Updated through Java 21



# Core Java for the Impatient

Fourth Edition

Cay S. Horstmann

This page intentionally left blank

| Functional interface | Parameter types | Return type | Abstract method name | Description | Other methods |
|---|---|---|---|---|---|
| `BinaryOperator<T>` | `T, T` | `T` | `apply` | A binary operator on the type `T`. | `andThen`, `maxBy`, `minBy` |
| `Predicate<T>` | `T` | `boolean` | `test` | A boolean-valued function. | `and`, `or`, `negate`, `isEqual` |
| `BiPredicate<T, U>` | `T, U` | `boolean` | `test` | A boolean-valued function with two parameters. | `and`, `or`, `negate` |

For example, suppose you write a method to process files that match a certain criterion. Should you use the descriptive `java.io.FileFilter` class or a `Predicate<File>`? I strongly recommend that you use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.

**Note:** Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, `Predicate.isEqual(a)` is the same as `a::equals`, but it also works if `a` is `null`. There are default methods `and`, `or`, `negate` for combining predicates. For example,

```
Predicate.isEqual(a).or(Predicate.isEqual(b))
```

is the same as

```
x -> a.equals(x) || b.equals(x)
```

Table 3.3 lists the 34 available specializations for primitive types `int`, `long`, and `double`. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

**Table 3.3:** Functional Interfaces for Primitive Types
*p*, *q* is `int`, `long`, `double`; *P*, *Q* is `Int`, `Long`, `Double`

| Functional interface | Parameter types | Return type | Abstract method name |
|---|---|---|---|
| `BooleanSupplier` | none | `boolean` | `getAsBoolean` |

| Functional interface | Parameter types | Return type | Abstract method name |
|---|---|---|---|
| *P*Supplier | none | *p* | getAs*P* |
| *P*Consumer | *p* | void | accept |
| Obj*P*Consumer<T> | T, *p* | void | accept |
| *P*Function<T> | *p* | T | apply |
| *P*To*Q*Function | *p* | *q* | applyAs*Q* |
| To*P*Function<T> | T | *p* | applyAs*P* |
| To*P*BiFunction<T, U> | T, U | *p* | applyAs*P* |
| *P*UnaryOperator | *p* | *p* | applyAs*P* |
| *P*BinaryOperator | *p*, *p* | *p* | applyAs*P* |
| *P*Predicate | *p* | boolean | test |

## 3.6.3. Implementing Your Own Functional Interfaces

Ever so often, you will be in a situation where none of the standard functional interfaces work for you. Then you need to roll your own.

Suppose you want to fill an image with color patterns, where the user supplies a function yielding the color for each pixel. There is no standard type for a mapping `(int, int) -> Color`. You could use `BiFunction<Integer, Integer, Color>`, but that involves autoboxing.

In this case, it makes sense to define a new interface

```
@FunctionalInterface
public interface PixelFunction {
    Color apply(int x, int y);
}
```

**Note:** It is a good idea to tag functional interfaces with the `@FunctionalInterface` annotation. This has two advantages. First, the compiler checks that the annotated entity is an interface with a single abstract method. Second, the javadoc page includes a statement that your interface is a functional interface.

Now you are ready to implement a method:

```
BufferedImage createImage(int width, int height, PixelFunction f) {
    var image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

To call it, supply a lambda expression that yields a color value for two integers:

```
BufferedImage frenchFlag = createImage(150, 100,
    (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

# 3.7. Lambda Expressions and Variable Scope

In the following sections, you will learn how variables work inside lambda expressions. This information is somewhat technical but essential for working with lambda expressions.

## 3.7.1. Scope of a Lambda Expression

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
int first = 0;
Comparator<String> comp = (first, second) -> first.length() - second.length();
    // Error: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, therefore you can't introduce such variables in a lambda expression either.

As another consequence of the "same scope" rule, the `this` keyword in a lambda expression denotes the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `Runnable` instance. There is nothing special about the use of `this` in a lambda expression.

The scope of the lambda expression is nested inside the `doWork` method, and `this` has the same meaning anywhere in that method.

## 3.7.2. Accessing Variables from the Enclosing Scope

Often, you want to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
    Thread.ofPlatform().start(r);
}
```

Note that the lambda expression accesses the parameter variables defined in the enclosing scope, not in the lambda expression itself.

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello 1000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How do the `text` and `count` variables stay around when the lambda expression is ready to execute?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables—that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has two free variables, `text` and `count`. The data structure representing the lambda expression must store the values for these variables—in our case, `"Hello"` and `1000`. We say that these values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)

**Note:** The technical term for a block of code together with the values of free variables is a *closure*. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. To ensure that the captured value is well defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. This is sometimes described by saying that lambda expressions capture values, not variables. For example, the following is a compile-time error:

```
for (int i = 0; i < n; i++) {
    Thread.ofPlatform().start(() -> System.out.println(i));
        // Error—cannot capture i
}
```

The lambda expression tries to capture `i`, but this is not legal because `i` changes. There is no single value to capture. The rule is that a lambda expression can only access local variables from an enclosing scope that are *effectively final*. An effectively final variable is never modified—it either is or could be declared as `final`.

**Note:** The same rule applies to variables captured by local classes (see Section 3.9). In the past, the rule was more draconian and required captured variables to actually be declared `final`. This is no longer the case.

**Note:** The variable of an enhanced `for` loop is effectively final since its scope is a single iteration. The following is perfectly legal:

```
for (String arg : args) {
    Thread.ofPlatform().start(() -> System.out.println(arg));
        // OK to capture arg
}
```

A new variable `arg` is created in each iteration and assigned the next value from the `args` array. In contrast, the scope of the variable `i` in the preceding example was the entire loop.

As a consequence of the "effectively final" rule, a lambda expression cannot mutate any captured variables. For example,

```
public static void repeatMessage(String text, int count, int threads) {
    Runnable r = () -> {
        while (count > 0) {
            count--; // Error: Can't mutate captured variable
            System.out.println(text);
        }
    };
    for (int i = 0; i < threads; i++) Thread.ofPlatform().start(r);
}
```

This is actually a good thing. As you will see in Chapter 10, if two threads update `count` at the same time, its value is undefined.

**Note:** Don't count on the compiler to catch all concurrent access errors. The prohibition against mutation only holds for local variables. If `count` is an instance variable or static variable of an enclosing class, then no error is reported even though the result is just as undefined.

**Caution:** One can circumvent the check for inappropriate mutations by using an array of length 1:

```
int[] counter = new int[1];
button.addActionListener(event -> counter[0]++);
```

The `counter` variable is effectively final—it is never changed since it always refers to the same array, so you can access it in the lambda expression.

Of course, code like this is not threadsafe. Except possibly for a callback in a single-threaded UI, this is a terrible idea. You will see how to implement a threadsafe shared counter in Chapter 10.

## 3.8. Higher-Order Functions

In a functional programming language, functions are first-class citizens. Just like you can pass numbers to methods and have methods that produce numbers, you can have arguments and return values that are functions. Functions that process or return functions are called *higher-order functions*. This sounds abstract, but it is very useful in practice. Java is not quite a functional language because it uses functional interfaces, but the principle is the same. In the following sections, we will look at some examples and examine the higher-order functions in the `Comparator` interface.

### 3.8.1. Methods That Return Functions

Suppose sometimes we want to sort an array of strings in ascending order and other times in descending order. We can make a method that produces the correct comparator:

```
public static Comparator<String> compareInDirecton(int direction) {
    return (x, y) -> direction * x.compareTo(y);
}
```

The call `compareInDirection(1)` yields an ascending comparator, and the call `compareInDirection(-1)` a descending comparator.