Achieving Readability in Java

This handout pieced together from Nick and Julie's archives.

A program is undoubtedly read many more times that it is written. A program must strive to be readable, and not just to the programmer who wrote it. Any program expresses an algorithm to the computer. A program is clear or "readable" if it also does a good job of communicating that algorithm to a human.

Readability is vital for projects involving more than one person. It's also important when the program is of sufficient size that you come across pieces of code you wrote, but which you don't remember. It's a traumatic experience the first time it happens. A bug is essentially a piece of code which does not say what the programmer intended. So readable code is easier to debug since the discrepancy between the intention and the code is easier to spot.

Documentation

Given that most programming languages are a rather cryptic means of communication, an English description is often needed to understand what a program is trying to accomplish or how it was designed. Comments can provide information which is difficult or impossible to get from reading the code. Some examples of information worth documenting:

- General overview. What are the goals and requirements of this program? this method?
- Data structures. How is data is stored? How is it ordered, searched, accessed?
- Design decisions. Why was a particular class design or algorithm chosen? Were other strategies were tried and rejected?
- Error handling. How are error conditions handled? What assumptions are made? What happens if those assumptions are violated?
- Nitty-gritty code details. Comments are invaluable for explaining the inner workings of particularly complicated (often labeled "clever") paths of the code.
- Planning for future. How might one might make modifications or extensions later?
- And much more... (This list is by no means exhaustive)

In a tale told by my office-mate, he once took a class at an un-named east-bay university where the commenting seemed to be judged on bulk alone. In reaction, he wrote a Pascal program which would go through a Pascal program and add comments. For each function, it would add a large box of *'s surrounding a list of the parameters. Essentially the program was able to produce comments about things which could be obviously deduced from the code. The fluffy mounds of low-content comments generated by the program were eaten up by the unimaginative grader.

Although this handout has a lot of information about suggesting commenting styles, in truth, the best commenting comes from giving types, variables, functions, etc. meaningful names and writing your code so cleanly to begin with that the code already explains itself. Add in a few comments where things still need to be further clarified and you're done. This is far preferable to a large number of low-content comments.

Program Overview Comments

Every program (or main class in a Java program) should begin with an overview comment. The overview can be the single most important comment in a program since it's the first thing that anyone reading your code will read. The overview comment explains, in general terms, what strategy the program uses to do its work. The overall program overview should lay out a road map of how the algorithm is designed— pointing out the important classes and discussing the data

behaviors. The program should mention the role of any other classes which the program depends on. Essentially, the overview contains all the information which is not specific or low-level enough to be in a class or method comment, but which is helpful for understanding the program as a whole.

In the latter paragraphs of the overview, you might include the engineering rational for the classes constructed or the algorithm chosen and discuss alternate approaches. The overview can also introduce the programmer's opinions or suggestions. It's often interesting to see the programmer's feelings on which parts of the program were the hardest or most interesting, or which parts most need to be improved.

For coursework, the overview should also include uninteresting but vital information like: your name, what class the program is for, your grader, and when the program is being handed in. In commercial code, the overview will also list, most recent first, all the revisions made to the code with author and date.

Class Comments

Each Java class deserves its own class overview which explains the class's purpose and design. It should explain what data is encapsulated by this class and what operations the class offers, as well as identifying any other classes it is related to or depends on. It may provide sample use of the operations to show how a client might access and manipulate this object. It's also a place to explain the design choices you made (why you used an array instead of a Vector or why certain operations are private and so on). Most of the built-in Java classes are very good examples of the type of documentation that is appropriate. For example, here is an excerpt from the class overview comment for Java's built-in String class:

```
/**
 * The String class represents character strings. All
 * string literals in Java programs, such as "abc", are
 * implemented as instances of this class.
 *
 * Strings are constant; their values cannot be changed after they
 * are created. String buffers support mutable strings.
 * Because String objects are immutable they can be shared. For example:
 *
 *     String str = "abc";
 *
 * is equivalent to:
 *
 *     char data[] = {'a', 'b', 'c'};
 *     String str = new String(data);
 *
 * Here are some more examples of how strings can be used:
 *
 *     System.out.println("abc");
 *     String cde = "cde";
 *     System.out.println("abc" + cde);
 *     String c = "abc".substring(2,3);
 *     String d = cde.substring(1, 2);
 *
 * The class String includes methods for examining
 * individual characters of the sequence, for comparing strings, for
 * searching strings, for extracting substrings, and for creating a
 * copy of a string with all characters translated to uppercase or to
 * lowercase.
 * ....
```

Choosing Good Identifiers

The first step in documenting code is choosing meaningful names for things. This is potentially the last step, since code with good identifiers often requires little additional commenting. For classes, local variables, parameters, and instance variables names the question is "What is it?" For methods, the question is "What does it do?" A well-named variable or method helps document all the code where it appears. By the way, there are approximately 230,000 words in the English Language— "temp" is only one of them, and not even a very meaningful one.

Naming classes

Naming a class in Java is like naming a type in C. You want the name to indicate the concept being encapsulated in this object. In many cases, when you are creating objects to map to real world analogs, you just use the name of that analog: Card, Deck, Account, Date, Button, etc. Some classes are not quite that concrete, and so you choose a name that tries to capture the essence of what the object is as best you can: Math or DateFormatter.

When naming a set of related classes, you often name the base class the plain form of the name, such as Exception, and then construct names for the more specialized versions by adding adjective phrases like OutOfBoundsException and ArithmeticException. The trend is toward have long descriptive class names without any cryptic abbreviations.

In Java, the convention is to capitalize the first letter of the class name, as well as letters of any following words in the name.

Common Idioms for Variables

There are a couple variable naming idioms that are so common among programmers, that even short names are meaningful, since they have been seen so often.

`i, j, k`	Integer loop counters.
`n, len, length`	Integer number of elements in some sort of aggregation
`x, y`	Cartesian coordinates. May be integer or real.

Nouns for Variables and Types

The uses of the above are so common, that I don't mind their lack of content. However, in all other cases, I prefer identifiers that mean something. Avoid content-less, terse, or cryptically abbreviated variable names. Names like "a", "temp", "nh" may be quick to type, but they're awful to read. Choose clear, descriptive labels: "average", "height", or "numHospitals". If a variable contains a list of floats which represent the heights of all the students, don't call it "list" or "floats", call it "heights". Plurals are good for variables which contain many things. This applies to names for instance variables and structure members as well as local variables.

Don't reiterate the data structure being used. e.g. `list`, `table`, `array` are really useless names when you think about it. Use something like `names`, `locations`, `scores`, etc. which indicates what data is being stored rather than the kind of storage used.

Don't reiterate the types involved if you know more specifically what the value is. e.g. `number`, `string`, `floatValue`, anything containing the word `Value` or `Temp`, are all worth avoiding. Try `numPeople`, `response`, `averageScore`, etc.

Do say what value is being stored. Use the most specific noun which is still accurate. e.g. `height`, `pixelCount`, `names`. If you have a collection of floating point numbers, but you don't know what they represent, then something less specific like `floats` is ok.

Defining Constants and Macros

Avoid embedding magic numbers and string literals into the body of your code. Instead you should define a symbolic name to represent the value. This improves the readability of the code and provides for localized editing. You only need change the value in one place and all uses will refer to the newly updated value. In Java, you do this by creating `final` variables.

Names of constants should make it readily apparent how the constant will be used. `MaxNumber` is a rather vague name: maximum number of what? `MaxNumberOfStudents` is better, because it gives more information about how the constant is used. You also may want to choose a capitalization scheme that identifies constants as such. The Java built-in classes tend to export constants in all upper case e.g. `PI, THURSDAY`.

Verbs for Method Names

Method names should clearly describe their behavior. Methods which perform actions are best identified by verbs , e.g. `addElement`, `shuffle`, `draw, setLength.` Predicate functions and functions which return information about a property of an object should be named accordingly: e.g. `isPrime`, `length`, `atEndOfLine`. You should stick with the standard Java capitalization scheme for method names which is to not uppercase the first letter, but to uppercase the start of each new word thereafter in the name.

Comments for Functions and Methods

The comments for a method need to address two different things:

1) What does it do?	(Abstraction comments)
2) How does it do it?	(Implementation comments)

The abstraction is of interest for someone who wants to use it. The implementation is of interest to someone trying to modify or debug it. A method with a well-chosen name and well-named parameters may not need any abstraction documentation. A routine where the implementation is very simple may not need any implementation documentation.

Abstraction Comments

For abstraction comments, you are telling the client how to use the method. Abstraction documentation is like an owner's manual— what the method does and what it can and cannot tolerate as input. One way to come up with a good abstraction comment is to look at the parameters of the routine, and then explain what the routine does to them. You should describe any special cases or error conditions the function handles (e.g. "...will abort if divisor is 0", or "...returns the constant NOT_FOUND if the word doesn't exist"). The abstraction comment also lists the possible exceptions thrown by this method and describes under what conditions. It is not necessary or appropriate to go into the gory details of how the method is implemented when creating abstraction commenting.

Implementation Comments

Specifics on the inner workings of a method (algorithm choice, calculations, data structures, etc.) should be included in the implementation comments, where your audience is a potential implementor who might extend or re-write the method. Implementation commenting is part is the traditional programmer-to-programmer documentation which describes how the code implements the abstraction.

The following examples were selected from the built-in Java classes to illustrate method documentation. They are not necessarily a representation of how much you should comment every method. The amount of commenting a method requires is related to its complexity and importance. Some programs break down so nicely that no one method is very complex, and the names of the method document most of what's going on. Other programs have a fundamental complexity which emerges in a few key operations. These routines deserve a lot more commenting.

In Java, since there are no header files separate from the source files, both types of comments will appear in the one source files. Some people like to label the "abstraction" and "implementation" documentation and write them as separate paragraphs. Some don't use the labels and document the routine in a single paragraph which addresses both. Some people prefer to defer more of the implementation discussion to inline comments. Please use whatever style you are most comfortable with. Any reasonable, consistent approach is acceptable.

Here is an abstraction comment from the String indexOf method. It explains what the method does, identifies each of the parameters, and then gives a little more information about the search strategy and expected return values. The implementation comments (which are sparse) were sprinkled through the code as inline comments.

```
/**
 * Returns the index within this string of the first occurrence of the
 * specified substring, starting at the specified index.
 *
 * str         the substring to search for.
 * fromIndex   the index to start the search from.
 *
 * If the string argument occurs as a substring within this
 * object at a starting index no smaller than
 * fromIndex, then the index of the first character
 * of the first such substring is returned. If it does not occur
 * as a substring starting at fromIndex or beyond,
 *  -1 is returned.
 */
public int indexOf(String str, int fromIndex)
```

The following comment from equalsIgnoreCase gives a general description of the flow of the method's algorithm using a more implementation-type comments. The comment takes advantage of the natural breakdown provided by the three cases of the testing. It doesn't get into the detail of individual lines— instead it outlines the general flow. Almost all routines have some sort of natural decomposition into "cases" or "phases" which can be a good starting point for the documentation.

```
/**
 * Compares this String to another object.
 * The result is true if and only if the argument is not
 * null and is a String object that represents
 * the same sequence of characters as this object, where case is ignored.
 *
 * Two characters are considered the same, ignoring case, if at
 * least one of the following is true:
 *
 *   The two characters are the same (as compared by the ==
 *     operator).
 *   Applying the method Character.toUppercase to each
 *     character produces the same result.
 *   Applying the method Character.toLowercase to each
 *     character produces the same result.
```

```
     *
     * Two sequences of characters are the same, ignoring case, if the
     * sequences have the same length and corresponding characters are
     * the same, ignoring case.
     *
     * anotherString   the String to compare this String against.
     * returns  true if the Strings are equal,
     *          ignoring case; false otherwise.
     * see also      java.lang.Character.toLowerCase(char)
     *               java.lang.Character.toUpperCase(char)
     */
    public boolean equalsIgnoreCase(String anotherString)
```

Javadoc

All of the standard classes are documented using an embedded comment format that can be processed by a Java tool called "javadoc" to create HTML documentation sheets. For example, the class spec sheets up on Sun's web site were generated by javadoc from the class source files. Expecting programmers to document as they go and directly generating the public documentation from the source is an interesting way to encourage that the two remain in sync. The javadoc utility produces nicely laid-out pages with sections for variables, methods, etc. and allows for cross-links to other classes and methods, which is pretty neat.

Javadoc is discussed in Chapter 2 of the Eckel text, and at some point, we may have a chance to take a bit about it in lecture. We definitely encourage you to try it out when commenting your assignments.

No Useless Comments!

A useless comment is worse than no comment at all— it still takes time to read and distracts from the code without adding information. Remember that the audience for all commenting is a literate programmer. Therefore you should not explain the workings of the language or basic programming techniques. Useless overcommenting can actually decrease the readability of your code, by creating muck for a reader to wade through. For example, the comments

```
int counter;                    /* declare a counter variable */
i = i + 1;                      /* add 1 to i */
while (index<length)...         /* while the index is less than the length */
num = num + 3 - (num % 3);      /* add 3 to num and subtract num mod 3 */
```

do not give any additional information that is not apparent in the code. Save your breath for important higher-level comments! Only illuminate low-level details of your implementation where the code is complex or unusual enough to warrant such explanation A good rule of thumb is: *explain what the code accomplishes rather than repeat what the code says*. If what the code accomplishes is obvious, then don't bother.

Inline Comments

Most of the rest of your comments will be "inline" comments. An inline comment explains the function of some nearby code. The golden rule for inline comments is: do not repeat what the code says. Code is a great vehicle for unambiguous, detail-oriented information. Comments should fill in the broader sort of information that code does not communicate.

If your identifiers are good, most lines will require no inline comments. An inline comment is appropriate if the code is complex enough that a comment could explain what is going on better than the code itself. Of the code snippet in the previous section, only the last is complex enough that its function is not completely obvious after a single reading. Complexity is probably the simplest reason a line might deserve a comment. A line may also deserve a comment if it's

important, unintuitive, dangerous, or just interesting. Here's a more useful comment to replace the one from above:

```
num := num + 3 - (num % 3); /* increment num to the next multiple of 3 */
```

Another useful role for inline comments is to narrate the flow of a routine. An inline comment might explain the role of a piece of code in terms of the overall strategy of the routine. Inline comments can introduce a logical block in the code. Begin-End blocks and the beginnings of loops are good spots for this sort of comment. As above, it's most useful to describe what is accomplished by the code.

```
/*
 * The following while loop locates the first vowel to occur
 * twice in succession in the array
 */
```

Another useful type of comment will assert what must be true at certain point.

```
/*
 * The file pointer must now be at the left hand side of a parenthesized
 * expression.
 */
```

or

```
/*
 * Because of the exit condition of the above loop, at least one of
 * the child fields must be null at this point.
 */
```

Such a condition is called an "invariant". Invariants are a useful sort of mental checkpoint to put in your code. You'll be less likely to get loop conditions, etc. wrong if you think about and put in invariants as you are writing.

Try not to allow inline comments to interfere visually with the code. Separate inline comments from the code with whitespace. Either set them off to the right, or put them on their own lines. In either case, it's visually helpful to align the left hand sides of the comments in a region. Alternately, some of the issues addressed in inline comments can be treated just as well in the implementation section of the method's comment. Whether you prefer inline comments or header implementation comments is a matter of personal choice.

Commenting Accuracy

Comments should correctly match the code; it's particularly unhelpful if the comment says one thing but the code does another thing. It's easy for such inconsistencies to creep in the course of developing and changing a class and its methods. Be careful to give your comments a once-over at the end to make sure they are still accurate to the final version of the program.

Attributions

All code copied from books, handouts or other sources, and any assistance received from other students, TAs, fairy godmothers, off the net, etc. must be cited. We consider this an important tenet of academic integrity, and as well it serves as useful information for the next person to come along and work with this code. For example,

```
/* IsLeapYear is adapted from Eric Roberts text,
 * _The Art and Science of C_, p. 200.
 */
```

or

```
/* I received help designing the Battleship data structure, in particular,
 * the idea for storing the ships in alphabetical order, from TA Albert Lin
 */
```

or

```
/* I found a cool applet at http://www.binky.com/code/Whizzy.html and used
 * the DoubleBufferedPane as the starting point for my animation routines.
 */
```

A few quick notes on Java style

Developing a good sense of style in a particular programming language takes practice and work. And, as with any complex activity, there is no one "right" way that can easily described, nor is there an easy-to-follow checklist of do's and don'ts. There's a lot of variation allowable in Java coding styles. If nothing else, try to be consistent. Here's a few style tips on the various Java constructs based on my ideas of what makes good sense.

Break

Using `break` is ok in loops. Ideally, the loop should be structured to iterate in the most straightforward way. The `break` in the body can detect the exception case which comes up during the iteration.

```
for (i = 0; i < length; i++ ) {
   if (<found>) break;
   ...
}
```

or

```
while (current != null) {
   if (<found>) break;
   ...
}
```

While

`While (true)` types loops are fine if they are really necessary. Often you need this for the loop-and-a-half-type situation where you need to first do some processing before you are able to test whether you need to exit the loop. If the first statement of a `while (true)` loop is the test, then you should just put the test directly into the `while` statement. If the bounds of iteration are known, then a `for` loop is preferable.

Return

A `return` in a place other than the end of a method body is potentially vulgar. The early `return` can be used nicely if it detects and immediately exits on an exceptional case. For example, a base case or error case right at the beginning of a function. Sometimes `return` can be used like a `break` inside a loop when some condition becomes true. Be careful with `return` in the bodies of your functions— experience shows they are responsible for a disproportionate number of bugs. The programmer forgets about the early-return case and assumes the function runs all the way to its end.

++, --

Nice obvious uses of these are fine, but nesting it inside something complicated is just asking for trouble. Find a more useful outlet for your cleverness.

```
for (i = 0; i < length; i++) {   // ok
```

```
...

while (--length) {               // ok
...

while (arentIClever(t++ * s++))  // ick!
...
```

Switch

If you ever exploit the fall-through property of cases within a `switch`, your documentation should definitely say so. It's pretty unusual.

Boolean Values

Boolean expressions and variables seem to be prone to redundancy and awkwardness. Replace repetitive constructions with the more concise and direct alternatives. If you have a boolean value, don't use additional != or == operators on it to test its value. Just use its value directly or with ! to invert. Also, watch for `if-else` statements which assign a variable to true or false. The result from evaluating the test can go right into the variable. A few examples:

```
if (flag == true)          is better written as      if (flag)


if (matches > 0)           is better written as      found = (matches > 0);
   found = true;
else
   found = false;


if (hadError == false)     is better written as      return !hadError
   return true;
else
   return false;
```

Formatting and White Space

One last little note. In the same way that you are attuned to the aesthetics of a paper, you should take care in the formatting and layout of your programs. The font should be large enough to be easily readable. Use white space to separate functions from one another. Properly indent the body of loops, if, and switch statements in order to emphasis the nested structure of the code.

There are many different styles you could adopt for formatting your code (how many spaces to indent for nested constructs, whether opening curly braces are at the end of the line or on a line by themselves, etc.). Choose one that is comfortable for you and be consistent!