COMS 1004 Updated Style Guide

By: The 1004 Teaching Staffs

This is a short but important collection of coding standards that we expect you to follow for this class. You will find that adhering to this guide results in readable, maintainable, and easy to debug code and allows you to receive full points on all programming assignments. Beyond this class, you will find that good code style is essential not only for yourself but also for any large codebase engineered by a team of developers.

For reference most of the tips on this list were inspired by Google's Java Style Guide which we encourage you to spend 7 minutes reading. Multiple TAs over the generations (James Lin, Andrew Goldin, Madhavan Somanathan, Linan Qiu, Griffin Newbold etc) have contributed to this guide.

Table of Contents:

1. Source File Basics

1.1 Naming Your Files:

Your file name should consist ONLY of the name of the public class plus the .java extension.

Good Examples:

- MyClass.java
- Calculator.java
- Search.java

Bad Examples:

- myclass.java
- Calculator(1).java
- Search.txt

1.2 File Header:

A brief multi-line comment should be present in each .java file you create. The contents should include: your name, your UNI, file name, and a concise description of the file.

```java
/* Griffin Newbold
 * gcn2106
 * Dog.java
 * Simple program to model a Dog
 * /

public class Dog {
  //class details omitted
}
```

2. General Formatting

2.1 Styling Braces:

You will need to use braces any time you write classes, methods, conditionals, and loops. This includes the times when it is generally *optional*. We require it in all possible cases because it will help you read your code and it denotes logical scope.

There are two possible styles: K&R and Horstmann. Choose one and stick to it, do NOT switch between them in your code:

```java
// K&R
for(int i = 0; i < n; i++) {
  System.out.println(i);
}

// Horstmann
for(int i = 0; i <  n; i++)
{
  System.out.println(i);
}
```

2.2 Indentation:

Be sure to use consistent and proper indentation in your code. It can help denote scope and quickly allow you to identify what code goes in which code block. In this course you can choose one of the following indentation styles: tabs, 2 spaces, 4 spaces. For convenience tabs are the recommended choice for beginners.

```java
//Proper indentation visually indicates scope!

int i = 0;

while (i <  array.length) {
  double random = Math.random();

  if (random < 0.5) {
    array[i] = -i;
  } else {
    array[i] = i;
  }
}
```

2.3 Line Length:

When writing code, be sure to keep your lines at **80 characters or less**, anything exceeding this will need to be wrapped or broken up.

Keeping your lines small makes your code more digestible for yourself and yourself 4 months from now when you want to review it.

On Codio, at the bottom of the file there is a counter in the form (xx:yy) your lines meet the requirement if yy <= 80

```java
String str = "This is a really really long string" +
             "that really needs to be broken up";
```

2.4 Method Length:

When writing methods, they should be no longer than 14 lines of code long. This does NOT include: the method signature, braces, whitespace, comments.

Limiting the length of a method encourages you to think about what the purpose of the method is and to keep your method doing only one thing. Methods that are too long decrease readability and enable you to run the risk of writing repetitive code.

Java
```java
public class CircleCalculator{

      public static void main(String[] args) {
        double radius = 5.0;

        double area = calculateArea(radius);

        System.out.println("Circle Area: " + area);
      }

      // Helper method to calculate the area of a rectangle
      // We'll also be able to reuse this code at many spots!
      private static double calculateArea(double radius) {
        return Math.PI * Math.pow(radius,2);
      }
}
```

2.5 Using Whitespace:

Use whitespace to separate chunks of logic within methods and to separate methods within a class, it improves within your method. Be generous but not too generous.

Java

```java
/* In this example, having spaces helps this program denote these
 * distinct tasks:
 * (1.) Declaring initial values,
 * (2.) Calculation,
 * (3.) Displaying the final result and
 * (4.) Separation between methods
 */
public class WhitespaceDemo {

  public void computeTriangleArea() {
    int height = 5;
    int base = 12;

    float area = 0.5 * base * height;

    System.out.println("The area of the triangle is: " + area);
  }

  public void computeRectangleArea() {
    int width = 6;
    int length = 7;

    int area = width * length;

    System.out.println("The area of the rectange is: " + area);
  }
}
```

3. Naming Conventions

3.1 Class Names

You should give your classes names that are meaningful and representative of the problem they are trying to solve. They are written in UpperCamelCase and are typically nouns or noun phrases.

3.2 Variable Names

You should give your variables names that are meaningful and representative of the value they contain. They are written in lowerCamelCase and should be short and easily identifiable.

3.2.1 Denoting a Constant

Whenever you denote a variable with the final keyword you are declaring a constant, constants follow all the same rules as normal variables except they should be written in CONSTANT_CASE which is simply all uppercase words separated by underscores.

3.3 Method Names

You should give your methods short and meaningful names that relate to what they are implementing. Methods are like variables in that they are written in lowerCamelCase. Since methods denote some sort of action, verbs make good method names.

3.4 Avoiding Hard Coding Magic Numbers

Any value that has any sort of significance and repeated usage should be turned into a constant rather than hard coded directly.

```java
// Bad
public double weight(double mass) {
  return mass * 9.81;
}
```

```java
// Good
public static final double GRAVITY = 9.81;

public double weight(double mass) {
      return mass *  GRAVITY;
}
```

4. Documentation Overview

4.1 Comments:

Comments are made within a .java file, and these are meant for you to communicate ideas about lower level details regarding the code you write. Typically comments are either specific to a line of code or to a method, when writing comments for a method it is in your best interest to denote inputs, outputs, and other relevant details such as those. For line specific comments it could be commentating on a specific algorithmic choice you made or a simple note reminding you to fix something during development. You are expected to write comments in every file for every programming assignment.

4.2 README.txt files:

READMES are separate files with a .txt extension. These are generally used to communicate high level ideas regarding the code you've written, any external source that you reference during development and the overall status of your project. When developers work with new and unfamiliar projects, they turn to a README to understand at a high-level what the program accomplishes and instructions to operate and run the program. You are expected to submit a README.txt with every programming assignment.

5. Writing a README

In 1004 and beyond a lot of the CS courses you take will require you to submit a readme.txt file along with the programs you write. Not submitting a readme.txt file may result in a lower grade so it is important to submit one, but it is also important to understand why and how to write them.

The why:

A readme file is an important part of the software engineering cycle. For users of your program it is important to write short descriptions of the files you write to give users a sense of how the program functions and to give them an idea of how to use the functions you wrote. For the teaching staff, these descriptions are a great opportunity for you to write about concepts you didn’t fully understand so we can teach you better when you come to office hours. It also allows you to cite sources for methods you use that do not come directly from lectures or the Big Java textbook.

The how:

Writing a readme.txt file is simple. First create a readme.txt file on codio, by clicking the file menu and then click new file. Once you have a readme.txt file, you can begin typing away.

Here is a generic form of what we are looking for in a readme.txt:

```
Student’s Name
Student’s UNI

Outside Sources Used: (OH, Websites, Sunday Study Sessions etc)

Program1.java
short and succinct description

Program2.java
short and succinct description
```

Here is an example of what an actual README.txt could look like:

```
Griffin Newbold
gcn2106

Outside Sources Used: (Went to Gabbie and Melody’s OH, Java String toUpperCase() Method With
Examples - GeeksforGeeks)

CreditCard.java
Implemented functionality for checking and returning the validity of a credit card using the
conditions given in the assignment spec.
```