Extracted from:

Java by Comparison

Become a Java Craftsman in 70 Examples

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;
    }
}
```
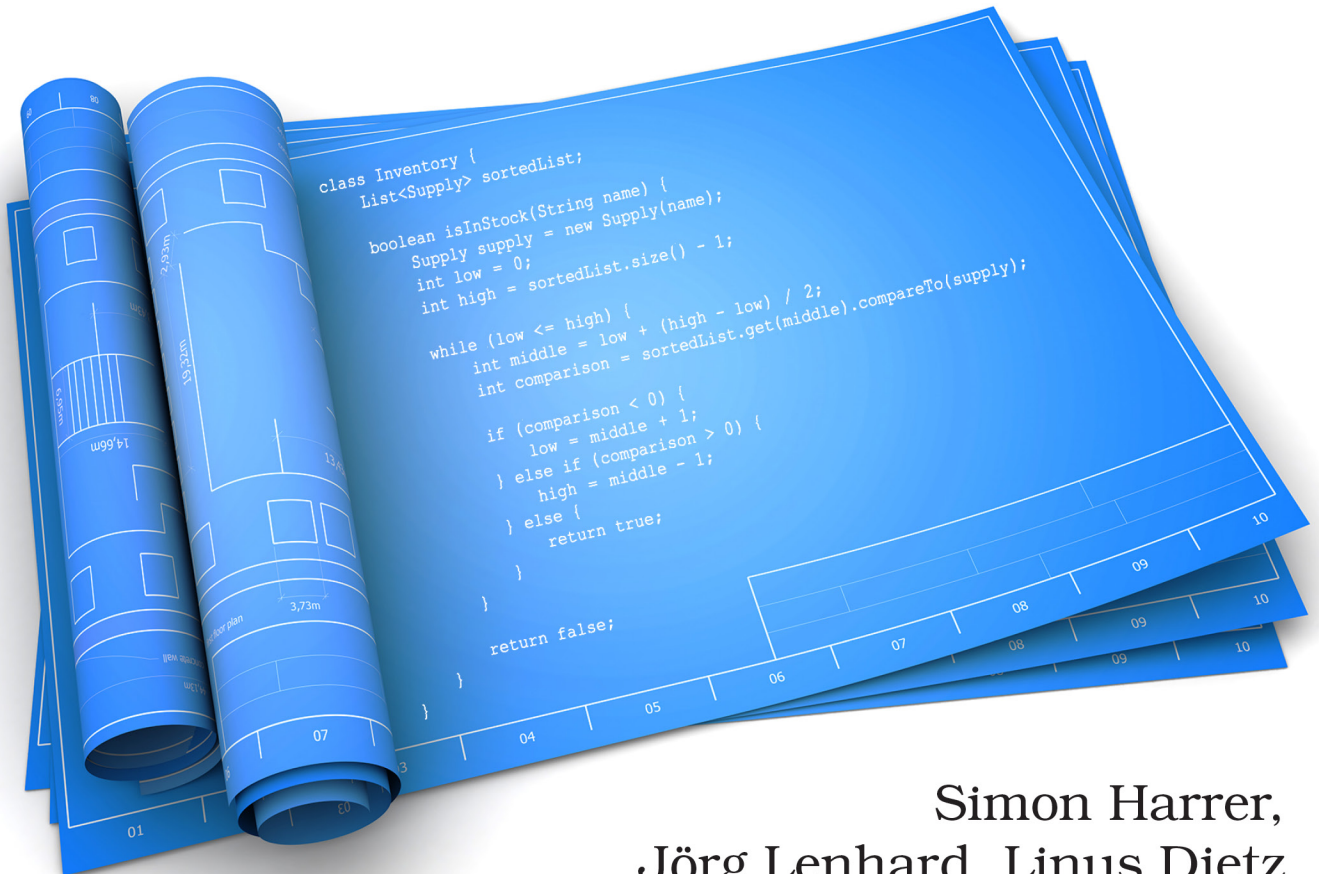
Simon Harrer,
Jörg Lenhard, Linus Dietz

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*.

Document Using Examples

```java
class Supply {

    /**
     * The code universally identifies a supply.
     *
     * It follows a strict format, beginning with an S (for supply), followed
     * by a five digit inventory number. Next comes a backslash that
     * separates the country code from the preceding inventory number. This
     * country code must be exactly two capital letters standing for one of
     * the participating nations (US, EU, RU, CN). After that follows a dot
     * and the actual name of the supply in lowercase letters.
     */
    static final Pattern CODE =
            Pattern.compile("^S\\d{5}\\\\(US|EU|RU|CN)\\.[a-z]+$");
}
```

Some programming constructs are very powerful, but also very complex. Regular expressions fall into this category. You should document complex constructs in a way that makes them easier to understand.

Above, you see a lengthy regular expression. Its name, CODE, doesn't help you to understand what it's good for, but there's also a lengthy comment.

This might seem like a good solution. After all, at least there's a comment and not just the code. The comment describes the sort of strings that the regex will match, and the code even takes care that the regex is compiled exactly once.

The problem isn't that the documentation is wrong (it's not). The problem is that it's less precise than it should be, and it only duplicates what a skilled developer can already read from the regex code itself.

"Lead by example" is always good advice. This comes in very handy for documenting a regex.

Comment Mode

You can also add comments within your regular expressions like this: B[1-9]# Beta Release Numbers. For that to work, you need to pass in the Pattern.COMMENTS flag. Those comments can help, especially for long and complex regular expressions, but we think that examples are even more helpful.

Take a look at how we can improve the documentation:

```java
class Supply {

    /**
     * The expression universally identifies a supply code.
     *
     * Format: "S<inventory-number>\<COUNTRY-CODE>.<name>"
     *
     * Valid examples: "S12345\US.pasta", "S08342\CN.wrench",
     * "S88888\EU.laptop", "S12233\RU.brush"
     *
     * Invalid examples:
     * "R12345\RU.fuel"       (Resource, not supply)
     * "S1234\US.light"       (Need five digits)
     * "S01234\AI.coconut"    (Wrong country code. Use US, EU, RU, or CN)
     * " S88888\EU.laptop "   (Trailing whitespaces)
     */
    static final Pattern SUPPLY_CODE =
            Pattern.compile("^S\\d{5}\\\\(US|EU|RU|CN)\\.[a-z]+$");
}
```

The comment above is a little more lengthy, but it's also more structured, and it provides a lot more information. In a nutshell, it describes the format in semi-natural language, and it gives several valid and invalid examples.

The starting sentence is the same as before, but the Format: part condenses the content of the prior example into a single line. This line's just a variation of the code, but with the actual semantics instead of regex syntax. <inventory-number> is just so much more understandable than \\d{5}. The parts that are just syntax, such as \ or ., don't need a further explanation.

Next, there are concrete examples. Usually, a valid example lets you understand the expression within a second. That's not something that the code or a lengthy explanation does. The invalid examples are a good quick reference when something goes wrong.

Sure, examples don't usually cover all possible cases. But they're good enough in 90 percent of all cases, and you'll find them so much easier to understand. Plus, we think you should add those examples as unit tests!

Last, we've also taken the opportunity to give the variable a more meaningful name, SUPPLY_CODE.