Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear way to represent one method interface using an expression.

It is very useful in collection library.

It helps to iterate, filter and extract data from collection.

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.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

1. (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.

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
}
```

Without Lambda Expression

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Test it Now

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

```
1.  @FunctionalInterface  //It is optional
2.  interface Drawable{
3.      public void draw();
4.  }
5.
6.  public class LambdaExpressionExample2 {
7.      public static void main(String[] args) {
8.          int width=10;
9.
10.         //with lambda
11.         Drawable d2=()->{
12.             System.out.println("Drawing "+width);
13.         };
14.         d2.draw();
15.     }
16. }
```

Test it Now

Output:

```
Drawing 10
```

A lambda expression can have zero or any number of arguments. Let's see the examples:

Java Lambda Expression Example: No Parameter

```
1. interface Sayable{
2.     public String say();
3. }
4. public class LambdaExpressionExample3{
5. public static void main(String[] args) {
6.     Sayable s=()->{
7.         return "I have nothing to say.";
8.     };
9.     System.out.println(s.say());
10. }
11. }
```

Test it Now

Output:

```
I have nothing to say.
```

Java Lambda Expression Example: Single Parameter

```
1. interface Sayable{
2.     public String say(String name);
3. }
4.
5. public class LambdaExpressionExample4{
6.     public static void main(String[] args) {
7.
8.         // Lambda expression with single parameter.
9.         Sayable s1=(name)->{
10.            return "Hello, "+name;
11.        };
12.        System.out.println(s1.say("Sonoo"));
13.
14.        // You can omit function parentheses
15.        Sayable s2= name ->{
16.            return "Hello, "+name;
17.        };
18.        System.out.println(s2.say("Sonoo"));
19.    }
20. }
```

Test it Now

Output:

```
Hello, Sonoo
Hello, Sonoo
```

Java Lambda Expression Example: Multiple Parameters

```
1. interface Addable{
2.     int add(int a,int b);
3. }
4.
5. public class LambdaExpressionExample5{
6.     public static void main(String[] args) {
7.
8.         // Multiple parameters in lambda expression
9.         Addable ad1=(a,b)->(a+b);
10.        System.out.println(ad1.add(10,20));
11.
12.        // Multiple parameters with data type in lambda expression
13.        Addable ad2=(int a,int b)->(a+b);
14.        System.out.println(ad2.add(100,200));
15.    }
16. }
```

Test it Now

Output:

```
30
300
```

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

```
1. interface Addable{
2.     int add(int a,int b);
3. }
4.
5. public class LambdaExpressionExample6 {
6.     public static void main(String[] args) {
7.
8.         // Lambda expression without return keyword.
9.         Addable ad1=(a,b)->(a+b);
10.        System.out.println(ad1.add(10,20));
11.
12.        // Lambda expression with return keyword.
13.        Addable ad2=(int a,int b)->{
14.                        return (a+b);
15.                        };
16.        System.out.println(ad2.add(100,200));
17.    }
18. }
```

Test it Now

Output:

```
30
300
```

Java Lambda Expression Example: Foreach Loop

```
1. import java.util.*;
2. public class LambdaExpressionExample7{
3.     public static void main(String[] args) {
4.
5.         List<String> list=new ArrayList<String>();
6.         list.add("ankit");
7.         list.add("mayank");
8.         list.add("irfan");
9.         list.add("jai");
10.
```

```
        list.forEach(
            (n)->System.out.println(n)
        );
    }
}
```

Test it Now

Output:

```
ankit
mayank
irfan
jai
```

Java Lambda Expression Example: Multiple Statements

```
@FunctionalInterface
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."));
    }
}
```

Test it Now

Output:

```
I would like to say, time is precious.
```

Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

```
1.  public class LambdaExpressionExample9{
2.      public static void main(String[] args) {
3.
4.          //Thread Example without lambda
5.          Runnable r1=new Runnable(){
6.              public void run(){
7.                  System.out.println("Thread1 is running...");
8.              }
9.          };
10.         Thread t1=new Thread(r1);
11.         t1.start();
12.         //Thread Example with lambda
13.         Runnable r2=()->{
14.                 System.out.println("Thread2 is running...");
15.         };
16.         Thread t2=new Thread(r2);
17.         t2.start();
18.     }
19. }
```

Test it Now

Output:

```
Thread1 is running...
Thread2 is running...
```

Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example 1

```
1.  @FunctionalInterface
2.  interface sayable{
3.      void say(String msg);
4.  }
5.  public class FunctionalInterfaceExample implements sayable{
6.      public void say(String msg){
7.          System.out.println(msg);
8.      }
9.      public static void main(String[] args) {
10.         FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
11.         fie.say("Hello there");
12.     }
13. }
```

Block Lambda Expressions in Java

Lambda expression is an unnamed method that is not executed on its own. These expressions cause anonymous class. These lambda expressions are called closures. Lambda's body consists of a block of code. If it has only a single expression they are called "Expression Bodies". Lambdas which contain expression bodies are known as "Expression Lambdas".

Block Lambda contains many operations that work on lambda expressions as it allows the lambda body to have many statements. This includes variables, loops, conditional statements like if, else and switch statements, nested blocks, etc. This is created by enclosing the block of statements in lambda body within braces {}. This can even have a return statement i.e return value.

```
// Java Program to illustrate Lambda expression

// Importing input output classes

import java.io.*;

interface If1 {


    boolean fun(int n);
}

class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        If1 isEven = (n) -> (n % 2) == 0;


        if (isEven.fun(21))

            System.out.println("21 is even");
        else

            System.out.println("21 is odd");
    }
}
```

Output

```
21 is odd
```

```
// Java Program to illustrate Block Lambda expression

import java.io.*;

interface Func {

    int fact(int n);
}

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

        Func f = (n) ->
        {
            int res = 1;
```

```
            for (int i = 1; i <= n; i++)
                res = i * res;
            return res;
        };


        System.out.println("Factorial of 5 : " + f.fact(5));
    }
}
```

Output

```
Factorial of 5: 120
```

```
    // Java Program to illustrate Block Lambda expression

    import java.io.*;

    interface New {

        boolean test(int n);
    }

    class GFG {

        public static void main(String[] args)
        {
            New leapyr = (year) ->
            {
                if (((year % 400 == 0)
                     || (year % 4 == 0) && (year % 100 !=
    0)))

                    return true;
                else


                    return false;
            };


            if (leapyr.test(2020))

                // Display message on the console
                System.out.println("leap year");
            else


                System.out.println("Non leap year");
        }
```

```
    }
```

Output

```
leap year
```

How to pass a lambda expression as a method parameter in Java?

A **lambda expression** is an **anonymous** or **unnamed method** in Java. It doesn't execute on its own and used to implement methods that are declared in a **functional interface**. If we want to pass a lambda expression as a **method parameter** in java, the type of method parameter that receives must be of **functional interface type**.

```
// An interface that declares a method for calculating
// the average of 3 numbers.
interface IAverage {
  double Avg(double a, double b, double c);
}
```

The next step is to declare a method that will receive a reference to the IAverage interface as a parameter. This method can be placed in some class as shown below

```
// The class containing the implementation of the method that
// receives as a parameter a reference to the IAverage interface
class SomeClass {

  // A method that displays the average of three numbers.
  // The method receives a reference to the IAverage functional interface.
  // The program code for calculating average is formed in a lambda
expression,
  // which is passed as an argument to this method.
  public void PrintAverage(IAverage ref) {
    System.out.println("ref = " + ref.Avg(3, 7, 8));
  }
}
```

After that, in some client method (for example, the main() method), you need to pass the corresponding lambda expression to the PrintAverage() method like this:

```
public static void main(String[] args) {
  // Client code - demonstrates passing a lambda expression to a method
  // 1. Create an instance of the SomeClass class
  SomeClass obj = new SomeClass();

  // 2. Call the PrintAverage() method and pass a lambda expression to it
  obj.PrintAverage((a,b,c) -> (a+b+c)/3.0);
}
```

Way 2. With this method, a reference to the interface is pre-formed. A lambda expression is then assigned to this reference. This reference is then passed to the PrintAverage() method. This approach is useful when the code for the lambda expression is too large and makes the method invocation difficult to read. The following is a modified code from the previous example.

```
...

public static void main(String[] args) {
  // Client code - demonstrates passing a lambda expression to a method
  // 1. Declare a reference to the IAverage interface
  IAverage ref;

  // 2. Assign a lambda expression that calculates average
  //   of 3 values to the IAverage interface reference
  ref = (a, b, c) -> (a+b+c)/3.0;

  // 3. Create the instance of SomeClass class
  SomeClass obj = new SomeClass();

  // 4. Invoke method PrintAverage()
  obj.PrintAverage(ref);
}
```

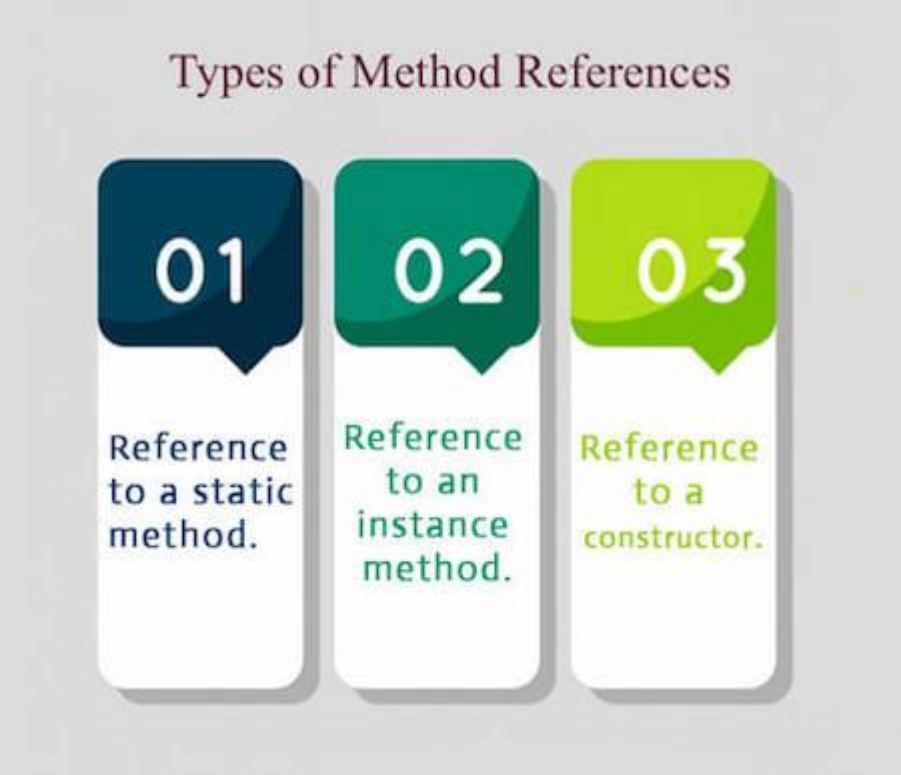
Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

Types of Method References

There are following types of method references in java:

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



---

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
1. interface Sayable{
2.     void say();
3. }
4. public class MethodReference {
5.     public static void saySomething(){
6.         System.out.println("Hello, this is static method.");
7.     }
8.     public static void main(String[] args) {
9.         // Referring static method
10.        Sayable sayable = MethodReference::saySomething;
11.        // Calling interface method
12.        sayable.say();
13.    }
14. }
```

Test it Now

Output:

```
Hello, this is static method.
```

Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

```
1. public class MethodReference2 {
2.     public static void ThreadStatus(){
3.         System.out.println("Thread is running...");
4.     }
```

```
5.      public static void main(String[] args) {
6.         Thread t2=new Thread(MethodReference2::ThreadStatus);
7.         t2.start();
8.      }
9.   }
```

Test it Now

Output:

```
Thread is running...
```

---

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

```
1.  containingObject::instanceMethodName
```

Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```
1.  interface Sayable{
2.     void say();
3.  }
4.  public class InstanceMethodReference {
5.     public void saySomething(){
6.        System.out.println("Hello, this is non-static method.");
7.     }
```

```
8.      public static void main(String[] args) {
9.          InstanceMethodReference methodReference = new InstanceMethodReference(); // Cre
    ating object
10.         // Referring non-static method using reference
11.             Sayable sayable = methodReference::saySomething;
12.         // Calling interface method
13.             sayable.say();
14.             // Referring non-static method using anonymous object
15.             Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use a
    nonymous object also
16.             // Calling interface method
17.             sayable2.say();
18.     }
19. }
```

Test it Now

Output:

```
Hello, this is non-static method.
Hello, this is non-static method.
```

Example 2

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

```
1.  public class InstanceMethodReference2 {
2.      public void printnMsg(){
3.          System.out.println("Hello, this is instance method");
4.      }
5.      public static void main(String[] args) {
6.      Thread t2=new Thread(new InstanceMethodReference2()::printnMsg);
7.          t2.start();
8.      }
9.  }
```

Test it Now

Output:

```
Hello, this is instance method
```

---

3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

1. ClassName::new

Example

```
1. interface Messageable{
2.     Message getMessage(String msg);
3. }
4. class Message{
5.     Message(String msg){
6.         System.out.print(msg);
7.     }
8. }
9. public class ConstructorReference {
10.     public static void main(String[] args) {
11.         Messageable hello = Message::new;
12.         hello.getMessage("Hello");
13.     }
14. }
```

Test it Now

Output:

```
Hello
```