Creating Documentation with `javadoc`

K.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. K.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

```
java.sun.com/javase/6/docs/technotes/guides/javadoc/index.html
```

K.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 the 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 <B>MySort</B> algorithm */
```

contains the HTML bold tags `<B>` and `</B>`. In the generated HTML files, `MySort` will appear in bold. As we will 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.

K.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.

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 (Fig. K.2). The **See Also:** note specifies other related classes that may be of interest to a programmer using this class. The `@author` tag

```
1  // Fig. H.1: Time.java
2  // Time class declaration with set and get methods.
3  package com.deitel.jhtp6.appenH; // place Time in a package
4
5  /**
6   * This class maintains the time in 24-hour format.
7   * @see java.lang.Object
8   * @author Deitel & Associates, Inc.
9   */
10 public class Time
11 {
12    private int hour;   // 0 - 23
13    private int minute; // 0 - 59
14    private int second; // 0 - 59
15
16    /**
17     *  Time no-argument constructor initializes each instance variable
18     *  to zero. This ensures that Time objects start in a consistent
19     *  state. @throws Exception In the case of an invalid time
20     */
21    public Time() throws Exception
22    {
23       this( 0, 0, 0 ); // invoke Time constructor with three arguments
24    } // end no-argument Time constructor
25
26    /**
27     *  Time constructor
28     *  @param h the hour
29     *  @throws Exception In the case of an invalid time
30     */
31    public Time( int h ) throws Exception
32    {
33       this( h, 0, 0 ); // invoke Time constructor with three arguments
34    } // end one-argument Time constructor
35
```

Fig. K.1 | Java source code file containing documentation comments. (Part 1 of 4.)

```
36    /**
37     *  Time constructor
38     *  @param h the hour
39     *  @param m the minute
40     *  @throws Exception In the case of an invalid time
41     */
42    public Time( int h, int m ) throws Exception
43    {
44       this( h, m, 0 ); // invoke Time constructor with three arguments
45    } // end two-argument Time constructor
46
47    /**
48     *  Time constructor
49     * @param h the hour
50     * @param m the minute
51     * @param s the second
52     * @throws Exception In the case of an invalid time
53     */
54    public Time( int h, int m, int s ) throws Exception
55    {
56       setTime( h, m, s ); // invoke setTime to validate time
57    } // end three-argument Time constructor
58
59    /**
60     *  Time constructor
61     *  @param time A Time object with which to initialize
62     *  @throws Exception In the case of an invalid time
63     */
64    public Time( Time time ) throws Exception
65    {
66       // invoke Time constructor with three arguments
67       this( time.getHour(), time.getMinute(), time.getSecond() );
68    } // end Time constructor with Time argument
69
70    /**
71     *  Set a new time value using universal time. Perform
72     *  validity checks on the data. Set invalid values to zero.
73     *  @param h the hour
74     *  @param m the minute
75     *  @param s the second
76     *  @see com.deitel.jhtp6.appenH.Time#setHour
77     *  @see Time#setMinute
78     *  @see #setSecond
79     *  @throws Exception In the case of an invalid time
80     */
81    public void setTime( int h, int m, int s ) throws Exception
82    {
83       setHour( h );   // set the hour
84       setMinute( m ); // set the minute
85       setSecond( s ); // set the second
86    } // end method setTime
87
```

Fig. K.1 | Java source code file containing documentation comments. (Part 2 of 4.)

```
88    /**
89     *  Sets the hour.
90     *  @param h the hour
91     *  @throws Exception In the case of an invalid time
92     */
93    public void setHour( int h ) throws Exception
94    {
95       if ( h >= 0 && h < 24 )
96          hour = h;
97       else
98          throw( new Exception() );
99    } // end method setHour
100
101   /**
102    *  Sets the minute.
103    *  @param m the minute
104    *  @throws Exception In the case of an invalid time
105    */
106   public void setMinute( int m ) throws Exception
107   {
108      if ( m >= 0 && m < 60 )
109         minute = m;
110      else
111         throw( new Exception() );
112   } // end method setMinute
113
114   /**
115    *  Sets the second.
116    *  @param s the second.
117    *  @throws Exception In the case of an invalid time
118    */
119   public void setSecond( int s ) throws Exception
120   {
121      if ( s >= 0 && s < 60 )
122         second = s;
123      else
124         throw( new Exception() );
125   } // end method setSecond
126
127   /**
128    *  Gets the hour.
129    *  @return an <code>integer</code> specifying the hour.
130    */
131   public int getHour()
132   {
133      return hour;
134   } // end method getHour
135
136   /**
137    *  Gets the minute.
138    *  @return an <code>integer</code> specifying the minute.
139    */
```

Fig. K.1 | Java source code file containing documentation comments. (Part 3 of 4.)

```
140    public int getMinute()
141    {
142       return minute;
143    } // end method getMinute
144
145    /**
146     *  Gets the second.
147     *  @return an <code>integer</code> specifying the second.
148     */
149    public int getSecond()
150    {
151       return second;
152    } // end method getSecond
153
154    /**
155     *  Convert to String in universal-time format
156     *  @return a <code>String</code> representation
157     *  of the time in universal-time format
158     */
159    public String toUniversalString()
160    {
161       return String.format(
162          "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
163    } // end method toUniversalString
164
165    /**
166     *  Convert to String in standard-time format
167     *  @return a <code>String</code> representation
168     *  of the time in standard-time format
169     */
170    public String toStandardString()
171    {
172       return String.format( "%d:%02d:%02d %s",
173          ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 ),
174          getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
175    } // end method toStandardString
176 } // end class Time
```

Fig. K.1 | Java source code file containing documentation comments. (Part 4 of 4.)

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 is also discarded.]

Note that 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.).

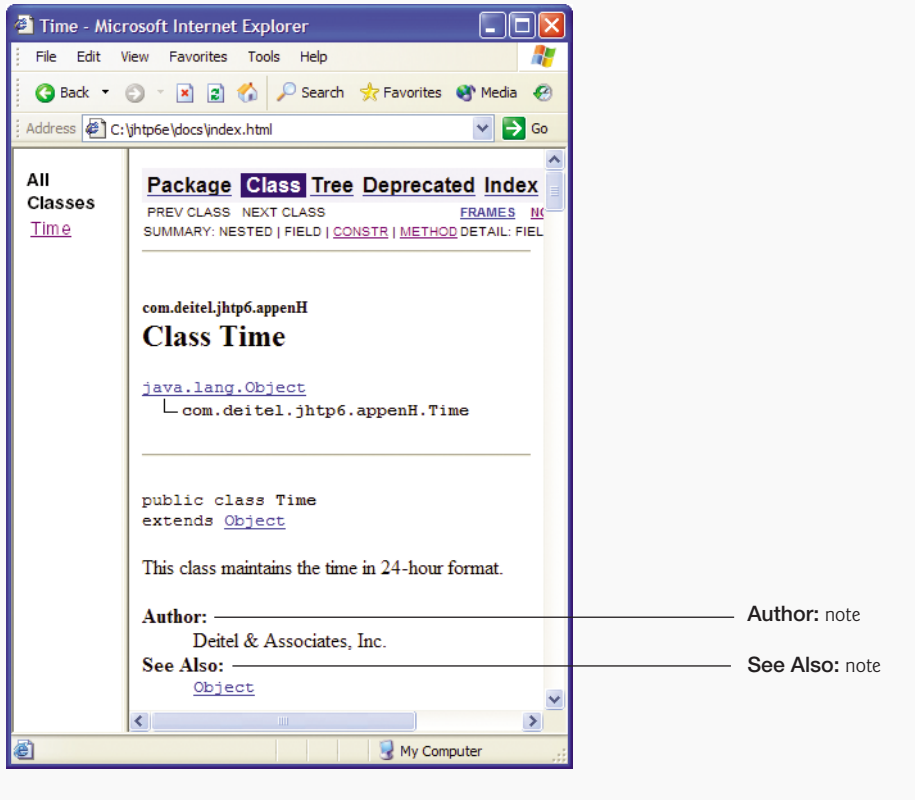

Fig. K.2 | **Author:** and **See Also:** notes generated by `javadoc`.

Common Programming Error K.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 K.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.

Software Engineering Observation K.2

To produce proper `javadoc` documentation, you must declare every instance variable on a separate line.

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. K.3) that is followed by a list of all parameters specified with the `@param` tag. For this constructor, the parameter's name is `h` and its description is `"the hour"`. Tag `@param` can be used only with methods and constructors.

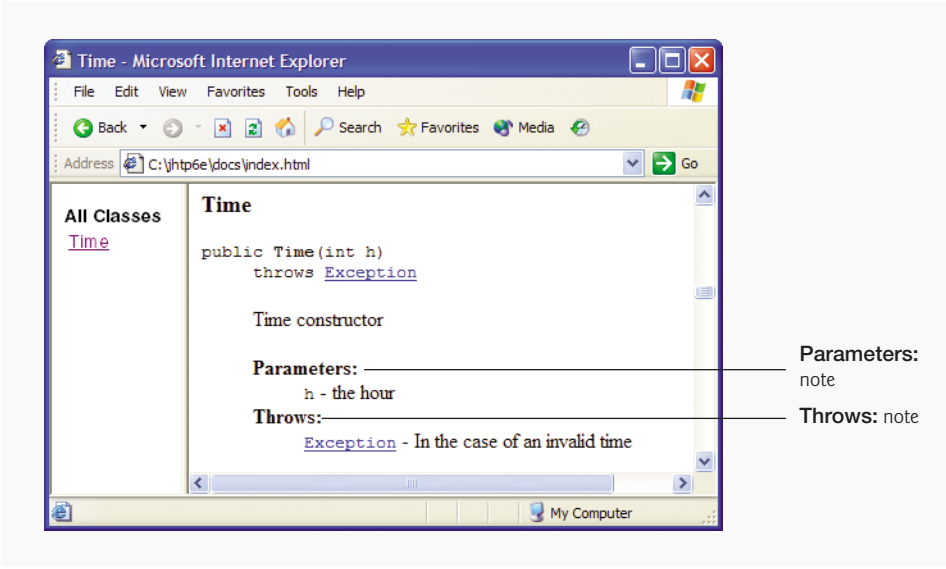

Fig. K.3 | **Parameters:** and **Throws:** note generated by `javadoc`.

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 `setTime`. The HTML generated for this method is shown in Fig. K.4. Three `@param` tags describe the method's parameters. This results in one **Parameters:** note which lists the three parameters. Methods `setHour`, `setMinute` and `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 `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. K.5). The comment on lines 127–130 documents method `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. K.6.

Good Programming Practice K.1

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

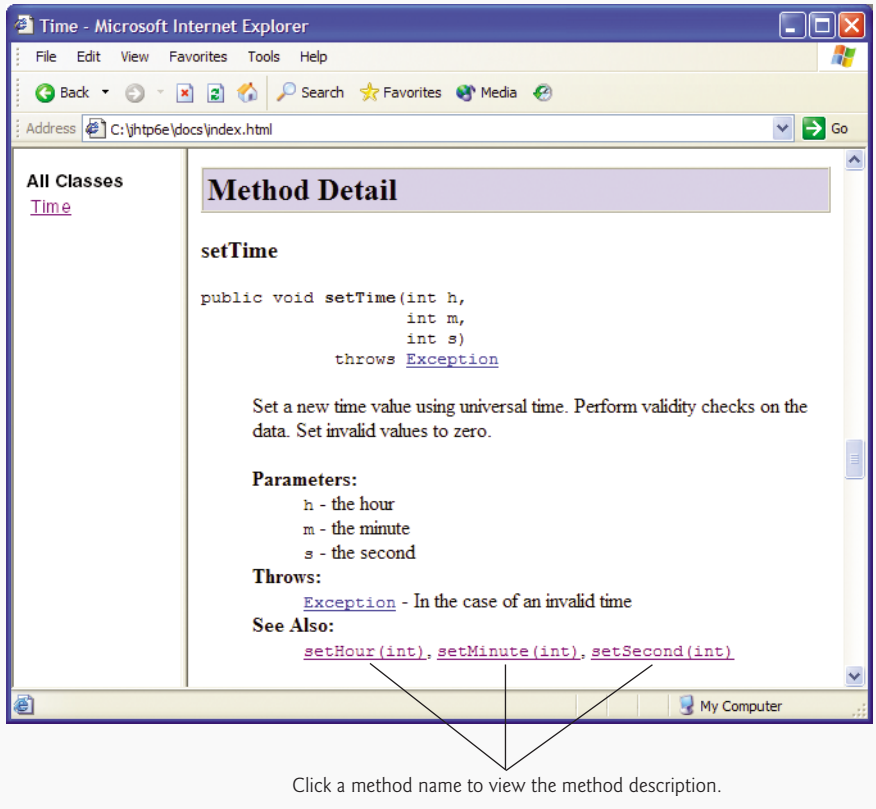

Fig. K.4 | HTML documentation for method `setTime`.

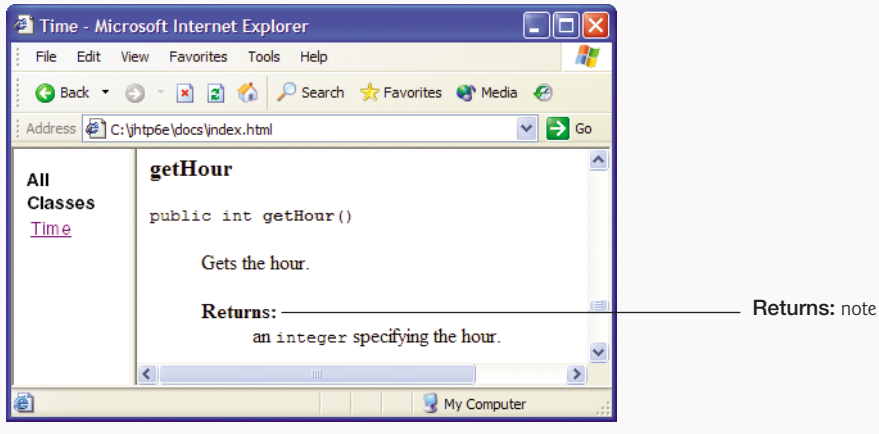

Fig. K.5 | HTML documentation for method `getHour`.

javadoc tag	Description
`@deprecated`	Adds a **Deprecated** note. These are notes to programmers indicating that they should not use the specified features of the class. **Deprecated** notes normally appear when a class has been enhanced with new and improved features, but older features are maintained for backwards compatibility.
`{@link}`	This allows the programmer to insert an explicit hyperlink to another HTML document.
`@since`	Adds a **Since:** 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.
`@version`	Adds a **Version** note. These notes help maintain version number of the software containing the class or method.

Fig. K.6 | Other `javadoc` tags.

K.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

`javadoc` *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 K.7 shows a **Command Prompt** window containing the `javadoc` command we typed to generate the HTML documentation. For detailed information on the `javadoc`

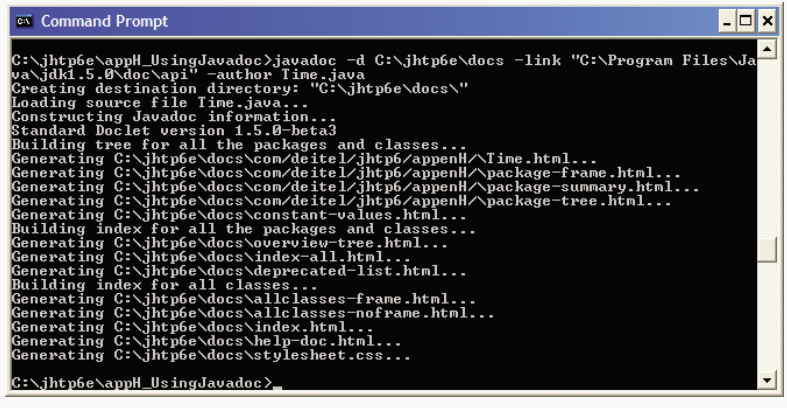

Fig. K.7 | Using the `javadoc` tool.

command, visit the `javadoc` reference guide and examples at `java.sun.com/j2se/5.0/docs/tooldocs/windows/javadoc.html`.

In Fig. K.7, the `-d` option specifies the directory (e.g., `C:\jhtp6\docs`) 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 `C:\Program Files\java\jdk1.5.0\docs` directory). If the Sun documentation located in a different directory, specify that directory here; otherwise, you will receive an error from the `javadoc` tool. This creates a hyperlink between our documentation and Sun's documentation (see Fig. K.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).

K.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. K.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, so the page will be created in the directory `C:\jhtp6\docs\com\deitel\jhtp3\appenH` (on Windows platforms). The `C:\jhtp6\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. K.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 is no frame listing the packages. Normally this frame would appear above the left frame (containing "All Classes"), as in Fig. K.1]

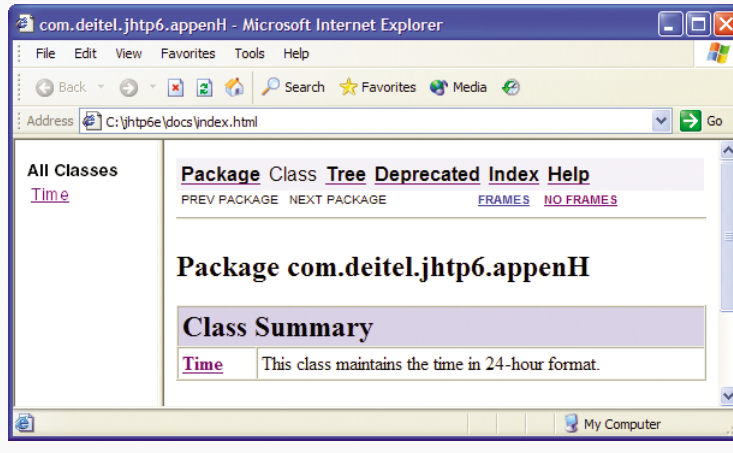

Fig. K.8 | **Index** page.

Figure K.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. K.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.

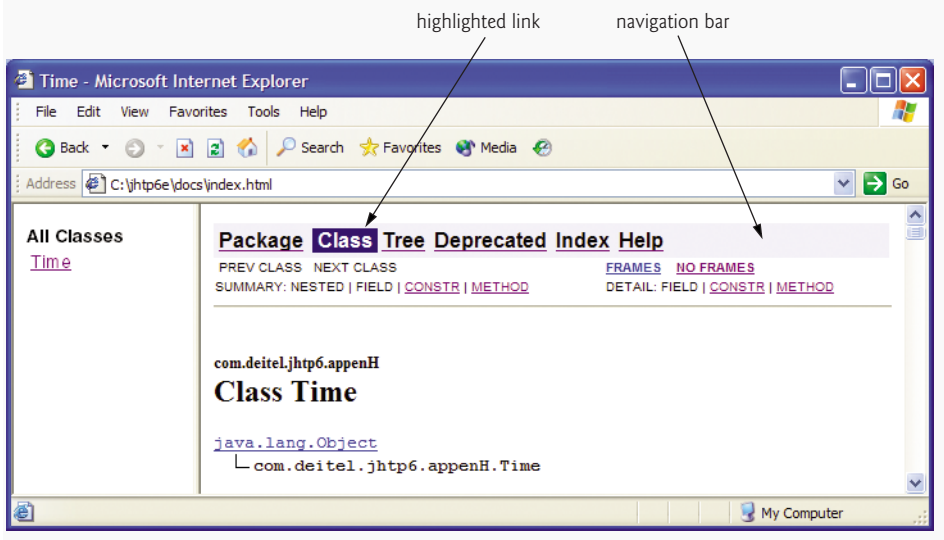

Fig. K.9 | **Class** page.

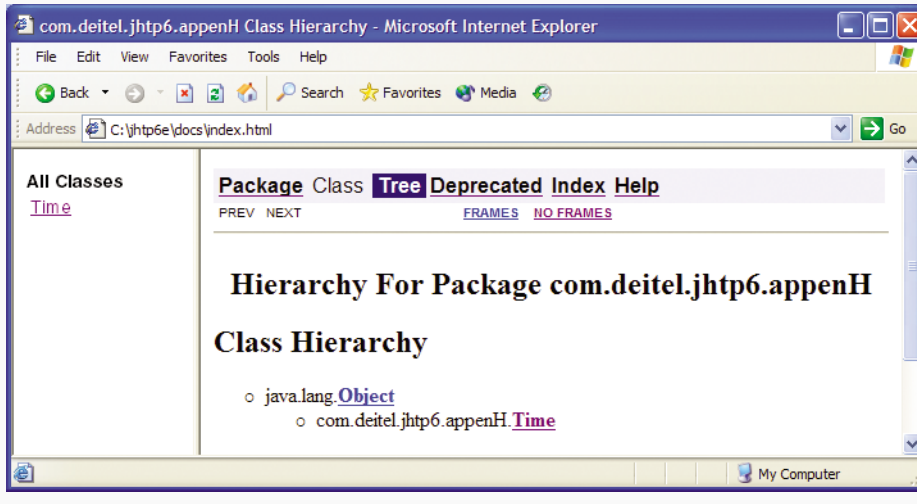

Fig. K.10 | **Tree** page.

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

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 documentation. 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.

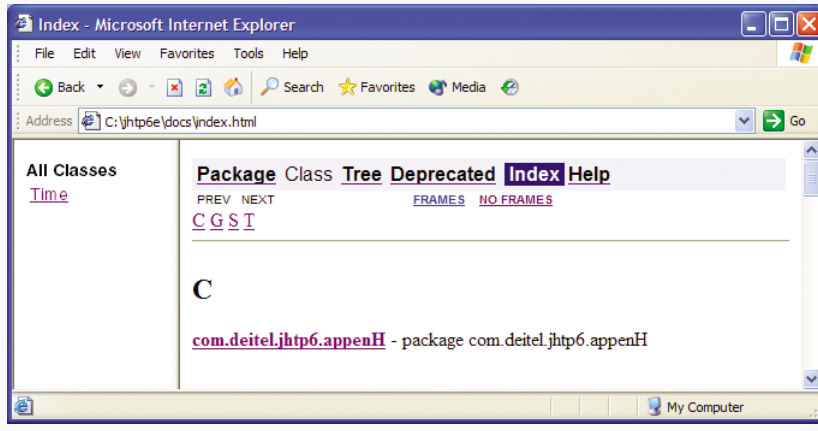

Fig. K.11 | **Index** page.

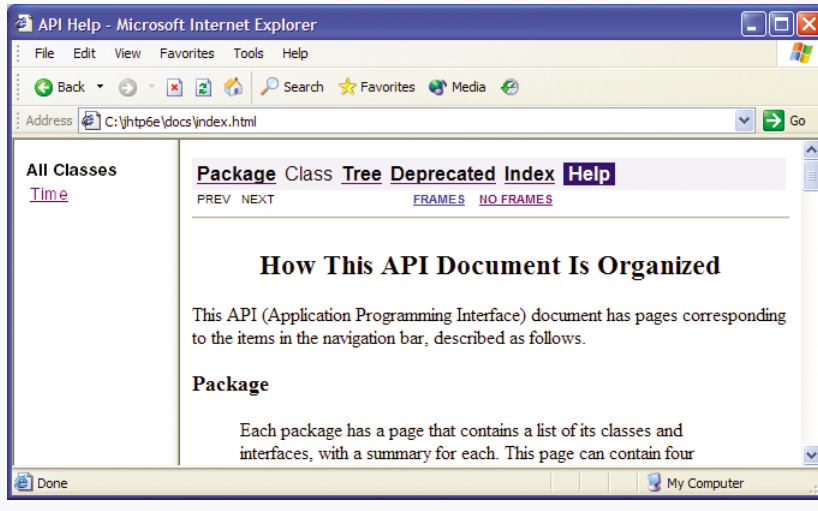

Fig. K.12 | **Help** page.