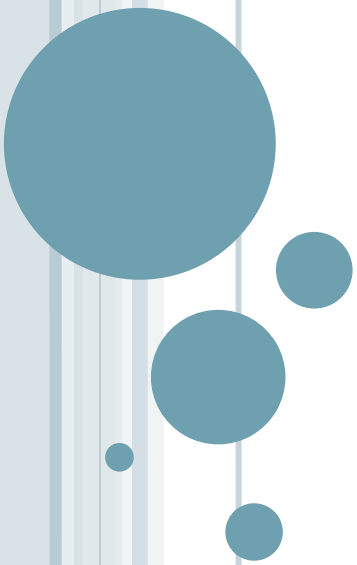
LAMBDA EXPRESSION



LAMBDA EXPRESSIONS

- The lambda expression was introduced first time in Java 8. Its main objective to increase the expressive power of the language.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

LAMBDA EXPRESSIONS

- **What is Functional Interface?**

If a Java interface contains one and only one abstract method then it is termed as functional interface. This only one method specifies the intended purpose of the interface.

For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

LAMBDA EXPRESSIONS

- **Define a Functional Interface in java**
- import java.lang.FunctionalInterface;
- @FunctionalInterface
- public interface MyInterface{
- // the single abstract method
- double getValue();
- }
- Here, we have used the annotation @FunctionalInterface. The annotation forces the Java compiler to indicate that the interface is a functional interface. Hence, does not allow to have more than one abstract method

LAMBDA EXPRESSIONS

- Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.
- **How to define lambda expression in Java?**

- (parameter list) -> lambda body

LAMBDA EXPRESSIONS

- Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.
- **Java Lambda Expression Syntax**

- (argument-list) -> {body}
- Java lambda expression is consisted of three components.

- 1) Argument-list: It can be empty or non-empty as well.

- 2) Arrow-token: It is used to link arguments-list and body of expression.

- 3) Body: It contains expressions and statements for lambda expression.

LAMBDA EXPRESSIONS

- Following are the important characteristics of a lambda expression.

- **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.

- **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.

- **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.

- **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

LAMBDA EXPRESSIONS

- **No Parameter Syntax**
- **() -> {**
- **//Body of no parameter lambda**
- **}**

- **One Parameter Syntax**
- **(p1) -> {**
- **//Body of single parameter lambda**
- **}**

- **Two Parameter Syntax**
- **(p1,p2) -> {**
- **//Body of multiple parameter lambda**
- **}**

LAMBDA EXPRESSIONS

- **Suppose, we have a method like this:**
- **double getPiValue() {**
- **   return 3.1415;**
- **}**

- **We can write this method using lambda expression as:**

- **() -> 3.1415**

LAMBDA EXPRESSIONS

- **Here, the method does not have any parameters.**
- **Hence, the left side of the operator includes an empty parameter.**
- **The right side is the lambda body that specifies the action of the lambda expression. In this case, it returns the value 3.1415.**

LAMBDA EXPRESSIONS

- **Types of Lambda Body**

- **In Java, the lambda body is of two types.**

- **1. A body with a single expression**

- **() -> System.out.println("Lambdas are great");**

- **This type of lambda body is known as the expression body.**

LAMBDA EXPRESSIONS

- **2. A body that consists of a block of code.**

- **() -> {**
- **   double pi = 3.1415;**
- **   return pi;**
- **};**

- **This type of the lambda body is known as a block body. The block body allows the lambda body to include multiple statements. These statements are enclosed inside the braces and you have to add a semi-colon after the braces.**

```
interface MyInterface{

    // abstract method
    double getPiValue();
}

public class LambadaEx1 {

    public static void main( String[] args ) {

    // declare a reference to MyInterface
    MyInterface ref;

    // lambda expression
    ref = () -> 3.1415;

    System.out.println("Value of Pi = " + ref.getPiValue());
    }
}
```

LAMBDA EXPRESSIONS WITH PARAMETERS

```
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

MULTIPLE PARAMETERS

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

MULTIPLE STATEMENTS

```
interface Sayable{
    String say(String message);
}

public class LambdaExpressionExample8{
    public static void main(String[] args) {

        // You can pass multiple statements in lambda expression
        Sayable person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
            return str2;
        };
            System.out.println(person.say("time is precious."));
    }
}
```

MULTIPLE STATEMENTS

```
interface MyInterface {

    // abstract method
    String reverse(String n);
}
public class LambadaExample9 {
    public static void main( String[] args ) {
        // declare a reference to MyInterface
        // assign a lambda expression to the reference
        MyInterface ref = (str) -> {

            String result = "";
            for (int i = str.length()-1; i >= 0 ; i--)
            result += str.charAt(i);
            return result;
        };

        // call the method of the interface
        System.out.println("Lambda reversed = " + ref.reverse("Lambda"));
    }
}
```

SCOPE

Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

METHOD REFERENCES AS LAMBDAS

In the case where all your lambda expression does is to call another method with the parameters passed to the lambda, the Java lambda implementation provides a shorter way to express the method call. First, here is an example single function interface:

```
public interface MyPrinter{
    public void print(String s);
}
```

And here is an example of creating a Java lambda instance implementing the MyPrinter interface:

```
MyPrinter myPrinter = (s) -> { System.out.println(s); };
```

METHOD REFERENCES AS LAMBDAS

Because the lambda body only consists of a single statement, we can actually omit the enclosing { } brackets. Also, since there is only one parameter for the lambda method, we can omit the enclosing ( ) brackets around the parameter. Here is how the resulting lambda declaration looks:

```
MyPrinter myPrinter = s -> System.out.println(s);
```

Since all the lambda body does is forward the string parameter to the System.out.println() method, we can replace the above lambda declaration with a method reference. Here is how a lambda method reference looks:

```
MyPrinter myPrinter = System.out::println;
```

METHOD REFERENCES AS LAMBDAS

Notice the double colons :: . These signal to the Java compiler that this is a method reference. The method referenced is what comes after the double colons. Whatever class or object that owns the referenced method comes before the double colons.

You can reference the following types of methods:

- Static method
- Parameter method reference
- Instance method
- Constructor

METHOD REFERENCES AS LAMBDAS

Static Method References

Here is a static method that we want to create a method reference to:

```
public class MyClass{
    public static int doFind(String s1, String s2){
        return s1.lastIndexOf(s2);
    }
}
```

METHOD REFERENCES AS LAMBDAS

Finally here is a Java lambda expression referencing the static method:

```
Finder finder = MyClass::doFind;
```

METHOD REFERENCES AS LAMBDAS

Parameter Method Reference

You can also reference a method of one of the parameters to the lambda. Imagine a single function interface that looks like this:

```
public interface Finder {
   public int find(String s1, String s2);
}
```

The interface is intended to represent a component able to search s1 for occurrences of s2. Here is an example of equivalent of this lambda definition:

```
Finder finder = (s1, s2) -> s1.indexOf(s2);
```

METHOD REFERENCES AS LAMBDAS

Instance Method References

It is also possible to reference an instance method from a lambda definition. First, let us look at a single method interface definition:

```
public interface Deserializer {
    public int deserialize(String v1);
}
```

This interface represents a component that is capable of "deserializing" a String into an int.

Now look at this StringConverter class:

```
public class StringConverter {
    public int convertToInt(String v1){
        return Integer.valueOf(v1);
    }
}
```

METHOD REFERENCES AS LAMBDAS

The convertToInt() method has the same signature as the deserialize() method of the Deserializer deserialize() method. Because of that, we can create an instance of StringConverter and reference its convertToInt() method from a Java lambda expression, like this:

```
StringConverter stringConverter = new StringConverter();

Deserializer des = stringConverter::convertToInt;
```

The lambda expression created by the second of the two lines references the convertToInt method of the StringConverter instance created on the first line.

METHOD REFERENCES AS LAMBDAS

Constructor References

Finally it is possible to reference a constructor of a class. You do that by writing the class name followed by ::new, like this:

MyClass::new

Too see how to use a constructor as a lambda expression, look at this interface definition:

```
interface MyInterface {
   public Student get(String str);
}
```

METHOD REFERENCES AS LAMBDAS

The create() method of this interface matches the signature of one of the constructors in the String class. Therefore this constructor can be used as a lambda. Here is an example of how that looks:

```
MyInterface constructorRef = Student :: new;
```

METHOD REFERENCES AS LAMBDAS

```
interface MyInterface {
   public Student get(String str);
}
class Student {
   private String str;
   public Student(String str) {
      this.str = str;
      System.out.println("The name of the student is: " + str);
   }
}
public class ConstrutorReferenceTest {
   public static void main(String[] args) {
      MyInterface constructorRef = Student :: new;   // constructor reference
      constructorRef.get("Adithya");
   }
}
```