Extracted from:

Java By Comparison

Become a Java Craftsman in 70 Examples

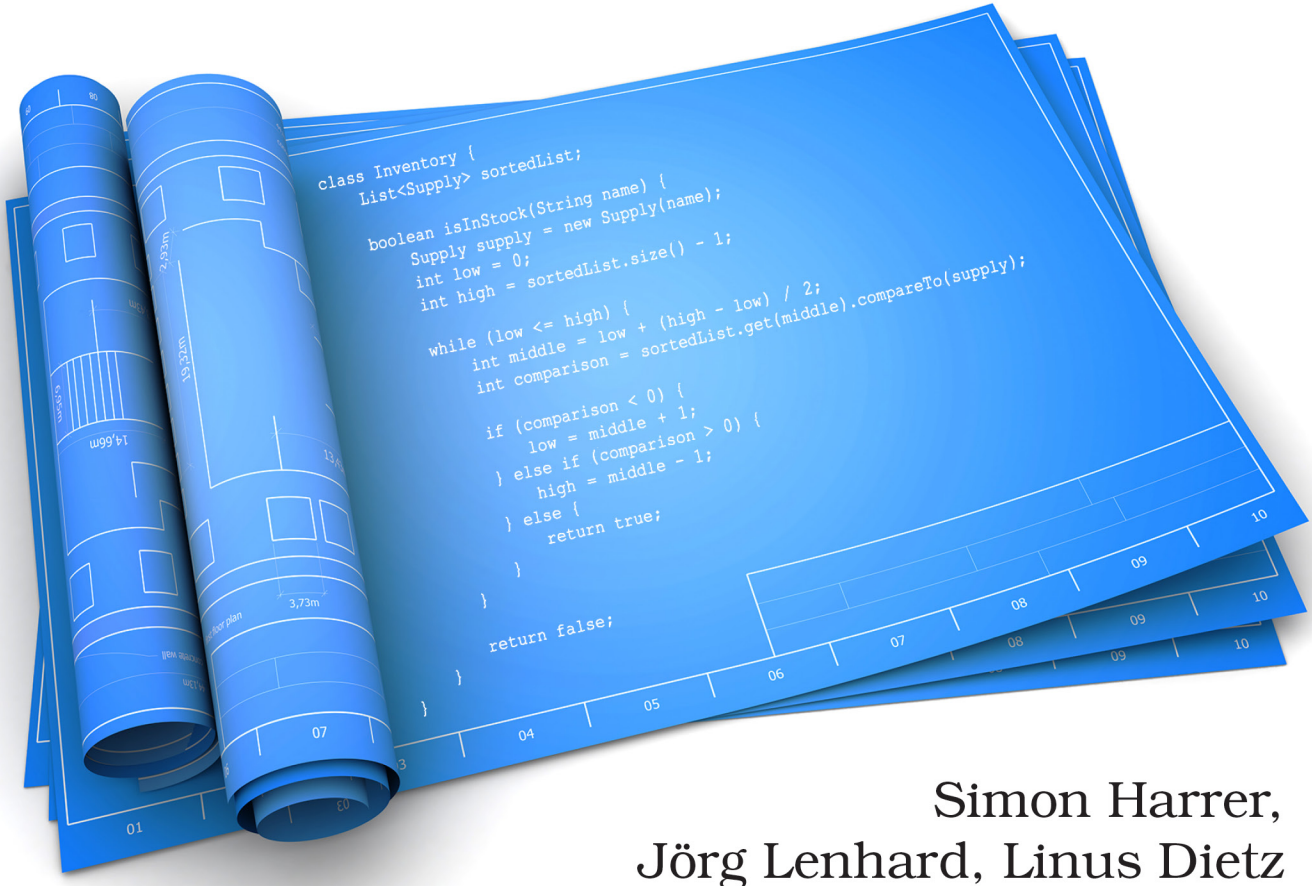
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Java by Comparison

Become a Java Craftsman in 70 Examples



Simon Harrer,
Jörg Lenhard, Linus Dietz

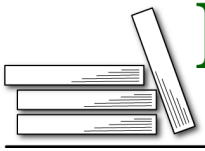
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

Java By Comparison

Become a Java Craftsman in 70 Examples

Simon Harrer
Jörg Lenhard
Linus Dietz

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-287-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—December 6, 2017

Split Method with Boolean Parameters

```
class Logbook {

    final static Path CAPTAIN_LOG = Paths.get("/var/log/captain.log");
    final static Path CREW_LOG = Paths.get("/var/log/crew.log");

➤   void log(String message, boolean classified) throws IOException {
        if (classified) {
            writeMessage(message, CAPTAIN_LOG);
        } else {
            writeMessage(message, CREW_LOG);
        }
    }

    void writeMessage(String message, Path location) throws IOException {
        String entry = LocalDate.now() + " " + message;
        Files.write(location, Collections.singleton(entry),
                StandardCharsets.UTF_8, StandardOpenOption.APPEND);
    }
}
```

In general, a method should specialize on a single task only. Boolean method parameters show that a method does more than that.

In this example, we're revisiting a refined version of the `Logbook` class. We separate messages into classified and non-classified messages via a `boolean` parameter of the `log()` method. Here's a usage example:

```
logbook.log("Aliens sighted!", true);
logbook.log("Toilet broken.", false);
```

This code doesn't contain a bug, but it's less readable and structured than it should be. Everybody who reads it will have to figure out what purpose that boolean parameter serves. If they can figure out the purpose, there's still a risk that they'll interpret it the wrong way and write a classified message in the crew log.

Using a `boolean` value as a method parameter helps you loudly to proclaim that the method does more than one thing. This is sometimes okay, but it generally makes your code less understandable because it's hard to see on the calling side what the `boolean` parameter actually achieves.

Check out how we can improve this:

```
class Logbook {

    final static Path CAPTAIN_LOG = Paths.get("/var/log/captain.log");
    final static Path CREW_LOG = Paths.get("/var/log/crew.log");

➤   void writeToCaptainLog(String message) throws IOException {
        writeMessage(message, CAPTAIN_LOG);
    }

➤   void writeToCrewLog(String message) throws IOException {
        writeMessage(message, CREW_LOG);
    }

    void writeMessage(String message, Path location) throws IOException {
        String entry = LocalDate.now() + " " + message;
        Files.write(location, Collections.singleton(entry),
                StandardCharsets.UTF_8, StandardOpenOption.APPEND);
    }
}
```

Whenever you see a method that uses a boolean input parameter, chances are that you can improve the code by separating it into multiple methods.

To do so, you remove the boolean method parameter and add a new method for each control-flow path that the parameter was distinguishing between. You can even give the new methods expressive and meaningful names and further enhance the readability of the code!

This is also what we've done in the code above: we've added the two new methods writeToCaptianLog() and writeToCrewLog(). Consider the new usage now:

```
logbook.writeToCaptainLog("Aliens sighted!");
logbook.writeToCrewLog("Toilet broken. Again...");
```

As you can see, this is much more readable than the previous version. The method names make it clear which log a message belongs to, and you can tell from the calling code what the method is intended to achieve.

Remember: Good Design is Difficult.

Hardly anybody gets it right on their first try. In the end, this is good news because it means that your skills will be in high demand when you master object-oriented design. But be aware that it's harder to tell good from bad design than it is to spot a botched line of code. Good design is less clear-cut, and it requires you to have an intuition and a "feeling" for it. The only real way to get this intuition is to do a lot of coding, to try out different designs, and to see what fails and what feels just right.