Eric's Unofficial Commenting Guide
CSE 142 / 143

What are comments?

Comments are specially marked lines of text in your code that can be used to describe what's happening in your program. Java ignores comments when it runs your program, so they're not necessary for actually making code work properly.

There's no exact formula to writing comments. The important details of a method or program are different depending on what the program or method does. The simplest way to know if your comments are sufficient is to ask yourself this: "If I didn't understand the code in this method, would I still be able to use it properly by reading only the comment and the method header?"

Why comments are useful

It may seem like comments are a bit unnecessary, since you're the one writing your code. The purpose of comments isn't necessarily for the author's benefit (though they can be very useful as programs get more and more complex), but rather for the benefit of others who read your code. You might have a useful method that someone wants to use for another program, for example. Another programmer shouldn't have to reinvent the wheel - they should use your method instead of writing their own. If your method is properly commented, all they should have to read to know how your method works is the comment and the first line of the method (the header).

Behavior vs. Implementation

Comments should describe the behavior of a method (what it does) but generally should not describe its implementation (how it does it). If someone is really interested in exactly how a method works they can read through the code. When writing a comment, avoid discussing the commands or variables you used or what calls the method, and don't discuss what you thought about when you wrote the method or what you want it to do. Just describe what it actually does.

Descriptions of the Program

Descriptions of the program should follow the same ideas as method comments but at a higher level. Describe concisely but completely what someone can expect when they use the program, and don't go through all of the details of how it works.

Internal vs. External Comments

Comments don't need to be restricted to the description of a method. You can comment lines of code inside your method as well. Internal comments should be used to describe the behavior (not the implementation) of more complicated pieces of code. Don't comment the obvious - it can actually make your code less readable.

Comment Tags

There are a couple of ways to designate a comment in Java.

```
// You can write comments with two slashes for every line of comments
// You need to put slashes on every line, which can be tedious.

/*
   You can also write blocks of comments using comment open/close tags.
   The commented section starts with /* and ends when a */ appears.
*/

////////////////////////////////////////////////////////////////////
/*
*   Comments can also be done more creatively, but use this style
*   in special cases.  It's useful for giving an overall description
*   of large files, such as an entire library of methods that has
*   hundreds or thousands of lines of code, or for multiple files
*   that all need a description. You shouldn't use this style with
*   the short programs assigned in CSE 142 / 143.
*/
////////////////////////////////////////////////////////////////////
```

Commenting Simple Methods

Let's look at examples of commenting a simple method that prints an introduction:

```
public static void intro() {
      System.out.println("This program compares two applicants to");
      System.out.println("determine which one seems like the stronger");
      System.out.println("applicant.  For each candidate I will need");
      System.out.println("either SAT or ACT scores plus a weighted GPA.");
}
```

The following comment is too brief. It doesn't answer the most basic question of "what does the method do?"

```
// Intro
public static void intro() {
```

The following comment is much better. It concisely describes the method's behavior.

```
// Prints an introduction to the program to the console
public static void intro() {
```

The next version of the comment isn't very helpful. It mentions that it prints an introduction to the program, which describes its behavior, but it also includes unnecessary details about the method's implementation. If someone wants to know how the printing happens they can read the code.

```
// Uses four println statements to print four lines of introduction to
// the program.  Each line is different, so each one needs a separate
// println statement.
public static void intro() {
```

Commenting Parameters

Suppose you have a useful method that prints a list of numbers up to and including a number passed in as a parameter. A good comment (and all someone using the method should have to read) is this:

```
// Takes an int representing a maximum and prints a list of numbers to the
// console up to and including max in the following format:  1, 2, 3, ..., max
public static void printNumberList(int max) {
	for (int i = 1; i < max; i++) {
		System.out.print(i + ", ");
	}
	System.out.println(max);
}
```

That seems pretty straightforward. The method and its parameters are well-named, and the comment is concise but descriptive. What if you were to encounter the exact same method but it was named and commented like this?

```
// Lists the numbers using a for loop
public static void list(int n) {
```

It's very difficult to tell what this method does. It could print a list up to but not including n (an important difference) or it could print the list starting at n and going down to one. You can't even tell if it prints a list. Maybe it adds the parameter n to some list, or maybe it makes "n" lists. The only way to know what this method does and how to use it is to read all of the code and figure it out. This can be very time consuming. It's also good to note that the name of a method and the names of its parameters - everything in the header - help describe what the method does.

Commenting Methods with Return Values

Sometimes what a method returns might not be immediately obvious. A method that returns an int might be returning a result it calculates, or it might print the result to the console and instead return the number of seconds the calculation took. It's important to describe what the return value of a method represents so that people know what to expect when they use it.

This first version of the method below is well commented. The comments are concise and correctly describe what the method takes as a parameter and what it returns. The code is straightforward and explains itself. Internal comments aren't really necessary here.

```
// Takes an integer n as a parameter and returns the factorial of n
public static int factorial(int n) {
	int result = 1;
	while(n > 1) {
		result *= n;
		n--;
	}
	return result;
}
```

The next comment isn't descriptive enough - it doesn't describe the input and output parameters. It calculates the factorial of what? What does the int it returns represent?

```
// Calculates the factorial
public static int factorial(int n) {
```

The next version of the comments contains too many implementation details, and the internal comments actually make the code less readable. If the code is correctly indented, it's perfectly obvious that the while loop is a while loop and that it ends where its brackets end. n-- is obviously an update of n, and we can tell the return statement is a return. Save internal comments for more complicated statements.

```
// Uses a result variable and a while loop to calculate the result of
// n factorial.  The while loop runs until the result is calculated,
// and a return statement is used to return result.
public static int factorial(int n) {
	int result = 1; // initialize result
	// while loop
	while(n > 1) {
		result *= n;
		n--; // update n
	} // end of while loop
	return result; // return
}
```

Describing Behavior and Internal Comments

The method below has appropriate internal and external comments. The comments concisely describe the behavior of the method as well as the input parameters without discussing the implementation. The internal comment is there to describe what an odd-looking piece of code is doing. It's sometimes good to leave whitespace before an internal comment so that it's easier to read.

```
// Takes a Scanner "input" and a PrintStream "output" as parameters.
// Prints the contents of the input Scanner to output with trimmed
// whitespace and line breaks every 10 tokens.  Existing line breaks
// in intro are ignored.
public static void addLineBreaks(Scanner input, PrintStream output) {
	int tokenCount = 0;
	while(input.hasNext()) {
		output.print(input.next());
		tokenCount++;

		// Add a line break every 10th line
		if(tokenCount % 10 == 0) {
			output.println();
		}
	}
}
```

The next method description isn't sufficient. It doesn't correctly answer the questions "what does this code do?" "what are the parameters?". If someone were to read these comments, they might be confused about what the Scanner was supposed to contain and where the output was being printed to.

```
// prints out a file with line breaks every 10 tokens
public static void addLineBreaks(Scanner input, PrintStream output) {
```

The next version of the comments isn't appropriate because it deals with implementation details. We don't need to know that the code uses a while loop in order to use it. If someone wants to find out HOW the code gets the job done, they can read the code.

```
// Uses a while loop and a scanner to output the Scanner file
// with line breaks every 10 tokens.
public static void addLineBreaks(Scanner input, PrintStream output) {
```

Commenting Exceptions and Pre/Post conditions

Exceptions are an important part of the behavior of the method. When commenting a method that may throw an exception, it's necessary to include the type of the exception and the conditions that throw it in your comments. You can also provide a short explanation of why the exception was thrown in the Exception itself. This is a nice way to give a user a little more information than just the exception type. When these exceptions are thrown, the string describing why shows up in the exception trace.

Pre conditions and post conditions are a way to assert what must be true before and after a method runs. For the findMax method to run properly, the preconditions for valid parameters must be true - it will throw an exception otherwise. A post condition might be used to describe something that must have changed after the method is run. A method that sorts a list, for example, has the post condition that the list is sorted. Some methods may not have a post condition or (more rarely) may not have a pre condition.

```
/*
*  Pre:  start >= 0, end <= list.length, start >= end
*  Throws an IllegalArgumentException if preconditions not met.
*
*  Takes a list of integers and two integers representing the start and
*  end of a range in the list in which to search.  Returns the maximum
*  value found in the list of integers between start and end, start
*  inclusive and end exclusive.
*/
public static int findMax(int[] list, int start, int end) {
      if (start < 0 || end > list.length)
            throw new IllegalArgumentException("start and/or end out of range");
      if (start >= end)
            throw new IllegalArgumentException("start is greater than end");

      int max = list[start];

      // Search the list for a maximum
      for (int i = start + 1; i < end; i++)
            max = Math.max(max, list[i]);

      return max;
}
```