Java Style Guide

Spring 2019

Contents

1 Introduction

This document is a Java style guide. Reading someone else's code (or even your own code at some later date) formatted in a standard style that you are familiar with makes understanding how that code operates much easier. For this reason (and others),[1] the use of style guides is now standard practice in commercial and open source software development. This guide is inspired by the Java code conventions published by Google. Experience following guidelines like the ones described here will serve you well beyond CS 18.

Please read this guide carefully. As in CS 17, your assignments will be graded on style as well as functionality, so this guide should be a valuable tool for you while coding.

2 Naming

Almost all identifiers should consist of only letters and digits. Only constants may additionally use underscores. Do not use any other types of characters in your identifiers.

Packages

Package names are all lower case, with consecutive words concatenated together. For example: `mypackage`

Classes and Interfaces

Class names and interface names are written in UpperCamelCase, in which words are concatenated and every word starts with an uppercase letter. For example: `ArrayList` and `Comparable`

Constants

Constant names are written in `CONSTANT_CASE`: all uppercase letters with words separated by underscores. For example: `MAX_NUMBER_OF_PIZZAS`

Type parameters

Type parameters (i.e., generics) are given one character names in Java. While this convention may seem limiting (e.g., `Key` and `Value` might be preferred to `K` and `V`), it in fact serves to differentiate type parameters from class and interface names.

Some common type parameter names are listed in Table 1.

[1] In industry, mandating a style standard also allows multiple people to work on the same code base more easily. Without a standard, people can waste hours formatting and reformatting code to adhere to their own personal style.

Name	Meaning
E	element
N	number
V	value
K	key
T	type
S, U, V	other

Table 1: Common Type Parameter Names

Everything Else

Everything else (method names, non-constant field names, parameter names, and variable names) are written in lowerCamelCase, in which words are concatenated and every word except the first starts with an uppercase letter.

One character names should be avoided, except when looping.

3 Formatting

Indentation

Indent with the space character only, not the tab character.

Each time a new block or block-like construct is opened, the indent level should increase by two spaces. When the block ends, the indent level should return to the previous level. The indent level applies to both code and comments throughout the block.

```
// Good style
public void myMethod(String myFavColor) {
  if (myFavColor == "Wisteria") {
    System.out.println("Your favorite color is Wisteria."); // Indented two
        spaces
  } else { // Indentation level returned to previous one
    System.out.println("Your favorite color is not Wisteria.");
  }
}
```

Line Wrapping

Lines of code should (almost) never exceed 80 characters in length.

When line-wrapping a long statement, each continuation line should be indented by four spaces from the original. For example:

```
aVeryLongArgumentName = anotherEvenLongerArgumentName +
    anotherStillLongerArgumentNameByJustOneCharacter;
}
```

Vertical Whitespace

You should insert a single blank line in the following places, *only*:

1. After any class declaration, and before the class' final closing brace.
2. Between consecutive members of a class, i.e. fields, constructors, methods, etc.
3. After most, but not all, closing curly braces. For example:

```
public class MyClass {

  public static void main(String[] args) {
    System.out.println("Hello world");
  } // A closing curly brace that ends a method

} // A closing curly brace that ends a class
```

There are exceptions to this general rule!

Do not skip a line until the very last curly brace in an **if/else** block. For example:

```
// Bad style: blank lines between blocks
if (exampleInt == 15) {
  System.out.println("The example integer is 15");
}

else if (exampleInt == 16) {
  System.out.println("The example integer is 16");
}

else {
  System.out.println("The example integer is neither 15 nor 16");
}
```

```
// Good style: no blank lines between blocks
if (anotherExampleInt == 17) {
  System.out.println("The example integer is 17");
} else if (anotherExampleInt == 18) {
  System.out.println("The example integer is 18");
} else {
  System.out.println("The example integer is neither 17 nor 18");
}
```

Or if a sole curly brace is the very last line of a method:

```
public static void main(String[] args) {
  int myAwesomeInt = 18;

  if (myAwesomeInt == 18) {
    System.out.println("I love CS18!");
  } else {
```

```
    System.out.println("I still love CS18!");
  }
} // There is no space between this curly brace and the one above it.
```

4. Before a method's final `return` statement (usually). For example:

```
// Bad style: no space before the return statement
myVar++;
return myVar;
```

```
// Good style: wow that space looks beautiful!
myVar++;

return myVar;
```

   Exception: If a method consists of only one line (`return` statement or otherwise) you should not include any blank lines in that method's definition. Likewise, if your method is particularly short, say less than three lines of code, you need not skip any lines.

   This same exception also applies to *blocks* of code. If a block of code actually only consists of a single `return` statement, or if a block of code it is short but nonetheless contains a `return` statement, you need not skip any lines.

5. Within method bodies as needed to create logical groupings of statements—*to make your code read like a book!*

Finally, in case you haven't noticed, let's be very explicit about it. You should *not* insert a blank line after a class or method declaration before the opening brace. The following is *bad* style:

```
publc class BadStyle
{
  ...

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

Horizontal Whitespace

A (single) space appears in the following places:

1. Separating a keyword from an opening parenthesis. For example:

```
// Good style
for (int i = 1; i < 10; i++)

// Bad style: no space between opening curly brace and keyword "for"
```

```
for(int i = 1; i < 10; i++)

// Good style
if (val < 10)

// Bad style
if(val < 10)
```

2. Separating a keyword from a closing curly brace that precedes it. For example:

```
// Good style
if (val == 10) {
// Do a thing.
} else {
// Do a different thing.
}

// Bad style: no space between closing curly brace and keyword "else"
if (val == 10) {
// Do a thing.
}else {
// Do a different thing.
}
```

3. Before any open curly brace.[2]

4. On both sides of any binary or ternary operator (operators that act on two or three elements) or operator-like symbols. Examples of binary operators include `+`, `-`, `*`, `/`operator-like symbol is the colon `:` in a `foreach` statement.

   In contrast, unary operators (such as the `++` in the statement `myNum++;`) should not have whitespace between them and the element they are operating on.

5. After commas and semicolons (but *never* before these).

6. After the closing parenthesis of a cast. For example:

```
// Good style
(Double) myInt;

// Bad style: squished
(Double)myInt;
```

   (Of course, casting itself isn't great style.)

7. On both sides of an end-of-line comment. For example:

```
int myInt; // This is an example of good comment spacing.
int myInt;// This is one example of bad comment spacing.
int myInt; //This is another example of bad comment spacing.
int myInt;//This is awful comment spacing!
```

[2]With the unusual exception of multi-dimensional array declarations.

8. Between the type and variable of a declaration. For example:

```
// Good style
int myInt;
List<Int> myList;

//Bad style: no space!
intmyInt; // This isn't correct Java syntax anyway.
List<Int>myList; // This is correct syntax, but is bad style!
```

Here's an example of a place where you should *not* insert any space: between a method's name and the opening parenthesis that delimits its arguments. For example:

```
// Bad style: a space after the method's name (UGH!)
public static void main (String[] args) {
  ...
}

// Good style
public static void main(String[] args) {
  ...
}
```

But in general, the space bar is your friend! Use it to improve the readability of your code.

4 Declarations

Variable declarations should declare only one variable. Thus,

```
int exampleVar1, exampleVar2;
```

is considered to be bad style, and should be replaced with

```
int exampleVar1;
int exampleVar2;
```

Variables should be declared very close to the point at which they are first used, and should be initialized either during, or immediately after, declaration, like this:

```
int myVar = 0;
```

or

```
int myVar;
myVar = 0;
```

not like this:

```
int myVar;
// Some unrelated code
myVar = 0;
```

5 Statements

5.1 Simple Statements

Each line should contain at most one statement. For example:

```
argv++;          // Correct
argc--;          // Correct
argv++; argc--; // Avoid!
```

5.2 Compound Statements

Compound statements are statements that contain lists of single statements enclosed in braces:

```
... {
  statement1;
  statement2;
  ...
  statementn;
}
```

(In what follows, we abbreviate the above as `statements;`.)

In a compound statement, the opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line, and be indented to level of the compound statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as an **`if-else`** or `for` statement. Braces are even used when a control structure has an empty body. This makes it easier to insert statements later as necessary without accidentally introducing compilation or logical bugs.

5.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. For example,

```
return;

return myDisk.size();

return (val > 17 && val < 19);
```

5.4 if, if-else, if else-if else Statements

Conditional should be layed out as follows:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
} else {
  statements;
}

if (condition) {
  statements;
} else if (condition) {
  statements;
} else {
  statements;
}
```

Note: `if` statements always use braces, {}. For more information, refer to the good coding practices guide.

5.5 for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
  statements;
}
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization) or at the end of the loop (for the update).

5.6 for-each Statements

A `foreach` statement should have the following form:

```
for (Type identifier : iterableCollection) {
  statements using identifier;
}
```

5.7 while Statements

A `while` statement should have the following form:

```
while (condition) {
  statements;
}
```

5.8 do-while Statements

A `do-while` statement should have the following form:

```
do {
  statements;
} while (condition);
```

5.9 switch Statements

A `switch` statement should have the following form:

```
switch (condition) {
  case ABC:
    statements;
    /* falls through */

  case DEF:
    statements;
    break;

  case XYZ:
    statements;
    break;

  default:
    statements;
    break;
}
```

Every time a case in a `switch` statement falls through (i.e., doesn't include a `break`), add a comment where the `break` statement would normally be that says something like `/* falls through */`. This way the reader of the code need not question whether the missing `break` was deliberate.

Every `switch` statement must include a `default` case. The `break` in the default case is redundant, but it can prevent a fall-through error if another case is inserted later.

5.10 try-catch Statements

A `try-catch` statement should have the following form:

```
try {
  statements;
} catch (ExceptionClass e) {
  statements;
}
```

Likewise, a `try-catch-finally` statement should have the following form:

```
try {
  statements;
} catch (ExceptionClass e) {
  statements;
} finally {
  statements;
}
```

6 Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those originally used in C, which are delimited by `/*...*/` , and `//`. Documentation comments (*a.k.a* "doc comments") are Java-only, and are delimited by `/**...*/`. Doc comments can be exported to HTML files using the `javadoc` tool.

Implementation comments are meant for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers (or TAs) who might not necessarily have the source code at hand.

Comments should be used to give an overview of code, and to provide additional information that is not readily available in the code itself. Discussion of non-trivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code.

Excessive commenting can be indicative of poor code quality. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should contain only information that is relevant to reading and understanding the program. Information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Comments should not be enclosed in large boxes drawn with asterisks around them, or any other characters. And they should never include any special characters, like back space or line break.

Note: You may find it useful to comment out multiple lines of code; however, any work you submit for CS 18 should not contain commented-out code.

6.1 Implementation Comments

Programs can have different styles of implementation comments: block, single-line, and end-of-line.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures, and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code. But code can then immediately follow a comment, without a new line. For example:

```
statements;

/**
 * Here is a block comment preceded by a blank line,
 * and followed immediately by code.
 */
int myVar = 0;
```

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. For example:

```
if (condition) {

  /* This is what is done to handle the condition. */
  ...
}
```

Note: Some people never use this style of commenting, preferring the next style for both single- and end-of-line comments.

End-Of-Line Comments

The `//` comment delimiter can comment out a complete line or only a partial line. If multiple lines of text comments are needed, a block comment should be used. For example:

```
if (foo > 1) {

  // Do a double-flip.
  ...
} else {
  return false; // Explain why here.
}
```

6.2 Documentation Comments

Documentation—"doc," for short—comments describe Java classes, interfaces, constructors, methods, and variables. Each doc comment is set inside the comment delimiters `/**...*/`, with no more than one comment per class, interface, or member.

The first doc comment should appear just before the class declaration:

```
/**
 * The Example class provides ...
 */
public class Example {
```

```
  ...
}
```

In work you submit for CS 18, doc comments should be present for every method unless one of the following applies:

1. The method name is self-explanatory: e.g., a getter like `getName` that simply returns the name of an object need not have doc comments.
2. The method overrides another method for which you have (necessarily) already provided doc comments.

Doc comments use tags, so that the '`javadoc`' tool knows knows what it should export to HTML. Here are some of the more popular tags, along with a description of their usage:

- `@param`: Documents a specific parameter (argument) of a method or class constructor. You should include one tag per parameter.
- `@return`: Documents the return value of a method. (Only one @return tag is allowed per method.)
- `@throws`: Documents the exceptions the method may throw, and under what conditions.

Below is an example of a well-documented class that uses doc comments, as well as standard comments to facilitate readability. Since the doc comments are only used to describe functionality at a high-level, we also include standard comments to describe particular implementation choices and/or explain any non-obvious logic in the code.

```
/**
 * A class that handles writing to a specific file
 *
 * Use the write() method to write to a file
 */
public class WriterClass {

  private File myFile;

  /**
   * Constructs a WriterClass object.
   * @param fileName - the file name to write to
   */
  public WriterClass(String fileName) {
    this.File = new File(fileName);
  }

  /**
   * Writes a string to a file specified by a file name
   *
   * @param str - the string to write
   * @param file - the file name to write to
```

```
   *
   * @throws IOException
   * @return false if the file is created by this method, true otherwise
   */
  protected boolean write(String str) throws IOException {
    boolean fileExists = myFile.exists();

    // FileWriter throws an IOException when the target file cannot be
       opened
    BufferedWriter writer = new BufferedWriter(new FileWriter(myFile));
    writer.write(str);

    return fileExists;
  }

}
```

While the `File` constructor in this method could throw a `NullPointerException` (i.e., when the input string is `null`), this exception, which is unchecked, is not documented. As a rule of thumb, *only checked exceptions should be documented in doc comments.* Unchecked exceptions are omitted, because they should be treated as bugs, and therefore fixed, rather than handled!

7 More Annotations

Code annotations are metadata that do not affect a program's operation, but instead provide data about the program. In CS 18, we use annotations to comment our code and to communicate with the compiler. In CS 18, whenever you override a method, you must explicitly write `@Override` on the line above the method's header. For example, in the case of a bank account's `toString` method:

```
@Override
public String toString() {
  return (this.name + "'s bank account contains " + this.balance + " dollars
      .");
}
```

This rule also applies to methods in a class that implements an interface. For example:

```
public interface IShape {
  public double getArea();
  public double getPerimeter();
}
```

```
public class Circle implements IShape {

  private area;
  private perimeter;

  ...
```

```
  @Override
  public double getArea() {
    return area;
  }

  @Override
  public double getPerimeter() {
    return perimieter;
  }
  ...

}
```

8 File Structure

A Java file generally consists of a bunch of headers (i.e., `package` declarations and `import` statements) followed by a single class definition.

8.1 Headers

The very first line in a Java file declares the name of the package to which the file belongs. Unlike almost everything else, the `package` declaration is not line wrapped and the 80 character limit does not apply. For example:

```
  package
      exampleofapackagenamethatisverylongbuttheeightycharacterlimitdoesnotapply
      ;
```

(Recall that package names are all lowercase, with consecutive words simply concatenated together.)

After the package declaration comes any `import` declarations for the Java classes, interfaces, etc. referred to in the file. It may be tempting to use a wildcard to import entire packages whenever more than one file from a single package is needed. But this is considered bad style, because it hides relevant information from the user. Instead, each file should be imported individually. For example:

```
  // Bad style: don't import wildcards!
  import java.util.*;
```

```
  // Good style: only import exactly what you need
  import java.util.List;
  import java.util.Map;
```

Java packages are organized in a hierarchical fashion. For example, the package `java.util` includes the packages `java.util.List` and `java.util.Map`.

Likewise, `import` declarations should be organized hierarchically by package, with all the `java` packages grouped together, all the `hw01` packages grouped together, all the `lab01` packages grouped

together, etc., and with a single blank line inserted between each group. Furthermore, official Java packages like `java` and `javafx` should appear before any CS 18-specific packages.

Also, within each group, the subpackages should be listed in sorted alphabetical[3] order. So `java.util.List` comes before `java.util.Map`.

The following examples illustrate good and bad importing:

```
// Bad style: Packages are not separated by group.
import java.util.List;
import javafx.application.Application;
import mylocalpackage;
```

```
// Bad style: Packages are not in the correct order.
import javafx.application.Application;

import mylocalpackage;

import java.util.List;
```

```
// Bad style: Packages are not sorted within groups.
import java.util.Map;
import java.util.List;

import javafx.collections.ObservableList;
import javafx.application.Application;

import mylocalpackage;
import anotherlocalpackage;
```

```
// Good style
import java.util.List;
import java.util.Map;

import javafx.application.Application;
import javafx.collections.ObservableList;

import anotherlocalpackage;
import mylocalpackage;
```

Finally, be sure to skip a line between package declarations and `import` declarations. For example:

```
// Bad style: missing requisite whitespace
package mynotsogreatpackage;
import java.util.List;
import java.util.Map;
```

[3] more specifically, ASCII sorted

```
// Good style: eminently readable
package myexcellentpackage;

import java.util.List;
import java.util.Map;
```

Note that like package declarations, `import` declarations are not line-wrapped, and thus the 80 character limit does not apply.

8.2 Class Structure

Java permits one class per file, in a file named for the class: e.g., `MyAwesomeClass` should be saved in a file called `MyAwesomeClass.java`. Just above the class itself, include a Javadoc comment that describes the purpose of the class. Inside the class, the constructors and any inner classes should appear near the top, while `main` should be the last method in the class.

Note: In order to help you (and readers of your code) easily determine whether a variable being accessed is an instance variable, instance variables should always be prefixed by `this.`.

```
/**
* A class for awesome things.
*
* Use the makeAwesome method to make things the most awesome they can be.
*/
public class MyAwesomeClass {

  private int count;
  private boolean isAwesome;
  private final static String AWESOME_STRING = "Super-totally-radical-groovy
      ";

  /**
   * A class for creating an AwesomeException.
   */
  private class AwesomeException extends Exception {
    /**
     * Constructor for a MyAwesomeClass.
     */
    public AwesomeException() {
      super("This is not so awesome, unfortunately!");
    }
  }

  /**
   * Constructor for a MyAwesomeClass.
   * @param count - number of awesome strings
   * @param isAwesome - true if it is awesome, false otherwise
   */
  public MyAwesomeClass(int count, boolean isAwesome) {
    this.count = count;
    this.isAwesome = isAwesome;
  }
```

```
  /**
   *  Proclaims how awesome things really are.
   *
   *  @throws AwesomeException when something not so awesome happens
   */
  public void howAwesome() throws AwesomeException {
    if (!(this.isAwesome)) {
      throw new AwesomeException();
    }

    for (int i = 0 ; i < this.count; i++) {
      System.out.println(AWESOME_STRING);
    }

    this.isAwesome = true;
  }

  public static void main(String[] args) {
    MyAwesomeClass awesome = new MyAwesomeClass(18, true);

    try {
      awesome.howAwesome();
    } catch (AwesomeException e) {
      System.out.println(e.getMessage());
    }
  }

}
```

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`.