Performance improvement of using lambda expressions with new features of Java 8 vs. other possible variants of iterating over ArrayList in Java

J. JURINOVÁ

Abstract

A fluent programming is the programming technique where operations return a value that allows the invocation of another operation. With the fluent programming, it is perfectly natural to end up with one huge statement that is the concatenation of as many operations as you like. The Java Development Kit (JDK) streams (added in Java 8) are designed to support fluent programming. Instead of looping over all elements in the sequence repeatedly (once for filter, then again for map, and eventually for toArray), the chain of filtermapper-collector can be applied to each element in just one pass over the sequence. In this context, we often encounter lambda expressions used to create locally defined anonymous functions. They provide a clear and concise way to represent one method interface using an expression. Oracle claims that use of lambda expressions also improve the collection libraries making it easier to iterate through, filter, and extract data from a collection. In addition, new concurrency features improve performance in multicore environments.

There are multiple ways to traverse, iterate, or loop collection in Java. Therefore, to solve one problem, we have several options for solutions that differ by undeniably increasing of the code readability. Searching for answers to the question of whether these new features really bring performance benefits over conventional way, is the subject of this paper.

1. INTRODUCTION

Iteration is the process of traversing a sequence. It can be performed in two ways: as an internal or external iteration. The external iteration uses an iterator for access to all sequence elements. The internal iteration is performed by the sequence itself; the user just supplies an operation to be applied to all sequence elements. Traditionally, Java collections offer external iteration. However, Java 8 provides excellent features to support iterating, filtering and extracting of elements in Java collections. Prior to Java 8, better (concise) way to iterate elements is by using *for each loop* (added in Java 5) or iterating over collection using *iterator* and selecting the required object. Though that approach works, it was very difficult to run them in parallel and take advantage of all the available cores in the processor. Many of the techniques that

10.2478/jamsi-2018-0007
©University of SS. Cyril and Methodius in Trnava

programmers have relied on in the past are now being replaced by better, more powerful constructs.

Lambda expressions are one of the most important features added in Java 8. They are such important since they add functional programming features to Java. In addition, a lambda expression addresses the bulkiness of anonymous inner classes by converting five or even more lines of code into a single statement. According Schildt (2014), the use of lambda expressions in Java "can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes." The conclusions of this paper demonstrate compliance with this statement. For example, this is particularly helpful when implementing many commonly used event handlers. Lambdas also make it easy to pass what is a piece of executable code as an argument to a method. Before introducing lambda expressions, programmers were forced into writing more words in code than needed. Now, we can pass in a function as a parameter.

2. DEFINITION OF OUR PROBLEM

The most common operation with collection classes are iterating over them and applying business logic on each element. In other words, anytime you have a collection of things you will need some mechanism to systematically step through the items in that collection. Suppose that you are creating an application and you want to create a feature that enables to perform any kind of action on members of the application that satisfy certain criteria. Frequency of the occurrence of this action is many times during the day. Therefore, the performance of implemented approaches would be very important for us. Our motivation for this work came from the diverse argument that the use of lambda expressions would result in more efficient code. According Angelika Langer (2015) the motivation of the designers of the Java programming language "for inventing streams for Java was performance or — more precisely — making parallelism more accessible to software developers." Another reason for our interest in this issue is an ongoing conflict between the younger and older generation of programmers, when many of the older programmers often have a

distrust of new approaches or a strong belief in the unnecessary change in validated and routine procedures and approaches. The question we are trying to answer is the following: *"Are lambda expressions with connection to the new features in Java 8 really faster than non-lambda expressions at accomplishing the same tasks?"* Therefore, our main goal is to determine whether the introduced lambda expressions have a speed advantage over non-lambda expressions for an identical task by getting a quantitative speed difference in nanoseconds. At the same time, we want to look at the presented approaches from a more complex point of view and together with an illustrative demonstration of legibility and the length of the code necessary for solving the identical task about the possible implemented variants.

3. MATERIALS AND METHODS

For implementation, we use open source framework Spring (http://spring.io/). Spring Tool Suite™ (STS) is an eclipse-based development environment that is customized for developing Spring applications. It provides a ready-to-use environment to implement, debug, run, and deploy Spring applications. We use Oracle 11g database, layered with Hibernate as the object mapper and then Spring Data JPA for abstraction and some nice boilerplate crud (create, read, update, delete) operations. As it is mentioned on the official web page of Spring (2017) "the goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores."

Users in applications are represented by the simple class with four fields (identification number, user name, age and e-mail addresses). Below you can see annotations that instruct hibernate how to persist the object.

```
@Entity
@Table(name = "PERF_TEST_USER")
public class User implements Serializable {
        private static final long serialVersionUID = 1L;

        @Id
```

```
        @SequenceGenerator(name = "PERF_TEST_USER_ID_GEN", sequenceName =
        "PERF_TEST_USER_SEQ", allocationSize = 1)
        @GeneratedValue(strategy = GenerationType.SEQUENCE,
                generator = "PERF_TEST_USER_ID_GEN")
        @Column(name = "ID")
        private Long id;

        @Column(name = "USER_NAME", nullable = false)
        private String userName;

        @Column(name = "EMAIL_ADDRESS", nullable = false)
        private String emailAddress;

        @Column(name = "AGE", nullable = false)
        private int age;

        public Long getId() { return this.id; }

        public void setId(final Long id) { this.id = id; }

        public String getUserName() { return this.userName; }

        public void setUserName(final String userName) {
                this.userName = userName; }

        public String getEmailAddress() { return this.emailAddress; }

        public void setEmailAddress(final String emailAddress) {
                this.emailAddress = emailAddress; }

        public int getAge() { return this.age; }

        public void setAge(final int age) { this.age = age; }
}
```

The reason for class simplicity is also the fact that we do not work with real data (reason: protection of personal data), but we generate them, what is sufficient for our test purposes. Work with the database, i.e. generating users, storing them, or removing them from the database is solved by using the *IUserService* interface implemented by the *UserService* class. This class for the data acquisition from the database implements two methods: ***public*** *List<User> getTestUsersAsList(****final int*** *countOfTestUsers)* and ***public*** *Iterator<User> getTestUsersAsIterator(****final int*** *countOfTestUsers)*, the

principle of which is the same. The goal is to return the required number of users from the database either as a collection or as an iterator. An *iterator* in Java is an instance of the *Iterator<E>* interface (Interface List<E> 2017). We can get an iterator for a collection using the *iterator()* method from the *Collection* interface. In the *getTestUsersAsList* method we use a method *List<E> subList(int fromIndex, int toIndex)*, which returns a view of the portion of the list between the specified *fromIndex*, inclusive, and *toIndex*, exclusive (Interface List<E> 2017).

```
public List<User> getTestUsersAsList(final int countOfTestUsers) {
     final List<User> testUsers = users.subList(0, countOfTestUsers - 1);
     return testUsers;
}

public Iterator<User> getTestUsersAsIterator(final int countOfTestUsers) {
     final List<User> testUsers =
             this.getTestUsersAsList(countOfTestUsers);
     return testUsers.iterator();
}
```

If there are not enough users in the database, they will be generated.

```
public User generateRandomUser() {
  final User user = new User();

  user.setAge(this.getRandomAge());
  user.setUserName(this.getRandomUserName());
  user.setEmailAddress(this.getRandomEmailAddress());

  return user;
}

// USER ATRIBUTES RANDOM GENERATOR

private int getRandomAge() {
  final int randomAge = new Random().nextInt(this.maxAgeTo -
this.maxAgeFrom + 1) + this.maxAgeFrom;
  return randomAge;
}

private String getRandomUserName() {
```

```
    final UUID randomUuid = UUID.randomUUID();
    return String.valueOf(randomUuid);
  }

  private String getRandomEmailAddress() {
    final String randomUserName = this.getRandomUserName();
    return randomUserName.concat("@user.com");
  }
```

A collection is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data (Collections 2017). They are fundamental to many programming tasks. We use an *ArrayList* class that is implemented using resizable array, because retrieval is the key operation for us. According Paul Javin (2017), the main difference between *ArrayList* and *LinkedList* is that *ArrayList* is implemented using resizable array while *LinkedList* is implemented using doubly linked list. Since array is an index-based data-structure, searching or getting element from array with index is fast. Array provides *O*(1) performance for *get(index)* method but remove is costly in *ArrayList* as you need to rearrange all elements. On the other hand, *LinkedList* does not provide random or index based access and you need to iterate over linked list to retrieve any element that is of order *O*(*n*). Insertions are easy and fast in *LinkedList* comparing to *ArrayList* because there is no risk of resizing array and copying content to new array if array gets full what makes adding into *ArrayList* of *O*(*n*) in worst case, while adding is *O*(1) operation in *LinkedList* in Java. *ArrayList* also needs to update its index if you insert something anywhere except at the end of array. Removal is like insertions better in *LinkedList* than *ArrayList. LinkedList* has more memory overhead than *ArrayList* because in *ArrayList* each index only holds actual object (data) but in case of *LinkedList* each node holds both data and address of next and previous node. According Paul Javin (2012) "*ArrayList* is more popular among Java programmer than *LinkedList*, but there are few scenarios on which *LinkedList* is a suitable collection than *ArrayList.*" This is not the subject of this paper. However, based on the above, it is obvious that the results achieved by using *LinkedList* would be different.

The developed Spring Boot application provides a simple Rest API interface for testing. These results are used to determine the difference, between the lambda and non-lambda expression in nanoseconds. For testing, we use a single method that can be called with the following mandatory parameters:

- *testTypes* (enum — for, iterator, stream…);
- *approach* (enum — 1, 2, 3…);
- *skipWarmUp* (boolean);
- *countOfWarmUpIterations* (number);
- *countOfAllIterations* (number);
- *countOfTestUsers* (number);
- *getDurationsList* (boolean — displaying also the results for each iteration of the test).

From the above, we can use multiple test variants (the *testTypes* attribute) for one tested approach at a time. With the *skipWarmUp* parameter, we can exclude the so-called "warm-up iterations" from the test results, the number of which is determined by the *countOfWarmUpIterations* attribute. The reason is the selection of data that could distort the results. The principle of testing could be defined in simple terms by this construction:

```
long startTime = System.nanoTime();
     // ... THE CODE BEING MEASURED ...
long estimatedTime = System.nanoTime() - startTime;
```

To communicate with our API, we use the SoapUI (5.3.0) tool, which is primarily designed for functional testing. At each call of our method we can see a clear result in the JSON format, from which we can unambiguously conclude testing for a given combination of input parameters.

4. DEFINITION OF LAMBDA EXPRESSIONS

Kishori Sharan (2014) defines a lambda expression as "an unnamed block of code (or an unnamed function) with a list of formal parameters and a body." However, this method is not executed on its own. Instead, it is used to implement a method

defined by a functional interface. The lambda expression signature must be the same as the functional interface method's signature, as the target type of the lambda expression is inferred from that context. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. A lambda expression is a poly expression — the type of a lambda expression is inferred from the target type thus the same lambda expression could have different types in different contexts. A functional interface is sometimes referred to as a SAM type, where SAM stands for Single Abstract Method. Lambda expressions are also commonly referred to as closures (Schildt 2014).

According Schildt (2014) "a functional interface is an interface that contains one and only one abstract method." A functional interface typically represents a single action. In the past, no method in an interface could include a body. Thus, all methods in an interface were implicitly abstract. With the release of JDK 8, this situation has changed dramatically. It is now possible for an interface method to define a default implementation. This new capability is called the default method (Default Methods 2017).

5. SYNTAX OF LAMBDA EXPRESSIONS IN JAVA (LAMBDA EXPRESSIONS 2017)

A lambda expression consists of the following: **(arguments) -> body**

- A comma-separated list of formal parameters enclosed in parentheses specifies parameters required by the lambda expression. If the parameter types of a lambda expression can be inferred, you can omit them. In addition, you can omit the parentheses if there is only one parameter. Parameters are optional, if no parameters are needed an empty parenthesis can be given.
- The arrow token -> (arrow operator, lambda operator).
- A body, which consists of a single expression or a statement block. If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement. A return statement is not an expression, in a lambda expression, you must

enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. It is illegal for a lambda expression to return a value in some branches but not in others. You can write following kind of code using lambdas:

- (params) -> expression
- (params) -> statement
- (params) -> { statements }

For example: lambda expression to test if the given number is odd or not: *n -> n % 2 != 0*. This lambda expression returns *true* if the value of parameter *n* is odd. Compared to a method a lambda definition lacks a return type, a throws clause and a name. Return type and exceptions are inferred by the compiler from the lambda body. So, the only thing that is really missing is the name.

6. RESULTS AND DISCUSSION

The Java platform includes a variety of ways to iterate over a collection of objects, including new options based on features introduced in Java 8. One common approaches to iterate over *ArrayList* in Java is *advanced for loop* (also known as *for each loop*, introduced to Java 5). If I have to remove elements while iterating then using *Iterator* or *ListIterator* is the best solution. Although iterators in Java have taken different forms, using an active iterator was essentially the only viable option prior to Java 8. For an active iterator (also known as explicit iterator or external iterator), the client controls the iteration in the sense that the client creates the iterator, tells it when to advance to the next element, tests to see if every element has been visited, and so on. For a passive iterator (also known as an implicit iterator, internal iterator, or callback iterator), the iterator itself controls the iteration. The client essentially says to the iterator, "perform this operation on the elements in the collection." Now we have possibility to use Java 8's *forEach()* method and features of the stream API which helps us fine-tune and parallelize the behaviour of Java iterators. For each tested option we defined a single class, as you can see in figure 1.

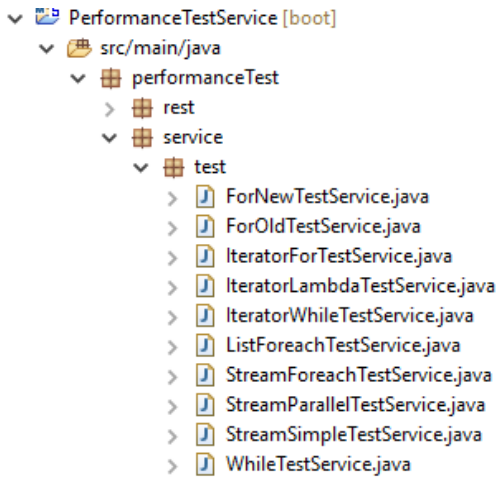

Figure. 1 List of classes for testing purposes.

In the examples below, we present these different iteration options to solve simple basic logic in the simplest way. One simplistic approach is to create several methods. Each method searches for members that match one or more characteristics. Our implemented methods return a collection of names of users who meet the simple filtering criterion. The task and even the selection criteria are not crucial, the applied approach is important. Consequently, the appropriate action can be taken over this data. There are more approaches to the problem solution that can be applied regarding the development of the Java programming language. We applied five different approaches that eliminate lack of the simplest approach presented below. It improves upon this approach with local and anonymous classes, and then finishes with an efficient and concise approach using lambda expressions. These will be presented in detail in another paper.

APPROACH 1: CREATE METHODS THAT SEARCH FOR MEMBERS THAT MATCH ONE CHARACTERISTIC

1st variant: Simple for loop and an integer index — old practice

```
public List<String> approach1(final int countOfTestUsers) {
  final List<User> users =
     this.userService.getTestUsersAsList(countOfTestUsers);
  final List<String> userNames = new ArrayList<>();

  for (int index = 0; index < users.size(); index++) {
     final User user = users.get(index);

     if ((user.getId() + user.getAge()) % 2 == 1) {
             userNames.add(user.getUserName());
     }
  }
  return userNames;
}
```

2nd variant: While loop — old practice

The use of the cycle with a condition at the beginning (*while loop*) if we need to iterate the whole collection has no logical justification. In such a case, a cycle with an explicitly defined number of repeats (*for loop*) is more appropriate and readable. For the sake of completeness, we present also this variant, as it is the case with the 4th variant. There is the assumption that there will be no performance differences between the 1st and 2nd (similarly between 3rd and 4th) variants.

```
public List<String> approach1(final int countOfTestUsers) {
  final List<User> users =
     this.userService.getTestUsersAsList(countOfTestUsers);
  final List<String> userNames = new ArrayList<>();

  int index = 0;
  while (index < users.size()) {
     final User user = users.get(index++);

     if ((user.getId() + user.getAge()) % 2 == 1) {
                     userNames.add(user.getUserName());
```

```
        }
    }
    return userNames;
}
```

3rd variant: Iterator + for loop

An iterator is a mechanism that permits all elements of a collection to be accessed sequentially, with some operation being performed on each element while isolating the user from the internal structure of the container. An iterator provides a means of "looping" over an encapsulated collection of objects. It was introduced in the Java JDK 1.2. An iterator should also be non-destructive in the sense that the act of iteration should not, by itself, change the collection. Of course, the operation being performed on the elements in a collection could possibly change some of the elements. It might also be possible for an iterator to support removing an element from a collection or inserting a new element at a point in the collection, but such changes should be explicit within the program and not a by-product of the iteration. Typically, the *hasNext()* and *next()* methods are used together in a loop. The *next()* method returns the next element from the collection. We should always call the *hasNext()* method before calling the *next()* method. If not, the *next()* method throws a *NoSuchElementException*. *ListIterator* extends *Iterator* to allow bidirectional traversal of a list, and the modification of elements, what we do not need now. Only collections that implement *List*, you can obtain an iterator by calling *ListIterator()*.

```
public List<String> approach1(final int countOfTestUsers) {
    final List<String> userNames = new ArrayList<>();

for (final Iterator<User> users =
        this.userService.getTestUsersAsIterator(countOfTestUsers);
        users.hasNext();) {
        final User user = users.next();

        if ((user.getId() + user.getAge()) % 2 == 1) {
                userNames.add(user.getUserName());
        }
    }
    return userNames;
```

```
}
```

4thvariant: Iterator + while loop

The same philosophy as in the 3rd variant, but using *while loop*. We have met such a use in the articles more often than the use *for loop*. This was the reason for the inclusion of this option.

```
public List<String> approach1(final int countOfTestUsers) {
  final List<String> userNames = new ArrayList<>();

  final Iterator<User> users =
     this.userService.getTestUsersAsIterator(countOfTestUsers);

  while (users.hasNext()) {
     final User user = users.next();

     if ((user.getId() + user.getAge()) % 2 == 1) {
            userNames.add(user.getUserName());
     }
  }
  return userNames;
}
```

5th variant: Advanced (enhanced) for loop = for each loop

From Java 5, we can use the *for each loop* to iterate over any collection whose implementation class implements the *Iterable* interface. This iteration is a more convenient way than using the *traditional for loop*. The creation of the iterator and calls to its *hasNext()* and *next()* methods are not expressed explicitly in the code, but they still take place behind the scenes. Thus, even though the code is more compact.

```
public List<String> approach1(final int countOfTestUsers) {
   final List<String> userNames = new ArrayList<>();

  for (final User user :
        this.userService.getTestUsersAsList(countOfTestUsers)) {
        if ((user.getId() + user.getAge()) % 2 == 1) {
               userNames.add(user.getUserName());
        }
```

```
  }
  return userNames;
}
```

6th variant: forEach method with lambda expressions

Collection classes that implement *Iterable* (for example all list classes) from Java 8 have a *void forEach(Consumer<? super T> action)* method (Interface Iterable<T> 2017). This method accepts a *consumer type*, which is a functional interface and that is why we can pass a lambda expression to it that has been passed as argument to every single element of the stream. Using it leads to shorter code constructs. *forEach()* is a terminal operation, which means that can only once be applied on a stream. The greatest difference comparing a *for loop* is that it cannot be interrupted ahead of time — neither with break nor with return. The form of using active iterators is not wrong and it is still being in use. Java 8 simply provides additional capabilities and new ways of performing iteration. For some scenarios, the new ways can be better. In particular, the *Iterable* interface provides a passive iterator in the form of a default method called *forEach().* In this case, the *forEach()* method is actually implemented using an active iterator in a manner similar to what you saw in *Advanced (enhanced) for loop = for each loop* example. Note the difference between the passive iterator in this example and the active iterator in the previous examples. There is no explicit loop. We simply tell the *forEach()* method what to do with the objects in the list. Control of the iteration resides within the *forEach()* method.

```
public List<String> approach1(final int countOfTestUsers) {
  final List<String> userNames = new ArrayList<>();

    this.userService
      .getTestUsersAsList(countOfTestUsers)
      .forEach(user -> {
                              if ((user.getId() + user.getAge()) % 2 == 1)
{
                                      userNames.add(user.getUserName());
                              }
                          });
  return userNames;
}
```

7th variant: Iterator + forEachRemaining method with lambda expressions

In Java 8, the *Iterator<T>* interface contains the extension method *void forEachRemaining(Consumer<? super T> action)* which performs the given action for each remaining element until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. The action is specified as a *consumer* that takes a lambda. A functional *consumer* interface represents an operation that accepts a single input argument and returns no result (API Specification 2017).

```
public List<String> approach1(final int countOfTestUsers) {
  final List<String> userNames = new ArrayList<>();

  this.userService.getTestUsersAsIterator(countOfTestUsers)
     .forEachRemaining(user -> {
             if ((user.getId() + user.getAge()) % 2 == 1) {
                     userNames.add(user.getUserName());
             }
     });

  return userNames;
}
```

8th variant: The stream API

Since Java 8, internal iteration is supported via streams. Stream is a key abstraction from the JDK collection framework that supports bulk operations with internal iteration. These bulk operations include *forEach()*, *filter()*, *map()*, *reduce()*, and many more methods. Thanks to the internal iteration, streams support sequential as well as parallel execution in a very convenient way. Arguably, one of the most important new features of JDK 8 is the stream API, which is packaged in *java.util.stream*. Collections and arrays can be used to generate streams — but beware: streams are not collections that store elements. Rather, think of a stream as a mechanism for carrying a sequence of values from a source through a pipeline of operations. Underlying data structures such as arrays or lists are therefore not changed. A stream pipeline consists of the

stream source, intermediate operations that transform the stream and produce a new stream, and a terminal operation that either produces a result or calls the *forEach()* method (Aggregate Operations 2017). In general, intermediate stream operations are lazy, which means that they are not immediately executed, but delayed until a terminal operation is applied. After the invocation of terminal methods, no further stream operations can be performed.

The key aspect of the stream API is its ability to perform pipeline operations that search, filter, map, or otherwise manipulate data. Often, you will use lambda expressions to specify the behaviour of these types of operations. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way (Schildt 2014).

```
public List<String> approach1(final int countOfTestUsers) {
  return this.userService
      .getTestUsersAsList(countOfTestUsers).stream()
      .filter(user -> (user.getId() + user.getAge()) % 2 == 1)
      .map(user -> user.getUserName())
      .collect(Collectors.toList());
}
```

The *Stream<T> filter(Predicate<? super T> predicate)* method expect a *predicate* instance which is also a functional interface as an argument and returns a new stream containing the elements that matched the conditions of predicate. Just remember that filter does not remove elements which matches the condition given in predicate, instead it selects them in output stream. Each predicate can have multiple conditions that need to be satisfied. A lambda expression can be passed into the *filter()* method instead, enabling you to customize how they behave. A lambda expression is the perfect way to express the actual parameter, which is simply a boolean-valued function. Aggregate operations process elements from a stream, not directly from a collection, which is the reason why the first method invoked in this example is *stream.*

Filter() is an intermediate operation, which means you can call any other method of stream e.g. *map()* as stream will not be closed due to filtering. The *<R> Stream<R> map(Function<? super T,? extends R> mapper)* method is an intermediate operation which returns a stream consisting of the results of applying the given function to the elements of this stream. Stream *map* method takes function as argument that is also a functional interface (Interface Stream<T>, 2017).

The *collect* operation is best suited for collections. Unlike the *reduce* method (Aggregate Operations 2017), which always creates a new value when it processes an element, the *collect* method modifies, or mutates, an existing value. If us *reduce* operation involves adding elements to a collection, then every time our accumulator function processes an element, it creates a new collection that includes the element, which is inefficient. It would be more efficient for us to update an existing collection instead. This version of the *collect* operation takes one parameter of type *Collector*. The *Collectors* class contains many useful reduction operations, such as accumulating elements into collections and summarizing elements according to various criteria. These reduction operations return instances of the class *Collector*, so you can use them as a parameter for the *collect* operation. We use the *Collectors.toList* operation, which accumulates the stream elements into a new instance of *List.* As with most operations in the *Collectors* class, the *toList* operator returns an instance of *Collector*, not a collection.

The intermediate operation *filter()* and *map()* shown in example is only two example of a transformation that can be performed on a stream. Other examples include *distinct()* and *sorted(),* which applies a function to the elements of the stream to produce a new stream. Similarly, there are a number of terminal operations other including *count(), average(), sum(), min(), max(),* and *forEach().* For more information about stream pipelines, see “Aggregate Operations” in the Java tutorial (Aggregate Operations 2017) or “Chapter 2: The Stream API” in Cay Horstmann’s book (2014).

9th variant: The stream API with forEach method with lambda expressions

The *forEach()* method is defined at two places, on *Iterable* interface (*6th variant: forEach method with lambda expressions example*) as well as on stream, which means *list.forEach()* and *list.stream.forEach()* both are valid.

```
public List<String> approach1(finalintcountOfTestUsers) {
  final List<String> userNames = new ArrayList<>();
  this.userService
        .getTestUsersAsList(countOfTestUsers).stream()
        .forEach(user -> {
                if ((user.getId() + user.getAge()) % 2 == 1) {
                        userNames.add(user.getUserName());
                }
        });
  return userNames;
}
```

10th variant: The parallel stream API

Collection classes do not only have a *stream()* method, which returns a sequential stream, but they also have a *parallelStream()* method, which returns a parallel stream. Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections provided that you do not modify the collection while you are operating on it. When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results. In other words: parallel streams have the potential to improve performance by allowing pipeline operations to be executed concurrently in separate Java threads; but thought that the order in which collection elements are processed can change.

Implementation of parallel bulk operations for collections requires a better separation of concerns. And this is what lambdas are for: they provide a convenient and concise notation for functionality, which can be passed as an argument to a bulk operation of a collection, which in turn applies this functionality in parallel to all its elements.

```
public List<String> approach1(final int countOfTestUsers) {
```

```
    return this.userService
         .getTestUsersAsList(countOfTestUsers).parallelStream()
         .filter(user -> (user.getId() + user.getAge()) % 2 == 1)
         .map(user -> user.getUserName())
         .collect(Collectors.toList());
}
```

Evaluation

Java 8 stream is very efficient replacement of looping both design and performance wise, because it separates what to do from how to do and supported internal iteration. In *the stream API* example, with one line of code a new list is created, containing only the desired names of users. Over traditional *for loop* we can write more succinct code, sometimes just one line long. You can pass lambda expression, which gives you the immense flexibility to change what you do in the loop. Aggregate operations do not contain a method like *next* to instruct them to process the next element of the collection. With internal delegation, your application determines what collection it iterates, the JDK determines how to iterate the collection. You did not iterate over the elements in the list when you use the stream. The stream did that for you internally. With external iteration, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially, that is the code can be executed only by one thread. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. In *the parallel stream API* example by changing only one word we take advantage of multiple CPU. So, you can perform the looping in parallel without writing a single line of multi-threading code. Of course, it is possible only on a computer with multiple cores, and with operations that can and should be executed in parallel. Kishori (2014) mentioned that Java has supported concurrent programming since the beginning. “It added support for parallel programming in Java 7 through the *Fork/Join framework*, which is not so simple to use, especially for beginners.” On the other side, *streams* come as a rescue. They are designed to process their elements in

parallel without you even noticing it. When you want to process elements in parallel you use *parallelStream()* method and the stream will take care of the rest. Streams take care of the details of using the *Fork/Join framework* internally.

It is obvious that these two approaches (traditional imperative programming vs. declarative programming) look very different. The imperative style intermingles the logic of iterating (the various nested loops) and the application of the declarative style, in contrast, separates concerns. It describes which functionality is supposed to be applied and leaves the details of how the functions are executed to the various operations such as *filter*, *map*, etc., what is according us more readable. Due to new *stream API* features you can write more expressive code, which is easy to understand, easy to read, easy to optimize and super easy to run in parallel without you worrying about multi-threading nightmare. The *stream API* supports the functional programming. Operations on a stream produce a result without modifying the data source. Like in the functional programming, when you use streams, you specify "what" operations do you want to perform on its elements using the built-in methods provided by the *stream API*, typically by passing a lambda expression to those methods, customizing the behaviour of those operations.

As we can see above (*6th a 9th variant*) the *forEach()* method is defined at two places. Prefer use *forEach()* with streams because streams are lazy and not evaluated until a terminal operation is called. Also obtaining stream gives you more choices e.g. *filtering*, *mapping* etc. (*8th variant*). The *forEach()* method is a terminal operation, which means you cannot reuse the stream after calling this method. It will throw *IllegalStateException* if you try to call another method on this stream. When you run *forEach()* method on parallel stream the order on which elements are processed is not guaranteed, though you can use *forEachOrdered()* to impose ordering. Based on the above, when comparing the 8th and 9th variants, for the reasons of legibility and clarity, we are for the use 8th variant. It is illogical to combine new approaches with old ones. The if-statement can be easily replaced by the filter method.

A non-lambda expression (*variants 1–5*) to accomplish the same feet requires creating a new list, using an if-statement to check the condition of each user, and then adding the wanted names of user to the list. So instead of filtering, the non-lambda expression

uses an if-statement. For inserting the specified elements into the list we use *java.util.ArrayList.add(int index, E element)* method (Class ArrayList<E> 2017). The *enhanced for loop* actually uses an iterator behind the scenes. That means the Java compiler will convert the *enhanced for loop* syntax to iterator construct when compiling. The new syntax just gives the programmers a more convenient way for iterating over collections than old fashion practices, but it is more limited. You cannot remove elements from the collection while iterating, there is no reference to the iterator to call the *remove()* method. Even though you would end up with with *ConcurrentModificationException*. The other practical difference between *for loop* and *enhanced for loop* is that, in the case of indexable objects, you do not have access to the index. Although you could manually create a separate index integer variable with *for each loop*. But there is extra operation, which is not needed. With *for each loop* we have no way to start from the middle of the collection. The *for each loop* provides no way to visit the previously visited elements. So, if you want to selective modify only certain object then your best bet is to use *traditional for loop* which keeps track of indexes. Also, streams do not offer random access on the source data via index or the like. Access to the first element is possible, however not on any following elements.

The *advanced for loop* is same as using *iterator* with *for* or *while loop*. However, both *while* and *for* in combination with *iterator* gives more flexibility and power as you can iterate collection. On the other hand, we can say that the *iterator* is error prone. Because, at the use of the *iterator* there is nothing special if the Java program goes into infinite loop, which eventually cause out of memory error. This error message (exception) may be caused by use of the *Iterator.next()* inside loop. An *iterator* is a one-time object. We cannot reset an *iterator* and it cannot be reused. To iterate over the elements of the same collection again, it is necessary to create a new *iterator* by calling the *iterator()* method of the collection. But using *forEachRemaining()* method with a lambda also lets you control the collection type or even to use a pre-existing collection instance if wanted. Lambda expressions have the great advantage of creating much more readable andclean code. I tend to use them wherever I get a real

benefit — in terms of readability — from them. Undoubtedly, there exists a lambda example that is not more or less readable then the example that does not use a lambda expression. In fact, there also exist an example, that has the similar number of code lines, but in the example above you can see that even when dealing with a simple basic logic, readability as well as verbosity are obvious.

You might still wonder why anyone would prefer the code in *Advanced (enhanced) for loop example* over *The stream API example*, which probably looks more familiar to most Java developers. It can be argued that the declarative approach in *The stream API example* is actually more readable and less error prone, at least once you understand the basic concepts of streams and stream operations. In particular, depending on the context, the logic in *Advanced (enhanced) for loop example* might not be thread-safe, while *The stream API example* is thread safe. It is also easier to parallelize the operations shown in *The stream API example* as you see in *The parallel stream API example.*

While there are several advantages to use the passive iterators and new features of Java 8, I thought that it would be interesting to test whether or not they provide a performance improvement.

The testing ran on the computer with an Intel® Core™ i7-5500U CPU @ 2.40GHz processor, 16 GB RAM, the 64-bit version of Windows 10 and a 64-bit version of Java (version 1.8.0_131). I timed the calls to each of these ten methods using *ArrayList* classes with collection sizes (1 000 users) but with different count of iterations. The data in table 1 presents the average duration of the operation for the presented approaches at a varying number of iterations. The aim was to find the answer to the research question, whether the number of iterations performed has a significant impact on the measured values. For the sake of the utmost accuracy, we measure the values in nanoseconds, but for the sake of clarity and conclusions, we also report them in milliseconds. It follows that for our other testing purposes, where we will only change the size of the data structure, it will be sufficient to perform 10 000 iterations.

Table 1. The result of testing — average duration of operation in ns and ms — different numbers of iterations.

approach	Approach1							
countOfIterations	1 000		10 000		100 000		1 000 000	
countOfTestUsers	1 000		1 000		1 000		1 000	
time unite	ns	ms	ns	ms	ns	ms	ns	ms
FOR_OLD	275 809,75	0,28	228 089,50	0,23	239 959,39	0,24	244 379,47	0,24
WHILE	292 745,69	0,29	221 698,49	0,22	241 535,65	0,24	235 898,24	0,24
ITERATOR_FOR	296 489,30	0,30	227 772,45	0,23	224 985,29	0,22	232 021,68	0,23
ITERATOR_WHILE	283 545,75	0,28	279 073,43	0,28	225 070,37	0,23	224 372,72	0,22
FOR_ADVANCED	326 107,87	0,33	237 142,25	0,24	215 781,47	0,22	214 921,40	0,21
LIST_FOREACH	226 403,02	0,23	192 308,70	0,19	179 811,89	0,18	178 805,51	0,18
ITERATOR_LAMBDA	233 293,20	0,23	188 423,71	0,19	169 199,60	0,17	173 992,42	0,17
STREAM_SIMPLE	248 457,09	0,25	205 321,19	0,21	174 555,27	0,17	178 316,26	0,18
STREAM_FOREACH	211 732,14	0,21	182 507,96	0,18	166 926,15	0,17	188 002,86	0,19
STREAM_PARALLEL	161 079,86	0,16	175 881,53	0,18	143 907,95	0,14	139 356,43	0,14

I timed the calls to each of these ten methods using *ArrayList* classes, with different collection sizes (1 000, 10 000 and 100 000 users). I ran the benchmark several times (10 000 iterations). To remove any startup or just-in-time (JIT) effects, we exclude first five iterations. The data presented in table 2 are arranged from the shortest processing of the operation to the longest one.

Table 2. The result of testing Approach 1 — average duration of operation in nanoseconds and milliseconds.

approach	Approach1							
countOfIterations	10 000		10 000			10 000		
countOfTestUsers	1 000		10 000			100 000		
time unit	ns	ms		ns	ms		ns	ms
STREAM_PARALLEL	161 159,79	0,16	STREAM_PARALLEL	913 255,42	0,91	STREAM_PARALLEL	11 974 573,92	11,97
STREAM_FOREACH	168 929,19	0,17	STREAM_SIMPLE	2 030 324,34	2,03	STREAM_FOREACH	26 971 814,24	26,97
ITERATOR_LAMBDA	169 719,38	0,17	ITERATOR_LAMBDA	2 129 308,08	2,13	ITERATOR_LAMBDA	27 185 775,28	27,19
STREAM_SIMPLE	174 435,76	0,17	STREAM_FOREACH	2 148 108,51	2,15	STREAM_SIMPLE	27 212 472,88	27,21
LIST_FOREACH	175 472,36	0,18	LIST_FOREACH	2 163 293,37	2,16	LIST_FOREACH	28 273 787,57	28,27
WHILE	210 944,88	0,21	WHILE	2 495 592,67	2,50	FOR_ADVANCED	32 324 300,40	32,32
FOR_ADVANCED	217 192,56	0,22	FOR_OLD	2 588 545,95	2,59	WHILE	32 880 768,40	32,88
ITERATOR_FOR	217 794,62	0,22	FOR_ADVANCED	2 645 102,00	2,65	ITERATOR_WHILE	33 434 215,47	33,43
FOR_OLD	217 822,30	0,22	ITERATOR_FOR	2 645 248,20	2,65	ITERATOR_FOR	33 703 254,11	33,70
ITERATOR_WHILE	221 575,12	0,22	ITERATOR_WHILE	2 652 470,27	2,65	FOR_OLD	34 245 669,42	34,25

In some cases, the benchmark results were not what we expected. For example, we expected that there would be no differences between using *for* (FOR_OLD) and *while loop*. Similarly, we did not expect any difference when implementing the *iterator* (ITERATOR_FOR and ITERATOR_WHILE). We can see that these differences are not significant, but they still exist at a certain level of detail. According Javin Paul (2016) "the *enhanced for loop* provides the cleanest way to loop through an array or collection in Java, as you don't need to keep track of counter or you don't need to call the *hasNext()* method to check whether there are more elements left." Paul (2016) also states that "an *enhanced for loop* is nothing but a syntactic sugar over *iterator* in Java." Also, based on the above facts, our expectation were that the use of an *enhanced for loop* (FOR_ADVANCED) will not bring any significant performance improvements over older approaches, which was confirmed by the investigation results. On the other hand, we expected the duration of the operation to be the same for these two approaches. However, we cannot confirm this definitively, since there are some minor differences. So, we can state that this is an improvement in terms of clarity and readability of the code, but not in terms of performance.

More extensive benchmarking should be performed using different computer configurations in a production or different simulated production scenarios before drawing any definitive conclusions. But based on this simple benchmark and the results shown in the table 2, the following statements appeared to be true: New features added to Java 8 improve performance, even though we were expecting greater differences between the individual options. As you can see the *parallel stream* is roughly 50% faster in big collection. At the processing of the data on the size of 1 000 users (small collection), these differences are not significant. We can say that using multiple threads to process them is overkill. The second shortest duration of the operation was achieved in two cases using the *list.stream.forEach()* method (STREAM_FOREACH). The use of the *forEach()* method applied directly over the collection (LIST_FOREACH) achieves slightly worse results. Very interesting results were achieved using *iterator* with the new extension method *forEachRemaining()* (ITERATOR_LAMBDA). We do not need to omit the use of streams with aggregate

operations (STREAM_SIMPLE), which, beside the code clarity, also brings positive performance results.

In general, benchmark experiment results must be taken with a grain of salt. Every benchmark is an experiment. Minor deviations in measurements can be caused either by running background processes, but most of them we eliminated, or we have to consider the fact that when measuring the duration of a process, we never achieve exactly the same results when performing multiple tests. Obviously, the deviations are minimal and do not have a significant impact on the conclusions. We tried to increase the accuracy of the data by many iterations as well as the calculation of the mean values.

APPROACH 2: CREATE MORE GENERALIZED SEARCH METHODS

The philosophy of all ten variants of iteration over the collection remains the same at Approach 2 as in Approach 1, only the number of conditions that are set for selecting suitable users changes. For this reason, just one example is illustrated.

1st variant: Simple for loop and an integer index — old practice

```
public List<String> approach2(final int countOfTestUsers) {
  final List<User> users =
        this.userService.getTestUsersAsList(countOfTestUsers);
  final List<String> userNames = new ArrayList<>();

  for (int index = 0; index < users.size(); index++) {
     final User user = users.get(index);

     if ((user.getId() + user.getAge()) % 2 == 1
             && user.getId() % 2 == 1
             && user.getUserName().contains("5")
             && user.getEmailAddress().contains("1")
             && user.getEmailAddress().startsWith("a")
             && user.getAge() % 2 == 0) {
                     userNames.add(user.getUserName());
                     }
     }
  return userNames;
```

```
}
```

We also timed the calls to each of these ten methods using *ArrayList* classes, with different collection sizes (1 000, 10 000 and 100 000 users). I ran the benchmark several times (10 000 iterations). To remove any startup or just-in-time (JIT) effects, we exclude first five iterations. The data presented in table 3 are arranged from the shortest duration of the operation processing to the longest.

Table 3. The result of testing Approach 2 — average duration of operation in nanoseconds and milliseconds.

approach	Approach2							
countOfIterations	10 000		10 000			10 000		
countOfTestUsers	1 000		10 000			100 000		
time unit	ns	ms		ns	ms		ns	ms
STREAM_PARALLEL	131 073,19	0,13	STREAM_PARALLEL	775 377,32	0,78	STREAM_PARALLEL	10 352 412,13	10,35
STREAM_FOREACH	145 108,63	0,15	STREAM_SIMPLE	1 754 125,24	1,75	STREAM_SIMPLE	23 272 473,62	23,27
STREAM_SIMPLE	151 853,15	0,15	STREAM_FOREACH	1 854 730,98	1,85	STREAM_FOREACH	24 315 980,06	24,32
ITERATOR_LAMBDA	152 857,10	0,15	LIST_FOREACH	1 926 932,75	1,93	ITERATOR_LAMBDA	25 028 585,49	25,03
LIST_FOREACH	167 999,86	0,17	ITERATOR_LAMBDA	1 931 942,53	1,93	LIST_FOREACH	25 575 949,43	25,58
FOR_ADVANCED	187 141,34	0,19	WHILE	2 213 931,07	2,21	WHILE	27 889 886,24	27,89
FOR_OLD	189 976,61	0,19	FOR_ADVANCED	2 231 369,21	2,23	ITERATOR_FOR	28 978 475,68	28,98
WHILE	192 609,15	0,19	FOR_OLD	2 270 359,61	2,27	FOR_ADVANCED	29 085 196,01	29,09
ITERATOR_WHILE	194 885,71	0,19	ITERATOR_WHILE	2 276 332,35	2,28	FOR_OLD	29 437 404,21	29,44
ITERATOR_FOR	195 339,55	0,20	ITERATOR_FOR	2 323 049,71	2,32	ITERATOR_WHILE	29 482 042,72	29,48

The order of individual variants remains more or less unchanged, minor changes in the order of some variants are not significant. However, it is worth to mention that the average duration of an operation is in each case shorter comparing Approach 1, even though the applied business logic of choice is more complex. Thus, we can conclude that the difficulty of the selection criterion does not have a significant effect on the prolongation of the operation duration.

CONCLUSION

A fluent programming (the chaining of operations) is a programming technique where operations return a value that allows the invocation of another operation. With fluent programming, it is perfectly natural to end up with one huge statement that is the concatenation of as many operations as you need. The JDK streams are designed

to support fluent programming: all intermediate operations return a result stream to which we can apply the next operation in the chain until we apply a terminal operation as the tail of the chain. The chaining of operations is a key feature of streams as it enables a number of optimizations. For instance, the stream implementation can arrange the execution of functions as a pipeline. Instead of looping over all elements in the sequence repeatedly (once for filter, then again for map, and eventually for *toArray*) the chain of filtermapper-collector can be applied to each element in just one pass over the sequence. Creating such a pipeline per element does not only reduce multiple passes over the sequence to a single pass, but also allows further optimizations. So, benefits of the new approach are that it can be more readable, less error prone, and more easily parallelized.

Our results can partly compare the results of research Ward and Diego (2015) who looked at the issue from a different perspective, but their main goal was also to determine if the newly introduced lambda expressions have a speed advantage over non-lambda expressions for an identical task. The conclusion of their research is that, "the addition of lambda expressions to Java 8 provide for more functional, concise, and readable coding. In addition, given enough execution time, the new lambda expressions can provide a performance advantage." We absolutely agree with them and results of our research is proof.

Of course, a readability is a matter of practice and of habit. Moreover, there is not a black and white answer to this problem. Javin Paul (2014) wrote: "When I first saw a Java code written using lambda expression, I was very disappointed with cryptic syntax and thinking they are making Java unreadable now, but I was wrong." We find similar opinions in practice, among IT students and at programming oriented blogs very often. Frequently, there is presented a belief that it is not worth paying attention to the new features of Java 8. As a result, these programmers remain in the use of old practices. However, we think that programmers who fail to adopt the new strategies will soon find themselves left behind. As far as the code length is concerned, in some simple examples, the version that uses aggregate operations is longer than the one that uses a *for each loop*, in some more complex tasks that versions that use bulk-data

operations will be more concise. We agree with Angelika Langer (2015) statement: "Expecting that parallel stream operations are always faster than sequential stream operations is naive." Using functional programming is not always better for various reasons. Some of them we try to mention briefly above and others require a deeper investigation of the problem. In fact, it is never "better", it is just different and allows us to reason about problems differently. Never expect a miracle, also, do not guess; instead, benchmark a lot.

REFERENCES

Aggregate Operations. The Java™ Tutorials. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://docs.oracle.com/javase/tutorial/collections/streams/index.html⟩.

API Specification. Java™ Platform, Standard Ed. 8. 2017. © Oracle 1993–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/overview-summary.html⟩.

Class ArrayList<E>. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html⟩.

Class Colectors. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html⟩.

Collections. The Java™ Tutorials. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://docs.oracle.com/javase/tutorial/collections/index.html⟩.

Default Methods. The Java™ Tutorials. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html⟩.

HORSTMANN, S. CAY. 2014. Java SE 8 for the Really Impatient. Addison-Wesley Professional, 2014. ISBN-13: 978-0-321-92776-7, ISBN-10: 0-321-92776-1. 240 pp.

Interface Iterable<T>. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html⟩.

Interface Iterator<E>. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html⟩.

Interface List<E>. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/List.html⟩.

Interface Stream<T>. Java™ Platform Standard Ed. 8. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html⟩.

JAVIN, P., 2012. Difference between LinkedList vs ArrayList in Java. © 2012–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://javarevisited.blogspot.sk/2012/02/difference-between-linkedlist-vs.html⟩.

JAVIN, P., 2014. 10 Example of Lambda Expressions and Streams in Java 8. © 2012–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://javarevisited.blogspot.sk/2014/02/10-example-of-lambda-expressions-in-java8.html#axzz4oaVpthXZ⟩.

JAVIN, P., 2016. How does Enhanced for loop works in Java? © 2012–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://javarevisited.blogspot.sk/2016/02/how-does-enhanced-for-loop-works-in-java.html#ixzz4jcEyMSvz⟩.

JAVIN, P., 2017. Java67: Java Programming tutorials and Interview Questions. © 2012–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://www.java67.com/2012/12/difference-between-array-vs-arraylist-java.html#ixzz4i7DJtfrq⟩.

KISHORI, S. 2014. Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams. Apress, 2014. ISBN 1430266597, 9781430266594. 704 pp.

Lambda Expressions. The Java™ Tutorials. 2017. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax⟩.

LANGER, A. and KREFT, K. 2013. Lambda Expressions in Java. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://www.angelikalanger.com/Lambdas/Lambdas.pdf⟩.

LANGER, A., 2015. Java performance tutorial — How fast are the Java 8 streams? [online]. [cit. 2017-03-10]. Available on internet: ⟨https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html⟩.

SCHILDT, H., 2014. Java 8 eSampler. Preview exclusive excerpts from brand-new and forthcoming Oracle Press Java JDK 8 books. Oracle Press. © 2014 McGraw-Hill Education. [online]. [cit. 2017-03-10]. © 2010. Available on internet: ⟨http://www.oracle.com/technetwork/java/newtojava/java8book-2172125.pdf⟩.

Spring. 2017. Working with Spring Data Repositories. © 2011–2017. [online]. [cit. 2017-03-10]. Available on internet: ⟨https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html⟩.

WARD, A., DEUGO, D. 2015. Performance of Lambda Expressions in Java 8. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://worldcomp-proceedings.com/proc/p2015/SER2509.pdf⟩.

WILLIAMS, M. and J. Q. NUNEZ, 2015. Java SE 8: Lambda Quick Start. © Oracle. [online]. [cit. 2017-03-10]. Available on internet: ⟨http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html⟩.

Jana Jurinová
Department of Applied Informatics and Mathematics,
University of SS. Cyril and Methodius, 917 01 Trnava,
Slovak Republic
Email: jana.jurinova@ucm.sk