Two Arguments Functional Interface

1) BiPredicate

-> Normal Predicate can take only one input Argument and preform some conditional check, for this requirement we should go for Predicate.

-> some times our program requirement is we have to take 2 input argument and preform some conditional check , for this requirement we should go for BiPredicate.

BiPredicate is exactly same as Predicate except that it will take 2 input arguments

two input arguments

Syntax:

```
Interface BiPredicate <T1 ,T2>
{
public boolean test ( T1 t1, T2 t2 );

//Remaining Default methods :   .and   ,  .or ,    .negate
}
```

Ex: To check the sum of 2 given integers is even or not by using BiPredicate ?

```
package com.bipredicate;

import java.util.function.BiPredicate;

public class BiPredicateExample {
public static void main(String[] args) {
BiPredicate<Integer, Integer> p = (a, b) -> (a + b) % 2 == 0;
System.out.println(p.test(10, 20));
System.out.println(p.test(15, 10));
}
}
```

output:
true
false

2) BiFunction

Function< T , R > - takes an input ->preform some operation ->produce some result [the result which is need not to boolean type.
Interface Function<T , R >

{
public R apply(T t);
//Default method : .andThen
}
T ->type of the argument of the function.
R ->Result of the function.

BiFunction

Interface BiFunction<T,U,R>

{

public R apply (T t ,U u);

}

- **Type Parameters:**

  T - the type of the first argument to the function

  U - the type of the second argument to the function

  R - the type of the result of the function

Represents a function that accepts two arguments and produces a result.
This is the two-arity specialization of `Function`.

This is a functional interface
whose functional method is `apply(Object, Object)`.

ex:

we are providing eno,ename arguments and return Employee object

```
package com.bifunction;

import java.util.ArrayList;
import java.util.function.BiFunction;

class Employee {
Integer eno;
String name;

public Employee(Integer eno, String name) {
super();
this.eno = eno;
this.name = name;
```

```
}

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

}

public class BiFunctionExample {
public static void main(String[] args) {
ArrayList<Employee> l = new ArrayList<Employee>();
BiFunction<Integer, String, Employee> f = (en, ena) -> new Employee(en, ena);
l.add(f.apply(100, "anand"));
l.add(f.apply(200, "ganesh"));
l.add(f.apply(222, "ramesh"));
l.add(f.apply(103, "suresh"));
l.add(f.apply(300, "uday"));
l.add(f.apply(111, "ram"));
for (Employee e1 : l) {
System.out.println("employee no :" + e1.eno);
System.out.println("employee name :" + e1.name);
System.out.println();
}
}
}
```

output:

```
employee no :100
employee name :anand

employee no :200
employee name :ganesh

employee no :222
employee name :ramesh

employee no :103
employee name :suresh

employee no :300
employee name :uday

employee no :111
employee name :ram
```

ex:2

```
package com.bifunction;

import java.util.function.BiFunction;

class Student {
int sno;
String sname;
```

```
public Student(int sno, String sname) {
super();
this.sno = sno;
this.sname = sname;
}

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

}

public class BiFunction2 {
public static void main(String[] args) {
BiFunction<Integer, String, Student> f = (sno, sname) -> new Student(sno, sname);
System.out.println(f.apply(1010, "dandi"));
System.out.println(f.apply(1012, "anand"));

}
}
```

output:
Student [sno=1010, sname=dandi]
Student [sno=1012, sname=anand]

BiConsumer(FI)

Consumer (FI) -> Consumer < T > - takes an input ->perform some operation -> it won't return any thing.

Syntax:

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

BiConsumer (FI):

```
public interface BiConsumer<T,U>
```

Represents an operation that accepts two input arguments and returns no result. This is the two-arity specialization of `Consumer`.
Unlike most other functional interfaces, `BiConsumer` is expected to operate via side-effects.

This is a functional interface
whose functional method is `accept(Object, Object)`.

Type Parameters:

T - the type of the first argument to the operation

U - the type of the second argument to the operation

Ex:
All the Employee Salary getting 500 bonus?

```
package com.biconsumer;

import java.util.ArrayList;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

class Employee {
String name;
double salary;

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

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

}

public class BiConsumerExample {
public static void main(String[] args) {
ArrayList<Employee> l = new ArrayList<Employee>();
BiConsumer<Employee, Double> c = (s, d) -> s.salary = s.salary + d;
Consumer<Employee> c1 = s -> {
System.out.println("student name :" + s.name);
System.out.println("studenmt salary :" + s.salary);
System.out.println();
};
get(l);
for (Employee s1 : l) {
c.accept(s1, (double) 500);
```

```
}
for (Employee s2 : l) {
c1.accept(s2);
}

}

private static void get(ArrayList<Employee> l) {
l.add(new Employee("anand", 2000));
l.add(new Employee("dandi", 4000));
l.add(new Employee("ramesh", 5000));
l.add(new Employee("suresh", 8000));
l.add(new Employee("ganesh", 10000));
l.add(new Employee("mahesh", 1000));

}

}
```

```
student name anand
studenmt salary 2500.0

student name dandi
studenmt salary 4500.0

student name ramesh
studenmt salary 5500.0

student name suresh
studenmt salary 8500.0

student name ganesh
studenmt salary 10500.0

student name mahesh
studenmt salary 1500.0
```