Unit 12: Writing good Java code

Strive to write clean, easily maintainable Java code

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Follow best practices for class sizes, method sizes, and method names
- Understand the importance of refactoring
- Gain consistency in coding style and use of comments
- Use built-in logging

Best coding practices

You're now about halfway through this learning path. You have enough Java syntax under your belt to write basic Java programs. Before you continue on to more-advanced topics, this is a good moment to learn a few best coding practices. Read on for some essential pointers that can help you write cleaner, more maintainable Java code.

Keep classes small

So far you've created a few classes. After generating getter/setter pairs for even the small number (by the standards of a real-world Java class) of attributes, the `Person` class has 150 lines of code. At that size, `Person` is a small class. It's not uncommon (and it's unfortunate) to see classes with 50 or 100 methods and a thousand lines or more of source. Some classes might be that large out of

necessity, but most likely they need to be *refactored*. Refactoring is changing the design of existing code without changing its results. I recommend that you follow this best practice.

In general, a class represents a conceptual entity in your application, and a class's size should reflect only the functionality to do whatever that entity needs to do. Keep your classes tightly focused to do a small number of things and do them well.

Keep only the methods that you need. If you need several helper methods that do essentially the same thing but take different parameters (such as the `printAudit()` method), that's a fine choice. But be sure to limit the list of methods to what you need, and no more.

Name methods carefully

A good coding pattern when it comes to method names is the *intention-revealing* method-names pattern. This pattern is easiest to understand with a simple example. Which of the following method names is easier to decipher at a glance?

- `a()`
- `computeInterest()`

The answer should be obvious, yet for some reason, programmers have a tendency to give methods (and variables, for that matter) small, abbreviated names. Certainly, a ridiculously long name can be inconvenient, but a name that conveys what a method does needn't be ridiculously long. Six months after you write a bunch of code, you might not remember what you meant to do with a method called `compInt()`, but it's obvious that a method called `computeInterest()`, well, probably computes interest.

Keep methods small

Small methods are as preferable as small classes, for similar reasons. One idiom I try to follow is to keep the size of a method to **one page** as I look at it on my screen. This practice makes my application classes more maintainable.

In the footsteps of Fowler
The best book in the industry (in my opinion, and I'm not alone) is *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al. This book is even fun to read. The authors talk about "code smells" that beg for refactoring, and they go into great detail about the various techniques for fixing them.

If a method grows beyond one page, I refactor it. Eclipse has a wonderful set of refactoring tools. Usually, a long method contains subgroups of functionality bunched together. Take this functionality and move it to another method (naming it accordingly) and pass in parameters as needed.

Limit each method to a single job. I've found that a method doing only one thing well doesn't usually take more than about 30 lines of code.

Refactoring and the ability to write test-first code are the most important skills for new programmers to learn. If everybody were good at both, it would revolutionize the industry. If you

become good at both, you will ultimately produce cleaner code and more-functional applications than many of your peers.

Use comments

Please, use comments. The people who follow along behind you (or even you, yourself, six months down the road) will thank you. You might have heard the old adage *Well-written code is self-documenting, so who needs comments?* I'll give you two reasons why I believe this adage is false:

- Most code is not well written.
- Try as we might, our code probably isn't as well written as we'd like to think.

So, comment your code. Period.

Use a consistent style

Coding style is a matter of personal preference, but I advise you to use standard Java syntax for braces:

```
public static void main(String[] args) {
}
```

Don't use this style:

```
public static void main(String[] args)
{
}
```

Or this one:

```
public static void main(String[] args)
  {
  }
```

Why? Well, it's standard, so most code you run across (as in, code you didn't write but might be paid to maintain) will most likely be written that way. Eclipse **does** allow you to define code styles and format your code any way you like. But, being new to Java, you probably don't have a style yet. So I suggest you adopt the Java standard from the start.

Use built-in logging

Before Java 1.4 introduced built-in logging, the canonical way to find out what your program was doing was to make a system call like this one:

```
public void someMethod() {
  // Do some stuff...
  // Now tell all about it
  System.out.println("Telling you all about it:");
  // Etc...
}
```

The Java language's built-in logging facility (refer back to Unit 5: Your first Java class) is a better alternative. I **never** use `System.out.println()` in my code, and I suggest you don't use it either. Another alternative is the commonly used log4j replacement library, part of the Apache umbrella project.

For further exploration

Java - Basic Syntax

Google Java Style Guide

Java Coding Guidelines

Speaking the Java language without an accent

Refactoring: Improving the Design of Existing Code by Martin Fowler et al.

Effective Java by Joshua Bloch

IBM Code: Java journeys

Previous: Archiving Java code
Next: Next steps with objects