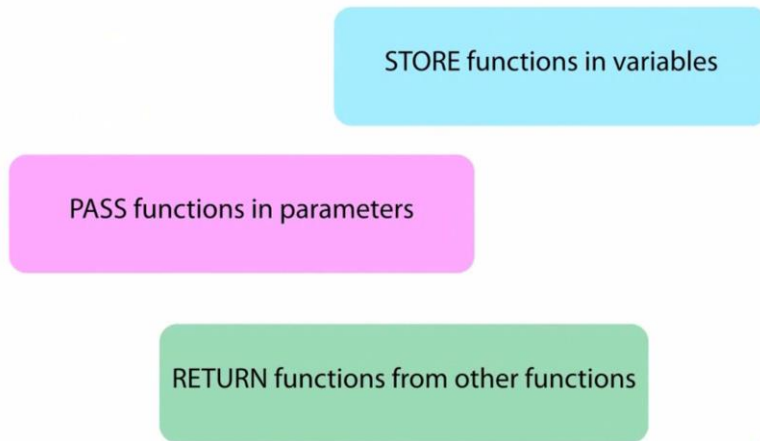


Java Functional Programming

Functions as Values



Imperative Programming

Programs are written in a series of instructions that tell the computer what to do (and how).

Declarative Programming

A programming paradigm that describes what the program should do, without specifying how it should be done.

Functional interfaces

The screenshot shows the Java SE 11 & JDK 11 documentation page for Functional interfaces. The page has a dark blue header with navigation links: OVERVIEW, MODULE, PACKAGE (highlighted), CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. The version "Java SE 11 & JDK 11" is displayed on the right. Below the header, there is a search bar with the text "SEARCH: Search" and a close button. The main content area shows the "Module java.base" and the "Package java.util.function". The text describes functional interfaces as providing target types for lambda expressions and method references, each having a single abstract method called the functional method.

Module java.base

Package java.util.function

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:

Predicate Interface

In Java, a predicate is a function that takes one or more arguments and returns a boolean value.

Predicate<T>

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

java.util.function

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
public interface **Predicate<T>**

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

This is a functional interface whose functional method is test(Object).

Since:
1.8

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type			Method and Description	
default Predicate<T>			and(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>			isEqual(Object targetRef)	Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).
default Predicate<T>			negate()	Returns a predicate that represents the logical negation of this predicate.
default Predicate<T>			or(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean			test(T t)	Evaluates this predicate on the given argument.

```
final Predicate<Person> personPredicate = person -> FEMALE.equals(person.gender);  
people.stream()  
    .filter(personPredicate) You, Moments ago • Uncommitted changes  
    .forEach(System.out::println);
```

Fuction Interface

@FunctionalInterface
public interface **Function<T,R>**

Represents a function that accepts one argument and produces a result.

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

Since:
1.8

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type			Method and Description	
default <V> Function<T,V>			andThen(Function<? super R,? 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			apply(T t)	Applies this function to the given argument.
default <V> Function<V,R>			compose(Function<? super V,? extends T> 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<T,T>			identity()	Returns a function that always returns its input argument.

Method Detail

apply

R apply(T t)
Applies this function to the given argument.

Parameters:
t - the function argument

Returns:
the function result

```

public static void main(String[] args) {
    int increment = incrementByOne( number: 0);
    System.out.println(increment);    You, Moments ago • Uncommitted changes

    final Integer incrementByOneLambda = incrementByOneFunction.apply( t: 1);
    System.out.println(incrementByOneLambda);
}

1 usage
static Function<Integer, Integer> incrementByOneFunction = number -> number + 1;

```

Chaining Lambdas (andThen())

The output of a lambda can be used as the input of another lambda.

```

2 usages
static Function<Integer, Integer> incrementByOneFunction = number -> number + 1;

2 usages
static Function<Integer, Integer> multiplyBy10Function = number -> number * 10;

```

```

Function<Integer, Integer> addByOneAndThenMultiplyBy10 = incrementByOneFunction.andThen(multiplyBy10Function);
int incrementAndMultiply = addByOneAndThenMultiplyBy10.apply( t: 4);
System.out.println(incrementAndMultiply);

```

Bifunction

@FunctionalInterface
public interface BiFunction<T,U,R>

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).

Since:

1.8

See Also:

Function

Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
default <V> BiFunction<T,U,V>		andThen(Function<? super R,? 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		apply(T t, U u)	
		Applies this function to the given arguments.	

```

static BiFunction<Integer, Integer, Integer> incrementByOneAndMultiplyBiFunction =
    (numberToIncrementByOne, numberToMultiplyBy)
    -> (numberToIncrementByOne + 1) * numberToMultiplyBy;

```

```

// BiFunction
int incrementAndMultiplyBiFunction = incrementByOneAndMultiplyBiFunction.apply( t: 1, u: 2);
System.out.println(incrementAndMultiplyBiFunction);

```