CSE 331 Software Design and Implementation

Handout C3: ***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 CSE 331 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.

Many other style guides are available. For example, you can see Oracle's Code Conventions for the Java Programming Language or Michael Ernst's document about coding conventions, which complements this document. We do not require you to follow those guidelines slavishly - they are just one way to write your code in a comprehensible fashion - but you might consider them while developing your own style. Even more valuable than coding style guides are descriptions of good ways to design and write code. For Java programmers, I highly recommend Josh Bloch's book *Effective Java*.

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. Also see *Effective Java* item #56.

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 look like. Examine the staff-supplied code, and the code printed in books, for sample styles; feel free to develop your own if it makes you more comfortable.

In Eclipse, `Ctrl-F` will indent your code, according to its built-in indentation rules (which you may or may not like).

Consistent (and adequate) horizontal spacing also helps the reader. There is no reason to try to save a column or two by eliminating spaces. 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. Remember that people reading your code may have a monitor of a different width, or may choose to print your code for a code review (the TAs will do this!). 80 columns is a commonly-accepted width in industry, and we require this for the convenience of the TAs when marking up your printouts. A longer line once in a while is acceptable, but not as a general rule. When you break lines, do so at logical, not arbitrary, locations.

Code files should never contain tab characters. They format differently in different IDEs, when printed, etc. A decent IDE should not insert tab characters in code files, or at least should have a setting that uses spaces instead. In Eclipse, do the following:

a. Go to Window > Preferences > Java > Code Style > Formatter
b. Create a new profile (if you haven't before -- you can't change the default Eclipse profiles, and they use tabs)
c. Set "Tab Policy" to Spaces only
d. Ctrl+Shift+F will reformat the current file to these settings
e. You can temporarily enable `General > Editors > Text Editors > show whitespace` to verify that it converts the tabs to spaces.

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 that would only help someone who does not know the programming language:

```
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. In fact, it's best to comment the code *before* you write it - doing so may expose weaknesses in your ideas and save you time on the implementation.
- 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, especially public ones - it will likely be to your advantage to do so.

Javadoc comments

Every class, every interface, every public method and field, and every nontrivial non-public method and field, should have an explanatory Javadoc comment. (Javadocs are useful even on non-public members, for two reasons. First, Javadoc has a special option that causes it to output documentation for all members, including private ones. You would never supply this to your clients, but it sure is helpful for the development team to have this handy. Second, an IDE such as Eclipse displays the Javadoc for a member when you hover over a use, and this is as useful for private as for public members.)

Good documentation is complete without being excessive, so try to be succinct, but unambiguous, when documenting your code.

Comments should be grammatical English. If more than a few words, a comment should consist of complete sentences that start with a capital letter and end with a period.

`HelloWorld` from HW0 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 Oracle'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 CSE 331, we use a few additional Javadoc tags that are not recognized by Oracle's Javadoc tool: **@requires**, **@modifies**, and **@effects**. See "Using javadoc to generate specs" for more information.

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.

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.

Commenting out code

Sometimes, you want to temporarily or permanently comment out some code.

Java has two ways to write comments: `/*`...`*/` comments out everything between the comment delimiters, and `//` starts a comment that ends at the end of the line. You should use `//` to comment out code. Reserve `/*`...`*/` comments for Javadoc comments, in which case the opening tag should have an additional asterisk: `/**`.

In Eclipse, you can comment out a block of code by highlighting the region and pressing **Ctrl+/**. Use **Ctrl+\** to uncomment code in Eclipse.

The rest of this section explains why you should use line comments such as `//` instead of block comments such as `/*`...`*/`: because block comments do not nest. For example, if you already did:

```
String a = "The HUB "; 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 = "The HUB ";
/* 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 = "The HUB ";
// 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.

Types

Generics

A type parameter must be supplied whenever a generic type is used. Never use a so-called "raw" type such as `List`, but instead use `List<Integer>` or the like.

All code must compile without warnings using `javac -g -Xlint`, without use of `@SuppressWarnings` except as permitted below.

Type casting

As a general rule, you should never have type casts in code you write (especially that for CSE 331). Casts are a work-around that hides information from the type system and prevents the compiler from flagging real bugs in your code.

However, there are some over-broad legacy interfaces in the Java library that require the use of casts. These were, in general, written in Java's dim past before it had the powerful type system it has today - some of these interfaces are quite widely used, however. Implementing these "over-broad" interfaces is the **only** time type casts should appear in your code.

The following is a fairly complete list of interfaces in the Java API that require casts. Unless you are implementing one of these, there should probably be no type casts in your code:

All implementations of:

- `Object.equals()`

Some (not all) implementations of:

- `Object.clone()`
- `Collection.contains()` (and subclasses of Collection, like `List` and `Set`)
- `Collection.remove()` (and subclasses)
- `Map.containsKey()`
- `Map.containsValue()`
- `Map.get()`
- `Map.remove()`
- `JComponent.paintComponent` (casting the `Graphics` object is encouraged in the official Oracle tutorial)

These may require *unchecked* casts (even worse!):

- `SortedMap.containsKey()`
- `SortedMap.containsValue()`
- `SortedMap.get()`
- `SortedMap.remove()`
- `SortedSet.contains()`
- `SortedSet.remove()`

CSE 331 requires that your code compile cleanly when `javac` is run with the `-Xlint` flag. You may use the `@SuppressWarnings("unchecked")` annotation to suppress warnings about unchecked casts in the last list of methods above. You may also use `@SuppressWarnings("nullness")` and `@SuppressWarnings("keyfor")` in certain uncommon situations when using the Nullness Checker. Additionally, if the official Oracle Java tutorials tell you to use a cast, it's OK. Other than that, you may not use `@SuppressWarnings` in your code. And, whenever you use one, do document your justification.

Local variables

Local variables should have the smallest possible scope. For example, declare it at its first; don't declare it at the beginning of the method and then have a lot of unrelated code intervening between its declaration and its first use. As another example, if you have a variable that is used within each loop iteration, declare it inside the loop to make clear to readers that there are no loop-carried dependencies.

Also see *Effective Java* item #45.

Documenting Code

Specification-level comments

Abstract data types. Every abstract data type (class or interface) should have:

a. An **overview** section that gives a one or two line explanation of what objects of the type represent and whether they are mutable.

b. A list of **specification fields**. There might be only one; for example, a set may have the field *elems* representing the set of elements. Each field should have a name, a type, and a short explanation. You may find it useful to define extra **derived fields** that make it easier to write the specifications of methods; for each of these, you should indicate that it is derived and say how it is obtained from the other fields. There may be **specification invariants** that constrain the possible values of the specification fields; if so, you should specify them.

Method Specifications. All public methods of classes should have specifications; tricky private methods should also be specified. Method specifications should follow the *requires, modifies, throws, effects, returns* structure described in the specifications handout and in class. Note that for CSE 331, you may assume arguments are non-null unless otherwise specified.

Implementation-level comments

Implementation notes. Class comments should include the following elements (which, for notable classes, appear also in the Runtime Structure section of the design documentation):

a. An **abstraction function** that defines each specification field in terms of the representation fields. Abstraction functions are only required for classes which are abstract data types, and not for classes like exceptions or some GUI widgets.
b. A **representation invariant**. RIs are required for any class that has a representation (e.g., not most exceptions). We strongly recommend that you test invariants in a `checkRep()` method where feasible. Take care to include in your invariants assumptions about what can and cannot be null.
c. For classes with complex representations, a note explaining the **choice of representation** (also called the **representation rationale**): what tradeoffs were made and what alternatives were considered and rejected (and why).

Runtime assertions. These should be used judiciously, as explained in lecture. For a longer discussion of the how runtime assertions can improve the quality of your code, see *Writing Solid Code* by Steve Maguire, Microsoft Press, 1995.

Comments. Your code should be commented carefully and tastefully. Stylistic guidelines are available in the handout Java Style Guide. For an excellent discussion of commenting style and for much good advice in general about programming, see *The Practice of Programming* by Brian W. Kernighan and Rob Pike, Addison-Wesley, Inc., 1999. A short and C-focused but helpful discussion is Six ways to write more comprehensible code: How to keep your code from destroying you from IBM's DeveloperWorks website.

Back to Conceptual Info
Back to the CSE 331 home page.