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 10:

- **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 >= -273.16`.

  Returns:
  the temperature `c` converted to Fahrenheit degrees

  Throws: `IllegalArgumentException`:
  Indicates that `c` is less than the smallest Celsius temperature (–273.16).

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
* <dt><b>Precondition:</b><dd>
*   c >= -273.16.
* @return
*   the temperature c converted to Fahrenheit
* @exception IllegalArgumentException
*   Indicates that c is less than the smallest Celsius temperature (-273.16).
**/
```

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** (hyper-text 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 the `celsiusToFahrenheit` documentation comment shown above, 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 that you implement, without knowing how you implemented those items. Let's look at the precise steps that 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. In order to enable a special "author" option, you should also include the word `-author` after the `javadoc` command. For example, we can run javadoc on `TemperatureConversion.java` by typing the following on the command line:

   ```
   javadoc -author TemperatureConversion.java
   ```

   Javadoc creates an `html` file. In this example, you'll get one file called `TemperatureConversion.html`.
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 at 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 printNumber`).

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 comment (and Javadoc ignores the asterisks).

Javadoc Documentation for a Description of the Whole Class

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, if you know a little bit about html pages, you can more or less create your own tags. For example, at the start of a precondition you can write this: `<dt><b>Precondition:</b><dd>`. The `<dt><b>` means that you are giving a boldface title, the `</b>` turns off the boldface, and the `<dd>` means that you are going to provide text to go under the title. So, our Javadoc comment lists the precondition as shown here:

```
* <dt><b>Precondition:</b><dd>
*  c >= -273.16.
```

4. **The Returns Condition or Postcondition.** You can list a return 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 this: `<dt><b>Postcondition:</b><dd>`. For example, suppose that 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:

```
* <dt><b>Postcondition:</b><dd>
*   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 `@exception`, 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:

```
* @exception IllegalArgumentException
*   Indicates that c is less than the smallest Celsius
*   temperature (-273.16).
```

6. **An Example.** For a complex method, you can include an example tag, like this:

```
* <dt><b>Example:</b><dd>
*   printNumber(12345,27, 8, 1); // Prints 12,345.3 in the U.S.
```

Controlling HTML Links and Fonts

The six Javadoc tags that have been described are sufficient to clearly document the code that you'll write throughout this text. There are two other features that 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 tage `<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. I usually run Javadoc from the command line, like this:

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

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 1.2 on page 14 or at
//   http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html

import java.text.NumberFormat; // Used in the printNumber method.

/*******************************************************************************
* 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
{
    /**
   * 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)
   {
      // Declare values that control the table's bounds.
      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
      final int    WIDTH       =     6; // Number of characters in output numbers
      final int    ACCURACY    =     2; // Number of digits to right of decimal point

      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)
      {  // The for-loop has set celsius equal to the next Celsius temperature of the table.
         fahrenheit = celsiusToFahrenheit(celsius);
         printNumber(celsius, WIDTH, ACCURACY);
         System.out.print("C        ");
         printNumber(fahrenheit, WIDTH, ACCURACY);
         System.out.println("F");
      }
      System.out.println("----------------------");
   }
```

```
/**
* Convert a temperature from Celsius degrees to Fahrenheit degrees.
* @param c
*   a temperature in Celsius degrees
* <dt><b>Precondition:</b><dd>
*   c >= -273.16.
* @return
*   the temperature c converted to Fahrenheit
* @exception IllegalArgumentException
*   Indicates that c is less than the smallest Celsius temperature (-273.16).
**/
public static double celsiusToFahrenheit(double c)
{
   final double MINIMUM_CELSIUS = -273.16;
   if (c < MINIMUM_CELSIUS)
      throw new IllegalArgumentException("Argument " + c + " is too small.");
   return (9.0/5.0) * c + 32;
}

/**
* Print a number to <CODE>System.out</CODE>, using a specified format.
* @param d
*   the number to be printed
* @param minimumWidth
*   the minimum number of characters in the entire output
* @param fractionDigits
*   the number of digits to print on the right side of the decimal point
* <dt><b>Precondition:</b><dd>
*   <CODE>fractionDigits</CODE> is not negative.
* <dt><b>Postcondition:</b><dd>
*   The number <CODE>d</CODE> has been printed to <CODE>System.out</CODE>.
*   This printed number is rounded to the specified number of digits on the
*   right of the decimal. If <CODE>fractionDigits</CODE> is 0,
*   then only the integer part of <CODE>d</CODE> is printed.
*   If necessary, spaces appear at the front of the number to raise
*   the total number of printed characters to the minimum. Additional
*   formatting details are obtained from the current locale. For example,
*   in the United States, a period is used for the decimal and commas are
*   used to separate groups of integer digits.
* @exception IllegalArgumentException
*   Indicates that <CODE>fractionDigits</CODE> is negative.
* <dt><b>Example:</b><dd>
*   <code>printNumber(12345.27, 8, 1); // Prints 12,345.3 in the U.S. </code>
**/
public static void printNumber(double d, int minimumWidth, int fractionDigits)
{
    // Note: getNumberInstance( ) creates a NumberFormat object using local
    // information about the characters for a decimal point and separators.
    NumberFormat form = NumberFormat.getNumberInstance( );
    String output;
```

```
            int i;

            // Set the number of digits to appear on the right of the decimal.
            if (fractionDigits < 0)
               throw new IllegalArgumentException("fractionDigits < 0:" + fractionDigits);
            form.setMinimumFractionDigits(fractionDigits);
            form.setMaximumFractionDigits(fractionDigits);

            // Round and format the number.
            if (d >= 0)
               output = form.format(d + 0.5 * Math.pow(10, -fractionDigits));
            else
               output = form.format(d - 0.5 * Math.pow(10, -fractionDigits));

            // Print any leading spaces and the number itself.
            for (i = output.length( ); i < minimumWidth; i++)
               System.out.print(' ');
            System.out.print(output);
         }

      }
```