CSE 142/143 Unofficial Style Guide

Below, things in `GREEN` are GOOD; things in `RED` are to be AVOIDED.

Commenting

Comment well. Follow the commenting rules for header, method, field, and inside-method comments in the accompanying commenting guide.

Naming

Follow the following naming conventions for different types of variables.

Type of variable	*Formatting*	*Examples*
Class name	**Capitalized** (first letter of each word is uppercased)	`MyProgram` `CritterMain`

Method/variable name	**camelCase** (first letter lowercase, each new word uppercase)	`greatMethod1()`
		`String firstName`
		`int countOfX`

Class constant	**UPPER_CASE** (words separated by underscores)	`MY_CONSTANT` `MAX_SIZE`

Name your variables and methods well. Try to be descriptive while also being concise. Avoid generic names such as `x`, `func`, and `stuff` when a better name exists. Also try to avoid many variables with similar names, such as `temp1`, `temp2`, and `temp3`. Single letter names are almost always bad; however, there are certain situations, such as the `i` counter in a for loop, where single-letter variables are the best option.

If you have a variable that holds the name of a musician, for example, then look at the following table of possible names:

Variable	*Good or bad?*	*Why?*
`n`	Bad	Too short
`text`	Bad	Not descriptive
`theNameOfThePersonWhoWritesTheMusic`	Bad	Too long/complex
`musicanName`	Good	Descriptive, short

Formatting

- Keep lines under 100 characters in length, preferably under 80 characters. It is very difficult to read (and grade) code when it is too long. This can mean that you have to split up a line of code, or that you must put a comment on a line by itself rather than on the same line as the code it refers to.

- Leave a blank line between methods. Don't be afraid of the white space.

```
public void calculatePrime() {
     plusByOne();
}
                                  <-- Put a blank line here
public void plusByOne() {
   ...
```

- Put only one statement on a line.

```
Bad:  int x = 3; int y = 4;
      if (x == y) { foo(); }
Good: int x = 3;
      int y = 4;
      if (x == y) {
           foo();
      }
```

- Leave spaces around all operators (`+`, `-`, `*`, `/`, `=`, `&&`, `||`, `!=`, `>`, `>=`, `<`, `<=`). This makes your code much easier to read.

```
Good: int x = 4 + 5;
      boolean b = 4 > (4 * 2);
Bad:  int x=4+5;
      boolean b =4>(4*2);
```

- Leave no spaces before the parentheses in a method call.

```
Good: method1();
Bad:  method1 ();
```

- Leave a space before opening curly braces.

```
Good: public static void calculatePrimes() {
      if (true) {
Bad:  public static void calculatePrimes(){
      if (true){
```

- Indent every code block (the stuff inside any curly braces `{ }`). Indent after every `if`, `while`, and `for` statement even if you don't use curly braces. An indent is usually 3 or 4 spaces (choose one and stick with it).

```
public void addOdds(int times) {
     int sum = 0;
     for (int i = 0; i < times; i++) {
```

```
            if (times % 2 == 0)
                sum += i;
        }
    }
```

- You can omit the curly braces after an `if`, `while`, or `for` if there is only one statement inside it. However, I strongly discourage this. For consistency, always use curly braces. This also helps prevent bugs that occur when you try to add a statement to a `for`-loop and forget to add the curly braces.

- But overall: BE CONSISTENT. If for some reason you don't agree with my conventions (which you should - these are generally accepted), for goodness sake be consistent within your code!

Boolean zen

Booleans are cool because you don't always have to do all the testing with == and `!=` that you do with other types. Treat them like the `true` and `false` that they are. Remember that you can use `!` to negate a boolean value, as in example 2 below.

If you have declared

```
boolean isAlive;
```

then...

BAD	GOOD

`if (isAlive == true) {`
`  eatFood();`
`}`
	`if (isAlive) {`
	`  eatFood();`
	`}`

`return isAlive == false;`	`return !isAlive;`

`if (isAlive) {`
`  return true;`
`} else {`
`  return false;`
`}`
	`return isAlive;`

Variable/field guidelines

- Fields should always be `private`, unless specifically stated otherwise. However, class constants (which have the keywords `static final` in their declaration and cannot be changed) can be `public`.

Good:
```
private String name;
public static final String NAME;
```
Bad:
```
public String name;
```

- If a value doesn't change when your code runs, or you find a number that appears repeatedly in your code and doesn't change, declare a class constant and give it a name. This improves reusability.

- Avoid static global variables. For the *vast* majority of purposes, you will never need static global variables. Instead, use either fields or constants.

  Bad:
  ```
  public static String bad;
  ```

- Always declare fields above the first constructor. Initialize them in the constructor, not outside it.

  Good:
  ```
  public class Counter {
      private String name;
      private int count;
      public Counter(String givenName) {
          name = givenName;
          count = 1;
      }
  }
  ```

  Bad:
  ```
  public class Counter {
      private String name = "bob";
      private int count = 1;
      public Counter(String givenName) {
          name = givenName;
      }
  }
  ```

- Localize variables as much as possible. This means that if something is only needed in one method, it should be a variable in that method and not a field. Only data that is part of your object's state and are needed in several methods should be stored in fields. If a variable is only needed inside a loop, declare it inside the loop and not before.

  Good:
  ```
  public int addTwos(int times) {
      int sum = 0;
      for (int i = 0; i < times; i++) {
          int temp = 2 * times;
          sum += temp;
  ```

```
        }
        return sum;
}
```

Bad:

```
public int addTwos(int times) {
        int sum = 0;
        int temp;
        for (int i = 0; i < times; i++) {
                temp = 2 * times;
                sum += temp;
        }
        return sum;
}
```

Redundancy

Redundancy is the most subtle style issue, so here are a few guidelines for avoiding it. However, use your best judgment and common sense as a reliable guide.

- If you find yourself doing the same thing multiple times, make a method out of it and call it whenever you need it. Use parameters to generalize the code.

  Bad:

  ```
  public void printAll() {
        System.out.println("x = 500, y = 100");
        System.out.println("x = 400, y = 200");
        System.out.println("x = 300, y = 300");
        System.out.println("x = 200, y = 400");
        System.out.println("x = 100, y = 500");
  }
  ```

  Good: use a helper method with one parameter to capture the redundancy of the printed format "x = *something*, y = *something*"

  ```
  public void printAll() {
        printLine(500);
        printLine(400);
        printLine(300);
        printLine(200);
        printLine(100);
  }

  public void printLine(int x) {
        int y = 600 - x;
        System.out.println("x = " + x + ", y = " + y);
  }
  ```

- Even better, if you find a pattern, use a loop to reduce the number of manual method calls or operations you do.

  Great:
```
public void printAll() {
      for (int i = 500; i > 0; i -= 100) {
            printLine(i);
      }
}

public void printLine(int x) {
      int y = 600 - x;
      System.out.println("x = " + x + ", y = " + y);
}
```

- Don't over-use methods. A one-line method that is called only once is probably not useful. Make new methods only where it makes sense to divide your code for reuse or readability.

- Avoid redundancy in constructors: use the `this` notation to have one constructor call another. Probably only one constructor should be doing major work.

  Bad:
```
public class Counter {
      private String name;
      private int count;

      public Counter(String givenName) {
            name = givenName;
            count = 0;
      }

      public Counter(int currentCount) {
            count = currentCount;
            givenName = null;
      }

      public Counter(String givenName, int currentCount) {
            name = givenName;
            count = currentCount;
      }
}
```

  Good:
```
public class Counter {
      private String name;
      private int count;
```

```
    public Counter(String givenName) {
        this(givenName, 0);
    }

    public Counter(int currentCount) {
        this(null, currentCount);
    }

    public Counter(String givenName, int currentCount) {
        name = givenName;
        count = currentCount;
    }
}
```

Other

- Remove any debugging code from your program before you turn it in.
- Make helper methods private. The client does not need to see them.
- Do not use `break`, `break`-like, or `continue` statements. If you don't know what those are, then don't worry about it. These are not always bad, but they are very easy to misuse, so we don't allow them in CSE 14X.
    - Also do not use an empty return (`return;`) in a void method. This is basically like a break statement.
- The homework specifications are the ultimate guide on what we expect, so refer to them for any special requirements.