G

Creating Documentation with `javadoc`

G.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. G.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

```
http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/
    index.html
```

G.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.

G.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.

```java
1   // Fig. G.1: Time.java
2   // Time class declaration with overloaded constructors.
3   package com.deitel; // 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 state
19      *   @throws IllegalArgumentException In the case of an invalid time
20      */
21     public Time()
22     {
23        this(0, 0, 0); // invoke constructor with three arguments
24     }
25
26     /**
27      *   Time constructor
28      *   @param hour the hour
29      *   @throws Exception In the case of an invalid time
30      */
31     public Time(int hour)
32     {
33        this(hour, 0, 0); // invoke constructor with three arguments
34     }
35
36     /**
37      *   Time constructor
38      *   @param hour the hour
39      *   @param minute the minute
40      *   @throws IllegalArgumentException In the case of an invalid time
41      */
42     public Time(int hour, int minute)
43     {
44        this(hour, minute, 0); // invoke constructor with three arguments
45     }
```

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

```
46
47        /**
48         *   Time constructor
49         * @param hour the hour
50         * @param minute the minute
51         * @param second the second
52         * @throws IllegalArgumentException In the case of an invalid time
53         */
54        public Time(int hour, int minute, int second)
55        {
56           if (hour < 0 || hour >= 24)
57              throw new IllegalArgumentException("hour must be 0-23");
58
59           if (minute < 0 || minute >= 60)
60              throw new IllegalArgumentException("minute must be 0-59");
61
62           if (second < 0 || second >= 60)
63              throw new IllegalArgumentException("second must be 0-59");
64
65           this.hour = hour;
66           this.minute = minute;
67           this.second = second;
68        }
69
70        /**
71         *   Time constructor
72         *   @param time A Time object with which to initialize
73         *   @throws IllegalArgumentException In the case of an invalid time
74         */
75        public Time(Time time)
76        {
77           // invoke constructor with three arguments
78           this(time.getHour(), time.getMinute(), time.getSecond());
79        }
80
81        /**
82         *   Set a new time value using universal time. Perform
83         *   validity checks on the data. Set invalid values to zero.
84         *   @param hour the hour
85         *   @param minute the minute
86         *   @param second the second
87         *   @see com.deitel.Time#setHour
88         *   @see Time#setMinute
89         *   @see #setSecond
90         *   @throws Exception In the case of an invalid time
91         */
92        public void setTime(int hour, int minute, int second)
93        {
94           if (hour < 0 || hour >= 24)
95              throw new IllegalArgumentException("hour must be 0-23");
96
97           if (minute < 0 || minute >= 60)
98              throw new IllegalArgumentException("minute must be 0-59");
```

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

```
 99
100            if (second < 0 || second >= 60)
101                throw new IllegalArgumentException("second must be 0-59");
102
103            this.hour = hour;
104            this.minute = minute;
105            this.second = second;
106        }
107
108        /**
109         *  Sets the hour.
110         *  @param hour the hour
111         *  @throws IllegalArgumentException In the case of an invalid hour
112         */
113        public void setHour(int hour)
114        {
115            if (hour < 0 || hour >= 24)
116                throw new IllegalArgumentException("hour must be 0-23");
117
118            this.hour = hour;
119        }
120
121        /**
122         *  Sets the minute.
123         *  @param minute the minute
124         *  @throws IllegalArgumentException In the case of an invalid minute
125         */
126        public void setMinute(int minute)
127        {
128            if (minute < 0 && minute >= 60)
129                throw new IllegalArgumentException("minute must be 0-59");
130
131            this.minute = minute;
132        }
133
134        /**
135         *  Sets the second.
136         *  @param second the second.
137         *  @throws Exception In the case of an invalid second
138         */
139        public void setSecond(int second)
140        {
141            if (second >= 0 && second < 60)
142                throw new IllegalArgumentException("second must be 0-59");
143
144            this.second = second;
145        }
146
147        /**
148         *  Gets the hour.
149         *  @return an <code>integer</code> specifying the hour.
150         */
```

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

```
151    public int getHour()
152    {
153       return hour;
154    }
155
156    /**
157     *  Gets the minute.
158     *  @return an <code>integer</code> specifying the minute.
159     */
160    public int getMinute()
161    {
162       return minute;
163    }
164
165    /**
166     *  Gets the second.
167     *  @return an <code>integer</code> specifying the second.
168     */
169    public int getSecond()
170    {
171       return second;
172    }
173
174    /**
175     *  Convert to String in universal-time format
176     *  @return a <code>String</code> representation
177     *  of the time in universal-time format
178     */
179    public String toUniversalString()
180    {
181       return String.format(
182          "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
183    }
184
185    /**
186     *  Convert to String in standard-time format
187     *  @return a <code>String</code> representation
188     *  of the time in standard-time format
189     */
190    public String toString()
191    {
192       return String.format("%d:%02d:%02d %s",
193          ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
194          getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
195    }
196 } // end class Time
```

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

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. G.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.]
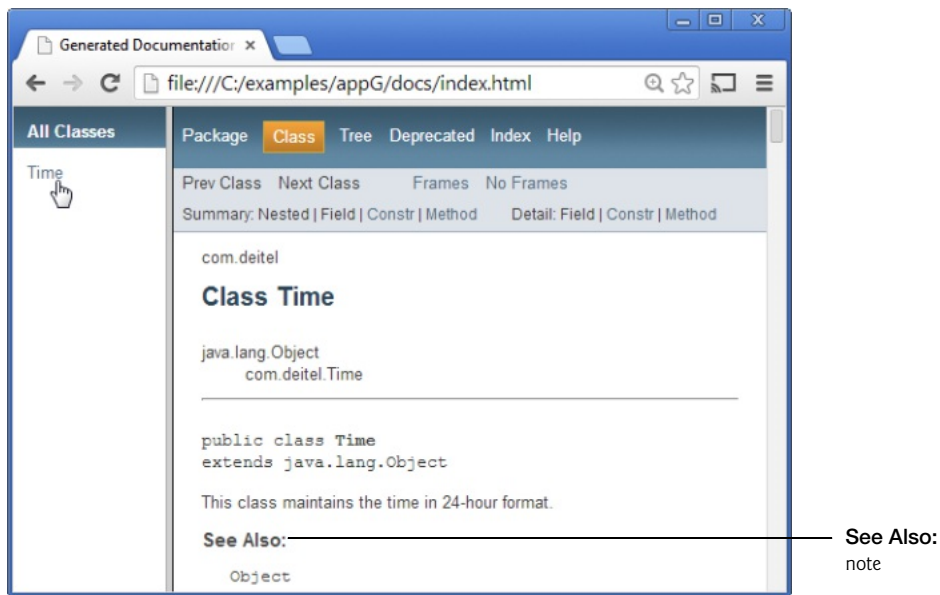
Fig. G.2 | **See Also:** note generated by `javadoc`.

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 G.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 G.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 G.2
> *To produce proper* `javadoc` *documentation, you must declare every instance variable on a separate line.*

The documentation comment on lines 26–30 describes one of the `Time` constructors. Tag **@param** describes a parameter to the constructor. Parameters appear in the HTML document in a **Parameters:** note (Fig. G.3) that is followed by a list of all parameters specified with the @param tag. For this constructor, the parameter's name is `hour` and its description is `"the hour"`. Tag @param can be used only with methods and constructors.
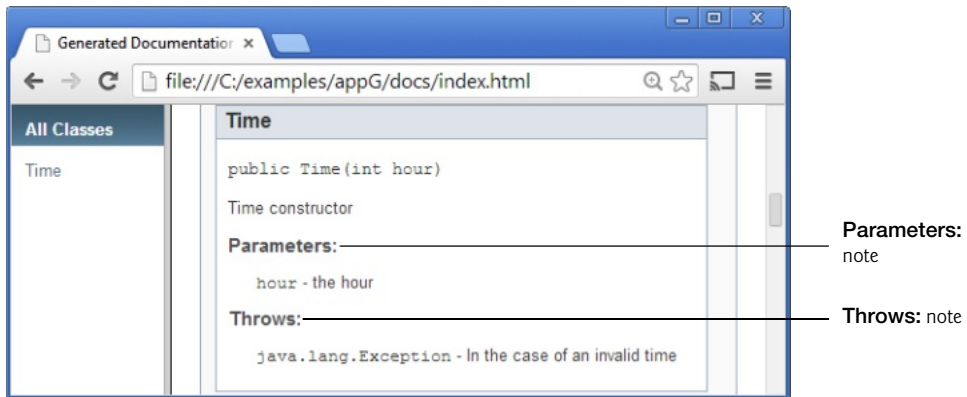
Fig. G.3 | **Parameters:** and **Throws:** notes generated by `javadoc`.

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 81–91 describes method `setTime`. The HTML generated for this method is shown in Fig. G.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 87–89. Line 87 uses the fully qualified name to tag the `setHour` method. If the fully qualified name is not given (lines 88 and 89), `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. G.5). The comment on lines 147–150 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. G.7.
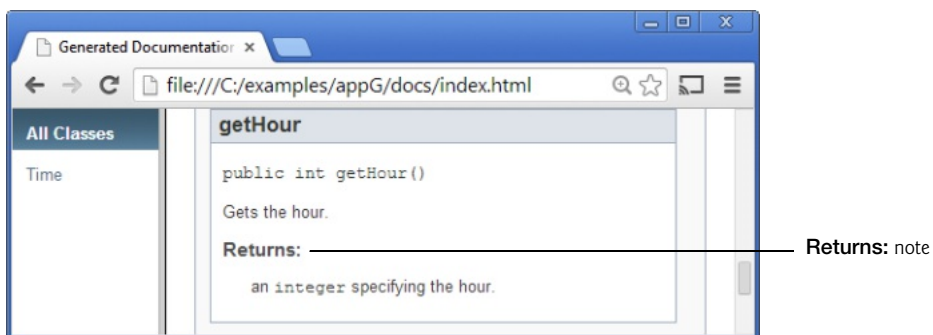
Good Programming Practice G.1

Changing source code fonts in `javadoc` tags helps code names stand out from the rest of the description.
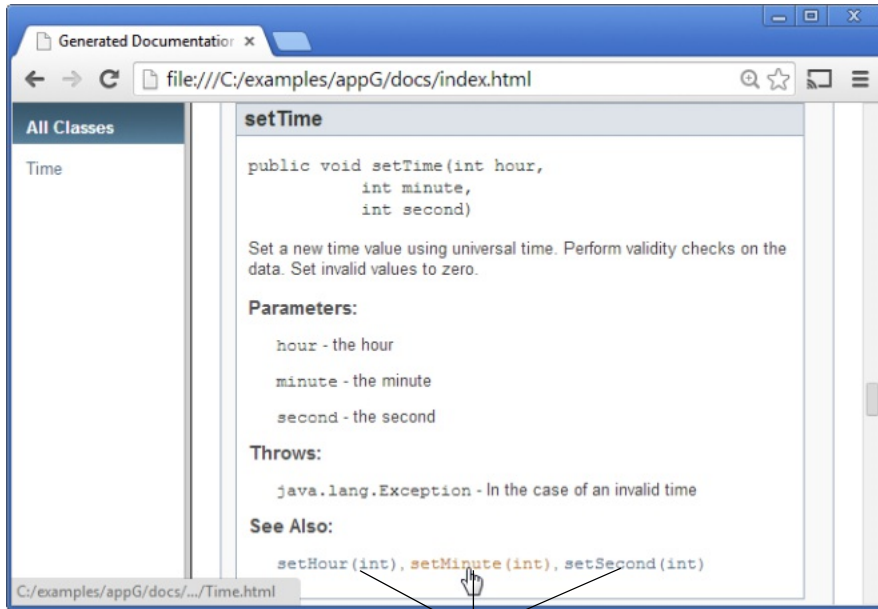
Click a method name to view the method description.

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

Returns: note

Fig. G.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. G.6 | Some additional `javadoc` tags—the complete list is located at `docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags`.

G.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.

Downloading the Java Documentation

When you generate documentation with `javadoc`, you can link your documentation to the Java API documentation. This is useful when your classes use features of the Java API, such as extending an exiting class. The `javadoc` tool will create links to the existing classes. To link to the Java API documentation, you should first download and extract the documentation from

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```
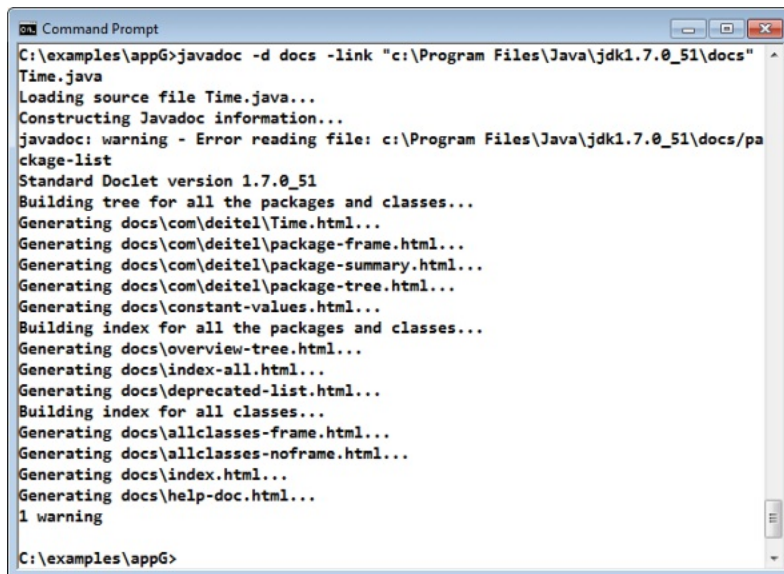
You can find the documentation download under additional resources. Normally, you'd extract the documentation into your JDK's installation folder.

Executing javadoc from the Command Line

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 G.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 `http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html`.

Fig. G.7 │ Using the `javadoc` tool.

In Fig. G.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. G.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).

G.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. G.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
```

The `docs` directory was specified with the `-d` command line option of `javadoc`, and the remaining directories were created based on the `package` statement.

The `javadoc` tool also creates **`index.html`**—the starting HTML page in the docu-mentation. To view the documentation you generate with `javadoc`, load `index.html` from the `docs` directory into your web browser. In Fig. G.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. G.2.]
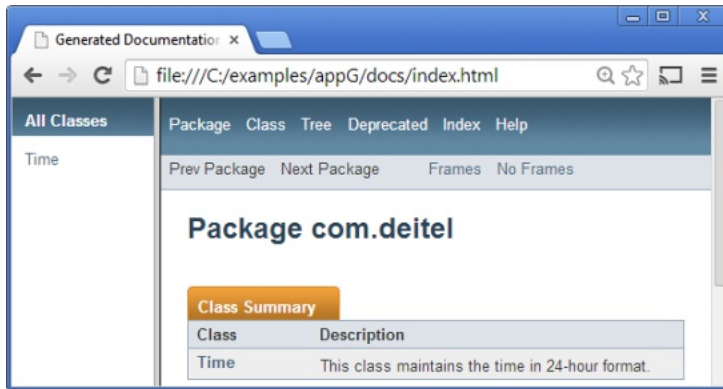

Fig. G.8 | **Index** page.

Figure G.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).
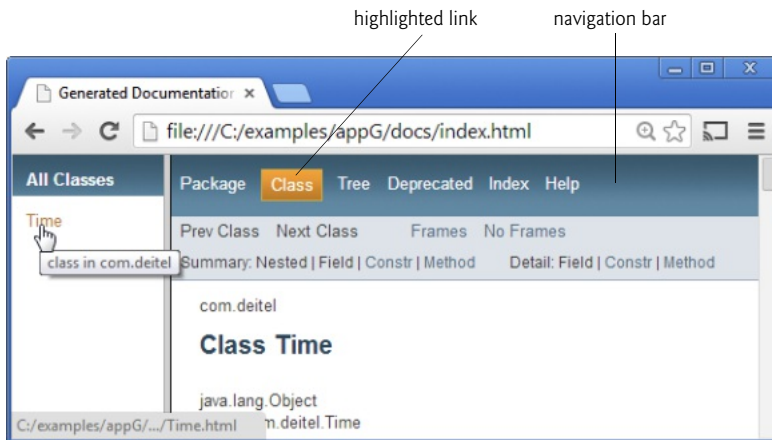

Fig. G.9 | **Class** page.

Clicking the **Tree** link (Fig. G.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.
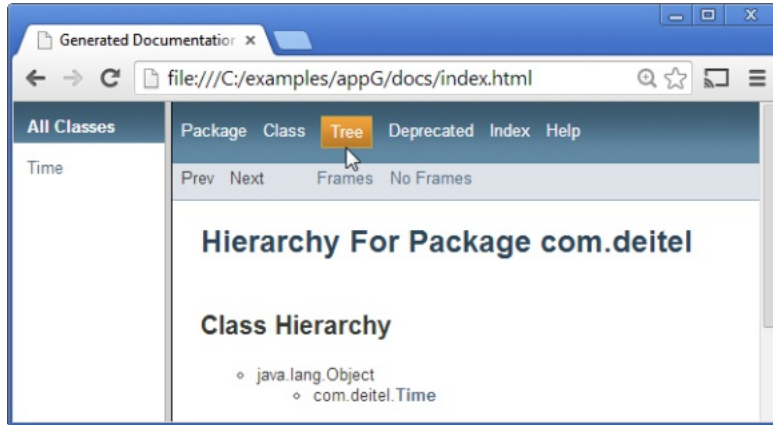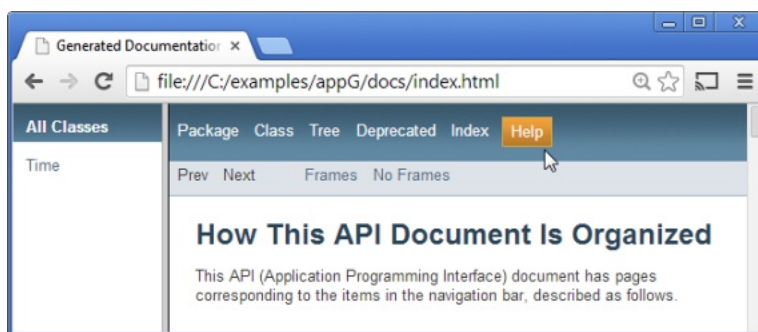

Fig. G.10 | **Tree** page.

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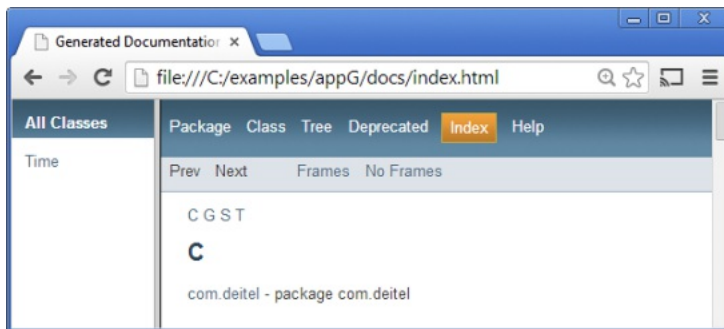
Fig. G.11 | **Index** page.

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 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. G.12 | **Help** page.