CS200 Java Style Guide

Introduction

Employing good and consistent coding conventions is very important. It helps make the programs readable so that other programmers (and yourself!) are better able to use, edit, and add to your code. For these reasons (and others), 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 Oracle. Experience following guidelines like the ones described here will serve you well beyond CS200.

This guide is broken up into two parts: Essential Style (style that you will be graded on) and Recommended Style (general good practices to follow). Please read this guide carefully *and in its entirety*. Refer back to it throughout the semester; in addition to functionality and design, your assignments this semester will be graded on style.

Don't worry if you're not sure what all of the terms and concepts below mean; you will soon get to know them thoroughly. After reading this document, make sure you keep it by your side when you code to answer your questions about format and style.

Essential Style

The conventions outlined in this section **must** be followed in all code you write. Below are the essential aspects of writing well-styled code in Java, and you will be graded on how well you adhere to them.

Naming

All identifiers (variable names, method names, class names, etc.) should consist of only letters and digits. The only exception to this rule is constants, which 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

This includes interfaces and enums

Classes should be named using nouns which describe what the class models. Class names should be written in UpperCamelCase, in which words are concatenated and every word starts with an uppercase letter. For example: `Ocean` and `TreeHouse`

Constants

Constants should have names consisting of nouns describing their content. Constant names are written in CONSTANT_CASE: all uppercase letters with words separated by underscores. For example: `MAX_NUMBER_OF_PIZZAS`

Everything Else

This includes fields, method names, parameter names, and variable names

Everything else is written in lowerCamelCase, in which words are concatenated and every word except the first starts with an uppercase letter. For example: funAndCoolMethod and numFriends

Note: Single-character names should be avoided, except when looping.

Summary

Wow, that's a lot! Here's a quick summary of naming conventions:

"Type"	Naming Convention	Example
Package	All lowercase	`mypackagename`
Class	First letter capitalized, subsequent words start with capital letter	`PetShop`
Constant	All uppercase, words separated by underscore	`MAX_WORD_LENGTH`
Method	First letter lowercase, subsequent words start with capital letter	`myMethod`

Variable / Field / Parameter	First letter lowercase, subsequent words start with capital letter	`numLetters`

Formatting

Indentation

A new block or block-like construct in Java is delineated by curly braces (`{}`). The code within each new block should be indented by four spaces relative to the previous level. When the block ends, the indent level should return to the previous level. *This indent level applies to both code and comments throughout the block.*

```
// Good style
public void checkAdult(int age) {
    if (age >= 18) { // Open brace means we should indent the  code that
follows
        System.out.println("You are officially an adult!");
    } else { // Indentation level returned to previous one
        System.out.println("Not an adult yet...");
    }
}
```

As you edit your program, you may need to change the indentation of your code. In IntelliJ, the keystrokes

Command/Control + Option/Alt + L

will automatically reformat your code to have proper indentation.

In VSCode, the keystrokes to reformat indentation are:

Option/Alt + shift + F

Line Wrapping

Lines of code should never exceed 80 characters in length, as anything longer than that becomes less and less readable. Such lines should be broken up in what’s called “line-wrapping” or “line-breaking.”

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

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

**see the Line Wrapping section in Recommended Style for more specific examples of how to properly break up a line.*

Vertical Whitespace

Another way of making major sections of your code easy to read is by skipping lines (i.e. leaving vertical whitespace). You should skip one line between methods and other sections within a class. You should also separate logical chunks of code within a method with one or two blank lines.

In short, you should skip a line in your code when it improves readability, such as between methods and between local sections within methods. Never skip more than one line in any of your code – excessive whitespace is bad and makes your code more difficult to read.

Finally, you should **not** insert a blank line before the opening brace of a class or method. The following is bad style:

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

**see the Vertical Whitespace section in Recommended Style for more specific examples of when to skip a line.*

Horizontal Whitespace

The spacebar is a shockingly useful tool when formatting your code. Much like with skipping lines, we want to leave spaces only when doing so will improve the readability of our code.

Specifically, adding a space can make code blocks feel less crowded and therefore more manageable to read and understand.

A (single) space must appear in the following places:

1. On both sides of any binary or ternary operators and operator-like symbols. Examples of these include (but are not limited to) `+`, `-`, `*`, and `:`
   In contrast, unary operators (such as the `++` in the statement `myNum++;`) should not have whitespace between them and the element they are operating on.
2. After commas and semicolons (but never before these).
3. Between the type and variable of a declaration. For example:

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

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

4. Before any open curly brace, and after any closing curly brace (if it's followed by a keyword). 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"
 * Bad style: no space between closing parentheses and opening curly
 * brace
 * Bad style: no space between keyword "else" and opening curly brace
 */
if (val == 10){
    // Do a thing.
}else{
    // Do a different thing.
}
```

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

In general, the spacebar is your friend! Use it to improve the readability of your code.

**see the Horizontal Whitespace section in Recommended Style for more examples of when to use the spacebar.*

Summary

At the end of the day, good formatting makes your program more readable for yourself, your partner, your TAs, and anyone else who has to look at and understand your code. Generally, you should be able to see when your code is readable or not. Here's a quick summary of the guidelines you should follow to get there!

Formatting Choices	**Style Rules to Follow**
Indentation	All blocks of code (as delineated by curly braces) should be indented from the previous block
Line Wrapping	Split up any line over 80 characters wherever you find a logical breaking point. Don't forget to indent the wrapped portion
Vertical Whitespace	Leave a blank line to separate logical chunks of code, both within methods and between them
Horizontal Whitespace	Use the spacebar to separate punctuation and symbols from the text surrounding them, much like you would when writing English.

Commenting

No matter how simple a program is to its author, it’s almost always confusing to another person reading it (such as the TAs who will be grading and helping debug your programs). Comment well, and you will have a happy TA. Comment poorly, and the TA will have trouble understanding what you are doing (and you may not get the benefit of the doubt).

Your comments should give an overview of code, and provide additional information that is not readily available in the code itself. They should only contain information that is relevant to reading and understanding the program, thus discussion of non-trivial or non-obvious design decisions is appropriate, but duplicating information that is present in (and clear from) the code is not.

While commenting your code is extremely important, commenting too much can be just as harmful as not commenting at all. Please be aware of this when commenting your code. Sometimes it’s easier to rewrite a small section of code to be clearer than to excessively write comments for it.

Java programs can have two kinds of comments: implementation comments and documentation (doc) comments. Implementation comments are meant to describe particular lines or sections of code that may be especially confusing at first glance. Doc comments are meant to describe the overall specifications of the code, to be read by developers (or TAs) who might not necessarily have the source code at hand.

Note: You may find it useful to comment out multiple lines of code when writing your program. All work turned in should not contain any commented-out code.

Implementation Comments

Implementation comments can come in three formats: ones that span multiple lines, ones that just span a single line, and ones that appear at the end of lines.

Multi-Line Comments

Comments that span multiple lines may be used to describe methods, data-structures, or major algorithms within your code. They should adhere to the following format:

```
/*
 * This is a multi-line comment (aka block comment)!
 * You use this type of comment to describe especially important /
 * tricky logic in your program as a means of internal documentation
 */
```

Single-Line Comments

Comments that will only span one line are often used in similar situations to multi-line comments, but when not as much explanation is required! They should adhere to one of the two following formats:

```
// This is one style of single-line comment

/* This is another style of single-line comment */
```

Using the `//` format tends to be more common for single-line comments, although both formats are acceptable and you can use either of them. Try to be consistent within your code with which format you use.

End-Of-Line Comments

End-Of-Line comments are written after a line of code (on that same line) and to explain the purpose of a given line of code, or the situations in which it is useful. End-Of-Line comments are always in the `//` format:

```
int threshold = 5; // Threshold for doing a flip
if (foo > threshold) {
    ... // Do a double-flip.
    return true;
} else {
    return false; // False because no flip was done
}
```

Documentation Comments

Documentation comments (aka doc or javadoc 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, constructor, method, or variable.

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

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

Javadoc comments should be present for every method. They follow a standardized format, using tags to reference important values and properties of the method. Here the tags you will be responsible for knowing and using, along with a description of their usage:

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

Let's put it all together! 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.myFile = 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 = this.myFile.exists();
```

```

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

      return fileExists;
    }
  }
```

Summary

Who would've thought that leaving comments explaining your code could be so complicated? Don't worry, we've broken it down into a quick review of everything we just went over.

First, a table summarizing implementation comments:

Comment Type	Formatting

Multi-Line	`/*`
	` * Write multi-line comments in`
	` * this format!`
	` */`

Single-Line	`// Write single-line comments like this`
	
	or
	
	`/* Write them like this! */`
	
	Just remember to be consistent in which one you choose

End-Of-Line	`int number; // Comment the ends of lines like this`

Now, a quick FAQ on Javadoc comments:

Q: Are Javadoc comments the same as documentation comments?

A: Yes! Documentation comments are also known as doc or Javadoc comments

Q: Where should I write Javadoc comments?

A: Before every class and method declaration in your program!

Q: What should I include in my Javadoc comments?

A: Any important information about the functioning of this class / method (the implementation of it doesn't matter here!)

Q: What are tags?

A: There are three types tags we will use in our Javadoc comments for CS200: `@param` to describe each parameter of a method, `@return` to describe the return value of a method (if it has one), and `@throws` to describe each exception the method throws.

Q: What is the format of a Javadoc comment?

A:

```
/**
 * Javadoc comments should be formatted like this. Make sure to
 * give a high level overview of each class or method with them!
 *
 * @param paramTag - this is an example of how you would use a tag
 */
```

Other Important Notes

`@Override`

Whenever you override a method, you must explicitly write `@Override` on the line above the method's header. This makes your code clearer to anyone reading and trying to understand it.

For example, overriding the toString method in a BankAccount class:

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

Or, implementing methods from an interface:

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

public class Circle implements IShape {
  private double area;
  private double perimeter;

  // Code elided

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

  @Override
  public double getPerimeter() {
    return this.perimeter;
  }
}
```

Finally, overriding a method from a superclass:

```
public class Mammal {
    // Code elided

    public void prepareForWinter() {
        this.growThickCoat();
        this.eatMore();
    }
}
```

```java

public class Bear extends Mammal {
    // Code elided

    @Override
    public void prepareForWinter() {
        super.prepareForWinter();
        this.hibernate();
    }
}
```

`this` Keyword

We use the keyword `this` to access methods and fields that are defined within the class they're referenced from. You should **always** use this when appropriate.

Import Statements

You must import predefined classes (such as `java.util.ArrayList` or `java.util.Map`) before being able to use them in your code. When importing classes, you should always import the specific class, rather than using `.*` to get the package as a whole. For example:

```java
// Bad style: this imports all packages in java.util
import java.util.*;

// Good style: this imports each necessary class individually
import java.util.ArrayList;
import java.util.Map;
```

Recommended Style

The style tips below are all great and recommended practices for Java style, but are not something you will be specifically graded on in CS200. Please read it in its entirety to get familiar with the concepts, and try to incorporate them in your code! It will serve you well beyond this course to get in the habit of following these tips now.

Formatting

This is a continuation of the “Essential Style” Formatting section, including more specific details on how to achieve the best Java formatting.

Line Wrapping

Sometimes, it makes more sense to break a long line of code well before its character limit to enhance readability. This can be done in lines that use operators, method names, or simply when there isn’t another logical way to break up the line. For example:

```java
int totalNumber = this.generateANumber() +
        this.generateAnotherNumber() +
        this.generateOneMoreNumber() +
        this.generateLastNumber();


public void myMethod (int intParameter, String stringParameter,
        boolean boolParameter, double doubleParameter,
        float floatParameter) {
    // Method code
}

double myValue =
        this.reallyLongMethodName() +
                this.anotherStrangelyLongMethodName() / 2;
```

Vertical Whitespace

You should insert a single blank line in the following scenarios,

1. Between different “sections” of a class, i.e. fields, constructors, methods, etc.
2. After closing curly braces that end a method or that are immediately followed by other (non-curly brace) code
3. Before the final return statement in a method with more than 3 lines.
4. Within method bodies as needed to create logical groupings of statements—to make your code read like a book :)

Let’s look at all of this in action!

```
public class GreatSpacing {
    private int myNumber;
    private String myString;

    // Notice the space between the field declarations and constructor!
    public GreatSpacing() {
        // Code elided
    }

    public void checkValue(int val) {
         // Notice the proper formatting for the if statements below
        if (val < 10) {
            System.out.println("Your value is smaller than 10!");
        } else if (val >= 10 && val < 20) {
            System.out.println("Your value is between 10 and 20!")
        } else {
            System.out.println("Your value is greater than 20!")
        }
    }

    public int getMyNumber() {
          return this.myNumber; // This is the only line of the method, so there is no whitespace before or after it
    }

    public double pythagoreanTheorem(double num1, double num2) {
        double num1Squared = num1 * num1;
        double num2Squared = num2 * num2;
        double sumOfSquares = num1Squared + num2Squared;

          return Math.sqrt(sumOfSquares); // Leave a space before the return statement in this more complicated method!
    }
}


```

Horizontal Whitespace

In addition to the situations mentioned in the Horizontal Whitespace section of Essential Style, these are a few more situations where using the spacebar is recommended to improve readability of your code.

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

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

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

3. 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!
```

Using Variables

This section does not appear in Essential Style at all, but contains important information on variable declarations that should be kept in mind at all times when coding!

Declaration

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;
```

Why not save space and declare multiple variables together on one line?

Initialization

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;`

or

`int myVar = 0;`

not like this:

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

One exception to this rule can be when your variable is a boolean and the value it is assigned to is based more on complicated logic that can't be computed in a single line. In that case, you wouldn't assign your declared variable to a value until after its value has been properly computed.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS200 document by filling out the anonymous feedback form! (you do have to sign in but we don't see it)