Functional Interfaces

java.util.function (Java Platform SE 8) (oracle.com)

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of `boolean`-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two `double`-valued operands and producing a `double`-valued result.
DoubleConsumer	Represents an operation that accepts a single `double`-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one `double`-valued argument.
DoubleSupplier	Represents a supplier of `double`-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single `double`-valued operand that produces a `double`-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two `int`-valued operands and producing an `int`-valued result.
IntConsumer	Represents an operation that accepts a single `int`-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.
IntPredicate	Represents a predicate (boolean-valued function) of one `int`-valued argument.
IntSupplier	Represents a supplier of `int`-valued results.

IntToDoubleFunction	Represents a function that accepts an int-valued argument and produces a double-valued result.
IntToLongFunction	Represents a function that accepts an int-valued argument and produces a long-valued result.
IntUnaryOperator	Represents an operation on a single `int`-valued operand that produces an `int`-valued result.
LongBinaryOperator	Represents an operation upon two `long`-valued operands and producing a `long`-valued result.
LongConsumer	Represents an operation that accepts a single `long`-valued argument and returns no result.
LongFunction<R>	Represents a function that accepts a long-valued argument and produces a result.
LongPredicate	Represents a predicate (boolean-valued function) of one `long`-valued argument.
LongSupplier	Represents a supplier of `long`-valued results.
LongToDoubleFunction	Represents a function that accepts a long-valued argument and produces a double-valued result.
LongToIntFunction	Represents a function that accepts a long-valued argument and produces an int-valued result.
LongUnaryOperator	Represents an operation on a single `long`-valued operand that produces a `long`-valued result.
ObjDoubleConsumer<T>	Represents an operation that accepts an object-valued and a `double`-valued argument, and returns no result.
ObjIntConsumer<T>	Represents an operation that accepts an object-valued and a `int`-valued argument, and returns no result.
ObjLongConsumer<T>	Represents an operation that accepts an object-valued and a `long`-valued argument, and returns no result.
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	Represents a supplier of results.
ToDoubleBiFunction<T,U>	Represents a function that accepts two arguments and produces a double-valued result.
ToDoubleFunction<T>	Represents a function that produces a double-valued result.
ToIntBiFunction<T,U>	Represents a function that accepts two arguments and produces an int-valued result.
ToIntFunction<T>	Represents a function that produces an int-valued result.
ToLongBiFunction<T,U>	Represents a function that accepts two arguments and produces a long-valued result.
ToLongFunction<T>	Represents a function that produces a long-valued result.
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.

Package java.util.function Description

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

```
// Assignment context

Predicate<String> p = String::isEmpty;


// Method invocation context

stream.filter(e -> e.getSize() > 10)...


// Cast context

stream.map((ToIntFunction) e -> e.getSize())...
```

The interfaces in this package are general purpose functional interfaces used by the JDK, and are available to be used by user code as well. While they do not identify a complete set of function shapes to which lambda expressions might be adapted, they provide enough to cover common requirements. Other functional interfaces provided for specific purposes, such as `FileFilter`, are defined in the packages where they are used.

The interfaces in this package are annotated with `FunctionalInterface`. This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.

Functional interfaces often represent abstract concepts like functions, actions, or predicates. In documenting functional interfaces, or referring to variables typed as functional interfaces, it is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object". When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...", this is understood to mean a *non-null* reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.

The functional interfaces in this package follow an extensible naming convention, as follows:

- There are several basic function shapes, including `Function` (unary function from `T` to `R`), `Consumer` (unary function from `T` to `void`), `Predicate` (unary function from `T` to `boolean`), and `Supplier` (nilary function to `R`).
- Function shapes have a natural arity based on how they are most commonly used. The basic shapes can be modified by an arity prefix to indicate a different arity, such as `BiFunction` (binary function from `T` and `U` to `R`).
- There are additional derived function shapes which extend the basic function shapes, including `UnaryOperator` (extends `Function`) and `BinaryOperator` (extends `BiFunction`).
- Type parameters of functional interfaces can be specialized to primitives with additional type prefixes. To specialize the return type for a type that has both generic return type and generic arguments, we prefix `ToXxx`, as in `ToIntFunction`. Otherwise, type arguments are specialized left-to-

right, as in `DoubleConsumer` or `ObjIntConsumer`. (The type prefix `Obj` is used to indicate that we don't want to specialize this parameter, but want to move on to the next parameter, as in `ObjIntConsumer`.) These schemes can be combined, as in `IntToDoubleFunction`.

- If there are specialization prefixes for all arguments, the arity prefix may be left out (as in `ObjIntConsumer`).

Since:
1.8
See Also:
`FunctionalInterface`

Main four category

Function	Predicate	Consumer	Supplier
BiFunction	Predicate	LongConsumer	LongSupplier
LongFunction	BiPredicate	IntConsumer	IntSupplier
IntToDoubleFunction	…	…	DoubleSupplier
…	…	…	…

Function:

Function (Java Platform SE 8) (oracle.com)

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

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

Predicate

Predicate (Java Platform SE 8) (oracle.com)

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

Consumer

Consumer (Java Platform SE 8) (oracle.com)

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

Supplier

Supplier (Java Platform SE 8) (oracle.com)

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

Java 7 way

```
package com.hdfc.java8;

interface MyInterface{
    void display();
}

class MyClass implements MyInterface{

    @Override
    public void display() {
        System.out.println("Hello World!");
    }
}

//Object Oriented, java -7
public class Java7Way {
    public static void main(String[] args) {
        MyInterface object = new MyClass();
        //MyClass output = new MyClass();
        object.display();

        //List interface
        //ArrayList  - implemation of List
        //List<String> list = new ArrayList<>();
    }
}
```

Java 8 way

```
package com.hdfc.java8;

interface MyJava8Interface{
    void display();
}

public class Java8Way {
    public static void main(String[] args) {
        //MyJava8Interface output = () -> System.out.println("Hello World!");
        //output.display();

        MyJava8Interface myCode = getMyCode();
        myMethod(myCode);

    }

    public static void myMethod(MyJava8Interface output ){
        output.display();
    }


    public static MyJava8Interface getMyCode(){
        return () -> System.out.println("Hello World!");
    }
}
```

```
package com.hdfc.java8;


import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.function.*;

public class Java8MoreExample {

    public static void main(String[] args) {

        Function<Integer, Integer> doubleTheNumber = i -> i * 2;
        int i = doubleTheNumber.apply(100);
        System.out.println(i);


        Function<String, Integer> stringLength = s -> s.length();
        int helloStringLength = stringLength.apply("Hello");
        System.out.println(helloStringLength);

        //Integer -> int
        //int -> Integer
        //boxing and autoboxing is done by Java

        //a+b  -> return c
        BiFunction<Integer, Integer, Integer> myAddFunction = (a, b) -> a + b;

        Integer result = myAddFunction.apply(100, 200);
        System.out.println(result);


        //you have 2 and more parameter -> Collection?
```

```
        Function<List<Integer>, Integer> myListAddFunction = list -> {
            int sum = 0;
            for (Integer number : list) {
                sum = sum + number;
            }
            return sum;
        };

        Integer result2 = myListAddFunction.apply(List.of(100, 200, 300));
        System.out.println(result2);


        List<Integer> list = new ArrayList<>();//mutable
        list.add(1);
        list.add(2);
        list.add(3);
        List<Integer> unmodifiableList = Collections.unmodifiableList(list);//java-7 way, immuta-
ble converted
        //unmodifiableList.add(4);//.UnsupportedOperationException


        //java -9
        List<Integer> list2 = List.of(1, 2, 3, 4, 5, 6, 8); //immutable
        //list2.add(9);//UnsupportedOperationException

        /* Predicate */

        //isEvenNumber , isOddNumber, isEvenLength "Hello", isEqualTo "Hello"
        Predicate<Integer> isEvenNumberFunction = num -> num%2==0;

        boolean is3Even = isEvenNumberFunction.test(3);
        System.out.println(is3Even);


        /* Consumer */
        //just print,log
        Consumer<String> consumerFunction = s -> System.out.println(s);
        consumerFunction.accept("Hello World!");
        consumerFunction.accept("India is my country!");

        /* Supplier */
        //to get
        //get me doubleNumber
        Supplier<Double> supplierFunction = () -> 5.5D;
        Double aDouble = supplierFunction.get();
        System.out.println(aDouble);
    }
}
```

```
<> IntFunction = R apply(int value);
------------------------------
solve right part
<> IntFunction = public Integer apply(Integer input){
                                    return input*3;
                                }

<> IntFunction =  (Integer input) -> { return input*3; }
```

```
<> IntFunction =  (Integer input) -> input*3;

<> IntFunction =  (input) -> input*3;
-------------------------------
solve left part
interface MyIntegerFunction{
	Integer apply(Integer input);
}

MyIntegerFunction IntFunction =  (input) -> input*3;


------------------
apply generics

interface MyIntegerFunction<T, R>{
	R apply(T input);
}

MyIntegerFunction<Integer, Integer> IntFunction =  (input) -> input*3;


------------------
apply @FunctionalInterface

@FunctionalInterface
interface MyIntegerFunction<T, R>{
	R apply(T input);
}

MyIntegerFunction<Integer, Integer> IntFunction =  (input) -> input*3;

---------------
remove MyIntegerFunction, beacuae java.util.fuction are present

IntFunction<Integer, Integer> IntFunction =  (input) -> input*3;

------------
```

More Reference:
[Lambda Expressions and Functional Interfaces: Tips and Best Practices | Baeldung]
[Functional Interfaces in Java - GeeksforGeeks]