Appendix H
Javadoc

Throughout the book, we have used nicely formatted comments to provide the specifications for Java classes. For example, the following specification for `celciusToFahrenheit` appears just before its implementation in Figure 1.1 on page 11:

♦ **celsiusToFahrenheit**

`public static double celsiusToFahrenheit(double c)`

Convert a temperature from Celsius degrees to Fahrenheit degrees.

Parameters:

`c` – a temperature in Celsius degrees

Precondition:

`c >= -273.15.`

Returns:

the temperature `c` converted to Fahrenheit degrees

Throws: `IllegalArgumentException:`

Indicates that `c` is less than the smallest Celsius temperature (–273.15).

How are pretty comments such as this generated? The answer is a special documentation comment that appears in a `.java` file, such as this:

```
/**
* Convert a temperature from Celsius degrees to Fahrenheit.
* @param c
*   a temperature in Celsius degrees
* @precondition
*   c >= -273.15.
* @return
*   the temperature c converted to Fahrenheit
* @throws IllegalArgumentException
*   Indicates that c is less than the smallest Celsius temperature (-273.15).
**/
```

This **documentation comment** is arranged to interact with a documentation tool called **Javadoc**, which is freely distributed with Sun Microsystems' Java Development Kit. Javadoc reads the documentation comments from a `.java` file and produces a collection of documentation pages in a format called **html** (hypertext markup language). As you may know, html pages are read and displayed by web browsers such as Netscape Navigator, Microsoft Internet Explorer, or Sun's HotJava Web Browser. For example, Javadoc can be applied to the preceding `celsiusToFahrenheit` documentation comment, and the resulting html page will contain the nicely formatted comment shown at the top of the page. The exact format may look different in your web browser, but the result will be roughly as shown. The html page created by Javadoc contains only documentation and no actual Java code. As you might guess, the motivation for this kind of documentation is *information hiding*.

The documentation allows other programmers to use the items you implement without knowing how you implemented those items. Let's look at the precise steps you carry out when you want to make your programming available to other programmers but still retain information hiding.

How to Use Javadoc to Provide Your Work to Other Programmers

1. **Documentation comments appear in your `.java` files.** Write your `.java` files in the usual way but include documentation comments such as the one shown earlier. We'll discuss the general format for these documentation comments in a moment.
2. **Other programmers need access to your `.class` files.** Compile each `.java` file in the usual way. Each `.java` file that you compile creates a corresponding `.classes` file. For example, when you compile `TemperatureConversion.java`, you get `TemperatureConversion.classes`. You can move your `.classes` files to a location where other programmers can access them.
3. **Run the Javadoc tool on each of your `.java` files.** From the command line, Javadoc is executed by typing the command `javadoc` and the name of the `.java` file. For example:

   ```
   javadoc TemperatureConversion.java
   ```

   Javadoc creates an html file. In this example, it is a file called `TemperatureConversion.html`. Later, we'll look at some additional options that can be put on the command line.
4. **Move all your Javadoc `html` files to a public place where a web browser can read them.** For example, I have moved all my Javadoc `html` files to the directory:

   ```
   http://www.cs.colorado.edu/~main/docs/
   ```

   If you point your web browser to this directory, you will see all of my public documentation. You may need to talk with your computer system administrator to find out how to create a public place for your own documentation to reside.
5. **Put the JDK graphics files in an images subdirectory.** Below your `javadocs` directory, you should create a subdirectory called `images`. The Java Development Kit provides a collection of graphics images, and you can copy these to your own `images` subdirectory. Javadoc will use the graphics in the html pages that it creates. (The latest JDK release places these images in a subdirectory `java\docs\api\images` for Windows or `java/docs/api/images` for Unix.)

Now you need to know how to write those Javadoc documentation comments. (They're not as cryptic as they look.)

How to Write Javadoc Documentation Comments

We will put Javadoc comments in two places: before each class definition (such as `public class TemperatureConversion`) and before most public methods (such as `public static void print-Number`).

Documentation comments begin with `/**` (two stars rather than just one). A documentation comment ends with `*/`, but for consistency, you can use `**/` instead. Many programmers place a single asterisk (*) at the start of each line in the comment. The single asterisk is not required, but it does make it easy to see the entire extent of the documentation (and Javadoc ignores the asterisks).

The top of each documentation comment contains a description of the class or method. Javadoc uses the first sentence of the description as part of an index, so aim for a concise account of the most important behavior in the first sentence. If needed, subsequent sentences can provide details.

After the description, the rest of the documentation consists of a series of **Javadoc tags**. Each tag alerts the Javadoc tool about certain information. For example, one of my classes, called `TemperatureConversion`, has the Javadoc comment shown here, just before the class declaration:

```
/**
 * The TemperatureConversion Java application prints a table
 * converting Celsius to Fahrenheit degrees.
 * @author
 *   Michael Main (main@colorado.edu)
**/
public class TemperatureConversion...
```

The Javadoc tag `@author` indicates that the name of the class's author will appear next (perhaps with extra information such as an e-mail address). In the html page produced by Javadoc, a large heading is provided for the whole class, and each individual tag is made into a boldface heading, so within a web browser, the html page contains a section similar to this:

❖ **public class TemperatureConversion**
The `TemperatureConversion` Java application prints a table converting Celsius to Fahrenheit degrees.
Author:
Michael Main (main@colorado.edu)

The documentation for a class will usually contain a description and an author tag.

Javadoc Documentation for Individual Public Methods

Each public method of a class may also have a Javadoc comment preceding its implementation. My documentation for a method generally includes these items:

1. **A description of the method.** For example, `celsiusToFahrenheit` has this description at the top of the documentation comment:

```
/**
 * Convert a temperature from Celsius degrees to Fahrenheit.
```

2. **Parameters.** Each parameter of the method is documented by using the tag `@param`, followed by the parameter name and a description of the parameter. For example, part of our first Javadoc comment is:

```
 * @param c
 *   a temperature in Celsius degrees
```


Warning: Some of the other tags shown below won't work unless you put at least one `@param` tag first. If there are no parameters, I suggest an `@param` tag that looks like this:

```
 * @param - none
```

3. **Precondition.** The most recent Javadoc tool does not have a tag for preconditions. However, you can create and use new tags of your own. So our Javadoc comment lists the precondition as shown here:

```
 * @precondition
 *   c >= -273.15.
```

For this new tag to work correctly, we will need to include an option on the Javadoc command line. In particular, to use the precondition tag on `Node.java`, we will run Javadoc like this:

```
javadoc -tag precondition:a:"Precondition:" Node.java
```

This option tells Javadoc that we are going to use a new tag that will be indicated by `@precondition` in our Javadoc comments. The `:a:` stands for "all" and means that all occurrences of the `@precondition` tag should be put in the documentation along with a heading, which in this case is "`Precondition:`". The heading will always appear in boldface in the documentation that Javadoc creates.

4. **The returns condition or postcondition.** You can list a returns condition after the Javadoc tag `@return`. For example, we wrote:

```
 * @return
 *   the temperature c converted to Fahrenheit
```

A returns condition is appropriate when the method's behavior is entirely described by its return value. More complex methods need a complete postcondition instead of a returns condition. A complete postcondition describes other effects of the method, such as printing output. There is no Javadoc postcondition tag, but you can use `@postcondition` as a new tag so long as you put the `-tag postcondition:a:"Postcondition:"` option on the command line when you run Javadoc. For example, suppose our method converted the parameter `c` to Fahrenheit degrees and printed the result instead of returning the value. Then the documentation could look like this:

```
 * @postcondition
 *   The Fahrenheit temperature equivalent to c has been
 *   printed to System.out.
```

5. **Exceptions.** Each exception that a method can throw should be listed in the documentation comment. Start with the tag `@throws` and then write the name of the exception type (such as `IllegalArgumentException`) followed by a description of what the exception indicates. For example, our exception was documented like this:

```
 * @throws IllegalArgumentException
 *   Indicates that c is less than the smallest Celsius
 *   temperature (-273.15).
```

Controlling html Links and Fonts

The Javadoc tags that have been described are sufficient to clearly document the code you'll write throughout this text. There are two other features you might want to add. Under the `@author` tag, you list your name and e-mail address. It would be nice if a reader could simply click on that address in a web browser to send e-mail to you. This is possible by using some simple html to revise the author section as shown in the shaded portion here:

```
/**
 * The <CODE>TemperatureConversion</CODE> Java application prints a table
 * converting Celsius to Fahrenheit degrees.
 * @author
 *   Michael Main
 *   <A HREF="mailto:main@colorado.edu"> (main@colorado.edu) </A>
 **/
```

The first part of the shaded line, `<A HREF="mailto:main@colorado.edu">`, is an html command that will put a **mailto link** on the documentation page. Within a web browser, the link appears as the text `(main@colorado.edu)`. To send e-mail to the address, a user clicks on the link. The characters `</A>` at the end of the line are an `html` command to indicate the end of the link. Within a browser, the new documenation appears like this:

❖ **public class TemperatureConversion**

The `TemperatureConversion` Java application prints a table converting Celsius to Fahrenheit degrees.

Author:

Michael Main (main@colorado.edu)

The difference between this and the original version is that the link (main@colorado.edu) is underlined and probably in a bright color to indicate that it is a link.

You may have noticed a second difference: In this version, the name `TemperatureConversion` appears in a special "code" font that looks like code from a program. This font was controlled by putting the html tag `<CODE>` to turn on the special font and the tag `</CODE>` to turn off the special font.

Running Javadoc

Once you have Javadoc comments in your code, you can run the Javadoc program to produce the html files. For example:

```
javadoc Names of one or more .java files to process
```


These are the complete options that I usually use:

```
javadoc -author -source 1.4 -public -tag param -tag
  -tag precondition:a:"Precondition:"
  -tag postcondition:a:"Postcondition:"
  -tag return
  -tag throws
  -tag example:a:"Example:"
  Names of one or more .java files to process
```

The `-author` option allows us to put `@author` in the Javadoc comments, followed by the author's name. The `-source 1.4` allows us to use certain features of Java Version 1.4. The `-public` option indicates that we want to generate documentation for only the public members of a class. The other options (all starting with `-tag`) indicate the order in which we want the documentation to appear: `precondition` first, then `postcondition`, then the `return` specification, then the `throws` specification, and finally an `example` tag (to let me give an example of how the method is used). Two of these options (`return` and `throws`) are built into the Javadoc system, so for these we need to write only `-tag return` and `-tag throws`. The precondition, postcondition, and example tags are not built into Javadoc, so the extra information at the end (such as `:a:"Precondition:"`) tells Javadoc how we want these tags to appear in the documentation.

The rest of this appendix lists the complete `TemperatureConversion` implementation, including all Javadoc comments that are printed in bold.

Java Application Program

```
// File: TemperatureConversion.java
// A Java application to print a temperature conversion table.
// Additional Javadoc information is available in Figure 1.1 on page 11 or at
//   http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html

/******************************************************************************
* The <CODE>TemperatureConversion</CODE> Java application prints a table
* converting Celsius to Fahrenheit degrees.
*
* @author Michael Main
*   <A HREF="mailto:main@colorado.edu"> (main@colorado.edu) </A>
******************************************************************************/
public class TemperatureConversion
```

```java
{
    /**
     * The main method prints a Celsius to Fahrenheit conversion table.
     * The String arguments (args) are not used in this implementation.
     * The bounds of the table range from -50C to +50C in 10 degree increments.
     **/
    public static void main(String[ ] args)
    {
        final double TABLE_BEGIN = -50.0; // The table's first Celsius temperature
        final double TABLE_END   =  50.0; // The table's final Celsius temperature
        final double TABLE_STEP  =  10.0; // Increment between temperatures in table

        double celsius;                   // A Celsius temperature
        double fahrenheit;                // The equivalent Fahrenheit temperature

        System.out.println("TEMPERATURE CONVERSION");
        System.out.println("----------------------");
        System.out.println("Celsius     Fahrenheit");

        for (celsius = TABLE_BEGIN; celsius <= TABLE_END; celsius += TABLE_STEP)
        {  // Each iteration prints one line of the conversion table.
            fahrenheit = celsiusToFahrenheit(celsius);
            System.out.printf("%6.2fC", celsius);
            System.out.printf("%14.2fF\n", fahrenheit);
        }
        System.out.println("----------------------");
    }


    /**
     * Convert a temperature from Celsius degrees to Fahrenheit degrees.
     * @param c
     *   a temperature in Celsius degrees
     * @precondition:
     *   c >= -273.15.
     * @return
     *   the temperature c converted to Fahrenheit
     * @throws IllegalArgumentException
     *   Indicates that c is less than the smallest Celsius temperature (-273.15).
     **/
    public static double celsiusToFahrenheit(double c)
    {
        final double MINIMUM_CELSIUS = -273.15;
        if (c < MINIMUM_CELSIUS)
            throw new IllegalArgumentException("Argument " + c + " is too small.");
        return (9.0/5.0) * c + 32;
    }

}
```

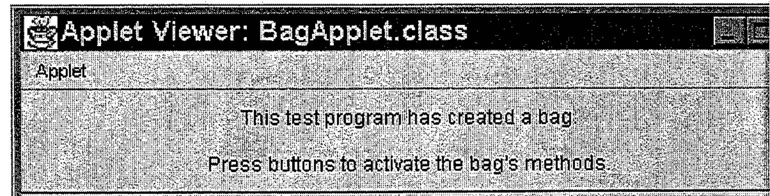

Appendix I
Applets for Interactive Testing

It's useful to have a small interactive test program to help you test c
can be written as a Java **applet**, which is a Java program written in a s
ical user interface. The graphical user interface is also called a GUI
allows a user to interact with a program by clicking the mouse, typin
performing other familiar actions. With a Java applet, GUIs are easy
run into such goo before.

This appendix shows one simple pattern for developing suc
pattern, we'll implement an applet that lets you test three of the ba
`countOccurrences` from Chapter 3). When the bag applet starts, a
drawing in Figure I.1(a).

By the way, the word "applet" means a particular kind of Java prog
I.1 to your boss and say, "My applet created this nice GUI." But you
to talk about the GUI itself, such as "The applet in Figure I.1(a) has t
tion." And, in fact, there are three buttons in that applet—the rectangle
`countOccurrences( )`.

The applet in Figure I.1 is intended to be used by the programme
class, to check interactively that the class is working correctly. When
appear at the top: "`This test program has created a bag. P`
`the bag's methods.`" Above these sentences are some extra items,



The display above our sentences is created automatically by the ap
exact form of this display varies from one system to another, but the
ally contains controls such as the ☒ in the top right corner. Clicki
closes the applet on this particular system.

A series of buttons appears in the middle part of the applet, like th