# Functional Interfaces

## Overview

In this lab you'll get a chance to use some of the functional interfaces defined in the java.util.function package.

## Modules for this lab (in the ModernJavaDev project)

* student.functionals
* solution.functionals

## Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Using the Function<T,R> functional interface
2. Using the Predicate<T> functional interface
3. Using the Supplier<T> and Consumer<T> functional interfaces
4. (If Time Permits) Using function composition

## Exercise 1: Using the Function<T,R> functional interface

In this exercise, you'll use the Function<T,R> functional interface to represent functions that take a value and return a result.

In the student.functionalsmodule, open Exercise1\_Functions.java. At the start of main(), we create a Calculator object. The Calculator class provides some simple mathematical functions that take a double parameter and return a double result. Take a look and see.

After the end of main(), implement a generic method named doOp(). The method should take two parameters:

* A data value of some type T.
* A Function<T,R> object representing the function to apply on that value.

Implement doOp() so that it applies the designated function on the supplied value, and displays the result on the console.

In main(), where indicated by the comment, add some code to call doOp() several times. In each call, pass in a double value representing an angle in degrees, plus a Function<Double,Double> representing the operation to apply (e.g. calc::sin). Run the program and verify you get the correct results.

*Additional Consideration #1*

The code you've just written uses Function<T,R> to represent the function to be performed. As it happens, all of the functions in the example have the same parameter type and return type (double). In this case, you can use UnaryOperator<T> rather than a Function<T,R>. With this in mind, define another generic method named doOpUsingUnaryOperator(). This method should be similar to doOp(), except that it takes a UnaryOperator<T> parameter rather than a Function<T,R>. Call doOpUsingUnaryOperator() several times from main().

*Additional Consideration #2*

Java defines type-specific specializations for many of the new functional interfaces. For example, there are several type-specific specializations of UnaryOperator<T>:

* DoubleUnaryOperator
* IntUnaryOperator
* LongUnaryOperator

All of the functions in the example use double, so you can make use of DoubleUnaryOperator. So… define a non-generic method named doOpUsingDoubleUnaryOperator(). This method should be similar to doOpUsingUnaryOperator(), except that it takes a DoubleUnaryOperator parameter rather than a UnaryOperator<T> (note that the method to apply the operation on a DoubleUnaryOperator is named applyAsDouble()). Call doOpUsingDoubleUnaryOperator() several times from main().

## Exercise 2: Using the Predicate<T> functional interface

In this exercise, you'll use the Predicate<T> functional interface to represent boolean tests on a value. You'll also learn how to combine predicates to perform NOT, AND, and OR tests.

In the student.functionalsmodule, open Exercise2\_Predicates. In main(), define three separate Predicate<Integer> objects to perform various tests on an integer (e.g. to test a person's age). Use lambda expressions to implement the tests:

* Is the integer 18 or above (i.e. is the person an adult)?
* Is the integer less than 65 (i.e. is the person not a pensioner yet)?
* Is the integer 100 or above (i.e. is the person a centurion)?

After the end of main(), implement a generic method named doTest(). The method should take two parameters:

* A data value of some type T.
* A Predicate<T> object representing the test to perform on that value.

Implement doTest() so that it performs the designated test on the supplied value, and displays the boolean result on the console.

In main(), where indicated by the comment, add some code to call doTest() several times. In each call, pass in an int value representing a person's age, plus one of the Predicate<Integer> objects you created earlier. Run the program and verify you get the correct results.

*Performing NOT tests*

The Predicate<T> interface has a default method named negate(), which allows you to perform a logical NOT test. The signature of negate() is as follows:

default Predicate<T> negate();

This is how you use negate() in your code:

boolean result = somePredicate.negate().test(someValue);

Extend your program so that it performs some logical NOT tests.

*Performing AND and OR tests*

The Predicate<T> interface also has default methods named and() and or(), which allow you to perform logical AND and OR tests. The methods have the following signatures:

default Predicate<T> and(Predicate<? super T> otherPredicate);

default Predicate<T> or(Predicate<? super T> otherPredicate);

This is how you use these methods in your code:

boolean result = predicate1.and(predicate2).test(someValue);

boolean result = predicate1.or(predicate2).test(someValue);

Extend your program so that it performs logical AND and OR tests.

## Exercise 3: Using the Supplier<T> and Consumer<T> functional interfaces

In this exercise, you'll use the Supplier<T> and Consumer<T> functional interfaces to supply and consume data values in an application, respectively.

Open Exercise3\_SupplierConsumer.java and note the following points:

* At the start of main(), we create a NumberSource object. The NumberSource class defines several functions that return (i.e. supply) an integer value in various ways. Take a quick look at this class now.
* After the main() method, we've implemented a method named displayOnConsole() that consumes a value (specifically, it displays the value on the console).

Where indicated by the comment at the end of the class, implement a generic method named processValue(). The method should take two parameters:

* A Supplier<T> object, which can be used to supply a value.
* A Consumer<T> object, which can be used to consume the value.

Implement processValue() so that it gets a value from the supplier object, and then passes the value to the consumer object to be consumed.

Inside main(), where indicated by the comment, add some code to call processValue() several times. In each call, pass in Supplier<T> object (i.e. a reference to one of the methods on the NumberSource object) and a Consumer<T> object (i.e. a reference to the displayOnConsole method).

## Exercise 4 (If Time Permits): Using function composition

Java 8 allows you to compose functional interfaces into a sequence of chained calls. For example, Function<T> has methods named compose() and andThen() for this purpose. We'll explain each of these methods in the next few paragraphs…

*Understanding compose()*

compose() allows you to specify another Function<T> that should be applied *before* the current one. The simplified signature of compose() is as follows:

default Function<V,R> compose(Function<V,R> before)

You can use compose() as follows in your code:

R result = func1.compose(func2).apply(someValue);

This will apply func2 first, then func1 afterwards. It's equivalent to the following separate calls:

R intermediateResult = func2.apply(someValue);

R result = func1.apply(intermediateResult);

*Understanding andThen()*

andThen() allows you to specify another Function<T> that should be applied *after* the current one. The simplified signature of andThen() is as follows:

default Function<V,R> andThen(Function<V,R> after)

You can use andThen() as follows in your code:

R result = func1.andThen(func2).apply(someValue);

This will apply func1 first, then func2 afterwards. It's equivalent to the following separate calls:

R intermediateResult = func1.apply(someValue);

R result = func2.apply(intermediateResult);

Your task is to make use of these mechanisms to chain multiple functions together. Open Exercise4\_ComposedFunctions and take a look at the starter code. We've created several Function<Double,Double> objects that reference some methods in a Calculator object. Add code in main() to chain these functions together in various ways, to see the effect of compose() and andThen(). Pass the resultant Function<T,T> into doOp() to evaluate the function chain and display the result on the console.