JAVA: FUNCTIONAL INTERFACE AND LAMBDA EXPRESSION.

By Saurav Saxena.

1. What is Functional Interface ?
->If an interface contains only 1 abstract method, that is known as Functional Interface.
->Also known as SAM interface [SINGLE ABSTRACT METHOD].
->@FunctionalInterface keyword can be used at top of the interface [But it's optional].

```
@FunctionalInterface
public interface Bird{
  void canFly(String val);
}
@FunctionalInterface
public interface Bird{
   void canFly(String val);
   default void getHeight(){
     //default method
   }
   static void canEat(){
   }
   //don't need to provide implementation
   // the who implemented this interface.
   //of Object class method.
   String toString();//Object class method
}
```

2. Different ways to implement @FunctionalInterface
2.1. Implements keyword through class.
->

```
@FunctionalInterface
public interface Bird{
  void canFly(String val);
}
public class Eagle implements Bird{
  @Override
  public void caFLy(String Val){
    System.out.println(val);
  }
}
```

2.2. Using Anonymous Class.

```
public class Main{
  public static void main(String[] args){
    Bird b = new Eagle();
    b.canFly("Call Eagle class Implementation");
    //through anonyoums class
    Bird eagleObj = new Bird(){
      @Override
      public void canFly(String Val){
        System.out.println(val);
      }
    }
    eagleObj.canFly("Anonyoums Eagle bird implementation");
  }
}
```

3. What is Lambda Expression.
->Lambda Expression is way to implement the Functional Interface.
->Functional interface contains only one abstract method which required implementation. and we don't need to mention method identity during overriding.
->So, That's why Lambda expression comes into picture.

```
public class Main{
   public static void main(String[] args){
     Bird eagleObj = (String val) ->{
       System.out.println(val);
     }
     eagleObj.canFly("Implementaion throught lambda expression");
   }
}
```

4. Types of Functional Interface.
4.1 Consumer.
->Represent an operation, that accept a single input parameter and return 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> oddOrEven = (Integer val)->{
        if(val%10==0){
           System.out.println("even");
        }else{
           System.out.println("odd");
        }
     }
```

```
  }
}
```

4.2 Supplier.

->Represent the supplier of the result. Accepts no input 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> quote = ()-> "I am not a quote";
    System.out.println(quote.get());
  }
}
```

4.3 Function

->Represent function,that accept one parameter and process it and produce 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)->num.toString();
    System.out.println(integerToString.apply(100));
  }
}
```

4.4 Predicate.

->Represent function, that accept one parameter argument and return the Boolean result.

->present in package: java.util.function;

```
@FunctionalInterface
public interface Function<T>{
  boolean apply(T t);
}
public class Main{
  public static void main(String[] args){
    Predicate<Integer> isEven= (val)-> (val %10)==0;
    System.out.println(isEven.test(121));
  }
}
output:
false
```

5. Handle use case when Functional interface extends from other Interfaces.

USE CASE 5.1 : Functional Interface extending Non Functional Interface.
->

```
@FunctionalInterface
public interface LivingThing{
  public void canBreadth();
}
//comiple time error
//parent abstract method is also the part
//extending child interface.
@FunctionalInterface
public interface Bird extends LivingThing{
  void canFly(String val);
}
```

```
public interface LivingThing{
  default public boolean canBreadth(){
    return true;
  }
}
@FunctionalInterface
public interface Bird extends LivingThing{
  void canFly(String val);
}
```

USE CASE 5.2 : Interface extending Functional Interface.
->

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

USE CASE 5.3 : Functional Interface extending Functional Interface.
->

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

```
@FunctionalInterface
public interface LivingThing{
  public void canBreadth();
}
@FunctionalInterface
public interface Bird extends LivingThing{
  void canBreadth();
}
public class Main{
  public static void main(String[] args){
    Bird eagle = ()->true;
    System.out.println(eagle.canBreadth());
  }
}
```