CSCI 2101 Java Style Guide

Fall 2017

This document describes the required style guidelines for writing Java code in CSCI 2101. Guidelines are provided for four areas of style: identifiers, indentation, whitespace, and documentation. **You must follow the guidelines given here. There will be penalties for not doing so.** Please ask if you have any questions.

1 Identifiers

The classes, methods, and variables in your programs must all be given names, or *identifiers*. Identifiers should be *descriptive* – i.e., choose names that suggest what the class, method, or variable does and help a reader to understand how your program works. There are two sets of naming rules that you must follow. First, there are rules imposed by the Java language that must be followed in order to satisfy the compiler. Second, there are additional style guidelines that will not be automatically flagged by the compiler, but must be followed in order to receive full style marks in CSCI 2101. Here are the Java rules:

- The first character of an identifier must be a letter, an underscore (_), or a dollar sign ($).
- The rest of the characters in the identifier can be a letter, digit, underscore, or dollar sign. Note, in particular, that spaces are not allowed.
- Identifiers are case-sensitive. This means that `square` and `Square` are different identifiers.
- Identifiers cannot be the same as any of Java's *reserved words*:

`abstract`	`assert`	`boolean`	`break`	`byte`	`case`	`catch`	`char`	`class`
`const`	`continue`	`default`	`do`	`double`	`else`	`enum`	`extends`	`false`
`final`	`finally`	`float`	`for`	`goto`	`if`	`implements`	`import`	`instanceof`
`int`	`interface`	`long`	`native`	`new`	`null`	`package`	`private`	`protected`
`public`	`return`	`short`	`static`	`strictfp`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`true`	`try`	`void`	`volatile`	`while`	

Additionally, you are required in CSCI 2101 to use a *widespread style*, which does not use the dollar sign at all in identifiers and requires the following:

1. For variables and methods, use only letters and digits, separating words in the identifier by capitalizing the first letter of each word, *except for the first word* (this is called camelCase, e.g. `myCoolNumber`),

2. For constants, use all upper case letters, separating words with an underscore (e.g., `SOME_CONSTANT`),
3. For class names, use only letters and digits, separating words in the identifier by capitalizing the first letter of each word, *including the first word* (e.g., `MyFancyClass`).

1.1 Variables and Constants

Here are some additional guidelines on creating identifiers. Noun phrases are good choices for variables and constants that describe *things*. For example:

```
private static final int NUM_TEMPS = 10;
private static final double NORMAL_TEMP = 98.6;
private static final int LOW_TEMP = 90;
private static final int HIGH_TEMP = 110;
private static final int TEMP_RANGE = HIGH_TEMP - LOW_TEMP;

private Random rand = new Random();
private Scanner scan = new Scanner(System.in);
private Circle target;

private char reply;
```

Note that these are all variables and assignments as they would appear if declared globally (outside all methods). If they were declared locally (in a single method), you would not use the `private` or `static` modifiers.

Note that you do not need to use named constants for values that are obvious and general, typically values like 0 and 1. You should name a constant (and use the name) when it means something special, e.g. `NORMAL_TEMP`, the normal human body temperature.

Boolean variables often represent *conditions*, and their names should reflect that, e.g.:

```
boolean done;  // we're done or we aren't
```

Identifiers such as `x` and `y` should only be used if the value has no significance other than the fact that it is a real number, such as in a mathematical formula:

```
double y = a * x + b;
```

Identifiers such as `i`, `j`, and `k` are typically used to stand for arbitrary integer values, especially those used as loop control variables.

1.2 Methods

A method that returns a boolean value can often be named by the yes/no question that it answers, such as `isWithinRange` or `wantsToPlay`. Other times a method returns a non-boolean value (e.g. a numerical value) and the name of the method should indicate what that value is, as in `greatestCommonDivisor`. Finally, a method might not answer a question or return a value, but, instead, perform some job on its inputs. A natural choice for naming this kind of method is a description of what it does, such as `playGame` or `runCalculator`.

1.3 Classes

Classes should be named in a way that gives the reader some idea of what the class does. Since a class is intended to group data and methods that are logically related and implement some kind of entity, a name that describes this entity is frequently a good choice:

```
IntegerSorter
Calculator
SudokuSolver
```

2 Indentation

Each of line of code within a code block (class, method, loop, conditional, etc) must be intended with a consistently-sized tab (either an actual tab, or a fixed number of regular spaces). In other words, each "level" of the code must be consistently indented. Inconsistent indentation is one of the best ways to produce a program that is an unintelligible mess, even if it still works. **Points will be taken off for indentation that is not consistent or makes it difficult to follow what you are doing.**

A related issue is the placement of curly braces that enclose a block of code. The opening brace should be on the same line as the construct that starts that block (`if`, `else`, `while`, or `for`) and the closing brace should be lined up with the `if`, `else`, `while`, or `for`, e.g.

```
if (x < 20) {
    x = y + z;
    System.out.println(x);
}
```

Note that code examples in the Java Structures book use a different style, where the braces are on a separate line underneath the header, like so:

```
if (x < 20)
{
    x = y + z;
    System.out.println(x);
}
```

While I prefer the first style for compactness and encourage you to use it, if you wish to use the second style, you may do so, **as long as you are consistent**. Mixing styles will results in penalties. If you are working with a partner, make sure you agree on style conventions in advance! Changing the style of your partner's code after the fact is a good way to simultaneously waste time and annoy your partner.

Lastly, for conditional and loop blocks that only have a single line of code, braces are technically optional as specified by Java. For example, consider the following:

```
if (x < 0) {
    System.out.println(x);
}
```

The above is equivalent to this:

```
if (x < 0)
    System.out.println(x);
```

However, I **strongly caution** against any use of the second form. It is easy to create a program that both looks deceiving and has unintended bugs when omitting the braces. For example, consider the following:

```
if (x < 0)
    System.out.println("x is negative!");
    x++;
System.out.println(x);
```

The above program is syntactically correct, and looks like the `x++` is part of the conditional due to the indentation, but it's actually not since the braces are omitted (the `x++` will always run here, and ought not to be indented). This type of error will never occur if you always include the braces.

Therefore, while omitting the braces is not strictly forbidden, if you aren't sure, always include them. Points will be taken off for any use of the second form that results in a hard-to-read program.

3 Whitespace

Use whitespace (blank spaces and lines) freely to make your code easier to read. Observe the following rules:

- Don't put more than one statement on a line.
- Separate all methods with a blank line.

- Separate all instance variables with a blank line.
- Put blank lines between groups of statements that perform well-defined tasks (e.g. doing some calculations that are related to each other).
- Put spaces between operands and operators of expressions (e.g., *x = 0*, not *x=0*).
- Put spaces around keywords (e.g., `if (x > 0)`, not `if(x > 0)`).

4 Documentation

Documentation refers to comments placed in your program that explain what the program is doing. *We will take points off for insufficient documentation.* For this course, there are two kinds of documentation: summary documentation (for classes and methods) and in-line documentation (for code within methods). Summary documentation must follow Javadoc conventions, as described below.

4.1 Javadoc

Javadoc is a set of conventions for writing Java documentation, as well as a program that processes Java source code containing Javadoc comments. Writing your documentation using Javadoc both standardizes your comments and allows the automatic creation of nice, easy-to-read web pages describing all your classes and methods.

A Javadoc comment consists of a freeform comment starting with `/**` and a set of *tags*, which document specific pieces of information (e.g., a function parameter or a return value). There are a few specific tags that you should use, which are described below.

4.2 Class Summary

The file containing your class should begin with a Javadoc comment that includes the lab name and a summary description of what the class does, including any assumptions that the program makes about how it is going to be used (particularly if there are cases that might go against reasonable user expectations). Your class summary should include your name (and your partner's, if applicable) via a Javadoc `@author` tag. Note that the class summary comes just *before* the start of the class (as opposed to in Python, where the docstring comes just after).

For example:

```
/**
 *      Java Exercises
 *
 *      This program contains four exercises in Java programming:
```

```
 *          1) solving a quadratic equation
 *          2) changing the format of a name
 *          3) calculating body temperature differences from normal
 *          4) finding Fibonacci numbers
 *      The user can select any exercise from a menu as many times
 *      as they want.
 *
 *      Unusual cases for each exercise are noted in the documentation
 *      for that exercise.
 *
 *      @author Jason Bourne
 */
public class JavaExercises {
    ...
}
```

Initially, when your programs are quite simple, the class descriptions may be quite brief. As your programs increase in complexity, the length of the summary will increase as well.

4.3 Method Summaries

Methods must have summaries, similar to class summaries, just above their header statements. The summary should include a description of what the method does and any assumptions that the method makes about the inputs. Each parameter must be documented using a `@param` Javadoc tag (one tag per parameter), and for non-`void` methods, there must also be a `@return` tag documenting the return value of the method.

For example:

```
    /**
     * Solves quadratic equations that do not have complex roots
     * (roots with an imaginary component).  The equation solved
     * is assumed to be of the form a x^2 + b x + c = 0.
     *
     * @param a The first coefficient.
     * @param b The second coefficient.
     * @param c The third coefficient.
     * @return An array containing the two roots, or null if the roots
     *         would be zero or if the first coefficient is zero.
     */
    public static double[] quadraticRoots(double a, double b, double c) {

        double discriminant = (b * b) - (4 * a * c);
        // Make sure we will not be dividing by 0 or taking the
        // square root of a negative number.
        if (a != 0 && discriminant >= 0) {

            double sqrtDiscriminant = Math.sqrt(discriminant);
            double root1 = (-b + sqrtDiscriminant) / (2 * a);
            double root2 = (-b - sqrtDiscriminant) / (2 * a);

            // put the two roots of the quadratic in array and return
            double[] roots = { root1, root2 };
            return roots;

        } else {
            // Print error messages if necessary.

            if (a == 0) {
                System.out.println("error: coefficient a must be non-zero");
            }

            if (discriminant < 0) {
                System.out.println("error: discriminant must be >= 0");
            }

            return null;
        }
    }
```

4.4 Inline Documentation

Comments placed between lines or blocks of code in a program are called *inline* comments. Often it is helpful to use comments to explain blocks of code in your program. For example:

```
    // get the coefficients of the quadratic equation
    System.out.print("a = ");
    double a = scan.nextDouble();
    System.out.print("b = ");
    double b = scan.nextDouble();
    System.out.print("c = ");
    double c = scan.nextDouble();
```

Inline comments are generally helpful in all but the simplest programs, but don't go overboard! Not every line of code needs a comment, and comments should always add something to the code (if they just restate the code, they're not useful). Lastly, if a method is getting very long such that many inline comments are needed, you should consider whether it would be better to split the method into multiple shorter methods.