

Predefined
functional
Interface

22nd dec Predefined Interfaces

23 December 2022 11:07

Functional interfaces

=====

=> If an interface contains only one abstract method then such interfaces are called as "Functional interface".

```
public interface java.util.function.Predicate<T> {
    public abstract boolean test(T);
    //default methods
    public java.util.function.Predicate<T> and(java.util.function.Predicate<? super T>);
    public java.util.function.Predicate<T> negate();
    public java.util.function.Predicate<T> or(java.util.function.Predicate<? super T>);
    //static method
    public static <T> java.util.function.Predicate<T> isEqual(java.lang.Object);
}
```

Usage of Predicate

=====

```
class MyPredicate implements Predicate<Integer>
{
    @Override
    public boolean test(Integer i){
        if (i>10)
            return true;
        else
            return false;
    }
}
```

Instead of writing a separate class, we can write lambda expression as shown below

=====

```
class Test {
    public static void main(String[] args){
        Predicate<Integer> p = i->i>10
        System.out.println(p.test(10)); //false
        System.out.println(p.test(100)); //true
    }
}
```

Write a Predicate to check whether the given String length is >=3 or not?

Instead of writing a separate class, we can write lambda expression as shown below

=====

```
class MyPredicate implements Predicate<String>
{
    @Override
    public boolean test(String name){
        if(name.length()>=3)
            return true;
        else
            return false;
    }
}

class Test {
    public static void main(String[] args){
        Predicate<String> p = name -> name.length() >= 3;
        System.out.println(p.test("PWC")); //true
        System.out.println(p.test("CS")); //false
    }
}
```

```
}
```

default methods available as utility methods for developer

```
=====
```

```
public default Predicate<T> and(Predicate p);
public default Predicate<T> negate();
public default Predicate<T> or(Predicate p);
```

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

        int[] arr = {0,5,10,15,20,25,30};
        Predicate<Integer> p1 = i-> i>10;
        System.out.println("Elements greater than 10 are :: ");
        m1(p1,arr);
        Predicate<Integer> p2 = i-> i%2==0;
        System.out.println("Elements which are even no :: ");
        m1(p2,arr);

        System.out.println("Eleemnts which are greater than 10 and should be even no");
        m1(p1.and(p2),arr);
        System.out.println("Eleemnts which are greater than 10 or should be even no");
        m1(p1.or(p2),arr);
        System.out.println("The elements which are not even are :: ");
        m1(p2.negate(),arr);
    }
    public static void m1(Predicate<Integer> p , int[] x){
        for (int ele: x )
            if (p.test(ele))//ele-> ele>10
                System.out.println(ele);
    }
}
```

Function()

```
=====
```

T-> input type

R-> return type

```
public interface java.util.function.Function<T, R> {
    // 1 abstract method
    public abstract R apply(T);

    //default methods
    public <V> java.util.function.Function<V, R> compose(java.util.function.Function<? super V, ? extends T>);
    public <V> java.util.function.Function<T, V> andThen(java.util.function.Function<? super R, ? extends V>);

    //static method
    public static <T> java.util.function.Function<T, T> identity();
}
```

Writing a code using Implementation class

```
=====
```

```
class MyFunction implements Function<String,Integer>
{
    @Override
    public Integer apply(String name){
        return name.length();
    }
}

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

```

        Function<String,Integer> f = new MyFunction();
        int output = f.apply("sachin");
        System.out.println(output);
        System.out.println("sachin".length());
    }
}

```

Coding using Lambda Expression

=====

```

public class Test {
    public static void main(String[] args){
        Function<String,Integer> f = name -> name.length();
        int output = f.apply("sachin");
        System.out.println(output);
    }
}

```

Note:

When to go for Predicate and When to go for Function?

Predicate -> To implement some conditional checks we should go for Predicate

Function -> To perform some operation and to return some result we should go for Function.

MethodReference(::) and Constructor reference(::)

=====

:: ==> Scope resolution operator

syntax for method reference

=====

1. static method

ClassName::methodName

2. instance method

object:: methodName

3.class :: new(Constructor Reference)

```

public class Test {
    public static void m1(){
        for (int i = 1;i<=10 ;i++ )
        {
            System.out.println("child thread");
        }
    }
    public static void main(String[] args) throws Exception{

        //using method reference binded the method call of run() of interface Runnable
        Runnable r = Test::m1;
        Thread t =new Thread(r);
        t.start();
        for (int i = 1;i<=10 ;i++ )
        {
            System.out.println("main thread");
            Thread.sleep(1000);
        }
    }
}

```

Constructor reference

```
class Sample {
    private String s;
    Sample(String s) {
        this.s = s;
        System.out.println("Constructor executed...." + s);
    }
}

@FunctionalInterface
interface Interf {
    public Sample get(String s);
}

public class Test {
    public static void main(String[] args) {
        Interf i = s -> new Sample(s);
        i.get("from lambda expression...");
        System.out.println();

        // constructor reference
        Interf i1 = Sample::new;

        i1.get("from constructor reference....");
    }
}
```

Example of Method reference

=====

```
@FunctionalInterface
interface Interf {
    public void m1(int i);
}

public class Test {
    // Logic coded by other developer
    public void m2(int i) {
        System.out.println(i * i);
        System.out.println("logic coming from method reference...");
    }
    public static void main(String[] args) {
        Interf i = x -> System.out.println(x);
        i.m1(10);
        System.out.println();
        // method reference(binding the body of m2() to abstract method m1)
        Interf i1 = new Test():m2;
        i1.m1(20);
    }
}
```

Eg: To demonstrate the usage of `forEach()` to print the elements of ArrayList

```
// public void forEach(java.util.function.Consumer<? super E>);
// public abstract void accept(T t)
```

```
class MyConsumer implements Consumer<String> {

    @Override
    public void accept(String name) {
        System.out.println("accept method got called...");
        System.out.println(name);
    }
}

public class Test {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<String>();
```

```
names.add("sachin");
names.add("dhoni");
names.add("kohli");
names.add("dravid");

// Traditional approach
Consumer<String> consumer = new MyConsumer();
names.forEach(consumer);
System.out.println();

// Lambda expression
names.forEach(name -> System.out.println(name));
System.out.println();

// method reference
names.forEach(System.out::println);
```

```
}
```

```
}
```