Introduction

This is a simple handout regarding Java programming style. There are many different opinions on what makes for proper programming style. This handout will guide you on the styles you should use when completing the assignments for this class.

What do these Style Guidelines cover? There are a variety of ideas that come under the umbrella of good programming style, including:

- The use of spaces, indentations, and blank lines
- Names for variables and functions
- Use of braces in control statements

What's the point? For the most part, this standard addresses items that the Java compiler does not care about. However, there are many good reasons why you need to use a consistent style when coding Java programs:

- It will make your program more readable, both for you and for others.
- It will help prevent programming bugs, and make debugging easier.
- It will prepare you for professional programming where your employer will most likely be using a coding standard, and you will be expected to follow it.

Caution Different people have different styles. Be careful if you decide to get help or ask questions from anyone outside of class, including friends, or students in other classes. Realize, too, that the text uses some different styles than we do. These guidelines follow those created by Sun Microsystems (`http://java.sun.com/docs/codeconv`). If you have a question about one of the style guidelines, ask me.

Variable and Object Names

You will be using variables and instance variables in your Java programs to store information for later processing and output, including integers, floating-point numbers, individual characters, and strings.

Good variable names	*Bad variable names*
`hoursWorked`	`x`
`payRate`	`I`
`grossPay`	`PyRt`
`firstName`	`grosspay`

Good variable and object names have the following qualities:

- The name of the variable reflects the data it stores. In the examples above, there can be little doubt as to what information is stored in `hoursWorked`, while the casual observer has no idea what's stored in a variable named `x`.

- The name of the variable or object uses whole words. You may be tempted to use abbreviations in your names, but they may become ambiguous, and abbreviations that make sense to you will not make sense to others (or to you if you look at your code three months down the road).
- The first letter of the variable name is lower case. The first letter of each subsequent word in the variable name is upper-case, and the rest of the letters are in lower case. Following this guideline will give visual clues to the reader as to where one part of your name begins and another ends — `grossPay` is easier to read than `grosspay`.
- Avoid the use of helper words (e.g., the, of, and). Don't use punctuation.
- Some programming languages (e.g., Visual Basic) encourage "Hungarian notation" where the first letter of the variable indicates the type (e.g. `fPayRate` – type is float, which begins with 'f'). This convention is not often used in Java and should be avoided.

Constants. Constants are used much like variables, but the contents of a constant cannot change. They are declared using the `final` keyword in Java. Constants are useful when we want to use the same value over and over again, but we don't want to write out that value each time. You should use constants frequently. Constants will follow the same naming conventions as variables, with the following exceptions:

- Use all capital letters when naming a constant
- Separate words in the name with a single underscore character.

Examples of good constant names include:

- `NAME_LENGTH`
- `DISCOUNT_RATE`

Method Names

Many of the same principles of good variable names hold for choosing method names, for the same reasons — you want to make your code as readable as possible. Computers can't understand commands in English, but by choosing good variable and function names, you can make your Java code read as close to English as possible.

Good method names	*Bad method names*
`calculateGrossPay`	`DoStuff`
`getPayData`	`BigFunction`
`displayOneAddressLabel`	`dispAddLbl`

Good method names have the following qualities:

- The name of the method reflects the action it does. Use whole words, and ensure one of the words you use is a verb. Make sure the action described is specific (as opposed to `doStuff`).
- Method names should use the same capitalization scheme that variables use.

As is the case for variables, use the most restrictive access modifier possible for methods. However, it is expected that methods will have less restrictive access than variables, especially because the variables will most likely be declared `private`.

Class and Interface Names

When naming classes, we want to apply the same overriding principle: creating readable code. A class name should describe the type of data that it stores or the type of operations available on an interface.

Good class/interface names have the following qualities:

- The name should be descriptive. Use whole words.
- The first character of each word in the name should be capitalized, including the first character of the class/interface name – e.g., `RetirementAccount`.
- If an interface exists to support a single method (named with a verb), it is common to create a name by appending the suffix "able" to the method name. For example, the `Runnable` interface declares a single method, `run`().

Control Statements

Control structures (`if, while, for`, etc.) have a number of rules in common. In the examples below, we talk about the `if` statement, but the same rules apply to the other statements as well.

Indentation. Always indent the body of an `if` statement. It makes the code much easier to follow, and it is crucial when there are multiple nested `if` statements.

Use of curly braces. While Java does not require it, we will always enclose the "then block" and the "else block" of an if statements in curly braces, even if either block is only one statement long:

Java allows this:	**But we will do this:**

`if (midtermGrade > 90)`
`  System.out.println("good job");`
`else`
`  System.out.println("not great");`
	`if (midtermGrade > 90) {`
	`  System.out.println("good job");`
	`} else {`
	`  System.out.println("not great");`
	`}`

This is a major difference between examples in many texts (including yours) and the way we will write our Java code. Using this technique will prevent errors.

With regards to the placement of the curly braces, there are two prevailing opinions on where the opening curly brace should be placed: on the same line with the `if`, or on a new line by itself.

Same line

```
if (midtermGrade > 90) {
   System.out.println("good job");
} else {
   System.out.println("not great");
}
```

Separate line

```
if (midtermGrade > 90)
{
   System.out.println("good job");
}
else
{
   System.out.println("not great");
}
```

Note that if the opening curly is placed on a separate line, it is placed in the same column as the 'i' in `if` – i.e., it is not indented. The closing curly is always on a line by itself and in the same column as the 'i'.

Either of these standards is acceptable, so long as you are consistent. Personally, I use the same line because it makes the code slightly denser, especially when we require curly braces for all `if` statements.

Indentation

Code enclosed in a pair of curly braces is known as a *block*. Blocks can be nested – i.e., you can have blocks within blocks. For example, nested `if` statements create nested blocks. In general, all statements within a block should be at the same level of indentation, and each nested block should be intended further than the enclosing block.

Bad – random indentation w/in a block

```
{
   x = 4;
        y = 40;
z=400;
}
```

Good

```
{
   x = 4;
   y = 40;
   z = 400;
}
```

Bad – no indentation of nested blocks

```
{
if (x == 7) {
y=20;
if (z > 10) {
g.drawString("good job");
}
}
}
```

Good

```
{
   if (x == 7) {
       y=20;
       if (z > 10) {
          g.drawString("good job");
       }
   }
}
```

Notice that in the code that lacks indentation of nested blocks you end up with a series of closing curly braces all at the same level of indentation. This is a flag telling you that something is wrong.
Hint: every time you type an opening curly, hit return and enter the closing brace immediately, and enter it at the right level of indentation. This will prevent you from forgetting closing curly braces and help you to get the indentation right the first time.

Variable Scope and Visibility

Always declare variables with the smallest scope and minimal visibility. Do not declare a variable as an instance variable if it is only used in one method; make it a local variable in the method. If a variable is only used in one small block within a method, declare it in that block. When declaring instance or class variables, use the most restrictive access modifier possible – i.e., use `private` instead of `protected` or `public`; use `protected` instead of `public`. In general, avoid default-access (no access modifier) unless there is some compelling reason to use it.

Import Statements

Do not use "* imports" – e.g., `import java.awt.*;` (Technically, these are known as type-import-on-demand declarations.) This is bad style. Given a class name used in the program, one cannot tell which package it came from. Instead, you should use `import java.applet.Scrollbar;` and `import java.awt.TextBox;`

Complex Expressions

Avoid complex expressions with multiple side-effects, especially within conditions of control structures. Java defines the order of execution of expressions, and the value and side-effect of pre- and post-increment and –decrement are well defined, but if a reader has to think hard to understand the code, that's a problem: such code is hard to initially debug and subsequently maintain.

For example, what are the values of `x` and `y` after the following statements?

```
int x = 4;
int y = --x - x++ * (x /= 2);
```

(The answer is that `x=2` and `y=-3`.)

Code like this is often written for one of two undesirable reasons: the mistaken idea that code that involves fewer character compiles or runs faster, or an effort of the writer to demonstrate his/her prowess in coding.[1] Break code up into multiple statements, possibly introducing temporary (local) variables to store intermediate results. Put pre/post-increment/decrement expressions in their own statements.

[1] I have seen C code with a comment along the lines of "you're not expected to understand this."