Lambda Expressions in Java 8: Why and How to Use Them

The role of lambda expressions in Java 8 in applying functional programming constructs in a pure Object-Oriented programming language.

Using lambda expression, sequential and parallel execution can be achieved by passing behavior into methods. In the Java world, lambdas can be thought of as an anonymous method with a more compact syntax. Here compact means that it is not mandatory to specify access modifiers, return type and parameter types while defining the expression.

Why Lambdas in Java

There are various reasons for addition of lambda expression in Java platform but the most beneficial of them is that we can easily distribute processing of collection over multiple threads. Prior to Java 8, if the processing of elements in a collection had to be done in parallel, the client code was supposed to perform the necessary steps and not the collection. In Java 8, using lambda expression and Stream API we can pass processing logic of elements into methods provided by collections and now collection is responsible for parallel processing of elements and not the client.

Also, parallel processing effectively utilizes multicore CPUs used nowadays.

About Lambda Expression

Syntax:
(parameters) -> expression
or
(parameters) -> { statements; }

Java 8 provide support for lambda expressions only with functional interfaces. Functional Interface is also a new feature introduced in Java 8. Any Interface with single abstract method is called Functional Interface. Since there is only one abstract method, there is no confusion in applying the lambda expression to that method.

Benefits of Lambda Expression

1. Fewer Lines of Code

One of the benefits of using lambda expression is the reduced amount of code. See the example below.

```
// Using Anonymous class
Runnable r1 = new Runnable() {
   @Override
   public void run() {
      System.out.println("Prior to Java 8");
   }
};

// Runnable using lambda expression
Runnable r2 = () -> {
   System.out.println("From Java 8");
};
```

- We know that in Java lambda can be used only with functional interfaces. In the above example, Runnable is a functional interface, so we can easily apply lambda expression here
- In this case, we are not passing any parameter in lambda expression because the run() method of the functional interface (Runnable) takes no argument.
- Also, the syntax of the lambda expression says that we can omit curly braces ({}) in case of a single statement in the method body. In case of multiple statements, we should use curly braces as done in the above example.

2. Sequential and Parallel Execution Support by passing behavior in methods

Prior to Java 8, processing the elements of any collection could be done by obtaining an iterator from the collection and then iterating over the elements and then processing each element. If the requirement is to process the elements in parallel, it would be done by the client code. With the introduction of Stream API in Java 8, functions can be passed to collection methods and now it is the responsibility of collection to process the elements either in a sequential or parallel manner.

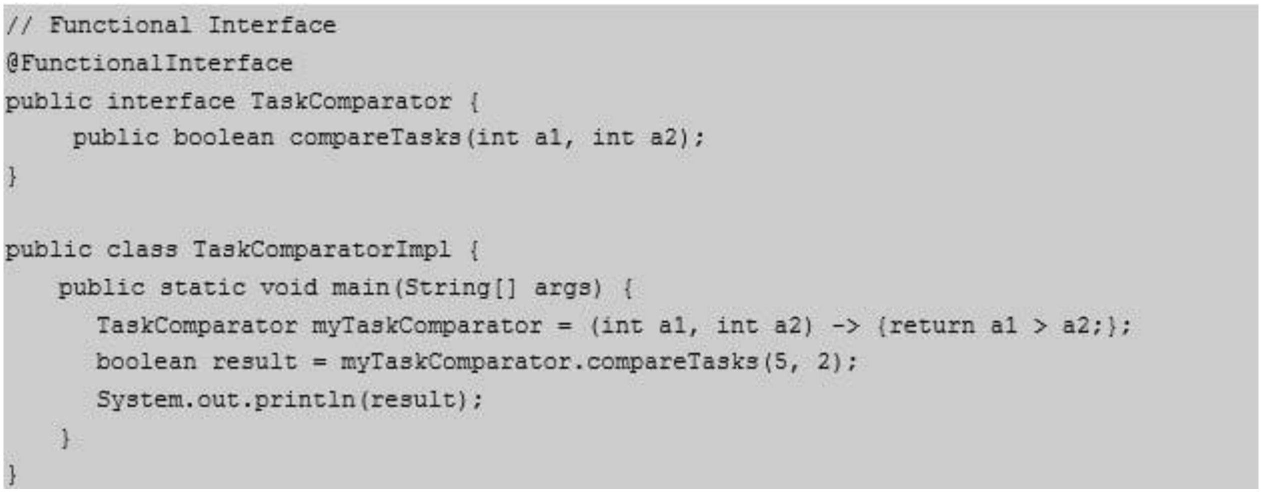
```
// Using for loop for iterating a list
public static void printUsingForLoop(){
   List<String> stringList = new ArrayList<String>();
   stringList.add("Hello");
   for (String string : stringList) {
        System.out.println("Content of List is: " + string);
   }

}

// Using forEach and passing lambda expression for iteration
public static void printUsingForEach() {
   List<String> stringList = new ArrayList<String>();
   stringList.add("Hello");
   stringList.stream().forEach((string) -> {
      System.out.println("Content of List is: " + string);
   });
}
```

3. Higher Efficiency (Utilizing Multicore CPU’s)

Using Stream API's and lambda expression we can achieve higher efficiency (parallel execution) in case of bulk operations on collections. Also, the lambda expressions can help in achieving internal iteration of collections rather than external iteration as shown in the above example. As nowadays we have CPUs with multicores, we can take advantage of these multicore CPU's by parallel processing of collections using lambda.

Lambda Expression and Objects

In Java, any lambda expression is an object as is an instance of a functional interface. We can assign a lambda expression to any variable and pass it like any other object. See the example below on how a lambda expression is assigned to a variable, and how it is invoked.



```java
// Functional Interface
@FunctionalInterface
public interface TaskComparator {
     public boolean compareTasks(int a1, int a2);
}

public class TaskComparatorImpl {
    public static void main(String[] args) {
       TaskComparator myTaskComparator = (int a1, int a2) -> {return a1 > a2;};
       boolean result = myTaskComparator.compareTasks(5, 2);
       System.out.println(result);
    }
}
```

Where you can use Lambda expressions

Lambda expressions can be used anywhere in Java 8 where we have a target type. In Java, we have target type in the following contexts

- Variable declarations and assignments
- Return statements
- Method or constructor arguments

What do you think about this new feature in Java 8? Leave your thoughts in the comments section of this post. In our next post, we will discuss about the new date and time API in Java 8.