Extracted from:

Java by Comparison

Become a Java Craftsman in 70 Examples

This PDF file contains pages extracted from *Java by Comparison*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

The Pragmatic Bookshelf

Raleigh, North Carolina

Java by Comparison

Become a Java Craftsman in 70 Examples

```
class Inventory {
    List<Supply> sortedList;

    boolean isInStock(String name) {
        Supply supply = new Supply(name);
        int low = 0;
        int high = sortedList.size() - 1;

        while (low <= high) {
            int middle = low + (high - low) / 2;
            int comparison = sortedList.get(middle).compareTo(supply);

            if (comparison < 0) {
                low = middle + 1;
            } else if (comparison > 0) {
                high = middle - 1;
            } else {
                return true;
            }
        }

        return false;
    }
}
```
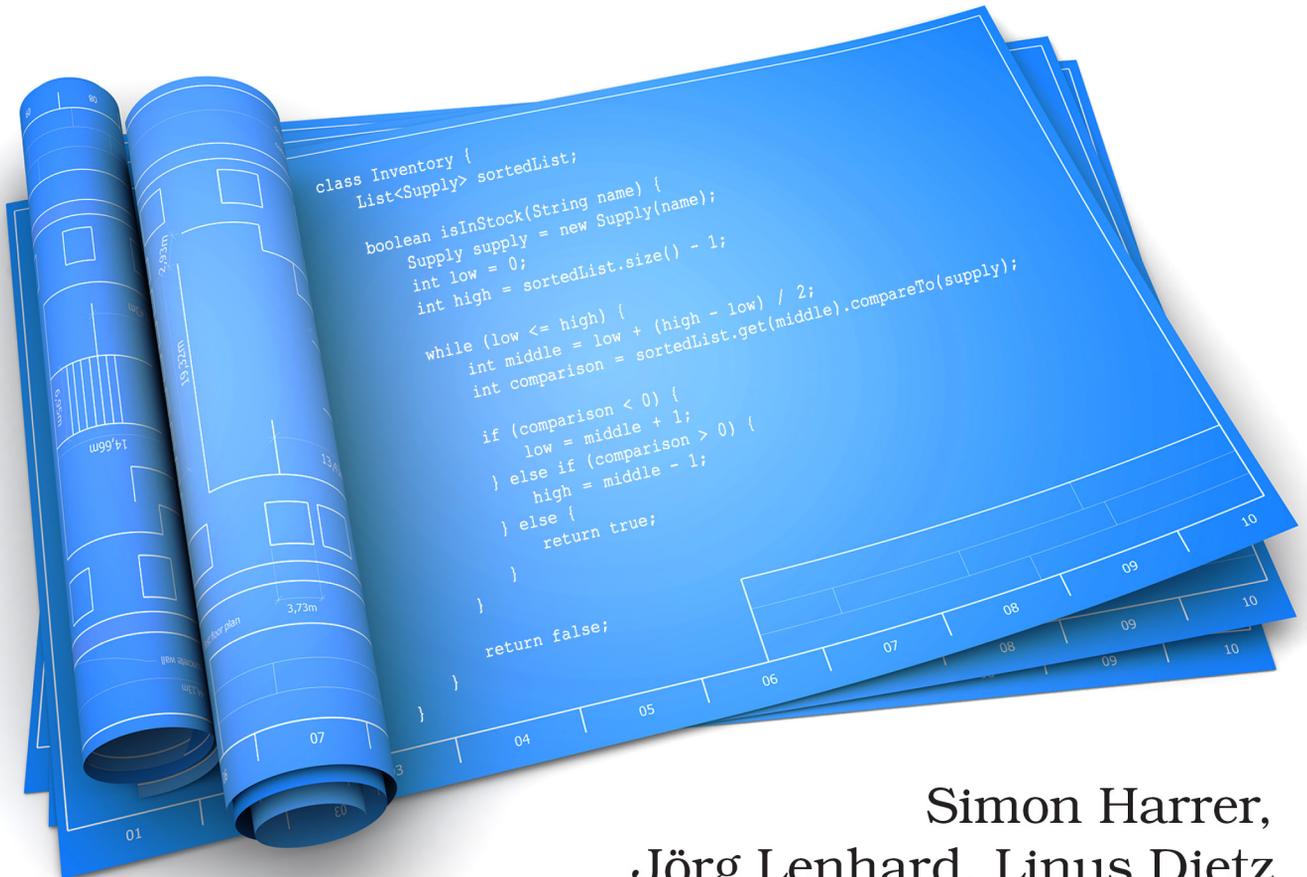
Simon Harrer,
Jörg Lenhard, Linus Dietz
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

Java by Comparison

Become a Java Craftsman in 70 Examples

Simon Harrer

Jörg Lenhard

Linus Dietz

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Andrea Stewart
Copy Editor: Liz Welch
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

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

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

Return Boolean Expressions Directly

```
class Astronaut {

    String name;
    int missions;

    boolean isValid() {
        if (missions < 0 || name == null || name.trim().isEmpty()) {
            return false;
        } else {
            return true;
        }
    }
}
```

Next, let's look at another way you can cut clutter in your code. In this case, we don't need the if statement to pursue our goals. Let's find out why.

You can see a typical validation method in the snippet. The method checks a couple of attributes of an object, an integer and a String.

The integer attribute refers to a number of missions. This number shouldn't be negative.

The String attribute shouldn't be null; otherwise we risk that a NullPointerException crashes the execution at some point.

Additionally, it should not be empty, because every Astronaut should have an actual name. The call to name.trim() removes all kinds of preceding or trailing space characters, such as white spaces or tabs. If there's something left after the call to trim(), it'll be an actual sequence of characters.

There's no functional error in the code. The problem, as is so often the case, is that the code is more complex and less readable than it could be. More precisely, the if statement is entirely unnecessary. It's clutter code that only distracts from the actual semantics.

Let's see what we can do about it! The key is the return type of our method: boolean. Because of this, we don't have to wrap everything in an if statement—we can return the value right away, like this:

```java
class Astronaut {

    String name;
    int missions;

    boolean isValid() {
        return missions >= 0 && name != null && !name.trim().isEmpty();
    }
}
```

This change removes a level of indentation and branching. The method is now very concise and much easier to read!

We condensed the five lines of the if statement into a single line of code. To retain the same semantics, we had to apply Boolean arithmetic to the condition. Essentially, we applied De Morgan's laws[1] and negated the condition. Here's a quick definition of these famous laws, which you'll need often:

```java
!A && !B == !(A || B) // true
!A || !B == !(A && B) // true
```

There are times when the condition is more complex than this one. If this is the case, you should think about breaking it into smaller chunks. Our advice is to capture parts of a condition with variables that have meaningful names. Consider the following example:

```java
boolean isValid() {
    boolean isValidMissions = missions >= 0;
    boolean isValidName = name != null && !name.trim().isEmpty();
    return isValidMissions && isValidName;
}
```

As a rule of thumb, you should consider such a simplification if you combine more than two different conditions. If you need parts of the condition elsewhere, consider putting them into separate methods, as explained in *Simplify Boolean Expressions*, on page ?.

Note that the solution depicted here only works with boolean return types.

---

1. https://en.wikipedia.org/wiki/De_Morgan%27s_laws