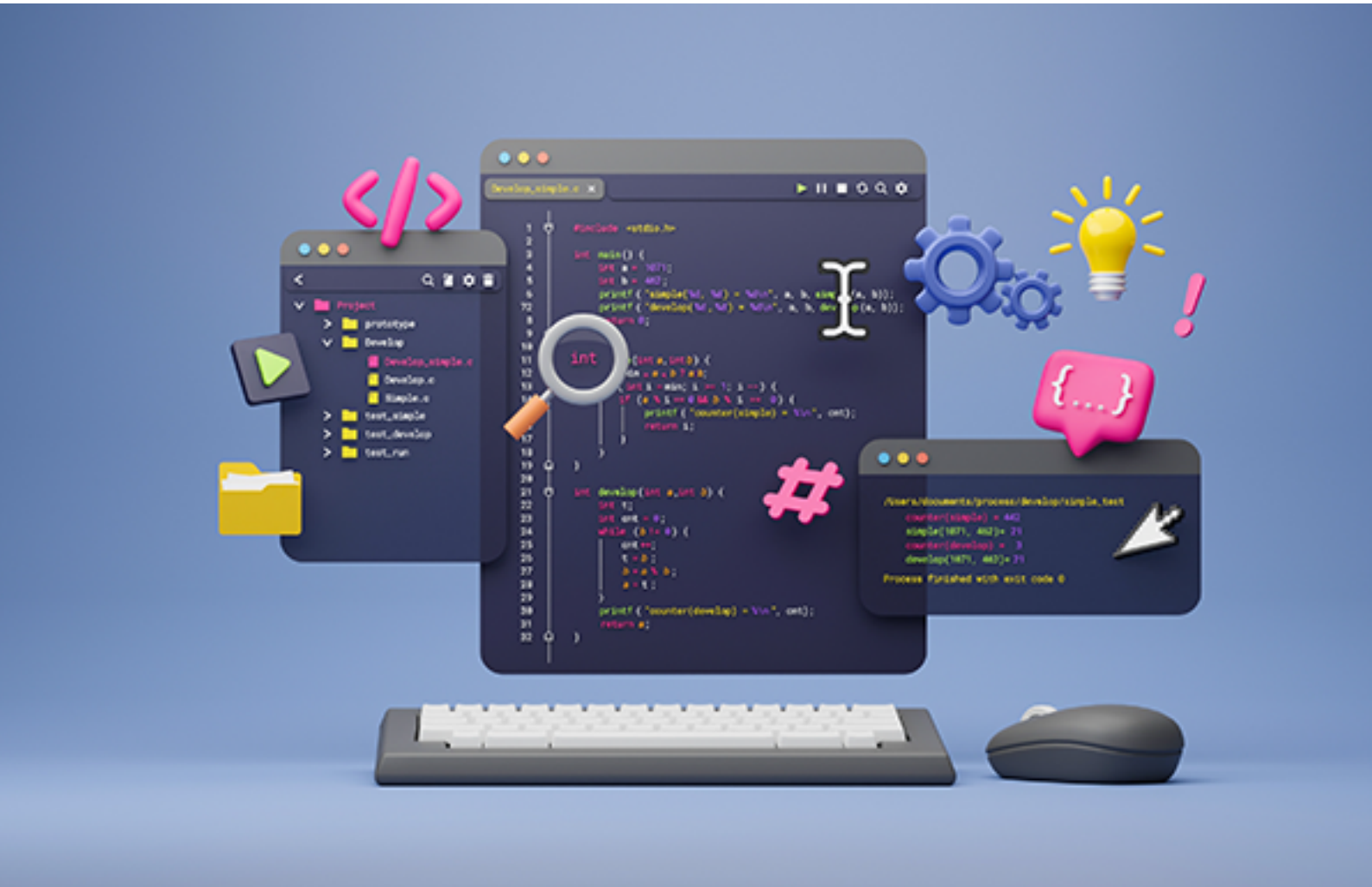
Whitepaper

JAVA 8 – Lambda Expressions for SDETs

JADE GLOBAL™

www.jadeglobal.com info@jadeglobal.com +1-408-899-7200

Contents



Introduction

Lambda expression is a new feature introduced in Java 8. It has brought functional programming capabilities to JAVA and is one of the most popular features of Java 8. Lambda expressions are used to create instances or objects of functional interfaces. A functional interface is an interface with a single abstract method (we already have interfaces like Runnable, Callable, ActionListener, etc.). A lambda expression can be expressed as an anonymous function with no name and doesn't belong to any class.

We can directly write an implementation for a method by using lambda expressions. In this case, the compiler doesn't create a specific class file for these expressions; it executes lambda expressions as a function. We can also manipulate collections such as performing iteration, extracting and filtering data, etc.

Syntax of Lambda Expression

To create a lambda expression, input parameters (if any) need to be specified on the left side of the lambda operator "→" and place the expression on the right side of the lambda operator.

(Parameters) → (Expression);

For Example, the lambda expression (x, y) → x + y specifies that lambda expression takes two arguments x and y and returns the sum.

Before Java 8

Interface with an abstract method

```
public interface add {
   void sum(int a, int b); //abstract method
}
```

To execute the abstract method, created a class and implemented the interface

Output:
The sum is:: 30

```
public class TestExecution implements add {
   @Override
   public void  sum(int a, int b) {
      int c = a+b;
      System.out.println("Sum is :: "+c);
   }
public static void main(String[] args) {
      TestExecution ts = new TestExecution();
      ts.sum(10, 20);
   }
}
```

```
@FunctionalInterface
public interface add {
   void sum(int a, int b);
}
public class TestExecution{
   public static void main(String[] args) {
      add a = (x, y) -> {
         System.out.println("Sum is :: "+(x+y));
      };
      a.sum(10, 20);
   }
}
```

After Java 8, using Lambda Expression

Interface with Functional Interface annotation

Output:
The sum is:: 30

Note:

1. Functional Interface annotation is used to create lambda expressions.

2. Functional Interface annotation allows a single abstract method. If another abstract method/non-abstract method is created, it throws a compilation error in the imple mentation, and lambdas are not supported.

Lambda Expressions are of Single Line and Multi-Lines

If it is a multi-line lambda, then we can use braces and ends with a semi-colon as shown below

```
add a = (x, y) -> {
                System.out.println("Sum is :: "+c);
    };
```

If it is a **single line lambda**, then as shown below,

```
add a = (x, y) -> System.out.println("Sum is :: "+ (x+y));
```

Here (**x,y**) is the **parameter** and **System.out.println**("**Sum is :: "**+ (**x+y**)) is the **expression**

Handling multiple parameters using Lambda

Interface with a single abstract method having multiple parameters

```
@FunctionalInterface
public interface EmployeeDetails {
   void empDetails(String firstName, String lastName, int age);
}
public class TestExecution{
   public static void main(String[] args) {
      EmployeeDetails emp = (fn, ln, age) -> {
        System.out.println("Name :: "+fn+" "+ln);
        System.out.println("Age :: "+age);
      };
      emp.empDetails("John", "Watson", 40);
   } }
```

Output:

Name:: John Watson

Age:: 40

Note:

1. The code in the bold letters is a lambda expression having multiple parameters.

Returning Values from Lambda

Interface with the abstract method of String return type

```
@FunctionalInterface
public interface EmployeeDetails {
   String empDetails(String firstName);
}
public class TestExecution{
   public static void main(String[] args) {
      EmployeeDetails emp = (fn) -> {
         return fn.toUpperCase();
      };
            System.out.println(emp.empDetails("Selenium"));
   } }
```

Output:

SELENIUM

The above Example returns the value according to the function mentioned in the Lambda.

As the above Example is a single line lambda with a return type, we can remove the

return keyword and write lambda expression as,

```
EmployeeDetails emp = (fn) -> fn.toUpperCase();
```

Functional Programming

Functional Programming is a way of building software by using pure functions and avoiding shared state and side effects. The shared state is a variable being accessed by more than one function, and side effects are nothing but a function modifies the variable.

f(x) = x+3

x is the input, and x+3 is the output. This is nothing but a lambda expression.

For Example,

```
EmployeeDetails emp = (fn) -> fn.toUpperCase();
```

Functional Programming – Key Concepts:

- Pure Function
- Function as the first-class object
- Higher-Order functions

Pure function: Consistent, predictable output for the given input

f(x) = x+3

This is a pure function because, for a given input, the output is consistent and known.

If x = 3, output is 6

If x = 5, output is 8

The below lambda expression is an example of pure function. The output is based on the given input.

Input – Selenium

Output - SELENIUM

```
EmployeeDetails emp = (fn) -> fn.toUpperCase();
```

For example,

f(x) = x+3+y

This is an impure function because the output is based on the outside variable 'y.'

If x = 3, the output is 6 + y. The output is inconsistent and depends on 'y' variable, which is not part of the function.

The below code is an example of an impure function:

```
@FunctionalInterface
public interface EmployeeDetails {

   String empDetails(String firstName);
}
public class TestExecution{

   public static void main(String[] args) {

      List<String> list = new ArrayList<>();
      list.add("Java");
      list.add("Linux");

      EmployeeDetails emp = (fn) -> {
        return fn.toUpperCase() + list.get(1);
      };
      System.out.println(emp.empDetails("Selenium"));
   }
}
```

Output:

SELENIUMLinux

The output is inconsistent and is dependent on the list object. If we change the index of the get() method in the above Example, then the output will be different.

Function as a first-class object:

For Example,

Interface with the abstract method without return type

```
public interface EmployeeDetails {

   void empDetails(String firstName);
}
```

```
public class TestExecution implements EmployeeDetails{

   @Override
   public void empDetails(String firstName) {
      System.out.println(firstName.toUpperCase());
   }

   public static void main(String[] args) {

      TestExecution ts = new TestExecution();
      ts.empDetails("Selenium");
   }
}
```

Output:

SELENIUM

In the above Example, to execute the abstract method, we are implementing the interface and then calling the method with the help of a class object as reference.

This is how we use to execute before Java 8.

Let's see the same Example by using Functional Interface Annotation

```
@FunctionalInterface
public interface EmployeeDetails {

   void empDetails(String firstName);
}
```

```
public class TestExecution{

   public static void main(String[] args) {

      EmployeeDetails emp = (s) -> System.out.println(s.toUpper-
Case());
      emp.empDetails("Selenium");
   }
}
```

Output:

SELENIUM

In the above Example, the function 'System.out.println(s.toUpperCase()' is assigned to a variable 'emp,' and this variable acts like a reference type.

Function as a first-class object is an ability to assign a function to a variable and having a reference of function.

Higher-Order Functions

A method that receives another function as a parameter is called a higher-order function.

Let us see with an example,

Output:

PRAMOD

Here, the method test() is a higher-order function, as it has received the reference as a function. The above test() method can also be written as,

```
@FunctionalInterface
public interface EmployeeDetails {

   String empDetails(String firstName);
}
public class TestExecution{

   public static void main(String[] args) {
      EmployeeDetails emp = (s) -> s.toUpperCase();
      test(emp);
   }

   private static void test(EmployeeDetails emp) {
      String name = emp.empDetails("Pramod");
      System.out.println(name);
   }
}
```

```
public class TestExecution{

   public static void main(String[] args) {
      test(s -> s.toUpperCase());
   }

   private static void test(EmployeeDetails emp) {
      String name = emp.empDetails("Pramod");
      System.out.println(name);
   }
}
```

Output:

PRAMOD

We can see, the method is accepting the parameter as a function. In this way, we can call the method with different functions as a parameter. Before Java 8, this was not

possible, where we used to write a method with other variables as a parameter.

```
public class TestExecution{

   public static void main(String[] args) {
      test(s -> s.toUpperCase());
      test((a) -> String.valueOf(a.length()));
   }
   private static void test(EmployeeDetails emp) {
      String name = emp.empDetails("Pramod");
      System.out.println(name);
   }
}
```

Output:

PRAMOD

6

How is higher-order function different from methods in Java

Let's create a method for converting a string to uppercase.

```
public class TestExecution {

   public static void main(String[] args) {

      convertUppercase("Selenium");
   }
   private static void convertUppercase(String s) {

      String str = s.toUpperCase();
      System.out.println(str);
   }
}
```

From the above Example, it is clear that the behavior is given inside a method, and the variable is passed as a reference. If we want to change the behavior, we need to create another method.

```
public class TestExecution {

   public static void main(String[] args) {

      convertUppercase("Selenium");
      convertLowercase("Java");
   }

   private static void convertUppercase(String s) {
         String str = s.toUpperCase();
         System.out.println(str);
      }
      private static void convertLowercase(String s){
         String str = s.toLowerCase();
         System.out.println(str);
      }
   }
```

Here, we have created another method with behavior where it converts a String to lowercase. And to execute, we are passing a variable as a reference. Let's create a higher-order function

```
@FunctionalInterface
public interface ConvertString {
   String convertString(String s);
}
```

```
public class TestExecution {
   public static void main(String[] args) {
      test(s->s.toUpperCase());
   }
   private static void test(ConvertString s){
     String str = s.convertString("Selenium");
      System.out.println(str);
   }
}
```

In higher-order function, the variable is present inside the method, and the behavior is passed as a reference. If we want to execute the method with another behavior, we need not create another method; instead, we can pass the behavior as a reference.

```
public class TestExecution {
   public static void main(String[] args) {
      test(s->s.toUpperCase());
      test(s->s.toLowerCase());
   }
   private static void test(ConvertString s){
     String str = s.convertString("Selenium");
      System.out.println(str);
   }
}
```

This is how higher-order function is different from the methods in Java.

Higher-Order Function returns Lambda

Higher-order function not only accepts the function as a reference but also returns a lambda.

```
@FunctionalInterface
public interface ConvertString {
   String convertString(String s);
}
```

```
public class TestExecution {
   public static void main(String[] args) {
      String str = test().convertString("data");
      System.out.println(str);
   }
   private static ConvertString test()
   {
      ConvertString cs = (s) -> s.toUpperCase();
      return cs;
   }
}
```

Here the test() method returns the lambda expression. We can also write the return statement as:

```
public class TestExecution {

   public static void main(String[] args) {

      String str = test().convertString("data");
      System.out.println(str);
   }

   private static ConvertString test()
   {
      return (s) -> s.toUpperCase();
   }
}
```

In a single line, the lambda expression is returned to the test() method.

Method References

Method references are another way of writing lambda expressions; usually, it is beneficial for single line lambda expressions, which call existing methods. It is used to refer to the method of functional interface. The main advantage of using method references is it is easy to read and compact.

For Example

```
@FunctionalInterface
public interface StringOperations {
   void accept(String s);
}
```

```
public class TestExecution {
   public static void main(String[] args) {
      StringOperations op1 = (s)-> System.out.println(s);
      op1.accept("data");
   }
}
```

Output:

data

The above code is an example of a standard way of creating a lambda expression. As mentioned before, Method reference is another way of writing a lambda expression. Whatever input is given, the output is printed by using the variable's'. And we know System.out.println is a static method because println can be called without creating an object. So, here we are trying to invoke another method to use the variable's'. Method references are used only for single lambda expressions. So, we can rewrite the lambda expression as:

```
public class TestExecution {

   public static void main(String[] args) {
      StringOperations op1 = System.out::println;
      op1.accept("data");
   }
}
```

Java understands that the given functional interface will accept only one variable, and System.out::println is a lambda expression using Method reference.

Let's see how we can use method references by referring to static methods, instance methods, and the constructor.

Higher-Order Function returns Lambda

1. Reference to a static method
2. Reference to an instance method
3. Reference to a constructor

1. Reference to a Static Method

Syntax:

ClassName::StaticMethodName

Example 1

```
@FunctionalInterface
public interface StringOperations {
   void accept(String s);
}
```

```
public class TestExecution {
   public static void stringDisplay(String s){
      System.out.println("String Displayed :: "+s);
   }
   public static void main(String[] args) {
      StringOperations op1 = TestExecution::stringDisplay;
      op1.accept("data");
   }
}
```

Output:

String Displayed:: data

In the above Example, we have defined a functional interface and referred to the static method 'stringDisplay()' to its functional method 'accept().'

The static method is called by its class name, and the lambda expression is written in a method reference format. The lambda expression behavior is stored in the 'op1' object, and by using this object, we are calling the accept() method.

Example 2

```
public class TestExecution {
   public static int add(int a, int b){
      return a+b;
   }
   public static void main(String[] args) {
      BiFunction<Integer, Integer, Integer> sum = TestExecution::add;
      int result = sum.apply(10, 20);
      System.out.println("Sum is :: "+result);
   }
}
```

Output:

The sum is:: 30

In this Example, we have used a predefined functional interface, i.e., BiFunction, and by

using its apply() method, we can execute the lambda expression behavior.

2. Reference to an instance method

Syntax:

ClassObjectName::InstanceMethodName

Example 1

```
@FunctionalInterface
public interface StringOperations {
   void accept(String s);
}
```

```
public class TestExecution {
   public void stringDisplay(String s){
      System.out.println("String Displayed :: "+s);
   }
   public static void main(String[] args) {
      TestExecution ts = new TestExecution();

      StringOperations op1 = ts::stringDisplay;
      op1.accept("data");

      StringOperations op2 = new TestExecution()::stringDisplay;
      op2.accept("java");
   }
}
```

Output:

String Displayed:: data

String Displayed:: Java

In the above Example, we have defined a functional interface and referring the non-static method 'stringDisplay()' to its functional method 'accept().'

The non-static method is called by its class object, and the lambda expression is written in a method reference format. The other way of doing it is by directly calling the class's

instance in the lambda expression.

Example 2

```
public class TestExecution {

   public int add(int a, int b){
      return a+b;
   }

   public static void main(String[] args) {
      BiFunction<Integer, Integer, Integer> sum = new TestExecution()::add;
      int result = sum.apply(10, 20);
      System.out.println("Sum is :: "+result);
   }
}
```

Output:

The sum is:: 30

In the above Example, we have used a predefined functional interface, i.e., BiFunction, and by using its apply() method, we can execute the behavior of the lambda expression.

As the add() method is a non-static method, it is called by the instance of the class to write a lambda expression.

3. Reference to a constructor

Syntax:

Classname/ConstructorName::new

```
@FunctionalInterface
public interface StringOperations {
   TestExecution accept(String s);
}
```

```
public class TestExecution {

   public TestExecution(String s){
      System.out.println("String Displayed :: "+s);
   }
   public static void main(String[] args) {
      StringOperations op1 = TestExecution::new;
      op1.accept("data");
   }
}
```

Output:
data

In the above Example, we can refer to a constructor by using the new keyword. The constructor is directed with the help of a functional interface.

Miscellaneous Examples

Concatenating two Strings

```
@FunctionalInterface
public interface StringOperations {
   String accept(String s1, String s2);
}
```

```
public class TestExecution {
   public static void main(String[] args) {
      StringOperations op1 = String::concat;
      System.out.println(op1.accept("Ja","va"));
   }
}
```

Output:

Java

In the above Example, we have defined a functional interface where it accepts two Strings and returns a String. The lambda expression is a reference to a static method type

of method reference. The static method 'concat' is called by its class name 'String.'

Conclusion

With lambda expressions, the objective is to minimize many of the disadvantages of using a single class or anonymous inner class when implementing a functional interface while at the same time maximizing the advantages. If you look at the above examples implemented using lambda expressions in Java, there's no denying that the code looks much more concise and compact.

These simple examples of lambda expressions show a few of their advantages in Java compared to other approaches. As you explore more advanced levels, you will find how they make iterative processing easier using the new, functional 'forEach' method. And because lambda expressions can be assigned to a variable, they provide an opportunity for reuse, which simply isn't possible with anonymous inner classes.

References

1. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

2. https://www.geeksforgeeks.org/lambda-expressions-java-8/

3. https://www.javatpoint.com/java-lambda-expressions

Join in

Partner with Jade
Let us guide your path amidst the clouds and help you build the future enterprise.

Be an expert
Become a master in the interplay of domains and industries. Drive higher benchmarks of success.

A leader
Knowledge powerhouse we all hold together-clients, partners and Jade experts.

Contact us to know more about Our Industry Solution

For more information

USA I CANADA I UK I AUSTRIA I INDIA

www.jadeglobal.com info@jadeglobal.com +1-408-899-7200

1731 Technology Drive, Suite 350 San Jose, CA 95110