

Java: Functional Interface and Lambda Expression

"Concept && Coding" YT Video Notes

- What is Functional Interface?
- What is Lambda Expression?
- How to use Functional Interface with Lambda expression
- Advantage of Functional Interface?
- Types of Functional Interface?
 - o Consumer
 - o Supplier
 - o Function
 - o Predicate
- How to handle use case when Functional Interface extends from other Interface(or Functional Interface)?

What is Functional Interface:

- If an interface contains only 1 abstract method, that is known as Functional Interface.
- ✓ Also know as SAM Interface (Single Abstract Method).
- @FunctionalInterface keyword can be used at top of the interface (But its optional).

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
}

OR

public interface Bird {
    void canFly(String val);
}
```

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
    void getWeight();
}
```

← @FunctionalInterface Annotation restrict us and throws compilation error, if we try to add more than 1 abstract method

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
    default void getWeight() {
        //default method implementation
    }
    static void canFly() {
        //for static method implementation
    }
    String toString(); //Object class method
}
```

✓ *Abstract Method*

← In Functional interface, only 1 abstract method is allowed, but we can have other methods like default, static method and methods inherited from the Object class

```
public interface TestInterface {
    String testString();
}
```

✓ *Object (or String)*

```
public class TestClass implements TestInterface {
}
```

What is Lambda Expression:

- Lambda expression is a way to implement the Functional Interface.

Before going into further into Lambda expression, lets first see:

Different Ways to Implement the Functional Interface:

1. Using "implements"

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
}

public class Eagle implements Bird {
    @Override
    public void canFly(String val) {
        System.out.println("Eagle Bird Implementation");
    }
}

Bird eagleObject = new Eagle();
eagleObject.canFly("vertical");
```

2. Using "anonymous class"

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
}

public class Main {
    public static void main(String args[]) {
        Bird eagleObject = new Bird() {
            @Override
            public void canFly(String val) {
                System.out.println("Eagle Bird Implementation");
            }
        };
        eagleObject.canFly("vertical");
    }
}
```

3. Using "Lambda Expression"

```
@FunctionalInterface
public interface Bird {
    void canFly(String val);
}

public class Main {
    public static void main(String args[]) {
        Bird eagleObject = (String val) -> {
            System.out.println("Eagle Bird Implementation");
        };
        eagleObject.canFly("vertical");
    }
}
```

() → parameter & method

Types of Functional Interface:

Consumer

- Represents an operation, that accept a single input parameter and returns no result.
- Present in package: java.util.function.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

public class Main {
    public static void main(String args[]) {
        Consumer<Integer> integerConsumer = (Integer val) -> {
            System.out.println(val);
        };
        integerConsumer.accept(123);
    }
}
```

Supplier

- Represent the supplier of the result. Accepts no input parameter but produce a result.
- Present in package: java.util.function.

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}

public class Main {
    public static void main(String args[]) {
        Supplier<Integer> integerSupplier = () -> "this is the data I am returning";
        System.out.println(integerSupplier.get());
    }
}

OR

public class Main {
    public static void main(String args[]) {
        Supplier<Integer> integerSupplier = () -> {
            return "this is the data I am returning";
        };
        System.out.println(integerSupplier.get());
    }
}
```

Function

- Represent function, that accepts one argument process it and produce a result.
- Present in package: java.util.function.

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

public class Main {
    public static void main(String args[]) {
        Function<Integer, String> integerFunction = (Integer num) -> {
            String string = num.toString();
            return output;
        };
        System.out.println(integerFunction.apply(123));
    }
}
```

Predicate

- Represent function, that accept one argument and return the boolean.
- Present in package: java.util.function.

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

public class Main {
    public static void main(String args[]) {
        Predicate<Integer> idEven = (Integer val) -> {
            (val % 2 == 0) ?
                return true;
            :
            return false;
        };
        System.out.println(idEven.test(123));
    }
}
```

Handle use case when Functional Interface extends from other Interface:

Use Case 1: Functional Interface extending Non Functional Interface

```
public interface LivingThing {
    public void canBreathe();
}
```

✗

```
@FunctionalInterface
public interface Bird extends LivingThing {
    void canFly(String val);
}
```

```
public interface LivingThing {
    default public boolean canBreathe() {
        return true;
    }
}
```

✓

```
@FunctionalInterface
public interface Bird extends LivingThing {
    void canFly(String val);
}
```

Use Case 2: Interface extending Functional Interface

```
@FunctionalInterface
public interface LivingThing {
    public boolean canBreathe();
}
```

✓

```
public interface Bird extends LivingThing {
    void canFly(String val);
}
```

✓

Use Case 3: Functional Interface extending other Functional Interface

```
@FunctionalInterface
public interface LivingThing {
    public boolean canBreathe();
}
```

✗

```
@FunctionalInterface
public interface Bird extends LivingThing {
    void canFly(String val);
}
```

✓

```
@FunctionalInterface
public interface LivingThing {
    public boolean canBreathe();
}
```

✓

```
@FunctionalInterface
public interface Bird extends LivingThing {
    boolean canBreathe();
}
```

✓

```
public class Main {
    public static void main(String args[]) {
        Bird eagle = () -> true;
        System.out.println(eagle.canBreathe());
    }
}
```

Bird obj = ()