Practical Functional Interface In Java – Part -1

# Consumer – Only Input, No Output – Consumes only

**Interface Consumer<T>**

* **Type Parameters:**

T - the type of the input to the operation

Represents an operation that accepts a single input argument and returns no result.  This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

|  |  |
| --- | --- |
| void | [**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t) : Performs this operation on the given argument. |
| default [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#andThen-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> after)  Returns a composed Consumer that performs, in sequence, this operation followed by the after operation. |

Basic Examples

Consumer<String> consumer1 = (s) -> System.***out***.println(s);

consumer1.accept("ABCD");

Consumer<List<Integer>> iconsumer1 =

(l) -> l.forEach(e -> System.***out***.print(e + "\t"));

List<Integer> intList = Arrays.*asList*(1, 2, 3, 4, 5, 6);

iconsumer1.accept(intList);

Consumer<String> c = (x) -> System.***out***.println("\n"+x.toUpperCase());

c.accept("Java2s.com");

List<Student> studentList =

Arrays.*asList*(

**new** Student("John", "Abraham"),

**new** Student("Abraham", "Lincoln"),

**new** Student("Vidya", "Balan"));

**Consumer<List<Student>> studentConsumer =**

**(sList) -> sList.forEach((e) -> System.*out*.println("\n" + e));**

**studentConsumer.accept(studentList);**

**OUTPUT**

ABCD

1 2 3 4 5 6

JAVA2S.COM

Student [firstName=John, surName=Abraham]

Student [firstName=Abraham, surName=Lincoln]

Student [firstName=Vidya, surName=Balan]

# BiConsumer – Only Input, No Output – Consumes only

**Interface BiConsumer<T,U> -** Only Input, No Output – Consumes only

* **Type Parameters:**

T - the type of the first argument to the operation

U - the type of the second argument to the operation

|  |  |
| --- | --- |
| void | [accept](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html#accept-T-U-)([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html) t, [U](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html) u) : Performs this operation on the given arguments. |
| default **[BiConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html" \o "interface in java.util.function)**<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html),[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)> | [andThen](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html#andThen-java.util.function.BiConsumer-)([BiConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html" \o "interface in java.util.function)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html),? super [U](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)> after)  Returns a composed BiConsumer that performs, in sequence, this operation followed by the afteroperation. |

An example is given below.

**public** **class** ConsumerTest1 {

**public** **static** **void** showDetails(String name, Integer age) {

System.***out***.println(name + " " + age);

}

**public** **static** **void** main(String[] args) {

BiConsumer<String, Integer> biCon = ConsumerTest1::*showDetails*;

biCon.accept("Rama", 20);

biCon.accept("Shyam", 25);

**BiConsumer<String, Integer> bicon1 = (a, b) -> System.*out*.println(a + " : " + b);**

**bicon1.accept("John", 23);**

// Using lambda expression

**BiConsumer<String, Integer> biCon2 = (name, age) -> System.*out*.println(name + " " + age);**

**biCon2.accept("Peter", 28);**

}

}

OUTPUT

Rama 20

Shyam 25

John : 23

Peter 28

Another example using Map.

**public** **class** ConsumerTest1 {

**public** **static** **void** showDetails(Map<Integer, String> map, String mapName) {

System.***out***.println("----------" + mapName + " records-----------");

**map.forEach((key, val) -> System.*out*.println(key + " " + val));**

}

**public** **static** **void** main(String[] args) {

Map<Integer, String> map = **new** HashMap<Integer, String>();

map.put(100, "Mohan");

map.put(110, "Sujeet");

map.put(115, "Tom");

map.put(120, "Danish");

// Referring method

**BiConsumer<Map<Integer, String>, String> biCon = ConsumerTest1::*showDetails*;**

**biCon.accept(map, "Student");**

// It can also be written as

**BiConsumer<Map<Integer, String>, String> biMap =**

**(myMap, someName) -> myMap.forEach((k, v) -> System.*out*.println(k + "<--->" + v));**

**biMap.accept(map, "SomeName");**

}

}

OUTPUT

----------Student records-----------

115 Tom

100 Mohan

120 Danish

110 Sujeet

115<--->Tom

100<--->Mohan

120<--->Danish

A Map’s key and value can be printed in the following manner.

Map<Integer, String> map = **new** HashMap<>();

map.put(1, "A");

map.put(2, "B");

map.put(3, "C");

**BiConsumer<Integer, String> biConsumer =**

**(key, value) -> System.*out*.println("Key:" + key + " Value:" + value);**

**map.forEach(biConsumer);**

OUTPUT

Key:1 Value:A

Key:2 Value:B

Key:3 Value:C

# Function – One Input and One Output

**Interface Function<T,R>**

* **Type Parameters:**

T - the type of the input to the function

R - the type of the result of the function

|  |  |
| --- | --- |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),V> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),? extends V> after)  Returns a composed function that first applies this function to its input, and then applies the afterfunction to the result. |
| [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) | [**apply**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-)**(**[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)**t) : Applies this function to the given argument.** |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<V,[**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> | [**compose**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#compose-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super V,? extends [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> before)  Returns a composed function that first applies the before function to its input, and then applies this function to the result. |
| **static <T>**[**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)**<T,T>** | [**identity**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#identity--)**() :** Returns a function that always returns its input argument. |

Examples are given below.

Function<Integer, Integer> sum = (a) -> a \* a;

**int** sumValue = sum.apply(10);

System.***out***.println("Sum Value : " + sumValue); //100

Function<String,Integer> number = s -> Integer.*valueOf*(s);

System.***out***.println("Value : "+number.apply("10"));//10

**Function<Student, String> namesOnly = (e) -> e.getFirstName();**

List<Student> sList =

Arrays.*asList*(**new** Student("John", "Abraham"), **new** Student("Vidya", "Balan"));

**for** (Student stud : sList) {

**System.*out*.println("First Name of the sutdent : " + namesOnly.apply(stud));**

}

OUTPUT

First Name of the sutdent : John

First Name of the sutdent : Vidya

Another type of example.

**Function<Integer, String> function =**

**(t) -> {**

**if (t % 2 == 0) {**

**return t + " is even number";**

**} else {**

**return t + " is odd number";**

**}**

**};**

System.***out***.println(function.apply(5));// 5 is odd number

System.***out***.println(function.apply(8));// 8 is even number

Function<Integer, Integer> function1 = t -> (t - 5);

Function<Integer, Integer> function2 = t -> (t \* 2);

*//Using andThen() method*

**int** a=**function1.andThen(function2).apply(50);**

System.out.println(a); //90

**int** c=**function1.compose(function2).apply(50);**

System.out.println(c); //95

# BiFunction – Two Input and one OutPut

**Interface BiFunction<T,U,R>**

* **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](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html).

|  |  |
| --- | --- |
| **default <V> [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html" \o "interface in java.util.function)<**[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)**,**[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)**,V>** | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html#andThen-java.util.function.Function-)**(**[**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)**<? super**[**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)**,? extends V> after)**  **Returns a composed function that first applies this function to its input, and then applies the after function to the result.** |
| [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) | [**apply**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html#apply-T-U-)**(**[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)**t,**[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)**u): Applies this function to the given arguments.** |

Examples

**BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;**

**// Implement add using apply()**

**System.out.println("Sum = " + add.apply(2, 3)); //5**

**// BiFunction to multiply 2 numbers**

**BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a \* b;**

**// Implement add using apply()**

**System.out.println("Product = " + multiply.apply(2, 3));//6**

# Predicate

**Interface Predicate<T>**

* **Type Parameters:**

T - the type of the input to the predicate

**Functional Interface:**

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

|  |  |
| --- | --- |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**and**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#and-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | [**isEqual**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#isEqual-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) targetRef)  Returns a predicate that tests if two arguments are equal according to **[Objects.equals(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html" \l "equals-java.lang.Object-java.lang.Object-)**. |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**negate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#negate--)()  Returns a predicate that represents the logical negation of this predicate. |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**or**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#or-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| **boolean** | [**test**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-)**(**[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)**t):Evaluates this predicate on the given argument.** |

Examples are given below

Predicate<Integer> lesserthan = i -> (i < 18);  
*// Calling Predicate method*System.***out***.println(lesserthan.test(10));//True

Predicate<Integer> greaterThanTen = (i) -> i > 10;  
*// Creating predicate*Predicate<Integer> lowerThanTwenty = (i) -> i < 20;  
**boolean** result = greaterThanTen.and(lowerThanTwenty).test(15);  
System.***out***.println(result);***//True*** *// Calling Predicate method***boolean** result2 = greaterThanTen.and(lowerThanTwenty).negate().test(15);  
System.***out***.println(result2**);*//False***