## Why do we write comments?

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

There's no exact formula to writing comments. The simplest way to know if comments are sufficient is to ask, "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?"

The purpose of comments isn't necessarily for the author's benefit, though they can be helpful as programs get more complex. Rather, comments are for the benefit of clients who are using the program. Ideally, a client should only have to read the method comment and the method signature (return type, name, parameters) to know how to use a method.

As such, comments should describe the behavior of a method (what it does) without its implementation (how it does it). If someone is really interested in seeing exactly how a method works, they can read the code.

Method comments in particular should be in either pre/post format, standard JavaDoc format, or some other format that includes all the required information. Use proper capitalization; do not write comments in all capital letters.

Limit comments to 100 characters per line. Longer lines hurt readability. Keep comments short and sweet. Most methods can be described in a single sentence, though more complicated methods might take two or three sentences.

Use either multiline comments `/* */` or single line comments `//`.

### Write for the client

The client is a person who wants to use a program without knowing how it works. For examples, see the Java API documentation (e.g. ArrayList).

### Avoid implementation details

Avoid things that describe how the program achieves its results. In particular, avoid mentioning:

- Local variable names, private field names.

- For loop, while loop, if/else, recursion, inner data structures, calls to other methods.

If some implementation detail is vitally important, reformat the detail in a client-oriented way. For example, rather than “Uses a SortedMap to maintain sorted order,” restate the detail as, “Maintains sorted order.”

## Describe parameters, results, and exceptions

Describe what each parameter represents, either in a list or as part of the method description. Describe why each parameter is needed. Comment any special cases the client may care about, e.g. “The given list should be in sorted order.”

If the program produces output, describe it.

**Where does the output appear?**

On the console, in a file, etc.

**What is the output?**

A list of numbers, a paragraph, the result of a computation, etc.

**What is the output format?**

On multiple lines, in reverse order, separated by spaces, etc.

If the program returns a value, describe what the returned value represents in relation to the input. Comment on any special cases the client may care about. For example, does the method return null in a particular case?

Tell the user precisely what will cause an exception. Include the exact type of any exceptions that are thrown next to their causes so that the client knows what could happen.

## Internal comments, private methods, and fields

While method comments should not include implementation details since they’re intended for the client, internal comments placed within the body of a method might be helpful for maintaining the program in the future.

Do not comment trivial code, only code that is important, confusing, or tricky. Use only single line comments `//`.

Method comments for private methods follow the same rules as public methods, but are slightly less strict. It's still important to avoid commenting on implementation details.

Put a brief comment on each field describing what it represents, if the function of the field is not immediately apparent from its name. Comments on public fields or constants in particular should avoid mentioning implementation details.

### Be direct

Reword sentences to jump straight to the point. Instead of, "This method calculates a sum…" just write, "Calculates a sum…"

Don't say the method "should" do something, but instead "will" do something.

Do not copy from the assignment specification. The specification usually gives too much information for commenting. It is a good idea to draw only the information the client needs from the specification and put them in your own words.

## Case studies

### Method comments

Consider this method that prints an introduction.

```
public static void intro() {
    System.out.println("This program computes the cost of postage required
to send mail via the"
    System.out.println("United States Postal Service. In addition to sender
and recipient,");
    System.out.println("the program will need each item's dimensions and
weight.");
}
```

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

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

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

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

While the next comment is more descriptive, it's arguably too descriptive. It mentions that it prints an introduction to the program, which describes its behavior, but also includes unnecessary details about the method's implementation details. If someone wants to know how the printing happens, they can read the code.

```java
/* 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() {
```

## Parameter comments

Consider this method that prints a list of numbers up to and including the value passed in as a parameter. The following is a good comment for the method.

```java
/* Takes an int representing a maximum and prints a list of numbers to the
console up to and including max in the 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. Contrast the above comment with this example.

```java
// 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` down to 1. It's not possible to tell if it prints a list.

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 help describe what the method does.

## 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 that 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 necessary here.

```java
// 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 returned `int` represent?

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

## Internal comments

The next version of the comments contains too many implementation details and the internal comments actually make the code less readable and more cluttered. The "while loop" comments are completely redundant with the `while` statement declared in code. Java programmers reading the code will know that `n--` will "update n" without needing the comment to say so. The `return` statement includes the `return` keyword, so it's not necessary to include a comment that just restates the keyword.

```java
/* 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
}
```

Save internal comments for more complicated statements.

### Implementation details

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 the function of a particularly odd-looking piece of code. 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 the input 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?” and, “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 is printed.

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

The next method comment 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) {
```

## Pre/postconditions

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 pre-conditions for valid parameters must be true (or else an exception will be thrown). A post-condition is used to describe the effects of the method after the method has completely executed. 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
 *
 *
 * post: Takes an integer list 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 or end out of range");
    } else 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;
}
```