16- Functional Interface and Lamda Expression

Q) What is a Functional Interface?

* If an interface contains only 1 abstract method, that is known as Functional Interface.
* Also known as SAM interface (Single Abstract Method)
* We can add @FunctionalInterface keyword at the top, (But its optional).

```
@FunctionalInterface
public interface Bird{

    void canFly();

}
```

or

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

```
@FunctionalInterface
public interface Bird{

    void canFly();
    void hasWings();
}
```

→ Compilation ERROR

By definition of Functional Interface.

It is said that it should have only a single Abstract method,
It can have default, static methods and methods inherited from Object Class.

Example.

```
public interface Bird {

    void canFly(); → Only Abstract Method.

    default void hasWings(){ → Default Method,
        // Some implementation.
    }

    static void canEat(){ → Static Method.
        // Some implementation
    }

    String toString(); → this is a method of the Object Class. this we can have in Interface and Functional Interface as well

}
```

Not an abstract method
Has implementation inside the Object Class.

Sample Example

```
public interface TestInterface{

    String toString();

}
```

```
public class TestClass implements TestInterface{

}
```

We do not need to implement this method Because bydefault any Class extends the Object Class

Lambda Expression.

Lambda Expression is a way to implement Functional Interface

Different ways to implement Functional Interface.

```
interface Bird{

    void canFly();

}
```

Making concrete Class

```
class BirdClass implements Bird{

    @Override
    void canFly(){
        // some implementation
    }
}
```

Making Anonymous Class

```
main(){

    Bird obj = new Bird(){
        @Override
        void canFly(){
        }
    };
}
```

Using Lambda Expression to implement Interface

(***) Main Intention of Lambda Expression is to reduce the verbose.

(***) In the previous two ways, we know that Functional Interface will have only 1 abstract method. Then why do we need to provide method Name as well.

```
@FunctionalInterface
public interface Bird {

    void canFly(String val);
}
```

```
Bird obj = (val) -> {
        sout("Print Object using Lambda");
};
obj.canFly("Lambda");
```

(val) → String

Q) How to write Lambda expression?

A)

```
(  )  ->  {

}
;
```

( ) : In the bracket pass the arguments the method is accepting.

-> : Arrow operator

{ } : write implementation inside the bracket

; : semicolon.

Write here the Function arguments as they were written in the declaration.

Types of Functional Interface

1. **Consumer <T>** → accepts Generic Class.

* Accepts a single input parameter, Return type is void.
* present in Java.util.Function ~~class~~

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

(Usage)

```
Consumer<String> obj
    = (String val) -> {
        sout(val + "consumer");
    };
```

2. **Supplier <T>** ⇒ accepts Generic Class

* Accepts no input but returns a result.
* Present in java.util.function

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

Usage

```
Supplier<String> obj
    = () -> {
        sout("Sample
        Supplier implement");
    };
```

(3) Function <I, R>

* Accepts one argument parameter and produces a result
* Present in java.util.function

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

(Usage) ⟹

```
Function<Integer, String> primeNumber
    = (Integer val) -> {
        if (val is prime) {
            return "Prime";
        }
        else
            return "Not Prime";
    }
```

Usage ⟸

```
Sout( primeNumber.apply(23));
```

(4) Predicate <T>

* Accepts one argument, but returns boolean
* Present in java.util.function.

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

Usage →

```
Predicate<Integer> p
    = (Integer a) -> return !(a%2);

p.test(2);
```

Functional Interfaces extending Non-Functional Interface

If the ~~functional interface~~ Non-functional interface has only static and default methods and 1 abstract method.
then the functional interface can extend it, provided it does not add its own abstract method.

Functional Interfaces extending other Functional Interfaces.

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

if the names are same then no problem is there.

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