Functional Interface

Predicate (Functional interface)
----
take input perform some conditional check operation and return boolean values then we should go for Predicate (FI).

where ever conditional check is required then we should go for Predicate(FI).
the return type of predicate is -boolean

Syntax:

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

predicate can take only one type of parameter.

Q)write a predicate whether the String is > 5 or not ?

Ex:

```
Predicate<String> p = s -> s.length() > 5
System.out.println(p.test("anand"));//false
```

<String>
this is the Type
Parameter

Ex:

```
Predicate<Integer> p = i -> i % 2 == 0;
System.out.println(p.test(10));
```

output: true

Ex:

```
String[] s = { "anand", "venky", "vijay", "anji", "arun", "kamal" };
//Predicate<String> p = i -> i.length() > 4;
Predicate<String> p1 = i -> i.length() % 2 == 0;

for (String s1 : s) {
if (p1.test(s1)) {
```

```
if (p1.test(s1)) {
System.out.println(s1);
}
}

}
```

output:

```
anji
arun
```

Ex:

```
int[] x = { 2, 4, 7, 5, 89, 34, 5, 7, 6 };
Predicate<Integer> p = i -> i > 2;
Predicate<Integer> p1 = i -> i % 2 == 0;
for (int x1 : x) {
if (p.and(p1).test(x1)) {
System.out.println(x1);
}
}
}
```

Output :

```
4
34
6
```

Ex:Write a Predicate Employee salary is > 5000 or not ?

```
import java.util.ArrayList;
import java.util.function.Predicate;

class Employee {
String name;
double salary;

public Employee(String name, double salary) {
super();
this.name = name;
this.salary = salary;
}

@Override
public String toString() {
return "Employee [name=" + name + ", salary=" + salary + "]";
}

}

public class EmployeeExample {
public static void main(String[] args) {
ArrayList<Employee> l = new ArrayList<Employee>();
l.add(new Employee("anand", 10000));
l.add(new Employee("ashok", 18297));
l.add(new Employee("amurth", 76767));
l.add(new Employee("anji", 89798));
l.add(new Employee("ghouse", 876876));
l.add(new Employee("vamsi", 8977));
l.add(new Employee("salman", 1000));
l.add(new Employee("salman", 1000));
l.add(new Employee("kalki", 98787));
l.add(new Employee("venkat", 77676));

Predicate<Employee> p = e -> e.salary > 5000;
for (Employee e1 : l) {
if (p.test(e1)) {
System.out.println(e1.name + ":" + e1.salary);
}
}

}
}
```

Output :

```
anand:10000.0
ashok:18297.0
amurth:76767.0
anji:89798.0
ghouse:876876.0
vamsi:8977.0
kalki:98787.0
venkat:77676.0
```

Ex:

```
import java.util.ArrayList;
import java.util.function.Predicate;

class Employee {
String name;
double salary;

public Employee(String name, double salary) {
super();
this.name = name;
this.salary = salary;
}

@Override
public String toString() {
return "Employee [name=" + name + ", salary=" + salary + "]";
}

}

public class EmployeeExample {
public static void main(String[] args) {
ArrayList<Employee> l = new ArrayList<Employee>();
l.add(new Employee("anand", 10000));
l.add(new Employee("ashok", 18297));
l.add(new Employee("amurth", 76767));
l.add(new Employee("anji", 89798));
l.add(new Employee("ghouse", 876876));
l.add(new Employee("vamsi", 8977));
l.add(new Employee("salman", 1000));
l.add(new Employee("kalki", 98787));
l.add(new Employee("venkat", 77676));
Predicate<Employee> p = e -> e.salary > 5000;
Predicate<Employee> p1 = e -> e.name.length() % 2 == 0;
for (Employee e1 : l) {
if (p.and(p1).test(e1)) {
System.out.println(e1.name + ":" + e1.salary);
}
}

}
}
```

output :

```
amurth:76767.0
anji:89798.0
ghouse:876876.0
venkat:77676.0
```

Predicate Joining

and,or,negate

The Functional Interface **PREDICATE** is defined in the *java.util.Function package*. It improves manageability of code, helps in unit-testing them separately, and contain some methods like:

1. **isEqual(Object targetRef) :** Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

```
static  Predicate isEqual(Object targetRef)
Returns a predicate that tests if two arguments are
equal according to Objects.equals(Object, Object).
T : the type of arguments to the predicate
Parameters:
targetRef : the object reference with which to
compare for equality, which may be null
Returns: a predicate that tests if two arguments
are equal according to Objects.equals(Object, Object)
```

2. **and(Predicate other) :** Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

```
default Predicate and(Predicate other)
Returns a composed predicate that represents a
short-circuiting logical AND of this predicate and another.
Parameters:
other: a predicate that will be logically-ANDed with this predicate
Returns : a composed predicate that represents the short-circuiting
logical AND of this predicate and the other predicate
Throws: NullPointerException - if other is null
```

3. **negate() :** Returns a predicate that represents the logical negation of this predicate.

```
default Predicate negate()
Returns:a predicate that represents the logical
negation of this predicate
```

4. **or(Predicate other) :** Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

```
default Predicate or(Predicate other)
Parameters:
other : a predicate that will be logically-ORed with this predicate
Returns:
a composed predicate that represents the short-circuiting
logical OR of this predicate and the other predicate
Throws : NullPointerException - if other is null
```

5. **test(T t) :** Evaluates this predicate on the given argument.boolean test(T t)

```
test(T t)
Parameters:
t - the input argument
Returns:
true if the input argument matches the predicate, otherwise false
```

p1.and(p2) -both condition should satisfy.
p1.or(p2) -atleast one condition satisfy or both.
p1.negate( ) -p1.nagate( ) -reverse of p1.

Ex:

```
import java.util.function.Predicate;

public class PredicateExample2 {
public static void main(String[] args) {
int[] a = { 12, 34, 45, 14, 35, 78, 98 };
Predicate<Integer> p = i -> i % 2 == 0;
Predicate<Integer> p1 = i -> i > 30;
System.out.println("the numbers are even and >30 ");
for (int a1 : a) {
//and-logical and,or -logical or  ||,negate
if (p.and(p1).test(a1)) {
System.out.println(a1);
}
}

}
}
```

output :

```
the numbers are even and >30
34
78
98
```

Ex: or

```
import java.util.function.Predicate;

public class PredicateOr {
public static void main(String[] args) {
int[] a = { 12, 34, 45, 14, 35, 78, 98 };
Predicate<Integer> p = i -> i % 2 == 0;
Predicate<Integer> p1 = i -> i > 30;
System.out.println("the numbers are even or >30 ");
for (int a1 : a) {
if (p.or(p1).test(a1)) {
System.out.println(a1);
}
}

}
}
```

output:

```
45
14
35
78
98
```

Ex: negate

```
import java.util.function.Predicate;

public class Predicatenegate {

public static void main(String[] args) {
int[] a = { 12, 34, 45, 14, 35, 78, 98 };
Predicate<Integer> p = i -> i % 2 == 0;
// Predicate<Integer> p1 = i -> i > 30;
System.out.println("the numbers are not even ");
for (int a1 : a) {
if (p.negate().test(a1)) {
System.out.println(a1);
}
}

}
}
```

output:

```
the numbers are not even
45
35
```

Function (FI)

Function-(Functional Interface)

input ->perform some operation ->output

4 ->square operation->16

The result need not be boolean type but anything.

Syntax:

```
Interface Function <T,R>{
T-Input Type
R-Retrun Type
public abstract R apply(T t);
}
```

Ex:

```
public class FunctionExample {
public static void main(String[] args) {
Function<Integer, Integer> f = i -> i * i;
System.out.println(f.apply(4));

}
}
```

output:

16

Ex:2

```
public class FunctionExample2 {
public static void main(String[] args) {
Function<String, String> f = s -> s.toUpperCase().trim();
System.out.println(f.apply(" sv college"));

}
}
```

trim( )- can be used to remove the blank space from begin of the String and Ending of the String and not in-between of the String.

output :

SV COLLEGE

Q)How to find a Grade for Student ?

```
package com.function;

import java.util.ArrayList;
import java.util.function.Function;

class Student {
String name;
int marks;

public Student(String name, int marks) {
super();
this.name = name;
this.marks = marks;
}

@Override
public String toString() {
return "Student [name=" + name + ", marks=" + marks + "]";
}

}

public class FunctionGrade {
public static void main(String[] args) {
Function<Student, String> f = s -> {
int marks = s.marks;
String grade = "";
if (marks >= 80)
grade = "A[Distiction]";
else if (marks > 60)
grade = "B[First Class]";
else if (marks > 50)
grade = "C[Second Class]";
else if (marks > 35)
grade = "D[Third Class]";
else
grade = "E[Failed]";

return grade;

};
Student[] s = { new Student("ammer", 30), new Student("ashok", 45), new Student("vinod", 85),
new Student("srinivas", 55), new Student("shankar", 65), new Student("ram", 95),
new Student("prasad", 20), };

for (Student s1 : s) {
System.out.println(s1.name);
System.out.println(s1.marks);
System.out.println(f.apply(s1));
System.out.println();
}
}

}
```

output:

```
ammer
30
E[Failed]

ashok
45
D[Third Class]

vinod
85
A[Distiction]
```

```
A[Distiction]

srinivas
55
C[Second Class]

shankar
65
B[First Class]

ram
95
A[Distiction]

prasad
20
E[Failed]
```

Q)if the Student Marks > 60 ?

```
import java.util.ArrayList;
import java.util.function.Function;
import java.util.function.Predicate;

class Student {
String name;
int marks;

public Student(String name, int marks) {
super();
this.name = name;
this.marks = marks;
}

@Override
public String toString() {
return "Student [name=" + name + ", marks=" + marks + "]";
}

}

public class FunctionGrade {
public static void main(String[] args) {
Function<Student, String> f = s -> {
int marks = s.marks;
String grade = "";
if (marks >= 80)
grade = "A[Distiction]";
else if (marks > 60)
grade = "B[First Class]";
else if (marks > 50)
grade = "C[Second Class]";
else if (marks > 35)
grade = "D[Third Class]";
else
grade = "E[Failed]";

return grade;
};

Predicate<Student> p = s1 -> s1.marks > 60;
Student[] s = { new Student("ammer", 30), new Student("ashok", 45), new Student("vinod", 85),
new Student("srinivas", 55), new Student("shankar", 65), new Student("ram", 95),
new Student("prasad", 20), };

for (Student s1 : s) {
if (p.test(s1)) {
System.out.println(s1.name);
System.out.println(s1.marks);
System.out.println(f.apply(s1));
System.out.println();
}
}
}

}
```

output:

```
vinod
85
A[Distiction]

shankar
65
B[First Class]

ram
95
A[Distiction]
```

FunctionChaining

1)f1.andThen(f2).apply(i); first f1 followed by f2

after applying f1 function for the result f2 function will applied ,two functions we can combine together to form more complex functions

2)f1.compose(f2).apply(i); first f2 and then f1

the difference is Syntactical Trick

Ex:

```
public class FunctionandThencompose {
public static void main(String[] args) {
Function<Integer, Integer> f = i -> i * 2;
Function<Integer, Integer> f1 = i -> i * i * i;
System.out.println(f.andThen(f1).apply(2));
System.out.println(f.compose(f1).apply(2));

}
}
```

```
output:
64
16
```