CS 1331 Style Guide - Spring 2012

Last updated January 12th, 2012

This document discusses proper Java style for the following:

Naming and Casing of Variables, Constants, Methods and Classes

Variables

Variable naming

Generally, variable names should be descriptive, but not unnecessarily long.

Improper Variable Naming	Proper Variable Naming
`String n;`	`String name;`
`double temp1, temp2, temp3;`	`double radius, diameter, area;`
`int totalNumberOfDartsLeft;`	`int dartsLeft;`

However, when a variable is used as a temporary counting variable, it is accepted to use a short variable name such as `int i;` or `int j;`.

Example scenario when it is appropriate to use a short variable name:

```
for (int i = 0; i < 5; i++) {
        ...
}
```

Variable Casing

If the variable name is made up of multiple words, the first letter of each word is capitalized – except for the first word. All other letters are lowercased.

If the variable name is only one word, all the letters are lowercased.

Improper Variable Casing	Proper Variable Casing
`double Height;`	`double height;`
`String firstname;`	`String firstName;`
`int ZipCode;`	`int zipCode;`

Constants

Constant naming

Constants should be named just like variables: descriptive, but not unnecessarily long.

Improper Constant Naming	Proper Constant Naming
`final double SALES_TAX_OF_CITY_OF_ATLANTA;`	`final double ATL_SALES_TAX;`

Constant casing

Every letter in constant names should be CAPITALIZED. If the constant name is multiple words, each word is separated by an underscore.

Improper Constant Casing	Proper Constant Casing
`final int MAXHEIGHT;`	`final int MAX_HEIGHT;`
`final Color myColor;`	`final Color MY_COLOR;`
`final double phi = 1.61803399;`	`final double PHI = 1.61803399;`

Methods

Method naming

Method names should be descriptive, but not unnecessarily long.

Improper Method Naming	Proper Method Naming
`public void blah()`	`public void calcArea()`
`public void finishTheCurrentGame()`	`public void endGame()`

Getters and setters

If a method is a getter (accessor) or a setter (mutator) for a class instance variable, there is a strict naming convention:

- Most getter method names start with “get” and then the name of the variable that is returned.

- Setter method names start with “set” and then the name of the variable to be set.

Example of a getter and setter:

```
public class Car {
      private int milesTraveled = 0;

      public int getMilesTraveled() {
            return milesTraveled;
      }

      public void setMilesTraveled(int milesTraveled) {
            this.milesTraveled = milesTraveled;
      }
}
```

- Getters for boolean variables should begin with "is".

```
public class Car {
      private boolean running;

      public boolean isRunning() {
            return running;
      }

      public void setRunning(boolean running) {
            this.running = running;
      }
}
```

Method casing

If the method name is only one word, all the letters are lowercased.

If the method name is made up of multiple words, the first letter of each word is capitalized, except for the first word. All other letters are lowercased.

Improper Method Casing	Proper Method Casing
`public int getmilestraveled()`	`public int getMilesTraveled()`
`public void Bet()`	`public void bet()`

Classes

Class naming

Class names should be descriptive, but not unnecessarily long.

Note: names for public classes must exactly match the corresponding filename of the .java file. For example: if the file is called “Dog.java,” the class within the file must be called “Dog.”

Class casing

For class names, the first letter of each word is capitalized. All other letters are lowercased.

Improper Class Casing	Proper Class Casing
`public class dog`	`public class Dog`
`public class paintcanvas`	`public class PaintCanvas`

Indentation, Brackets, and Line Length

Indentation

While Java doesn't require code to be indented in a specific way, it is good style for each block of code to be indented one level more than the preceding block.

Improper Indentation

```
public class Dog {
private String name;
private int hunger;

public Dog(String name) {
this.name = name;
hunger = 10;
}

public void eat() {
while (hunger > 0) {
hunger--;
bark();
}
}

public void bark() {
System.out.print("roof");
}
```

Proper Indentation

```
public class Dog {
     private String name;
     private int hunger;

     public Dog(String name) {
          this.name = name;
          hunger = 10;
     }

     public void eat() {
          while (hunger > 0) {
               hunger--;
               bark();
          }
     }

     public void bark() {
          System.out.print("roof");
     }
}
```

Brackets

Opening brackets can be placed on the same line or on a separate line. Either style is acceptable, however the same style should be used consistently throughout a project.

Example with opening bracket on the same line:

```
public class Dog {
     ...

     public void eat(int foodAmount) {
          while (hunger > 0 && foodAmount > 0) {
               System.out.println("Om nom nom nom.");
               hunger--;
               foodAmount--;
               System.out.println("Yum.");
          }
     }
}
```

Example with opening bracket on a separate line:

```
public class Dog
{
      ...

      public void eat()
      {
            while (hunger > 0 && foodAmount > 0)
            {
                  System.out.println(“Om nom nom nom.”);
                  hunger--;
                  foodAmount--;
                  System.out.println(“Yum.”);
            }
      }
}
```

Line Length

If a line of code is really long, it is good style to break it up into multiple lines. Generally, it is suggested to split up lines of code that is more than 80 characters long.

Example with multiple method parameters split up into multiple lines:

```
public class Dog {
      ...

      public Dog(String name, int age, String breed,
            String sex, String breed, Color coatColor,
            double weight) {
            ...
      }
}
```

Example with multiple variables split up into multiple lines:

```
public class ConvenienceStore {
      ...

      public int totalItemCount() {
            int total = sodaCount + breadCount + popsicleCount
                + candyCount + milkCount + pretzelCount
                + soupCount + bagOfChipsCount;
            return total;
      }
}
```

Comments

Whenever a piece of code inside a method is not obvious as to what it does, there should be an accompanying comment to clarify what the code accomplishes and how. Comments are used to prevent issues where a programmer (you or someone else) can't understand what a piece of code accomplishes.

Technically, comments can be placed anywhere in Java code. However, comments are most appropriate to describe code *inside* of methods and constructors. Comments shouldn't describe whole methods or classes – that is the purpose of Javadocs.

Code inside a comment is never compiled nor run at run-time. In addition to adding clarifying statements, comments are useful for debugging purposes.

Types of Comments

There are two three types of comments in Java:

- End-of-line comments, which begins with `//` and a single space, then the comment text continues on that line of code.
- Block comment, which are generally used for comments that occupy multiple lines. It begins with `/*` and then ends with `*/`. If there are multiple lines between the first and last lines of the block comment, there should be a `*` on each line and it should be horizontally aligned with the `*` on the first line. After each `*` there should be a single space before any text.
- Javadoc comments, which are used to describe entire methods or classes. See Javadocs section for more information about styling Javadocs.

Example of an end-of-line comment:

```
public void paint(Graphics g) {
        ...

        // draw a green square in the center
        g.setColor(Color.GREEN);
        g.fillRect(getWidth()/2 - 10, getHeight()/2 - 10, 20, 20);
}
```

Example of a block comment:

```
public void paint(Graphics g) {
      ...

      /*
       * Draw a rainbow in the center using
       * a big red arc, a medium green arc,
       * and a small magenta arc.
       */
      g.setColor(Color.RED);
      g.fillArc(xCenter - 75, yCenter - 75, 150, 150, 0, 180);
      g.setColor(Color.GREEN);
      g.fillArc(xCenter - 50, yCenter - 50, 100, 100, 0, 180);
      g.setColor(Color.MAGENTA);
      g.fillArc(xCenter - 25, yCenter - 25, 50, 50, 0, 180);
}
```

Writing Useful Comments

Comments can describe what some code is doing or why it's necessary, or possibly both, but you should not restate what is already obvious from the code to any reasonably knowledgeable programmer.

Useless Comment	Improved Comment and Improved Variable Names

`// add a, b and c and divide by 2`
`double s = (a + b + c) / 2;`
	`/*`
	` * Calculate the semiperimeter of`
	` * the triangle (semiperimeter is`
	` * half of the sum of all three`
	` * sides.`
	` */`
	`double semiperimeter = (side1 +`
	`side2 + side3) / 2;`

Additional Tips

- Don't excessively comment. Comment enough so it fully describes the code, but try to keep the comments brief.
- If your comment is on a separate line than the code you are commenting, indent your comment to the same indentation as the code.
- Don't write comments within comments.

Javadocs

Javadocs are a special type of comment used to describe and give important information about classes and methods. Where a regular comment is used to describe lines of code within a method, a Javadoc comment is written to describe and give important information about a class or a method.

Every class and method must be documented with a Javadoc. Even if the method seems obvious and simple, it should have a proper Javadoc with a brief description and all necessary tags.

All Javadocs should be indented to be aligned with the first character of the class or method. Javadocs begin with `/**` on the first line and end with `*/` on the last line. With every line between the first and last lines, there should be a `*` that is aligned with the first `*` of the first line. On each line, there should be a single space between the `*` and the text.

Javadoc Tags

Javadoc tags are predetermined keywords used to explicitly declare a definition of an important part of a class or method. Depending on the method or class being documented, there are specific tags that must be included in the Javadoc. Tags go within the Javadoc in a specific order, usually below the description.

Requirements and proper ordering of Javadoc tags:

Tag	Required in Class Javadocs	Required in Method Javadocs
`@author`		
`@version`		
`@param`		required for each parameter
`@return`		required if method doesn't return `void`

Class Javadocs

All class Javadocs must immediately precede the class header for which they describe. In particular, when there are import statements, the class Javadoc goes between the import statements and the class header.

Class Javadocs must have the following:

- a brief one-sentence description of the class, followed by an empty line, followed by a full paragraph description
- the author of the class, using the `@author` tag
- the version number of the class, using the `@version` tag

Example class Javadoc with a `@author` tag and a `@version` tag:

```
import java.util.*;

/**
 * The RightTriangle class represents a right triangle.
 *
 * It has two legs and a hypotenuse, with a right angle between the two
 * legs.
 *
 * @author Ryan Ashcraft
 * @version 1.0 03/14/2011
 */
public class RightTriangle {
     ...
}
```

Method/Constructor Javadocs

All method/constructor Javadocs must immediately precede the method header for which they describe.

Method/constructor Javadocs must have the following:

- a descriptive, yet brief description of the method/constructor
- a separate `@param` tag for each parameter of the method that includes the parameter name and a brief description of the parameter
- for method javadocs, if the method isn't `void`, a `@return` tag and a description of what the method returns

Example of a constructor Javadoc with two `@param` tags:

```
...
public class RightTriangle {
      private int leg1, leg2, hypotenuse;

      /**
       * Construct a right triangle with given length amounts
       * for the two legs.
       *
       * @param leg1 the first leg
       * @param leg2 the second leg
       */
      public RightTriangle(int leg1, int leg2) {
            this.leg1 = leg1;
            this.leg2 = leg2;
            hypotenuse = Math.sqrt(Math.pow(leg1, 2) +
                  Math.pow(leg2, 2));
      }
      ...
}
```

Examples of method Javadocs – the first includes a `@return` tag, the second includes a `@param` tag and a `@return` tag:

```
public class RightTriangle {
     ...

     /**
      * Returns the area of this right triangle.
      *
      * @return the area of this right triangle
      */
     public int area() {
          return leg1 * leg2 * 0.5;
     }

     /**
      * Return true if the provided argument is a RightTriangle
      * object, the first leg of the argument is equal to this
      * triangle's first leg, and the second leg of the argument
      * is equal to this triangle's second leg. Return false
      * otherwise.
      *
      * @param o the object to compare to
      * @return true if the argument is equal to this right triangle
      */
     public boolean equals(Object o) {
          if (o instanceof RightTriangle) {
               if ((RightTriangle o).getLeg1() == leg1
                         && ((RightTriangle o).getLeg2() == leg2) {
                    return true;
               }
          }

          return false;
     }
}
```