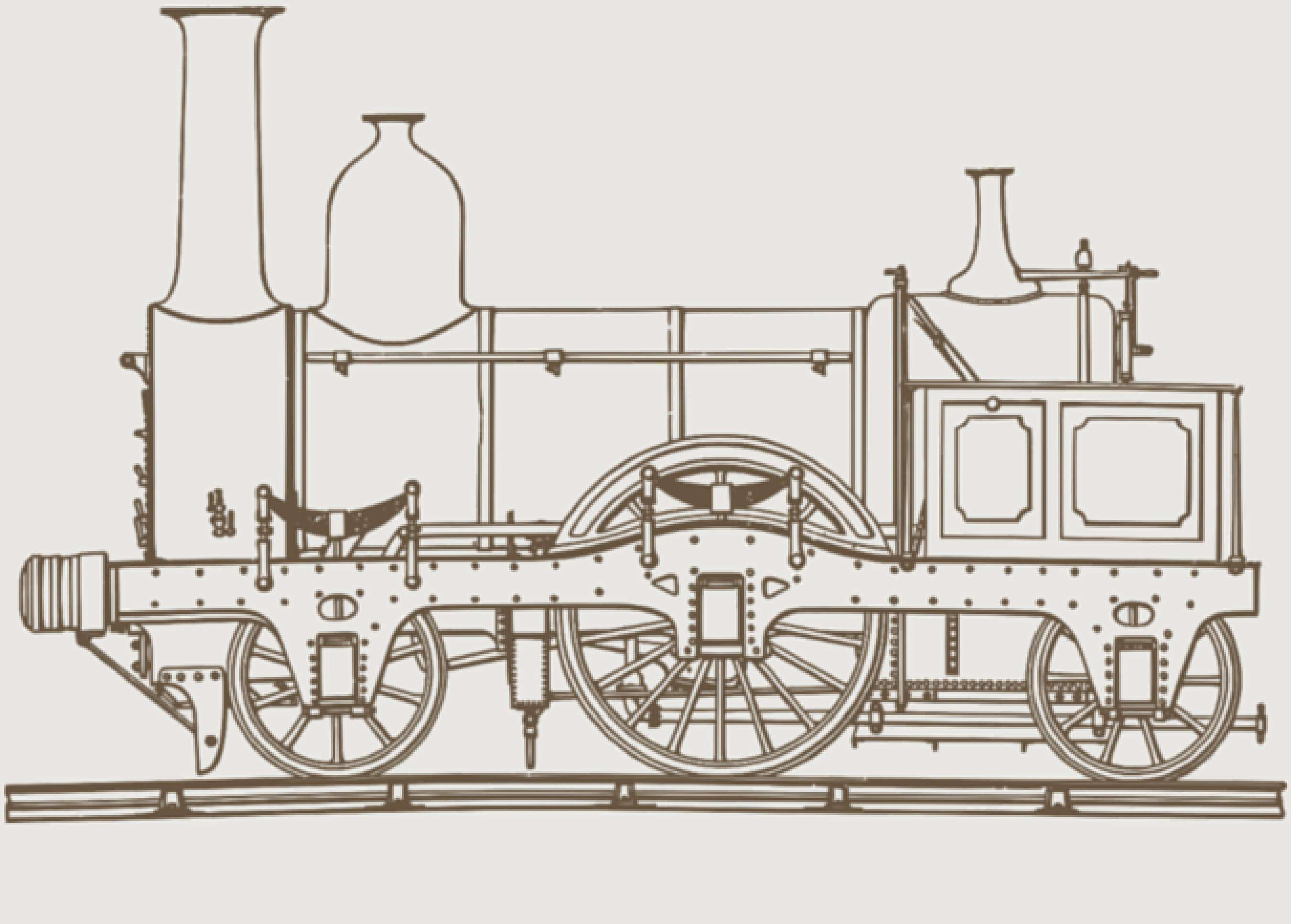
Beautiful Java



Sam Atkinson

Beautiful Java

Sam Atkinson

This book is for sale at http://leanpub.com/beautifuljava

This version was published on 2016-09-17

Contents

Introduction

Welcome to Beautiful Java. Thanks for buying the book, particularly if you've bought it in preview on leanpub. Your encouragement makes the writing process so much more enjoyable, and also allows you to shape the book. If you've any suggestions or corrections then just send me an email via book@samatkinson.com[1]. I promise I'll respond, and possibly send a reward :).

As you may or may not have heard of me, let me take the time to briefly introduce myself. My name's Sam, and I'm a Java lead who's been kicking around the financial services industry for a fair few years now. I also run a website called Core Java Interview Questions[2], which was started out of a frustration with the lack of decent materials online for prepping for "traditional" Java interviews. If you're ever in London buy me a drink and I'll explain why those interviews are terrible, but that's by the by.

I'm an incredibly passionate developer. I believe strongly that software development is primarily a craft and secondarily a science. In programming there are infinite ways to skin a cat. The infinite solutions can all produce the same outputs for a given set of inputs, but they can all look completely different. Some of these solutions will be good, and some will be bad. To the end user they don't know or care, but to the developer (and future maintainers) the quality of the solution matters.

If you don't believe me about this variation, check out my onrunning series at Dzone.com on Java Code Challenges. Every 2 weeks or so I post a challenge from Reddit's daily programmer, and then a week later analyze the responses not based on being functionally correct but on the quality of the code. The most popular post has over 40 solutions (read it here[3]), ranging from one-class one-method through to large domain design, and including no test, test after and TDD solutions.

Even the simplest challenge can be done in countless different ways. The biggest problem in our industry is there is no objective way to define what good looks like in code. There are certain facets which we can all agree are desirable, including but not limited to readability and maintainability (more on that later), the problem is there is very little in the way of empircal evidence to link certain elements of code style to these attributes. Can we say for certain Spring applications are easier or harder to maintain over the long term? Does thoroughly documenting every method with comments result in an easier job for the maintainer or a hellish world of out-of-date and misplaced clues?

What we do have though is anecdotal evidence, heavily influenced by opinion, and that is what you will find in this book. This is my take on what beautiful Java code looks like. These pages are full of my opinions, derived over the last decade from working with a variety of brilliant and not-so-brilliant developers on greenfield and legacy systems. I can safely say that systems I've seen implemented which use these techniques are what I like to call "malleable"- easy to understand,

[1]book@samatkinson.com

[2]http://www.corejavainterviewquestions.com

[3]https://dzone.com/articles/java-code-challenge-chemical-symbol-naming-part-on

easy to change, and importantly safe to change. Some of the points in here will be controversial, and will make you baulk at first read. I know when I was introduced to some of these ideas they went against everything I had been taught and I reacted with fury, having ferocious arguements with the developers in question. I was however coerced into trying them out and having been preeching them ever since. Please keep an open mind, and please experiment. There is a relatively low cost to trying out new ideas such as the ones from this book- even you think they smell, give them a go and see if you can reap the benefits. Some of them you may still disagree with, but some of them you won't.

And finally, if you want to discuss anything, get some clarification, or even sit down and pair for a bit, then drop me an email. I'm friendly, honest.

Sam.

Part One: Clean Code

Clean code trumps all other requirements and eager optimisation is the enemy

I am officially declaring war on the eager optimisation crowd. You may be part of it and not even know it. Let's take this comment from a DZone article (https://dzone.com/articles/always-start-with-eager-initialisation) I posted a while back:

"But if you already know something is slow and how to write it correctly then that is not premature optimisation, that is smart coding. "

See, this guy thinks he isn't part of the eager optimisation crowd. Because he doesn't do it, unless he *knows* that a certain method or algo is "slow" and needs to be written performantly. Then he'll write it performantly from the off.

This is eager optimisation damnit.

This is what results in codebases with outrageously complicated code that can be replaced with a for-loop. Because you "know better". I'm here to tell you you don't.

Measure everything

Let's go back to the start. When you're writing a piece of code, whatever it is, you will have performance concerns. Perhaps it's part of a batch job that runs overnight, in which case it can probably go as slow as you want. Counterpoint, it could be a dependency as part of a super low latency system. Or maybe it's returning a response to a user on a webpage.

These are three very differing requirements. The important thing is that for each of them you can attach some sort of numbers to it: * Total Batch needs to complete in under 7 hours * Super Low Latency system needs a response in under 1ms * Webpage needs a response in under 300ms

The beauty of this is that you have an actual number to target. This means you can empirically prove if you're doing the job or not. When people start waffling to me about needing a complex algorithm for a piece of work to go fast enough, I can simply say one thing.

Prove it.

There are genuinely some times when you need your outrageously complex algorithm from a textbook. But these are rare. Normally a bunch of for-loops and/or hashmaps will do the job perfectly well. The inbuilt Java algorithms for searching aren't bad.

I’m a massive proponent of TDD. TDD dictates to implement the simplest, most stupid thing to get the test to pass and then go from there. This is absolutely how you should approach any potential complexity of your code.

First, write the simplest, cleanest thing you can to make it functional. Cleanliness is the most important goal in code (along with ensuring it works). Clean, well tested code results in long term maintainable systems. Use the built in Java libraries where possible, as the consumer of your code will understand these at a minimum, and as I mentioned before, they’re usually pretty good and certainly well tested.

Now, test the performance. Actually measure it against your target. In my experience the majority of the time you’ll hit your measure and can move on with your life.

Now I can almost guarantee the outrage in the comments. And whilst I look forward to reading them, I encourage the rest of you to give this a go. Don’t optimise anything at the beginning of your coding. Test. Improve.

The other benefit is that your tests will be even nicer. By using simple implementations it will help you generate a clean API for your consumers (whether that’s yourself or a third party). If you do need to improve your code (and that’s totally ok!) you’ve got a full suite of working tests you can ensure you conform to to guide you through the process.

The biggest culprit for me is in search. I’m not talking about massive datasets (which would probably be in an appropriate dataset) but instead in memory- things like lists and maps that are stored intra-application. The amount of times I’ve seen people implement their own overly complex caching algorithms on top which are hard to understand and often don’t work makes me cry. Usually they can be ripped out and replaced with a for loop with no detriment (and often with performance gain).

If you really do need a performance boost then never forget there are people out there who are smarter than you who have implemented this stuff and have been tested by the community it. It’ll be easier to understand and a lot more likely to work. Google Collections are your friend. You don’t need to reinvent the wheel in an attempt to be more performant. No one will thank you for it.

Using Java 8? Plese avoid functional vomit

As the old adage goes, with great power comes great responsibility. The advancements of Java 8 have provided us that power. It's brilliant! We have true functional syntax in Java which allows us to write beautiful code in a much more terse fashion. We can peform functions in a parallel fashion using streams. We Java developers have finally entered the 21st century.

The thing is, most developers suck. Obviously this is a heavily loaded statement, and fundamentally what I think is good code is different to what you think good code is, but I think everyone can acknowledge there is a significant portion of the developer community in any language (but particularly Java) that just doesn't know how to write good code. The introduction of the functional programming paradigm into Java 8 is in many cases like giving a Ferrari to someone who's only just gotten their training wheels off (more cheesy analogies to follow).

I can't even claim to be the first person to have had this fear. If you wanted to perform functional programming pre-Java 8 the only real option was Google Guava, a wonderful library that did it's best to make up for some of Java's greatest shortfalls via the medium of functional programming.

On the Guave wiki they have this beautiful line which has always stuck with me:

"Excessive use of Guava's functional programming idioms can lead to verbose, confusing, unreadable and inefficient code. .. when you go to preposterous lengths to make your code "a one-line", the Guava team weeps."

In my experience and much to my chagrin, few people headed this advice. This is great advice and absolutely applies to Java 8. One of the greatest features of Java 8 is that it can reduce the verbosity; anonymous functions are a great example of this.

```
return findBy(new Predicate() {
        public boolean matches(Movie movie) {
            return movie.title().toUpperCase().contains(partialTitle.toUpperCase\
());
        }
    });

    …becomes…

    return findBy(movie ->   movie.title().toUpperCase().contains(partialTitle.t\
oUpperCase()));
```

That's great. The code is now clearer and easier to understand. What a lot of developers don't seem to understand is that the goal of functional programming is not just to reduce verbosity. Verbosity is not the problem.

Yes, it takes more physical characters in Java than in other languages to achieve things. But honestly, it doesn't matter. We have great IDE's like IntelliJ (and sure, Eclipse, but IntelliJ is better. I won't even mention Netbeans) which do all the hard work for you. I can go just as fast in Java with my IDE as you can in any other language.

Which is why I'm going to say something controversial and against the grain and is going to make everyone angry in the comments. Java's verbosity is, in most cases, a good thing.

Our goal when programming is not to produce the least amount of code possible, but to produce performant systems that are easy to maintain. An easy to maintain system makes it painfully obvious what all the code does. This can, and often does, mean intentionally writing extra code to make it really really clear what's going on.

if(barrier.value() > LIMIT && barrier.value() > 0){

if(barrierHasPositiveLimitBreach()){

Oh no! Extra code in an extracted method! But it's a lot easier for the person reading the code to understand what's going on, and it's much less likely they're going to destroy the code by misunderstanding what it's doing.

Extra code is not the enemy

By this virtue, do not use functional program as some brilliant way to turn all your code into one liners that are impossible to understand. If that's your jam, try something like JS1K or one use Perl. Everyone else hates your code.

Let's take this example from "10 Java one liners to impress your friends". https://github.com/aruld/java-oneliners/wiki on how to print the song Happy Birthday.

range(1, 5).boxed().map(i -> { out.print("Happy Birthday "); if (i == 3) return "dear NAME"; else return "to You"; }).forEach(out::println);

For me, this code would be much, much easier in the Java 7 style:

```
  for(int i = 1; i <=5; i++){
        System.out.println("Happy Birthday " + (i == 3 ? "dear NAME" : "to you")\
);
    }
```

Write beautiful code. Write easy to understand code. Do not make me and your co-workers weep. Use Java 8 functionality and one liners for good, not just to save characters. Characters are cheap.