Java 8: Rebirth to a Software Supergiant

Kaushik N S Iyer
Student, Dept. of Information Science & Engg.
BMS College of Engineering
Bangalore, India

Gururaja H S
Asst. Professor, Dept. of Information Science & Engg.
BMS College of Engineering
Bangalore, India

***Abstract*— Java is one of the biggest software currently being implemented in many spheres of the software industry and also in many robotic technologies because of its highly methodical and structural object oriented approach. In this paper, the authors talk on the latest version of the language popularly known as Java 8 which has included many aspects of closure style programming related to Python. Some of such implementations like that of the lambda expression, functional interfaces, default methods, method references, streams, a thread safe implementation of the date class and a small introduction to Java 8 support to dynamic languages like Javascript is discussed in the paper.**

Keywords*— *Lambda Expressions, Functional Interfaces, Method references.

I. INTRODUCTION

Java 8 is one of the most awaited and featured release of this programming language. This version mainly was intended to push the language more towards Closure style of programming language. Like Python and C# which provide inbuilt interfaces to ease the work of developers; now Java 8 has also implemented such styles and hence pushing the language to new technological heights. Closure style of programming involves binding of data to function which has no name to reference. Some examples are lambda expressions, streams and map and filter functions on Collections.

JSR is an Acronym for Java Specification Requirement. They are the description and an outline of specifications for a Java version. Java 8 is popularly known as JSR 337 .This is an Open source implementation to show the community what the latest version has to offer. Even though there is an increase in the support of annotations with Java 8, it is not a new feature which has been bundled with the environment. Hence it is not explained in a detailed manner.

II. LAMBDA EXPRESSIONS

By far the most famous implementation in Java 8 is that of the lambda expressions. It is targeted to remove some redundant use of Anonymous Inner Classes such as when you are implementing ActionListeners or for functional interfaces such as comparator interfaces.

The most common syntax of a lambda expression is:

Variable_name = (Parameters) -> {expression body};

The JVM compiler checks the type of the variable name and then searches its interface for a method signature which matches the parameters in the lambda expression; it then replaces the default implementation of the interface with the expression body defined in the lambda expression. Compared to anonymous inner classes, this is very useful when you have functional interfaces like comparator interface which is used to sort a collection of any generic based on a parameter given in the compare function which in-turn returns the compareTo( ) of the two objects.

Consider a situation where we have to sort a set of consecutive integer pairs based on their leading or first numbers. In Java SE 7, we would have to do this with the help of an anonymous inner class like shown below:

```
Collections.sort(list, new Comparator<integers>() {
@Override
    public int compare(integers a, integers b) {
    return b.getN1().compareTo(a.getN1());
    }
    });
```

Where N1 represents the member of the class. Now consider we have to set an ActionListener to a button. We generally do it as follows:

```
button.addActionListener(new ActionListener() {
@Override
    public void actionPerformed(ActionEvent e) {
    return null;
    }
    });
```

In both the above examples, we have to repeat the inner class for every instance. In Java 8 we can reduce this as follows:

a) For the first example, the lambda expression can be written as:

```
Collections.sort(list, (integers a, integers b) -> b.getN1().compareTo(a.getN1()));
```

b) For the second button example, the lambda expression can be written as:

```
button.addActionListener((ActionEvent e) -> {return null; });
```

This is briefly instructed in Figure 1 given below:

Java SE 7	Java 8
Collections.sort(list, new Comparator<integers>() { @Override public int compare(integers a, integers b) { return b.getN1().compareTo (a.getN1()); } });	Collections.sort(list, (integers a, integers b) -> b.getN1().compareTo(a.get N1()));
button.addActionListener(n ew ActionListener() { @Override public void actionPerformed(ActionEve nt e) { return null; } });	button.addActionListener((ActionEvent e) -> {return null; });

Figure 1: Comparison between Java SE 7 and Java 8. Use of Lambda Expressions in Java 8

III. STREAMS AND PIPELINING

A Stream is a collection of endless instances of objects. A most common implementation of a stream is a List. Sometimes a stream can also be thought of as a product of a query which in most cases give a result set as a response.

In Java 8, a Collection or more specifically a list can be converted into a stream by calling the stream() method which returns a stream of instances of the same generic as that of the input matrix.

One major question would be as to why do we have to convert the Collection to a stream? A Simple explanation would be that when a Collection is converted to a stream, operations can be done simultaneously or parallel by the JVM.

In Java 8, two kinds of stream option exist and the advantage is that they can be interchanged:

• *Stream()* gives a simple stream considering a collection as a source.

• *ParallelStream()* is a major advancement in pipelining the stream to perform several filters at a superscalar rate.

Aggregating operations are filters or basically conditions based on which the elements of the collection can be rearranged or basically the data of the Collection can be converted to information and can also be collected in any structure like a string or even another collection so as to simplify the information processing.

a) *Map()* operation is a combination of a simple iterator and an arithmetic operation that can be expressed as a lambda expression.

As an example, take a list of numbers and store its squares into another list:

List <Object>intlist = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

List<Integer> sqnum=intlist.stream().map(i -> (Integer) i (Integer)i).collect(Collectors.toList());*

We see that instead of instantiating an iterator manually, a foreach loop will get implemented where instead of doing a manual addition of object to the list, a BiPredicate functional interface will be instantiated with the given lambda expression and the iterator will be run with the above BiPredicate on each element.

b) *Filter()* is just another way of applying a condition. Consider an example where you want 10 random numbers which are positive, even and less than 50.

In Java SE 7, you will have to run a while loop and also a collection to store the result. But in Java 8, the implementation is as follows:

IntStream integers = r.ints().filter (i->{return (i>0&& i<50); }).limit(15);

In the above example, an iterator loop will be setup and a Predicate Object will be instantiated with the filter expression as a lambda expression and the loop gets executed until the limit is reached.

c) *forEach* is another stream function which runs a for each loop or advanced for loop through all the elements in the Collection specified.Consider an example where you want to print the numbers present in the stream mentioned in the filter function as follows:

integers.forEach(System.out::println);

The parameter passed to the construct can be a lambda expression or a method reference like shown in the example. Method reference follows the syntax:

ClassName::MethodName

IV. NASHHORN JS ENGINE

Till Java SE & Java, the platform was shipped with Mozilla Rhino JS Engine. But with the advent of Java 8, it will be replaced with Nashhorn JS Engine. The following are some of its main features.

Oracle Nashhorn is ECMA Copliant. ECMA is a trademarked scripting specification which was initially developed by Sun Microsystems and ECMA International. According to the latest version of this standard which was released in june 2015, it includes new features such as iterators and for/if loops and programming constructs which are of a Python flavour like lambda functions and generator expressions, arrow functions, binary data and proxies.

Some interesting features with this new Extension is that it can run these scripts as a Java fx application and an Interesting Scripting mode can be enabled where system scripts can be written in Javascript.

The scripting mode mainly comprises of two language heredocs and shell invocation.

It is based on the Da Vinci machine. Based on the oracle Documentation, the Da Vinci project is referred to as JSR 292. It was developed by Sun Microsystems. Its main purpose is to ease the implementation of dynamic languages like Javascript

on top of JVM. Till Java SE 7, there was no support for dynamically programmable languages. Like Javascript, the JSR 292 adds a new "invokedyanmic" instruction at the JVM Level. With the help of JIT (Just In Time Comilation) where the bytecodes are verified prior to execution. Javascript can now be rendered seamlessly without having the fuss to verify multiple bytecodes.

V. DATE AND TIME CLASS

Along with the advent of Java 8, the Date class has undergone a tremendous amount of change. The first change is the introduction of zonal based Time and Local Time without no fuss of handling the various time zones. Hence the Local Time can be obtained by the class. Second change would be that the new LocalTime class is thread safe for which Java SE 7 had got a lot of backlashes.

LocalDateTime and the zonal time by the class ZonedDateTime and are initialized as follows:

• *LocalDateTime time1 = LocalDateTime.now();*

• *ZonedDateTime time2 = ZonedDateTime.parse ("the_zone_of_the_region");*

Each of the above class has separate set of functionalities because of the type of input given to its constructor. LocalDateTime class coupled with the LocalDate and LocalTime class can be used to query multiple parameters out of the LocalDateTime class like just the date part or the time which was a tedious task in the Date class which returned the entire date and Time string and the developer had to parse the string to get the required portion of the String. Consider the following example to obtain the current date and time:

a) Date Class:

Date d= new Date();

which would give the output as: Wed Mar 09 01:38:09 IST 2016

b) LocalDateTime Class:

LocalDateTime time1= LocalDateTime.now();

which would give the output as: 2016-03-09T01:38:10.381

As we can observe, the Date object gives the entire details but whereas the LocalDateTime is giving just the date and time separated by a „T". Now if we were to get just the day of week for today"s date, it would be given as:

a) Date Class:

System.out.println("Day: "+d.toString().substring(0, 3));

which would give the output as: Day: Wed which can be put under a string switch to get the complete name: "WEDNESDAY".

b) LocalDateTime Class:

LocalDate date1 = time1.toLocalDate();

System.out.println("Day: "+time1.getDayOfWeek());

which would give the output as: Day: WEDNESDAY. Without any additional functions required.

Along with the addition of LocalDateTime and its helper classes, Java 8 also encompasses a Class called ChronoUnit which is a enumerative way to store the data of the metrics required to calculate relative date and time based on the current time. Some examples of ChronoUnits are ChronoUnit.WEEKS and ChronoUnit.DECADES. Consider an example where we want to find the day of the week for the same date next but for next month. Then the syntax would be like follows:

a) Date Class:

You will have to get the current month with the help of the Date object and reset the month with the help of the SetMonth() function. The main problem is that both the above mentioned helper functions are deprecated for they are quite tedious to reinitialize the parameters at every instance:

Date d= new Date();

int month= d.getMonth();

d.setMonth(month+1);

System.out.println("Day: "+d.toString().substring(0, 3));

The Output will be: Day: Sat

b) In the LocalDateTime class, you will just have to update the time with the help of helper functions such as plus() with takes in ChronoUnits.

LocalDateTime time1= LocalDateTime.now();

time1 =time1.plus(1,ChronoUnit.MONTHS);

System.out.println("Day: "+time1.getDayOfWeek());

The Output will be: Day: SATURDAY

The above illustrated examples can be summarized in Figure 2 as follows:

Java SE 7	Java 8

Date d= new Date();
 	
System.out.println("Day: "+d.toString().substring(0, 3));
 	
int month= d.getMonth();
d.setMonth(month+1);
System.out.println("Day: "+d.toString().substring(0, 3));
	LocalDateTime time1 = LocalDateTime.now();
	
	System.out.println("Day: "+time1.getDayOfWeek());
	
	time1 = time1.plus (1,ChronoUnit.MONTHS);
	System.out.println("Day: "+time1.getDayOfWeek());

Figure 2: Comparison between Date class and LocalDateTime class

VI. CONCLUSION

The authors would like to conclude that the latest version of the Java language popularly known as Java 8 has included many aspects of closure style programming related to Python. This helps the implementations like lambda expressions, functional interfaces, default methods, method references and streams much more easily as spoken in the paper. The examples also suggest that Java 8 is easier to implement compared to Java SE 7.

REFERENCES

[1] Oracle Documentation [https://docs.oracle.com/javase/tutorial].
[2] Java 8 Tutorials [https://tutorialspoint.com].
[3] Head First Java by Cathy Sierra.