Creating Documentation with javadoc

M.1 Introduction
In this appendix, we provide an introduction to javadoc—a tool used to create HTML files that document Java code. This tool is used by Sun to create the Java API documentation (Fig. M.1). We discuss the special Java comments and tags required by javadoc to create documentation based on your source code and how to execute the javadoc tool. For detailed information on javadoc, visit the javadoc home page at

```
download.oracle.com/javase/6/docs/technotes/guides/javadoc/index.html
```

M.2 Documentation Comments
Before HTML files can be generated with the javadoc tool, programmers must insert special comments—called documentation comments—into their source files. Documentation comments are the only comments recognized by javadoc. Documentation comments begin with /** and end with */. Like traditional comments, documentation comments can span multiple lines. An example of a simple documentation comment is

```
/** Sorts integer array using MySort algorithm */
```

Like other comments, documentation comments are not translated into bytecodes. Because javadoc is used to create HTML files, documentation comments can contain HTML tags. For example, the documentation comment

```
/** Sorts integer array using <strong>MySort</strong> algorithm */
```

contains the HTML bold tags <strong> and </strong>. In the generated HTML files, MySort will appear in bold. As we’ll see, javadoc tags can also be inserted into the documentation comments to help javadoc document your source code. These tags—which begin with an @ symbol—are not HTML tags.

M.3 Documenting Java Source Code
In this section, we document a modified version of the Time2 class from Fig. 8.5 using documentation comments. In the text that follows the example, we thoroughly discuss each of the javadoc tags used in the documentation comments. In the next section, we discuss how to use the javadoc tool to generate HTML documentation from this file.
// Fig. M.1: Time.java
// Time class declaration with set and get methods.
package com.deitel; // place Time in a package

/**
 * This class maintains the time in 24-hour format.
 * @see java.lang.Object
 * @author Deitel & Associates, Inc.
 */
public class Time
{
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    /**
     * Time no-argument constructor initializes each instance variable
     * to zero. This ensures that Time objects start in a consistent
     * state. @throws Exception In the case of an invalid time
     */
    public Time() throws Exception
    {
        this(0, 0, 0); // invoke Time constructor with three arguments
    } // end no-argument Time constructor

    /**
     * Time constructor
     * @param h the hour
     * @throws Exception In the case of an invalid time
     */
    public Time(int h) throws Exception
    {
        this(h, 0, 0); // invoke Time constructor with three arguments
    } // end one-argument Time constructor

    /**
     * Time constructor
     * @param h the hour
     * @param m the minute
     * @throws Exception In the case of an invalid time
     */
    public Time(int h, int m) throws Exception
    {
        this(h, m, 0); // invoke Time constructor with three arguments
    } // end two-argument Time constructor

    /**
     * Time constructor
     * @param h the hour
     * @param m the minute
     * @param s the second
     * @throws Exception In the case of an invalid time
     */
public Time( int h, int m, int s) throws Exception {
    setTime( h, m, s); // invoke setTime to validate time
} // end three-argument Time constructor

/**
 * Time constructor
 * @param time A Time object with which to initialize
 * @throws Exception In the case of an invalid time
 */
public Time( Time time ) throws Exception {
    // invoke Time constructor with three arguments
    this( time.getHour(), time.getMinute(), time.getSecond() );
} // end Time constructor with Time argument

/**
 * Set a new time value using universal time. Perform
 * validity checks on the data. Set invalid values to zero.
 * @param h the hour
 * @param m the minute
 * @param s the second
 * @see com.deitel.Time#setHour
 * @see Time#setMinute
 * @see #setSecond
 * @throws Exception In the case of an invalid time
 */
public void setTime( int h, int m, int s ) throws Exception {
    setHour( h); // set the hour
    setMinute( m); // set the minute
    setSecond( s); // set the second
} // end method setTime

/**
 * Sets the hour.
 * @param h the hour
 * @throws Exception In the case of an invalid time
 */
public void setHour( int h ) throws Exception {
    if ( h >= 0 && h < 24 )
        hour = h;
    else
        throw( new Exception() );
} // end method setHour

/**
 * Sets the minute.
 * @param m the minute
 * @throws Exception In the case of an invalid time
 */
public void setMinute( int m ) throws Exception {
} // end method setMinute

/**
 * Sets the second.
 * @param s the second
 * @throws Exception In the case of an invalid time
 */
public void setSecond( int s ) throws Exception {
} // end method setSecond

Fig. M.1 | Java source code file containing documentation comments. (Part 2 of 4.)
public void setMinute( int m ) throws Exception
{
    if ( m >= 0 && m < 60 )
        minute = m;
    else
        throw( new Exception() );
} // end method setMinute

/**
* Sets the second.
* @param s the second.
* @throws Exception In the case of an invalid time
*/
public void setSecond( int s ) throws Exception
{
    if ( s >= 0 && s < 60 )
        second = s;
    else
        throw( new Exception() );
} // end method setSecond

/**
* Gets the hour.
* @return an <code>integer</code> specifying the hour.
*/
public int getHour()
{
    return hour;
} // end method getHour

/**
* Gets the minute.
* @return an <code>integer</code> specifying the minute.
*/
public int getMinute()
{
    return minute;
} // end method getMinute

/**
* Gets the second.
* @return an <code>integer</code> specifying the second.
*/
public int getSecond()
{
    return second;
} // end method getSecond

/**
* Convert to String in universal-time format
* @return a <code>String</code> representation
* of the time in universal-time format
*/

Documentation comments are placed on the line before a class declaration, an interface declaration, a constructor, a method and a field (i.e., an instance variable or a reference). The first documentation comment (lines 5–9) introduces class Time. Line 6 is a description of class Time provided by the programmer. The description can contain as many lines as necessary to provide a description of the class to any programmer who may use it. Tags @see and @author are used to specify a See Also: note and an Author: note, respectively in the HTML documentation. The See Also: note (Fig. M.2) specifies other related classes that may be of interest to a programmer using this class. The @author tag specifies the author of the class. More than one @author tag can be used to document multiple authors. [Note: The asterisks (*) on each line between /** and */ are not required. However, this is the recommended convention for aligning descriptions and javadoc tags. When parsing a documentation comment, javadoc discards all white-space characters up to the first non-white-space character in each line. If the first non-white-space character encountered is an asterisk, it’s also discarded.] This documentation comment immediately precedes the class declaration—any code placed between the documentation comment and the class declaration causes javadoc to ignore the documentation comment. This is also true of other code structures (e.g., constructors, methods, instance variables.).

**Common Programming Error M.1**
Placing an import statement between the class comment and the class declaration is a logic error. This causes the class comment to be ignored by javadoc.

**Software Engineering Observation M.1**
Defining several fields in one comma-separated statement with a single comment above that statement will result in javadoc using that comment for all of the fields.
The documentation comment on lines 26–30 describes the `Time` constructor. Tag `@param` describes a parameter to the constructor. Parameters appear in the HTML document in a **Parameters:** note (Fig. M.3) that is followed by a list of all parameters specified.
with the @param tag. For this constructor, the parameter’s name is \texttt{h} and its description is “the \texttt{hour}”. Tag @param can be used only with methods and constructors.

The @throws tag specifies the exceptions thrown by this constructor. Like @param tags, @throws tags are only used with methods and constructors. One @throws should be supplied for each type of exception thrown by the method.

Documentation comments can contain multiple @param and @see tags. The documentation comment on lines 70–80 describes method \texttt{setTime}. The HTML generated for this method is shown in Fig. M.4. Three @param tags describe the method’s parameters. This results in one Parameters: note which lists the three parameters. Methods \texttt{setHour}, \texttt{setMinute} and \texttt{setSecond} are tagged with @see to create hyperlinks to their descriptions in the HTML document. A # character is used instead of a dot when tagging a method or a field. This creates a link to the field or method name that follows the # character. We demonstrate three different ways (i.e., the fully qualified name, the class name qualification and no qualification) to tag methods using @see on lines 76–78. Line 76 uses the fully qualified name to tag the \texttt{setHour} method. If the fully qualified name is not given (lines 77 and 78), javadoc looks for the specified method or field in the following order: current class, superclasses, package and imported files.

The only other tag used in this file is @return, which specifies a Returns: note in the HTML documentation (Fig. M.5). The comment on lines 127–130 documents method \texttt{getHour}. Tag @return describes a method’s return type to help the programmer understand how to use the return value of the method. By javadoc convention, programmers
typeset source code (i.e., keywords, identifiers and expressions) with the HTML tags `<code>` and `</code>`. Several other javadoc tags are briefly summarized in Fig. M.6.

**Good Programming Practice M.1**

Changing source code fonts in javadoc tags helps code names stand out from the rest of the description.

<table>
<thead>
<tr>
<th>javadoc tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@deprecated</code></td>
<td>Adds a <code>Deprecated</code> note. These are notes to programmers indicating that they should not use the specified features of the class. <code>Deprecated</code> notes normally appear when a class has been enhanced with new and improved features, but older features are maintained for backwards compatibility.</td>
</tr>
<tr>
<td><code>[@link]</code></td>
<td>This allows the programmer to insert an explicit hyperlink to another HTML document.</td>
</tr>
<tr>
<td><code>@since</code></td>
<td>Adds a <code>Since:</code> note. These notes are used for new versions of a class to indicate when a feature was first introduced. For example, the Java API documentation uses this to indicate features that were introduced in Java 1.5.</td>
</tr>
<tr>
<td><code>@version</code></td>
<td>Adds a <code>Version</code> note. These notes help maintain version number of the software containing the class or method.</td>
</tr>
</tbody>
</table>

**Fig. M.5** | HTML documentation for method `getHour`.  

**Fig. M.6** | Other javadoc tags.

### M.4 javadoc

In this section, we discuss how to execute the javadoc tool on a Java source file to create HTML documentation for the class in the file. Like other tools, javadoc is executed from the command line. The general form of the javadoc command is

```
jadoc options packages sources @files
```

where `options` is a list of command-line options, `packages` is a list of packages the user would like to document, `sources` is a list of java source files to document and `@files` is a list of text
files containing the javadoc options, the names of packages and/or source files to send to the javadoc utility. [Note: All items are separated by spaces and @files is one word.] Figure M.7 shows a Command Prompt window containing the javadoc command we typed to generate the HTML documentation. For detailed information on the javadoc command, visit the javadoc reference guide and examples at java.sun.com/j2se/5.0/docs/tooldocs/windows/javadoc.html.

In Fig. M.7, the -d option specifies the directory (e.g., docs within the current folder) where the HTML files will be stored on disk. We use the -link option so that our documentation links to Sun’s documentation (installed in the docs directory within the JDK’s installation directory). If the Sun documentation located in a different directory, specify that directory here; otherwise, you’ll receive an error from the javadoc tool. This creates a hyperlink between our documentation and Sun’s documentation (see Fig. M.4, where Java class Exception from package java.lang is hyperlinked). Without the -link argument, Exception appears as text in the HTML document—not a hyperlink to the Java API documentation for class Exception. The -author option instructs javadoc to process the @author tag (it ignores this tag by default).

M.5 Files Produced by javadoc

In the last section, we executed the javadoc tool on the Time.java file. When javadoc executes, it displays the name of each HTML file it creates (see Fig. M.7). From the source file, javadoc created an HTML document for the class named Time.html. If the source file contains multiple classes or interfaces, a separate HTML document is created for each
class. Because class Time belongs to a package, the page will be created in the directory docs\com\deitel\jhtp3\appenH (on Windows platforms). The docs directory was specified with the -d command line option of javadoc, and the remaining directories were created based on the package statement.

Another file that javadoc creates is index.html. This is the starting HTML page in the documentation. To view the documentation you generate with javadoc, load index.html into your web browser. In Fig. M.8, the right frame contains the page index.html and the left frame contains the page allclasses-frame.html which contains links to the source code’s classes. [Note: Our example does not contain multiple packages, so there’s no frame listing the packages. Normally this frame would appear above the left frame (containing “All Classes”), as in Fig. M.2.]

Figure M.9 shows class Time’s index.html. Click Time in the left frame to load the Time class description. The navigation bar (at the top of the right frame) indicates which HTML page is currently loaded by highlighting the page’s link (e.g., the Class link).
Clicking the Tree link (Fig. M.10) displays a class hierarchy for all the classes displayed in the left frame. In our example, we documented only class Time—which extends Object. Clicking the Deprecated link loads deprecated-list.html into the right frame. This page contains a list of all deprecated names. Because we did not use the @deprecated tag in this example, this page does not contain any information.

![Fig. M.10](image)

**Fig. M.10** | Tree page.

Clicking the Index link loads the index-all.html page (Fig. M.11), which contains an alphabetical list of all classes, interfaces, methods and fields. Clicking the Help link loads helpdoc.html (Fig. M.12). This is a help file for navigating the documentation. A default help file is provided, but the programmer can specify other help files.

![Fig. M.11](image)

**Fig. M.11** | Index page.

Among the other files generated by javadoc are serialized-form.html which documents Serializable and Externalizable classes and package-list, a text file rather than an HTML file, which lists package names and is not actually part of the documenta-
tion. The package-list file is used by the -link command-line argument to resolve the external cross references, i.e., allows other documentations to link to this documentation.

Fig. M.12 | Help page.