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.
import java.util.*;

public class QuoteOfTheDay{

    // quotes are a good candidate for an array of string constants
    private static final String[] QUOTE = {
        "'Fuddle duddle'", Trudeau",
        "'Fly on, little wing'", Hendrix",
        "'Yubba dubba do'", Fred Flintstone",
        "'Blood, sweat, and tears'", Churchill",
        "'I didn't inhale'", Clinton",
        "'Look up, look way up'", Friendly Giant",
        "'It ain't over 'til it's over'", Yogi Berra",
    }

    public static void main(String[] args) {
        // instantiate a Random object
        Random quoteNumber = new Random();

        // get a random integer
        int i = quoteNumber.nextInt();

        // make sure it's zero or positive
        i = Math.abs(i);

        // constrain the index, as per the number of quotes
        i = i % QUOTE.length;

        // output today's quote
        System.out.println("Today's quote...");
        System.out.println("\n\t" + QUOTE[i]);
    }
}

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).

```
Class QuoteOfTheDay

java.lang.Object
   |
   +---QuoteOfTheDay

public class QuoteOfTheDay
extends java.lang.Object

Constructor Summary

QuoteOfTheDay()

Method Summary

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

Methods inherited from class java.lang.Object:
clon, 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:

```java
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.
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
*

class City implements Comparable
{
    /** The city name. */
    private String name;
    /** The city population. */
    private int pop;

    /**
     * Creates a new city with the specified name and population.
     */
    public City(String name, int pop)
    {
        this.name = name;
        this.pop = pop;
    }

    /**
     * Creates a new city with the specified name and an initial population of 0.
     */
    public City(String name)
    {
        this.name = name;
        this.pop = 0;
    }

    /** Returns the city's name. */
    public String getName() { return name; }

    /** Returns the city's population. */
    public int getPop() { return pop; }

    /** Set the city's population to the specified value. */
    public void setPop(int newPop) { pop = newPop; }

    /** Returns a string representing the city. */
    public String toString()
```java
57    
58        return "City [name: " + name + " pop: " + pop + "]";
59    }
60
61    /** Tests the <code>City</code> class.
62      *
63      * To test the <code>City</code> class, execute
64      * as follows:<p>
65      * &nbsp;&nbsp;&nbsp;&lt;p&gt;&nbsp;&nbsp;&nbsp;&nbsp;&lt;code&gt;java City
66      * test_name test_pop&lt;/code&gt;&lt;p&gt;
67      * &lt;p&gt;
68      * where &lt;code&gt;test_name&lt;/code&gt; is the name of a city, and
69      * &lt;code&gt;test_pop&lt;/code&gt; is the city's population. The test
70      * will instantiate a &lt;code&gt;City&lt;/code&gt; object and then
71      * convert it to a string using the &lt;code&gt;toString()&lt;/code&gt;
72      * method.
73      * The string is sent it to
74      * stdout and will appear as follows:<p>
75      * &nbsp;&nbsp;&nbsp;&lt;p&gt;
76      * &lt;p&gt;
77      * &lt;p&gt;&nbsp;&nbsp;&nbsp;&lt;code&gt;City [name: test_name pop:
78      * test_pop]&lt;/code&gt;
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 &lt;code&gt;name&lt;/code&gt; field, much the
96      * same as the &lt;code&gt;compareTo()&lt;/code&gt; in the
97      * &lt;code&gt;String&lt;/code&gt; 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
java.lang.Object
   └-- City
```

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

A City object defines a city with a name and population 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

- `City(String name)`
  Creates a new City with the specified name and an initial population of 0.

- `City(String name, int pop)`
  Creates a new City with the specified name and population.

### Method Summary

- `compareTo(Object o)`
  Implements Comparable.

- `String getName()`
  Returns the city's name.

- `int getPop()`
  Returns the city's population.

- `void main(String[] args)`
  Tests the City class.

- `void setPop(int newPop)`
  Set the city's population to the specified value.

- `String toString()`
  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**

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

Returns a string representing the city.

**Overrides:**

toString in class java.lang.Object

---

**main**

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

Tests the City class.

To test the City class, execute as follows:

```shell
PRMTP.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:

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

---

**compareTo**

```java
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:**

cmpareTo in interface java.lang.Comparable

---

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 is 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 processes 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.