Functional Programming in Java 8

# **Functional Interfaces**

All functional interfaces are recommended to have an informative *@FunctionalInterface* annotation. This not only clearly communicates the purpose of this interface, but also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

**Any interface with a SAM(Single Abstract Method) is a functional interface**, and its implementation may be treated as lambda expressions.

Note that Java 8’s *default* methods are not *abstract* and do not count: a functional interface may still have multiple *default* methods.

# **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](https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html" \o "interface in java.io), are defined in the packages where they are used.

The interfaces in this package are annotated with [FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html" \o "annotation in java.lang). 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](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) (unary function from T to R), [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) (unary function from T to void), [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) (unary function from T to boolean), and [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) (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](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) (binary function from T and U to R).
* There are additional derived function shapes which extend the basic function shapes, including [UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html" \o "interface in java.util.function) (extends Function) and [BinaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html" \o "interface in java.util.function) (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](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html" \o "interface in java.util.function). Otherwise, type arguments are specialized left-to-right, as in [DoubleConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html" \o "interface in java.util.function) or [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html" \o "interface in java.util.function). (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](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html" \o "interface in java.util.function).) 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](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html" \o "interface in java.util.function)).

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [**BiConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)<T,U> | Represents an operation that accepts two input arguments and returns no result. |
| [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<T,U,R> | Represents a function that accepts two arguments and produces a result. |
| [**BinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html)<T> | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| [**BiPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)<T,U> | Represents a predicate (boolean-valued function) of two arguments. |
| [**BooleanSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BooleanSupplier.html) | Represents a supplier of boolean-valued results. |
| [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<T> | Represents an operation that accepts a single input argument and returns no result. |
| [**DoubleBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleBinaryOperator.html) | Represents an operation upon two double-valued operands and producing a double-valued result. |
| [**DoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) | Represents an operation that accepts a single double-valued argument and returns no result. |
| [**DoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleFunction.html)<R> | Represents a function that accepts a double-valued argument and produces a result. |
| [**DoublePredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoublePredicate.html) | Represents a predicate (boolean-valued function) of one double-valued argument. |
| [**DoubleSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleSupplier.html) | Represents a supplier of double-valued results. |
| [**DoubleToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToIntFunction.html) | Represents a function that accepts a double-valued argument and produces an int-valued result. |
| [**DoubleToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToLongFunction.html) | Represents a function that accepts a double-valued argument and produces a long-valued result. |
| [**DoubleUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleUnaryOperator.html) | Represents an operation on a single double-valued operand that produces a double-valued result. |
| [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,R> | Represents a function that accepts one argument and produces a result. |
| [**IntBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntBinaryOperator.html) | Represents an operation upon two int-valued operands and producing an int-valued result. |
| [**IntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntConsumer.html) | Represents an operation that accepts a single int-valued argument and returns no result. |
| [**IntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntFunction.html)<R> | Represents a function that accepts an int-valued argument and produces a result. |
| [**IntPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntPredicate.html) | Represents a predicate (boolean-valued function) of one int-valued argument. |
| [**IntSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntSupplier.html) | Represents a supplier of int-valued results. |
| [**IntToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToDoubleFunction.html) | Represents a function that accepts an int-valued argument and produces a double-valued result. |
| [**IntToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToLongFunction.html) | Represents a function that accepts an int-valued argument and produces a long-valued result. |
| [**IntUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntUnaryOperator.html) | Represents an operation on a single int-valued operand that produces an int-valued result. |
| [**LongBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongBinaryOperator.html) | Represents an operation upon two long-valued operands and producing a long-valued result. |
| [**LongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongConsumer.html) | Represents an operation that accepts a single long-valued argument and returns no result. |
| [**LongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongFunction.html)<R> | Represents a function that accepts a long-valued argument and produces a result. |
| [**LongPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongPredicate.html) | Represents a predicate (boolean-valued function) of one long-valued argument. |
| [**LongSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongSupplier.html) | Represents a supplier of long-valued results. |
| [**LongToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToDoubleFunction.html) | Represents a function that accepts a long-valued argument and produces a double-valued result. |
| [**LongToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToIntFunction.html) | Represents a function that accepts a long-valued argument and produces an int-valued result. |
| [**LongUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongUnaryOperator.html) | Represents an operation on a single long-valued operand that produces a long-valued result. |
| [**ObjDoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjDoubleConsumer.html)<T> | Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| [**ObjIntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html)<T> | Represents an operation that accepts an object-valued and a int-valued argument, and returns no result. |
| [**ObjLongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjLongConsumer.html)<T> | Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | Represents a predicate (boolean-valued function) of one argument. |
| [**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<T> | Represents a supplier of results. |
| [**ToDoubleBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a double-valued result. |
| [**ToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleFunction.html)<T> | Represents a function that produces a double-valued result. |
| [**ToIntBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces an int-valued result. |
| [**ToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html)<T> | Represents a function that produces an int-valued result. |
| [**ToLongBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a long-valued result. |
| [**ToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongFunction.html)<T> | Represents a function that produces a long-valued result. |
| [**UnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html)<T> | Represents an operation on a single operand that produces a result of the same type as its operand. |

# **Interface** [**BiConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)**<T,U>**

java.util.function.BiConsumer is a java 8 functional interface. BiConsumer does not return value. It accepts two parameter as an argument. BiConsumer functional method is *accept(Object, Object)*.

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.text.MessageFormat;  
**import** java.util.HashMap;  
**import** java.util.Map;  
**import** java.util.function.BiConsumer;  
  
**public class** BiConsumerDemo {  
 **public static void** showMap(Map<Integer, String> map) {  
 BiConsumer<Integer, String> biConsumer =  
 (k, v) -> {  
 System.***out***.println(MessageFormat.*format*(**"Key : {0} ----- Value : {1}"**, k, v));  
 };  
 map.forEach(biConsumer);  
 }  
  
 **public static void** main(String[] args) {  
 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);  
  
 *showMap*(map);  
 }  
}

**Another simple example on BiConsumer**

**import** java.util.function.BiConsumer;

**public** **class** Main {

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

BiConsumer<String, String> biConsumer = (x, y) -> {

System.out.println(x);

System.out.println(y);

};

biConsumer.accept(**"java2s.com"**, **" tutorials"**);

}

}

**Using andThen in BiConsumer**

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.function.BiConsumer;  
  
**public class** BiConsumerExample2 {  
 **public static void** main(String[] args) {  
 BiConsumer<Integer, Integer> addition =  
 (a, b) -> {  
 System.***out***.println(a + b);  
 };  
  
 BiConsumer<Integer, Integer> subtraction =  
 (a, b) -> {  
 System.***out***.println(a - b);  
 };  
 *// Using andThen()* addition.andThen(subtraction).accept(10, 6);  
 }  
}

**OUTPUT**

**16**

**4**

# **interface Consumer<T>**

Represents an operation that accepts a single input argument and returns no result.

public class Main {

public static void main(String[] args) {

Consumer<String> c = (x) -> System.out.println(x.toLowerCase());

c.accept("Java2s.com");

}

}

OUTPUT

Java2s.com

**Another example is given below**

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.Arrays;  
**import** java.util.List;  
**import** java.util.function.Consumer;  
  
**public class** ConsumerFunctionExample {  
 **public static void** main(String args[]) {  
 Consumer<Integer> consumer = i -> System.***out***.print(**" "** + i);  
 List<Integer> integerList =  
 Arrays.*asList*(  
 **new** Integer(1),  
 **new** Integer(10),  
 **new** Integer(200),  
 **new** Integer(101),  
 **new** Integer(-10),  
 **new** Integer(0));  
 *printList*(integerList, consumer);  
 }  
  
 **public static void** printList(List<Integer> listOfIntegers, Consumer<Integer> consumer) {  
 **for** (Integer integer : listOfIntegers) {  
 consumer.accept(integer);  
 }  
 }  
}

**OUTPUT**

1 10 200 101 -10 0

Consumer specializations are given below.

* IntConsumer having one abstract method ' accept(int)' and one default method ' default IntConsumer andThen(IntConsumer after)'
* DoubleConsumer having one abstract method ' accept(double)' and one default method ' default DoubleConsumer andThen(DoubleConsumer after)'
* LongConsumer having one abstract method ' accept(long)' and one default method ' default LongConsumer andThen(LongConsumer after)'

# **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.

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.function.BiFunction;  
**public class** BiFunctionDemo {  
 **public static void** main(String[] args) {  
 BiFunction<Integer, Integer, String> biFunction = (num1, num2) -> **"Result:"** + (num1 + num2);  
 System.***out***.println(biFunction.apply(20, 25));  
 }  
}

**Another example is given below**

**import** java.util.function.BiFunction;

**public** **class** Main {

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

BiFunction<String, String,String> bi = (x, y) -> {

**return** x + y;

};

System.out.println(bi.apply(**"java2s.com"**, **" tutorial"**));

}

}

# Interface UnaryOperator<T>

* **Type Parameters:**

T - the type of the operand and result of the operator

public interface **UnaryOperator<T>**

extends [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,T>

Represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

**Simple example**

**import** java.util.function.UnaryOperator;

**public** **class** Main {

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

UnaryOperator<String> i = (x)-> x.toUpperCase();

System.out.println(i.apply(**"java2s.com"**));

}

}

**The following example shows how to use identity.**

**import** java.util.function.UnaryOperator;

**public** **class** Main {

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

UnaryOperator<String> i = (x)-> x.toUpperCase();

System.out.println(i.apply(**"java2s.com"**)); //JAVA2S.COM

}

}

# **Interface BinaryOperator<T>**

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **BinaryOperator<T>**

extends [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<T,T,T>

Represents an operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html" \o "interface in java.util.function) for the case where the operands and the result are all of the same type. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [BiFunction.apply(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html" \l "apply-T-U-).

**import** java.util.function.BinaryOperator;

**public** **class** Main {

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

BinaryOperator<**Integer**> adder = (n1, n2) -> n1 + n2;

System.out.println(adder.apply(3, 4)); //Prints 7

}

}

**Another example**

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.Comparator;  
**import** java.util.function.BinaryOperator;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 BinaryOperator<Integer> minBO = BinaryOperator.*minBy*(Comparator.*naturalOrder*());  
 System.***out***.println(minBO.apply(2, 3)); *// Prints 2* BinaryOperator<Integer> minReverse = BinaryOperator.*minBy*(Comparator.*reverseOrder*());  
 System.***out***.println(minReverse.apply(2, 3)); *// Prints 3* }  
}

**Simple example**  
**import** java.util.Comparator;  
**import** java.util.function.BinaryOperator;  
**public class** BinaryOperatorExample3 {  
 **public static void** main(String[] args) {  
 Comparator<Integer> comparator = (a, b) -> (a.compareTo(b));  
 *// Using maxBy()* BinaryOperator<Integer> opMax = BinaryOperator.*maxBy*(comparator);  
 System.***out***.println(**"Max: "** + opMax.apply(5, 6));*//Prints 6* System.***out***.println(**"Max: "** + opMax.apply(9, 6));*//Prints 9  
  
 // Using minBy()* BinaryOperator<Integer> opMin = BinaryOperator.*minBy*(comparator);  
 System.***out***.println(**"Min: "** + opMin.apply(5, 6));*//Prints 5* System.***out***.println(**"Min: "** + opMin.apply(9, 6));*//Prints 6* }  
}

A real time example

**package** com.ddlab.rnd.core.fns.type1;  
**public class** Student {  
 **private** String **name**;  
 **private** Integer **age**;  
 **private** String **className**;  
  
 **public** Student(String name, Integer age, String className) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.**className** = className;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public** String getClassName() {  
 **return className**;  
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.Arrays;  
**import** java.util.Comparator;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.Optional;  
**import** java.util.function.BinaryOperator;  
**import** java.util.stream.Collectors;  
  
**public class** MaxByMinBy {  
 **public static void** main(String[] args) {  
 Student s1 = **new** Student(**"Shyam"**, 22, **"A"**);  
 Student s2 = **new** Student(**"Ram"**, 23, **"A"**);  
 Student s3 = **new** Student(**"Mohan"**, 22, **"B"**);  
 Student s4 = **new** Student(**"Ramesh"**, 21, **"B"**);  
 List<Student> list = Arrays.*asList*(s1, s2, s3, s4);  
 Comparator<Student> ageComparator = Comparator.*comparing*(Student::getAge);  
 *// Using BinaryOperator.maxBy* System.***out***.println(**"---BinaryOperator.maxBy---"**);  
 Map<String, Optional<Student>> eldestByClass =  
 list.stream()  
 .collect(  
 Collectors.*groupingBy*(  
 Student::getClassName,  
 Collectors.*reducing*(BinaryOperator.*maxBy*(ageComparator))));  
 eldestByClass.forEach(  
 (k, v) ->  
 System.***out***.println(  
 **"Class:"** + k  
 + **" Age:"** + ((Optional<Student>) v).get().getAge()  
 + **" Name:"** + ((Optional<Student>) v).get().getName()));  
  
 *// Using BinaryOperator.minBy* System.***out***.println(**"---BinaryOperator.minBy---"**);  
 Map<String, Optional<Student>> youngestByClass =  
 list.stream()  
 .collect(  
 Collectors.*groupingBy*(  
 Student::getClassName,  
 Collectors.*reducing*(BinaryOperator.*minBy*(ageComparator))));  
 youngestByClass.forEach(  
 (k, v) ->  
 System.***out***.println(  
 **"Class:"** + k  
 + **" Age:"** + ((Optional<Student>) v).get().getAge()  
 + **" Name:"** + ((Optional<Student>) v).get().getName()));  
 }  
}

OUTPUT

---BinaryOperator.maxBy---

Class:A Age:23 Name:Ram

Class:B Age:22 Name:Mohan

---BinaryOperator.minBy---

Class:A Age:22 Name:Shyam

Class:B Age:21 Name:Ramesh

# **Interface BiPredicate<T,U>**

* **Type Parameters:**

T - the type of the first argument to the predicate

U - the type of the second argument the predicate

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference. Represents a predicate (boolean-valued function) of two arguments. This is the two-arity specialization of [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html). This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html#test-T-U-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| default **[BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html" \o "interface in java.util.function)**<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html),[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)> | [**and**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html#and-java.util.function.BiPredicate-)(**[BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html" \o "interface in java.util.function)**<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html),? super [**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| default **[BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html" \o "interface in java.util.function)**<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html),[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)> | [**negate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html#negate--)()  Returns a predicate that represents the logical negation of this predicate. |
| default **[BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html" \o "interface in java.util.function)**<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html),[**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)> | [**or**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html#or-java.util.function.BiPredicate-)(**[BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html" \o "interface in java.util.function)**<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html),? super [**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.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/BiPredicate.html#test-T-U-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html) t, [**U**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html) u)  Evaluates this predicate on the given arguments. |

Example

import java.util.function.BiPredicate;

public class BiPredicateDemo {

public static void main(String[] args){

BiPredicate<Integer, String> condition = (i,s)-> i>20 && s.startsWith("R");

System.out.println(condition.test(10,"Ram"));//False

System.out.println(condition.test(30,"Shyam"));//False

System.out.println(condition.test(30,"Ram"));//True

}

}

**Basic example**

**import** java.util.function.BiPredicate;

**public** **class** Main {

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

BiPredicate<**Integer**, **Integer**> bi = (x, y) -> x > y;

System.out.println(bi.test(2, 3)); //False

}

}

**Example**

**import** java.util.function.BiPredicate;

**public** **class** Main {

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

**boolean** result = **compare((a, b) -> a / 2 == b, 10, 5);**

System.out.println(**"Compare result: "** + result);

}

**public** **static** **boolean** compare(BiPredicate<**Integer**, **Integer**> bi, **Integer** i1, **Integer** i2) {

**return** bi.test(i1, i2);

}

}

**Example**

**import** java.util.function.BiPredicate;

**public** **class** Main {

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

BiPredicate<**Integer**, **Integer**> bi = (x, y) -> x > y;

System.out.println(bi.test(2, 3)); //False

System.out.println(bi.negate().test(2, 3)); //True

}

}

**Example**

**import** java.util.function.BiPredicate;

**public** **class** Main {

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

BiPredicate<**Integer**, **Integer**> bi = (x, y) -> x > y;

BiPredicate<**Integer**, **Integer**> eq = (x, y) -> x -2 > y;

System.out.println(bi.test(2, 3)); // false

System.out.println(bi.or(eq).test(2, 3)); // false

System.out.println(bi.or(eq).test(8, 3)); // true

}

}

**Example**

**import** java.util.function.BiPredicate;

**public** **class** Main {

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

BiPredicate<**Integer**, **Integer**> bi = (x, y) -> x > y;

BiPredicate<**Integer**, **Integer**> eq = (x, y) -> x -2 > y;

System.out.println(bi.test(2, 3)); //false

System.out.println(bi.and(eq).test(2, 3)); //false

System.out.println(bi.and(eq).test(8, 3)); //true

}

}

# **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.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Predicate<T>**

Represents a predicate (boolean-valued function) of one argument.

This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| 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. |

**Example**

**import** java.util.function.Predicate;

**public** **class** Main {

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

Predicate<String> i = (s)-> s.length() > 5;

System.out.println(**i.test**(**"java2s.com "**)); //True

}

}

**Example**

**import** java.util.function.Predicate;

**public** **class** Main {

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

Predicate<String> i = (s)-> s.length() > 5;

Predicate<String> j = (s)-> s.length() < 3;

System.out.println(**i.and(j).test**(**"java2s.com "**)); //false

}

}

**Example**

**import** java.util.function.Predicate;

**public** **class** Main {

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

Predicate<String> i = (s)-> s.length() > 5;

System.out.println(**i.negate().test**(**"java2s.com "**));//false

}

}

**Example**

**import** java.util.function.Predicate;

**public** **class** Main {

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

Predicate<String> i = (s)-> s.length() > 5;

Predicate<String> j = (s)-> s.length() < 3;

System.out.println**(i.or(j).test**(**"java2s.com "**));//true

}

}

**Examples**

**import** java.util.function.Predicate;

**public** **class** Main {

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

Predicate<String> i = **Predicate.isEqual**(**"asdf"**);

System.out.println(i.test(**"java2s.com "**));//false

}

}

## **Practical example on Predicate**

**package** com.ddlab.rnd.core.fns.type1;  
**public class** Employee {  
  
 **private** Integer **id**;  
 **private** Integer **age**;  
 **private** String **gender**;  
 **private** String **firstName**;  
 **private** String **lastName**;  
  
 **public** Employee(Integer id, Integer age, String gender, String fName, String lName) {  
 **this**.**id** = id;  
 **this**.**age** = age;  
 **this**.**gender** = gender;  
 **this**.**firstName** = fName;  
 **this**.**lastName** = lName;  
 }  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public** String getGender() {  
 **return gender**;  
 }  
  
 **public** String getFirstName() {  
 **return firstName**;  
 }  
  
 **public** String getLastName() {  
 **return lastName**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return this**.**id**.toString()  
 + **" - "** + **this**.**age**.toString();   
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.List;  
**import** java.util.function.Predicate;  
**import** java.util.stream.Collectors;  
  
**public class** EmployeePredicates {  
 **public static** Predicate<Employee> isAdultMale() {  
 **return** p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase(**"M"**);  
 }  
  
 **public static** Predicate<Employee> isAdultFemale() {  
 **return** p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase(**"F"**);  
 }  
  
 **public static** Predicate<Employee> isAgeMoreThan(Integer age) {  
 **return** p -> p.getAge() > age;  
 }  
  
 **public static** List<Employee> filterEmployees(  
 List<Employee> employees, Predicate<Employee> predicate) {  
 **return** employees.stream().filter(predicate).collect(Collectors.<Employee>*toList*());  
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
**import** java.util.stream.Collectors;  
**import static** com.ddlab.rnd.core.fns.type1.EmployeePredicates.*isAdultFemale*;  
**import static** com.ddlab.rnd.core.fns.type1.EmployeePredicates.*isAdultMale*;  
**import static** com.ddlab.rnd.core.fns.type1.EmployeePredicates.*isAgeMoreThan*;  
  
**public class** TestEmployeePredicates {  
  
 **public static void** main(String[] args) {  
 Employee e1 = **new** Employee(1, 23, **"M"**, **"Rick"**, **"Beethovan"**);  
 Employee e2 = **new** Employee(2, 13, **"F"**, **"Martina"**, **"Hengis"**);  
 Employee e3 = **new** Employee(3, 43, **"M"**, **"Ricky"**, **"Martin"**);  
 Employee e4 = **new** Employee(4, 26, **"M"**, **"Jon"**, **"Lowman"**);  
 Employee e5 = **new** Employee(5, 19, **"F"**, **"Cristine"**, **"Maria"**);  
 Employee e6 = **new** Employee(6, 15, **"M"**, **"David"**, **"Feezor"**);  
 Employee e7 = **new** Employee(7, 68, **"F"**, **"Melissa"**, **"Roy"**);  
 Employee e8 = **new** Employee(8, 79, **"M"**, **"Alex"**, **"Gussin"**);  
 Employee e9 = **new** Employee(9, 15, **"F"**, **"Neetu"**, **"Singh"**);  
 Employee e10 = **new** Employee(10, 45, **"M"**, **"Naveen"**, **"Jain"**);  
  
 List<Employee> employees = **new** ArrayList<Employee>();  
 employees.addAll(Arrays.*asList*(**new** Employee[] {e1, e2, e3, e4, e5, e6, e7, e8, e9, e10}));  
  
 List<Employee> empList = employees.stream().filter(*isAdultMale*()).collect(Collectors.*toList*());  
 System.***out***.println(empList);  
 System.***out***.println(employees.stream().filter(*isAdultFemale*()).collect(Collectors.*toList*()));  
 System.***out***.println(employees.stream().filter(*isAgeMoreThan*(35)).collect(Collectors.*toList*()));  
 }  
}

# **Interface Supplier<T>**

* **Type Parameters:**

T - the type of results supplied by this supplier

**Supplier<T>** is an in-built [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) introduced in Java 8 in the **java.util.function** package. Supplier can be used in all contexts where there is no input but an output is expected.

**import** java.util.function.Supplier;

**public** **class** Main {

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

Supplier<String> i = ()-> **"java2s.com"**;

System.out.println(i.get()); //java2s.com

}

}

**Practical Example**

**package** com.ddlab.rnd.core.fns.type1;  
**public class** Item {  
 **private** String **name**;  
  
 **public** Item() {}  
  
 **public** Item(String name) {  
 **this**.**name** = name;  
 }  
  
 **public static** String getStaticVal() {  
 **return "Static Val"**;  
 }  
  
 **public** String getMsg() {  
 **return "Hello World!"**;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.function.Supplier;  
**public class** SupplierReturnObject {  
 **public static void** main(String[] args) {  
 Supplier<Item> supplier = Item::**new**;  
 Item item = supplier.get();  
 System.***out***.println(item.getMsg());  
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.function.Supplier;  
**public class** SupplierFetchMethod {  
 **public static void** main(String[] args) {  
 Supplier<String> supplier = Item::*getStaticVal*;  
 String val = supplier.get();  
 System.***out***.println(**"Calling Method:"** + val);  
 }  
}

**package** com.ddlab.rnd.core.fns.type1;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.stream.Stream;  
  
**public class** SupplierWithStream {  
 **public static void** main(String[] args) {  
 List<Item> list = **new** ArrayList<>();  
 list.add(**new** Item(**"AA"**));  
 list.add(**new** Item(**"BB"**));  
 list.add(**new** Item(**"CC"**));  
 Stream<String> names = list.stream().map(Item::getName);  
 names.forEach(n -> System.***out***.println(n));  
 }  
}

There are primitive specializations of the Supplier interface:

* IntSupplier having one abstract method getAsInt()
* LongSupplier having one abstract method getAsLong()
* DoubleSupplier having one abstract method getAsDouble()
* BooleanSupplier having one abstract method getAsBoolean()

# **Functional programming**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **functional programming** is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm)—a style of building the structure and elements of [computer programs](https://en.wikipedia.org/wiki/Computer_program)—that treats [computation](https://en.wikipedia.org/wiki/Computation) as the evaluation of [mathematical functions](https://en.wikipedia.org/wiki/Function_(mathematics)) and avoids changing-[state](https://en.wikipedia.org/wiki/Program_state) and [mutable](https://en.wikipedia.org/wiki/Immutable_object) data. It is a [declarative programming](https://en.wikipedia.org/wiki/Declarative_programming) paradigm, which means programming is done with [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science))[[1]](https://en.wikipedia.org/wiki/Functional_programming#cite_note-expression_style-1) or declarations[[2]](https://en.wikipedia.org/wiki/Functional_programming#cite_note-declaration_style-2) instead of [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)). In functional code, the output value of a function depends only on the [arguments](https://en.wikipedia.org/wiki/Function_argument) that are passed to the function, so calling a function *f*twice with the same value for an argument *x* produces the same result *f(x)* each time; this is in contrast to [procedures](https://en.wikipedia.org/wiki/Subroutine) depending on a [local](https://en.wikipedia.org/wiki/Local_state) or [global state](https://en.wikipedia.org/wiki/Global_variable), which may produce different results at different times when called with the same arguments but a different program state. Eliminating [side effects](https://en.wikipedia.org/wiki/Side_effect_(computer_science)), i.e., changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **imperative programming** is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) that uses [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)) that change a program's [state](https://en.wikipedia.org/wiki/State_(computer_science)).

The term is often used in contrast to [declarative programming](https://en.wikipedia.org/wiki/Declarative_programming), which focuses on *what* the program should accomplish without specifying *how* the program should achieve the result.

**Functional programming** (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state,** **mutable data,**and **side-effects**. Functional programming is **declarative** rather than **imperative**, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

Functional programming is a **programming paradigm**, meaning that it is a way of thinking about software construction based on some fundamental, defining principles (listed above). Other examples of programming paradigms include object oriented programming and procedural programming.

Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code — but if you’re unfamiliar with it and the common patterns associated with it, functional code can also seem a lot more dense, and the related literature can be impenetrable to newcomers.

A **pure function** is a function which:

* Given the same inputs, always returns the same output, and
* Has no side-effects

Pure functions have lots of properties that are important in functional programming, including **referential transparency** (you can replace a function call with its resulting value without changing the meaning of the program). Read [“What is a Pure Function?”](https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976) for more details.

**Function composition**is the process of combining two or more functions in order to produce a new function or perform some computation. For example, the composition f . g (the dot means “composed with”) is equivalent to f(g(x)) in JavaScript. Understanding function composition is an important step towards understanding how software is constructed using the functional programming.