Java by Comparison

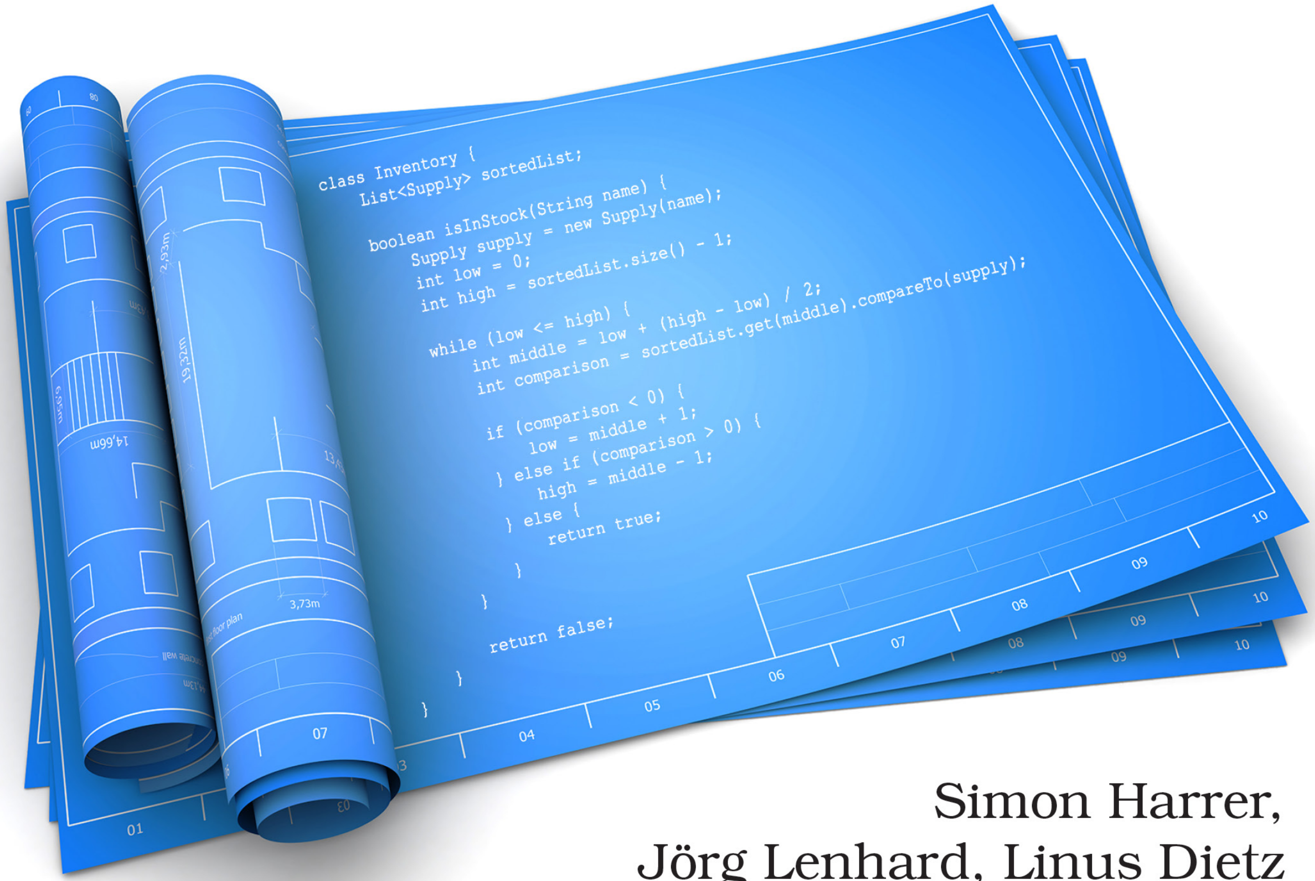
Become a Java Craftsman in 70 Examples

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Java by Comparison

Become a Java Craftsman in 70 Examples



```
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

The Pragmatic Bookshelf
Raleigh, North Carolina

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*.

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: P1.0—March 2018

If you can’t provide a convincing before/after code for your pattern, you’re selling snake oil.

➤ *David Heinemeier Hanson*

Welcome!

We agree with David Heinemeier Hanson.

If you propose a programming technique, you should be able to demonstrate as concisely as possible how your proposal is better than what was there before. The before/after approach puts all the facts on the table: you can directly compare the new code to the old code. Only then can you make an informed decision about which one’s better.

The same holds true when you’re learning how to program. Comparing good code to bad code is really helpful when you’re trying to figure out how to code in Java.

We used to teach programming at a university for over six years. After a few lectures, there were *always* a few students asking us, “How can I improve my coding skills further?” These students excelled in their programming assignments, anyway, so we gave them the default advice: “Read code from professionals.”

Honestly, our advice wasn’t very helpful. There’s plenty of code out in the open source software world, but it’s hard to tell where to start. Beginners get quickly overwhelmed with the complexity of professional code in real-life projects. And how do you know if a piece of source code is actually of high quality? Even if it is, how can someone with only a few months of programming experience distinguish it from a flawed hack?

That’s where this book comes into play. It’s a companion that guides your reading in the right direction. We’ll help you learn how to write good Java code by comparing pieces of bad code with good code. And be assured: we’ve seen our fair share of bad and good code over the years, reviewing code written by students in academia, professionals in industry, and contributors in open source projects.

This brings us back to the before/after approach. In this book, we’ll provide you with 70 before/after code snippets. These snippets will help any beginner in Java programming to improve. We identified these snippets during our time of teaching Java to undergraduates at the university. They're all based on code we faced when correcting our students' programming assignments.

Our approach in this book is simple: Given a code snippet, we'll first explain what's wrong and why. Then, we'll show you how you can transform the code into a better solution.

Who Should Read This Book

This book is for people who are learning to program in Java at a beginner or intermediate level. It's also a classroom resource for teachers who coach new developers in their journey to become programmers. Here, we're giving you tips and tricks based on more recent Java 8[1] syntax for resource handling, functional programming, or testing.

You should read this book after you've learned the basic Java syntax—after you're able to write small programs with conditions and loops and you know the basics of object-oriented programming. You should be able to write code that compiles, runs, and solves small tasks like FizzBuzz (see *Are You Ready? Try the Self-Assessment*, on page vii). You should be able to implement simple algorithms, and you should know how to use basic data structures like a list, queue, stack, or map. And obviously, you should be having fun while doing all that!

If you feel a deep satisfaction when you solve a complicated problem, then that's an excellent start. But of course, you also know that there's still a lot to learn. When you reflect on your skills and you have to confess that you don't have a lot of experience in programming in Java (or even programming in general), then you can get the maximum benefit out of this book. This means that you probably haven't yet developed a sense for *clean code* and the best practices an experienced developer applies.

It's about time to change this!

Of course, if you already know more advanced books on code quality, readability, maintainability, and clean code in Java, such as *Effective Java [Blo18]* and *Clean Code [Mar08]*, then you've already come a long way. Nevertheless, you can still find something new here and there.

1. We know that Oracle already released Java 9. Rest assured that everything in this book is still valid in Java 9.

Teaching Clean Code Using This Book

If you teach programming to newcomers as a senior developer at a company, you're certainly aware of the best practices we'll go over in this book. You might even disagree with some, depending on what you are working on. Still, this book can make your life easier when training a junior developer—just use it as a reference. When you spot problems in your apprentice's code, point her to the corresponding item in this book. Your student can read a concise and simple example for the problem you're hinting at, as well as how to get to the solution of the problem. At the very least, this saves you time because you won't have to write an explanation yourself.

Our experience in developing teaching concepts for Java learners in their second to third year can also be useful for teaching in academia. This book is the product of our combined knowledge of over fifteen years of teaching Java to undergraduate college students in an advanced course with focus on code quality. It might not teach Java from the ground up, but it can accompany basically any course that involves programming tasks. In particular, you can use the book as a reference when you asses student code, as we've described in a workshop paper: *Teaching Clean Code [DMHL18]*.

Are You Ready? Try the Self-Assessment

If you're a new programmer, we suggest that you do a short self-assessment to see if you're ready for the material in this book: the FizzBuzz Test (see the *Fizz Buzz Test*[2] or *Using FizzBuzz to Find Developers who Grok Coding*[3]). Some employers use this test in job interviews to determine if an applicant can program at all. The task goes like this:

> Write a Java program that prints the numbers from 1 to 100 to the console. But for multiples of three, print *Fizz* instead of the number and for multiples of five, print *Buzz*. For numbers that are multiples of both three and five, print *FizzBuzz*.

To make the test more interesting, we'll extend it a bit here by making sure that you can apply object orientation and use classes and interfaces as well. You should implement the FizzBuzz algorithm in a class called `ConsoleBasedFizzBuzz`, which implements a `FizzBuzz` interface. This interface provides a method that takes the first and last numbers to print as arguments. In the `main` method of a separate `Main` class, you should use the `FizzBuzz` interface with its `ConsoleBasedFizzBuzz` implementation to count from 1 up to the value passed from the console. Here, you'll see the outlined structure in a short template.

2. http://c2.com/cgi/wiki?FizzBuzzTest
3. https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

```
interface FizzBuzz {
    void print(int from, int to);
}

class ConsoleBasedFizzBuzz implements FizzBuzz {
    // TODO implement FizzBuzz interface
}

class Main {
    // TODO use a main method
    // TODO print fizz buzz from 1 to max
    // TODO max is passed from the console
}
```

You should be able to finish this exercise in about *15 minutes*. One of the links we listed also contains solutions to the FizzBuzz challenge that you can compare to your own. If you can do it, then you're ready to get the most out of this book. If not, don't worry! Keep reading anyway. It might take you a little longer, and you might have a harder time understanding a comparison here and there. But if you practice programming by solving small exercises like the ones in this book, you'll get on track quickly.

Many good resources are available online for practicing your programming skills and getting feedback on your code. Have a look at codewars.com[4] or cyber-dojo.[5] These pages let you train your programming skills in various levels of difficulty. If you have a mathematical background, you'll find solving the problems of Project Euler[6] quite appealing.

If, on the other hand, you find the FizzBuzz test terribly easy, and your solution compiles and runs within seconds, be aware that you might already know some of the practices that we outline in this book. You can still get something out of it, of course. We've made all the comparisons self-contained. So feel free to jump around and skip the parts that you already know.

Why Read This Book?

Every developer has a number of requirements in mind that she considers prerequisites for *good* or *clean code*. As long as a piece of code doesn't violate any of these requirements, it qualifies as *good* or *clean* from the viewpoint of the developer. Different people have different requirements. And programming languages differ, of course. But still, for a given language, there's typically a set of *"core"* requirements and best practices. These are aspects that the

4. https://www.codewars.com/?language=java
5. http://cyber-dojo.org/
6. https://projecteuler.net

community of developers recognizes and accepts, even if they aren't written down explicitly. In this book, we're trying to provide you—someone who might not yet be aware of many of the practices in the Java community—with a set of best practices for clean code in Java.

As a beginner, your list of requirements for good Java code might be as short as this one:

- The code must compile.
- The output must be correct.

These items are about the functional correctness of your program, but they don't tell much about the quality of your code. An experienced programmer cares about a lot more than that, and her checklist is much longer. She just needs a quick glance at a piece of code to detect flaws, bad naming, hard-to-test methods, inconsistencies, bad practices, and much more.

The aim of this book is to train your brain to internalize more checklist items, helping you on your way to becoming an experienced and professional programmer. Each of the items in this book represents such a checklist item.

Conventions Used in This Book

Throughout the book, we use a specific structure for explaining each checklist item. We call it "comparison," and we zealously stick to it.

Each comparison has a catchy name. This helps you to memorize it for your mental checklist. Take a look at the table of contents—it acts as the checklist of this book. The name of a comparison makes it easier to talk to other people about what you've read. We've named the comparisons in a way that encourages you to consider them as a recommendation. You can also put "You should" in front of a comparison name—for example: You should *avoid unnecessary comparisons*. For the upcoming example, we selected the catchy name "Never Trust Your Input."

Directly after the name, we'll present a snippet of code and highlight a problem in it. This can be a block of several lines or just a single one, like this:

```
class HelloWorld {

    public static void main(String[] args) {
➤       System.out.println("Hello " + args[0]);
    }
}
```

In these code snippets, we want to distract you as little as possible. That's why we have to shorten the code a little bit to get to the essentials:

1. We leave out `import` statements and `package` declarations. The downloadable code[7] contains those declarations, of course—it would not compile otherwise.
2. We avoid visibility modifiers, such as `public` or `private`, unless they're explicitly required.

Following the code snippet, we explain to you what the problem is. On the way, we provide references to further reading, such as JavaDoc, related items, or web pages.

Then, we show you the code of a solution and highlight the solution-specific parts.

```java
class HelloWorld {

    public static void main(String[] arguments) {
➤       if (isEmpty(arguments)) {
            return;
        }

➤       System.out.println("Hello " + arguments[0]);
    }

➤   private static boolean isEmpty(String[] array) {
        return (array == null) || (array.length == 0);
    }
}
```

Sometimes, there's very little difference between the problem code and the solution. Sometimes, there's a lot of difference. Either way, the snippets themselves should already teach you something—a certain style that helps you to produce better code. We deliberately keep each comparison on two pages. The left-hand side (or first page) shows the problem, and the right-hand side (or second page) shows the solution. This way, you can always compare both code snippets without having to flip pages back and forth. This makes it a lot easier to learn what an item is about. If you're reading this book as a PDF with Adobe Reader, make sure you've enabled two-page view and that the cover page is shown—it's almost like reading from the printed book.

7. https://pragprog.com/titles/javacomp/source_code

Why Should I Learn This? There Are Static Code Analysis Tools…

You might be thinking that you already have static code analysis tools, such as Checkstyle,[8] FindBugs,[9] SpotBugs,[10] Error Prone,[11] and PMD,[12] that detect flaws in your code. So why should you keep reading this?

That's true for some, but it's not true for all the problems in this book. We'd love to see a tool that can *automatically* and *correctly* assess the concerns in your code. The existing tools are good, but they're not perfect. Think of them as rigorous, intolerant, nitpicky robots that were taught a set of rules and seek any violation. They can't understand that sometimes there are circumstances where you must violate a rule to improve the code. To use them successfully, they need to be fine-tuned, and that requires in-depth knowledge about code quality and the project at hand.[13] But even then, they can never beat an experienced programmer, although they can help so that some issues aren't missed.

That's good news, because it means that as an experienced programmer, your skills will be in high demand. Tools can check for a lot of things, but they often lack a detailed explanation for why something was detected. And they rarely show you how to solve a specific issue. Oftentimes, the solution to an issue isn't apparent from the warning of a detection tool, and it's rare that you can correct the code automatically. You have to do this manually, and that process is error prone—especially if you're still learning Java. As a new programmer, the tools won't guide you to a sensible correction. That's where this book can help you. We'll raise your awareness for common programming errors that many people make when learning Java.

How to Read This Book

If you're in your first year of Java, we suggest you read the book from start to finish. It starts with rather basic comparisons first, but then we'll advance to more challenging topics, like testing, object-oriented design, and functional programming in Java. If you're more experienced, you can probably skip the first two chapters and dig right into the more specialized topics.

8. http://checkstyle.sourceforge.net/
9. http://findbugs.sourceforge.net/
10. https://github.com/spotbugs/spotbugs
11. http://errorprone.info/
12. https://pmd.github.io/
13. Keep in mind: A fool with a tool is still a fool!

Here's a brief outline:

- In Chapter 1, *Start Cleaning Up*, on page ? we'll give you general advice on how to write correct code that's readable and understandable. We'll touch on a lot of things related to the basic Java syntax, such as conditions and braces.
- In Chapter 2, *Level Up Your Code Style*, on page ? we'll discuss a few more advanced coding concepts and problems, such as iteration, formatting, and using the Java API.
- Chapter 3, *Use Comments Wisely*, on page ? is about documenting code well. We'll give you some advice on how to write comments and when you should get rid of them.
- In Chapter 4, *Name Things Right*, on page ? we'll explain how you assign proper and concise names to code elements in Java that other programmers will easily understand.
- Chapter 5, *Prepare for Things Going Wrong*, on page ? will make you aware of how you should handle exceptions in Java.
- Chapter 6, *Assert Things Going Right*, on page ? gives you advice on how to write good unit tests with JUnit5.
- Chapter 7, *Design Your Objects*, on page ? outlines object-oriented design principles.
- Chapter 8, *Let Your Data Flow*, on page ? focuses on functional programming in Java using lambda expressions.
- Finally, Chapter 9, *Prepare for the Real World*, on page ? directs you to tons of material on building, releasing, and maintaining your software in the real world.

So let's move on and have fun!

Online Resources

This book has its own web page at www.pragprog.com.[14] There, you can get in touch with us in the discussion forum,[15] post errata,[16] or download the source code[17] of this book.

14. https://pragprog.com/titles/javacomp
15. http://forums.pragprog.com/forums/javacomp
16. http://pragprog.com/titles/javacomp/errata
17. https://pragprog.com/titles/javacomp/source_code

Get Ready for Your Mission to Mars

As a beginner or intermediate programmer, getting a sense for good and clean code that others will recognize as such might appear to you as something far, far away, a goal almost unreachable—a long journey, full of unexpected problems and unknown experiences. It's a bit like traveling through space to a destination that humanity hasn't reached yet: a journey to Mars! That was sort of how we felt when we started to learn programming many years ago.

We've written the code in this book with the theme of a Mars mission, so all the code examples have something spacey about them. But the real reason is that we don't want to distract you from the problem in a code snippet with the setting that the code is in. We used the astronaut theme to give you a better read, avoid meaningless variable names like `x` and `y`, or the umpteenth inheritance hierarchy from animals to mammals to cats. (We have nothing against cats—we just think they're overused in teaching material.)

Our comparisons are based on code and coding problems that we've found that a beginner or intermediate programmer often runs into when she's learning to program. We've extracted the underlying problems and put them into the domain of space travel so it's easier to imagine the context around them.

We can't guarantee that you'll become an astronaut after reading this book, but we're very confident that you'll be a better programmer. So fasten your seatbelt, and onward into (programmer) space!