Functional interfaces and lambdas

What is a functional interface?

- An **interface with exactly one abstract method**.
- Can have **default** and **static methods**.
- Annotated with **@FunctionalInterface** (optional but recommended). If we add the annotation then it will not allow us to add more than one abstract method =>

Multiple non-overriding abstract methods found in interface org. practice. oops. LivingThing

```
4   @FunctionalInterface  no usages  new *
5   public interface LivingThing {
6
7       void canFly(String value);  no usages  new *
8
9       boolean canEat(String value);  no usages  new *
10  }
```

What is lambda expression? And how to use a functional interface with lambda expression?

- A short way to write anonymous methods (functional code).
- Used to implement the abstract method of a functional interface.

There are 3 ways to implement a functional interface =>

Consider the below functional interface –

```
3   @FunctionalInterface  no usages  new *
4   public interface AdditionOperator {
5       int add(int a, int b);  no usages  new *
6   }
```

1. Using class =>

```
3   public class Addition implements AdditionOperator{  new *
4
5       public static void main(String[] args) {  new *
6           Addition addition = new Addition();
7           System.out.println(addition.add( a: 10, b: 20));
8       }
9
10      @Override  1 usage  new *
11      public int add(int a, int b) {
12          return a + b;
13      }
14  }
```

2. Using anonymous class and lambda expression =>

```java
public class Test {  new *

    public static void main(String[] args) {  new *

        //anonymous class
        AdditionOperator addition = new AdditionOperator() {  new *
            @Override  2 usages  new *
            public int add(int a, int b) {
                return a + b;
            }
        };
        addition.add( a: 10,  b: 20);

        //lambda expression
        AdditionOperator operator = (a,b) -> a + b;
        operator.add( a: 10, b: 20);
    }

```

Advantages of functional interface.

- Helps in writing **cleaner and concise code**.
- Supports **functional programming** in Java.
- Useful in **streams, APIs**, and **event handling**.
- Easy to pass **behavior as a parameter** (like a method).

Types of functional interfaces => Consumer, Supplier, Function, Predicate

Consumer:

```java
public static void main(String[] args) {  new *
    //Accept single input parameter and returns no result.
    Consumer<Integer> consumer = (val) -> {
        if (val > 10) {
            System.out.println("greater");
        }
    };
    consumer.accept( t: 19);
```

Supplier:

```
10      public static void main(String[] args) {  new *
11          //Accept no input parameter but produces a result.
12          Supplier<String> supplier = () -> {
13              return "I am supplier";
14          };
15          System.out.println(supplier.get());
```

Function:

```
10      public static void main(String[] args) {  new *
11          //Accept one input parameter process it and  produces a result.
12          Function<Integer, String> intToString = (num) -> {
13              return num.toString();
14          };
15          System.out.println(intToString.apply( t: 10));
```

Predicate:

```
10      public static void main(String[] args) {  new *
11          //Accept one input parameter and produces boolean result.
12          Predicate<Integer> isValid = (num) -> {
13              return (num > 10) ? true : false;
14          };
15          System.out.println(isValid.test( t: 50));
16      }
17
18  }
```

How to handle use case when functional interface extends from another interface (or functional interface)?

1. **Functional interface extending non-functional interface:**
   It must have only one abstract method (either in parent or own) otherwise we will get below error:
   "Multiple non-overriding abstract methods found in interface org. practice. oops. AdditionOperator"

```java
@FunctionalInterface
public interface AdditionOperator extends Operator {
    int add(int a, int b);
}
```

```java
public interface Operator{
    void operate();
}
```

This will work =>

```java
@FunctionalInterface
public interface AdditionOperator extends Operator {
    int add(int a, int b);
}
```

```java
public interface Operator{
    void operate();
}
```

Also, this will work (same abstract method in parent and child) =>

```java
@FunctionalInterface
public interface AdditionOperator extends Operator {
    int add(int a, int b);
}
```

```java
public interface Operator{
    int add(int a, int b);
}
```

2. **Functional interface extending another functional interface:**
   Both must have the same abstract method but can have different static/default methods =>

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

```java
@FunctionalInterface
public interface Bird extends LivingThing {
    void canBreathe();

    default void fly(){
        System.out.println("Bird is flying...");
    }
}
```

3. **Interface extending functional interface:**

```java
@FunctionalInterface
public interface LivingThing {
    void canFly();
}
```

```java
public interface Bird extends LivingThing {
    void canBreathe();
}
```

This works because even though Bird is extending LivingThing, LivingThing still has only one abstract method.