6.170 Laboratory in Software Engineering
Java Style Guide

Contents:

Overview

Coding style is an important part of good software engineering practice. The goal is to write code that is clear and easy to understand, reducing the effort required to make future extensions or modifications.

In 6.170 we do not specify a detailed coding style that you must follow. However we expect your code to be clear and easy to understand. This handout provides overall guidelines within which you should develop your own effective coding style.

Descriptive names

Names for packages, types, variables, and branch labels should document their meaning and/or use. This does not mean that names need to be very long. For example, names like *i* and *j* are fine for indexes in short loops, since programmers understand the meanings and uses of these variables by convention.

You should follow the standard Java convention of capitalizing names of classes, but starting method, field, variable, and package names with a lower case letter. Constants are named using all uppercase letters. The Java Language Specification provides some common Naming Conventions that you may want to use when naming your classes, methods, etc.

Consistent indentation and spacing

Indenting code consistently makes it easy to see where `if` statements and `while` loops end, etc. You should choose consistent strategies; for example, be consistent about whether you put the open curly brace on the same line as the *if* or on the next line, or what your try-catch-finally blocks looks like. Examine the code in the textbook for a sample style; feel free to develop your own if it makes you more comfortable.

In Emacs's Java mode, return indents the next line to (its guess at) the correct column, and the tab key re-indents the current line. There are also commands for re-indenting an entire region of lines.

In Eclipse Ctrl-F will correctly indent your code.

Consistent (and adequate) spacing also helps the reader. There is no reason to try to jam your code into as few columns as possible. You should leave a space after the comma that separates method arguments. You should leave a space between `for`, `if`, or `while` and the following open parenthesis; otherwise, the statement looks too much like a method call, which is confusing. In general, you should place only one statement on each line.

Informative comments

Don't make the mistake of writing comments everywhere -- a bad or useless comment is worse than no comment. If information is obvious from the code, adding a comment merely creates extra work for the reader. For example, this is a useless comment:

```
i++;    // increment
```

Good comments add information to the code in a concise and clear way. For example, comments are informative if they:

- *Enable the reader to avoid reading some code.* The following comment saves the reader the effort of figuring out the effect of some complicated formulas, and states the programmer's intention so the formulas can be checked later on.

  ```
  // Compute the standard deviation of list elements that are
  // less than the cutoff value.
  for (int i=0; i<n; i++) {
      // ...
  }
  ```

  An important application of this type of comment is to document the arguments and return values of functions so clients need not read the implementation to understand how to use the function.

- *Explain an obscure step or algorithm.* This is especially important when the effects of some step are not immediately obvious in the code itself. You should explain tricky algorithms, operations with side effects, magic numbers in the code, etc.

  ```
  // Signal that a new transfer request is complete and ready
  // to process.  The manager thread will begin the disk transfer
  // the next time it wakes up and notices that this variable has changed.
  buffer_manager.active_requests ++;
  ```

- *Record assumptions.* Under what assumptions does a piece of code work properly?

  ```
  // The buffer contains at least one character.
  // (If the buffer is empty, the interrupt handler returns without
  // invoking this function.)
  c = buffer.get_next_character();
  ```

- *Record limitations and incomplete code.* Frequently the first version of code is not complete; it is important to record which code is known to be incorrect. If you run out of time on an assignment and turn in a program that does not function correctly on all inputs, we will expect your code to show that you understand its limitations.

```
if (n > 0) {
    average = sum / n;
} else {
    // XXX Need to use decayed average from previous iteration.
    // For now, just use an arbitrary value.
    average = 15;
}
```

Hints:

- Don't write code first and then comment it -- comment it as you go along. It is easier to comment it while you are thinking about it and still remember its details, and you are unlikely to go back and do it later.
- We do not require you to write comments on every program object. However, your grade depends substantially on the clarity of your code, and some piece of the program that seems clear to you may not be clear to the reader. Therefore, we strongly recommend that you add explanatory comments to all classes, fields, and methods -- it will likely be to your advantage to do so.

Commenting code

In Java, `/*` and `*/` can be used to block off large chunks of code whereas `//` is used to comment out everything from a `//` to the end of a line. In practice, `/*` and `*/` should be reserved for Javadoc comments (see the next section), in which case the opening tag should have an additional asterisk: `/**`. If you need to highlight a large block of code, then highlight the region and use **Ctrl+/** in Eclipse. This is better because block comments do not nest. For example, if you already did:

```
String a = "Athena ";
String b = "bites";
/* String b = "brings me happiness"; */
String c = "closes? Nope. Never.";
String d = "doesn't have anywhere to sleep comfortably.";
```

But then you wanted to comment out the creation of variables `b` and `c` using a block comment, you would have:

```
String a = "Athena ";
/* String b = "bites";
/* String b = "brings me happiness"; */
String c = "closes? Nope. Never.";
*/
String d = "doesn't have anywhere to sleep comfortably.";
```

(The two block comment characters that have been added are in red and the code that is commented out by the new block comment is underlined.) Notice that this failed to comment out the statement where `c` is created. Also, this code will no longer compile because there is a `*/` dangling by itself after the

definition of `c`. This may seem easy to fix now, but if you have commented a large block of code, it may be a pain to find the nested block comment that is causing the compilation error. You can avoid this mess entirely by using the `//` comment:

```
String a = "Athena ";
// String b = "bites";
// // String b = "brings me happiness";
// String c = "closes? Nope. Never.";
String d = "doesn't have anywhere to sleep comfortably.";
```

This also makes it easier to uncomment smaller blocks of commented regions. Use **Ctrl**+\ to uncomment code in Eclipse.

TODO comments

If you want to leave yourself a note about a piece of code that you need to fix, preface the comment with TODO. You will notice that TODO will appear in bold and that if you do Window > Show View > Tasks, then a a "Tasks" pane will come up with all of the things you have left yourself TODO. You can jump to these points in your code quickly by double-clicking on them in the "Tasks" pane.

6.170 Javadocs

At a minimum, every class or interface that you write, along with every nontrivial public method you create, should have a Javadoc comment. Good documentation is complete without being excessive, so try to be succinct, but unambiguous, when documenting your code.

`HelloWorld` has a several Javadoc comments: at the top of the file, before the class declaration, before the `greeting` field, and before each method. You can tell that these are Javadoc comments because of where they appear in the code and because they start with `/**` instead of `/*`.

It is important to use this syntax to document your code so that your comments will appear in the HTML documentation generated by the Javadoc tool (this is how the documentation for Sun's API is generated, as well). There are a number of Javadoc Tags that get formatted in a special way when the HTML documentation is generated. You can identify these tags because they start with the @ sign, such as **@param** and **@return**.

For 6.170, we use a few additional Javadoc tags that are not recognized by Sun's Javadoc tool: **@requires**, **@modifies**, and **@effects**. Fortunately, Sun's Javadoc tool is extensible, so we have extended it with our own **6.170 doclet**.

You will generate the Javadoc for your code by running

```
ant javadoc
```

which invokes an Ant script that invokes the **javadoc** command with our 6.170 doclet. Ant is a cross-platform build tool that is similar to the Unix *make* tool. A build tool can be used to create scripts that automate tasks that are done frequently, such as compiling code or generating documentation. Because

Ant is a cross-platform build tool, it takes care of platform differences, such as the use of backslashes in Windows paths and the use of forward slashes in paths on every other platform.

We have created an Ant script for you, and its contents are in `build.xml`. You can look at this file if you like, or you can just do the following to generate the documentation for your code:

1. Right-click on `build.xml` in Package Explorer
2. Select **Run Ant...**
3. Toggle the checkboxes so that **javadoc** is the only box selected
4. Click **Run**
5. After you see a **BUILD SUCCESSFUL** message in the console at the bottom of the screen, right-click on `doc` in Package Explorer
6. Select **Refresh**
7. Open the `doc/api` folder and double-click `index.html`. If this opens index.html inside the text editor instead of a web browser, then open your web browser and point it to the `index.html` file to view it: `/mit/$USER/6.170/ps0/doc/api/index.html`. Alternatively, explore **Window >> Preferences >> General >> File Associations** to set the default application to open `.html` files in Eclipse. In the **File Associations** preference panel, choose the `*.html` file type and add `/usr/athena/bin/mozilla` as an **Associated editor**.

You should now be looking at your generated Javadoc in a web browser.

When someone else (such as your TA) is trying to understand your Java code, he or she will often first look at the generated Javadoc to figure out what it does. Thus, it is important that you check the generated HTML yourself to ensure that it clearly and accurately communicates the contracts of your code.

To demonstrate that you have successfully generated your Javadoc, edit the contents of the `package.html` file inside the `ps0` package so that the value between the `<body></body>` tags is a description of where the text in `package.html` appears in the generated Javadoc. Specifically:

1. Run the Javadoc command above to generate the Javadoc.
2. Look at the content of `package.html`.
3. View the generated Javadoc in a web browser.
4. Find where the content of `package.html` appears in the rendered HTML page (look closely – it appears twice!).
5. Replace the contents of `package.html` with a description of where the text in `package.html` will appear.
6. With the updated `package.html` file, regenerate the Javadoc and reload it in your web browser to make sure that it makes sense.

Again, the purpose of this exercise is to verify that you have generated and examined the Javadoc. We could have verified this by having you commit the Javadoc to CVS; however, committing generated files to CVS is bad style.