Java SE 8 for the Really Impatient

Cay S. Horstmann

Chapter 1

Java was designed in the 1990s as an object-oriented programming language, when object-oriented programming was the principal paradigm for software development. Long before there was object-oriented programming, there were functional programming languages such as Lisp and Scheme, but their benefits were not much appreciated outside academic circles. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or "reactive") programming. That doesn't mean that objects are bad. Instead, the winning strategy is to blend object-oriented and functional programming. This makes sense even if you are not interested in concurrency. For example, collection libraries can be given powerful APIs if the language has a convenient syntax for function expressions.

The principal enhancement in Java 8 is the addition of functional programming constructs to its object-oriented roots. In this chapter, you will learn the basic syntax. The next chapter shows you how to put that syntax to use with Java collections, and in Chapter 3 you will learn how to build your own functional libraries.

The key points of this chapter are:

- A lambda expression is a block of code with parameters.
- Use a lambda expression whenever you want a block of code executed at a later point in time.
- Lambda expressions can be converted to functional interfaces.

- Lambda expressions can access effectively final variables from the enclosing scope.
- Method and constructor references refer to methods or constructors without invoking them.
- You can now add default and static methods to interfaces that provide concrete implementations.
- You must resolve any conflicts between default methods from multiple interfaces.

1.1 Why Lambdas?

A "lambda expression" is a block of code that you can pass around so it can be executed later, once or multiple times. Before getting into the syntax (or even the curious name), let's step back and see where you have used similar code blocks in Java all along.

When you want to do work in a separate thread, you put the work into the `run` method of a `Runnable`, like this:

```
class Worker implements Runnable {
   public void run() {
      for (int i = 0; i < 1000; i++)
         doWork();
   }
   ...
}
```

Then, when you want to execute this code, you construct an instance of the `Worker` class. You can then submit the instance to a thread pool, or, to keep it simple, start a new thread:

```
Worker w = new Worker();
new Thread(w).start();
```

The key point is that the `run` method contains code that you want to execute in a separate thread.

Or consider sorting with a custom comparator. If you want to sort strings by length instead of the default dictionary order, you can pass a `Comparator` object to the `sort` method:

```
class LengthComparator implements Comparator<String> {
   public int compare(String first, String second) {
      return Integer.compare(first.length(), second.length());
   }
}
```

```
Arrays.sort(strings, new LengthComparator());
```

The `sort` method keeps calling the `compare` method, rearranging the elements if they are out of order, until the array is sorted. You give the `sort` method a snippet of code needed to compare elements, and that code is integrated into the rest of the sorting logic, which you'd probably not care to reimplement.

NOTE: The call `Integer.compare(x, y)` returns zero if `x` and `y` are equal, a negative number if `x < y`, and a positive number if `x > y`. This static method has been added to Java 7 (see Chapter 9). Note that you shouldn't compute `x - y` to compare `x` and `y` since that computation can overflow for large operands of opposite sign.

As another example for deferred execution, consider a button callback. You put the callback action into a method of a class implementing the listener interface, construct an instance, and register the instance with the button. That happens so often that many programmers use the "anonymous instance of anonymous class" syntax:

```
button.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent event) {
      System.out.println("Thanks for clicking!");
   }
});
```

What matters is the code inside the `handle` method. That code is executed whenever the button is clicked.

NOTE: Since Java 8 positions JavaFX as the successor to the Swing GUI toolkit, I use JavaFX in these examples. (See Chapter 4 for more information on JavaFX.) Of course, the details don't matter. In every user interface toolkit, be it Swing, JavaFX, or Android, you give a button some code that you want to run when the button is clicked.

In all three examples, you saw the same approach. A block of code was passed to someone—a thread pool, a `sort` method, or a button. The code was called at some later time.

Up to now, giving someone a block of code hasn't been easy in Java. You couldn't just pass code blocks around. Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.

In other languages, it is possible to work with blocks of code directly. The Java designers have resisted adding this feature for a long time. After all, a great strength of Java is its simplicity and consistency. A language can become an unmaintainable mess if it includes every feature that yields marginally more concise code. However, in those other languages it isn't just easier to spawn a thread or to register a button click handler; large swaths of their APIs are simpler, more consistent, and more powerful. In Java, one could have written similar APIs that take objects of classes implementing a particular function, but such APIs would be unpleasant to use.

For some time now, the question was not whether to augment Java for functional programming, but how to do it. It took several years of experimentation before a design emerged that is a good fit for Java. In the next section, you will see how you can work with blocks of code in Java 8.

1.2 The Syntax of Lambda Expressions

Consider again the sorting example from the preceding section. We pass code that checks whether one string is shorter than another. We compute

```
Integer.compare(first.length(), second.length())
```

What are `first` and `second`? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second)
   -> Integer.compare(first.length(), second.length())
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters. Had he known about the Java API, he would have written

```
λfirst.λsecond.Integer.compare(first.length(), second.length())
```

NOTE: Why the letter λ? Did Church run out of other letters of the alphabet? Actually, the venerable *Principia Mathematica* used the ^ accent to denote free variables, which inspired Church to use an uppercase lambda Λ for parameters. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

You have just seen one form of lambda expressions in Java: parameters, the -> arrow, and an expression. If the code carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit `return` statements. For example,

```
(String first, String second) -> {
   if (first.length() < second.length()) return -1;
   else if (first.length() > second.length()) return 1;
   else return 0;
}
```

If a lambda expression has no parameters, you still supply empty parentheses, just as with a parameterless method:

```
() -> { for (int i = 0; i < 1000; i++) doWork(); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp
   = (first, second) // Same as (String first, String second)
      -> Integer.compare(first.length(), second.length());
```

Here, the compiler can deduce that `first` and `second` must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
EventHandler<ActionEvent> listener = event ->
   System.out.println("Thanks for clicking!");
      // Instead of (event) -> or (ActionEvent event) ->
```

NOTE: You can add annotations or the `final` modifier to lambda parameters in the same way as for method parameters:

```
(final String name) -> ...
(@NonNull String name) -> ...
```

You never specify the result type of a lambda expression. It is always inferred from context. For example, the expression

```
(String first, String second) -> Integer.compare(first.length(), second.length())
```

can be used in a context where a result of type `int` is expected.

NOTE: It is illegal for a lambda expression to return a value in some branches but not in others. For example, `(int x) -> { if (x >= 0) return 1; }` is invalid.

1.3 Functional Interfaces

As we discussed, there are many existing interfaces in Java that encapsulate blocks of code, such as `Runnable` or `Comparator`. Lambdas are backwards compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a *functional interface*.

NOTE: You may wonder why a functional interface must have a single *abstract* method. Aren't all methods in an interface abstract? Actually, it has always been possible for an interface to redeclare methods from the `Object` class such as `toString` or `clone`, and these declarations do not make the methods abstract. (Some interfaces in the Java API redeclare `Object` methods in order to attach javadoc comments. Check out the `Comparator` API for an example.) More importantly, as you will see in Section 1.7, "Default Methods," on page 14, in Java 8, interfaces can declare nonabstract methods.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,
   (first, second) -> Integer.compare(first.length(), second.length()));
```

Behind the scenes, the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation dependent, and it can be much more efficient than using traditional inner classes. It is best to think of a lambda expression as a function, not an object, and to accept that it can be passed to a functional interface.

This conversion to interfaces is what makes lambda expressions so compelling. The syntax is short and simple. Here is another example:

```
button.setOnAction(event ->
   System.out.println("Thanks for clicking!"));
```

That's a lot easier to read than the alternative with inner classes.

In fact, conversion to a functional interface is the *only* thing that you can do with a lambda expression in Java. In other programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, and use the variables to save function expressions. However, the Java designers decided to stick with the familiar concept of interfaces instead of adding function types to the language.

NOTE: You can't even assign a lambda expression to a variable of type `Object`—`Object` is not a functional interface.

The Java API defines a number of very generic functional interfaces in the `java.util.function` package. (We will have a closer look at these interfaces in Chapters 2 and 3.) One of the interfaces, `BiFunction<T, U, R>`, describes functions with parameter types `T` and `U` and return type `R`. You can save our string comparison lambda in a variable of that type:

```
BiFunction<String, String, Integer> comp
   = (first, second) -> Integer.compare(first.length(), second.length());
```

However, that does not help you with sorting. There is no `Arrays.sort` method that wants a `BiFunction`. If you have used a functional programming language before, you may find this curious. But for Java programmers, it's pretty natural. An interface such as `Comparator` has a specific purpose, not just a method with given parameter and return types. Java 8 retains this flavor. When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

The interfaces in `java.util.function` are used in several Java 8 APIs, and you will likely see them elsewhere in the future. But keep in mind that you can equally well convert a lambda expression into a functional interface that is a part of whatever API you use today.

NOTE: You can tag any functional interface with the `@FunctionalInterface` annotation. This has two advantages. The compiler checks that the annotated entity is an interface with a single abstract method. And the javadoc page includes a statement that your interface is a functional interface.

It is not required to use the annotation. Any interface with a single abstract method is, by definition, a functional interface. But using the `@FunctionalInterface` annotation is a good idea.

Finally, note that checked exceptions matter when a lambda is converted to an instance of a functional interface. If the body of a lambda expression may throw a checked exception, that exception needs to be declared in the abstract method of the target interface. For example, the following would be an error:

```
Runnable sleeper = () -> { System.out.println("Zzz"); Thread.sleep(1000); };
  // Error: Thread.sleep can throw a checked InterruptedException
```

Since the `Runnable.run` cannot throw any exception, this assignment is illegal. To fix the error, you have two choices. You can catch the exception in the body of the lambda expression. Or assign the lambda to an interface whose single abstract method can throw the exception. For example, the `call` method of the `Callable` interface can throw any exception. Therefore, you can assign the lambda to a `Callable<Void>` (if you add a statement `return null`).

1.4 Method References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. For example, suppose you simply want to print the event object whenever a button is clicked. Of course, you could call

```
button.setOnAction(event -> System.out.println(event));
```

It would be nicer if you could just pass the `println` method to the `setOnAction` method. Here is how you do that:

```
button.setOnAction(System.out::println);
```

The expression `System.out::println` is a *method reference* that is equivalent to the lambda expression `x -> System.out.println(x)`.

As another example, suppose you want to sort strings regardless of letter case. You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

As you can see from these examples, the `::` operator separates the method name from the name of an object or class. There are three principal cases:

- *object*`::`*instanceMethod*
- *Class*`::`*staticMethod*
- *Class*`::`*instanceMethod*

In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method. As already mentioned, `System.out::println` is equivalent to `x -> System.out.println(x)`. Similarly, `Math::pow` is equivalent to `(x, y) -> Math.pow(x, y)`.

In the third case, the first parameter becomes the target of the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.

NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are two versions of the `Math.max` method, one for integers and one for `double` values. Which one gets picked depends on the method parameters of the functional interface to which `Math::max` is converted. Just like lambda expressions, method references don't live in isolation. They are always turned into instances of functional interfaces.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`. It is also valid to use `super`. The method expression

`super::`*instanceMethod*

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
class Greeter {
   public void greet() {
      System.out.println("Hello, world!");
   }
}

class ConcurrentGreeter extends Greeter {
   public void greet() {
      Thread t = new Thread(super::greet);
      t.start();
   }
}
```

When the thread starts, its `Runnable` is invoked, and `super::greet` is executed, calling the `greet` method of the superclass.

NOTE: In an inner class, you can capture the `this` reference of an enclosing class as *EnclosingClass*`.this::`*method* or *EnclosingClass*`.super::`*method*.

1.5 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Button::new` is a reference to a `Button` constructor. Which constructor? It depends on the context. Suppose you have a list of strings. Then you can turn it into an array of buttons, by calling the constructor on each of the strings, with the following invocation:

```
List<String> labels = ...;
Stream<Button> stream = labels.stream().map(Button::new);
List<Button> buttons = stream.collect(Collectors.toList());
```

We will discuss the details of the `stream`, `map`, and `collect` methods in Chapter 2. For now, what's important is that the `map` method calls the `Button(String)` constructor for each list element. There are multiple `Button` constructors, but the compiler picks the one with a `String` parameter because it infers from the context that the constructor is called with a string.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `x -> new int[x]`.

Array constructor references are useful to overcome a limitation of Java. It is not possible to construct an array of a generic type `T`. The expression `new T[n]` is an error since it would be erased to `new Object[n]`. That is a problem for library authors. For example, suppose we want to have an array of buttons. The `Stream` interface has a `toArray` method that returns an `Object` array:

```
Object[] buttons = stream.toArray();
```

But that is unsatisfactory. The user wants an array of buttons, not objects. The stream library solves that problem with constructor references. Pass `Button[]::new` to the `toArray` method:

```
Button[] buttons = stream.toArray(Button[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

1.6 Variable Scope

Often, you want to be able 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.yield();
      }
   };
   new Thread(r).start();
}
```

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello 1,000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. Note that these variables are *not* defined in the lambda expression. Instead, these are parameter variables of the `repeatMessage` method.

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?

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 the free variables is a *closure*. If someone gloats that their language has closures, rest assured that Java has them as well. In Java, lambda expressions are closures. In fact, inner classes have been closures all along. Java 8 gives us closures with an attractive syntax.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. In Java, 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. For example, the following is illegal:

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

There is a reason for this restriction. Mutating variables in a lambda expression is not threadsafe. Consider a sequence of concurrent tasks, each updating a shared counter.

```
int matches = 0;
for (Path p : files)
   new Thread(() -> { if (p has some property) matches++; }).start();
      // Illegal to mutate matches
```

If this code were legal, it would be very, very bad. The increment `matches++` is not atomic, and there is no way of knowing what would happen if multiple threads execute that increment concurrently.

NOTE: Inner classes can also capture values from an enclosing scope. Before Java 8, inner classes were only allowed to access `final` local variables. This rule has now been relaxed to match that for lambda expressions. An inner class can access any effectively final local variable—that is, any variable whose value does not change.

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

Also, it's perfectly legal to mutate a shared object, even though it is unsound. For example,

```
List<Path> matches = new ArrayList<>();
for (Path p : files)
   new Thread(() -> { if (p has some property) matches.add(p); }).start();
      // Legal to mutate matches, but unsafe
```

Note that the variable `matches` is *effectively final*. (An effectively final variable is a variable that is never assigned a new value after it has been initialized.) In our case, `matches` always refers to the same `ArrayList` object. However, the object is mutated, and that is not threadsafe. If multiple threads call `add`, the result is unpredictable.

There are safe mechanisms for counting and collecting values concurrently. In Chapter 2, you will see how to use streams to collect values with certain properties. In other situations, you may want to use threadsafe counters and collections. See Chapter 6 for more information on this important topic.

NOTE: As with inner classes, there is an escape hatch that lets a lambda expression update a counter in an enclosing local scope. Use an array of length 1, like this:

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

Of course, code like this is not threadsafe. For a button callback, that doesn't matter, but in general, you should think twice before using this trick. You will see how to implement a threadsafe shared counter in Chapter 6.

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.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp =
   (first, second) -> Integer.compare(first.length(), second.length());
   // Error: Variable first already defined
```

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

When you use the `this` keyword in a lambda expression, you refer to 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.