In Lab 0.1, you used many methods from classes which you didn’t write: Math.log(), String.split(), String.equals(), and so on. If you didn’t write them, how do you know what the methods do? How do you know what arguments they take?

Some of you may have discovered the Java API (“application programmatic interface” – a detailed list of objects and methods that programs can use), informally called “the” Javadoc:

[Links to Javadocs for JDK 5 and JDK 6]

Tip: Bookmark these APIs! They will become invaluable in helping you use the Java library classes.

This API is more formally called a specification: detailed documentation that describes exactly what each class and interface does, what each method requires and returns, and so on.

Take a look at the Java API. The top-left frame lists each package; the bottom-left frame lists each class within the currently selected package (or initially, all classes); the main frame shows the specification for the currently selected package or class.

Examine the Math class (click “Math” in the bottom-left frame) and take a look at the Math.log() and Math.log10() methods. This is an example of the importance of specifications – they detail subtle differences that clients may need to know about.

You’ll find that as you start to write larger and more complex programs, specifications become downright necessary, especially when it’s more than just you who’s working with the code. There are many ways to write formal specifications, but we’ll stick with the Javadoc style for this class, so let’s analyze the specification for a class and its methods.

We’ll use the String class as an example. When we examine the specification for this class, we notice a very detailed description of what the String class represents, how it’s implemented (this part is not always necessary, we’ll cover this more later), its subtleties, but most importantly, all important information clients (users) of this class should know.

Now let’s look at a specific method, like String.split():

(Next page)
The basic description of the method is shown, of course, but there are also more detailed descriptions of the parameters that the method takes, exactly what value it returns, and exactly what exceptions it throws (and in which cases it throws them).

As you can see, all of these specifications are presented very professionally, with consistent formatting and hyperlinks to referenced classes. We’d like the same for our code, and fortunately, we can get it – the entire specification can exist as comments inside your code, and if it’s formatted according to the Javadoc style, Java can generate these Javadocs automatically!

The format is very simple:

1. All specifications must go inside blocks of comment that begin with /** and end with */.
2. All specifications must go directly before the method or class signature.

Here is a simple example:

```java
/**<n
* Reverses the letters of the given string and returns the result.
* Throws a NullPointerException if null is given.
* */
public static String reverseLetters(String text) {
    ...
}
```

Javadoc also supports specialized annotations for parameters needed, values returned and exceptions thrown, in order to get special formatting in the HTML (like for the String.split() example). You should still put an English description – the first sentence serves as a method summary.

The format for these annotations is as follows:

- For parameters: `@param [name] [description]`
- For return values: `@return [description]` (note there’s no “s” at the end of @return)
- For exceptions: `@throws [exception] [description]`
Here is the same example using these annotations:

```java
/**
 * Reverses the letters of the given string and returns the result.
 * @param text the string to reverse.
 * @return a new string whose letters are the reverse of text.
 * @throws NullPointerException if text is null.
 */
public static String reverseLetters(String text) {
    ...
}
```

In general, you should get in the habit of writing Javadoc specifications for every method you write. They don’t need to be lengthy; for trivial methods, they should help confirm that the method is indeed trivial, and for non-trivial methods, they should explain all non-obvious aspects. In particular, you should document **pre-conditions** (e.g. “the list must be sorted”, “x must not be null”, etc.) and **post-conditions**.

Javadocs are most often shown for public methods only, but it’s good to document private methods too; it helps you to understand the code later. Either way, it’s important that your specifications be especially polished if your code will be used by others. The Java API is a good example of this.

It’s important to maintain modularity through the documentation, too. Just as we want to reduce dependencies as much as possible in code, we should reduce dependencies as much as possible in specifications. In general, isolate the description to the functionality of what you’re describing.

Here’s an example of unnecessary dependencies in a specification:

```java
/**
 * A class for encrypting messages. Needed for the Security class so
 * that passwords can be encrypted, a requirement of PasswordManager.
 */
public class Encrypter {
    ...
}
```

However, when dependencies are inherent, it can be better to inform the user:

```java
/**
 * Returns the next element, or throws an IllegalStateException if
 * there are no more elements. Thus, you should call this method
 * only if the hasNext() method returns true.
 */
public Object next() {
    ...
}
```

Writing great specifications is an art that comes with time. If you’re interested in learning more, Sun’s Javadoc guide at [http://java.sun.com/j2se/javadoc](http://java.sun.com/j2se/javadoc) is a good place to start; the sections “How to Write Doc Comments for Javadoc” and “Requirements for Writing API Specifications” are particularly useful. Happy documenting!
How to Attach the Java API in Eclipse

Whenever you hover your mouse over a built-in class or method in Eclipse, e.g. over the methods String.split() or Math.log(), you should see the Javadoc for that class or method. If you don’t, you’ll need to attach the location of the Java API in Eclipse.

You can set the location to be online, i.e. to the links on the first page, or you can download the entire Javadoc onto your machine and set the location to be this local copy. To download a zip of the Javadoc, go to Sun’s Java downloads page:


From there, scroll down to “J2SE 5.0 Documentation” or “Java SE 6 Documentation”, depending on your version, and click Download.

To set the location of the Java API:

1. Expand the “JRE System Library” inside your project.
2. Right-click on either “rt.jar” (for Windows and Linux) or “classes.jar” (for Mac).
3. Select “Javadoc Location” in the left-hand list.
4. Choose either “Javadoc URL” (if using the online location, or if your local copy is unzipped) or “Javadoc in archive” (if your local copy is zipped).
5. Hit OK to apply the changes.

How to Generate Javadoc in Eclipse

You can generate formatted HTML for your Javadoc specifications automatically in Eclipse:

1. With your project (or package or class) selected, go to File > Export...
3. Make sure your project (or package or class) is checked.
4. (Optional) By default, “public” visibility is set. This means HTML for private methods won’t get generated. To change this, check “private”.
5. Set the export location. This should generally be a folder called “docs” inside your project folder. Hit Next.
6. (Optional) Set the document title to something descriptive Otherwise, it’s simply “Overview”.
7. (Optional) To create links to built-in Java library classes or objects, check “rt.jar” (for Windows and Linux) or “classes.jar” (for Mac). This will also simplify the appearance of classes from names like “java.lang.String” to just “String”.
8. (Optional) Hit Next, and check “Open generated index file in browser”.
9. Hit Finished to begin the process, which may take a few minutes.