- What is Functional Interface?
- What is Lambda Expression?
- How to use Functional Interface with Lambda expression
- Advantage of Functional Interface?
- Types of Functional Interface?
  - Consumer
  - Supplier
  - Function
  - 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).

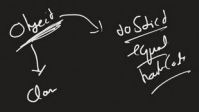
```java
@FunctionalInterface
public interface Bird  {
     void canFly(String val);
}
```

OR

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

```java
@FunctionalInterface
public interface Bird  {

     void canFly(String val);
     void getHeight();
}
```

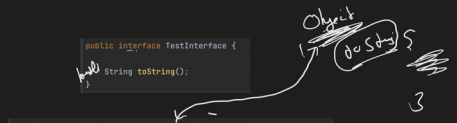
<----- @FunctionalInterface Annotation restrict us and throws compilation error, if we try to add more than 1 abstract method




```java
@FunctionalInterface
public interface Bird {

    void canFly(String val);

    default void getHeight(){
        //default method implementation
    }

    static void canEat(){
        //my static method implementation
    }

    String toString(); //Object class method
}
```

<----- In Functional Interface, only 1 abstract method is allowed, but we can have other methods like default, static method or Methods inherited from the Object Class

```java
public interface TestInterface {

    String toString();
}
```




```java
public class TestClassImplements 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 Implements 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( val: "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 value) -> {
                System.out.println("Eagle Bird Implementation");
            };

        eagleObject.canFly( val: "vertical");
    }
}
```

arrow function

( ) → ( )

( )

Types of Functional Interface:

Consumer

- **Represent 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> loggingObject = (Integer val) -> {
            if(val>10) {
                System.out.println("Logging");
            }
        };

        loggingObject.accept( t: 11);
    }
```

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<String> isEvenNumber = () -> "this is the data i am returning";

        System.out.println(isEvenNumber.get());
    }
}
```

OR

```
public class Main {

    public static void main(String args[]) {

        Supplier<String> isEvenNumber = () -> {
            return "this is the data i am returning";
        };

        System.out.println(isEvenNumber.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> integerToString =
                (Integer num) -> {
                    String output = num.toString();
                    return output;
                };

        System.out.println(integerToString.apply( t: 64));
    }
}
```

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> isEven = (Integer val)

            if(val%2 == 0){
                return true;
            } else {
                return false;
            }
        };

        System.out.println(isEven.test( t: 19));
    }
}
```

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

```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

```java
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

Use Case 3: Functional Interface extending other Functional Interface

```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    boolean canBreathe();
}
```

```java
public class Main {

    public static void main(String args[]) {

        Bird eagle = () -> true;

        System.out.println(eagle.canBreathe());

    }
}
```