Java Functional Interfaces

softserve

Agenda

1. Functional Interface
2. Lambda Expression
3. Built-in Functional Interfaces in Java
    - Function
    - Predicate
    - UnaryOperator
    - BinaryOperator
    - Supplier
    - Consumer

Functional Interfaces

A **functional interface** in Java is an interface that contains only a single abstract (unimplemented) method.

A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

```
public interface MyFunctionalInterface {
    public void execute();
}
```

Functional Interfaces

Normally a Java interface does not contain implementations of the methods it declares.

But it can contain implementations

- in default methods,
- or in static methods

```
public interface MyFunctionalInterface2{
    public void execute();

    public default void print(String text) {
        System.out.println(text);
    }

    public static void print(String text, PrintWriter writer) throws IOException {
        writer.write(text);
    }
}
```

@FunctionalInterface Annotation

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

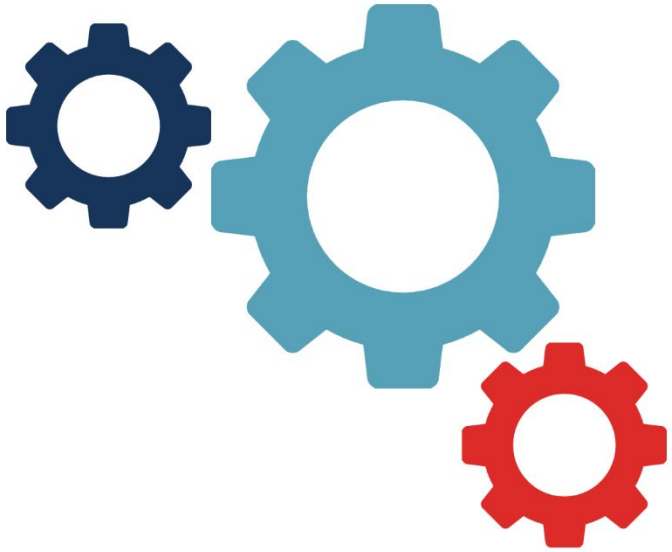
In case more than one abstract methods are present, the compiler flags an '**Unexpected @FunctionalInterface annotation**' message.

However, it is not mandatory to use this annotation.

Lambda expressions

lambda expressions are added in Java 8 and provide below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.



Syntax of lambdas

```
lambda operator -> body
```

where lambda operator can be:

Zero parameter:

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

One parameter:–

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

It is not mandatory to use parentheses, if the type of that variable can be inferred from the context

Multiple parameters :

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

Lambda Expression

A Java functional interface can be implemented by a **Java Lambda Expression.**

```
MyFunctionalInterface lambda = () -> {
    System.out.println("Executing...");
}
```

A Java lambda expression implements a single method from a Java interface.

In order to know what method the lambda expression implements, the interface can only contain a single unimplemented method. (The interface must be a Java functional interface.)

Built-in Functional Interfaces in Java

softserve

Function

The Function interface (`java.util.function.Function`) represents a function (method) that takes a single parameter and returns a single value.

```
public interface Function<T,R> {

    public <R> apply(T parameter);
}
```

The Function interface actually contains a few extra methods in addition to the method shown above, but since they all come with a default implementation, you do not have to implement these extra methods.

Function

The only method you have to implement to implement the **Function** interface is the **apply()** method:

```
public class AddThree implements Function<Long, Long> {

    @Override
    public Long apply(Long aLong) {
        return aLong + 3;
    }
}
```

Function

An example of using the above AddThree class:

```
Function<Long, Long> adder = new AddThree();
Long result = adder.apply((long) 4);
System.out.println("result = " + result);
```

- First this example creates a new AddThree instance and assigns it to a Function variable.
- Second, the example calls the apply() method on the AddThree instance.
- Third, the example prints out the result (which is 7).

Function

You can also implement the Function interface using a Java lambda expression:

```
Function<Long, Long> adder = (value) -> value + 3;
Long resultLambda = adder.apply((long) 8);
System.out.println("resultLambda = " + resultLambda);
```

The Function interface implementation is now inlined in the declaration of the adderLambda variable, rather than in a separate class.

Predicate

The Java Predicate interface (`java.util.function.Predicate`) represents a simple function that takes a single value as parameter, and returns true or false:

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

The Predicate interface contains more methods than the test() method, but the rest of the methods are default or static methods which you don't have to implement.

Predicate

You can implement the Predicate interface using a class, like this:

```
public class CheckForNull implements Predicate {
    @Override
    public boolean test(Object o) {
        return o != null;
    }
}
```

The Predicate interface contains more methods than the test() method, but the rest of the methods are default or static methods which you don't have to implement.

Predicate

You can also implement the Java Predicate interface using a Lambda expression. Here is an example of implementing the Predicate interface using a Java lambda expression:

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

This lambda implementation of the Predicate interface effectively does the same as the implementation that uses a class.

UnaryOperator

The Java UnaryOperator interface is a functional interface that represents an operation which takes a single parameter and returns a parameter of the same type.

```
UnaryOperator<Person> unaryOperator =
        (person) -> { person.name = "New Name"; return person;
};
```

The UnaryOperator interface can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it again - possibly as part of a functional stream processing chain.

BinaryOperator

The Java BinaryOperator interface is a functional interface that represents an operation which takes two parameters and returns a single value. Both parameters and the return type must be of the same type:

```
BinaryOperator<MyValue> binaryOperator =
        (value1, value2) -> { value1.add(value2); return value1; };
```

The Java BinaryOperator interface is useful when implementing functions that sum, subtract, divide, multiply etc. two elements of the same type, and returns a third element of the same type

Supplier

The Java Supplier interface is a functional interface that represents an function that supplies a value of some sorts. The Supplier interface can also be thought of as a factory interface.

```
Supplier<Integer> supplier = () -> new Integer((int) (Math.random() * 1000D));
```

This Java Supplier implementation returns a new Integer instance with a random value between 0 and 1000.

Consumer

The Java Consumer interface is a functional interface that represents an function that consumes a value without returning any value. A Java Consumer implementation could be printing out a value, or writing it to a file, or over the network etc.

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

This Java Consumer implementation prints the value passed as parameter to it out to System.out.

Method Reference

- Method reference is used to refer a method without invoking it.
- Instead of providing implementation of a method like lambdas do, method references refer to a method of an existing class or object.
- Operator :: called as **Method Reference Delimiter**.

```
1.  System::getProperty
2.  "abc"::length
3.  String::length
4.  super::toString
5.  ArrayList::new
```

Constructor reference

References

http://tutorials.jenkov.com/java-functional-programming/functional-interfaces.html

https://metanit.com/java/tutorial/3.16.php

https://metanit.com/java/tutorial/9.1.php

THANKS