Extracted from:

Functional Programming in Java, Second Edition

Harness the Power of Streams and Lambda Expressions

The Pragmatic Bookshelf

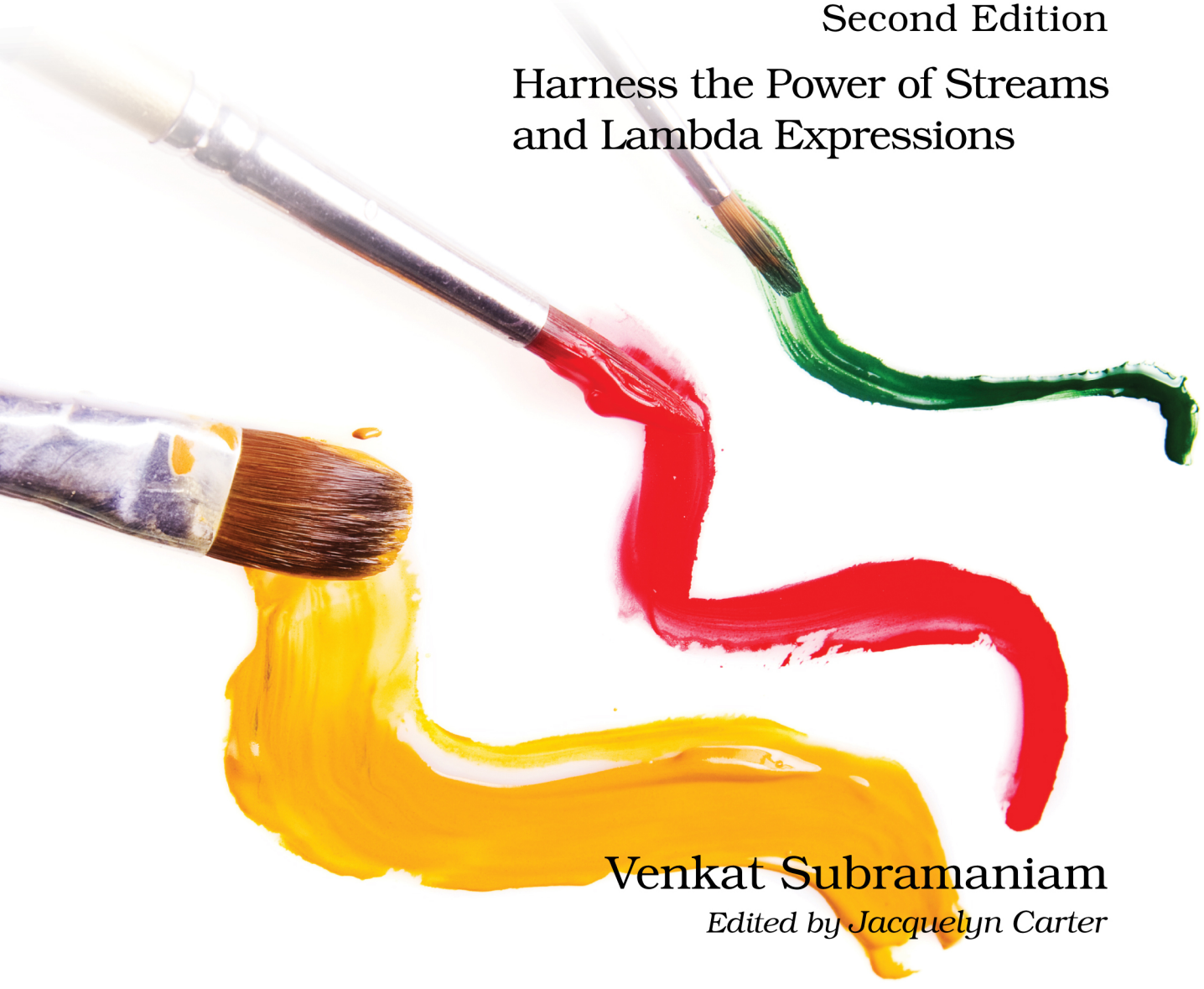
Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming in Java

Second Edition

Harness the Power of Streams and Lambda Expressions



Venkat Subramaniam

Edited by Jacquelyn Carter

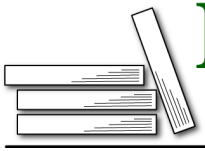
Functional Programming in Java, Second Edition

Harness the Power of Streams and Lambda Expressions

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

Sir Charles Antony Richard Hoare's quote is used by permission of the ACM.
Abelson and Sussman's quote is used under Creative Commons license.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-979-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2023

To perseverance.

CHAPTER 6

Working with Resources

We make extensive use of resources when programming—we access files, communicate to remote services, use database connections, and so on. And, that often involves working with issues like the timely release of the resources, locking for synchronization, and handling exceptions that may arise. Dealing with all of these concerns at the same time can get daunting. In this chapter we'll see how we can structure our code, using lambda expressions, to alleviate the pain of managing resource access in general—that is, to deal with the mundane tasks that we'd better not get wrong.

We may have been led to believe that the JVM automates all garbage collection (GC). It's true that we could let the JVM handle it if we're only using internal resources. But GC is our responsibility if we use external resources, such as when we connect to databases, open files and sockets, or use native resources.

Java provides a few options to properly clean up resources, but, as we'll see in this chapter, none are as effective as what we can do with lambda expressions. We'll use lambda expressions to implement the *execute around method (EAM)* pattern, which gives us better control over the sequencing of operations.[1] By using this pattern, as we'll see, we move the burden of managing the resource lifetime from the user of a piece of code to its developer who has better knowledge and control over those details.

We'll then take the ideas of managing resources further to streamline more operations around the use of resources. We'll see how to manage the critical and error-prone task of managing locks in a safe way. Finally, we'll look at how these ideas can also help us with writing exception tests in a concise and elegant way.

1. http://c2.com/cgi/wiki?ExecuteAroundMethod

Cleaning Up Resources

GC can be a pain to deal with. A company asked me to help debug a problem—one programmer described the issue as "it works fine…most of the time." The application failed during peak usage. It turned out that the code was relying on the finalize() method to release database connections. The JVM figured it had enough memory and opted not to run GC. Since the finalizer was rarely invoked, it led to external resource clogging and the resulting failure.

We need to manage situations like this in a better way, and lambda expressions can help. Let's start with an example problem that involves GC. We'll build the example using a few different approaches, discussing the merits and deficiencies of each. This will help us see the strengths of the final solution using lambda expressions.

Peeking into the Problem

We're concerned with external resource cleanup, so let's start with a simple example class that uses a FileWriter to write some messages.

resources/fpij/FileWriterExample.java

```
public class FileWriterExample {
  private final FileWriter writer;

  public FileWriterExample(final String fileName) throws IOException {
    writer = new FileWriter(fileName);
  }
  public void writeStuff(final String message) throws IOException {
    writer.write(message);
  }
  public void finalize() throws IOException { //Deprecated in Java 9
    writer.close();
  }
  //...
}
```

In the FileWriterExample class's constructor, we initialize an instance of FileWriter, giving it the name of a file to write to. In the writeStuff() method we write the given message to the file using the instance of the FileWriter we created. Then, in the finalize() method we clean up the resource, calling close() on it with the hope that it will flush the content to the file and close it.

At first glance, the code seems reasonable. After all, classes written in many Java applications use the finalize() method to clean up resources, a standard practice until Java 8, and a lot of legacy code still uses that function. In reality, expecting the resources to be cleaned up automatically is rather wishful thinking.

If the JVM finds that sufficient memory is available, then the GC won't be invoked and thus the finalize() method won't be called for a long time. This will result in the resource not being released in a timely manner and can also lead to resource contention issues. This is one of the reasons why the finalize() method was deprecated in Java 9, to encourage programmers to move away from using that method. We'll look at alternatives to the finalize() method shortly, but first, let's write a main() method to use the FileWriterExample class.

resources/fpij/FileWriterExample.java

```
public static void main(final String[] args) throws IOException {
  final FileWriterExample writerExample =
    new FileWriterExample("peekaboo.txt");

  writerExample.writeStuff("peek-a-boo");
}
```

We created an instance of the FileWriterExample class and invoked the writeStuff() method on it, but if we ran this code, we'd see that the peekaboo.txt file was created but it's empty. The finalizer never ran; the JVM decided it wasn't necessary as there was enough memory. As a result, the file was never closed, and the content we wrote was not flushed from memory.

If we create several instances of the FileWriterExample class in a long-running process, we'll end up with several open files. Many of these files won't be closed in a timely manner since the JVM has a lot of memory and sees no reason to run GC.

Let's fix the problem by adding an explicit call to close(), and let's get rid of the finalize() method.

Say Farewell to finalize()

The finalize() method was deprecated in Java 9. Take a few minutes to examine your own production code to see if the finalize() method is still present in any of the classes. If you find them, note the occurrences down as technical debt and schedule time to clean those up using the techniques you learn in this chapter.

Closing the Resource

Even though the object's memory cleanup is still at the mercy of the JVM's GC, we could convince ourselves that the external resources used by an instance may be quickly cleaned up with an explicit call. That, unfortunately, will result in more issues. To see this, let's write a close() method.

resources/fpij/FileWriterExample.java
```
public void close() throws IOException { //Not a good solution
  writer.close();
}
```

In the close() method, in turn, we call the FileWriter instance's close() method. If we used any other external resources in the FileWriterExample, we can clean them up here, as well. Let's make explicit use of this method in the main() method.

resources/fpij/FileWriterExample.java
```
final FileWriterExample writerExample =
  new FileWriterExample("peekaboo.txt");

writerExample.writeStuff("peek-a-boo");
writerExample.close();
```

If we run the code now and look into the peekaboo.txt file, we'll see the peek-a-boo message. The code works, but it's far from perfect.

The explicit call to close() cleans up any external resources the instance uses as soon as we indicate the instance is no longer needed. But we may not reach the call to the close() method if there was an exception in the code leading up to it. We'll have to do a bit more work to ensure the call to close() happens. Let's take care of that next.

Ensuring Cleanup

We need to ensure the call to close() happens whether or not there's an exception. To achieve this, we can wrap the call in a finally block.

resources/fpij/FileWriterExample.java
```
final FileWriterExample writerExample =
  new FileWriterExample("peekaboo.txt");

try { //Rather verbose
  writerExample.writeStuff("peek-a-boo");
} finally {
  writerExample.close();
}
```

This version will ensure resource cleanup even if an exception occurs in the code, but that's a lot of effort and the code is verbose and smelly. Java 7 introduced a feature to reduce such smells, as we'll see next.

Using ARM

The automatic resource management (ARM) is a feature that has been available since Java 7 and is useful for automatically releasing a resource at the end of its usage. When used properly, ARM can reduce verbosity in code. Rather

than using both the try and finally blocks that we used in the previous example, we can use the ARM feature with a special form of the try block with a resource attached to it. When this syntax is used, the Java compiler takes care of automatically inserting, in the bytecode, the finally block and a call to the close() method.

Let's see how the code would look with ARM; we'll use an instance of a new FileWriterARM class.

resources/fpij/FileWriterARM.java

```
try(final FileWriterARM writerARM = new FileWriterARM("peekaboo.txt")) {
  writerARM.writeStuff("peek-a-boo");

  System.out.println("done with the resource...");
}
```

We created the instance of the class FileWriterARM within the safe haven of the *try-with-resources* form and invoked the writeStuff() method within its block. When we leave the scope of the try block, the close() method is automatically called on the instance/resource managed by this try block. For this to work, the compiler requires the managed resource class to implement the AutoCloseable interface, which has just one method, close().

The rules around AutoCloseable have gone through a few changes in Java. First, Stream implements AutoCloseable and, as a result, all input/output (I/O)-backed streams can be used with try-with-resources. The contract of AutoCloseable has been modified from a strict "the resource *must* be closed" to a more relaxed "the resource *can* be closed." If we're certain that our code uses an I/O resource, then we should use try-with-resources.

Here's the FileWriterARM class used in the previous code.

resources/fpij/FileWriterARM.java

```
public class FileWriterARM implements AutoCloseable {
  private final FileWriter writer;

  public FileWriterARM(final String fileName) throws IOException {
    writer = new FileWriter(fileName);
  }

  public void writeStuff(final String message) throws IOException {
    writer.write(message);
  }

  public void close() throws IOException {
    System.out.println("close called automatically...");
    writer.close();
  }

  //...
}
```

Let's run the code and look at the peekaboo.txt file and the console for the code's output.

```
done with the resource...
close called automatically...
```

We can see the close() method was called as soon as we left the try block. The instance we created when entering the try block isn't accessible beyond the point of leaving the block. The memory that instance uses will be garbage-collected eventually based on the GC strategy the JVM employs.

The previous code using ARM is concise and charming, but the programmers have to remember to use it. The code won't complain if we ignore this elegant construct; it will simply create an instance and call methods like writeStuff() outside of any try blocks. If we're looking for a way to ensure timely cleanup and avoid programmer errors, we have to look beyond ARM, as we'll do next.