

## Java 8 New Features Improvements and Complications

Jaza Mahmood Abdullah, Mohammed Anwar Mohammed and Danial Abdulkareem Muhammed

University of Sulaimani\Computer Science Department, Statistics & Computer, Iraq

Accepted 04 May April 2015, Available online 13 May 2015, Vol.3 (May/June 2015 issue)

### Abstract

*This paper will discuss the last version of JDK platform (JDK 8). As well as, it compares this new version with the previous versions of JDK and showing the change made in java 8 memory management. In addition, this paper aimed to present the advantages and disadvantages of the new features of JDK8. Nowadays, java language is used by many programmers to develop their projects. Therefore, appearing new advancements in java platforms are lead to change developer's attitudes about using such language. This paper tries to answer the questioning that whether these improvements have positive or negative effects on the experienced programmer's point of views.*

**Keywords:** JDK 1.8, JVM, Stream, Lambda Expressions, Type annotations, Reflection.

### 1. Introduction

Features advancement of JDK java platform is a controversial among many java developers. Acronym of Java Development Kit is JDK. Constructing application, applets, and components using the Java programming language needs JDK which is an environment growth for them [1]. Programs written in the Java programming language and running on the Java platform can be developed and tested by tools are involved in the JDK [1]. There are many versions of JDK for instance, JDK 5.0 which presents a number of new extensions to the Java programming language. Introducing generics is one of these features [2]. Abstracting over types is permitted by generics. Container types are the best illustrations, such as those in the Collections hierarchy [2].

Next version JDK 6.0 was released with new ability such as **Scripting** gives the capability of mixing JavaScript technology source code, which makes a better prototype. Database is another feature, when database application is developed; finding and configuring JDBC database is no more need. Nevertheless, efficient JDBC 4.0, an API with lots of vital advancements will also achieve. such as exceptional care for XML as an SQL data type and enhanced addition of Binary Large Objects (BLOBs) and Character Large Objects (CLOBs) into the APIs [3]

Before the latest version was JDK 7 that conceded many new features such as using Strings in Switch Statements. It is case sensitive and compared to if-then-else statement it is more competent. Superior care for Linux fonts: five logical fonts from the time when Java 1.0 necessary to be mapped for physical fonts such as Serif, Sans-serif, Monospaced, Dialog, and DialogInput [3].

Today, many developers use java programming language in developing their projects. Java users work with this language five days a week around eight hours a day. From this point, improvements in the features of java platform have impact on both expert programmers and beginner of this language. The experts believe that the new features help them to do a lot of work within a shorter time and less effort. Expert's point of views comes from their familiarity with almost all features of the different levels of JDK platform. Hence, old users of java language have positive attitudes with inventing new capabilities for JDK platform in order to be simpler, easier, and convenience. Inversely, new features of the java platform for the beginner of this language might create negative attitudes while they have lack of experience and of none verbose in writing programs.

Recently, new version of JDK 1.8 (Java Development Kite) has been released by Oracle (Oracle Corporation) with many enhancement and advancement. For example, lambda, expression, Stream, Base64 and From Permanent Generation to Metaspace.

In this paper, the cons and pros of the new JDK features will be discussed. Moreover, it illustrates the impact of these new features on learning curve of new programmers, furthermore, questioning that whether these improvements have positive or negative effects on the experienced programmer.

### 2. Literature Review

#### A. Lambda Expressions

One of the most controversial and significant upgrade that have made in java 8 model is a Lambda Expression; is

the most important theme and probably the greatest awaited feature for Java developer. Arguably, the highest motivation behind lambda feature is the hardware trend of going towards multi-core. Hardware designers prefer parallel approaches and software developers need to utilize the features of the underlying hardware [4] [5] [6].

On the other hand, all Java developers are not convinced of their usefulness. In particular they think adding new functional features to Java could be a mistake, because they argue that this could lead of compromising it is imperative and tough object oriented nature. [1][2] [4] [5]

Lambda Expression are anonymous methods for representing behavior as data, also known as closure, "This enables the development of libraries that do a better job of abstracting over behavior, which in turn leads to more-expressive, less error-prone code" As Brian Goetz, Oracle's Java language architect explained [4][5]. In this section, lambda expression will be discussed with straightforward examples. Also shows how much the Java core has been upgraded to support more-query-like calculations over collections.

Lambda Expression has this syntax (argument(s)) -> (body) for instance

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

In term of parameters, lambda expression may have none, one or more parameters. Also parameters enclosed in parentheses are separated by comma as usual, parameters type might be declared explicitly or they can be inferred from the context. Moreover, the body of lambda expression may include of more than one line of code, if the body is consisted of a single statement then curly brackets are not mandatory.

```
(int m, int n) -> { return m + n; }
(int m, int n) -> return m + n;
```

Also considered as a valid lambda expression, here are some other examples:

```
b(m, n) -> { return m + n }
() -> System.out.println("Hello World");
(String s) -> { System.out.println(s); }
```

Using lambda expression makes code statements shorter, which make codes more clear, readable and easier for maintenance. For example, handling events in swing:

```
//before java 8
JButton play = new JButton("Play");
Play.addActionListener(new ActionListener() {
    @Override
    Public void actionPerformed(ActionEvent e) {
        System.out.println("Without using Lambda Expression");
    }
});

//java 8 using Lambda Expression
Play.addActionListener(e -> {
    System.out.println("Using Lambda Expression");
});
```

However, these shorten in code statements come with some complications in JVM (Java Virtual Machine), because of the complex lambda nested structure, these complications can be understood during exception handling and error tracking. For example, when the line numbers from the stack traces correlated to the source code, much longer synthetic call stacks can be seen compared to the older version of java. As a result, tracing the cause of the exception will no longer be easy for the developers as used to be [7].

### B. Stream

The verbose way to express the iteration of a collection can be considered as shortcomings of Java before JDK 8. Moreover, the fact that collection can only be manipulated through iterators (for-loops, while-loops or manually) was frustrating the java developers. Example below shows how to iterate list of employees as we used to do before java 8.

```
List <Employee>listOfEmployees = /* ... */ ;
for (Employee em : listOfEmployees) {
    if (em.getSalary() >= 3000) { // $3000 & up
        System.out.print(em.getName());
    }
}
```

Fortunately, JDK 1.8 comes with Stream API package (java.util.stream) known as the Stream interface. It outlines a list of operators of several different categories. Each of these operators are usually functional interfaces and takes zero or more arguments, in this way, they can be expressed with lambda expressions [8]. By using combination of stream function with lambda expressions previous example can be expressed as follow.

```
listOfEmployees.stream().filter(em ->em.getSalary() >= 3000)
    .forEach(em ->System.out.print(em.getName()));
```

Also, the result of iteration can be collected as shown below, since stream does not offer any mechanism for storing these results. The Collectors class have used, it contains many common collectors; toList() and toSet() are the most frequently used, nonetheless, there are many other collector methods that can be used to perform sophisticated transforms on the data [9].

```
List<Employee> employeesAbove$3000 = listOfEmployees.stream()
    .filter(em -> em.getSalary() >= 3000).collect(Collectors.toList());
```

### C. Base64

It is describe a standard API in JDK 1.8 for Base64 encoding and decoding. Java programmers have had to depend on third-party libraries for encoding and decoding Base-64. Because these encoding schemes are often used to encode binary/octet sequences, which then transmitted as a textual data. It is regularly used by applications using Multipurpose Internal Mail Extensions (MIME), encoding passwords, message and others. Thus a project might contain several different implementations of Base64. For instance: Spring, Guava and Apache commons-codec have separate implementations [10].

Now, Java 8 has it is own Base64. Which is `java.util.Base64`; it has been designed to acts as a factory for Base64 encoders and decoders. Here are some methods that offered by Base64 class.

```
getEncoder ()
getDecoder ()
getUrlEncoder ()
getUrlDecoder ()
```

### D. From Permanent Generation to Metaspace

Another significant improvement was made in memory management. Pre-java 8 developers have to deal with `OutOfMemoryError` sometimes, due to the permanent generation (PermGen) space depletion of the HotSpot VM. Usually, this exception pops out because of dynamic re-deployments of the application. For example, load and unload of the Java EE application from the application server, which might triggering Class metadata leak; eventually lead to the complete depletion of the fixed PermGen space.

Now, HotSpot JVM in java 8 uses native memory for representing the class metadata instead of PermGen, as it has used separate memory from native memory (C heap). Removing PermGen space means configuration and tuning of this memory space via `-XX:MaxPermSize` and `-XX:PermSize` will no longer needed. Subsequently, the Class metadata will be relocated to the native memory and OldGen space [11] [12]. Figure () shows the change made in java 8 memory management.

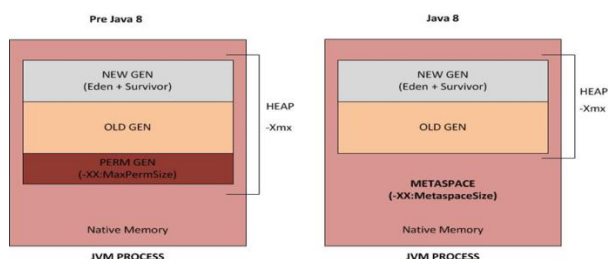


Figure 1: Java Memory Management comparison [27]

### E. Default methods on interfaces

Basically, in the pre-Java 8, interfaces can only declare a method without implementing it. Nevertheless, it can now implement methods; these methods are called Default methods, or as they are often called Virtual extension methods. As a result, it's possible to add a new default method to the interface without breaking the implementations [6].

```
public interface A{
    default void defaultMethod(){
        System.out.println("Calling A.defaultMethod()")
    }
}
public class Test implementsA {
}
```

From here, it is possible to call `defaultMethod` from class `Test` without implementing it, since it has been implemented inside `A` interface class.

```
Test t = new Test();
t.defaultMethod() // calling A.defaultMethod()
```

Now, the question might come "what will happen if there are two interfaces with the same method name?", considering previous example, let say interface `B` also created as follow:

```
public interface B{
    default void defaultMethod(){
        System.out.println("Calling B.defaultMethod()");
    }
}
public class Test implementsA, B {
}
```

Result, completion error with this message

```
"java: class Test inherits unrelated defaults for
defaultMethod() from types A and B"
```

This is because class `Test` has implemented both interface `A` and `B`, therefor, the compiler cannot decide which one to choose, the error can be resolved by overriding the conflicting method manually and refer to the targeted interface method. For instance, the `defaultMethod` of interface `A` can be summoned as follow: [6] [13]

```
public class Test implementsA, B {
    public void defaultMethod() {
        A.super.defaultMethod();
    }
}
```

### F. Interface static methods

Static methods are very much like to default methods which has been explained in previous section, with the

exception of overriding features. No more method overriding capability will help to avoid undesired results of weak implementation in child classes.

```
public interface Employee {
    default void print(String data) {
        if (!isNull(data))
            System.out.println("Employee Info ::" + data);
    }

    static boolean isNull(String data) {
        System.out.println("Interface Null Check");

        return data == null ? true : "".equals(data) ? true : false;
    }
}
```

Below additional class with poor implementation of isNull() method.

```
public class Address implements Employee {
    public boolean isNull(String data) {
        System.out.println("Address Null Check");

        return data == null ? true : false;
    }

    public static void main(String args[]) {
        Address phone = new Address();
        phone.print("");
        phone.isNull("07701995388");
    }
}
```

If annotation sign @Override added to the isNull() method in the Address class. Consequently, it will result in compiler error. That is because isNull(String data) is just a simple class, not an override method. After running the above code the result will be:

```
Interface Null Check
Address Null Check
```

In case the interface method has changed from static to default, the output will be completely dissimilar.

```
Address Null Check Employee Info ::
Address Null Check
```

That is because the static methods are accessible to interface methods only. Moreover, if the isNull() method removed from the Address class, Address object cannot use it any more. However, interface static methods can

be accessed by using its class name as used to be in pre-java 8 [13].

```
boolean result = Employee.isNull("009647701234567");
```

### G. Type annotations

It is one of the features of Java with some improvements in Java SE 8, prior to Java SE 8 declarations was the only place in program for applying annotations. Since the new release annotation can be used whenever a type used. For instance, it can be used with the new keyword, casts, throws and implements clauses [14].

```
@NotNull String str1 = ...
@email String str2 = ...
@NotNull @NotBlank String str3 = ...
```

Type definition with annotations [15]

```
new @Interned MyObject()
new @NotEmpty @ReadOnly List<String>(myNonEmptyStringSet)
```

Constructor with annotations [15]

```
myString = (@NotNull String) myObject;
query = (@Untainted String) str;
```

Type casts with annotations [15]

Stronger type checking was the main aim of creating type annotations. However, type checking framework does not supported by Java SE 8, it only allows the programmer to write or download a type checking framework that is implemented as one or more pluggable modules that are used in combination with the Java compiler [14].

### H. Reflection

Reflection can provide Java programmers the power of inspecting interfaces, classes, methods and fields at runtime, without having information about their names at compile time [17]. In addition, it is usually used by programs which require the capability to change the runtime activities of applications running in the Java virtual machine [17]. Also, it helps retrieving the definition of final or protected members, with the ability to remove the protection and use it as if it had been declared variable. However, this ability makes many guarantees of the program to be weakened [18].

Additionally, it is a powerful technique which is recommended to be used by developers who have a great knowledge of the fundamentals of the language [17]. Moreover, a useful usage of reflection is when writing a framework, which deals with the user-defined class without knowing what the members or class will be [18]. Furthermore, class inspection is often the first thing a programmer can do when using Reflection, then from the class information about class name, class modifiers (public,



private, synchronized), package, super class, implemented interfaces, constructors, methods, fields and annotations can be obtained [16]. Before performing any inspection the java.lang.Class object must be obtained, which is associated with any all types in Java [16]. In order to get the class of an object one of these methods can be used [19]:

1. Static variable class.
2. getClass() method of object.
3. java.lang.Class.forName(String ClassName).

```
Class classObj = Class.forName(classPath.className);
```

Using Class.forName()

```
String className = classObj.getName();
```

Getting the full class name including package name [16]

```
String simpleName = classObj.getSimpleName();
```

Getting only the class name [16]

```
Method[] method = classObj.getMethods();
```

Accessing the class methods [16]

The new Java SE 8 allows a new call for getParameters() which returns an array of parameter objects, also the method getName() can be called on the parameter's object [20]. Although it is preferable to not use reflection indiscriminately, the following titles should be concerned once accessing a code via reflection [17].

- Performance overhead: While using reflection some JVM optimizations cannot be completed, due to the involving of dynamically resolved types. As a result, reflective operations have slower performance than their non-reflective counterparts.
- Security restriction: Runtime permission is required for reflection which may not be allowed under a security manager. Therefore, codes that run in a restricted security context must be considered, for instance, an Applet.
- Exposure of Internals: Reflection may cause code to be dysfunctional, since it performs operations which may be not allowed in non-reflective code such as accessing private fields and methods.

### I. Compact Profiles

It was first introduced in Java SE 8 in order to reduce the memory footprint of applications that run on resource-

constrained devices, because it is a subset of the full Java SE platform API [21]. Additionally, Java SE 8 has released three Profiles which are compact1, compact2 and compact3. They are organized in additive order, which means that the last Profile contains all the APIs of the previous smaller Compact Profiles and adds proper APIs on top when used. Also the second Profile has all the APIs of first Profile with other additional APIs, whereas, it contains less APIs comparing to the third Profile. In addition, this ordering helps reducing memory usage by the applications. For instance, if an application does not require Swing/AWT/2D graphics libraries, it uses a profile which does not contain those libraries [22]. Nonetheless, Compact Profiles deal only with the API choices; they are independent from the Java virtual machine, the language proper, or tools. The bellow table shows the high-level structure of the Compact Profiles:

```
DateFormat dateFormat = new SimpleDateFormat("dd/MM/yy HH:mm:ss");
Date currDate = new Date();
System.out.println(dateFormat.format(currDate));
```

### J. Advantages of Compact Profiles

The main inspiration of releasing Compact Profiles is running applications on resource-constraint devices, without using the complete Java SE platform [22], as consequence of this advantage other benefits can be derived, such as, increasing performance and start up time, reducing unused code is an admirable idea from a security perspective, and helps downloadable applications to be downloaded quickly [23].

### Date and Time

A new Date and Time API has been introduced in Java 8, which is easier to read, safer and more comprehensive than the previous API. The implementation of Calendar class was not improved enough since its introduction; nevertheless, the introduction of new API Joda-Time was a great replacement for it. On the other hand, the new Date and Time API has initialized many different classes to represent time, date, time period, and time zone specific data, besides there are transformers for dates and times [24].

Prior to Java 8 getting date and time was as follow:

```
DateFormat dateFormat = new SimpleDateFormat("dd/MM/yy HH:mm:ss");
Calendar calendarObj = Calendar.getInstance();
System.out.println(dateFormat.format(calendarObj.getTime()));
```

### Using Date Class Prior to Java 8

**Table 1:** High-level composition of the Compact Profiles [8]

Full SE API	Beans	JNI	JAX-WS
	Preferences	Accessibility	IDL
	RMI-IIOP	CORBA	Print Service
	Sound	Swing	Java 2D
	AWT	Drag and Drop	Input Methods
	Image I/O		
compact3	Security <sup>1</sup>	JMX	
	XML JAXP <sup>2</sup>	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (java.lang.*)	Security	Serialization
	Networking	Ref Objects	Regular Expressions
	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNDI	Override Mechanism
	Extension Mechanism	Scripting	

Using Calendar class prior to Java 8

In Java 8 these classes are used for getting dates and times:

- **LocalDate** - Day, month and year without time zone.
- **LocalTime** - Time of day only without time zone.
- **LocalDateTime** - Both date and time without time zone.
- **ZonedDateTime** - For timezone specific time.

Previous to Java 8, in order to calculate the time eight hours in the future the bellow would be written:

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.HOUR, 8);
cal.getTime();
```

Calculating eight hours in the future before Java 8 [24]

In Java 8 it can be done simply like the following:

```
LocalTime now = LocalTime.now();
LocalTime later = now.plus(8, HOURS);
```

Calculating eight hours in the future using Java 8 [24]

There are also methods such as plusDays, plusMonths, minusDays, and minusMonths. For instance:

```
LocalDate today = LocalDate.now();
LocalDate thirtyDaysFromNow = today.plusDays(30);
LocalDate nextMonth = today.plusMonths(1);
LocalDate aMonthAgo = today.minusMonths(1);
```

Using different methods [24]

Since the new Date and Time types are immutable, each method returns a different instance of LocalDate, and the

original instance, today, remains unchanged. As a result, it allows them to be thread-safe and cacheable.

In Java 8 a number of enums for representing days and hours has been introduced such as java.time.temporal.ChronoUnit instead of using constant integers as used in Calendar API. In addition, Clock class can also be used in combination with dates and times. Moreover, Period and Duration is another feature of Java 8 for representing time differences as humans understand them. Period is a date-based amount of time, for instance, '4 years, 1 month and 8 days'. Duration is a time-based amount of time, for example, '25.8 seconds'. Additionally, for determining Periods and Durations the between method is used, also they can be subtracted or added to Java 8 date types. [24].

```
Period p = Period.between(date1, date2);
Duration d = Duration.between(time1, time2);
```

Using between method for determining Periods and Durations [24]

```
Duration twoHours = Duration.ofHours(2);
Duration tenMinutes = Duration.ofMinutes(10);
Duration thirtySecs = Duration.ofSeconds(30);
```

Creating durations [24]

```
LocalTime t2 = time.plus(twoHours);
```

Adding Duration to the LocalTime [24]

Furthermore, TemporalAdjuster is another class which is introduced in Java 8, used to do tricky date "math" that is popular in business applications, for instance, finding the next Tuesday or first Tuesday of the month. It contains many useful methods for creating TemporalAdjuster, such as:

- firstDayOfMonth()
- firstDayOfNextMonth()
- firstInMonth(DayOfWeek)
- lastDayOfMont()
- next(DayOfWeek)
- nextOrSame(DayOfWeek)
- previous(DayOfWeek)
- previousOrSame(DayOfWeek)

TemporalAdjuster uses with method, which returns a modified copy of the date-time or date object. For example:

```
import static java.time.temporal.TemporalAdjusters.*;
//...
LocalDate nextTuesday = LocalDate.now().with(next(DayOfWeek.TUESDAY));
```

Using with method [24]

Java 8 provides developers to work on Time Zones by using a new class known as ZonedDateTime. There are two kinds of

Zonelds, geographical regions and fixed offsets. This is to compensate for things like “daylight saving time” which can be very complex.

```
ZoneId mountainTime = ZoneId.of("America/Denver");
ZoneId systemZone = ZoneId.systemDefault();
```

An example of getting ZoneId instance [24]

### Optional

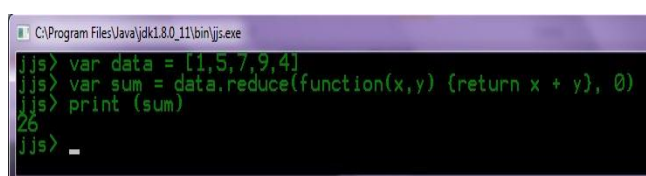
In order to avoid null return values (and as a consequence NullPointerException) the new Java SE 8 has introduced a new class called Optional in the java.util package. The key point of Optional is to provide a means for a function returning a value to denote the absence of a return value. In addition, Optional has many helpful static, instant and concrete methods for dealing with missing values such as [24]:

- Optional.of(x) wraps a non-null value.
- Optional.empty() represents a missing value.
- Optional.ofNullable(x) creates an Optional from a reference that may or may not be null.
- isPresent() determines if there is a value.
- get() gets the value.
- orElse(T) returns the given default value if the Optional is empty.

On the other hand, others are arguing that Optional is only a wrapper that has a reference to some other object and is not close to being a solution for NullPointerExceptions. Additionally, it increases the heap size, makes debugging more difficult, throws NullPointerExceptions and can itself be null, causing a NullPointerException [25].

### Nashorn

Nashorn is a JavaScript engine which released in Java SE 8 and it replaces Rhino as the default JavaScript engine for the Oracle JVM [26]. In addition, it has a better performance because of the using of invokedynamic feature of the JVM. It includes a command-line tool (jjs) which is located in \$JAVA\_HOME/bin [24]. Moreover, Nashorn can be used as a standalone engine using the command-line or it can be used as an embedded scripting engine inside Java applications. Java and JavaScript can work interoperability; Java types can be implemented and extended from JavaScript.



```
C:\Program Files\Java\jdk1.8.0_11\bin>jjs.exe
jjs> var data = [1,5,7,9,4]
jjs> var sum = data.reduce(function(x,y) {return x + y}, 0)
jjs> print (sum)
26
jjs>
```

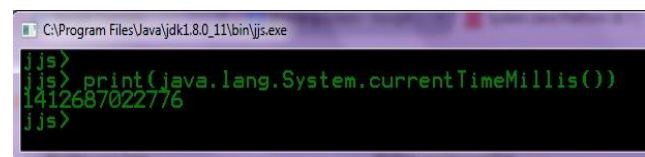
Using jjs to find the sum of numbers

### Running JavaScript from Java



```
1 import javax.script.ScriptEngine;
2 import javax.script.ScriptEngineManager;
3 import javax.script.ScriptExceptions;
4
5 public class EngineTest{
6     public static void main(String args[]) throws ScriptException{
7         ScriptEngineManager engineManager = new ScriptEngineManager();
8         ScriptEngine engine = engineManager.getEngineByName("nashorn");
9         engine.eval("function p(s) { print(s) }");
10        engine.eval("p('Hello Nashorn')");
11    }
12 }
```

The output will be “Hello Nashorn”. However, the typical variables which are available in browsers such as window and document are not available here [24].



```
C:\Program Files\Java\jdk1.8.0_11\bin>jjs.exe
jjs>
jjs> print(java.lang.System.currentTimeMillis())
1412687022776
jjs>
```

Using Java in jjs

### 3. Discussion

For every major JDK release, there are two sides must be addressed, firstly, bright side which is been discussed previously in this paper, secondly the dark side which it will be debated in this section. Over all, JDK 8 is a great successful improvement of java. However, before getting too much enthusiastic about it, one must admit that some of the newly placed futures were already existed in other languages, for instances, lambda expressions were exist in Java script, the first mainstream language. Here are some minor disadvantages:

- It is true that lambda expression has shortened the program lines. However, these shorten in code statements come with some complications in JVM (Java Virtual Machine), because of the complex lambda nested structure, these complications can be understood during exception handling and error tracking. For example, when the line numbers from the stack traces correlated to the source code, much longer synthetic call stacks can be seen compared to the older version of java. As a result, tracing the cause of the exception will no longer be easy for the developers as used to be [7].
- Features such as Overloading, generics and varargs are became even more complicated than ever before, these might not be an everyday problems, however, it might lengthen the java learning curve for the beginners.
- Some keywords are not supported in the default methods on interfaces, they cannot be finale, and for example the code blow will not work.

```
public interface A {
    default final void run(Runnable r) {
        run(r, 1); // just one time
    }

    default void run(Runnable r, int times) {
        for (int i = 0; i < times; i++)
            r.run();
    }
}
```

Also default methods on interfaces cannot be synchronized,

```
public interface A {
    default synchronized void noSynchronizedAllowed() {
        // modifier synchronized not allowed
        System.out.println("no Synchronized");
    }
}
```

- Due to not completion of some JVM optimizations reflective operations are slower than non reflective operations.

Security considerations must be taken into account, this is because of reflective requires runtime permissions.

## Conclusions

Nowadays, life has become simple and fast compare to the past. This is also correct in developing system via different programming languages. Newest versions of programming languages' components contain several features and advancements that give new abilities to the programming languages for developing system. In this paper, some of the new features and capabilities of the latest version of JDK has been presented. Moreover, we present a brief comparison between the previous version of the JDK components' features and the latest version's new features. Furthermore, Advantage and disadvantages of the latest version of this platform were presented and discussed. Finally, new abilities has been discussed to show that whether these improvements have positive or negative effects on the experienced programmer, as well as to show that whether these advancements have positive or negative impact on the beginner users of java programming.

## References

- [1] Oracle 2014, README Java™ Platform, Standard Edition 8 Development Kit JDK™ 8 [Online]. Available at <http://www.oracle.com/technetwork/java/javase/jdk-8-readme-2095712.html> [Accessed at 14/02/2015]
- [2] Oracle 2014, The Java™ Tutorials [Online]. Available at <http://docs.oracle.com/javase/tutorial/extra/generics/intro.html> [Accessed at 15/02/2015]
- [3] Slides from deck presented at EclipseCon Europe 2011, on November 2nd in Ludwigsburg, Germany [Online]. Available at <http://www.oracle.com/technetwork/articles/javase/beta2-135158.html> [Accessed at 15/02/2015]
- [4] Mario, F. (2013) Why We Need Lambda Expressions in Java - Part 1 [Online]. Available at <http://java.dzone.com/articles/why-we-need-lambda-expressions> 03.27.2013 [Accessed at 25/02/2015]
- [5] Brian, G. (2014) java magazine, Lambda Expression, pg6 [Online]. Available at [http://www.oraclejavamagazine-digital.com/javamagazine\\_open/20140304#pg7](http://www.oraclejavamagazine-digital.com/javamagazine_open/20140304#pg7) [Accessed at 17/02/2015]
- [6] Anton, A. (2013) Java 8 Revealed: Lambdas, Default Methods and Bulk Data Operations [Online]. Available at <http://zeroturnaround.com/rebellabs/java-8-revealed-lambdas-default-methods-and-bulk-data-operations/> [Accessed at 19/02/2015]
- [7] Tal, W. (2014) The Dark Side Of Lambda Expressions in Java 8 [Online]. Available at <http://www.takipiblog.com/the-dark-side-of-lambda-expressions-in-java-8/> [Accessed at 20/01/2015]
- [8] David, H. (2013) JDK 8 3/3 - The Stream API [Online]. Available at <http://blog.hartveld.com/> [Accessed at 20/01/2015]
- [9] Brian, G. (2014) Lambdas and Streams in Java 8 Libraries [Online]. Available at <http://www.drdobbs.com/jvm/lambdas-and-streams-in-java-8-libraries/240166818?pgno=2> [Accessed at 22/01/2015]
- [10] Alan, B. (2014) JEP 135: Base64 Encoding & Decoding [Online]. Available at <http://openjdk.java.net/jeps/135> [Accessed at 08/02/2015]
- [11] Pierre, H. (2013) Java 8 : from PermGen to Metaspace [Online]. Available at <http://javaeesupportpatterns.blogspot.co.uk/2013/02/java-8-from-permgen-to-metaspace.html> [Accessed at 12/02/2015]
- [12] Jon, M. (2014) JEP 122: Remove the Permanent Generation [Online]. Available at <http://openjdk.java.net/jeps/122> [Accessed at 17/02/2015]
- [13] Pankaj, K. (2014) Java 8 Interface Changes – static methods, default methods, functional Interfaces [Online]. Available at <http://www.journaldev.com/2752/java-8-interface-changes-static-methods-default-methods-functional-interfaces> [Accessed at 21/02/2015]
- [14] Oracle. (2014) Type Annotations and Pluggable Type Systems [Online]. Available at [http://docs.oracle.com/javase/tutorial/java/annotations/type\\_annotations.html](http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html) [Accessed at 23/02/2015]
- [15] Michael, S. (2014) Java 8 Type Annotations [Online]. Available at <http://java.dzone.com/articles/java-8-type-annotations> [Accessed at 24/02/2015]
- [16] Jakob, J. (2014) Java Reflection Tutorial [Online]. Available at <http://tutorials.jenkov.com/java-reflection/index.html> [Accessed at 25/02/2015]
- [17] author. (2014) , Java Reflection Tutorial – List Methods Of A Class [Online]. Available at <http://sanjaal.com/java/tag/drawbacks-of-java-reflections/> [Accessed at 25/02/2015]
- [18] Dec 8 '11 Kilian Foth , Why should I use reflection? [Online]. Available at <http://programmers.stackexchange.com/questions/123956/why-should-i-use-reflection> [Accessed at 27/01/2015]



- [19] Pankaj Kumar August 3, 2013 [Online]. Available at <http://www.journaldev.com/1789/java-reflection-tutorial-for-classes-methods-fields-constructors-annotations-and-much-more#get-class-object> [Accessed at 27/01/2015]
- [20] Andreas Jan 30 '14, How to get Method Parameter names in Java 8 using reflection? [Online]. Available at <http://stackoverflow.com/questions/21455403/how-to-get-method-parameter-names-in-java-8-using-reflection> [Accessed at 25/01/2015]
- [21] Oracle. (2014) Compact Profiles [Online]. Available at <http://docs.oracle.com/javase/8/docs/technotes/guides/compactprofiles/compactprofiles.html> [Accessed at 20/01/2015]
- [22] AJITESH, K. (2014) Why & When Use Java 8 Compact Profiles? [Online]. Available at <http://vitalflux.com/why-when-use-java-8-compact-profiles/> [Accessed at 05/02/2015]
- [23] Jim Connors-Oracle. (2013) An Introduction to Java 8 Compact Profiles [Online]. Available at [https://blogs.oracle.com/jtc/entry/a\\_first\\_look\\_at\\_compact](https://blogs.oracle.com/jtc/entry/a_first_look_at_compact) [Accessed at 07/02/2015]
- [24] Adam L. (2015) What's New in Java [Online]. Available at <https://leanpub.com/whatsnewinjava8/read#leanpub-auto-new-date-and-time-api> [Accessed at 03/03/2015]
- [25] Hugues, J. (2014) Java 8 Optional: What's the Point? [Online]. Available at <http://java.dzone.com/articles/java-8-optional-whats-point> [Accessed at 10/03/2015]
- [26] Julien, P. (2014) Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM [Online]. Available at <http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html> [Accessed at 15/03/2015]
- [27] Mentor, (2014) One important change in Memory Management in Java 8 [Online]. Available at <http://karunsubramanian.com/websphere/one-important-change-in-memory-management-in-java-8/> [Accessed at 01/04/2015]