

javadoc

So far we have used only two utilities in the JDK. These are `javac`, the Java compiler, and `java`, the Java run-time interpreter that executes our java programs. In this section, we introduce another useful utility in the JDK, `javadoc`.

`javadoc` is used to create HTML documentation for Java methods, classes, and packages. If you have viewed the Java API documentation using a web browser, then you are already familiar with `javadoc`'s capabilities. The Java API documentation was created with `javadoc`. To create equally as spiffy documentation for your programs, read on.

Documentation comments, usually called *doc comments*, start with the three characters `/**` and continue until the next `*/`. Each doc comment describes the element of the identifier that immediately follows. The identifier typically represents a Java method, class, or field. Leading `*` characters are ignored, as are newlines and whitespaces preceding the `*` on each line.

The first sentence of a doc comment is special. Everything up to the first period is the “summary” of the identifier. This portion of the doc comment appears in the summary portion of the identifier’s documentation. It is usually a succinct statement of the identifier’s purpose. The summary, as well as everything following the summary, appears in the detailed description of the identifier.

At this point, it’s probably useful to introduce an example. A good starting point, is to show the output of `javadoc` for a Java program that has no special comments added. Figure 1 is a Java program that outputs a quotation, randomly selected from an array.

```

1  import java.util.*;
2
3  public class QuoteOfTheDay
4  {
5      // quotes are a good candidate for an array of string constants
6      private static final String[] QUOTE =
7      {
8          "\"Fuddle duddle\", Trudeau",
9          "\"Fly on, little wing\", Hendrix",
10         "\"Yubba dubba do\", Fred Flintstone",
11         "\"Blood, sweat, and tears\", Churchill",
12         "\"I didn't inhale\", Clinton",
13         "\"Look up, look way up\", Friendly Giant",
14         "\"It ain't over 'til it's over\", Yogi Berra",
15     };
16
17     public static void main(String[] args)
18     {
19         // instantiate a Random object
20         Random quoteNumber = new Random();
21
22         // get a random integer
23         int i = quoteNumber.nextInt();
24
25         // make sure it's zero or positive
26         i = Math.abs(i);
27
28         // constrain the index, as per the number of quotes
29         i = i % QUOTE.length;
30
31         // output today's quote
32         System.out.println("Today's quote...");
33         System.out.println("\n\t" + QUOTE[i]);
34     }
35 }

```

Figure 1. QuoteOfTheDay.java

There is nothing in `QuoteOfTheDay` that we haven't already discussed. Note, however, the frequent use of the escape sequence `\'` (lines 8-14) to frame each quote with single quotation marks. As sample dialogue follows (user input is underlined):

```

PROMPT>java QuoteOfTheDay
Today's quote...

        'Yubba dubba do', Fred Flintstone

```

OK, let's move on to `javadoc`. To build HTML documentation for `QuoteOfTheDay`, the source file is processed by `javadoc` as follows:

```

PROMPT>javadoc QuoteOfTheDay

```

`javadoc` processes `QuoteOfTheDay.java`, according to an elaborate set of rules, and creates an HTML output file, `QuoteOfTheDay.html`. If you have a taste for HTML code, have a peak in `QuoteOfTheDay.html`. We won't discuss the HTML code here, however.

If QuoteOfTheDay.html is opened from a browser, such as Netscape, the result is a web page reminiscent of those in the Java API documentation (see Figure 2).

The screenshot shows the Java API documentation for the `QuoteOfTheDay` class. At the top, there are navigation links: [Class](#), [Tree](#), [Deprecated](#), [Index](#), and [Help](#). Below these are links for [PREV CLASS](#), [NEXT CLASS](#), [FRAMES](#), and [NO FRAMES](#). There are also links for [SUMMARY: INNER | FIELD | CONSTR | METHOD](#) and [DETAIL: FIELD | CONSTR | METHOD](#).

Class QuoteOfTheDay

`java.lang.Object`
|
+--`QuoteOfTheDay`

```
public class QuoteOfTheDay
    extends java.lang.Object
```

Constructor Summary

[QuoteOfTheDay\(\)](#)

Method Summary

static void	main (<code>java.lang.String[] args</code>)
-------------	---

Methods inherited from class `java.lang.Object`

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Detail

QuoteOfTheDay

```
public QuoteOfTheDay()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Figure 2. QuoteOfTheDay documentation, as viewed in a browser window

Take a few moments to compare the documentation in Figure 2 with the Java source code in Figure 1. The `QuoteOfTheDay` program is itself a class, and, so, its relationship to `Object` is clearly shown. In fact, the relationship is shown twice. At the top of the browser window, we see

the relationship in a class hierarchy diagram. Just below, we see the relationship in the class signature:

```
public class QuoteOfTheDay
    extends java.lang.Object
```

Since `QuoteOfTheDay` “extends” `Object`, it inherits methods of the `Object` class, and these are also shown in the browser window.

With this introduction, let’s move on. It doesn’t make much sense to use `javadoc` on any of our small example programs. More likely, `javadoc` will be used with project files or, at the very least, general purpose classes designed as factories for objects that are instantiated and acted up on in other Java programs.

We’ll use the `City` class for an example. The source code listing of the `City` class shown earlier was stripped bare of comments, just to shorten the illustration. However, we now want the full story on doc comments. Figure 3 shows the complete listing of the source file `City.java`.

```

1
2  /**
3   * A <code>City</code> object defines a city with a
4   * name/pop pair where 'name' is the name of the city
5   * as a <code>String</code>
6   * and 'pop' is the city's population as an <code>int</code>.
7   *
8   * @version 1.0
9   * @author Scott MacKenzie
10  */
11
12 public class City implements Comparable
13 {
14     /** The city name. */
15     private String name;
16     /** The city population. */
17     private int pop;
18
19     /**
20      * Creates a new city with the specified name and
21      * population.
22      */
23     public City(String name, int pop)
24     {
25         this.name = name;
26         this.pop = pop;
27     }
28
29     /**
30      * Creates a new city with the specified name and
31      * an initial population of 0.
32      */
33     public City(String name)
34     {
35         this.name = name;
36         this.pop = 0;
37     }
38
39     /** Returns the city's name.
40      */
41     public String getName()      { return name; }
42
43     /** Returns the city's population.
44      * @return a <code>int</code> containing the city's population
45      */
46     public int getPop()         { return pop; }
47
48     /** Set the city's population to the specified value.
49      * @param newPop an <code>int</code> representing the
50      * new population
51      * of the city
52      */
53     public void setPop(int newPop) { pop = newPop; }
54
55     /** Returns a string representing the city. */
56     public String toString()

```

```

57     {
58         return "City [name: " + name + " pop: " + pop + " ]";
59     }
60
61     /** Tests the <code>City</code> class.
62     *
63     * <p>To test the <code>City</code> class, execute
64     * as follows:<p>
65     * <p>
66     * <p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<code>PROMPT>java City
67     * test_name test_pop</code><p>
68     * <p>
69     * where <code>test_name</code> is the name of a city, and
70     * <code>test_pop</code> is the city's population. The test
71     * will instantiate a <code>City</code> object and then
72     * convert it to a string using the <code>toString()</code>
73     * method.
74     * The string is sent it to
75     * stdout and will appear as follows:<p>
76     * <p>
77     * <p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<code>City [name: test_name pop:
78     * test_pop]</code>
79     */
80     public static void main(String[] args)
81     {
82         if (args.length != 2)
83         {
84             System.out.println("Usage: java City name pop");
85             return;
86         }
87         String testName = args[0];
88         int testPop = Integer.parseInt(args[1]);
89         City c = new City(testName, testPop);
90         System.out.println(c);
91     }
92
93     /** Implements comparable. Allows two city objects
94     * to be compared. Comparison is lexicographical
95     * based on the <code>name</code> field, much the
96     * same as the <code>compareTo()</code> in the
97     * <code>String</code> class.
98     */
99     public int compareTo(Object o)
100    {
101        return (this.name).compareTo(((City)o).name);
102    }
103 }

```

Figure 3. City.java (complete listing)

To build City.html, the following command is used

```
PROMPT>javadoc City
```

The output can be viewed by opening City.html in a browser window. It's a tad long, but the entire page is shown in Figure 4 for your viewing pleasure.

Class City

```
java.lang.Object
|
+--City
```

```
public class City
  extends java.lang.Object
  implements java.lang.Comparable
```

A `City` object defines a city with a name/pop pair where 'name' is the name of the city as a `String` and 'pop' is the city's population as an `int`.

Version:

1.0

Author:

Scott MacKenzie

Constructor Summary

<code>City</code> (<code>java.lang.String</code> name)	Creates a new city with the specified name and an initial population of 0.
<code>City</code> (<code>java.lang.String</code> name, <code>int</code> pop)	Creates a new city with the specified name and population.

Method Summary

<code>int</code>	<code>compareTo</code> (<code>java.lang.Object</code> o) Implements comparable.
<code>java.lang.String</code>	<code>getName</code> () Returns the city's name.
<code>int</code>	<code>getPop</code> () Returns the city's population.
<code>static void</code>	<code>main</code> (<code>java.lang.String[]</code> args) Tests the <code>City</code> class.
<code>void</code>	<code>setPop</code> (<code>int</code> newPop) Set the city's population to the specified value.
<code>java.lang.String</code>	<code>toString</code> () Returns a string representing the city.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

City

```
public City(java.lang.String name,  
            int pop)
```

Creates a new city with the specified name and population.

City

```
public City(java.lang.String name)
```

Creates a new city with the specified name and an initial population of 0.

Method Detail

getName

```
public java.lang.String getName()
```

Returns the city's name.

getPop

```
public int getPop()
```

Returns the city's population.

Returns:

a int containing the city's population

setPop

```
public void setPop(int newPop)
```

Set the city's population to the specified value.

Parameters:

`newPop` - an int representing the new population of the city

toString

```
public java.lang.String toString()
```

Returns a string representing the city.

Overrides:

`toString` in class `java.lang.Object`

main

```
public static void main(java.lang.String[] args)
```

Tests the `City` class.

To test the `City` class, execute as follows:

```
PROMPT>java City test_name test_pop
```

where `test_name` is the name of a city, and `test_pop` is the city's population. The test will instantiate a `City` object and then convert it to a string using the `toString()` method. The string is sent to `stdout` and will appear as follows:

```
City [name: test_name pop: test_pop]
```

compareTo

```
public int compareTo(java.lang.Object o)
```

Implements `Comparable`. Allows two city objects to be compared. Comparison is lexicographical based on the `name` field, much the same as the `compareTo()` in the `String` class.

Specified by:

`compareTo` in interface `java.lang.Comparable`

Class Tree Deprecated Help	
PREV CLASS NEXT CLASS	FRAMES NO FRAMES
SUMMARY : INNER FIELD CONSTR METHOD	DETAIL : FIELD CONSTR METHOD

Figure 4. `City` Class documentation created by `javadoc`

Take a few moments to compare the source code in Figure 3 with with the browser rendering of the HTML code in Figure 4. Note that for the `main()` and `compareTo()` methods, only the first sentence in the source code comment appears in the summary. This is line 61 for `main()` and line 93 for `compareTo()`. The entire comments appear later in the browser window, where the full details of the methods are given.

HTML codes embedded in the source code will be rendered as expected in the browser window. See, in particular, lines 61-79 for the `main()` method and the corresponding appearance of the `main()` method details in the browser window.

One special feature of `javadoc`, which we mention only briefly, is the implementation of “tagged paragraphs” to hold certain kinds of information. The tags begin with the `@` character. Tagged paragraphs are given special treatment when the source code is processed by `javadoc`, and result in marked paragraphs, links to other documents, etc. Four are used in `City.java`. These are `version`, in line 8, `author`, in line 9, `return` in line 44, and `param` in line 49. The effect is seen in the browser window at the expected location. `javadoc` will always process `return` and `param` tagged paragraphs; however, `version` and `author` are only processed if an appropriate command-line argument is used. The exact command used to generate the HTML code in Figure 4 was as follows:

```
PROMPT>javadoc -version -author -noindex City
```

The `-noindex` option disables generation of an `index.html` file.

A complete list of the supported command-line arguments can be viewed by issuing the `javadoc` command without any arguments. We’ll leave this for you to explore.

So, what do you think of `javadoc`? If you’re just writing small Java programs to solve interesting problems for an introductory course on Java, you probably think `javadoc` is little more than a curiosity. However, if you are part of a team developing a large suite of Java tools, then `javadoc` is something to take seriously.

There are few pitfalls worth noting, however. Good programmers are not necessarily good documentation writers. Often an organization has dedicated, talented writing staff to develop documentation for their software tools. The technical writers must have write permission for the Java source code files. This may pose a problem in some organizations, but it is a problem that must be dealt with, because doc comments must be part and parcel of the Java source files.

Another pitfall is known as “document skew”, wherein the code evolves but the comments do not. This is most likely if comments address implementation details (e.g., the details of an algorithm). The best way to avoid this is to limit comments to “contractual information”; that is, information essential to the use of the Java package or class. Details on the implementation are usually of little or no concern to the programmers using the code, and, so, they need not be included in doc comments.