Estd. 2000

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 21

Lecture 21

- Functional Interfaces
- Lambda Expression

Functional Interfaces

- A functional interface is an interface that contains only one abstract method.
- They can have only one functionality to exhibit.
- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
- A functional interface can have any number of default methods.
- Runnable, ActionListener, and Comparable are some of the examples of functional interfaces.

- Functional Interface is additionally recognized as Single Abstract Method Interfaces.
- In short, they are also known as SAM interfaces.
- Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.
- Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward.

Functional interfaces are interfaces that ensure that they include precisely only one abstract method.

Functional interfaces are used and executed by representing the interface with an annotation called @FunctionalInterface.

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only.

We can also call Lambda expressions as the instance of functional interface.

```
class Test {
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable() {
            @Override public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

// functional interface using lambda expressions

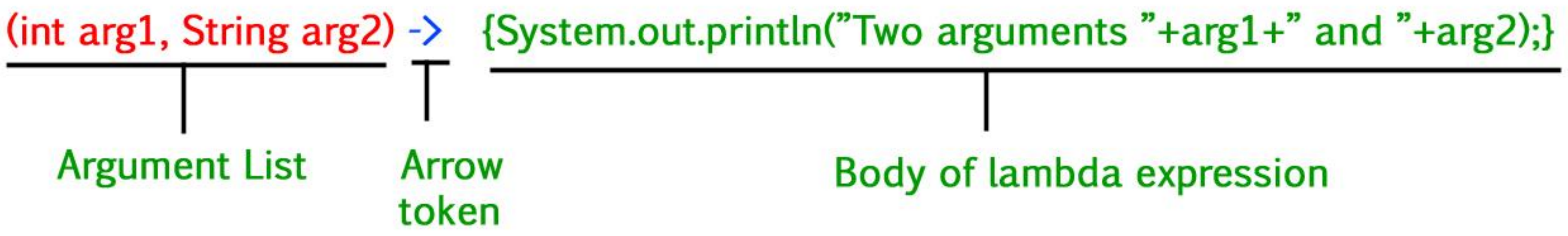
```
class Test {
    public static void main(String args[])
    {
        // lambda expression to create the object
        new Thread(() -> {
          System.out.println("New thread created");
        }).start();
    }
}
```

Lambda Expression in Java

Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value.

Lambda Expression Syntax

lambda operator -> body


(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}
Argument List
Arrow token
Body of lambda expression

There are three Lambda Expression Parameters are mentioned below:

- Zero Parameter
- Single Parameter
- Multiple Parameters

1. Lambda Expression with Zero parameter

```
() -> System.out.println("Zero parameter lambda");
```

2. Lambda Expression with Single parameter

```
(p) -> System.out.println("One parameter: " + p);
```

3. Lambda Expression with Multiple parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Example of Lambda Expression with one parameter

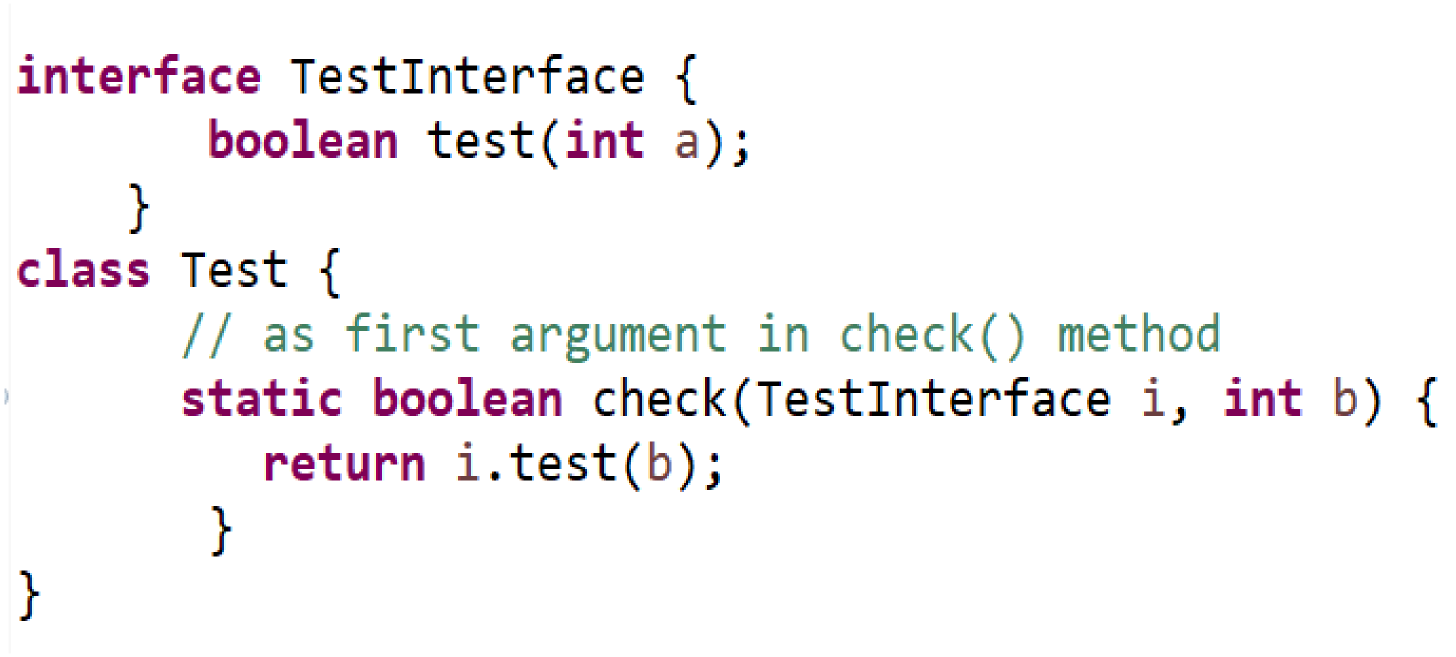
```
//functional interface
interface NumericTest{
boolean test(int n);
}
public class LambdaDemo1 {
  public static void main(String args[]) {

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

  if(isEven.test(10)) System.out.println("10 is
even");
  if(!isEven.test(9)) System.out.println("9 is not
even");
  }
}
```

Passing Lambda Expression as arguments

To pass a lambda expression as a method parameter in java, the type of method parameter that receives must be of **functional interface** type.



```
interface TestInterface {
       boolean test(int a);
    }
class Test {
      // as first argument in check() method
      static boolean check(TestInterface i, int b) {
         return i.test(b);
       }
}
```

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method.

In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}
class Test1 {
    public static void main(String args[])
    {
        int a = 5;
        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;
        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }}
```

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface.

These interfaces are as follows

- **Runnable –>** This interface only contains the run() method.
- **Comparable –>** This interface only contains the compareTo() method.
- **ActionListener –>** This interface only contains the actionPerformed() method.
- **Callable –>** This interface only contains the call() method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

1. Consumer

- The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument.
- The consumer interface has no return value. It returns nothing.
- There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer.
- These variants accept primitive values as arguments.

.

Bi-Consumer

- Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer
- Bi-Consumer is the most exciting variant of the Consumer interface.
- Bi-Consumer interface takes two arguments.
- Both, Consumer and Bi-Consumer have no return value.
- It also returns nothing just like the Consumer interface.
- It is used in iterating through the entries of the map.

Syntax / Prototype of Consumer Functional Interface –

```
Consumer<Integer> consumer = (value) -> System.out.println(value);
```

2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate.

Syntax of Predicate Functional Interface –

```
public interface Predicate<T>
 {
    boolean test(T t);
 }
```

Bi-Predicate

Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.

Example of the implementation of the Predicate functional interface is

```
Predicate predicate = (value) -> value != null;
```

3. Function

- A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing.
- There are many versions of Function interfaces because a primitive type can’t imply a general type argument, so we need these versions of function interfaces.

Bi-Function

The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

syntax of Bi-Function

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{
   R apply(T t, U u);
 }
```

T and U are the inputs, and there is only one output which is R.

Unary Operator and Binary Operator

- There are also two other functional interfaces which are named Unary Operator and Binary Operator.
- They both extend the Function and Bi-Function, respectively.
- In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

The prototype of the Unary Operator and Binary Operator is mentioned below

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, U>
{
    ………
}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, U,
R>
{
    ………
}
```

4. Supplier

- The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output.
- This type of functional interface is generally used in the lazy generation of values.
- Supplier functional interfaces are also used for defining the logic for the generation of any sequence.

For example –

The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.

// lambda expression to create object

```
Predicate<String> p = (s) -> s.startsWith("G");
```