Java Lambda Expressions

Java is a first-class object-oriented language. With the exception of primitive data types, everything in Java is an object. Even an array is an Object. Every class creates instances that are objects. There is no way of defining just a function / method which stays in Java all by itself. There is no way of passing a method as argument or returning a method body for that instance.

Since the old days of Swing, we always had written anonymous classes if we wanted to pass some functionality to any method. For example the old event listener code used to look like:

someObject.addMouseListener(new MouseAdapter() {

public void mouseClicked(MouseEvent e) {

//Event listener implementation goes here...

}

});

Here we wanted to add some custom code to Mouse listener. We defined an anonymous inner class MouseAdapter and created its object. This way we passed some functionality to addMouseListener method.

In short, it is not easy to pass plain methods / functionalities in Java that can be passed as arguments. Due to this limitation Java 8 adds a brand new language level feature called **Lambda Expressions**

Functions are first class citizens in a functional programming language. They exists on their own. You can assign them to a variable and pass them as arguments to other functions. JavaScript is one of the best example of an FP language.

Lambda expression adds that missing link of functional programming to Java. Lambda expression let us have functions as first class citizen. In languages that support first class functions, the type of the lambda expression would be a function; but in Java, the lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface.

A lambda expression is an anonymous function. Simply put, it’s a method without a declaration, i.e., access modifier, return value declaration, and name.

It’s a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

Lambda expressions in Java is usual written using syntax (argument) -> (body). For example:

(arg1, arg2...) -> { body }

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

Following are some examples of Lambda expressions.

(int a, int b) -> { return a + b; }

() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

## Structure of Lambda Expressions

Let’s check the structure of lambda expressions.

* A lambda expression can have zero, one or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
* Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
* Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a
* The body of the lambda expressions can contain zero, one or more statements.
* If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

## What are Functional Interfaces

In Java, a Marker interface is an interface with no methods or fields declaration. In simple words, marker interface is an empty interface. Similarly, a Functional Interface is an interface with just one abstract method declared in it.

java.lang.Runnable is an example of a Functional Interface. There is only one method void run() declared in Runnable interface. Similarly ActionListener interface is also a Functional Interface. We use Anonymous inner classes to instantiate objects of functional interface. With Lambda expressions, this can be simplified.

Each lambda expression can be implicitly assigned to one of the interface called Functional interface. For example we can create Runnable interface’s reference from lambda expression like below:

Runnable r = () -> System.out.println("hello world");