Code Style Guide

Sam McCauley
Data Structures, Spring 2026

This handout summarizes the required code style conventions for CS 136. Writing clean, consistent code is an important part of software development: it makes code easier to read, debug, and collaborate on. The rules below will be enforced on all further assignments. This document will continue to be updated when we learn new code style requirements as the semester progresses.

Comments

Java has two styles of comments:

- **Single-line comments:** any text after `//` on a line is ignored by Java.
- **Multi-line comments:** any text between `/*` and `*/` is ignored by Java. These can span multiple lines.

```
// This is a single-line comment

/* This is a multi-line comment.
   It can span several lines. */
```

You should use comments to explain any code that is *not self-explanatory*.

- `x = x - 1` is self-explanatory—no comment needed.
- `if ((n & (n - 1)) == 0)` is not obvious—it needs a comment explaining what it checks.

Javadoc Comments

Every class and every public method you write in Java should have a Javadoc comment placed immediately before it. Real-world Java code also requires a javadoc comment for each field (that is to say: for each instance variable and each static variable); this is not required in this class.

Javadoc comments begin with `/**` and end with `*/`. Every line in between the beginning and end should start with a `*`.

The first line of the javadoc comment should concisely explain the method or class. Any further lines should give more details.

Some lines of the javadoc comment start with "block tags" that start with `@`. In fact, these are *required* for methods. In particular, any method should have the following block tags:

- `@param` describes a parameter. There should be a block tag on its own line for each parameter of the method.
- `@return` describes what the method returns. It is required for any method with a non-`void` return type.
- `@pre` describes any *preconditions* for the method: does the method make any assumptions before it begins. Any assumptions about the parameters go in the `@param` blocks and are not required here.
- `@post` describes any *postconditions* for the method: what does the method do/what does it update. Any postconditions involving what the method returns go in the `@return` block and are not required here.

Indentation

Indentation in Java is purely visual—it does not affect how the program runs. You should indent anyway. While it's required to indent correctly before submitting, you should maintain proper indentation while programming yourself. Consistent indentation makes code vastly easier to read and debug.

- Indent the body of every class, method, loop, and conditional one level relative to its enclosing block.
- Your editor can auto-indent your code: use **Option-Shift-F** (Mac) or **Alt-Shift-F** (PC/Linux) in Visual Studio Code.

Naming Conventions

Class names. Class names are always capitalized (UpperCamelCase). For example: `Student`, `WorkOfArt`, `LinkedList`.

File names. A class must be stored in a file with the **exact same name** as the class, with a `.java` extension. For example, the class `Student` must be in `Student.java`. (Java enforces this requirement.) Note that this is not required for nested classes.

Variable and method names. Variable and method names use lowerCamelCase: the first word is lowercase, and subsequent words are capitalized. For example: `graduationYear`, `idNumber`, `getStudentYear()`.

Generic type parameters. When writing a generic class, type parameters should **always be a single capital letter**. This is a nearly-universal Java style convention. Common choices are E (element), T (type), K (key), V (value), F and S (first/second for pairs).

```java
// Correct: single capital letter for type parameter
public class MyList<E> { ... }

// Not conventional: multi-character or lowercase type parameter
public class MyList<Element> { ... }
```

Access Modifiers

Instance variables. **All instance variables must be `private`.** You should never make an instance variable `public`. Instance variables should only be accessed from outside the class through getter and setter methods.

```java
public class Student {
    private String name;
    private int graduationYear;
    private int idNumber;
    ...
}
```

Methods. Methods that are part of the class's public interface—getters, setters, constructors, and other methods intended for use by other classes—should be `public`. Internal helper methods that are not meant to be called from outside the class should be `private`.

- **Public:** getters, setters, constructors, and any method designed for external use.
- **Private:** helper methods, or methods that perform "dangerous" internal operations (e.g., a method that swaps elements in a sorted list).

The `public` methods of a class are like its API: they define how the class is meant to be used. The `private` methods are the internal implementation details.

Getters and Setters

Instance variables should *only ever* be accessed through the methods of their own class. You should not directly read or modify another class's instance variables (e.g., writing `s1.name` from outside the `Student` class is incorrect). Instead, use *getter* and *setter* methods.

```java
public class Student {
    private String name;
    private int graduationYear;

    public String getName() {
        return name;
    }

    public void setName(String newName) {
        name = newName;
    }

    public int getGraduationYear() {
        return graduationYear;
    }

    public void setGraduationYear(int newValue) {
        graduationYear = newValue;
    }
}
```

By only exposing certain getters and setters, you control which data other classes can read or modify. This also allows you to add validation inside setters, ensuring the data stays valid.

Constructors

Constructors should be lightweight: they should only be used to initialize the state of the object. They should not do any further work beyond setting up instance variables.

If initializing the state of an object requires significant computation, that computation should be split off into a *helper method* that the constructor then calls.

Always initialize all instance variables in the constructor. Java assigns default values to uninitialized instance variables (e.g., `0` for numeric types, `null` for objects), but you should never rely on default values intentionally. Forgetting to initialize a variable will not cause an error—Java silently uses the default—which can lead to hard-to-find bugs.

```java
public class Student {
    private String name;
    private int graduationYear;

    public Student() {
        name = "";
        graduationYear = -1;
    }
}
```

Object Equality

Never use == to compare objects. The == operator tests whether two variables refer to the *same object in memory* (reference equality), not whether they store the same value. Instead, use the `.equals()` method.

```java
String s1 = "hello";
String s2 = "hello";

// Incorrect: may give wrong result
if (s1 == s2) { ... }

// Correct
if (s1.equals(s2)) { ... }
```

This is especially important for `String`s. Using == on `String`s will *usually* work due to a Java optimization, but not always—`.equals()` is always correct.

Note that this requirement is not just a matter of style; it is important for correctness.

Writing your own `equals()` method. When you write a class, you may need to implement your own `equals()` method. Java's data structures (e.g., `ArrayList`) use `.equals()` internally, so methods like `contains()` and `indexOf()` will not work correctly for your custom classes unless you implement it.

Most `equals()` methods follow this pattern: first use `instanceof` to check that the argument is the right type; if not, return `false`; if so, compare the relevant internal data.

```java
//equals method for WorkOfArt
public boolean equals(Object other) {
    if (!(other instanceof WorkOfArt)) {
        return false;
    }
    WorkOfArt otherWork = (WorkOfArt) other;
    // getReference() returns an int (primitive), so == is correct here
    return this.getReference() == otherWork.getReference();
}
```

What counts as equality for your class is up to you—define it in a way that makes sense for the underlying data.

Static Constants

When storing constants shared across all instances of a class, use `public static final`. Unlike instance variables (which must be `private`), static constants may be `public`. The `final` keyword ensures the value cannot be accidentally changed.

```java
public class WorkOfArt {
    private String title;
    private int reference;

    public static final int MIN_REFERENCE_NUMBER = 0;
    public static final int MAX_REFERENCE_NUMBER = 999999;
    ...
}
```

Static constants can be accessed directly via the class name, even before any objects of the class are created:

```java
System.out.println(WorkOfArt.MAX_REFERENCE_NUMBER); // 999999
```

Generics: No Raw Types

When using a generic class (such as `ArrayList` or `LinkedList`), you must **always specify a type**. Leaving out the type creates a “raw type,” which is not allowed in this course.

```java
// Correct
ArrayList<String> names = new ArrayList<String>();

// Not allowed: raw type
ArrayList names = new ArrayList();
```

Raw types suppress type-checking and can hide bugs. Always specify the generic type parameter.

Conditionals

Always use braces. Java allows you to omit braces when the body of an `if`, `while`, or `for` is a single line. In this course, you should **always include braces**, even for single-line bodies.

Always use parentheses with multiple logical operators. When combining && and || in a single expression, **always use parentheses** to make the intended grouping explicit. Operator precedence with logical operators can be subtle and counterintuitive; parentheses ensure your code does what you expect.

```java
// Correct: parentheses make the grouping clear
if (x > 0 && (isPrime(x) || isSquare(x))) {
    System.out.println("Interesting number: " + x);
}
```

`break` and `continue`. Use `break` and `continue` sparingly. Often the loop condition can be rewritten to make the intent clearer without these keywords. Before using either, consider whether refactoring the loop condition would make the code more readable. Given the option, it is better to have a loop exit because its condition becomes false than to use `break`. On the other hand, if refactoring the loop makes it much more complicated, or involves adding extra variables, `break` may be the better option.