# Good Coding Practices

*Spring 2017*

## Contents



## 1 Introduction

This document is the CS 18 "good" coding practices guide. Writing code that follows these practices will serve you well beyond CS 18, because when your code is well structured and well organized, it is easier to read, debug, test, optimize, etc.

Please read this guide carefully. In addition to functionality, your assignments will be graded on how well you abide by these good practices, so this guide should be a valuable tool to you while coding.

Note that this guide is an evolving document, subject to revision as the semester progresses. If we observe any particularly good or bad coding practices in the assignments you turn in, we will extend this document so that we can share those practices with other students.

Finally, the examples in this document are written in Java, but they apply equally well to Scala, and all languages which support the features mentioned. These are language-independent good coding practices; practices you should embrace in all your future programming endeavors.

## 2 The Don'ts

This list is a compilation of what *not* to do while writing code in CS 18 (and beyond).

**Repeat Code** It is bad practice to repeat code. Instead, repeated code should be abstracted out to a helper method, an abstract class, or a utility class so that it can be shared.

**Write Overly Complicated Logic** You should always aim to avoid using overly complicated logic in your code. *The simplest solution is best!*

If you find yourself using convoluted logic in an assignment during CS 18, you are probably not approaching the problem correctly. You should go back to the drawing board.

If you *must* hand in complicated code, be sure to add clear and concise comments, and, as always, give your variables meaningful names.

**Overuse Public** It is bad practice to declare member variables `public`, as they can then be accessed/modified by other code. Instead, they should be declared `private` (or `protected`), and accessed via getters and setters.

**Use `instanceOf`** When programming, you should (almost) always know the type of your data. Needing to use `instanceof` implies a bad design and should make you rethink your class heirarchy.

An exception to this rule is the use of `instanceof` when overriding the `equals` method.

**Boolean Logic** The following are similar to CS 17 guidelines on the correct use of booleans:

- A variable should never be directly compared to a boolean, as doing so is redundant. For example, this condition

  ```
  if (x == true) {
    ...
  }
  ```

  evaluates to `true` if $x$ is `true`, and `false` if $x$ is `false`. Thus, the above statement should instead be written as:

  ```
  if (x) {
    ...
  }
  ```

- An `if`/`else` statement should never set a variable's value to a boolean, as the variable can be set to the value of the predicate instead. For example, this statement

  ```
  if (x == y) {
    myBool = true;
  } else {
    myBool = false;
  }
  ```

  returns `true` if $x$ is equal to $y$, and `false` otherwise. Thus, the above statement should instead be written as:

  ```
  myBool = (x == y);
  ```

- It is bad practice to leave a section of an `if`/`else` statement empty. For example:

  ```
  if (x == y) {
  } else {
    myFunction();
  }
  ```

Instead, this statement should be rewritten as:

```
if (x != y) {
  myFunction();
}
```

**Exceptions** The following good coding practices pertain to throwing and handling exceptions.

- **You should never catch an unchecked exception.** Unchecked exceptions, like `ArrayOutOfBoundsException` and `NullPointerException`, which inherit from `RunTimeException`, represent things that should never happen (e.g., accessing the $-1$st item in an array; getting the first item of an empty collection; etc.). When your program encounters an unchecked exception, it exits ungracefully, dumping error information to the console and confusing your users. If your program is throwing an unchecked exception, don't catch it: *fix your code!* Instead of writing this:

  ```
  try {
    ...
  } catch (NullPointerException npe) {
    ...
  }
  ```

  you should aim to write code such that does not encounter `NullPointerException`s!

- **You should never catch** `Exception`. The `Exception` class is the root of the exception hierarchy—the class from which all exceptions (including `RunTimeException`) inherit. You should never catch `Exception`, as it is unchecked (it is a superclass of the class of all unchecked exceptions!).

  Regardless, you should always catch the most specific exception possible. If you catch only very general exceptions, catastrophic behavior can occur without explanation, making your code difficult to debug. Instead of writing this:

  ```
  try {
    ...
  } catch (Exception e) {
    ...
  }
  ```

  you should catch the specific checked exceptions that you are expecting to encounter and handle them appropriately, like this:

  ```
  try {
    ...
  } catch (FileNotFoundException fnfe) {
    ...
  } catch (IOException ioe) {
    ...
  }
  ```

Furthermore, you should prefer a custom exception to a built-in exception, because you can include information specific to the state of your program in a custom exception, whereas you can only include an error message in a built-in one.

- **You shouldn't write useless `try`/`catch` blocks.**

  Instead, your `try`/`catch` blocks should do some useful work, like printing out an informative error message or handling the error. Printing out a stack trace is generally not all that useful to a user.

  Another pattern to avoid is catching an exception just to throw it again. For example:

```
try {
  ...
} catch (FileNotFoundException fnfe) {
  ...
  throw new FileNotFoundException();
}
```

  You should either not catch the `FileNotFoundException` in the first place or handle it gracefully.

  **Note:** Printing a stack trace or catching and re-throwing an exception may be helpful while debugging, but should be removed before making code available to other users (or graders).

## 3 The Dos

This list is a compilation of what to do while writing code in CS 18 (and beyond).

**Be Lavish with Interfaces** An interface is a set of behaviors. Interfaces are not essential to a program's functionality, but they are useful (perhaps even essential) to debugging because they create requirements for programmers. As a rule of thumb, you should *be lavish with interfaces*. More interfaces make your code more readable, more extensible, and more easily testable. When in doubt, use an interface!

**Properly Format Code** Poorly-formatted code can be difficult to read and should be avoided. In CS 18, we provide you with an Eclipse autoformatter to help make sure that all code you write is nicely formatted. Instructions for installing it can be found in Lab 1. If you use our formatter, you won't lose points for poorly-formatted code.

**Constants** Constants should be declared at the top of a class rather than hardcoded throughout. For example:

```
int[] array1 = new int[10];
int[] array2 = new int[10];
int[] array3 = new int[10];
```

should be rewritten as:

```
int ARRAY_SIZE = 10;

int[] array1 = new int[ARRAY_SIZE];
int[] array2 = new int[ARRAY_SIZE];
int[] array3 = new int[ARRAY_SIZE];
```

In this latter code snippet, it is much easier to understand the choice of the value 10. In addition, it is much easier to change the size of the arrays from 10 to 100 a year from now, when you have long since forgotten where all the hardcoded instances of 10 appear in your code base. (Likewise, it makes it easier for someone other than you to make this change.)

**Use Brackets** While your Java code may well compile without brackets, you should *always* use them when writing `if` statements, `for` loops, and `while` loops.[1] For example:

```
if (x == y) {
  return "x equals y, yay!";
}
```

is preferred to

```
if (x == y)
  return "x equals y, yay!";
```

**Use Descriptive Variable Names** Your variable names should be as informative as possible. The following are examples of poorly-named variables:

```
p
var
array16
```

These names do not help the reader (often you!) understand what they represent. The following are examples of well-named variables:

```
counter
inputFile
gradeArray
```

## 4 CS 18-Specific Practices

**Support Code** Never copy and paste support code from the CS 18 directory into your own code. Instead, you should import support code as directed in our handouts.

**Important:** Before turning in an assignment, be sure to run your code on a deparment machine (either in person or through '`ssh`') to ensure that any support code imports correctly.

[1] Tim once wasted an entire day debugging because he had taken this shortcut and not grouped statements correctly.

**Delete Debugging Comments** You should not turn in code that includes code you write for debugging purposes only, rather than functionality. For example, do not include code that prints stack traces or contains TODOs, `println` statements, commented out code, method stubs, or methods that are never called. All of the above are useful tools to have at your disposal while debugging, but you must delete them before turning in an assignment, as they complicate the grading process. You will be penalized if you do not follow this instruction.

## 5 Style

Another good coding practice is to abide by the conventions of the language in which you are programming. These conventions are usually outlined in a **style guide**. Indeed, such a guide has been developed expressly for CS 18. You should read this style guide to better understand the conventions enforced by the CS 18 Java and Scala autoformattera.

If your computer science career extends beyond Brown, you are certain to encounter a style guide. So one last good coding practice is to get in the habit of following one now!

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs018/feedback`.