XML

In this chapter

The preface of the book *Essential XML* by Don Box et al. (Addison-Wesley, 2000) states only half-jokingly: "The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger." Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Many of these libraries have now been integrated into the Java platform.

This chapter introduces XML and covers the XML features of the Java library. As always, we'll point out along the way when the hype surrounding XML is justified—and when you have to take it with a grain of salt and try solving your problems the old-fashioned way: through good design and code.

3.1 Introducing XML

In Chapter 13 of Volume I, you have seen the use of *property files* to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

You can use the `Properties` class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname`/`fontsize` entries in the example. It would be more object-oriented to have a single entry:

```
font=Times Roman 12
```

But then, parsing the font description gets ugly as you have to figure out when the font name ends and the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names like

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Another shortcoming of the property file format is the requirement that keys must be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

The XML format solves these problems. It can express hierarchical structures and is thus more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </body>
  <window>
    <width>400</width>
    <height>200</height>
  </window>
  <color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
  </color>
  <menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
  </menu>
</configuration>
```

The XML format allows you to express the hierarchy and record repeated elements without contortions.

The format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason for that—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation—in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet.

As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

NOTE: You can find a very nice version of the XML standard, with annotations by Tim Bray, at `www.xml.com/axml/axml.html`.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case-sensitive. For example, `<H1>` and `<h1>` are different XML tags.
- In HTML, you can omit end tags, such as `</p>` or `</li>`, if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a /, as in `<img src="coffeecup.png"/>`. That way, the parser knows not to look for a `</img>` tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML. In XML, you have to use quotation marks: `width="300"`.
- In HTML, you can have attribute names without values, such as `<input type="radio" name="language" value="Java" checked>`. In XML, all attributes must have values, such as `checked="true"` or (ugh) `checked="checked"`.

3.1.1 The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

NOTE: Since SGML was created for processing of real documents, XML files are called *documents* even though many of them describe data sets that one would not normally call documents.

The header can be followed by a *document type definition* (DTD), such as

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We will discuss them later in this chapter.

Finally, the body of the XML document contains the *root element*, which can contain other elements. For example,

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
   <title>
      <font>
         <name>Helvetica</name>
         <size>36</size>
      </font>
   </title>
   . . .
</configuration>
```

An element can contain *child elements*, text, or both. In the preceding example, the `font` element has two child elements, `name` and `size`. The `name` element contains the text `"Helvetica"`.

TIP: It is best to structure your XML documents so that an element contains *either* child elements *or* text. In other words, you should avoid situations such as

```
<font>
   Helvetica
   <size>36</size>
</font>
```

This is called *mixed content* in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed content.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>
   <name>Helvetica</name>
   <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, you will have to add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string `"36 pt"`, just the kind of hassle that XML was designed to avoid. Adding an attribute to the `size` element is much cleaner:

```
<font>
   <name>Helvetica</name>
   <size unit="pt">36</size>
</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

NOTE: In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string `Java Technology` is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- *Character references* have the form `&#`*decimalValue*`;` or `&#x`*hexValue*`;`. For example, the é character can be denoted with either of the following:

```
&#233; &#xE9;
```

- *Entity references* have the form `&`*name*`;`. The entity references

```
&lt; &gt; &amp; &quot; &apos;
```

have predefined meanings: the less-than, greater-than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

- *CDATA sections* are delimited by <![CDATA[and]]>. They are a special form of character data. You can use them to include strings that contain characters such as < > & without having them interpreted as markup, for example:

```
<![CDATA[< & > are my favorite delimiters]]>
```

CDATA sections cannot contain the string]]>. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

- *Processing instructions* are instructions for applications that process XML documents. They are delimited by <? and ?>, for example

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Every XML document starts with a processing instruction

```
<?xml version="1.0"?>
```

- *Comments* are delimited by <!-- and -->, for example

```
<!-- This is a comment. -->
```

Comments should not contain the string --. Comments should only be information for human readers. They should never contain hidden commands; use processing instructions for commands.

3.2 Parsing an XML Document

To process an XML document, you need to *parse* it. A parser is a program that reads a file, confirms that the file has the correct format, breaks it up into the constituent elements, and lets a programmer access those elements. The Java library supplies two kinds of XML parsers:

- Tree parsers, such as the Document Object Model (DOM) parser, that read an XML document into a tree structure.
- Streaming parsers, such as the Simple API for XML (SAX) parser, that generate events as they read an XML document.

The DOM parser is easier to use for most purposes, and we explain it first. You may consider a streaming parser if you process very long documents whose tree structures would use up a lot of memory, or if you are only interested in a few

elements and don't care about their context. For more information, see Section 3.6, "Streaming Parsers," on p. 199.

The DOM parser interface is standardized by the World Wide Web Consortium (W3C). The `org.w3c.dom` package contains the definitions of interface types such as `Document` and `Element`. Different suppliers, such as the Apache Organization and IBM, have written DOM parsers whose classes implement these interfaces. The Java API for XML Processing (JAXP) library actually makes it possible to plug in any of these parsers. But the JDK also comes with a DOM parser that is derived from the Apache parser.

To read an XML document, you need a `DocumentBuilder` object that you get from a `DocumentBuilderFactory` like this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

You can now read a document from a file:

```
File f = . . .
Document doc = builder.parse(f);
```

Alternatively, you can use a URL:

```
URL u = . . .
Document doc = builder.parse(u);
```

You can even specify an arbitrary input stream:

```
InputStream in = . . .
Document doc = builder.parse(in);
```

NOTE: If you use an input stream as an input source, the parser will not be able to locate other files that are referenced relative to the location of the document, such as a DTD in the same directory. You can install an "entity resolver" to overcome that problem. See `www.xml.com/pub/a/2004/03/03/catalogs.html` or `www.ibm.com/developerworks/xml/library/x-mxd3.html` for more information.

The `Document` object is an in-memory representation of the tree structure of the XML document. It is composed of objects whose classes implement the `Node` interface and its various subinterfaces. Figure 3.1 shows the inheritance hierarchy of the subinterfaces.

Start analyzing the contents of a document by calling the `getDocumentElement` method. It returns the root element.

```
Element root = doc.getDocumentElement();
```

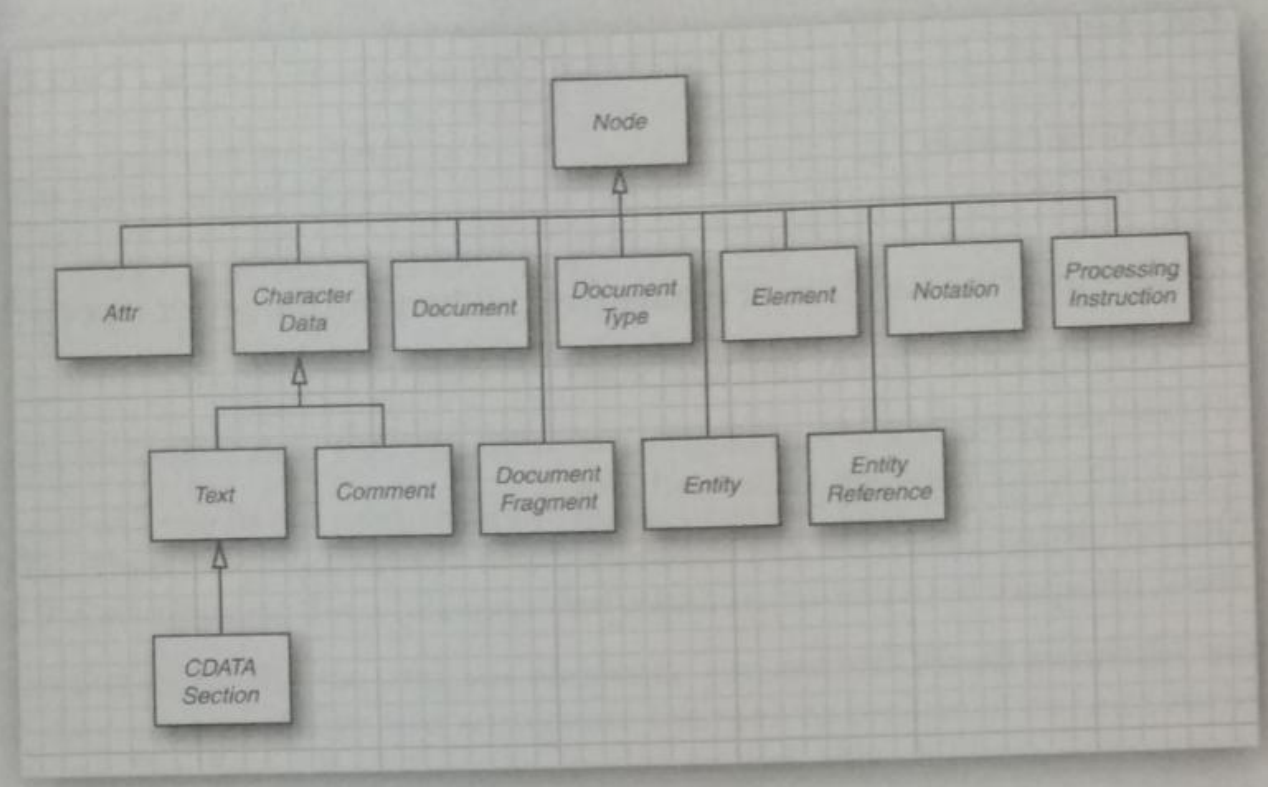

Figure 3.1 The `Node` interface and its subinterfaces

For example, if you are processing a document

```
<?xml version="1.0"?>
<font>
   . . .
</font>
```

then calling `getDocumentElement` returns the `font` element.

The `getTagName` method returns the tag name of an element. In the preceding example, `root.getTagName()` returns the string `"font"`.

To get the element's children (which may be subelements, text, comments, or other nodes), use the `getChildNodes` method. That method returns a collection of type `NodeList`. That type was standardized before the standard Java collections, so it has a different access protocol. The `item` method gets the item with a given index, and the `getLength` method gives the total count of the items. Therefore, you can enumerate all children like this:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
   Node child = children.item(i);
   . . .
}
```

Be careful when analyzing children. Suppose, for example, that you are processing the document

```
<font>
   <name>Helvetica</name>
   <size>36</size>
</font>
```

You would expect the `font` element to have two children, but the parser reports five:

- The whitespace between `<font>` and `<name>`
- The `name` element
- The whitespace between `</name>` and `<size>`
- The `size` element
- The whitespace between `</size>` and `</font>`

Figure 3.2 shows the DOM tree.

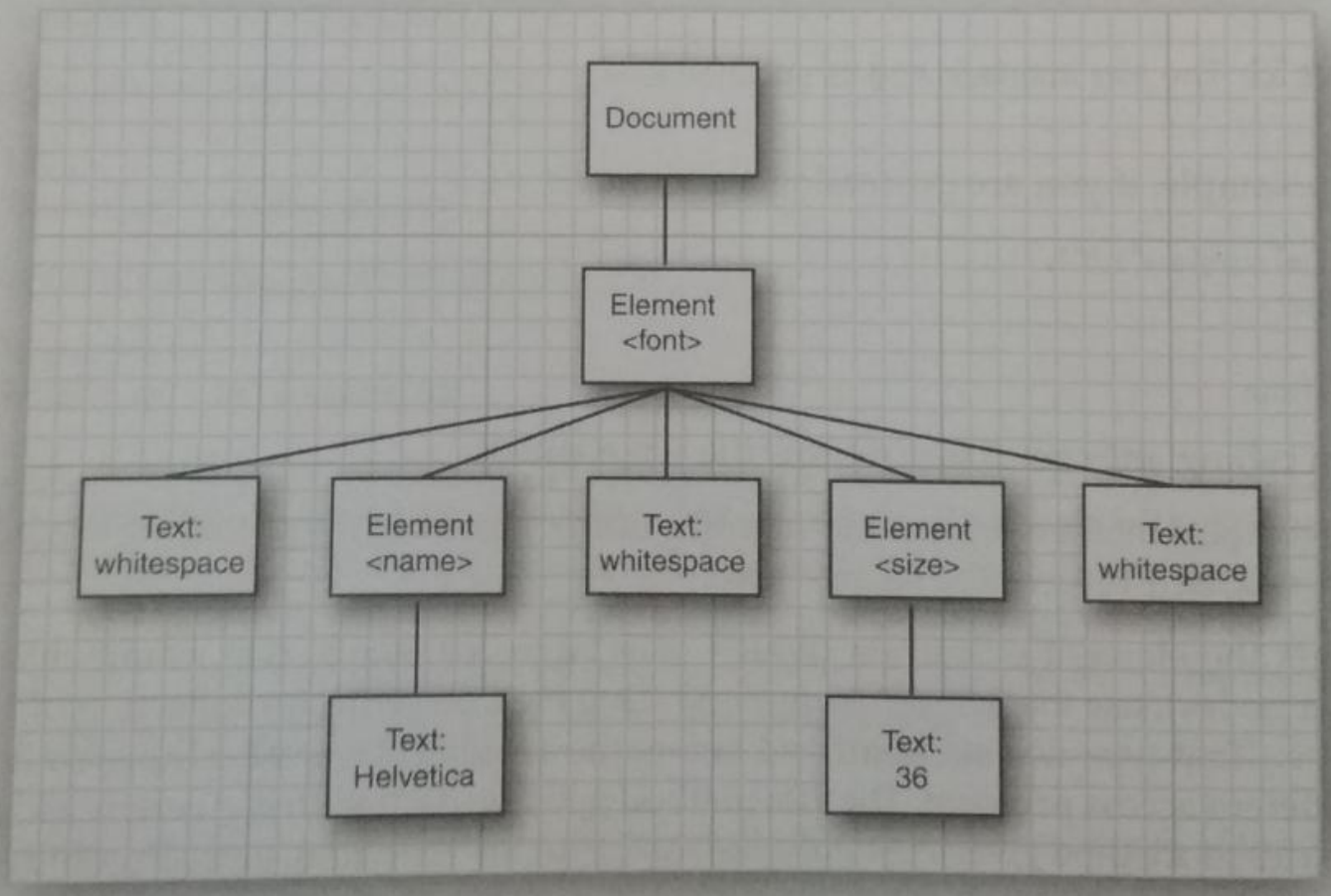

Figure 3.2 A simple DOM tree

If you expect only subelements, you can ignore the whitespace:

```
for (int i = 0; i < children.getLength(); i++)
{
   Node child = children.item(i);
   if (child instanceof Element)
   {
      Element childElement = (Element) child;
      . . .
   }
}
```

Now you look at only two elements, with tag names `name` and `size`.

As you will see in the next section, you can do even better if your document has a DTD. Then the parser knows which elements don't have text nodes as children, and it can suppress the whitespace for you.

When analyzing the `name` and `size` elements, you want to retrieve the text strings that they contain. Those text strings are themselves contained in child nodes of type `Text`. You know that these `Text` nodes are the only children, so you can use the `getFirstChild` method without having to traverse another `NodeList`. Then, use the `getData` method to retrieve the string stored in the `Text` node.

```
for (int i = 0; i < children.getLength(); i++)
{
   Node child = children.item(i);
   if (child instanceof Element)
   {
      Element childElement = (Element) child;
      Text textNode = (Text) childElement.getFirstChild();
      String text = textNode.getData().trim();
      if (childElement.getTagName().equals("name"))
         name = text;
      else if (childElement.getTagName().equals("size"))
         size = Integer.parseInt(text);
   }
}
```

TIP: It is a good idea to call `trim` on the return value of the `getData` method. If the author of an XML file puts the beginning and the ending tags on separate lines, such as

```
<size>
   36
</size>
```

then the parser will include all line breaks and spaces in the text node data. Calling the `trim` method removes the whitespace surrounding the actual data.

You can also get the last child with the `getLastChild` method, and the next sibling of a node with `getNextSibling`. Therefore, another way of traversing a node's children is

```
for (Node childNode = element.getFirstChild();
     childNode != null;
     childNode = childNode.getNextSibling())
{
   . . .
}
```

To enumerate the attributes of a node, call the `getAttributes` method. It returns a `NamedNodeMap` object that contains `Node` objects describing the attributes. You can traverse the nodes in a `NamedNodeMap` in the same way as a `NodeList`. Then, call the `getNodeName` and `getNodeValue` methods to get the attribute names and values.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
   Node attribute = attributes.item(i);
   String name = attribute.getNodeName();
   String value = attribute.getNodeValue();
   . . .
}
```

Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:

```
String unit = element.getAttribute("unit");
```

You have now seen how to analyze a DOM tree. The program in Listing 3.1 puts these techniques to work. You can use the File → Open menu option to read in an XML file. A `DocumentBuilder` object parses the XML file and produces a `Document` object. The program displays the `Document` object as a tree (see Figure 3.3).

The tree display clearly shows how child elements are surrounded by text containing whitespace and comments. For greater clarity, the program displays newline and return characters as `\n` and `\r`. (Otherwise, they would show up as hollow boxes—which is the default symbol for a character in a string that Swing cannot draw.)

In Chapter 10, you will learn the techniques this program uses to display the tree and the attribute tables. The `DOMTreeModel` class implements the `TreeModel` interface. The `getRoot` method returns the root element of the document. The `getChild` method gets the node list of children and returns the item with the requested index. The tree cell renderer displays the following:

- For elements, the element tag name and a table of all attributes

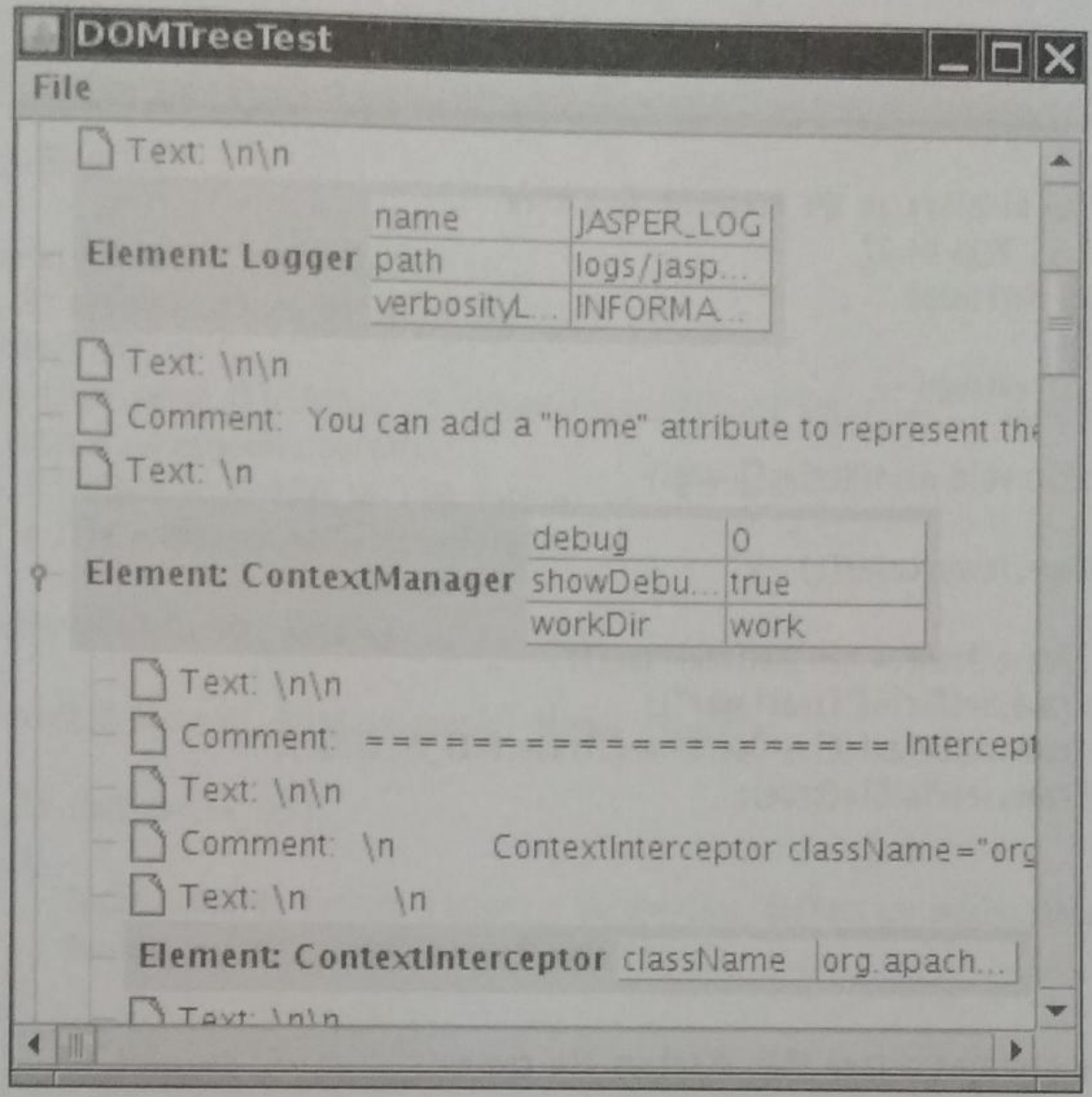

Figure 3.3 A parse tree of an XML document

- For character data, the interface (`Text`, `Comment`, or `CDATASection`), followed by the data, with newline and return characters replaced by `\n` and `\r`
- For all other node types, the class name followed by the result of `toString`