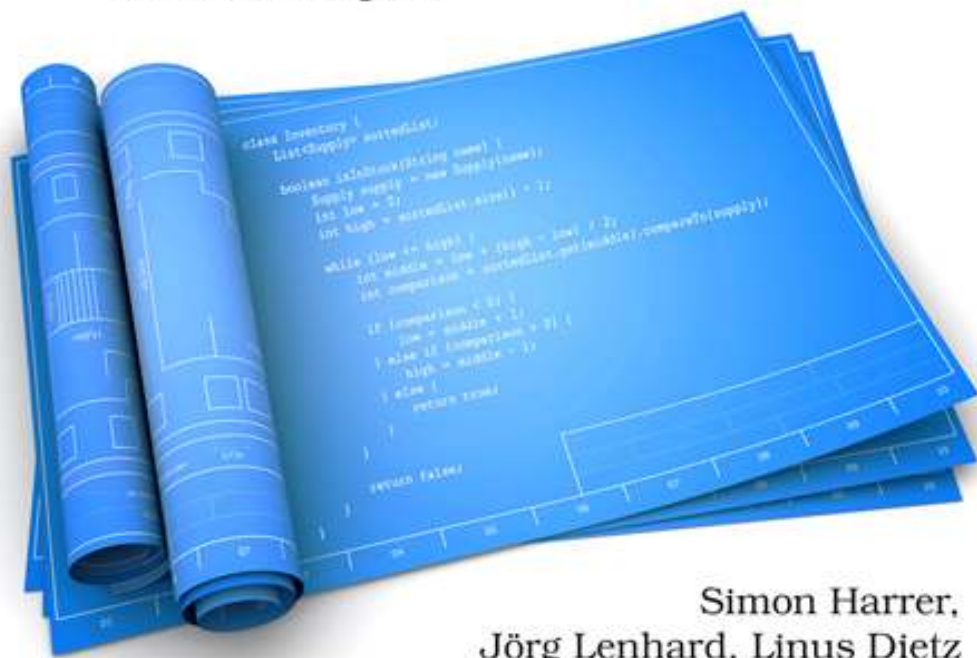


The
Pragmatic
Programmers

Java by Comparison

Become a Java Craftsman
in 70 Examples



Simon Harrer,
Jörg Lenhard, Linus Dietz
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

20.03.2023

JavaLand

KATA

Dr. **Simon** Harrer
Falk Sippach
Sebastian Rose

INNOQ

Agenda

- 1. Einführung Kata (15 min)**
- 2. Java by Comparison Kata (120 min)**
- 3. Retro (15 min)**

Actionable Name

Favor Format Over Concatenation

Bad Code

```
1 class SearchResult {
2     Path file;
3     int lineNumber, startIndex;
4     String token;
5
6     void print() {
7         System.out.println(file + ":[\" + lineNumber + \";\"
8             + startIndex + \"-\" + (startIndex +
9             token.length() - 1) + \"]> \"
10            + token + System.lineSeparator());
11     };
12 }
```

The problem with this code is that it is *very* hard to see how the output will look like.

Because there are so many plusses, quotes and spaces that obfuscate the code it takes longer to read and the chance of making mistakes is higher. Additionally, performing inline computations like summing ints is not really acceptable any more as the plus operator has different semantics in the very same line what makes it ever harder to read for us.

Problem Description

```
1 class SearchResult {
2     Path file;
3     int lineNumber, startIndex;
4     String token;
5
6     void print() {
7         System.out.printf(\"%s:[%d;%d-%d]> %s\\n\",
8             file,
9             lineNumber,
10            startIndex,
11            startIndex + token.length() - 1,
12            token);
13     };
14 }
```

Better Code

By using `System.out.printf()` (or `String.format()` for working with Strings without printing them) we achieve enhanced separation of syntax (how the String is printed) and semantics (which data is used for printing). Furthermore, it is more acceptable to make inline computations, as we can format the code such that each parameter is in its own line.

On a side note it should be mentioned that this way of formatting Strings is very common in most imperative programming languages and works the very same way in C/C++ and other programming languages.

Solution Description

Favor Format Over Concatenation

```
class Mission {
    Logbook logbook;
    LocalDate start;

    void update(String author, String message) {
        LocalDate today = LocalDate.now();
        String month = String.valueOf(today.getMonthValue());
        String formattedMonth = month.length() < 2 ? "0" + month : month;
        String entry = author.toUpperCase() + ": [" + formattedMonth + "-" +
            today.getDayOfMonth() + "-" + today.getYear() + "](Day " +
            (ChronoUnit.DAYS.between(start, today) + 1) + ")> " +
            message + System.lineSeparator();
        logbook.write(entry);
    }
}
```

Understandability and readability don't just matter in your code—they're also important for the output your code produces.

If you have to build large strings, you can use format strings to make them more readable.

Here, we're revisiting the log book we already mentioned in [Avoid NullPointerException in Conditionals, on page 10](#). The problem is that it's quite hard to see what the output will actually look like. There are multiple String variables, some of them using Java's in-line notation.

The code isn't really complex. There's no deep nesting, no conditional branching, and even the names of variables and methods are meaningful. Despite all this, it's comparably hard to read. The number of plusses, quotes, and spaces obfuscates the actual processing going on (humans are generally better at understanding words in a natural language than combinations of symbols). This makes the code harder to read and increases the chance of making mistakes when writing it.

Additionally, performing in-line computations like summing int values adds to the confusion, as the + operator has different semantics when it comes to Strings and ints. Using it with different semantics in the very same line makes it even harder to maintain an overview of what's happening. Don't do this!

We can trim this down a lot. Take a look at the following code:

```
class Mission {
    Logbook logbook;
    LocalDate start;

    void update(String author, String message) {
        final LocalDate today = LocalDate.now();
        String entry = String.format("%S: [%tm-%<te-%<tY](Day %d)> %s%n",
            author, today,
            ChronoUnit.DAYS.between(start, today) + 1, message);
        logbook.write(entry);
    }
}
```

Format strings help to overcome this issue, and they're available in practically any contemporary programming language—not just in Java. The key lies in separating the layout of a String (*how* it is printed) from the data (*what* is being printed). Format strings define a coherent String in a single block using special placeholder characters, marked by %.

Format methods, such as String.format(), or System.out.printf(), accept data using placeholder characters in the order in which they're listed after the String.

In our example here, %S converts an object into an upper case String using its toString() method. The parameter author will be processed this way. The date variable today serves as data for %tm, the month, %te the day of the month, and %tY, the year. The additional character < makes sure that all three placeholders read the same input data. %d processes a decimal value and %s accepts a String. Finally, %n is the symbol for a line break.

The actual data is listed in a nicely formatted fashion after the String layout. This makes it a little more acceptable to perform in-line computations, as we can use one line for each parameter. Point taken that it's not so easy to see what "%S: [%tm-%<te-%<tY](Day %d)> %s%n" will print in the end. But it's a well-documented standard³ and actually a good alternative to the cluttered code from the problem—for large strings we recommend StringTemplate,⁴ a powerful template engine.

Consider documenting the formatted string with a few examples similar to [Document Using Examples, on page 48](#) so that the next developer reading your code doesn't have to look up how %S or any other special formatting syntax behaves to know what the resulting string will be.

1. Start Cleaning Up

2. Level Up Your Code Style

3. Use Comments Wisely

4. Name Things Right

5. Prepare for Things Going Wrong

6. Assert Things Going Right

7. Design Your Objects

8. Let Your Data Flow

9. Prepare for the Real World

JAVAPRO
JavaSPEKTRUM
Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen



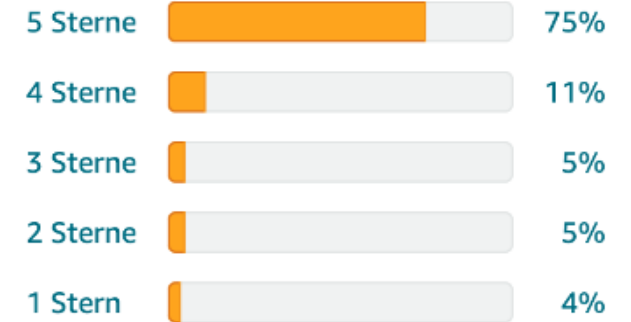
InfoQ
heise **Developer**



Kundenrezensionen

★★★★☆ 4,5 von 5

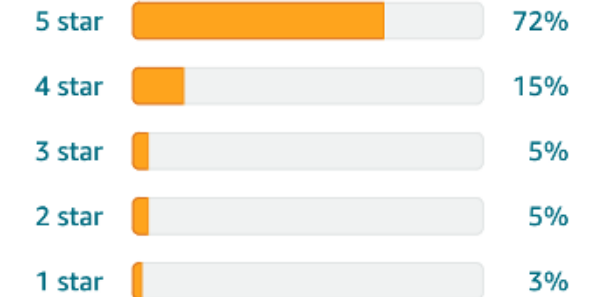
71 globale Bewertungen



Customer reviews

★★★★☆ 4.5 out of 5

54 global ratings



Java by Comparison Kata

<https://java.by-comparison.com/kata>

<https://github.com/javabycomparison/kata>





1
pair

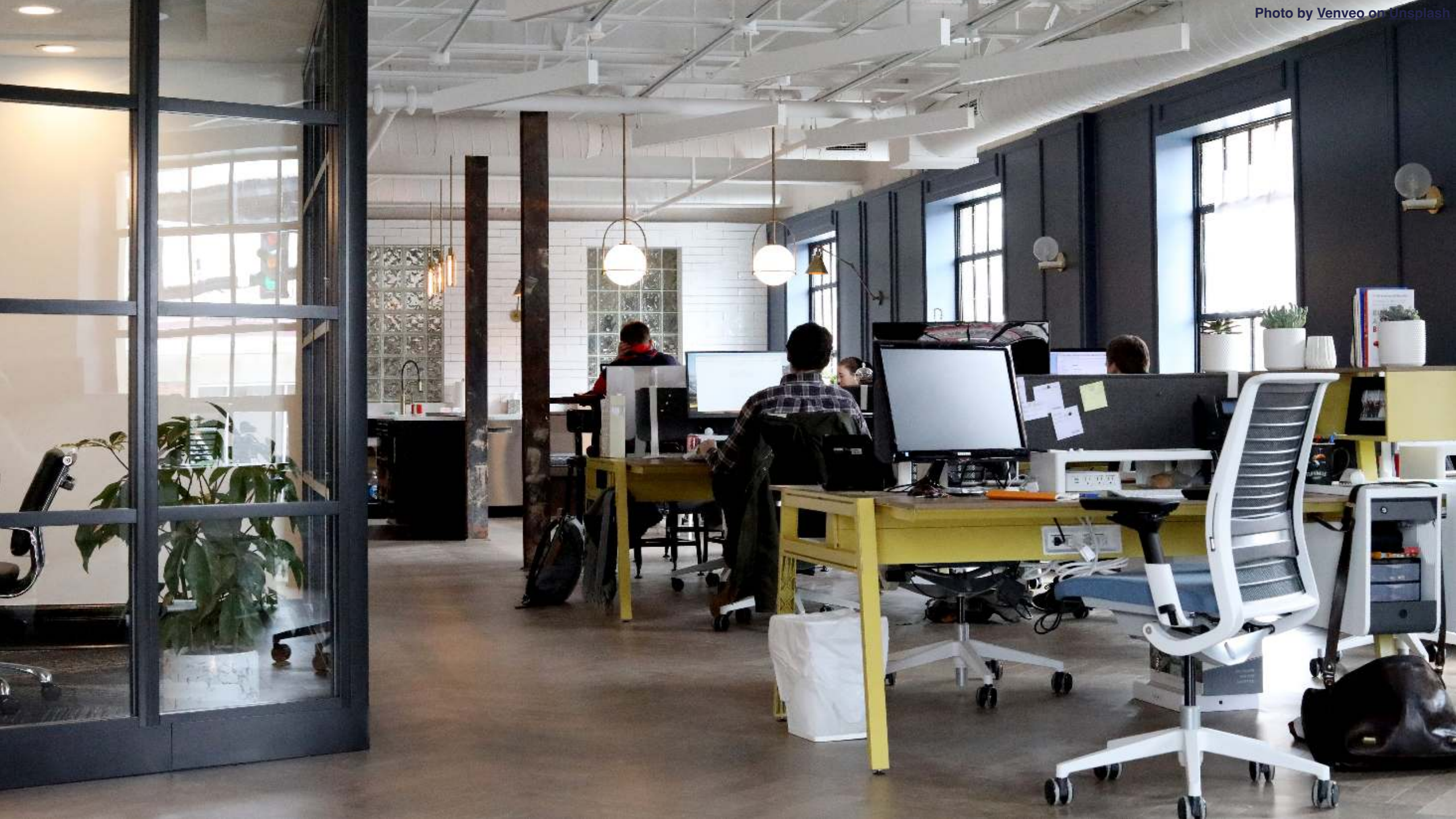


1
book



1
list

DOWNLOAD



Today is your first day at a new job.

Boss: Great to have you here.

You: Yes, I'm excited to dive into the world of software metrics.

Boss: Awesome. Let me show you the software we are currently making a ton of money with.

Boss points you to the code of this kata

Boss: It's calculating metrics for Java and even Python source code. We have literally thousands of customers that work with this release everyday. Thing is, our backlog is packed and we need you to pump out features quickly, otherwise, our customers will leave.

Boss: But, of course, you just started. So have a look at the code. You may clean it up so you can start those features.

Boss: Let's say, you have 2 hours? Should be fine, I think.

You: ...

So you start cleaning that legacy code up.

Code (Java Version 8):

521

lines of code

71

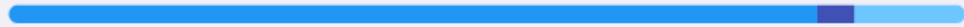
blank lines

11

files

24

comments



Tests:

127

lines of code

19

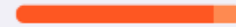
blank lines

3

files

84%

line coverage



README.md

```
# download kata
```

```
$ git clone https://github.com/javabycomparison/kata
```

```
$ cd kata
```

```
# execute tests
```

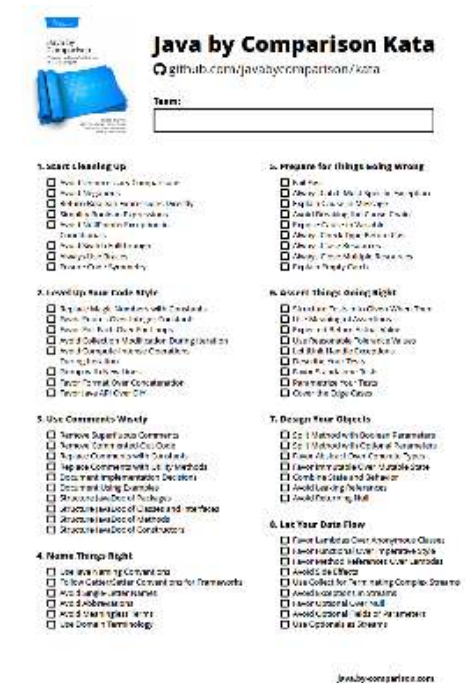
```
$ ./mvnw test
```

```
# run
```

```
$ ./mvnw exec:java -Dexec.args="src smry"
```

1. **Bildet** Pairs: 2 Personen, 1 Buch, 1 Laptop
2. **Importiert** das Kata in eure IDE
3. **Testet** das Projekt
4. **Verbessert** das Projekt

1. **Sucht** den für euch schlimmsten Code Smell für das es eine Comparison gibt
2. **Wendet** Comparison an
3. Tests **Grün?** **Hakt** Comparison ab
4. **REPEAT**



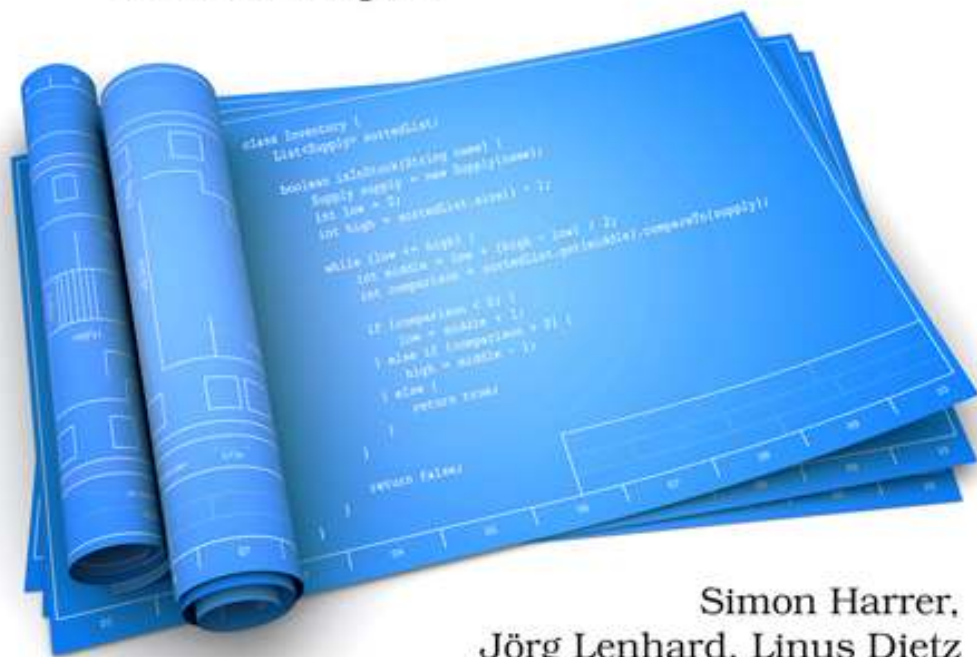
Ihr habt 2 Stunden Zeit. **Viel Spaß!**

Retro

The
Pragmatic
Programmers

Java by Comparison

Become a Java Craftsman
in 70 Examples



Danke fürs Teilnehmen!

Euch hat das Buch gefallen?

Wir würden uns über eine Bewertung
auf Amazon und GoodReads freuen!

Und Danke an INNOQ für das Buch-
Sponsoring

Dr. **Simon** Harrer
Falk Sippach
Sebastian Rose