Extracted from:

Java By Comparison

Become a Java Craftsman in 70 Examples

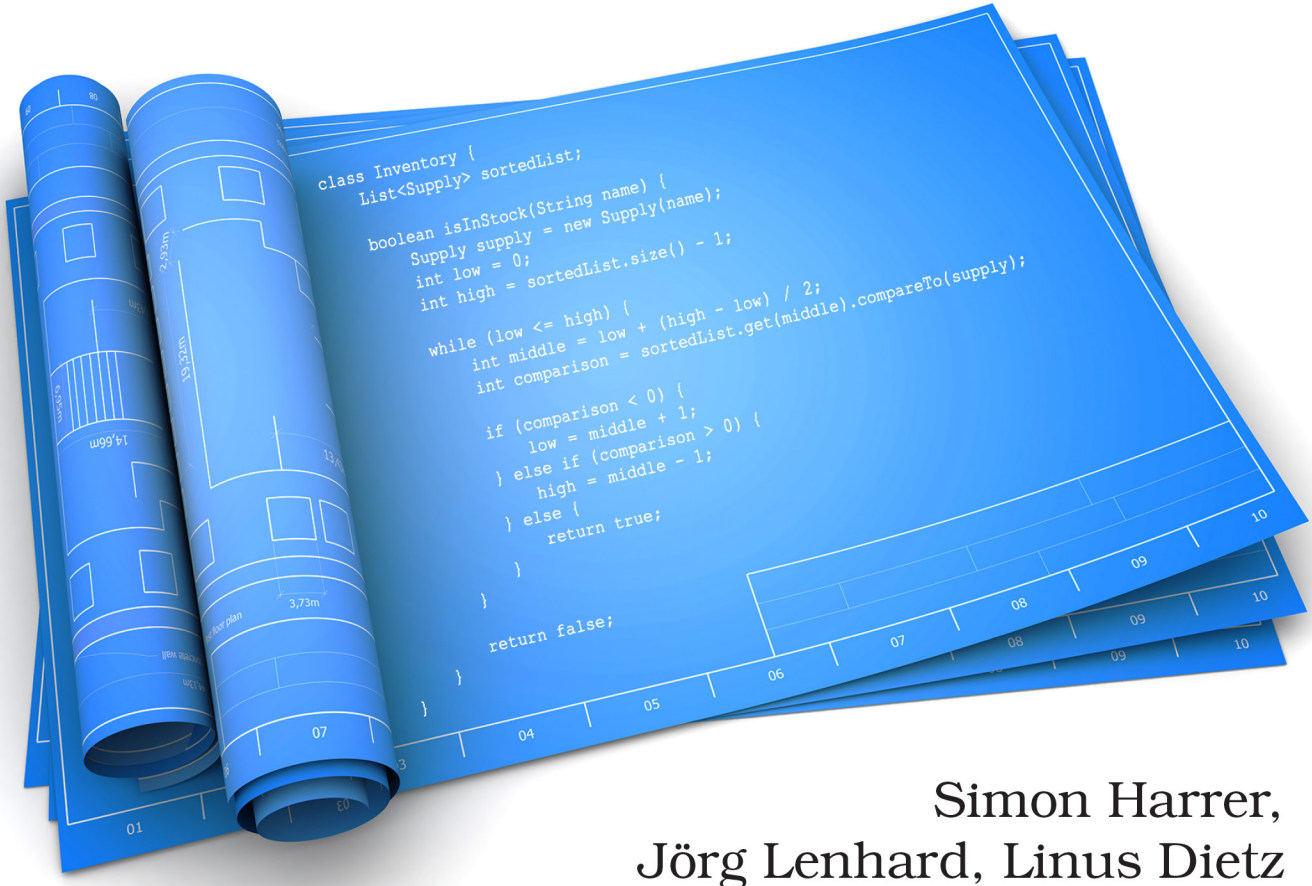
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Java by Comparison

Become a Java Craftsman in 70 Examples



Simon Harrer,
Jörg Lenhard, Linus Dietz

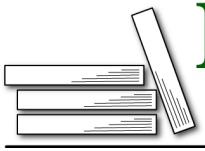
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

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

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 so often, 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:

```
class Astronaut {

    String name;
    int missions;

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

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 arithmetics to the condition. Essentially, we applied De Morgan's laws[1] and negated the condition.

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

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:

```
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. In Chapter 5, *Prepare for Things Going Wrong*, on page ?, you'll see that it's often preferable to throw Exceptions instead of returning false for invalid arguments in validation methods.

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