## Introduction to Functional Programming in Java 8

Java 8 is the current version of Java that was released in March, 2014. While there are many new features in Java 8, the core addition is functional programming with lambda expressions. In this section we describe the benefits of functional programming and give a few examples of the programming style. Most of the features in Java 8 are more appropriate for an advanced Java text but the concepts apply to material we have discussed, particularly when we are working with collections.

A lambda expression is a nameless function. In functional programming, a function is the same thing as a method. Related concepts include closures, anonymous functions, and function literals. As a nameless function, a lambda expression is essentially a little chunk of code that you can pass around as data but have it treated like a function with parameters. Lambda expressions provide a neat way to implement a class that normally has only one function and to make it easy to modify methods on the spot rather than go through the work of defining a method to perform a specialized task. Additionally, lambda expressions help Java parallelize itself to run more efficiently on multi-core or parallel machines. For example, normally we will process elements in an `ArrayList` by creating a `for` loop that accesses each element one by one. This is considered *external* access to the loop. In contrast, with lambda expressions we can *internally* iterate through the `ArrayList` by providing a function that tells Java how to process each element. The Java Virtual Machine can then parallelize the operating by farming computation on the elements to different processors.

The format to define a lambda expression looks like this:

```
parameters -> body
```

The arrow separates the parameters from the body. In many cases the body is short and just a single line of code. If it were longer, than a traditional method may make more sense. Here is a lambda expression with a function that takes no parameters and returns the number 68:

```
() -> { return 68; }
```

Here is a lambda expression that returns the sum of two integers `x` and `y`:

```
(int x, int y) -> { return (x+y); }
```

In many cases Java can infer the type of the parameters, in which case we can leave the data type off. We can also simply provide an expression on the right side and it automatically becomes the return value without requiring the keyword `return`. The following is equivalent to the previous example:

```
(x, y) -> x+y
```

As an example to motivate the use of lambda functions, consider the case where we want a class to implement the `Runnable` interface.The `Runnable` interface has only one method to implement, the `run()` method. If we are using threads then we'd invoke the `start()` method but in this case we can directly invoke the `run()`

method. The following code illustrates the traditional way we would create an object that implements Runnable:

```
public class OldStyleRunnable implements
                                Runnable
{
      public void run()
      {
            System.out.println
               ("Running in a class!");
      }
}

public class NotLambda1
{
      public static void main(String[] args)
      {
         OldStyleRunnable r0 = new
                        OldStyleRunnable();
         r0.run(); // Not running in a thread
      }
}
```

Sample Dialogue:

```
Running in a class!
```

This is fine for one object, but what if we wanted multiple objects, and we wanted different code in the `run()` method for each? Then we would have to explicitly create a separate class for each object. An alternative is to use an anonymous class in which we declare and instantiate the class in a single statement:

```
public class NotLambda2
{
      public static void main(String[] args)
      {
        // Anonymous class that overrides
        // the run() method
        Runnable r = new Runnable()
        {
          public void run(){
            System.out.println
               ("In an anonymous class!");
          }
        };
        r.run();
      }
}
```

Sample Dialogue:

```
In an anonymous class!
```

This is an improvement over the first version because we can now create unique `Runnable` objects with the `run()` method of our choice without the need to

assign a name to derived `Runnable` class. However, lambda functions allow us to assign a function to a `Runnable` object in a single line:

```
public class LambdaRunnable
{
      public static void main(String[] args)
      {
        Runnable r =
            () -> System.out.println
               ("In a lambda expression!");
       r.run();
      }
}
```

Sample Dialogue:

```
In a lambda expression!
```

The lambda format is the simplest of all and lets us directly insert the method where needed. The same concept applies to implementing an `actionListener` for a GUI component. For example, instead of this old style code that uses an anonymous class:

```
button.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    System.out.println("You clicked me!");
  }
});
```

we can now use the much shorter and easier to read:

```
button.addActionListener
     (e -> System.out.println
           ("You clicked me!"));
```

Java's lambda expressions are particularly useful when applied to collections. Three common operations that we typically perform are to *filter*, *map*, or *reduce* the collection. In this section we give a short example of each.

Let's start with the concept of a filter. Consider the following code, which creates a list of doubles:

```
ArrayList<Double> nums = new ArrayList<>();
nums.add(3.5);
nums.add(56.3);
nums.add(81.1);
nums.add(4.8);
```

If we only want to output the values in the array that are over 50 then in traditional Java-style (external processing) we would make a loop with an if statement:

```
for (int i = 0; i < nums.size(); i++)
  if (nums.get(i) > 50)
     System.out.println(nums.get(i));
```

Using Java 8's lambda expressions we can do the same thing by creating a stream of the elements in the `ArrayList` and then filtering them. This is accomplished through a sequence of function calls:

```
nums.stream().filter((Double val) -> val > 50).forEach((Double
val) -> System.out.println(val));
```

For readability purposes it is common to put each function call on a separate line:

```
nums.stream()
    .filter((Double val) -> val > 50)
    .forEach((Double val) -> System.out.println(val));
```

The `stream()` method creates a stream which generates a list that we can iterate once. Not to be confused with data streams, this new type of stream can be accessed in parallel or sequentially. In our case we are only using sequential streams. Once the stream is generated then we invoke `filter` and the `forEach`. Inside `filter` we specify a lambda expression. Each element in the `ArrayList` is filtered according to the lambda expression. In this case, the variable `val` is an element in the `ArrayList` that is being processed and the function says to filter only those elements whose value is greater than 50. Next, the `forEach` iterates through the filtered elements and outputs each one via `println`. In our example, this would output 56.3 and 81.1.

We can simplify the code a little bit more by leaving out the data type because Java is able to infer it from the context. The resulting code becomes:

```
nums.stream()
    .filter(val -> val > 50)
    .forEach(val -> System.out.println(val));
```

The new format is quite different than the traditional method but the style is more concise, can be more easily parallelized, and in general will require less code than the old technique.

Next, consider the concept of a map. A map takes elements in the collection and transforms them in some way. First, consider a simple mapping where we would like to add 100 to every element in the `ArrayList`. We can do so as follows:

```
nums.stream()
  .map(val -> (val + 100))
  .forEach(val -> System.out.println(val));
```

This will output 100 added to each value (i.e. 103.5, 156.3, 181.1, 104.8). Note that each function is invoked in sequence. If we add our previous filter to the beginning then we would only get 156.3 and 181.1:

```
nums.stream()
  .filter(val -> val > 50)
  .map(val -> (val + 100))
  .forEach(val -> System.out.println(val));
```

Finally, consider the concept of collecting. Collecting means that we process all of our elements in some way and collect the final result. The result is often a single value. Examples include summing, averaging, finding the minimum, or finding the maximum of a set of data. The following code shows how we could compute the sum of all elements in our `ArrayList`:

```
double d = nums.stream()
  .mapToDouble(v -> v)
  .sum();
```

```
System.out.println("The sum is " + d);
```

The `mapToDouble` function takes each element and maps it as a `double` (a bit redundant here since we are starting with `double`s) and then accumulates them into a sum. As you might surmise, there are also the methods `mapToInt()`, `mapToLong()`, etc. and methods to compute `min()`, `max()`, `average()`, and other values.

More customization is possible using the `reduce` function. In our case we'll use the version that takes as input a seed value and a binary function. Consider a collection with values *v1*, *v2*, and *v3*. If we start with a seed value *s*, then reduce will first apply the binary function to *s* and *v1*, producing *r1*. The binary function is then applied with *r1* and *v2*, producing *r2*. Then the binary function applies *r2* and *v3*, producing *r3* which is returned as the final value. The following code computes the sum of all values using `reduce`:

```
d = nums.stream()
  .reduce(0.0, (v1, v2) -> v1 + v2);
System.out.println("The sum is " + d);
```

In this case, 0.0 is the seed value and the second parameter is the function that specifies how to accumulate the sum of the value. For the first step, `v1` corresponds to 0.0 and `v2` corresponds to 3.5. This produces the intermediate sum of 3.5. In the second step, `v1` corresponds to 3.5 and `v2` corresponds to 56.3 to produce 59.8. In the third step, `v1` corresponds to 59.8 and `v2` to 81.1, and so on until the sum is produced.

For an additional example, consider the following list of names:

```
ArrayList<String> names = new ArrayList<>();
names.add("Paco");
names.add("Enrique");
names.add("Bob");
```

If we want to compute the average length of all names then we could map the length to an integer:

```
d = names.stream()
    .mapToInt(name -> name.length())
    .average()
    .getAsDouble();
System.out.println("The average is " + d)
```

In this case we map each name to an `int` using the `length()` method, compute the average, and get the value as a `double`.

For the final example, say that we want to get the largest name. We can use the reduction technique:

```
String s = names.stream()
     .reduce("", (n1, n2) ->
             {
               if (n1.length() > n2.length())
                     return n1;
               else
                     return n2;
             }
            );
System.out.println("longest Name: " + s);
```

We use a block in this case where the function compares the length of the strings and returns the largest one. This is one case where we would commonly use the `?` operator to shorten the code:

```
String s = names.stream()
     .reduce("", (n1, n2) ->
          (n1.length() > n2.length()) ? n1 : n2);
System.out.println("longest Name: " + s);
```

These examples should give you an idea of what Java lambda expressions look like and what they can do. While there is definitely a learning curve, lambda expressions will allow you to write code that is more concise while enabling parallel processing. Java 8's new syntax supports both functional programming and object-oriented programming in a way that reaps the benefits of both styles.