Built in Functional interfaces

Agenda

1. Functional Interfaces
2. Function Interface
3. Predicate Interface
4. Consumer Interface
5. Supplier Interface

Functional Interfaces

Functional Interface

- A functional interface, introduced in Java 8, is an interface which has **only a single abstract method**
- Conversely, if you have any interface which has only a single abstract method, then that will effectively be a functional interface
- This interface can then be used anywhere where a functional interface is eligible to be used
- One of the most important uses of Functional Interfaces is that implementations of their abstract method can be passed around as lambda expressions
- **Functional interfaces are interfaces which represent some functionality (instead of representing some data)**

@FunctionalInterface annotation

- **@FunctionalInterface** annotation can be used to explicitly specify that a given interface is to be treated as a functional interface
- Then the compiler would check and give a compile-time error in case the annotated interface does not satisfy the basic condition of qualifying as a functional interface

Example:

```
@FunctionalInterface
public interface TestFunctionalInterface {    //Compilation Error: TestFunctionalInterface is not a functional interface
abstract void helloOne();
abstract void helloTwo();
}
```

Pre-existing Functional interfaces prior to Java 8

- All these interfaces represent functionalities and hence are called functional interfaces
- All of them have one single abstract method

```
@FunctionalInterface
public interface Runnable {
 public abstract void run();
}
@FunctionalInterface
public interface Comparator<T> {
 int compare(T o1, T o2);
}
```

Example1

- Create a User class with a single attribute id of type int
- Create necessary getter and setter method
- Create constructors with and without arguments
- Override toString method and print the UserId

User.java

```
public class User {
    int id;
  // remaining code goes here
}
```

Example1 continued…

SortUser.java

```
package com.wipro;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class SortUser implements Comparator<User> {

        @Override
        public int compare(User u1, User u2) {
                // TODO Auto-generated method stub
                return u1.getId()-u2.getId();
        }
```

Example1 continued…

```
public static void main(String[] args) {
                List<User> list= new ArrayList<User>();
                list.add(new User(200));
                list.add(new User(41));
                list.add(new User(6789));
                list.add(new User(12));


                // Java documentation for sort method
                // public static <T> void sort(List<T> list, Comparator<? super T> c)
                Collections.sort(list,new SortUser());


                for(User u:list)
                System.out.println(u);
}
}
```

Output:
UserId=12
UserId=41
UserId=200
UserId=6789

Example2 – Modification of Example1

```
public class SortUser2 {
public static void main(String[] args) {
   List<User> list= new ArrayList<User>();
   list.add(new User(200));
   list.add(new User(41));
   list.add(new User(6789));
   list.add(new User(12));
   Comparator<User> comparator=new Comparator<User>() {
                                  //Anonymous inner class
                                 @Override
                                 public int compare(User u1, User u2) {
                                          return u1.getId()-u2.getId();
                                          }};
  Collections.sort(list,comparator);
  for(User u:list)
  System.out.println(u); }}
```

Output:
UserId=12
UserId=41
UserId=200
UserId=6789

Example3 – Modification of Example2

```
public class SortUser3 {
public static void main(String[] args) {
   List<User> list= new ArrayList<User>();
   list.add(new User(200));
   list.add(new User(41));
   list.add(new User(6789));
   list.add(new User(12));

    // Using Lambda Expression
   Collections.sort(list,(u1,u2)->u1.getId()-u2.getId());

   for(User u:list)
   System.out.println(u);
}
}
```

Output:
UserId=12
UserId=41
UserId=200
UserId=6789

Some New Functional Interfaces in Java8

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

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

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

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

Predicate Interface

Predicate Interface

- Predicate is a new functional interface defined in java.util.function package
- It can be used in all the contexts where an object needs to be evaluated for a given test condition and a boolean value needs to be returned based on whether the condition was successfully met or not
- Whenever an object needs to be evaluated and a boolean value needs to be returned the Predicate functional interface can be used
- The user need not define his/her own predicate-type functional interface

java.util.function.Predicate source

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface

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

Example1

```
class BiggerThan5 implements Predicate<String>
{
@Override
public boolean test(String t) {
    return t.length()>5;
}
}

public class PredicateExample {
public static void main(String args[]) {

    String arr[]= {"Chennai","Pune","Goa","London","Reading"};
    BiggerThan5 b=new BiggerThan5();
   Arrays.stream(arr).filter(b).forEach(System.out::println);

}
}
```

Output:
Chennai
London
Reading

Default methods in Predicate

Default Method Name	Explanation
and()	It does logical AND of the predicate on which it is called with another predicate. Example: predicate1.and(predicate2)
or()	It does logical OR of the predicate on which it is called with another predicate. Example: predicate1.or(predicate2)
negate()	It does boolean negation of the predicate on which it is invoked. Example: predicate1.negate()

Example2 – and method with lambda expression

```
public class PredicateAndExample {

public static void main(String args[]) {

String arr[]= {"Mumbai","Delhi","Pune","Kolkatta","London","Paris","Pet"};


Predicate<String> p1=x->x.length()>3;

Predicate<String> p2=x->x.startsWith("P");

Arrays.stream(arr).filter(p1.and(p2)).forEach(System.out::println);

}

}
```

Output:
Pune
Paris

Example2 – or method with lambda expression

```
public class PredicateOrExample {

public static void main(String args[]) {

String arr[]= {"Mumbai","Delhi","Pune","Dog","London","Paris","Pet"};

Predicate<String> NameStartsWithP=x->x.startsWith("P");

Predicate<String> LengthGT5=x->x.length()>5;

Arrays.stream(arr).filter(NameStartsWithP.or(LengthGT5)).forEach(System.out::println);

}

}
```

Output:
Mumbai
Pune
London
Paris
Pet

Example2 – negate method with lambda expression

```
public class PredicateNegateExample {

public static void main(String args[]) {

List<String> originalList=Arrays.asList("Mumbai","Delhi","Pune","Kolkatta","London","Paris");


Predicate<String> p=x->x.length()>5;

for(String city:originalList)

if (p.negate().test(city))

System.out.println(city);

}}
```

Output:
Delhi
Pune
Paris

Consumer Interface

Consumer Interface

- **Consumer<T>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package
- Consumer can be used in all contexts where an object needs to be consumed i.e. taken as input, and some operation is to be performed on the object without returning any result
- **Example:** Object is taken as input to the printing function and the value of the object is printed
- Since Consumer is a functional interface, it can be used as the assignment target for a lambda expression or a method reference

java.util.function.Consumer source

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

- **accept()** method is the primary abstract method of the Consumer functional interface

- It's function descriptor being T -> ( ) i.e. accept() method takes an input of type T and returns no value

Example for Consumer function

Example 1:

```
public class Demo {
    public static void main(String[] args) {
            //takes an Integer object and prints it
            Consumer<Integer> c1 = i-> System.out.println(i);
            List<Integer> li = new ArrayList<Integer>();
            li.add(10);
            li.add(20);
            li.forEach(n->c1.accept(n));
    }
}
```

Output:

10
20

Example for Consumer function

Example 2:

```
public class Demo {
    public static void main(String[] args) {
        Consumer<Integer> c1 = i-> System.out.println(i);
        //takes a List of type Integer and multiplies each element by 2
        Consumer<List<Integer>> c2 = list ->{
        for (int j = 0; j < list.size(); j++)
            list.set(j, 2 * list.get(j));
        };
        List<Integer> li = new ArrayList<Integer>();
        li.add(10); li.add(20);
        c2.accept(li);
        li.forEach(n->c1.accept(n));
}}
```

Output:

20
40

Default methods in Consumer

Default Method Name	Explanation
andThen()	It returns a composed Consumer that performs in sequence.

Example for default method

Example:

```
public class Demo {
    public static void main(String[] args) {
        Consumer<String> c1 = i-> System.out.print(i);
        Consumer<String> c2 = c1.andThen( i-> System.out.println(" -> Hello "+i));
        List<String> li = new ArrayList<String>();
        li.add("admin");
        li.add("user");
        li.forEach(n->c2.accept(n));
    }
}
```

Output:

admin -> Hello admin
user -> Hello user

Supplier Interface

Supplier Interface

- **Supplier<T>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package.
- Supplier can be used in all contexts where there is no input but an output is expected.
- Since Supplier is a functional interface, it can be used as the assignment target for a lambda expression or a method reference.
- Supplier's function descriptor is **() ->T**
- This means that there is no input in the lambda definition and the return output is an object of type T.

java.util.function.Supplier source

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

- **get( )** method is the primary abstract method of the Supplier functional interface.

- Its function descriptor being ( ) **->**T i.e. get() method takes no input and returns an output of type T.

Example for Supplier function

Example 1:

```
public class Demo {
	public static void main(String[] args) {
		Supplier<Double> s1 = ()-> {
		    return Math.random()*4;
		};
	System.out.println(s1.get());
	}
}
```

Output:

<Returns a random number between 0-3>

Example for Supplier function

Example 2:

```
public class Demo {
	public static void main(String[] args) {
		Supplier<java.util.Date> s1 = ()-> {
		    return new java.util.Date();
		};
	System.out.println(s1.get());
	}
}
```

Output:

<Prints the current system date and time>

Function Interface

Function Interface

- Function interface is a functional interface
- A Function interface is more of a generic one that takes one argument and produces a result
- This has a Single Abstract Method (SAM) apply which accepts an argument of a type T and produces a result of type R
- One of the common use cases of this interface is Stream.map method

Example

```
public class FunctionInterfaceDemo {

public static void main(String[] args) {

                    int x = 4;

                    Function<Integer, Integer> fn1 = (num) ->(x * x);

                    System.out.println(fn1.apply(x));

                    String s="Welcome";

                    Function<String,Integer> fn2=(s1) ->  s1.length();

                    System.out.println(fn2.apply(s));

}}
```

Output
16
7

Thank you