COURSEWARE

Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

Maven

Testing (Foundation)

Java Intermediate

- Optionals
- JDBC CRUD
- Exceptions
- SOLID Principles
- Single Responsibility
- Open/Closed
- Liskov Substituiton
- Interface Segregation
- Dependency Inversion
- Best Practice
- Design Patterns
- Creational Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns
- Collection & Map
- HashSets
- HashMaps
- Enums
- Logging
- Generics
- Lambda Expressions
- Streams
- Complexity
- Input and Output
- Local Type Inference

HTML

Best Practice

Contents

- Prerequisites
- Overview
- Style Guides
  - Formatting
    - Brackets
    - Line-breaks and whitespace
  - Specific Constructs
  - Comments
- Java Specification Recommendations
- Conventions
- Common Sense: The Three Virtues of Programming
  - Laziness
  - Impatience
  - Hubris
- Tutorial
- Exercises

Prerequisites

It is recommended to have already read through the Naming Conventions Cheat Sheet first, as good naming conventions are a key aspect of Best Practice.

Overview

80% of the lifecycle of any piece of code goes to maintaining that code. Only 20% of its life is spent being written, refactored, tested, and packaged.

This is true for all programming languages.

As such, what has come to be known commonly as *best practice* is the art of making your code more readable, both to other developers and to yourself.

Generally, *best practice* is a murky term that can refer to any and all of the following:

- organisation-specific style-guides (when on-site you'll likely be pointed in the direction of one of these)
- layout/formatting recommendations made within the Java specification itself
- coding conventions which have become commonplace over time (Java has been around since 1995, after all!)

and, of course:

- straight-up common sense

Style Guides

Convergence in programming style allows for *consistency* in your programs.

If you're working in a team, one of the first things to ask your team-mates is what conventions they personally follow. Once the ensuing argument ends, you can then adhere to a single style of programming that is *consistent*.

CSS
Javascript
Spring Boot
Selenium
Sonarqube
Advanced Testing (Theory)
Cucumber
MongoDB
Express
NodeJS
React
Express-Testing
Networking
Security
Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

Every organisation in the industry will have some kind of style, across the business, which they adhere to above all else. The argument will have already been had - so it's best not to rock the boat.

The most common things you'll find in a style guide will relate to:

- *formatting*
- *specific constructs*
- *comments*
- *naming* (covered in the Naming Conventions Cheat Sheet)
- *convention* (such as those covered above)

One of the best examples of a good Java programming style is the Google Java Style Guide.

Formatting

Brackets

The most common examples of bracketing format are *the One True Brace style (1TBS)* and the *Allman style*:

▶ 1TBS vs. Allman

While it makes no difference in practice which of these styles you use, you should remain *consistent* throughout the course of your program.

Line-breaks and whitespace

Line-breaks generally tend to take place after "significant code".

Most commonly, they are used between different constructs:

▶ Whitespace

Space-padding operators is also usually a good idea, for visual separation:

```
int foo = a + b + 1;
```

Disambiguating your order of operations is also generally recommended:

```
int foo = ((a * b) / (c + 1)) - 5;
```

Finally, when indenting your code, **don't use tabs** - the size of a tab character differs between tools. **Use spaces instead** - four is generally the best to go with.

Specific Constructs

This refers to the layout of certain types of objects in Java.

For instance, you can declare multiple variables separately or in a single line:

```
//individually - best practice
int i;
int j;
int k;

//single-line - avoid this
char c, d, e, f, g, h;
```

Other constructs, such as enums, can be written in several different ways (though this usually depends on circumstance):

```
//single-line
private enum Suit { PENTACLES, WANDS, CUPS, SWORDS }

//multi-line
private enum Answer {
    YES {
        @Override public String toString(){
            return "yes";
        }
    },

    NO,
    PERHAPS
}
```

Comments

A well-written piece of code should read like a beautiful essay.

Generally, comments should be used sparingly, on the basis of *cognitive complexity*.

Cognitive complexity, in Java, refers to how easy it is for a particular line of code to be understood after the first reading.

It is therefore needless to explain every single line of code:

```
public static void main(String[] args){
    // this String holds a name of any length
    public String name = "Jeff";
    //this int holds an age up to the maximum integer value
    public int age = 35;
    // this line will print out the contents of the name and age variables in a
sentence
    System.out.println(name + " is " + age + " years old.");
}
```

Other lines might need further explanation, in which case comments are usually perfectly fine to use.

However, the point of comments is not to use them, but rather to avoid using them to explain your code.

Java Specification Recommendations

An excellent document from the creators of Java was released in 1997 which covers several gold-standard recommendations for writing readable Java.

You'll find that there's a bit of overlap between this, common conventions, and the styles that organisations use.

Some of the more obvious recommendations are listed below:

- *each line should contain at most one statement* - in other words, don't use multiple semicolons on a single line
- *don't use the same variable name twice* - don't declare `int age = 25` in two separate if-else statements, for instance
- *put declarations at the beginning of code blocks* - if you've opened a code block with a { curly braces, always put the variables you expect to use at the start of that block
- *avoid lines that are too long* - 80 characters are generally the limit, but you can separate long pieces of code at a comma, period, or operator

Conventions

Generally, *conventions* are ways of programming that are not set in stone, but are still useful.

There are several good conventions which are worth following from now onwards if you do not already:

- **Always use the `@Override` annotation**: more on this in the Inheritance module
- **Code to an interface**: more on this in the Interfaces module
- **Cascade your exceptions**: more on this in the Exceptions module
- **Don't make classes overly long**
- **Don't make lines longer than 80 columns**

There are several more besides these out there - and you will likely develop a personal programming style that includes several of these!

Common Sense: The Three Virtues of Programming

Common sense programming can be summed up in the form of Larry Wall's 'Three Virtues of Programming':

- *laziness* - programmers should maximise efficiency wherever possible
- *impatience* - why do the work yourself when the magic box is both faster and more accurate than you?
- *hubris*, or excessive pride - if you care about what you're doing, you should care about the quality of your code as well

While these may seem counter-intuitive, the application of these three virtues in practice usually results in well-written, applicable, actually *useful* code.

Laziness

Laziness is commonly conflated with shirking:

- laziness is the art of cutting corners to increase your efficiency
- shirking is the act of avoiding responsibility altogether, thus reducing your efficiency to zero

True laziness actually takes hard work, but has a huge pay-off.

Typically, this takes the form of re-using code that you have done before.

Reusing code between projects saves development time (and therefore cost), but also, if that code has been tested, then it saves time twice, and, ultimately, results in more reliable and efficient programs!

Impatience

The impatient programmer gets the computer to do the work!

- even during the course of normal operation, humans make mistakes
- computers, if told properly what to do, typically make far fewer mistakes

For instance, let's say you have recently ripped several old cassette tapes and digitised them so that you can play that music on your phone.

Adding the artist and album metadata to those albums would take a huge amount of time if you did it manually.

It makes logical sense to simply write a program that writes whatever metadata you need to every song file in a given album for you, rather than to do it manually for every single song in every single album.

This also has the added upside of being correct *every time you do it* - so, as a result, it reduces human error in the future, for if and when you mess something up.

If there is no tool or utility to do what you need, then the best way of solving that problem is to write that tool yourself - or find someone who already has.

Hubris

Hubris, or excessive pride, is often considered a sin.

In programming terms, this simply means being responsible for the code that you write.

If you have pride in something you have created, then you should endeavour to make it the best possible version of itself.

Part of the reason why people refactor their code, or why there are incremental updates to almost every program, is because the people responsible for building those programs have hubris.

Ultimately, hubris means that we care about what we do.

Tutorial

There is no tutorial for this module.

Exercises

Take a look at some of your previous code, and try to refactor it according to the Google Java Style Guide.