Java Style Guidelines

Formatting Code

These guidelines are for improving the readability of your code. They involve issues of indenting and spacing as well as how to choose good names for your variables and constants. While there is some choice, these guidelines should not be violated unless it cannot be helped.

Good formatting not only makes your code more readable, it is also beneficial during debugging. Finding missing or misplaced braces and parentheses is easier if your code is well formatted. You are advised to get into the habit of writing well formatted code early on so that it becomes second nature.

Indent nested code

The best way to make your code readable is to indent nested code. Nested code means code that is "inside" other code. One easy heuristic is to indent every time you go inside braces. Some code, such as if statements and for loops, use braces optionally. Even if you do not use the braces for these statement, consider the braces *invisible* and indent according to this rule anyway.

You may choose to indent by using tab or by using a certain number of spaces. The length of tabs can vary depending on your text editor so don't be surprised if your code looks different when you load it into different editors. If you use tabs, be careful to not mix tabs and spaces because printing might not look right. If you use spaces, indent using 2 or more spaces.

```
public class Example
{
...void someFunction (int arg)
...{
......if (arg > 10)
.........System.out.println("big");

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

Also indent after labels in switch statements even though the code after the label is not technically a separate block. This makes the labels easier to distinguish.

```
void someFunction (int arg)
{
...switch (arg)
...{
......case 1:
.........System.out.println("one");
.........break;
......case 2:
.........System.out.println("two");
...}
}
```

Use of braces

Many syntactic structures – such as class definitions and methods – require the use of braces. However other structures, such as if statements, and for loops do not require braces if the inner code is just one line long. It may be best to use braces even when they are not required. This makes it easier to identify what the body of the statement is and also makes it easier to add more lines of code to the inner block later on. Using braces at all times is an optional style consideration.

```
if (classSize > 140)
   return false;              // Braces unnecessary and not used.


if (classSize > 140) {
   return false;              // Braces unnecessary but used anyway.
}

if (classSize > 140) {
   System.out.println("Too many students!");
   return false;              // Braces necessary.
}

if (classSize > 140)
   System.out.println("Too many students!");
   return false;              // Logic error.  Return is not in the if body.
```

Location of braces

Locate the opening brace '{' of each block statement either as the last character on the statement line or on the next line by itself. This is a matter of personal preference, but if you place the brace on the next line, do not begin the inside code on the same line. Locate the closing brace '}' on a line by itself. The closing brace should be indented to line up with the first character of the block statement.

`public class Example {`
`   ...`
`}`
	`public class Example`
	`{`
	`   ...`
	`}`

`void someFunction (int arg) {`
`   ...`
`}`
	`void someFunction (int arg)`
	`{`
	`   ...`
	`}`

`for (int i=0; i < 10; i++) {`
`   ...`
`}`
	`for (int i=0; i < 10; i++)`
	`{`
	`   …`
	`}`

`switch (arg) {`
`   case 1:`
`      ...`
`   case 2:`
`      ...`
`}`
	`switch (arg)`
	`{`
	`   case 1:`
	`      ...`
	`   case 2:`
	`      ...`
	`}`

Use white space

White space consists of characters such as ‘space’ and ‘tab’ and ‘newline’ that do not appear in the text editor but take up space none-the-less. Using white space can make your code much more readable if it is used in between arithmatic.

```
//Bad style:
double d=(-b-Math.sqrt(b*b-4*a*c))/(2*a);

// Better style:
double d = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);

//Bad style:
if (x>LEFT&&x<RIGHT&&y>TOP&&y<BOTTOM)
   return true;

// Better style:
if (x > LEFT && x < RIGHT && y > TOP && y < BOTTOM)
   return true;
```

Break up long lines

If you are writing a statement that is too long, the editor might cause the line to wrap around to the next. Some text editors allow lines to be indefinitely long by scrolling horizontally. However, if the line is longer than 80 characters, a printer will either force the line to wrap or will cut off the end of the line.

Some lines of code can be broken up into several individual statements.

```
double length = Math.sqrt(Math.pow(Math.random(), 2.0) + Math.pow(
Math.random(), 2.0));     // Too long!

// Better style:
double xSquared = Math.pow(Math.random(), 2.0);
double ySquared = Math.pow(Math.random(), 2.0);
double length = Math.sqrt(xSquared + ySquared);
```

If this is not feasible, another possibility is to break the long line in strategic positions. If a method call is long and has several parameters, the line can be broken up at the comma between parameters. Relational statements, such as if statements with many conditions can be broken up just after the OR and AND operators.

```
double finalSolution = functionThatComputesTheAnswer (argument1, argument2,
argument3);      // Too long!

// Better style:
double finalSolution = functionThatComputesTheAnswer (argument1,
                                                      argument2,
                                                      argument3);

if (positionX >= LEFT_BOUNDARY && positionX <= RIGHT_BOUNDARY && positionY >=
TOP_BOUNDARY && positionY <= BOTTOM_BOUNDARY)     // Too long!
{
   return true;
}
```

```
// Better style:
if (positionX >= LEFT_BOUNDARY &&
    positionX <= RIGHT_BOUNDARY &&
    positionY >= TOP_BOUNDARY &&
    positionY <= BOTTOM_BOUNDARY)
{
   return true;
}
```

Use meaningful names

When declaring variables and constants, use names that accurately describe what the data is going to be used for. Class names should describe what the objects will do or represent. Method names should describe what they do. However, if a meaningful name becomes too long, then readability will also be diminished. Use multiple words to describe the meaning of your variables and methods if a single word would be ambiguous. If you use more than one word in a name, capitalize the first letter of each word except the first word. Do not use underscores '_'.

```
int heightOfSearsTower;
int heightOfSpaceNeedle;

void computeVelocityAtImpact (double height)
{
   ...
}
```

One exception to this rule is for variables used to iterate through loops. In the following example, the variable i is created only for iterating though the loop.

```
for (int i=0; i < 10; i++);
```

Avoid abbreviations in your variable and method names. The exception to this rule is for common words such as "number" and "seconds, "which could be abbreviated as "num" and "sec," respectively. If you abbreviate common words, make sure you abbreviate consistently.

Capitalize the first letter of class names.

```
public class Car
{
   ...
}

public class RaceCar extends Car
{
   ...
}

public class Formula1RaceCar extends RaceCar
{
   ...
}
```

Documentation and Commenting

Documentation and commenting is used to explain to yourself and others exactly what your code is trying to accomplish and how you are approaching the problem. Often times commenting *before* you write code can help organize your thoughts and make programming easier. Mostly, documentation is for the benefit of others who will read your code after it has been completed.

Documentation can be done during programming or after programming. You are advised to document your code as you write it because it is often the case that, after writing a large program, you have forgotten what your earlier code does and how it works.

Comment types

Java supports three different types of comments, *C-style* comments (first introduced in the programming language, C), *in-line* comments (first introduced in the programming language C++), and JavaDoc.

```
/* This is a single line C-style comment. */

/*
 * This is a multi-line
 * C-style comment.
 */

// This is a single line in-line comment.

// This is a multi-line
// in-line comment.

/**
 * This is a JavaDoc comment
 * @author Jason Schwarz (Jason_Schwarz@ncsu.edu)
 */
```

You must document each instance variable, method and class with JavaDoc style comments. You should use either C or in-line style comments inside of a method to document what you intend the code to do when executed.

Class documentation

Each class in your code must be preceded by documentation that describes what the class does and what it will be used for plus the author's name.

```
/**
 * The Circle class maintains information about the size, shape,
 * and area of a circle.
 * @author Alex Famous (Alex Famous@ncsu.edu)
 * @version 1.0
 */
 public class Circle
 {
    ...
 }
```

Method documentation

Each method in a class should be preceded by documentation that describes the parameters that will be passed in, what the method will do with that data, and what results will be returned.

```
/**
 * Set the radius of the circle.
 * @param radius new radius of the circle
 */
 public void setRadius (double radius)
 {
    ...
 }


/**
 * Get the radius of the circle.
 * @return radius of the circle
 */
 public double getRadius ()
 {
    ...
 }
```

Variable documentation

For every instance variable that is declared in your program, that declaration should be preceded by a JavaDoc comment that describes what the variable is for. This should be done even when the name of the variable is obvious.

```
/**
 * Radius of the circle
 */
 private int radius;
```

Internal comments

For long methods, it is sometimes useful to break the code up into chunks that serve a similar purpose and then comment that purpose. The internal documentation should precede the grouping of statements and should describe what they will accomplish. Break up groups with empty lines to improve readability.

```
Public void HelloWorldGUI ()
{
   // Initialize the window.
   super("Example");
   setLocation(200, 200);
   setSize(400, 200);

   // Initialize the label.
   lblHello = new JLabel("Hello, World!");
   lblHello.setFont(new Font("Helvetica", Font.BOLD, 48));

   // Install the label in the window.
   Container c = getContentPane();
   c.setBackground(Color.white);
   c.setForeground(Color.black);
   c.setLayout(new FlowLayout());
   c.add(lblHello);

   // Make sure the window is visible.
   setVisible(true);
}
```

Highly nested code

If your code is highly nested, then it may not be enough to merely line up the closing braces with the originating statement. This is especially true when blocks of nested code are longer than what you can see on the screen. In this case, even though the closing braces are indented properly, it may still be hard to tell which brace closes which block of code. When this happens, you should comment the closing brace to indicate which statement it is pared with.

```
for (i=0; i < 10; i++) {
   for (j = 0; j < i; j++) {
      ...
      if (j + i > SENTINEL) {
         ...
         switch (j + i) {
            ...
         } // switch
      } // if
   } // for j
} // for i
```