. And when we cannot save them into the variable, we can pass them as the parameters of the methods.

Using of lambda expression can be demonstrated to students in a number of examples, the most often of which are presentations of sorting using the user-defined comparators. In [8] their application is demonstrated for using instances of the Repeater class which are able to repeat the entered code part as many times as is required. After finishing the required set of repeating the defined code is called which announces to the applicant that the required repeating has been finished. The signature of the discussed method is as follows:

```
public void repeat(final int times, Runnable action, Runnable finished)
```

5 Data streams

In computer technology the data streams are generally understood as sequences of data elements made available over time. They may acquire different, more specific meanings in various areas. Let’s precise firstly in which meaning this term will be used in this paper.

In programming we mostly name by this term the objects mediating the transfer of data from the source to the target – we are speaking about the input, output and input-output streams. This paper does not deal with streams comprehended like that; it is focused on objects which are named by this term in the type theory and in functional programming. The term stream is used there in the sense of potentially endless list which (contrary to the classic list) does not save data, but only „knows about them“.

Mostly those data are included into these streams at which we suppose their lazy evaluation. Therefore we independently define which data will be included into the stream (how the stream obtains these data), and how these data will be processed. Then, in a suitable moment, we ask the stream to apply the entered operations with these data.

These data streams can be introduced to students as an equivalent of a conveyor used during the assembly line production.

- At the beginning of the stream there is an input. In case of assembly line production it is a stock of parts, in case of a stream it will be a source of data - mostly some source container.
- Along to the assembly line there are workplaces where the trained workers carry out individual operations. In case of the stream we replace the workers by methods (more precisely by lambda expressions) that will carry out the required operation with the object. The processed object continues along the stream to the next workplace.
- At the end of the assembly line the accomplished product drops out, at the end of the stream we receive (at least we hope to receive ☺) the required result.

The streams differ from the collections and the arrays, as the main representatives of containers, in several important issues:

- They do not block any memory for processing data. The data are „flowing“, the stream processes them and sends them further ahead.
- In most cases they do not influence the source data, which is very important during multiple processing of data by several processors, because then everybody call rely on the expected properties of input data.
- The planned operations behave similarly as the workers at the assembly line. They do not flock to the stock (container) to process all entered data but they wait until the processed object „flows“ to them with the stream and then they look after them.
- The streams are not interested in the data input, if it is the final (classic container), or endless (data continuously flowing from certain external source).

The operations we order at the stream (stand of the assembly line) can be divided into two groups:

- **The intermediate** operations takeover the object, process it and send it further along the stream. Therefore the output value of current operations is again the data stream. Thanks to it we can concatenate these operations simply so that further operation will be applied at the result of the first one, similarly as we do it e.g. when further text is added into instances of the `StreamBuilder` class.

  When using concurrent operations we have to have on mind that some of them return their stream, but the others return the newly created stream. Therefore, generally, we cannot call the current method without remembering the stream which will return it. It is better to learn not to delete the returned streams and immediately call some of their methods, or at least save them.
- **The terminal** operations complete the stream activity and give over the received result to the surrounding program. The output value of these operations may be a collection, an individual object or void – this comes in case that the result is the required processing of an object.

The basic rule of the stream work is that during calling the methods incorporating current actions the stream only remembers what should be done at the given stop. But nothing is done at that moment, i.e. during incorporating the action. The activity will be started only at the moment of incorporating the finishing action. Only this will start the imaginary conveyor that transports the source objects from one workplace to the other, so that we could receive the required result at the end.

6 Internal iterators

Teaching with classical conception of processing the set of objects saved in a collection is based on using external processing of saved objects delivered through sequence iterators – the program asks the collection for an iterator which then delivers one saved object after another one and the program processes the given object. This processing is running quite under the charge of that program and the programmer who processes the program decides if he/she will use the possibility of parallel processing of more objects to achieve increased efficiency.

When using the streams the internal, batch processing of given objects is preferred. It means that the surrounding program does not require any iterator, but only provides the code (optimally the lambda-expression) describing how the elements in the stream should be processed. However, the organization of this processing is in charge of the stream and it is defined by the library authors. The program that asks for this processing informs the addressed stream if the processing of individual objects can be considered as sufficiently independent, so that they could be realized parallelly. The stream then decides how to divide processing to individual accessible processors.

The advantage of this access is multiple. As the most significant I would like to quote two of them:

- The auxiliary code that was responsible for controlling of iterations disappears from the program and the description of actions for processing the objects is far clear.
- In case some more sophisticated techniques of parallelization of executed activities will appear in future, it is not necessary to modify the program's solving application logics, but it is sufficient only to improve the stream library and thus automatically the program processing, which these streams use, will be made more efficient.

7 The new constructs and methodology *Architecture First*

The above analyzed techniques play into the hands of the *Architecture First* methodology ([9], [10], [11], [12]) which teaches in initial phases only the architectonic view on solving the problem and entrusts the creating of the necessary code to certain code generator. When using the above described constructs, employing of classic algorithmic constructs (as conditioned statements and loops) is considerably decreased because the area of tasks, for solving of which we need only the statement sequence, is significantly enlarged. Thus, at the same time, the area of tasks, for programming of which relatively simple code generators are sufficient, is also extended. The students can concentrate for studying and absorbing the key architectonic principles for a longer time, not being distracted by the necessity to get at the code level and define the operations by their own, because the used code generator does not suffice for their definition.

8 Conclusion

This paper introduced the new programming constructs that are included into the most used programming languages in the last time. It demonstrated their basic properties based on the Java language and showed how these constructs can be presented to students of introductory courses of programming. It also showed the impact of using these constructs on the architecture of the system and briefly outlined their most important advantages.

At the conclusion it also showed how including of these constructions according to the methodology *Architecture First* can influence the teaching and outlined its possible advantages.

Literature

[1] AMDAHL, G.: *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings 1967.

[2] *Developing Parallel Programs – A Discussion of Popular Models*. An Oracle White Paper September 2010. Available at http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-parallel-programs-170709.pdf.

[3] GOETZ B., PEIERLS T., BLOCH J., BOWBEER J., HOLMES D., LEA D.: *Java Concurrency in Practice*. Addison-Wesley Professional 2006. ISBN 978-0-32-134960-6

[4] *Implicit parallelism.* Available at http://en.wikipedia.org/wiki/Implicit_parallelism.

[5] NIKHIL, R. Arvind: *Implicit Parallel Programming in pH*. ISBN 1-55860-644-0

[6] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java – kompletní učebnice pro začátečníky, 2. aktualizované a rozšířené vydání.* Grada Publishing, 2008. ISBN 978-80-247-2653-3.

[7] PECINOVSKÝ Rudolf: *OOP – příručka pro naprosté začátečníky*. Computer Press, 2009, ISBN 978-80-251-2126-9.

[8] PECINOVSKÝ Rudolf: *Java 8 – učebnice objektové architektury pro mírně pokročilé*. Grada, 2013, ISBN 978-80-247-4638-8.

[9] PECINOVSKÝ, R, KOFRÁNEK, J. How to improve understanding of OOP constructs. Wroclaw 09.09.2012 – 12.09.2012. In: *Science Education Research Conference.* [online] Wroclaw : PTI, 2012, s. 19–24. URL: http://fedcsis.org/proceedings/fedcsis2012/pliks/136.pdf

[10] PECINOVSKÝ R.: Principles of the Methodology Architecture First. *Objekty 2012 – Proceedings of the 17th international conference on object-oriented technologies*, Praha. ISBN 978-80-86847-63-4.

[11] PECINOVSKÝ R.: Methodology Architecture First. *Proceedings of the international conference DidactIG 2013*, Liberec. http://jtie.upol.cz/clanky_1_2013/JTIE-1-2013.pdf

[12] PECINOVSKÝ, R., KOFRÁNEK, J.: *The Experience with After-School Teaching of Programming for Parents and Their Children*. Las Vegas 22.07.2013 – 25.07.2013. In FECS'13 -- The 2013 International Conference on Frontiers in Education: Computer Science and Computer Engineering.