23.10.2019

SWK Köln @ REWE Digital

Die Geschichte(n) von Java by Comparison

Dr. Simon Harrer — simonharrer

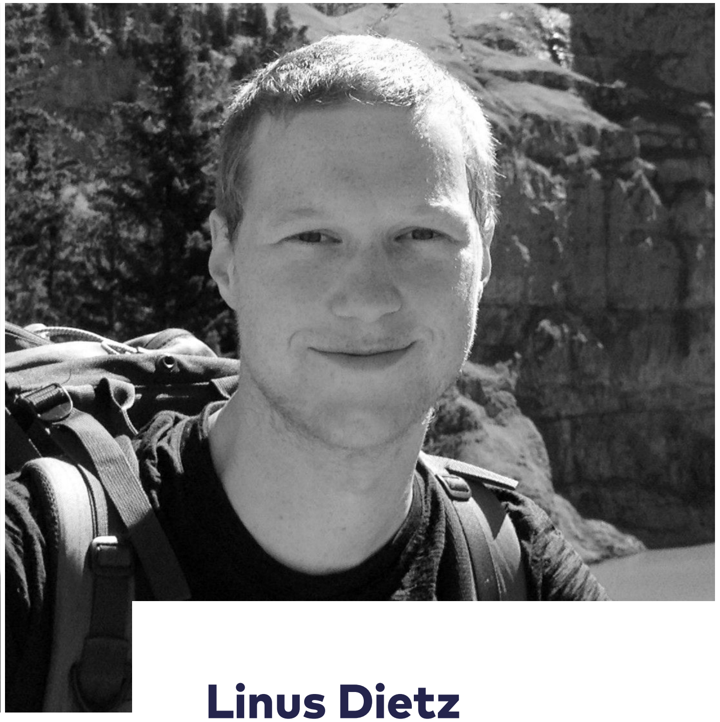
Linus Dietz — Lynyus

INNOQ

Technische Universität München

Dr. Simon Harrer
Senior Consultant
innoQ Deutschland GmbH

Simon entwickelt Software
in einem Remote Mob



Linus Dietz
Wissenschaftlicher Mitarbeiter & Doktorand
Technische Universität München

Linus forscht über
Empfehlungsdienste für Reisende

Agenda

1. Geschichte(n) des Buches
2. Comparison Jeopardy

2011


Fortgeschrittene Java Programmierung
Programmierung komplexer und interagierender Systeme

Typische Lehrveranstaltungen

Theoretische Vorlesung

Plenarübung

Hausaufgaben

Schriftliche Klausur

Aufgabenstellung

Lösung

Note

Wie bringt man anderen bei **sauberen Code** zu schreiben?

1. Code Reviews interaktiv miterleben
2. Code Reviews schriftlich bekommen
3. Code Reviews selbst durchführen

- Wurde der Code ausreichend dokumentiert?
Ja

- Code Conventions eingehalten?
Nein. Ihr solltet Klassen- und VariablenNamen in CamelCase schreiben. Im Fall von Findspots ist das nicht der Fall. Klassennamen sollten auch generell Nomen sein. Also in diesem Fall `SpotFinder`.

##Code

- Ressourcen korrekt geschlossen?
Ja

- Exceptions korrekt behandelt (inkl. weiterwerfen mit Cause)?
Nein. Wenn der Validator fehlschlaegt darf das Programm nicht mit `System.exit()` beendet werden! Stattdessen soll eine `TokenfinderException` mit einer Nachricht geworfen werden. Fehler sollten frueh geworfen werden. Das bedeutet, dass sobald im `Validator` ein Check fehlschlaegt, sofort eine passende Exception geworfen werden sollte und der Rest nicht mehr ueberprueft werden muss.
Ihr verwendet die `javax.xml.bind.ValidationException`. Das ist schlechte Praxix, da sich diese auf Fehler im Kontext von XMLvalidierung bezieht. Hier haettet ihr einfach die `TokenFinderException` verwenden koennen. Weiterhin habt ihr im `Validator` die Methoden von `java.nio` noch einmal in eigene gleichnamigen Methoden gesteckt nur um jeweils eine `javax.xml.ValidationException` werfen zu koennen. Das ist sehr viel unnoetiger und falscher Code, weil diese Methoden ja ihre eigenen Exceptiontypen (meist `IOException`) werfen, die ihr dann jeweils fangen sollt.
Was auch unschoen ist, dass ihr in `SimpleTokenFinder.search()`#55-74 einen `try-catch-block` habt, der eine `TokenFinderException faengt`. Diese sollte einfach durch throws nach oben weitergegeben werden anstatt die Nachricht auszugeben.

- Wird die Java NIO Bibliothek verwendet?
Ja

- Wurde die Aufgabe sinnvoll (bspw. in mehrere Klassen) strukturiert?
Ja, alledings haben eure Methoden teilweise noch viele Einrueckungen. Da solltet ihr noch etwas mehr

2016

Sammlung typischer Fehler

Common Stuff

- Warnings still in the Code
- FIXME and TODO in the Code
- Commented out code
- Do not use System.exit()

Issues

- Always use braces if(CONDITON) { STATEMENT; }, never if(condition) STATEMENT; (see http://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/)
- Do never catch Exception and NullPointerException, except you have an extremely good reason for it
- Instead of if (CONDITION) { return true; } else { return false; } simply write return CONDITION;
- Avoid unnecessary documentation such as // ... return true ... to document return true;. Focus on the important things instead.
- Do not document your name in the classes to avoid marking bias
- Use final as much as possible for your fields - this simplifies your code tremendously

Erstmals eine Gruppe mit 80 von 80 Punkten...

Übergabe in kompetente Hände

https://www.flickr.com/photos/prayitnophotography/6883792695

Preis für Gute Lehre

In 6 Jahren ...

... 500+ Assignments korrigiert

... 1.000.000+ Java Codezeilen gelesen

... 1000de der gleichen Fehlern immer wieder angemerkt

... gelernt andere zu überzeugen "Clean Code" zu schreiben

... neue Herausforderungen gesucht.

War's das jetzt?

Können wir daraus nicht etwas cooles machen?

Common Stuff

- Warnings still in the Code
- FIXME and TODO in the Code
- Commented out code
- Do not use System.exit()

Issues

- Always use braces if(CONDITON) { STATEMENT; }, never if(condition) STATEMENT; (see http://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/)
- Do never catch Exception and NullPointerException, except you have an extremely good reason for it
- Instead of if (CONDITION) { return true; } else { return false; } simply write return CONDITION;
- Avoid unnecessary documentation such as // ... return true ... to document return true;. Focus on the important things instead.
- Do not document your name in the classes to avoid marking bias
- Use final as much as possible for your fields - this simplifies your code tremendously

Na klar!

Actionable Name

Favor Format Over Concatenation

Bad Code

```java
class SearchResult {
    Path file;
    int lineNumber, startIndex;
    String token;

    void print() {
        System.out.println(file + ":[" + lineNumber + ";"
                + startIndex + "-" + (startIndex +
                token.length() - 1) + "]> "
                + token + System.lineSeparator());
    };
}
```

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 `int`s 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

Better Code

```java
class SearchResult {
    Path file;
    int lineNumber, startIndex;
    String token;

    void print() {
        System.out.printf("%s:[%d;%d-%d]> %s%n",
                          file,
                          lineNumber,
                          startIndex,
                          startIndex + token.length() - 1,
                          token);
    };
}
```

By using `System.out.printf()` (or `String.format()` for working with `String`s 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.

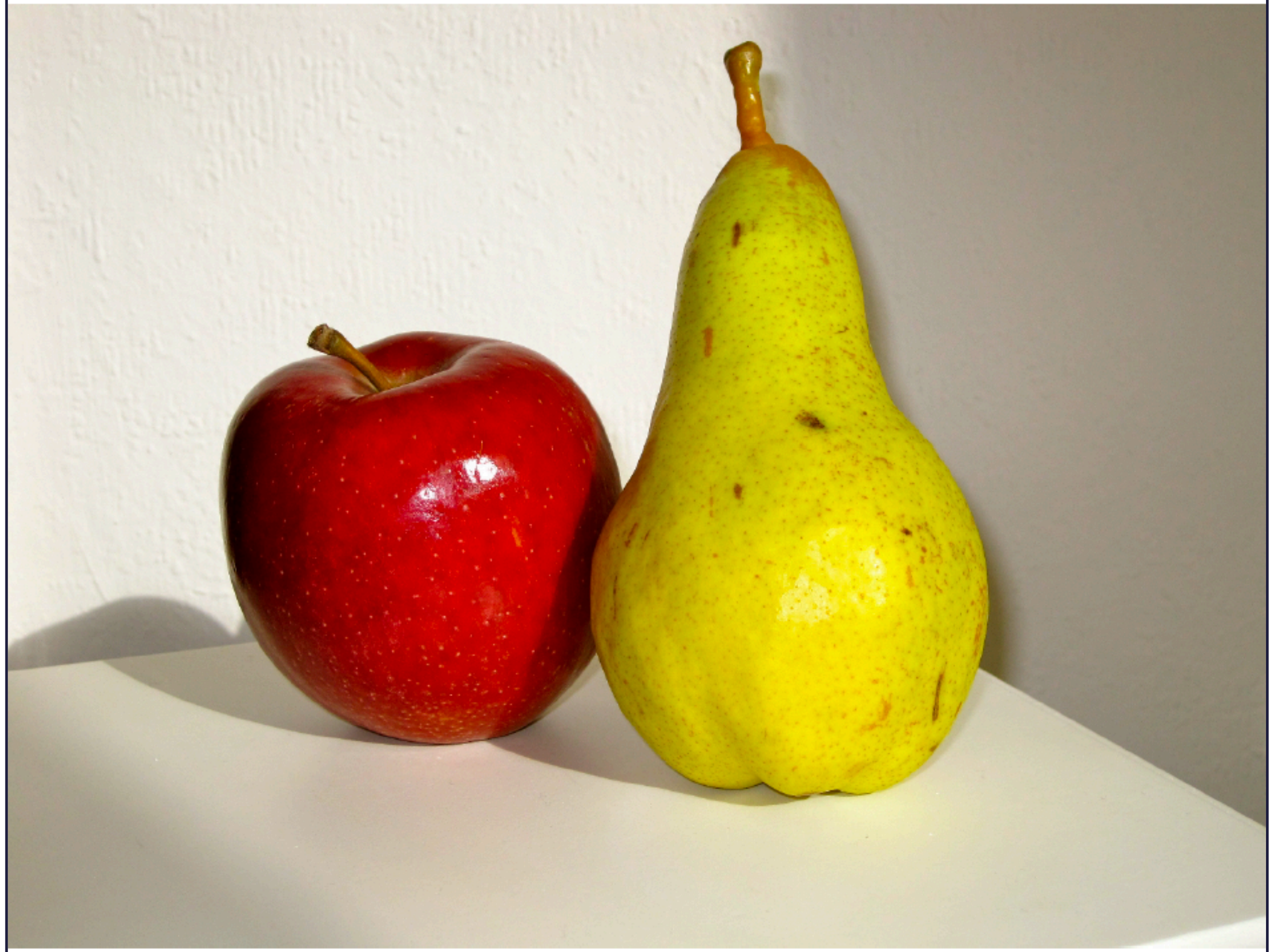
On a side note it should be mentioned that this way of formatting `String`s is very common in most imperative programming languages and works the very same way in `C/C++` and other programming languages.

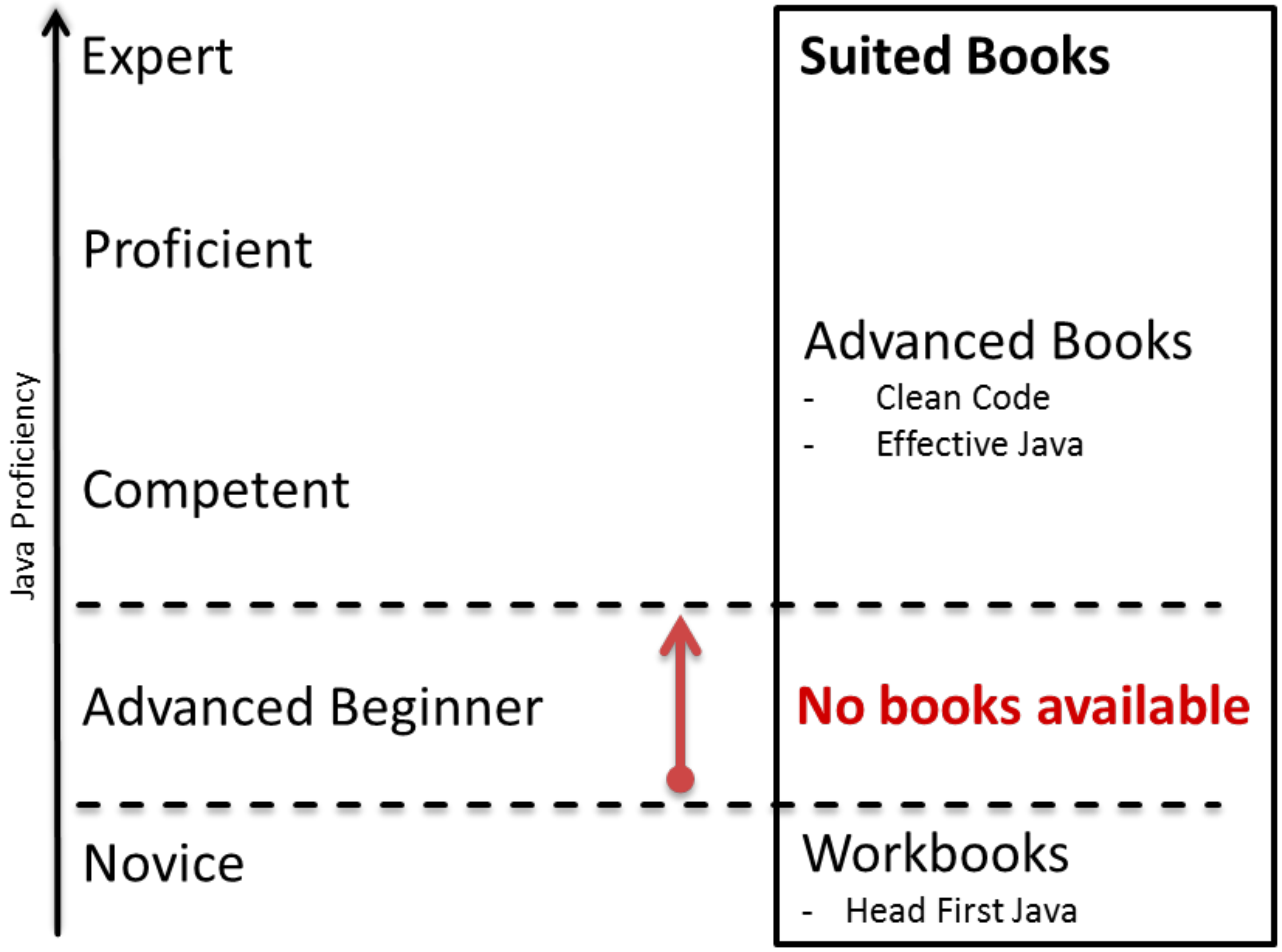
Solution Description

Java by Comparison

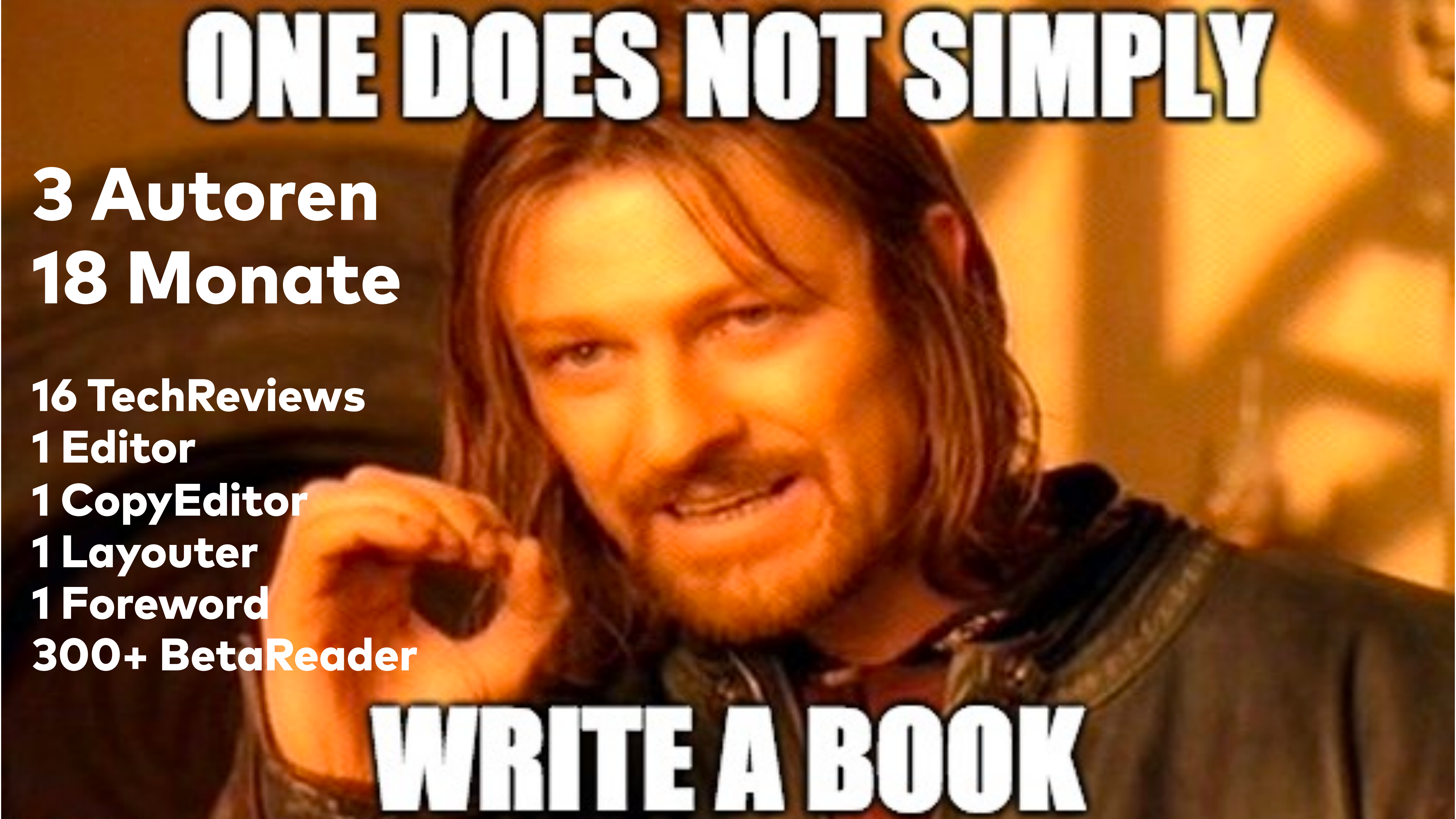
A Software Craftsman's First Book

Simon Harrer and Linus Dietz




Java Proficiency
Expert
Proficient
Competent
Advanced Beginner
Novice
Suited Books
Advanced Books
- Clean Code
- Effective Java
No books available
Workbooks
- Head First Java


Drei Tage später der Vertrag!!1!


ONE DOES NOT SIMPLY
3 Autoren
18 Monate
16 TechReviews
1 Editor
1 CopyEditor
1 Layouter
1 Foreword
300+ BetaReader
WRITE A BOOK

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 standard[3] 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

Java magazine
By and for the Java community

JavaSPEKTRUM
Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

@Autoweird fm

Anwendungsentwickler-Podcast

Software. Architektur. Systeme.
INNOQ

InfoQ

heise Developer

TUM

Otto-Friedrich-Universität Bamberg

DAWSON COLLEGE

BMW

Kundenrezensionen

4,8 von 5

30 Sternebewertungen

5 Sterne	90%
4 Sterne	5%
3 Sterne	0%
2 Sterne	5%
1 Stern	0%

Customer reviews

5 out of 5

12 customer ratings

5 star	100%
4 star	0%
3 star	0%
2 star	0%
1 star	0%

Begeisterte Rezensenten, zufriedene Leser

Nach einer kurzen Pause geht es weiter mit

Comparison Jeopardy!

Let's Play!

Comparison Jeopardy

Actionable Name

Bad Code

Problem Description

Better Code

Solution Description

Spielregeln

- Wir zeigen euch einen Codevergleich
- Findet einen Namen der das "Refactoring" beschreibt
  - Name muss mit einem Verb anfangen

- Beispiele
  - Simplify Boolean Expression
  - Favor Format Over Concatenation

Danke fürs Teilnehmen!

Das schönste Geschenk, dass ihr einem Autor machen könnt, ist eine Amazon Rezension zu dessen Buch zu schreiben.

Simon Harrer simonharrer
Linus Dietz Lynyus

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

https://www.remotemobprogramming.org/