C H A P T E R 2

Markup Languages
XHTML 1.0

The previous chapter presented an overview of how computers communicate over the Internet, particularly as part of the World Wide Web. It also discussed the functions of web browsers and servers. While many types of information can be communicated between browsers and servers, most documents are written using the Hypertext Markup Language (HTML), which is a primary focus of this chapter.

Actually, HTML is not a single language, but the name for a family of related languages that have evolved over the years. We will cover one of the newer members of this family, XHTML 1.0. In order to fully understand XHTML, you'll need some familiarity with another language, the Extensible Markup Language (XML). So we will also cover enough XML to allow you to understand the formal definition of XHTML 1.0. XML is important in other contexts as well; additional XML details will be covered in later chapters, particularly Chapter 7.

We'll begin this chapter by looking at a simple HTML example. Next, you'll learn a bit about the history of HTML, why HTML standards are important, and why we will study XHTML 1.0 rather than some other version of HTML. After that, many of the basic features of XHTML 1.0 will be covered. Then XML and its relationship to XHTML will be presented. The blogging case study as well as some key online references for XHTML, XML, and browser handling of HTML are included at the end of the chapter.

When you have finished this chapter, you should be able to:

- Create standards-compliant static HTML documents using a variety of HTML elements.
- Know where to find the reference definitions of HTML and XML and be able to understand (at least most of) these definitions.
- Determine whether or not an XHTML document is syntactically correct by consulting an XML document type definition.
- Describe some of the history of HTML and the relationships between HTML, XML, and XHTML.
- Discuss pros and cons of following standards in web development.

2.1 An Introduction to HTML

Before saying any more about the Hypertext Markup Language, let's briefly look at a small example file to gain a more concrete understanding of HTML syntax and semantics. Figure 2.1 presents an HTML "Hello World!" document. Figure 2.2 shows how this document would appear if it were opened using the Mozilla browser as discussed in the previous chapter (the figure may not look much like a browser window, because toolbars and menus

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      HelloWorld.html
    </title>
  </head>
  <body>
    <p>
      Hello World!
    </p>
  </body>
</html>
```

FIGURE 2.1 "Hello World!" HTML file.

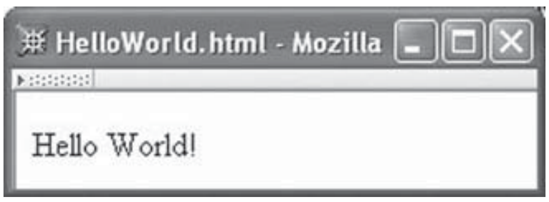
have been suppressed). You can try this example and others in this chapter by downloading the examples from the Web site for this textbook (see the Preface for the address), navigating your browser to the examples for this chapter, and selecting the file indicated by the title of the browser window (`HelloWorld.html` in this example).

This document, like every HTML document, contains two types of information:

- The markup information, which is contained in *tags* consisting of angle bracket tag delimiters (< and >) plus the text contained between these delimiters.
- The *character data* of the document, which is everything outside of the markup tags and is generally information that is intended to be displayed by the browser. In this case, the character data consists of the two strings `HelloWorld.html` and `Hello World!` as well as some white space.

The first tag appearing in the "Hello World!" document (the tag beginning with `<!DOCTYPE`) is special markup information called the *document type declaration*. We'll have more to say about this declaration later. For now, it is enough to observe that a primary function of this tag is to identify the particular version of HTML used to write the remainder of the file. In this case, the file is written in the XHTML 1.0 Strict version. How this version relates to other versions of HTML is discussed in Section 2.2.2.

Everything in the "Hello World!" document following the document type declaration—the text from `<html` down—is called the *document instance*. Each of the tags



Hello World!

FIGURE 2.2 Appearance of "Hello World!" document when opened by a web browser.

in this example document instance is either a *start tag* or an *end tag*. Syntactically, within start tags a word—the *element name*—immediately follows the `<` of the tag, while in end tags the element name is preceded by a slash (`/`). As indicated by the indentation in this example, each start tag can be viewed as starting a nesting level that is closed by its corresponding end tag, much as an open curly brace ({) in C++ or Java begins a block that is closed by a corresponding closing curly brace (}). The markup tags therefore impose a tree structure on the document (the reader unfamiliar with the notion of "trees" in computer science should refer to any introductory textbook on data structures). The start tag and its corresponding end tag, along with all of the document between the tags, is called an *element* of the document. The portion between the tags (not including the tags themselves) is called the *content* of the element.

We have seen that the document type declaration indicates the version of HTML used in the file. Another piece of information contained in the document type declaration is the name of the *root element* of the document. The first word after the `DOCTYPE` keyword is the name of the root element. For HTML documents, the root element is always named, appropriately enough, `html`. The first tag in the document instance of an HTML file must be a start tag for the root element, and the root element can only occur once in the document. In order to strictly conform with the XHTML 1.0 standard, the `html` start tag must also contain the `xmlns="..."` string shown. That string is an example of an *attribute specification*, which consists of an *attribute name* (`xmlns` in this case) and an *attribute value* (the string within quotes). We'll have much more to say about attributes later.

Viewed as a tree, the elements of our example document are shown in Figure 2.3. In all XHTML 1.0 Strict documents, the root `html` element has two children: `head` and `body`. Any text contained in the `head` element does not appear directly in the primary window area (the *client area*) of the browser window. Instead, the `head` element is used for providing certain instructions to the browser, as we will see in later chapters. The only such instruction to the browser in this example is provided by the `title` element, which directs the browser to display the element content as the window title (displayed in the title bar at the very top of the browser window; see Fig. 2.2). Also, if you bookmark this page in Mozilla, the content of the `title` element will appear in the list of bookmarks.

The `body` element contains the information that is to be displayed in the client area of the browser. This document's body contains a single paragraph (`p`) element. Notice that only the content of this element is displayed; the `p` start and end tags are used to inform the browser about the content and are not displayed themselves. A `p` element in particular tells the browser that its content represents a single paragraph of text (and possibly other elements) and should be displayed accordingly.

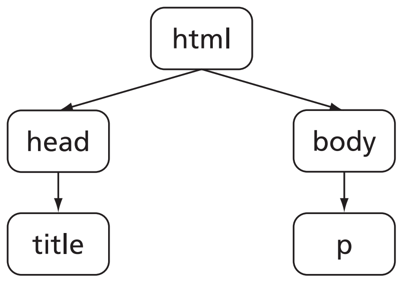


FIGURE 2.3 The element tree for "Hello World!"

Now that we've covered a few HTML basics, we will consider some of the history of HTML and its different versions.

2.2 HTML's History and Versions

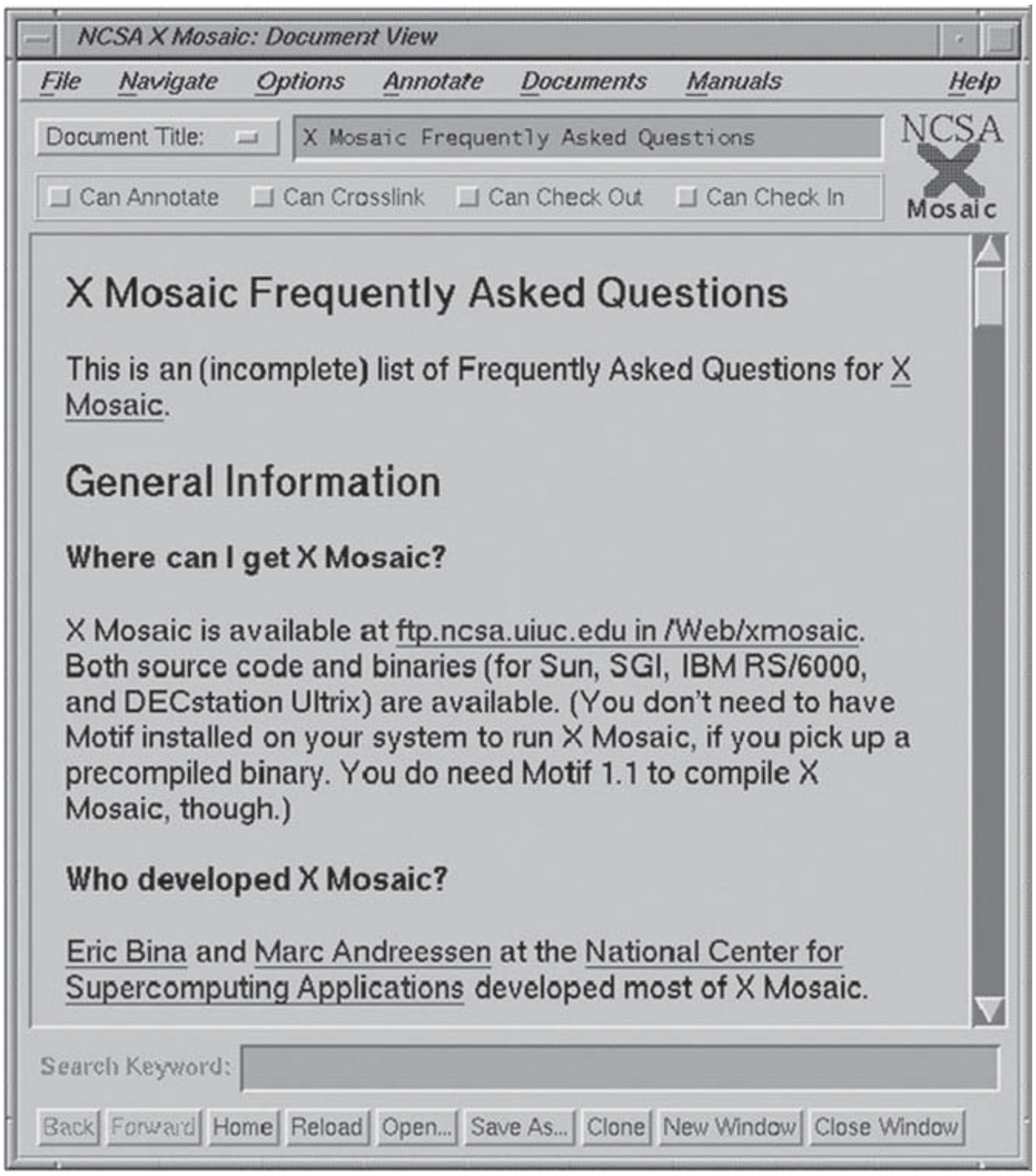
HTML was initially defined by a single person, Tim Berners-Lee, in 1990. Berners-Lee was working at a European high-energy physics research center (CERN) when he began developing HTML, and the early language was designed with science and engineering interests in mind. Even after a few years of use and revision, the elements of the language could still be described in a short document [W3C-HTML-HIST]. Specifically, the elements in use as of November 1992 included the title and paragraph elements that we have already seen, along with elements for creating hyperlinks, headings, simple lists, glossaries, examples (text with monospace fonts and any white space retained), and address blocks (containing information about the document author, and typically italicized). There was also an element that could be included in a web document to indicate that the web server providing the document would accept search terms appended to the URL. That was all! There was no facility for producing tables or fill-in forms, much less for including images within a document.

2.2.1 The "War" Years

Around this time, Marc Andreessen and Eric Bina of the National Center for Supercomputer Applications (NCSA), a unit of the University of Illinois at Urbana-Champaign, were working on a graphical web browser designed for UNIX® systems as part of a larger project called Mosaic. By February 1993 they had publicly released a preliminary version of their browser. Figure 2.4 shows the screen shot example contained in Andreessen's short technical report announcing the project and public availability of preliminary software (a revised version of this report with later screen shots is available at [NCSA-MOSAIC]). By September of the same year an initial release of this browser, along with Windows and Macintosh versions, was made available. In addition to displaying images within documents, the NCSA Mosaic browser could play video clips as well as sounds. Its user-friendly interface, multimedia support, and implementation on widely available systems jump-started the transformation of the Web from a tool used primarily by a small number of researchers in engineering and the sciences to the ubiquitous entity that we know today.

Many of the key individuals involved in the early Mosaic development at NCSA left to begin the company that became Netscape Communications. This included Andreessen, who had worked on Mosaic as an undergraduate at UIUC and was now Netscape's chief executive officer. The company soon had hundreds of employees working on various aspects of web software development. Meanwhile, after an initial delay, Microsoft deployed a similarly large development team to work on its Internet Explorer browser, initiating what became known as the "browser wars" between Netscape and Microsoft. Innovation in web technology in general—and in the HTML definition in particular—proceeded at a furious pace. HTML therefore went quickly from being a language defined by Berners-Lee and others interested in producing a "clean" language to being a language defined by browser developers working under intense market pressures.

During the period from 1993 through 1997, HTML was being defined operationally by the elements that browser software developers chose to implement and the ways in which their browsers responded to these elements. In an attempt to gain competitive advantage,


NCSA X Mosaic: Document View
File Navigate Options Annotate Documents Manuals Help
Document Title:
X Mosaic Frequently Asked Questions
Can Annotate Can Crosslink Can Check Out Can Check In
NCSA Mosaic
Search Keyword:
Back Forward Home Reload Open... Save As... Clone New Window Close Window

X Mosaic Frequently Asked Questions

This is an (incomplete) list of Frequently Asked Questions for X Mosaic.

General Information

Where can I get X Mosaic?

X Mosaic is available at ftp.ncsa.uiuc.edu in /Web/xmosaic. Both source code and binaries (for Sun, SGI, IBM RS/6000, and DECstation Ultrix) are available. (You don't need to have Motif installed on your system to run X Mosaic, if you pick up a precompiled binary. You do need Motif 1.1 to compile X Mosaic, though.)

Who developed X Mosaic?

Eric Bina and Marc Andreessen at the National Center for Supercomputing Applications developed most of X Mosaic.

FIGURE 2.4 Screen shot of early Mosaic web browser. Courtesy of the National Center for Supercomputing Applications (NCSA) and the Board of Trustees of the University of Illinois.

each of the two major browser manufacturers sought to incorporate new features (often HTML elements) into its browser so that it could tout the benefits of its browser over the competitor's. This led to significant HTML differences, not only between the latest products of each manufacturer, but also between newer and older versions of browsers from the same manufacturer. On top of this, because of the rush to get products to market and the inherent complexities of software development, browsers often had quirks or even outright bugs that had to be considered when writing the HTML for a web page. Since there were generally many end users of each of these different browsers, developing a sophisticated web page that would look right to almost all web users often required writing carefully crafted HTML.

From a page writer's perspective, this situation proved onerous. Not only did you have to write pages that took into account idiosyncrasies of current and past browsers, but you also were faced with maintaining these pages as other changes were rapidly introduced. Nearly everyone involved in web development at this time was painfully aware of the need for standardization.

In October of 1994, Tim Berners-Lee launched the World Wide Web Consortium (W3C®), in part with the goal of producing standards for HTML as well as other web technologies. During the next several years, the W3C's efforts at standards development trailed well behind the de facto standards development being carried out by the browser manufacturers. For example, HTML version 2.0 became a standard over six months after a draft for version 3.0 had been published, and 3.0 was never formally adopted as a standard because of the rapid browser changes. Version 3.2 was adopted as a standard by the W3C in January of 1997, and by its own admission in the 3.2 specification document [W3C-HTML-3.2] aimed "to capture recommended practice as of early '96," so was still at least a year behind the browser manufacturers. Finally, the "browser wars" slowed and the standards community caught up. The W3C released its HTML 4 recommendation in December of 1997. The current version of this recommendation, HTML 4.01, is the standard that is more or less followed by many if not most HTML documents on the Web at the time of this writing.

2.2.2 The Clean-up Effort

Following the "war" years, the push for further change in HTML standards seems to have come from the standards community more than from browser developers. In particular, the W3C has been engaged in several efforts to clean up the definition of HTML in various ways. One of these directions has involved changing the way in which HTML is defined. Defining a language such as HTML (or any computer language, for that matter) involves two aspects: its syntax and its semantics. The *syntax* of a computer language defines which strings of characters represent a document that conforms to the language and which do not. For a programming language such as Java, a program that compiles is a syntactically correct document. The *semantics* of a language is a description of what the various elements of a syntactically correct document mean. For example, a syntactically correct assignment statement in Java has a certain meaning: a variable is associated with a value that can later be referenced by the variable's name. Similarly, the `p` element in HTML 4.01 also has a certain meaning: its content is to be displayed as a paragraph in the browser that is reading the document containing the `p` element.

Although precise formal methods for semantic definition have been developed and are sometimes used, the semantics of many languages is defined using natural-language descriptions such as the examples just given. In particular, the semantics of the elements and attributes in HTML 4.01 are defined using natural language [W3C-HTML-4.01]. On the other hand, the syntax for a computer language is almost always defined using some other language specially designed for the purpose of defining language syntax. A language used to describe the syntax of other languages is sometimes referred to as a *metalanguage*. The metalanguage commonly used to describe the syntax of programming languages such as Java is called Backus-Naur Form (BNF) notation. In fact, BNF notation could also be used to define the syntax for HTML. However, HTML and other similar markup languages are simpler than typical programming languages, and therefore specialized metalanguages can be used to describe them.

The metalanguage used to define the syntax for HTML 4.01 is SGML, the Standard Generalized Markup Language. As the "Generalized" part of its name implies, even this metalanguage is fairly general. This generality can complicate the *parsing* of an HTML document. Loosely speaking, parsing an HTML document involves inputting the document and creating an internal element tree (an *abstract syntax tree* or *parse tree*) representing the document, such as the tree in Figure 2.3. One example of a way in which SGML's generality increases the difficulty of parsing is its feature allowing certain tags to be omitted. For example, in HTML 4.01, the end tag of a `p` element can be omitted from a document. In fact, both start and end tags can be omitted for some elements, including the `head` and `body` elements. An HTML parser must therefore be able to correctly parse a document whether or not it contains tags that can be omitted. It is obviously more difficult to write a parser that allows for omitted tags than to write one that requires that all start and end tags be present.

In February 1998, the W3C introduced the Extensible Markup Language (XML), a restricted version of SGML. XML limits some of the generality of SGML while retaining enough power to define syntaxes for languages such as HTML. In fact, the syntaxes for several HTML versions have been defined using XML. A hypertext markup language whose syntax is defined using XML rather than SGML is called an XHTML language.

The first of the XHTML languages, XHTML 1.0, is semantically identical to HTML 4.01. Syntactically, XHTML 1.0 is also the same as HTML 4.01 except that XHTML restricts some of HTML's generality in a few small ways. In order to more precisely understand the nature of these restrictions, it will be helpful to define a few more terms. The *abstract syntax* of a language defines a language at the level of abstract syntax trees. For HTML and XHTML languages, this primarily involves defining what elements can be contained in the tree; what attributes can be associated with each element and what values these attributes can take on; and what children an element can have and the order in which the children must appear. The *concrete syntax* of a language defines how this tree structure is represented within the language. In the case of HTML and XHTML, this involves a variety of low-level details, such as what characters are used to delimit tags, how these characters can be escaped so that they do not have a tag-delimiting meaning, whether or not element names are case sensitive, how attribute values should be quoted, if at all, and so on.

Now that we have made this distinction between abstract and concrete syntax, we can be more precise about the difference between XHTML 1.0 and HTML 4.01: these languages are equivalent at the semantic and abstract syntactic levels. They differ *only* in terms of concrete syntax. The primary concrete syntactic restrictions on XHTML include the following:

- Omitted tags are not allowed.
- All element and attribute names must be lowercased (HTML 4.01 names are case insensitive).
- All attribute values must be quoted (not always necessary in HTML 4.01 documents).

As you can see, these restrictions are not too burdensome, and may actually be helpful to human as well as machine readers of an HTML document.

A primary advantage of following the XHTML 1.0 restrictions is that an XHTML 1.0 document is a particular form of XML document, and a wide variety of tools have been developed for processing XML documents. As a simple example, one XML tool can easily extract the content of the `title` element of an XHTML document; such a tool might be helpful in a larger application that produces a table of contents for a directory containing XHTML files. A number of XML tools and technologies are covered in Chapter 7. While there are also some SGML tools that can be used to process SGML-based documents such as those written in HTML 4.01, the SGML tools are few in comparison with the wide array of XML tools. In addition, since XML is a restricted version of SGML, these SGML tools can be applied to XHTML documents as well, if desired. Figure 2.5 summarizes the relationships between SGML, XML, HTML 4.01, and XHTML 1.0.

XHTML 1.0 and HTML 4.01 are both currently "recommendations" of the W3C, which means that "consensus has been reached among the Consortium Members that [each] specification is appropriate for widespread use." Another current recommendation is XHTML Basic 1.0, which is a subset of XHTML 1.0 designed for use with limited devices such as cell phones. Yet another recommendation is XHTML 1.1, which is identical to XHTML 1.0 in both semantics and syntax. The only difference between the XHTML 1.0 and 1.1 languages is grammatical. A *grammar* is the collection of rules (XML-based rules in the case of these languages) defining the syntax of a language. The difference between the XHTML 1.0 and 1.1 languages is that XHTML 1.1 is defined using a grammar that is more modular (and somewhat more complicated) than the grammar used to define XHTML 1.0.

Given this history as well as current development trends, I have chosen to follow the XHTML 1.0 standard in this textbook. With few exceptions, all modern browsers properly implement the elements of XHTML 1.0, so writing your documents according to this standard means that they should be highly portable. Also, if you understand the material covered in this chapter, you should not have much difficulty in switching to a different standard at a later time if necessary.

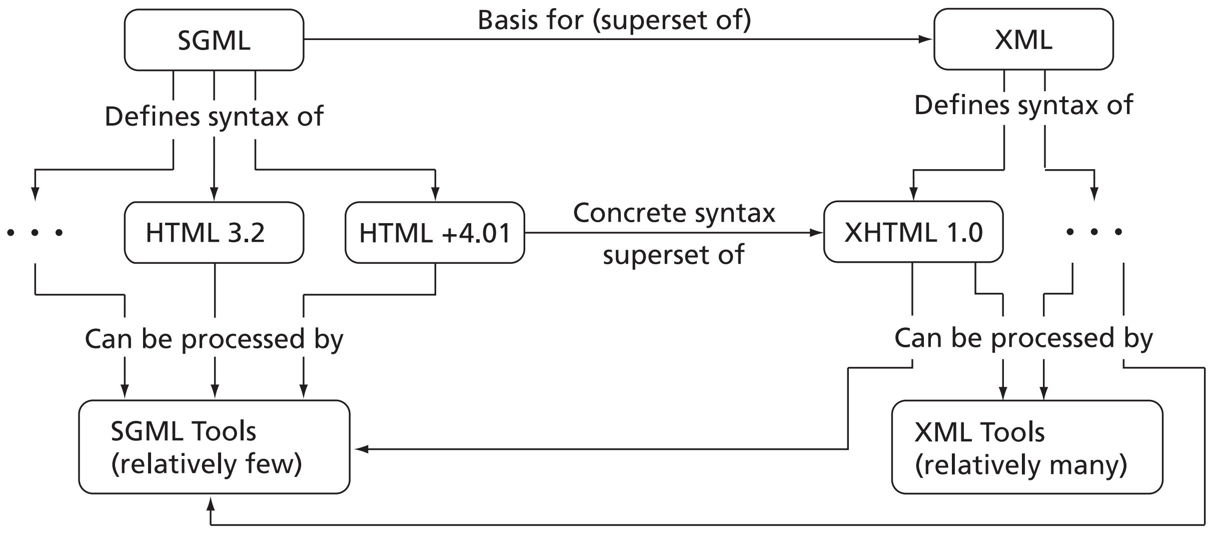


FIGURE 2.5 Relationships between SGML, XML, HTML, and XHTML.

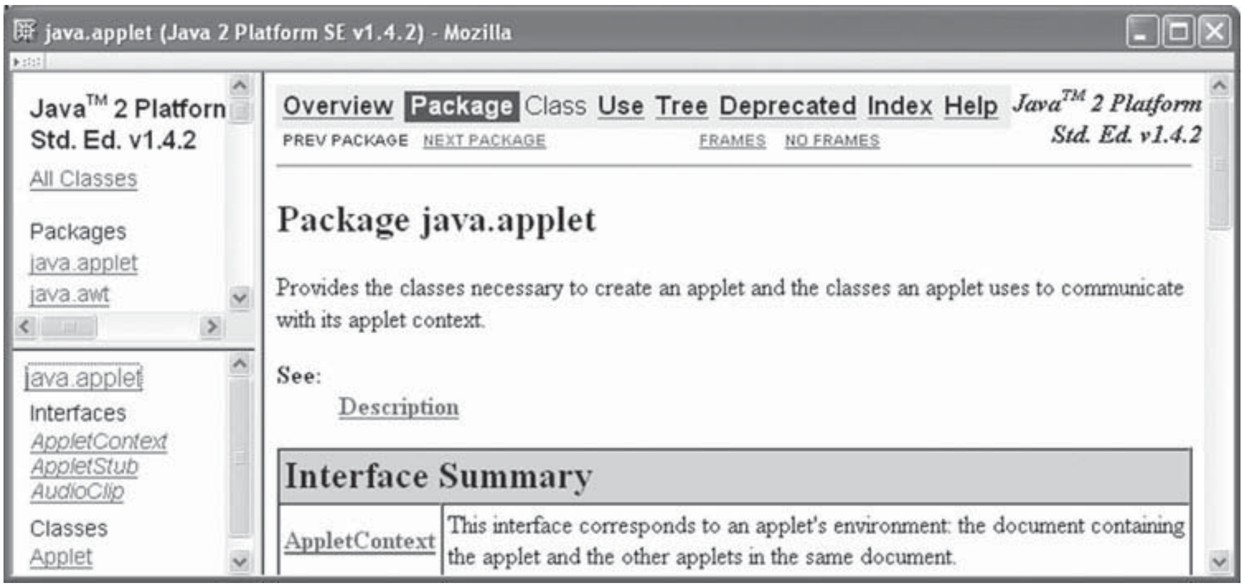
The next section will focus on some of the concrete syntactic and semantic basics of XHTML 1.0. Following that, several sections will cover the semantics of a variety of elements; the same semantics apply to all of the other current HTML specifications as well. Then you'll learn how to read XHTML 1.0's XML grammar, which defines the abstract syntax of XHTML 1.0. The chapter will close with a brief discussion of tools for writing HTML documents and the case study. (Here and throughout the rest of this text, when I use the term "HTML" without qualification I will have in mind both XHTML and HTML documents.)

2.3 Basic XHTML Syntax and Semantics

2.3.1 Document Type Declaration

We have already seen that every XHTML document must begin with a document type declaration. Each HTML specification provides such a declaration that can be used at the beginning of documents intended to conform with the specification. However, there are three *flavors* of both the HTML 4.01 and XHTML 1.0 specifications, each with its own document type declaration. Each flavor includes a somewhat different set of elements and attributes. The three flavors are:

1. Strict: The W3C's ideal for HTML as of late 1997.
2. Transitional: A superset of Strict HTML that includes *deprecated* elements and attributes, that is, elements and attributes that should not be used if possible because they will likely be eliminated from HTML recommendations at some future time.
3. Frameset: A superset of the Transitional flavor that includes a feature allowing several subwindows (*frames*) to be displayed within a browser's client area. You've probably seen frames if, for example, you've viewed the Sun Java API specification (Fig. 2.6).


java.applet (Java 2 Platform SE v1.4.2) - Mozilla

Java™ 2 Platform Std. Ed. v1.4.2

All Classes

Packages
java.applet
java.awt

java.applet
Interfaces
AppletContext
AppletStub
AudioClip
Classes
Applet

Overview Package Class Use Tree Deprecated Index Help *Java™ 2 Platform Std. Ed. v1.4.2*

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package java.applet

Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.

See:
Description

Interface Summary	
AppletContext	This interface corresponds to an applet's environment: the document containing the applet and the other applets in the same document.

FIGURE 2.6 Example of a web page with three frames. (Browser content copyright ©2006 Sun Microsystems, Inc. All rights reserved. Reproduced by permission of Sun Microsystems Inc.)

Many documents on the Web today begin with a document type declaration for the HTML 4.01 Transitional flavor. However, almost all usage of deprecated elements and attributes included in the Transitional flavor can be replaced by using *style sheet* technology, which is supported by almost all browsers in use today (style sheets are covered in the next chapter). So there is little if any reason to use the Transitional HTML flavor any longer, and I will avoid it in this text. Instead, we will focus primarily on the Strict XHTML 1.0 flavor. The Frameset flavor is covered briefly in a later section, but—as we will see—there are also some good reasons to avoid its use except for certain specialized applications. The recommended XHTML 1.0 Strict, XHTML 1.0 Frameset, and HTML 4.01 Transitional document type declarations are:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!DOCTYPE HTML
PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

The last of these is included for informational purposes, so that you can recognize it if it is included in a document.

2.3.2 White Space in Character Data

Recall that the character data of an HTML document is the information that lies outside the markup of the document, and to a large extent is the textual content of the web page produced by the document. With a few exceptions that are covered later, any XHTML white space characters (Table 2.1) within character data are treated by the browser as word separators, and the specific white space character(s) used to separate words, as well as the number of characters, is considered irrelevant. In a language such as English, the net effect of this treatment of white space is that the browser replaces any string of white space characters within character data by a single blank.

TABLE 2.1 XHTML (and XML) White Space Characters

Character	ASCII Code (Decimal)	Unicode Standard Value (Hex)
Carriage return	13	000D
Line feed	10	000A
Space	32	0020
Tab	9	0009

As an example of browser handling of white space in element content, consider the following HTML document, which changes the content of the `p` element of the original "Hello World!" example:

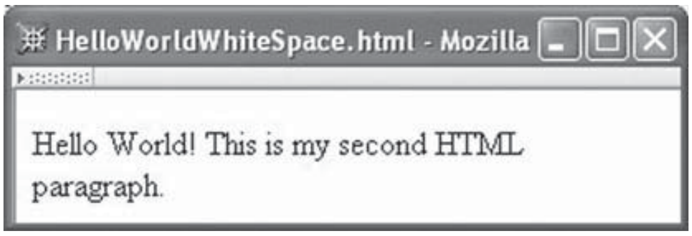
```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      HelloWorldWhiteSpace.html
    </title>
  </head>
  <body>
    <p>
      Hello World!

      This is my second HTML paragraph.
    </p>
  </body>
</html>
```

Figure 2.7 shows a browser window loaded with this HTML file. Notice that although the text within the `p` element is typed into the HTML document as two paragraphs (there is a blank line between two pieces of text), the browser displays all of the text as a single paragraph with a single space between the two sentences, and in fact even performs rewrapping of the paragraph (moves the last word to a second line in this example) so that the paragraph fits within the browser window.

A simple way to tell the browser that we want the text in this example to be displayed as two paragraphs is to use two `p` elements instead of one:

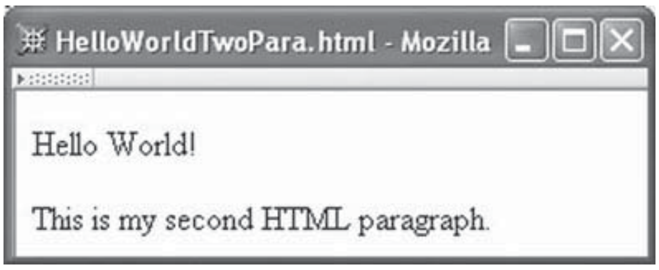
```
<p>
  Hello World!
</p>
<p>
  This is my second HTML paragraph.
</p>
```

An example of a browser loaded with such a document is shown in Figure 2.8.



Hello World! This is my second HTML paragraph.

FIGURE 2.7 Browser collapses white space in modified "Hello World!"



Hello World!

This is my second HTML paragraph.

FIGURE 2.8 "Hello World!" with two `p` elements.

2.3.3 Unrecognized Elements and Attributes

A second feature of HTML that sometimes confuses beginning web authors is that browsers don't complain if a document contains element or attribute names that the browser does not recognize. This is different from what we're accustomed to when writing programs: if we mistype a keyword such as `while` in a Java program, the compiler will issue an error and the program will not run. But if we mistype an element name such as `p`, the browser will still attempt to display the entire web page. For attributes with unrecognized names, the browser acts as if the attribute is not present at all. For unrecognized element names, the browser displays the content of the element as if the markup were not present.

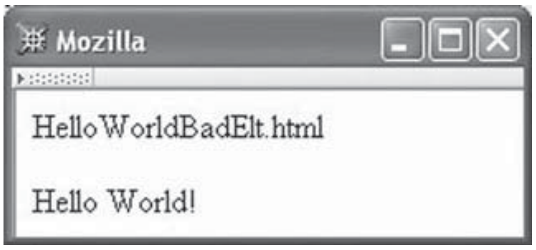
For example, let's say that we leave off the "e" in "title" in the the `title` start tag, as in the following example:

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <titl>
      HelloWorldBadElt.html
    </title>
  </head>
  <body>
    <p>
      Hello World!
    </p>
  </body>
</html>
```

Mozilla displays this page as shown in Figure 2.9.

In this example, the browser treats the content of the `titl` element as if it were text typed directly within the `head` element. Text is not supposed to appear here, and the HTML standard does not specify how a browser should display such information. Mozilla chooses to display the text as if it were the initial content of the `body`, as shown in the figure. Notice that the title bar of the window does not display this text.

This handling of unrecognized names is important because it allows HTML to continue to evolve. For instance, if an XHTML 1.2 standard is someday released that contains a `sproing` element that causes character data within the content of the element to bounce up



HelloWorldBadElt.html

Hello World!

FIGURE 2.9 Browser displaying HTML file with misspelled element name `titl`.

and down a bit when displayed, page authors can immediately begin including the `sproing` element in their documents. Browsers that don't recognize the `sproing` element will still display character data contained in this element; they just won't jiggle this data. (No, I am *not* suggesting that I want a `sproing` element in my next browser!)

One implication of HTML's handling of unrecognized tag names is that an HTML page may display correctly in a browser but still have typographical errors in its markup. For example, consider the following document body:

```
<p>
  Hello World!
</p>
<l>
  This is my second HTML paragraph.
</p>
```

The second paragraph mistakenly begins with an `l` tag. Since `l` is not a valid element name in HTML, this tag will be ignored. Yet Mozilla will still display this document as shown in Figure 2.8. This is because, for display purposes, Mozilla treats text that is contained directly in the `body` element as if it were the content of a `p` element. Although it displays properly, such a document could lead to other problems. For example, if this document were later processed by some other software—say a program that converts XHTML documents to plain text—it would likely produce an error.

To avoid such problems, it is a good idea to check the validity of the HTML in a document using means other than simply loading the document into a browser. An XHTML document is *valid* if it conforms with the XML grammar defining the syntax of the language. One simple way to perform validation checking is to use an *HTML validator*, such as the one available at the W3C [W3C-VAL]. This is a program that will analyze your document and not only catch typographical errors, but also help you to ensure that the HTML you generate conforms to the standards of the HTML version you are using.

2.3.4 Special Characters

Another troublesome aspect of HTML is that a few characters must be used carefully in HTML documents. For example, the less-than symbol (<) is the special symbol used to begin tags. You might reasonably assume that the less-than symbol would only be treated specially if it was followed by an element name such as `head` or `p`, but, as we have just seen in the previous subsection, this is not the case. Instead, a browser will almost always view a less-than as the beginning of a tag, regardless of what follows.

So how do we produce a document that displays a less-than symbol? Instead of typing the symbol itself into the document, we use a type of markup known as a *reference*. For example, `&lt;` is a reference that represents the less-than symbol. A reference within an HTML document always begins with an ampersand (`&`) and ends with a semicolon (`;`). The form of the data between these characters determines the type of reference. A reference such as `&lt;`, which uses a mnemonic name for the character referenced, is called an *entity reference*. Not all characters have entity references associated with them, but many do; some examples of entity references are contained in Table 2.2. This table also contains a second type of reference for each symbol, a *character reference*. In this case, a number sign (`#`) follows the ampersand beginning the reference and is followed by the Unicode Standard value of the character (prefixed with a lowercase `x` if hexadecimal is used, or with no prefix for decimal). A complete list of entity references defined for XHTML 1.0 can be found in Section A.2 of [W3C-XHTML-DTDS].

The bottom line is that you must use `&lt;` (or `&#60;` or `&#x3C;` or `&#x3c;`) to include a less-than sign as part of the content of a document. Similarly, you must use a reference for the ampersand (&), because it is the special character used to begin a reference. Also, there is one particular three-character string—`]]>`—that cannot be used as the content of an element; just to be on the safe side, it is therefore probably best to be in the habit of using a reference for the greater-than symbol as well. Finally, references can also be used to produce symbols that do not appear on your keyboard, such as the copyright symbol ©.

One other HTML entity reference that is frequently used is `&nbsp;`, the *nonbreaking space* character. The defined purpose of this character is to insert a space between two strings while also informing the browser that it should not perform word-wrapping between these strings. For example, a browser displaying the HTML source

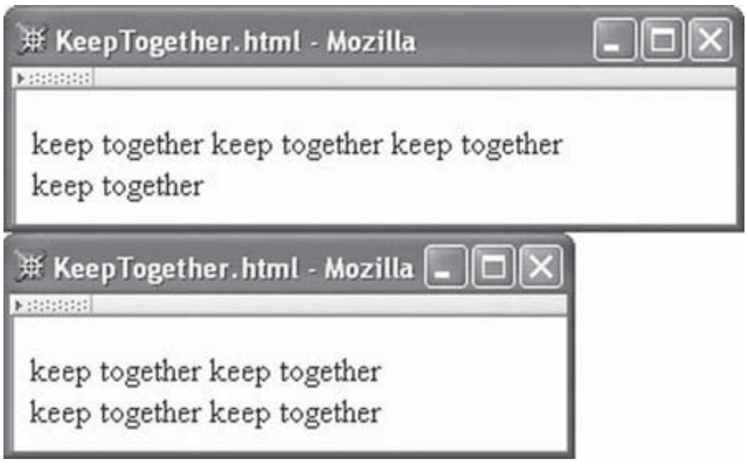
```
<p>
keep&nbsp;together keep&nbsp;together keep&nbsp;together keep&nbsp;together
</p>
```

will never end a line with the word "keep," as illustrated by Figure 2.10.

Although the nonbreaking property of `&nbsp;` is at times useful, the main reason that it is frequently used is that it is displayed as a space character but is not one of the four

TABLE 2.2 Example Entity and Character References

Character	Entity Reference	Character Reference (Decimal)
<	`&lt;`	`&#60;`
>	`&gt;`	`&#62;`
&	`&amp;`	`&#38;`
"	`&quot;`	`&#34;`
'	`&apos;`	`&#39;`
©	`&copy;`	`&#169;`
ñ	`&ntilde;`	`&#241;`
α	`&alpha;`	`&#945;`
$\forall$	`&forall;`	`&#8704;`



keep together keep together keep together
keep together

keep together keep together
keep together keep together

FIGURE 2.10 Two browser windows of different widths displaying an HTML file using the `&nbsp;` reference.

XHTML white space characters (Table 2.1). This means that we can force a browser to display multiple consecutive spaces, even though HTML specifies that consecutive white space characters must be collapsed to a single character. So, for example, if we want two spaces to follow a sentence-ending period, we can used HTML such as the following:

```
<p>
  Hey, you.&nbsp; Yes.&nbsp; I am talking to you.
</p>
```

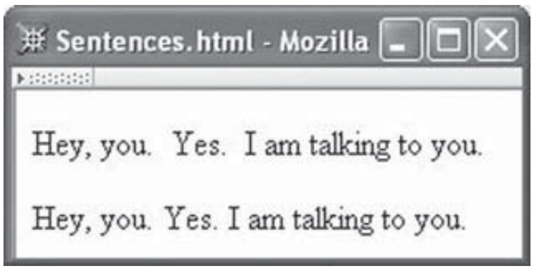
which produces better-looking output than does the following:

```
<p>
  Hey, you.  Yes.  I am talking to you.
</p>
```

as shown in Figure 2.11.

2.3.5 Attributes

In our earlier "Hello World!" example Figure 2.1, we learned that the `html` element of any XHTML 1.0 document must contain an `xmlns` attribute specification. It turns out that every



Hey, you. Yes. I am talking to you.

Hey, you. Yes. I am talking to you.

FIGURE 2.11 Sentences with (top) and without (bottom) the use of `&nbsp;`.

HTML element has a set of associated attributes that can be specified for it. The values of an element's attributes typically influence how the element is displayed or how it behaves, or they may supply identifying information. For example, the `xmlns` attribute identifies the XML *namespace* for the document, which can be considered identifying information. We'll learn more about namespaces in Chapter 7, and we'll learn about several other common HTML attributes later in this chapter. For now, we'll just cover some syntactic aspects of attributes.

All XHTML attribute specifications have the form shown for `xmlns`: white space (Table 2.1) separates the attribute name from the element name in the start tag of the element; the attribute name is followed by an equals sign (=), optionally preceded and followed by white space; and the value of the attribute, enclosed in quotes, follows the equals sign. Either a pair of single quotes or a pair of double quotes may be used to quote the attribute value. The attribute value string may not contain the character used to quote the string, but it may contain the other quote character. So, for example, an attribute specification such as

```
value = "Ain't this grand!"
```

is legal, but

```
value = "He said, "She said", then sighed."
```

is not. However, references may appear within an attribute value, so

```
value = "He said, &quot;She said&quot;, then sighed."
```

is valid. The `&quot;` references will be converted to double quotes when the document is parsed. Also note that, as in the case of element content, the less-than (<) and ampersand (&) symbols cannot be used to represent themselves within an attribute value but instead must be included using a reference. To be safe, you should probably use a reference for the greater-than symbol (>) as well.

Multiple attribute specifications can be included within a single tag by separating the specifications with white space. For example, it can be useful to certain applications, such as search engines and accessibility software, to identify the human language in which the character data of the document is written. A standard way to do this is to include `lang` and `xml:lang` attribute specifications in the `html` start tag. Both attributes are used so that the document will be compatible with software that understands HTML 4.01, which does not contain the `xml:lang` attribute (but which will ignore it due to the unrecognized-name feature described earlier), as well as with software that understands XML, which defines the `xml:lang` attribute for use across arbitrary XML-based languages, including XHTML. An `html` start tag containing attribute specifications for both of these attributes as well as `xmlns` is

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
```

This assigns the value `en` (English) to both of the language attributes. Multiple attribute specifications can appear in any order, so

```
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml" lang="en">
```

is equivalent to the previous start tag.

Finally, it is good practice to observe certain restrictions on attribute values to ensure compatibility across different browsers. First, newline characters should not appear within an attribute value; in other words, an attribute value should appear on a single line. In fact, of the four white space characters, it is best to use only the space character within an attribute value. Furthermore, avoid including any leading or trailing white space, and also avoid having multiple adjacent white space characters within attribute values. If you follow these conventions, your attribute values will be *normalized*. Some browsers may normalize all attribute values whereas others may not, so normalizing the values yourself should ensure consistency across browsers.

Now that we've learned some of the foundational aspects of XHTML's semantics (the "meaning" assigned to white space and unrecognized elements and attributes) and concrete syntax, we're ready to move on to learning about a number of fundamental HTML elements and their semantics.

2.4 Some Fundamental HTML Elements

This section introduces a number of structurally simple HTML elements. While simple, these elements include some of the most fundamental, such as elements for creating hyperlinks and displaying images. We will use a single example to illustrate the elements described in this section. The `body` element of the HTML for this example is shown in Figure 2.12, and a browser displaying a rendering of this HTML is shown in Figure 2.13. Each of the new elements introduced in this example is described briefly in Sections 2.4.1–2.4.6.

2.4.1 Headings: `h1` and Friends

`h1` and `h2` are examples of HTML *heading* elements. As shown in the example of Figure 2.13 and Figure 2.12, HTML markup such as

```
<h1>
  Some Common HTML Elements
</h1>
<h2>
  Simple formatting elements
</h2>
```

can be used to produce section headings for an HTML document. `h1` represents a top-level heading, `h2` a subheading, and so on. In all, six different levels (`h1` through `h6`) are provided in HTML. The content of each heading element is shown on a separate line. Browsers will typically display each heading in a different type face, with `h1` the largest and in bold while

```
<body>
  <h1>
    Some Common HTML Elements
  </h1>
  <h2>
    Simple formatting elements
  </h2>
  <pre>
Use pre (for "preformatted") to
  preserve white space and use
    monospace type.
    (But note that tags such as<br />still work!)
  </pre>
  <p>
    A horizontal <span style="font-style:italic">separating line</span>
    is produced using
    <tt><strong>hr</strong></tt>:
  </p>
  <hr />
  <h2>
    Other elements
  </h2>
  <!-- Notice that img must nest within a "block" element,
       such as p -->
  <p>
    <img
        src="http://www.w3.org/Icons/valid-xhtml10"
        alt="Valid XHTML 1.0!" height="31" width="88"
        style="float:right" />
    See
    <a href="http://www.w3.org/TR/html4/index/elements.html">the
      W3C HTML 4.01 Element Index</a>
    for a complete list of elements.
  </p>
</body>
```

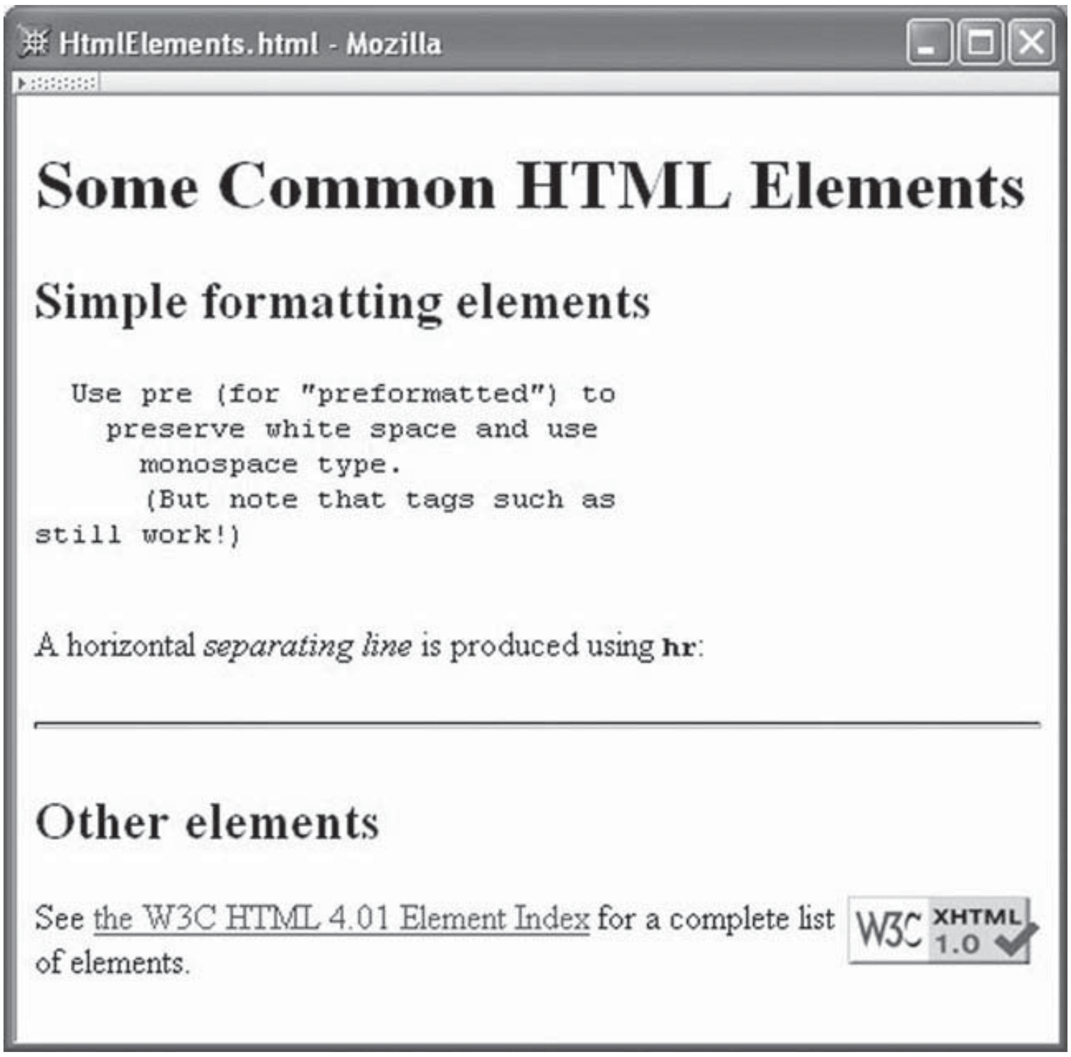
FIGURE 2.12 Body of an HTML document containing some common elements.

`h6` may look much like normal text. This default formatting can be overridden as described later in the chapter on style sheets.

Since the heading elements carry some semantic meaning (concerning section levels) as well as default formatting, it is generally considered poor practice to skip heading levels. For instance, an `h1` element should be followed by an `h1` or `h2` element, not by a higher-numbered heading element.

2.4.2 Spacing: `pre` and `br`

The `pre` element is used to override a browser's normal white space processing. So, in the example, the HTML markup


HtmlElements.html - Mozilla

Some Common HTML Elements

Simple formatting elements

```
  Use pre (for "preformatted") to
    preserve white space and use
      monospace type.
      (But note that tags such as
still work!)
```

A horizontal *separating line* is produced using **hr**:

Other elements

See the W3C HTML 4.01 Element Index for a complete list of elements.

FIGURE 2.13 Browser rendering of some common HTML elements.

```
  <pre>
Use pre (for "preformatted") to
  preserve white space and use
    monospace type.
    (But note that tags such as<br />still work!)
  </pre>
```

produces output that looks almost identical to the HTML source. In fact, most browsers will not perform word wrapping on this text even if a line is too long to fit within the width of the browser window. Instead, the browser will provide a horizontal scroll bar that the user can manipulate to see all of the text. Also, most browsers will display the content of the `pre` element using a monospace font. This is particularly useful for displaying a Java program listing, for example.

However, a potential difficulty with using `pre` is that the content of a `pre` element is still considered to be HTML by the browser. This means, for example, that if a less-than symbol (<) appears in the content, it will be viewed as the beginning of a tag. This is why the text `still work!)` appears on a line by itself: the browser encounters the string

`<br />` and interprets it as markup, not as text. In fact, the `br` element in HTML represents a line break. It causes the browser to start a new line, much as a `\n` character causes a new line of output to begin when written by a C++ or Java program.

The `br` element is an example of an *empty element*. An empty element is one that is not allowed to contain content. That is, it is syntactically illegal to write HTML markup such as

```
<br>
  Content of the br element.
</br>
```

The `img` element (discussed in Section 2.4.5) is another example of an empty element. We will learn later in this chapter how to know for sure whether or not an element is defined to be empty by a given version of HTML. For now, it is important to know that such elements should be written as shown by these examples: follow the element name and any attribute name–value pairs by white space and the string `/>`. A tag ending with this string is known as an *empty-element tag*. Technically, there are other ways to write an empty XHTML element, such as without the white space preceding the `/` or as a start–end pair of tags. However, the syntax shown here should be more compatible with most current browsers and is therefore the form we will always use for XHTML.

2.4.3 Formatting Text Phrases: `span`, `strong`, `tt`, etc.

HTML provides a number of different means for performing the sorts of text-oriented tasks that we identify with word processing, such as boldfacing or changing the font or even the color of a word or phrase. One way to specify the style of words and phrases is by making the text the content of a `span` element and setting the value of the `style` attribute appropriately. For example,

```
<span style="font-style:italic">separating line</span>
```

will display `separating line` in italics, assuming an italic font is available on the display device. We'll learn much more about the `style` attribute in the next chapter.

The `span` element itself has no effect on the text. It is merely a *wrapper* that allows style and other attributes to be applied to portions of a document (see Section 2.4.8 for more on what can be contained within a `span` element).

The technique of wrapping text in a `span` with appropriate values for the `style` attribute can be used to perform a wide variety of text operations. However, there are shorter and simpler alternatives for some of the most common text operations. For example, text can be made boldface by making it the content of a `strong` element:

```
<strong>hr</strong>
```

Technically, this only marks the text "hr" as being something that has a certain semantic meaning, specifically, that it is text that should be "made strong." How this is actually displayed is not specified by the HTML standard, but in practice it is displayed in bold by

TABLE 2.3 HTML Font Style Elements

Element	Font Used for Content
`b`	Boldface
`i`	Italic
`tt`	Monospace ("teletype": fixed-width font)
`big`	Increased font size
`small`	Decreased font size

modern browsers. Another element, `em`, marks its content as something that should be given "emphasis," which in practice means that the content is displayed in italics in most browsers. However, such semantic elements also have meaning to other user agents. For example, a user agent based on a speech synthesizer might represent the `strong` element by increasing volume.

Yet another way to mark up text phrases is by using one of the *font style elements*. The (undeprecated) font style elements available in HTML 4.01 are shown in Table 2.3. These differ from the *phrase elements* such as `strong` and `em` in that they specify the actual typography to be used rather than associating semantics with text.

All of these font effects can be achieved using `span` with appropriate values of the `style` attribute, and in fact, even though these font style elements are part of the Strict standard, the W3C recommends that a style sheet approach (of which the `style` attribute is one example) be used rather than these elements. The phrase elements `strong` and `em` are similarly generally preferable to their font style counterparts `b` and `i` because they provide semantic information. The font style elements are discussed here mainly so that you will be familiar with them if you see them used—as they often are—on other web pages.

Finally, you may have noticed that we haven't mentioned underlining, another common word-processing feature. The reason is that most web users associate underlined text with hyperlinks. Therefore, it's generally a good idea to avoid using underlining for other purposes. However, if you must underline text, there is a style—`text-decoration:underline`—that can be used. Transitional HTML also includes a `u` element for this purpose.

2.4.4 Horizontal Rule: `hr`

The `hr` element adds a horizontal line to the document. This line appears below the preceding HTML content and above the content following the `hr` element.

Like the `br` element, `hr` is an empty element. The `hr` element defines several attributes that can be used to modify its style, but these have been deprecated in favor of the use of the `style` attribute, which again is covered in more detail in the next chapter.

2.4.5 Images: The `img` Element

The "image" element `img` is the primary means of including a graphic in a document, and is illustrated in our example by

```
<img
    src="http://www.w3.org/Icons/valid-xhtml10"
    alt="Valid XHTML 1.0!" height="31" width="88"
    style="float:right" />
```

The `src` attribute of this element specifies the URL of an image to be requested via the HTTP GET method. That is, in order for the browser to produce the display shown in Figure 2.13, it must perform two GET requests: first, the GET to request the HTML document `HtmlElements.html`; then, after the browser has recognized the `img` element, the GET to request the graphic displayed in the lower right corner of the browser window. In this example, the image is being loaded from a server with fully qualified domain name `www.w3.org`. We'll learn in Section 2.5 a somewhat simpler way to load images when an HTML document and its associated images come from the same server.

The `alt` attribute on the `img` element specifies text that will be displayed by a browser that is unable to display images or that can be used to provide information about the image to visually impaired users. This text should therefore be descriptive of the image. Both the `src` and `alt` attributes are required. (Providing descriptive `alt` attributes is just one of many ways in which you can help make your web pages more accessible to people with disabilities; see [W3C-WAI] for a full set of accessibility guidelines.)

The optional `height` and `width` attributes can be used to tell the browser to scale an image to a size other than the one in which it was recorded. This can be useful for displaying a thumbnail version of an image, for example. Even if you do not want to rescale an image, it is good practice to include these attributes in each `img` start tag with values that represent the original (unscaled) size of the image. Specifying values for the `height` and `width` attributes in all `img` elements makes it possible for the browser to reserve space for page images before downloading them. Otherwise, the browser may reserve a default amount of space for each image in the page, initially display the document with a placeholder inserted in place of each image, and then adjust the layout of the document as it determines the actual size of each image during image downloading. If you've ever seen a document change layout in this way while you were trying to read it or click on one of its links, you may know how annoying this can be to a user.

The value specified for a `width` or `height` attribute is by default interpreted as a length in pixels. The term *pixel* is short for "picture element" and represents one "dot" on a display. A typical display is composed of a grid of such dots, and an image is formed on the monitor by causing each dot to be displayed in a particular color. The resolution of a display is specified in terms of pixels. For example, a display resolution of 1280 by 1024 corresponds to a grid of 1280 pixels across by 1024 pixels from top to bottom.

An alternative to specifying a length in pixels is to specify it as a percentage of the height or width of the client area of the browser. For example, markup such as

```
<img ... height="4" width="100%" ... />
```

could be used to create a custom horizontal rule that stretches an image out so that it spans the entire width of the browser's client area. That is, a `height` or `width` attribute value ending in a percent sign (`%`) is interpreted as a percentage rather than as a length in pixels.

By the way: if you don't know the pixel dimensions of an image that you want to include in an HTML document, you can load the image into Mozilla, right-click on it, and select Properties from the pop-up context menu. The dimension of the image in pixels will be displayed, along with other information.

Each image will by default be placed at the location of its `img` element in the document without any preceding or trailing line breaks. In other words, the browser by default includes each image in the document as if it were a single character. This default behavior can be overridden by the `style` attribute. For example, the `img` element in Figure 2.12 causes the associated image to be displayed side by side with text, as illustrated in Figure 2.13.

2.4.6 Links: The `a` Element

Finally, we come to the core "hypertext" part of HTML: the `a`, or *anchor*, element (the reason for this name will be discussed in a moment). This element is the primary means of creating a clickable link (a *hyperlink*) within a document. The anchor in our example appeared in the following context:

```
See
<a href="http://www.w3.org/TR/html4/index/elements.html">the
  W3C HTML 4.01 Element Index</a>
for a complete list of elements.
```

Most browsers display the textual content of an anchor element underlined and in a distinctive color, and the browser's cursor will normally change in some way when placed over this content to indicate that it is a hyperlink (although the default appearance of a hyperlink can be changed using style sheets). The `href` attribute of an anchor element specifies the URL of a document to be requested via the HTTP GET method if the link corresponding to the anchor is clicked by the user. When the browser receives a document in the HTTP response to this request, it will by default display this new document in place of the one containing the hyperlink. This default behavior can be overridden by certain attribute settings, as we'll learn later. In order to avoid possible browser incompatibilities, it is best to have no leading or trailing white space in the content of an anchor element, as shown.

Although the content of an anchor is typically text, anchor elements can also contain certain other elements. Images are probably the most frequent alternative to text within anchors. As an example of including an image in an anchor, consider the following HTML taken from the W3C site (these lines of markup can be included in any XHTML 1.0 document that passes the W3C's validator tests):

```
<a href="http://validator.w3.org/check/referer"><img
    src="http://www.w3.org/Icons/valid-xhtml10"
    alt="Valid XHTML 1.0!" height="31" width="88" /></a>
```

This markup causes a graphics-capable browser to display an image that, when clicked, will cause the browser to generate an HTTP request for the URL specified as the value of the `href` attribute of the anchor.

You may be wondering why an a element is called an anchor. According to Chapter 12 of the HTML 4.01 recommendation, [W3C-HTML-4.01]: " A link has two ends—called anchors—and a direction. The link starts at the "source" anchor and points to the "destination" anchor, which may be any Web resource ... ".

What we have seen so far is the use of a as a source anchor. In XHTML, a destination anchor is specified by including an id attribute in the start tag of an a element. So, for example, an element such as

```
<a id="section1"></a>
```

could be included in a document, perhaps immediately before an h1 element with content Section 1. The syntax for legal strings that can be assigned to id attributes is given in Section 2.10.2. Also, note that HTML 4.01 browsers expect the name of a destination anchor to be specified using the name attribute of the a element rather than id. So it is wise to include specifications for both attributes, using the same value for both:

```
<a id="section1" name="section1"></a>
```

Furthermore, an attribute value that is intended to be used to identify a destination anchor should begin with a letter and consist entirely of letters, digits, and the four characters _:.- (underscore, colon, period, and hyphen).

To specify an anchor as the destination of a hyperlink, a string consisting of the anchor identifier along with a preceding crosshatch (#) is appended to a URL specifying the document containing the anchor (recall that such a string is called the *fragment* of the URL). So, for example, if a page with this destination anchor was at the URL http://www.example.org/PageWithAnchor.html, then the anchor could be referenced by a source anchor such as

```
<a href="http://www.example.org/PageWithAnchor.html#section1">...
```

If the hyperlink corresponding to this source anchor is clicked, the browser will load the referenced document and automatically scroll the page so that the location of the anchor specified by the fragment identifier is at the top of the page (or so that the page is scrolled to the bottom if the anchor is near the end of the page). Of course, if no anchor is specified in a URL, the browser scrolls the page so that the top of the page is at the top of the window. In essence, in this situation the browser acts as if there is a destination anchor at the very top of the document. We'll cover a shorter syntax for URLs containing fragments in Section 2.5.

2.4.7 Comments

A comment in HTML, like comments in other computer languages, is something that is intended to be read by programmers but to be ignored by the software processing the document. The comment in our example is

```
<!-- Notice that img must nest within a "block" element,
     such as p -->
```

As shown, a comment begins with the string of characters `<!--`, which must contain no white space. A comment ends with the string `-->`, again with no white space.

For obscure reasons related to XML's derivation from SGML, a pair of consecutive hyphens is not allowed following the initial `<--` of a comment except as part of the `-->` that closes the comment. So don't do anything like this:

```
<!-- This is NOT
  -- a good comment.
  -->
```

or even this:

```
<!-- Can't end with more than two hyphens! --->
```

2.4.8 Nesting Elements

Figure 2.12 shows several examples of *element nesting*, that is, including one element as part of the content of another element. For example, the markup

```
<tt><strong>hr</strong></tt>
```

nests a `strong` element within a `tt` element. This means that the text that is the content of the `strong` element will be displayed in a monospace font (every character has the same width) as well as a bold typeface (or some other "strong" fashion). Furthermore, in our example, both of these elements are part of the content of a `p` element. There is no HTML-imposed limit on the number of child elements that can be contained within a parent element or on the maximum depth of nesting of elements within a document. However, XHTML does require that every inner element must be ended before its enclosing outer element ends. So, for example,

```
<tt><strong>hr</tt></strong>
```

is not valid XHTML, even though most browsers will properly display the content.

It is also important to know which elements can and cannot nest within other elements. To a large extent, element nesting is related to which of two categories an element falls into, block or inline. Conceptually, a *block* element is an element such as `p` for which preceding and trailing line breaks are automatically generated by the browser. An *inline* element is one such as `span` that causes no automatic line break. In terms of nesting, generally speaking, block elements may have children that are either block or inline elements, but the children of inline elements must themselves be inline elements. So, nesting a `strong` element within a `tt` element is valid because both `tt` and `strong` are inline elements, but

```
<tt><p>hr</p></tt>
```

is invalid because we cannot nest the block element `p` within the inline element `tt`.

In Figure 2.12 I have put the start and end tags of block elements on lines by themselves, while for the most part I have written the tags for inline elements, well, "in line." If you compare this figure with Figure 2.13, the automatic line breaking effect of block elements should be apparent. As an aside, since the `br` element also generates a line break, you might conclude that it is a block element. However, generation of a line break is implicit for block elements, while for `br` it is the explicit purpose of the element. Therefore, `br` is considered an inline element.

As noted in the comment in Figure 2.12, the `img` element cannot appear directly as a child of the `body` element of a document. This is because—with few exceptions—the children of the `body` element must belong to the block category.

The detailed rules concerning how elements nest vary somewhat between HTML versions, and can be expected to continue to change somewhat in the future. After you have learned how to read XML grammars later in this chapter, you will be able to read the source for yourself to know exactly what the nesting relationships are in any version of XHTML. So I will not attempt to provide all of those details here. However, we will look in later sections at several types of elements—lists, tables, framesets, and forms—that each have fairly well-defined nested element content. Before moving to those elements, though, we will cover some details about URLs that are important for every HTML developer to understand.

2.5 Relative URLs

As we have seen, URLs are used as the values of some attributes in HTML documents. In this section, we'll look more closely at the forms of URLs that may be used as values within HTML documents and at how URLs are interpreted by web servers such as Tomcat.

First, recall that by default the `webapps/ROOT` subdirectory within your JWSDP 1.3 installation directory is the document base for the root URL path `/` of your Tomcat web server. In other words, given that you have installed a default JWSDP 1.3 on your machine and have started Tomcat, browsing on your machine to the URL `http://localhost:8080/MultiFile.html` will cause your server to look for a file named `MultiFile.html` in the `webapps/ROOT` directory.

Now let's assume that you have placed a file named `MultiFile.html` in your Tomcat `ROOT` directory that is identical to the document in Figure 2.12 except that the `src` attribute specification of the `img` element is changed to

```
src="valid-xhtml10.png"
```

(`MultiFile.html` is available for download from the textbook Web site listed in the Preface.) Furthermore, assume that you have placed a file (also available at the textbook site) named `valid-xhtml10.png` in the `ROOT` directory, and that this file contains the graphic shown in the lower right corner of Figure 2.13. Then if you browse to `http://localhost:8080/MultiFile.html`, you should see content identical to that shown in Figure 2.13.

As you might guess, the reason that this works is that the browser interprets a `src` attribute value that looks like a file name—such as `valid-xhtml10.png`—as shorthand for a URL. In this case, since the URL of the document containing the `src` attribute was `http://localhost:8080/MultiFile.html`, the browser will interpret the string `valid-xhtml10.png` as shorthand for the URL `http://localhost:8080/valid-xhtml10.png`. This URL in turn represents the file named `valid-xhtml10.png` in the Tomcat `ROOT` subdirectory, so this file will be returned by the server when this URL is requested.

In general, a string that is consistent with the URL path syntax given earlier (Sec. 1.6.2) is known as a *relative URL*; as we are about to learn, certain other strings are also valid relative URLs. A complete URL—one beginning with a scheme—is known as an *absolute URL*. Every relative URL is shorthand for some absolute URL. Unless otherwise noted, a relative URL can be used within an HTML document anywhere that an absolute URL can be used.

To convert a relative URL to an absolute URL, a *base URL* is used. When a browser requests an HTML document, by default the base URL used for converting any relative URLs contained within the document to absolute URLs is the document's URL, with any query string and fragment removed. Thus, for a document at `http://localhost:8080/MultiFile.html`, the default base URL is also `http://localhost:8080/MultiFile.html`. (The default base URL can be overridden by using the HTML `base` element; see the exercises.)

A relative URL that looks like a file name, such as `valid-xhtml10.png`, is converted to an absolute URL by replacing any characters following the final slash (`/`) in the base URL with the relative URL. This is why in the example just described the relative URL corresponded to `http://localhost:8080/valid-xhtml10.png`. Another valid relative URL is the empty string. In this case, the corresponding absolute URL is the entire base URL (any characters following the final slash are retained).

If a relative URL contains a query string and/or fragment, then the rules just given will be applied to the portion of the relative URL preceding the query and fragment strings, and then these strings will be appended to the generated absolute URL. For example, if a document at `http://www.example.org/PageWithAnchor.html` contained the markup

```
<a href="#section1">...
```

then the corresponding absolute URL would be `http://www.example.org/PageWithAnchor.html#section1` (because the relative URL with the fragment excluded is the empty string, which corresponds to the full base URL).

More sophisticated relative URLs can be formed using the syntax (similar to that used in file systems such as Linux) illustrated in Table 2.4. As shown, the string `../` in a relative URL means that a segment should be removed from the path component of the base URL before appending the remainder of the relative URL to the base. Repeating this string means that multiple segments should be removed. Beginning a relative URL with a slash (`/`) character means to replace the entire path component of the base URL with the given relative URL. Complete URL details, including unusual relative URLs such as `/../a.html`, are contained in [STD-66].

Such relative URLs are particularly useful if you store files in different directories on your web server. For instance, suppose that you would like to keep your HTML documents

TABLE 2.4 Absolute URLs Corresponding to Relative URLs When the Base URL is `http://www.example.org/a/b/c.html`

Relative URL	Absolute URL
`d/e.html`	`http://www.example.org/a/b/d/e.html`
`../f.html`	`http://www.example.org/a/f.html`
`../../g.html`	`http://www.example.org/g.html`
`../h/i.html`	`http://www.example.org/a/h/i.html`
`/j.html`	`http://www.example.org/j.html`
`/k/l.html`	`http://www.example.org/k/l.html`

separate from image files, so you create directories named `doc` and `img` within the `ROOT` directory of your Tomcat server. You then put a copy of `MultiFile.html` in the `doc` directory and a copy of `valid-xhtml10.png` in the `img` directory. You could then modify this copy of `MultiFile.html` as follows:

```
src="http://localhost:8080/img/valid-xhtml10.png"
```

Then browsing to `http://localhost:8080/doc/MultiFile.html` would once again produce Figure 2.13. However, so would

```
src="../img/valid-xhtml10.png"
```

In addition to being shorter to type, the relative URL version of this attribute has the advantage that if you were to later move the `doc` and `img` directories to another location on your server—or even to a different web server entirely—you would not need to modify the `src` value. Thus, relative URLs should be used whenever possible to facilitate portability of HTML documents.

The next several sections consider specific HTML elements used for structuring data within a document.

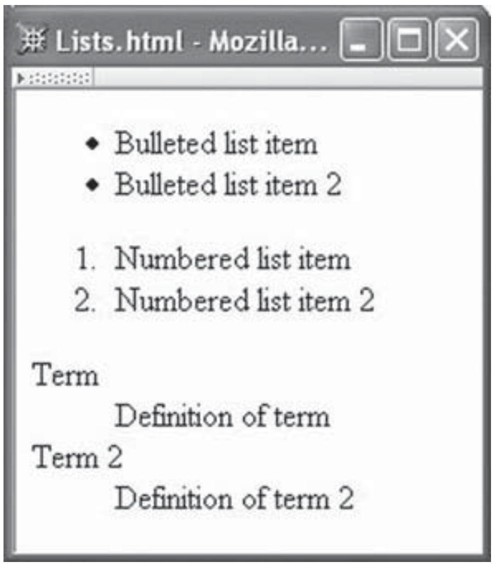
2.6 Lists

Figure 2.14 illustrates the three types of lists supported by HTML:

- *Unordered:* A bullet list
- *Ordered:* A numbered list
- *Definition:* A list of terms and definitions for each

This figure was produced by the following HTML:

```
<ul>
  <li>Bulleted list item</li>
  <li>Bulleted list item 2</li>
</ul>
```



- Bulleted list item
- Bulleted list item 2

1. Numbered list item
2. Numbered list item 2

Term
        Definition of term
Term 2
        Definition of term 2

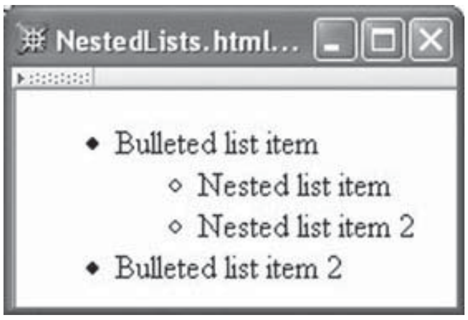
FIGURE 2.14 Browser rendering the three HTML list types.

```
<ol>
  <li>Numbered list item</li>
  <li>Numbered list item 2</li>
</ol>
<dl>
  <dt>Term</dt>
  <dd>Definition of term</dd>
  <dt>Term 2</dt>
  <dd>Definition of term 2</dd>
</dl>
```

As shown, the HTML syntax for all three types of lists is similar. First, the type of list is indicated by using either a `ul` (unordered list), `ol` (ordered list), or `dl` (definition list) start tag (each of the tag names ends in the letter "el," not the number "one"). All three elements are block elements, so a list by default begins on a new line when displayed in the browser. Each item in an unordered or ordered list is made the content of an `li` (list item) element. For a definition list, each term is made the content of a `dt` (definition term) element, and each term description is made the content of a `dd` element that immediately follows the `dt` element being described. For all three types of list, the list is terminated by following the last item in the list with the appropriate end tag.

Lists can be nested to produce an outline layout. For example, the markup

```
<ul>
  <li>Bulleted list item
    <ul>
      <li>Nested list item</li>
      <li>Nested list item 2</li>
    </ul>
  </li>
  <li>Bulleted list item 2</li>
</ul>
```



- Bulleted list item
  - Nested list item
  - Nested list item 2
- Bulleted list item 2

FIGURE 2.15 Browser rendering nested unordered lists.

can be used to produce the output shown in Figure. 2.15. The format of the bullets in this display (or the type of numbers used, if this was an ordered list) can be defined using style sheets as discussed in the next chapter.

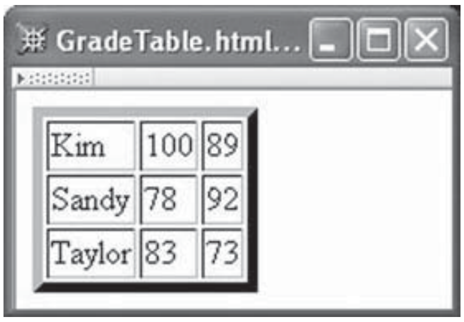
2.7 Tables

HTML provides a fairly sophisticated model for presenting data in tabular form. Columns and rows will automatically size to contain their data, although there are also various ways to specify column widths; individual table cells can span multiple rows and/or columns; header and/or footer rows can be supplied; and so on. There are also various options for changing the visual appearance of a table, such as the widths of its internal cell-separating lines (*rules*) and external borders. Most of the visual features will be discussed in the next chapter on style sheets. In this chapter, some of the basic features of structuring and formatting HTML tables will be presented.

Simple tables are simple to represent in HTML. For example, a table of student grades could be written as follows and produces the table shown in Figure 2.16:

```
<table border="5">
  <tr>
    <td>Kim</td><td>100</td><td>89</td>
  </tr>
  <tr>
    <td>Sandy</td><td>78</td><td>92</td>
  </tr>
  <tr>
    <td>Taylor</td><td>83</td><td>73</td>
  </tr>
</table>
```

As shown in this example, the `table` element is used to define an HTML table. We will come back to the `border` attribute used in this tag in a moment. A `tr` (table row) element is used to contain each row. Within a row, a `td` (table data) element marks each element of the row. Notice that we don't need to specify the number of rows and columns in the table explicitly. Instead, these values are determined automatically: in a simple table, the number of rows is determined by the number of `tr` elements in the table, and the number of columns is determined by the maximum number of `td` elements contained within any row.



Kim	100	89
Sandy	78	92
Taylor	83	73

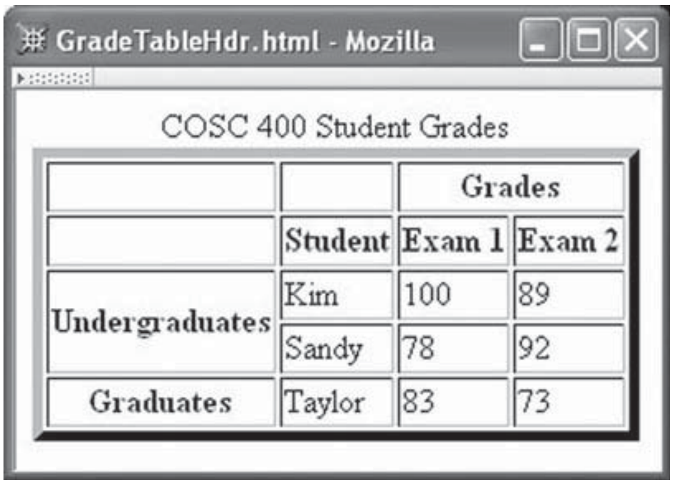
FIGURE 2.16 A simple table of grades.

In this example, since there are three `tr` elements, each containing three `td` elements, the table is 3 by 3. Finally, notice that the width of table columns is also automatically adjusted to contain the maximum width item in any column, although this can be overridden via the `style` attribute.

In this example, the `border` attribute in the `table` start tag tells the browser to display the table using a 5-pixel-wide border and 1-pixel-wide rules. In general, if any positive integer *n* is used for the value of `border`, then an *n*-pixel-wide border and 1-pixel-wide rules will be displayed. A value of 0 for this attribute turns off both the border and the rules. Additional `table` attributes are available that can be used to control the style of a table, but it is probably better to use style sheets for more advanced style settings, as discussed in the next chapter.

The table in this example is not very informative by itself. For example, there is no table caption, and there are no headers to define what the columns represent. This is easily corrected as shown in the next example and the accompanying Figure 2.17:

```
<table border="5">
  <caption>
    COSC 400 Student Grades
  </caption>
  <tr>
    <td>&nbsp;</td><td>&nbsp;</td><th colspan="2">Grades</th>
  </tr>
  <tr>
    <td>&nbsp;</td><th>Student</th><th>Exam 1</th><th>Exam 2</th>
  </tr>
  <tr>
    <th rowspan="2">Undergraduates</th><td>Kim</td><td>100</td><td>89</td>
  </tr>
  <tr>
    <td>Sandy</td><td>78</td><td>92</td>
  </tr>
  <tr>
    <th>Graduates</th><td>Taylor</td><td>83</td><td>73</td>
  </tr>
</table>
```



COSC 400 Student Grades

		Grades	
	Student	**Exam 1**	**Exam 2**
Undergraduates	Kim	100	89
	Sandy	78	92
Graduates	Taylor	83	73

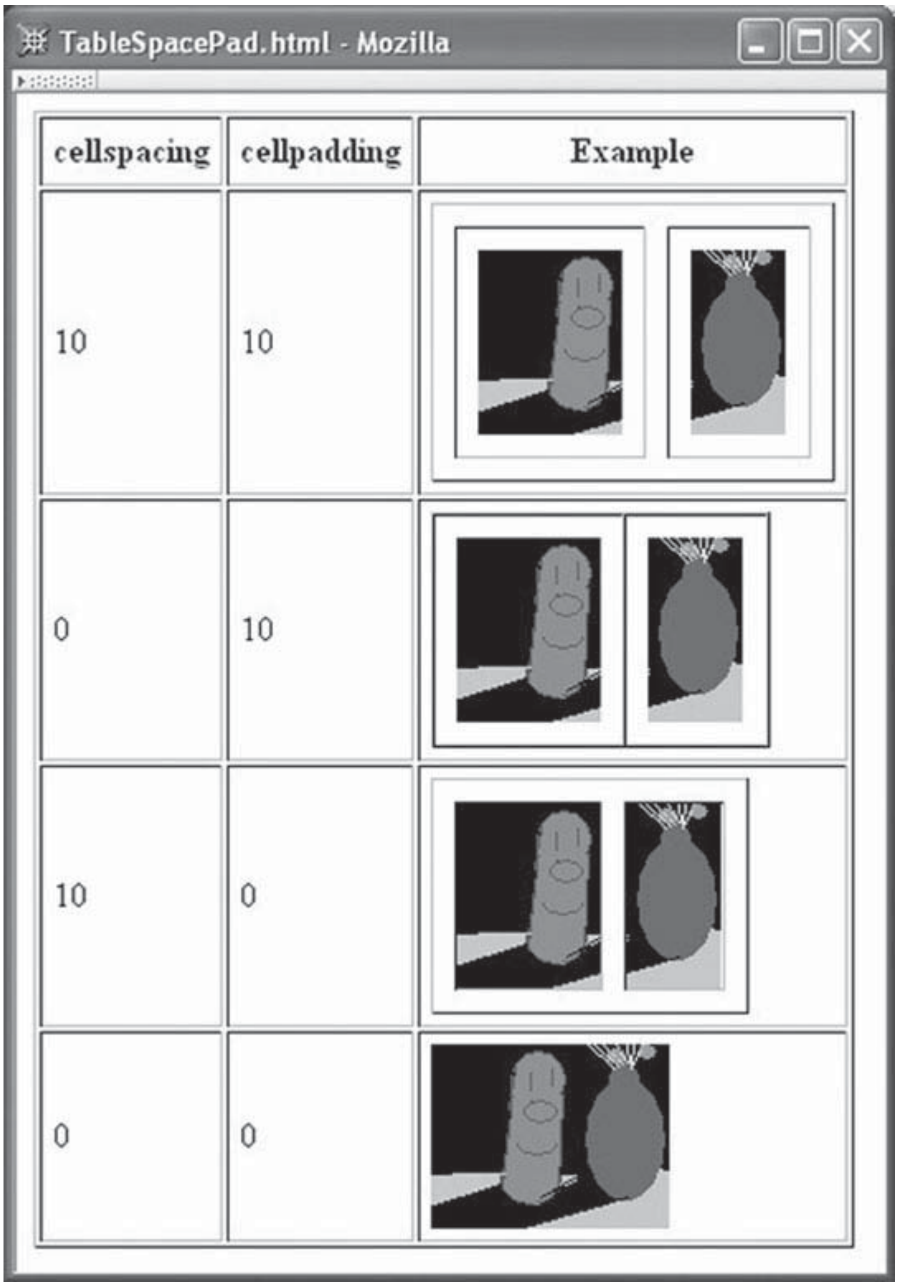
FIGURE 2.17 Table with headings and caption.

Two new elements are used in this example. The `caption` element, as the name implies, is used to define a caption for the table. If a `caption` element is used with a table, the `caption` start tag must must appear immediately after the start tag of the `table` element. The second new element, `th` (table header), is much like the `td` element, except that a typical browser will format the content of a `th` element in boldface and center it horizontally within the column.

Also notice in this example the use of empty table elements to skip a column. For example, the second row of the table begins with `<td>&nbsp;</td>` to indicate that the first column of the second row should be left blank. The `&nbsp;` reference is included to ensure that the cell rules are displayed; my version of Internet Explorer 6, for one, will not display the cell rules if the content of a `td` element is empty or solely white space.

Finally, the example illustrates two new attributes that can be used with `td` and `th` elements: `colspan` and `rowspan`. Here `colspan` is used to tell the browser that a table element should cover more than one column, as is the case for the `Grades` heading in the example. `rowspan` is used for an element that covers more than one row, as illustrated by the `Undergraduates` heading in the example. Notice that on the row for student Sandy, no empty element is used: the row simply begins with an element containing `Sandy`. This is because the `Undergraduates` cell already occupies the first column due to the use of `rowspan`.

For performance and other reasons, if a large image is to be displayed on a web page, it will often be sliced into several smaller images that are downloaded separately and displayed next to one another to recreate the large image. Tables are frequently used to position the smaller images adjacent to one another so that they appear to be a single larger image. In order to achieve this effect, some table defaults must be overridden. Specifically, the `table` element has two attributes that control spacing within the table: `cellspacing` and `cellpadding`. Figure 2.18 illustrates how changes to the values of these attributes affect the spacing between table cells. The `cellspacing` attribute determines the amount of space between two adjacent cells, or between a side of the cell and the border of the table, while `cellpadding` determines the amount of space between the content of a cell (an image in this example) and the edge of the cell. In the top row of the example, the rule around each cell is visible, as is the border of the overall table containing the cells. In the second row, with



cellspacing	cellpadding	Example
10	10	
0	10	
10	0	
0	0	

FIGURE 2.18 Effects of `cellpadding` and `cellspacing` attributes. Each element in the Example column is a table containing one row of two image elements. The top three Example tables have `border` set to 1 so that the table border and rules will be visible; the last Example table has `border` set to 0 so that there is no line between the two images. The image is courtesy of Ben Jackson.

`cellspacing` turned off, the cell rules are immediately adjacent to the table border and to one another. The third row shows that when `cellpadding` is turned off, the cell rules are immediately adjacent to the content of the cells. Finally, the last row uses the following table start tag:

```
<table border="0" cellspacing="0" cellpadding="0">
```

and, as shown, the two smaller images appear as if they are a single image.

In fact, this example illustrates another feature of HTML tables: tables can be recursively nested within tables. The markup that generated Figure 2.18 begins as follows:

```
<table border="1" cellpadding="5">
  <tr>
    <th>cellspacing</th><th>cellpadding</th><th>Example</th>
  </tr>
  <tr>
    <td>10</td><td>10</td>
    <td id="nested">
      <table border="1" cellspacing="10" cellpadding="10">
        <tr>
          <td>
            <img src="CFP1.png" style="display:block"
              alt="Cucumber and Flower Pot" height="86" width="67" />
          </td>
          <td>
            <img src="CFP2.png" style="display:block"
              alt="Cucumber and Flower Pot" height="86" width="45" />
          </td>
        </tr>
      </table>
    </td>
  </tr>
  ...
```

Notice that the content of the `td` element with `id` value `nested` (the third element of the second row of the outer table) is another table. This inner table could also contain tables, and so on to any desired depth of recursion.

Finally, you should also observe that each `img` element in this example assigns a value of `display:block` to its `style` attribute. This is required because images are considered inline HTML elements, and such elements by default are displayed with a little bit of space underneath them (allocated for the *descenders* of certain characters, such as `p` and `q`, that display below the baseline of text). The `style` attribute specification given overrides this default behavior by indicating that the image should be treated as a block element for display purposes.

Tables, then, are one way of laying data out on a display. We turn next to an alternative HTML layout mechanism that has some advantages—and disadvantages—compared with the more traditional table concept.

2.8 Frames

HTML frames are essentially a means of having several browser windows open within a single larger window. Figure 2.6 is an example of a browser window containing three frames (two on the left and one larger one on the right). Such a window is created by using one or more `frameset` elements after the `heading` element, rather than using a `body` element as all of our previous pages have used. The document type declaration is also different for framed pages than it is for standard web pages. For example, the window in Figure 2.6 could be created by HTML similar to the following:

```
<!DOCTYPE html
   PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Java 2 Platform SE v1.4.2</title>
  </head>
  <frameset cols="20%,80%">
    <frameset rows="1*,2*">
      <frame src="overview-frame.html"
        id="upperLeftFrame" name="upperLeftFrame"></frame>
      <frame src="allclasses-frame.html"
        id="lowerLeftFrame" name="lowerLeftFrame"></frame>
    </frameset>
    <frame src="overview-summary.html"
        id="rightFrame" name="rightFrame"></frame>
  </frameset>
</html>
```

The first `frameset` statement says to create two rectangular subspaces, or views, within the browser window. The first (and therefore leftmost) of these subspaces covers 20% of the width of the browser window, and the second covers the remaining 80% of the window. Both subspaces cover the browser window from top to bottom, because only the `cols` (columns) attribute is specified. This top-level frameset element contains two child elements: another frameset and a frame (the one named `rightFrame`; as with fragment identifiers in anchors, `id` is the attribute used for naming frames in XHTML, and `name` in HTML 4.01). The child frameset specifies that its subspace (the left 20% of the browser window) is further divided into two views. Since these two views are specified using the `rows` attribute, they are stacked one on top of the other and both occupy the full width of the child frameset's subspace. The notation `1*,2*` indicates that the vertical space should be allocated so that the second (and therefore lower) view is twice as tall as the first view. If the value of the `rows` attribute had instead been `3*,2*`, the top view would have occupied 3/5 of the height of the subspace, and the lower view 2/5.

Framesets, then, can be viewed as something like tables: they can be used to lay out information within the browser. Also like tables, framesets can be defined recursively, with one frameset defined within another. A key difference between framesets and tables, however, concerns the contents: ultimately, at the leaves of a tree of frameset elements, frame elements are required. Each frame is essentially a browser window, which occupies a subspace of the screen as defined by the frameset(s) containing the frame. In the example considered, the frame named `upperLeftFrame` occupies the upper left corner (20% wide by 1/3 high) of the browser window, the frame `lowerLeftFrame` the lower left corner (20% wide by 2/3 high), and the frame `rightFrame` the right side (80% wide by 100% high).

The `src` attribute of a `frame` tells the browser the URL of a document to be loaded into the frame initially. If an HTML document is loaded into the frame and the user clicks a hyperlink within that frame, then the document named in the `href` attribute of the hyperlink's a element will be loaded into the frame. Other frames in the browser window will not be affected.

However, it is also possible for activity in one frame to cause a change in another frame, and this is in fact one of the main reasons for using frames. For example, the HTML contained in the frame named upperLeftFrame might contain an anchor such as the following:

```
<a href="java/applet/package-frame.html" target="lowerLeftFrame">
```

Because this anchor specifies lowerLeftFrame as the value of its target attribute, if the user clicks the link corresponding to this anchor, then the URL specified by the href attribute of the anchor will be loaded not into upperLeftFrame—the frame that contains the link—but instead into lowerLeftFrame.

One use of frames, then, is to provide a relatively easy means of writing HTML documents that provide navigation tools. That is, one page (a navigation page) can be written that contains a number of links to other documents in some collection of documents. This navigation page can be loaded into its own frame, and a second frame can be specified as a target for each of the anchor elements contained in the navigation page. A nonframe alternative is to include the navigation links directly in every page of the collection. This not only increases the difficulty of writing these pages, but also does not provide a separate scrollbar for the navigation links, which is an automatic and useful feature of a frame-based approach.

While frames can therefore be useful, there are also reasons to avoid using frames. The major reason is that frames can cause confusion for end users. For example, if the user is at a page that consists of multiple frames and clicks the browser's Print button, what is printed may not be what the user expected. The confusion can be especially acute for someone who is visually impaired and is attempting to "view" a frames-based document by using assistive software to translate the document into a verbal description. Another reason to avoid frames is that they assume that the document is going to be displayed on a monitor large enough to comfortably accommodate them. This is probably not a good assumption if the document is displayed on a cell phone.

For these and other reasons, many web developers avoid using frames. A key exception is that frames are still used for certain specialized technical applications—such as documents produced using the Javadoc^{TM} documentation system—in which the end users may be expected to have the expertise needed to deal with the vagaries of frames and to be viewing documents on standard monitors.

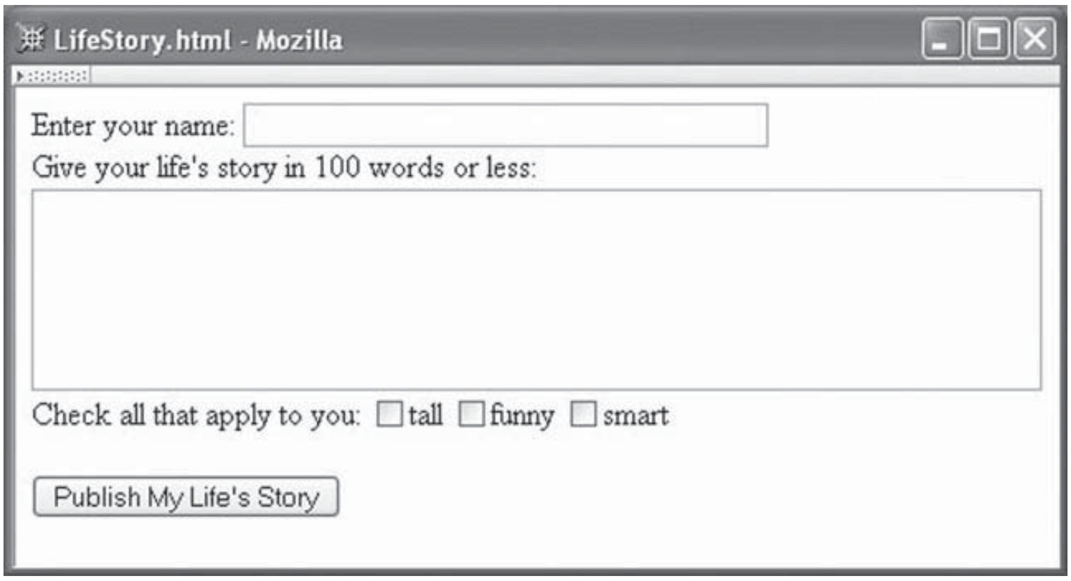
2.9 Forms

An HTML *form* is used to allow a user to input data on a web page. Figure 2.20 is the form produced by the HTML form element of Figure 2.19.

We'll begin by considering the attributes used in this example's form element. The value of the required action attribute specifies a URL to which the information collected on the form should be sent when the user *submits* the form (more on form submission toward the end of this section). In this example, the form information will be placed in an HTTP request and sent to the URL www.example.org (which is a host name used for

```
<form action="http://www.example.org" method="get">
  <div>
    <label>
      Enter your name: <input type="text" name="username" size="40" />
    </label>
    <br />
    <label>
      Give your life's story in 100 words or less:
      <br />
      <textarea name="lifestory" rows="5" cols="60"></textarea>
    </label>
    <br />
    Check all that apply to you:
    <label>
      <input type="checkbox" name="boxgroup1" value="tall" />tall
    </label>
    <label>
      <input type="checkbox" name="boxgroup1" value="funny" />funny
    </label>
    <label>
      <input type="checkbox" name="boxgroup1" value="smart" />smart
    </label>
    <br /><br />
    <input type="submit" name="doit" value="Publish My Life's Story" />
  </div>
</form>
```

FIGURE 2.19 An example HTML form element.



LifeStory.html - Mozilla

Enter your name:

Give your life's story in 100 words or less:

Check all that apply to you: ☐ tall ☐ funny ☐ smart

Publish My Life's Story

FIGURE 2.20 Example of an HTML form.

example purposes only, so submitting this form won't do anything interesting). The `method` attribute is used to specify the HTTP method that will be used to make the request. There are only two choices for this attribute value: `get` (the default value) and `post`. In XHTML, these values must be lowercase; however, the HTTP convention is to write these method names in uppercase, and I will follow this convention when referring to these methods in the text. We'll defer a discussion of when to use each method to Chapter 6. Suffice it to say that the GET method has been used here for example purposes and would normally not be the appropriate choice for a form such as this.

Next, notice that the first element contained within the form—and in fact the only immediate descendant of the `form` in the element tree—is a `div` element. `div` is almost identical to the `span` element encountered earlier: both elements are used to wrap other information in a document so that the wrapped information can be treated as a unit. The difference is that `div` is considered a block-type element, while `span` is an inline element. The `div` element is inserted here because the XHTML grammar requires that any child of a `form` element be a block, and most of the other elements we want to insert in the form are inline rather than block elements. Wrapping all of this other content of the `form` within a `div` is a simple way of satisfying the form element's constraint on its children. There are also other alternatives, such as using a table or a `p` element to wrap the form content. I have used a `div` here because it is a little simpler to use than a table for laying out this form, and it does not insert any vertical space at the beginning of the form, as a `p` element would.

The next element in this example is a `label`. This element is used to associate text with another element of the form. Only one form element may be contained in the content of a `label` element.

The `input` and `textarea` elements contained within the first two labels are examples of HTML *controls*. Each control provides a particular type of input element on a web page. The first control on this page is a *text field*, which is generated using an `input` element having a `type` attribute with value `text`. This control simply displays a box, one character tall by a number of characters wide that is specified using the `size` attribute (40 characters in this case). The text field acts like a small text editor, allowing the user to enter characters, modify previously entered information, cut and paste data in the field, etc. Default text can be included in a text field when the form is displayed by including a `value` attribute in the text field's `input` start tag; the value of this attribute is displayed as the default text. Finally, text fields, like every form control, should have a `name` attribute, for reasons discussed in later chapters.

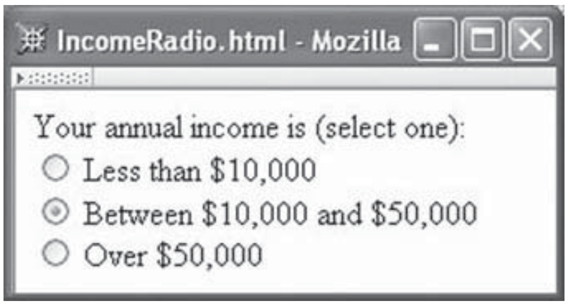
The second control in this form is a *textarea*. This is similar to a text field, but allows the user to enter multiple lines of data. One other key difference is that `input` is an empty element (has no end tag), while `textarea` is not empty. Any character data placed between the start and end tags of a textarea is displayed as default text in the textarea box when the page is loaded. Another difference is that the size of a textarea is specified using the two attributes `rows` and `cols`, which specify the number of lines and the number of characters per line, respectively; the `size` attribute of text fields is not an attribute of `textarea`. Finally, like text fields, any information in the textarea can be edited by the user.

The next controls on this form are three checkboxes. A *checkbox* is a simple control: clicking on it toggles whether or not the checkbox appears to be *checked* (in Mozilla 1.4, a checked box is darker than an unchecked box). As shown here, the `input` element specifying a checkbox normally has a `value` attribute assigned to it. This value is the information that will be returned to the web server if the user checks the corresponding checkbox and *submits* the form as described toward the end of this section.

The checkboxes in this example can be checked or not in any possible combination. If instead we wanted to allow exactly one of several options to be selected, we would use a set of *radio button* controls. An example using radio buttons is

```
Your annual income is (select one):<br />
<label>
  <input type="radio" name="radgroup1" value="0-10" />
    Less than $10,000
</label><br />
<label>
  <input type="radio" name="radgroup1" value="10-50"
         checked="checked" />
    Between $10,000 and $50,000
</label><br />
<label>
  <input type="radio" name="radgroup1" value="&gt;50" />
    Over $50,000
</label>
```

which is displayed in Figure 2.21. The key HTML difference is that we would use a `type` value of `radio` rather than `checkbox`. One or more radio button controls that all have the same value for their `name` attribute form a radio button *set*. The browser will always display exactly one of the radio buttons in a set as the selected button (the middle button is selected in Fig. 2.21). An HTML form indicates which button in a set should be checked initially by including a `checked` attribute on the appropriate control, as shown. If this is not done, the browser will arbitrarily choose a radio button to check initially. The `checked` attribute can also be used with a checkbox control to initialize it to a checked state rather than to its default unchecked state. `checked` is an example of a *boolean attribute*, which is an attribute that has a value that is false by default and that is made true by assigning the attribute a value that is the attribute's name (e.g., `checked="checked"`). One other small point to notice is that I have used an entity reference rather than a greater-than symbol in the value of the third radio button. The browser will replace this reference with the character code for greater-than before using this value for any purpose.



Your annual income is (select one):
○ Less than $10,000
● Between $10,000 and $50,000
○ Over $50,000

FIGURE 2.21 Form using radio buttons.

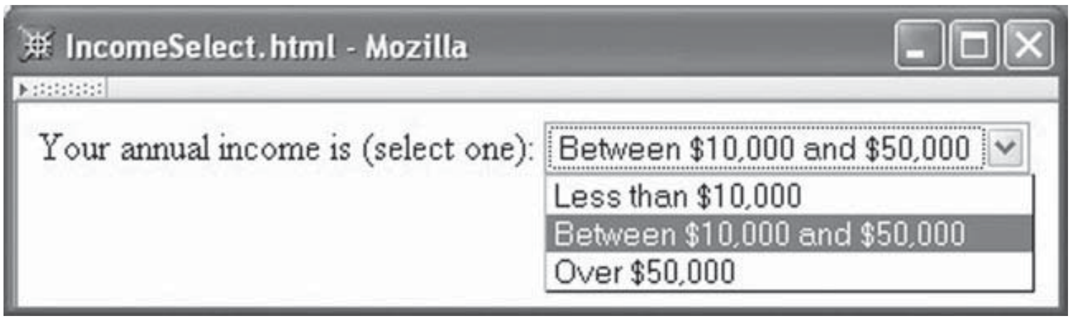

IncomeSelect.html - Mozilla
Your annual income is (select one):
Between $10,000 and $50,000
Less than $10,000
Between $10,000 and $50,000
Over $50,000

FIGURE 2.22 Form using `select` element. The user has just clicked the dropdown arrow to the right of the box representing the menu control. The “Between $10,000 and $50,000” value is initially shown in the menu box and is initially highlighted in the dropdown list.

Often, rather than using radio buttons to select a single option, a *menu* control is used. A menu is defined using yet another element, `select`. For example, we can rewrite the income controls using a menu rather than radio buttons as follows:

```
Your annual income is (select one):
<select name="income">
  <option value="0-10">Less than $10,000</option>
  <option value="10-50" selected="selected">
    Between $10,000 and $50,000
  </option>
  <option value="&gt;50">Over $50,000</option>
</select>
```

This markup produces the form shown in Figure 2.22. As shown in this example, the `option` element is used within a `select` to define a menu item. Notice two key differences between `option` and `input` with `type` `radio`. First, text is associated with a radio button only indirectly through the use of a label element, while the content of an `option` element directly specifies the text to be displayed in the menu. This can be an important difference, because it means that the look of a menu is not affected by resizing the browser window, whereas the look of text next to radio buttons can be significantly affected (Fig. 2.23).

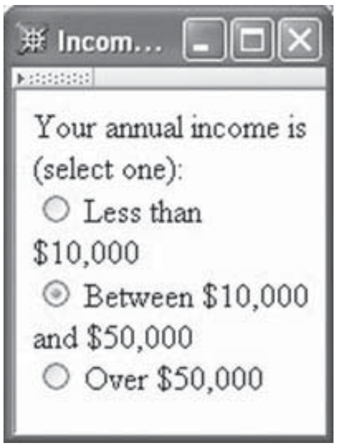

Incom...
Your annual income is (select one):
Less than $10,000
Between $10,000 and $50,000
Over $50,000

FIGURE 2.23 Resized window containing `IncomeRadio.html`.

A second difference is that the boolean `selected` attribute is used rather than `checked` to specify which option should be selected by default. As with radio buttons, it is good practice to write markup that sets the `selected` attribute on exactly one of the options of a menu to ensure that all users see the same initial menu value regardless of the browser used. However, there are situations where there is no obvious choice of an initial value for a menu. A common technique used in this situation is to add a menu item that essentially means “no value has been selected from this menu yet.” In the present example, we might apply this technique as follows:

```
Your annual income is (select one):
<select name="income">
  <option value="select1" selected="selected">
    Select one of the following:
  </option>
  <option value="0-10">Less than $10,000</option>
  <option value="10-50">Between $10,000 and $50,000</option>
  <option value="&gt;50">Over $50,000</option>
</select>
```

This produces the form shown in Figure 2.24.

The final control in Figure 2.20 is a *submit button*, which is defined using an `input` tag with `submit` as the value of the `type` attribute. When a submit button is clicked, the *values* of the form—such as text entered into text fields and textareas and the data associated with `value` attributes of checked radio buttons and checkboxes and selected menu items—are normally sent to the web server that generated the page. Alternatively, it is possible to process the values of a form using program code downloaded as part of the web page itself. Much more will be said about processing form data in later chapters. For now, you’ll notice that if you fill in the `LifeStory.html` form and submit it, the browser attempts to load the home page from `www.example.org` and the data entered on the form (possibly somewhat encoded) is included in the URL visited, as shown in the Location bar of the browser (Fig. 2.25).

Along with text input controls (text fields and textareas), checkboxes, radio buttons, menus, and submit buttons, there are several other controls defined in HTML. A

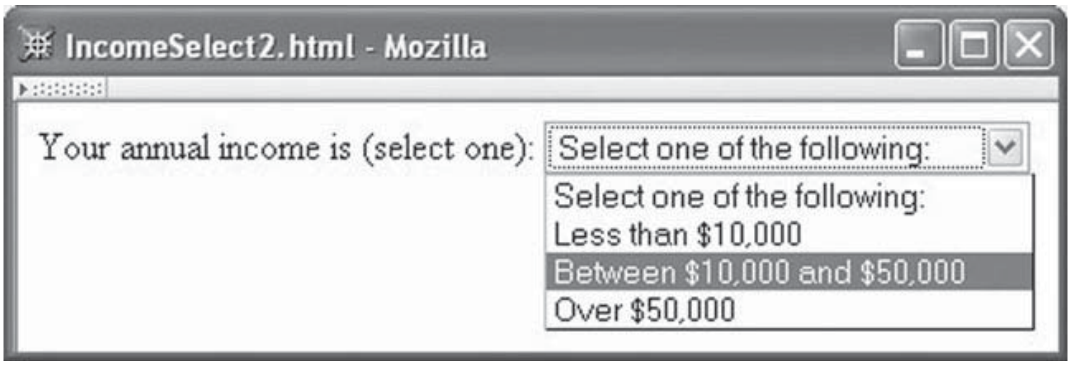
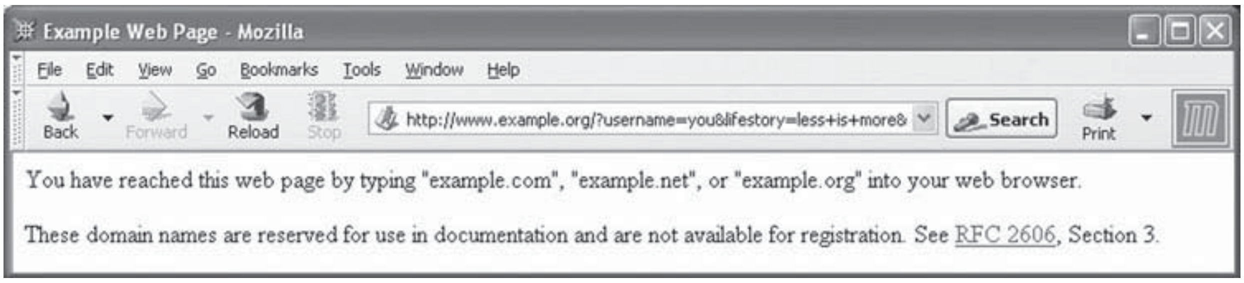

IncomeSelect2.html - Mozilla
Your annual income is (select one):
Select one of the following:
Select one of the following:
Less than $10,000
Between $10,000 and $50,000
Over $50,000

FIGURE 2.24 `Select` element with “select one” choice. The user is in the process of selecting the third item in the list.


Example Web Page - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
Search
Print

http://www.example.org/?username=you&lifestory=less+is+more&

You have reached this web page by typing "example.com", "example.net", or "example.org" into your web browser.

These domain names are reserved for use in documentation and are not available for registration. See RFC 2606, Section 3.

FIGURE 2.25 Page displayed when submit button is clicked after filling in the form. Note that several control names and values are displayed as part of the URL in the Location bar. The browser content is subject to copyright and used by permission of IANA.

complete list of nondeprecated form elements is contained in Table 2.5, and Figure 2.26 illustrates the visual appearance of the controls not already covered (`button` elements all have the same visual appearance regardless of the value of the `type` attribute, so only one `button` is shown). Detailed information about form controls is contained in the HTML 4.01 specification [W3C-HTML-4.01].

TABLE 2.5 HTML 4.01/XHTML 1.0 Nondeprecated Form Controls

Element	Type Attribute	Control
`input`	`text`	Text input
`input`	`password`	Password input
`input`	`checkbox`	Checkbox
`input`	`radio`	Radio button
`input`	`submit`	Submit button
`input`	`image`	Graphical submit button
`input`	`reset`	Reset button (form clear)
`input`	`button`	Push button (for use with scripts)
`input`	`hidden`	Nondisplayed control (stores server-supplied information)
`input`	`file`	File select
`button`	`submit`	Submit button with content (not an empty element)
`button`	`reset`	Cancel button with content (not an empty element)
`button`	`button`	Button with content but no predefined action
`select`	N/A	Menu
`option`	N/A	Menu item
`optgroup`	N/A	Heading in a hierarchical menu
`textarea`	N/A	Multiline text input
`label`	N/A	Associate label with control(s)
`fieldset`	N/A	Groups controls
`legend`	N/A	Add caption to a fieldset

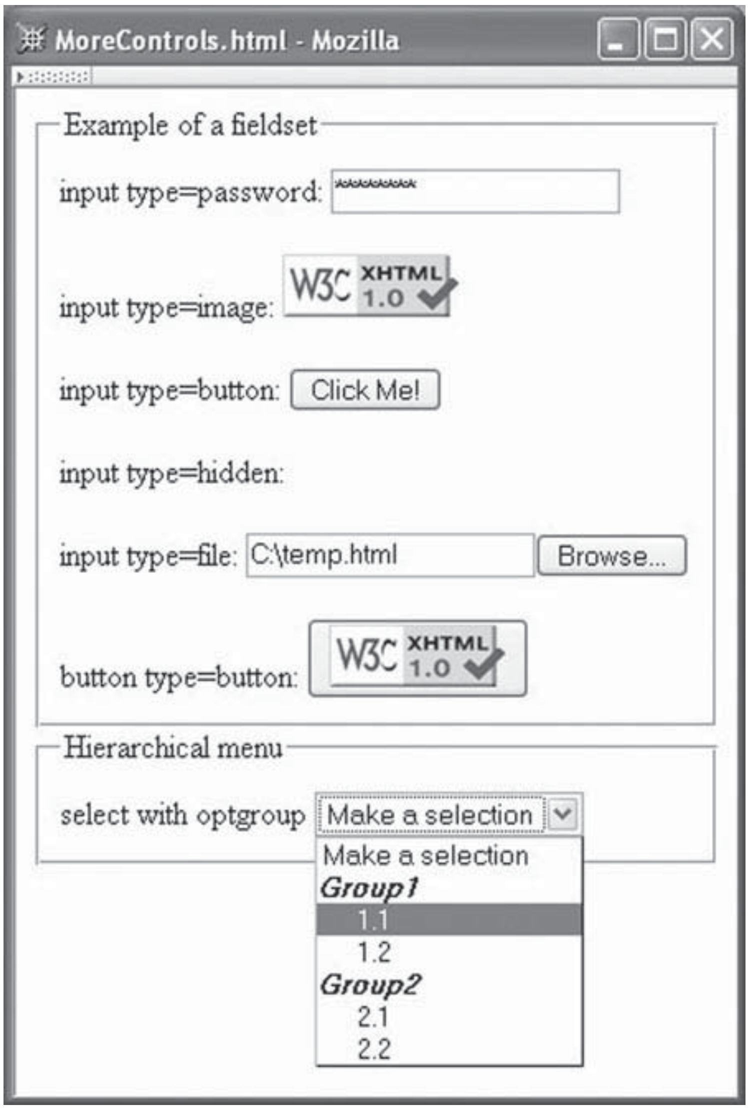

MoreControls.html - Mozilla
Example of a fieldset
input type=password: ********
input type=image:
W3C XHTML 1.0
input type=button: Click Me!
input type=hidden:
input type=file: C:\temp.html Browse...
button type=button:
Hierarchical menu
select with optgroup Make a selection
Make a selection
Group1
1.1
1.2
Group2
2.1
2.2

FIGURE 2.26 Additional HTML form controls.

2.10 Defining XHTML's Abstract Syntax: XML

Now that we understand basic XHTML concrete syntax and are familiar with the semantics of a number of XHTML elements, we're ready to see how the abstract syntax of a version of XHTML is defined using XML. By the end of this section, you should be able to read and understand the formal definition for any of the three flavors of XHTML 1.0.

The abstract syntax for each flavor of XHTML 1.0 is defined by a set of text files known collectively as an XML *document type definition* (DTD). To introduce you to the basic elements of a DTD, let's begin with a simple example drawn from an XHTML DTD:

```
<!ELEMENT html (head, body)>
<!ATTLIST html
  lang        NMTOKEN   #IMPLIED
  xml:lang    NMTOKEN   #IMPLIED
  dir         (ltr|rtl) #IMPLIED
  id          ID        #IMPLIED
  xmlns       CDATA     #FIXED 'http://www.w3.org/1999/xhtml'>
<!ENTITY gt       "&#62;">
```

The first of these lines is an example of an *element type declaration*. Element type declarations are used to specify the set of all valid elements in the language defined by the DTD. This example shows the actual element type declaration for the XHTML 1.0 `html` element. The information following the element type name is known as the *content specification* for the element; it provides information about the valid content of the element type being declared. This particular declaration says that, in XHTML 1.0 Strict, the `html` element must have two children, a `head` element followed by a `body` element. We'll describe content specifications in detail in Section 2.10.1.

The second line begins a tag that represents an XML *attribute list declaration*. As you might guess, this provides information about the valid attributes for an element, in this case for the `html` element. The attribute list declaration shown is equivalent to the actual XHTML 1.0 Strict declaration for `html` and says that this element type has five attributes: `lang`, `xml:lang`, `dir`, `id`, and `xmlns`. It also provides information such as the valid set of values for each attribute and default value information. More will be said about this in Section 2.10.2.

The final line is an example of an XML *entity declaration*. Such a tag is essentially a macro definition, associating the name `gt` (an *entity*) with the string `&#62;`. We have already learned how to "call" such macros using entity references such as `&gt;`. Now we can see more clearly how they are processed: the application reading a document containing an entity reference simply replaces the reference with the string represented by the entity, and then recursively processes this string. In this case, the string is a character reference, and the recursive processing will replace this reference with an appropriate encoding of the Unicode Standard value for the greater-than character. As we will learn in Section 2.10.3, the XHTML DTDs make extensive use of entity references within the DTDs themselves.

Each of the XHTML 1.0 DTDs is composed entirely of the three types of tag illustrated, along with entity references and comments (which have the same syntax as XHTML comments). So, at a high level, you now know about all of the pieces of these DTDs. In the next several subsections, we'll look at details of each of these three types of tags. We'll also see how DTDs are related to document type declarations.

2.10.1 Element Type Declarations

Let's begin by considering in more detail the `html` declaration from the XHTML 1.0 Strict DTD:

```
<!ELEMENT html (head, body)>
```

The string immediately following `ELEMENT` is the name of the *element type* being declared, in this case `html`. The XHTML DTD contains exactly one element type declaration for each element in the language. As already noted, the string following the element type name is the content specification for the element type (also referred to in some XML reference material

TABLE 2.6 Basic XML Content Specifications

Specification Type	Syntax	Content Allowed
Empty	`EMPTY`	None
Arbitrary	`ANY`	Any content (no restrictions)
Sequence	`(elt1, elt2, ...)`	Sequence of elements that must appear in order specified
Choice	`(elt1 | elt2 | ...)`	Exactly one of the specified elements must appear
Character data	`(#PCDATA)`	Arbitrary character data, but no elements
Mixed	`(#PCDATA | elt1 | elt2 |... )*`	Any mixture of character data and the specified elements in any order

as the *content model*). Several basic XML content specifications are shown in Table 2.6. These basic specifications are sufficient for defining the content model for many XHTML elements. For instance, the element type declaration for the `br` element is

```
<!ELEMENT br EMPTY>
```

In addition to the basic specifications shown, more sophisticated specifications may be formed by appending one of the iterator characters of Table 2.7 to the basic sequence and choice content specification types. So, for example, the XHTML DTD element type declaration

```
<!ELEMENT select (optgroup|option)+>
```

is a choice specification type modified with the `+` iteration character, and says that a `select` element may contain any number of `optgroup` and `option` elements in any order, as long as one or the other of these two elements appears at least once. Furthermore, sequence and choice specifications (optionally with iterator characters) can be nested, and an element name within a sequence or choice may have an iterator character suffixed to it. For example, the XHTML 1.0 Strict element type declaration for `table` is

```
<!ELEMENT table
     (caption?, (col*|colgroup*), thead?, tfoot?, (tbody+|tr+))>
```

TABLE 2.7 XML Content Specification Iterator Characters

Character	Meaning
?	Sequence/choice is optional (appears zero or one times).
*	Sequence/choice may be repeated an arbitrary number of times, including none.
+	Sequence/choice may appear one or more times.

This says that a `table` may optionally begin with a caption, followed optionally by either a sequence of `col` or `colgroup` elements, followed optionally by a `thead` and then optionally by a `tfoot` and finally a sequence of one or more `tbody` or `tr` elements. If you are familiar with BNF notation, you should see some clear similarities between the XML syntax for content specifications and BNF notation.

The keyword `#PCDATA` ("parsed character data") used in defining the character data and mixed content types represents any string of characters excluding less-than and ampersand, which are excluded because they represent the start characters for markup.

2.10.2 Attribute List Declarations

An *attribute list declaration* is included in the DTD for each element that has attributes. As noted earlier, an attribute list declaration begins with the keyword `ATTLIST` followed by an element type name and specifies the names for all attributes of the named element, the type of data that can be used as the value of each attribute, and default value information. For example, consider the example from the beginning of this section.

```
<!ATTLIST html
  lang        NMTOKEN   #IMPLIED
  xml:lang    NMTOKEN   #IMPLIED
  dir         (ltr|rtl) #IMPLIED
  id          ID        #IMPLIED
  xmlns       CDATA     #FIXED 'http://www.w3.org/1999/xhtml'>
```

Each line after the first is a sequence of three elements: the attribute name, the attribute type, and the default declaration.

The *attribute type* specifies the type of data that may be specified as the attribute value. Table 2.8 gives the attribute types used in the definition of XHTML 1.0 Strict. The attribute type for the first two attributes of the `html` element is `NMTOKEN` (name token), which is an XML keyword for a string of characters representing a name ("word"). The ASCII characters that can be used in a `NMTOKEN` are letters, digits, and the four characters period (.), hyphen (-), underscore(_), and colon (:). A variety of non-ASCII characters that are included in the Unicode Standard can also be used; see [W3C-XML-1.0] for details.

TABLE 2.8 Key Attribute Types Used in XHTML 1.0 Strict DTD

Attribute Type	Syntax	Usage
Name token	`NMTOKEN`	Name (word)
Enumerated	`(string1 | string2 | ... )`	List of all possible attribute values
Identifier	ID	Type for `id` attribute
Identifier reference	IDREF	Reference to an `id` attribute value
Identifier reference list	IDREFS	List of references to `id` attribute values
Character data	CDATA	Arbitrary character data (except < and &)

The string (ltr|rtl) is an example of an *enumerated* attribute type. Specifically, the attribute list declaration just given specifies that the dir attribute can be assigned one of two values: either the string ltr or the string rtl. As shown, the allowable values for an enumerated type are separated by OR (|) symbols. Note that boolean attributes—such as checked attribute of the input element—are also enumerated types, but they have only a single allowable value. So, for example, the checked attribute is declared as follows:

```
checked    (checked)     #IMPLIED
```

The ID attribute type is illustrated by the id attribute of the html attribute list declaration. An id attribute, as the name implies, supplies an identifying name for its element. Every XHTML 1.0 element has an id attribute, and we have already seen some uses for it on the a and frame elements. We will see many more uses in later chapters. Syntactically, ID values are almost identical to NMTOKEN values, except that an ID value is required to begin with a letter, underscore, or colon. Furthermore, XML imposes the constraint that no two attributes in a document are allowed to be assigned the same ID value, while no similar constraint applies to NMTOKENs. In XHTML 1.0, this means that every id attribute must be assigned a value that is distinct from the value of every other id attribute in the document. So the following markup fragment cannot appear in a valid XHTML 1.0 document:

```
<html id="anId" xmlns="http://www.w3.org/1999/xhtml">
  <head id="anId">
  ...
```

An attribute type of IDREF (an id reference) indicates that the value of the associated attribute must be identical to the value of the id of some element of the document. Attributes of this type are used for linking one element with another element. For example, a label element can be linked with a form control (such as a text field) in two ways: by including the form control as a child of the label element, as illustrated in Section 2.9, or by explicitly linking to the form control using the for attribute of a label element, which is declared in the XHTML 1.0 Strict DTD as follows:

```
<!ATTLIST label
  ...
  for         IDREF          #IMPLIED
  ...
>
```

The explicit form of label-to-control linkage is demonstrated by the following markup:

```
<table>
  <tr>
    <td><label for="username">Your Name</label></td>
    <td><input id="username" type="text" name="username" /></td>
  </tr>
  ...
</table>
```

TABLE 2.9 XML Attribute Default-value Declarations

Default Type	Syntax
No default value provided by DTD, attribute optional	`#IMPLIED`
Default provided by DTD, may not be changed	`#FIXED` followed by any valid value (quoted)
Default provided by DTD, may be overridden by user	Any valid value (quoted)
No default value provided by DTD, attribute required	`#REQUIRED`

In a valid XHTML document, the value of each `for` attribute—since it is of type `IDREF`—must exactly match the value of some `id` attribute within the document. The `IDREFS` attribute type is similar, except that it allows for a white-space-separated list of `id` values rather than the single `id` value allowed by `IDREF`.

The last attribute in the `html` element's attribute list declaration, `xmlns`, has a data type of `CDATA`. In general, the XML keyword CDATA represents any string of characters that excludes the two characters less-than (`<`) and ampersand (`&`), much like the `#PCDATA` keyword used in element content specifications. In the context of an attribute value, a string of type `CDATA` must also not contain the quoting character (either `"` or `'`) used to enclose the attribute value, as described earlier in Section 2.3.5. As noted in that section, the quote character can be included in an attribute value by using an entity or character reference, such as `&quot;` or `&#34;`.

The *default declaration* for an attribute specifies what value should be used if no value is specified for the attribute in an element of the document or if a value is assigned but does not conform to the attribute's type (for example, if a number is specified as the value for an `id` attribute in an XHTML document). The default declaration for an attribute can take one of the forms shown in Table 2.9. A value of `#IMPLIED` means that the corresponding attribute need not be assigned a value in the start tag for the element and that the DTD does not define a default value for the attribute. Instead, it is left up to the application reading the XML document to determine an appropriate default value. For XHTML, this means that the browser assigns a default value of its choice for any attribute marked with the `#IMPLIED` default.

The DTD itself can also supply a default value for an attribute. The following attribute list declaration illustrates this form of default declaration:

```
<!ATTLIST form
  ...
  method      (get|post)     "get"
  ...
>
```

In some cases, the default value of an attribute is not allowed to be overridden by the document. The is indicated by preceding the default value by the XML keyword `#FIXED`, as illustrated by the `xmlns` attribute of the `html` element in the earlier attribute list declaration. So, for example, the following tag is not allowed in a valid XHTML 1.0 document:

```
<html xmlns="http://www.w3.org">
```

An attribute with a default declaration of either `#IMPLIED` or an explicit default value is not required to be assigned a value in the start tag of the element (at least, not by the DTD; the XHTML 1.0 recommendation requires the presence of the `xmlns` attribute even though the DTD itself does not). The last possible default declaration, `#REQUIRED`, indicates that a value must be specified for the corresponding attribute whenever the element containing that attribute appears in a valid document. For example, the attribute list declaration for the XHTML `img` element contains lines equivalent to the following:

```
src  CDATA  #REQUIRED
alt  CDATA  #REQUIRED
```

This implies that every `img` start tag in an XHTML document must contain assignments to these two attributes.

2.10.3 Entity Declarations

We have already learned that an XML DTD can contain entity declarations, each of which begins with the keyword `ENTITY` followed by an entity name and its replacement text, such as

```
<!ENTITY gt      "&#62;">
```

This statement defines the value of the entity reference `&gt;` to be the string `&#62;`, which in turn is an XML character reference to the greater-than character (which corresponds to the code point 62 in the Unicode Standard).

An entity such as `gt` is known as a *general entity* and can only be referenced from within documents defined by the DTD. XML also provides for a different type of entity that can be referenced from within DTDs and not from documents. Such entities are called *parameter entities*. A parameter entity declaration is indicated in the DTD by following the `ENTITY` keyword with a percent sign (`%`), as in these two declarations taken from the XHTML 1.0 Strict DTD:

```
<!ENTITY % LanguageCode "NMTOKEN">
<!ENTITY % URI "CDATA">
```

Such an entity is referenced by prefixing the entity name with a percent sign rather than with the ampersand used for beginning a general entity reference. So, for example, given that the `LanguageCode` parameter entity has been defined as we did, the entity declaration (which is also contained in the XHTML 1.0 Strict DTD)

```
<!ENTITY % i18n
 "lang        %LanguageCode; #IMPLIED
  xml:lang    %LanguageCode; #IMPLIED
  dir         (ltr|rtl)      #IMPLIED"
  >
```

is equivalent to the declaration

```
<!ENTITY % i18n
 "lang        NMTOKEN        #IMPLIED
  xml:lang    NMTOKEN        #IMPLIED
  dir         (ltr|rtl)      #IMPLIED"
  >
```

As another example of the use of parameter entities, the actual XHTML 1.0 Strict attribute list declaration for the `html` element is

```
<!ATTLIST html
  %i18n;
  id          ID             #IMPLIED
  xmlns       %URI;          #FIXED 'http://www.w3.org/1999/xhtml'
  >
```

You should now be able to see that that is equivalent to the version of this declaration given earlier.

2.10.4 DTD Files

At this point, you should understand XML well enough to be able to understand the DTDs for any of the three XHTML 1.0 flavors. But where are these DTDs located, and how does a browser know which DTD to use for a given XHTML document?

Recall that for the Strict flavor of XHTML 1.0, the document type declaration we have been using is

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

As you might guess, the URL at the end of this tag is the location of a copy of the DTD for the document instance that follows the `DOCTYPE` tag. Technically, this portion of the `DOCTYPE` tag is known as the *system identifier* for the DTD. The string immediately following the `PUBLIC` keyword, on the other hand, is called the *formal public identifier* for the DTD. It is a name assigned to the DTD by the DTD's author according to SGML standards (which are not important to us here) and is essentially a URN for the DTD. While every XHTML 1.0 Strict document you write is required to begin with a `DOCTYPE` tag having the formal public identifier just given, the system identifier can be any URL of your choosing, as long as a copy of the DTD is available at the specified URL.

Actually, the file at the URL shown contains only a part of the overall XHTML 1.0 Strict DTD. Other portions of the DTD are contained in separate files that are imported into the DTD through a mechanism somewhat like the `#include` facility in C++. For example, the `xhtml1-strict.dtd` file contains the following lines of markup:

```
<!ENTITY % HTMLlat1 PUBLIC
   "-//W3C//ENTITIES Latin 1 for XHTML//EN"
   "xhtml-lat1.ent">
%HTMLlat1;
```

The first three lines are a special type of parameter entity declaration. This declaration is different from the ones we have seen previously in that its value is not a string but instead consists of a formal public identifier followed by a system identifier, much as would be found in a `DOCTYPE` tag. The system identifier in this case is a relative rather than absolute URL, and is considered to be relative to the URL of the document containing the entity declaration. So, if the system identifier in the `DOCTYPE` is as shown earlier, then the absolute URL corresponding to the relative `HTMLlat1` system identifier would be `http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent`. If you visit this file, you will see a long list of entity declarations such as

```
<!ENTITY nbsp   "&#160;"> <!-- no-break space = non-breaking space,
                                  U+00A0 ISOnum -->
```

A parameter entity such as `HTMLlat1` that is associated with a file is known as an *external entity*, and the file it includes—which must contain only entity declarations, not declarations for elements or attribute lists—is known as an *entity set*.

Notice that the declaration of the `HTMLlat1` entity is followed by a reference to the entity (`%HTMLlat1;`). This reference is needed because the entity declaration itself simply associates a name (`HTMLlat1` in this case) with a file. The entity reference then causes the file to be imported into the DTD.

Finally, although you can find the XHTML 1.0 Strict DTD at the URL given and from there locate the entity sets that it includes, for human consumption I recommend viewing the marked-up DTD versions found at [W3C-XHTML-DTDS].

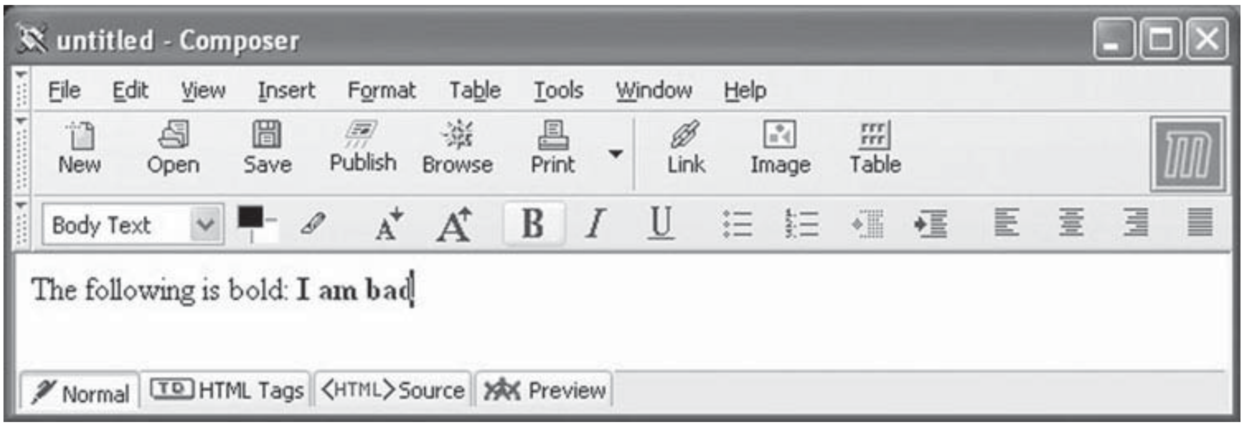
We now turn to the last topic in this chapter: how are HTML documents created in practice?

2.11 Creating HTML Documents

HTML documents can be created in a number of ways. One way is to open a simple text editor, such as Notepad in Microsoft Windows, and simply type in the markup and content of the document. However, this approach is seldom used in practice. A slightly more automated approach is to use an editor such as Emacs that can be customized to provide certain HTML/XML-specific features, such as tag highlighting, "smart" tabbing, automated generation of closing tags, and much more.

Higher-level tools that generate HTML from user input while for the most part hiding the underlying HTML syntax are also available. For example, most current word processors, such as Microsoft Word, allow you to save a document in HTML format (in Word, this may be accomplished by selecting the Web Page file type in the Save As window). So you can create a document using familiar word-processing features and have the word processor automatically convert the text formatting and other markup-type operations you applied to the document into an equivalent HTML document. This approach has the disadvantage that you do not have direct access to (or even much control over) the generated HTML.

Alternatively, several applications are available that provide word-processing-type functionality but also closely integrate access to an HTML representation of the document. For instance, the full Mozilla 1.4 package includes a Composer feature under the Window


untitled - Composer
File Edit View Insert Format Table Tools Window Help
New Open Save Publish Browse Print Link Image Table
Body Text
Normal HTML Tags <HTML> Source Preview

The following is bold: **I am bad**

FIGURE 2.27 Example Composer window after user has entered text and boldfaced some of it.

menu. From a user perspective, Composer appears to be like many other word processors: it has WYSIWYG features for selecting text size and weight, creating bulleted and numbered items, performing text justification, inserting graphics, and so on (Fig. 2.27). But unlike typical word processors, Composer incorporates many HTML-specific features, such as the ability to insert named anchor elements and to directly edit the HTML generated by Composer. Other applications that provide similar or greater functionality include the Microsoft FrontPage® and Macromedia® Dreamweaver® software tools for Web site creation and management.

Given that there are tools for generating HTML from WYSIWYG input, you may wonder why we have covered HTML at all. Why not just use these tools whenever we want an HTML document? There are several answers to this question. One is that knowing HTML is somewhat like knowing assembly language: even if you never use it directly, it can be helpful in your use of other tools. Similarly, sometimes handwritten assembly code is better (by some measure) than compiler-generated code, and the same is sometimes true of handwritten vs. tool-generated HTML. Finally, and probably most importantly, many HTML documents are generated "on the fly" in response to HTTP requests rather than being generated in advance and stored as a file. That is, many HTML documents are produced by programs running on web servers. In later chapters we will learn how to write such programs, and our understanding of HTML will be invaluable to us then.

2.12 Case Study

We'll now apply what we've learned to developing some of the HTML documents for the blogging application described in Section 1.8. The files discussed here are located in the `CaseStudy` subdirectory of the `chapter2` directory of the example files available from the textbook Web site (see the Preface for the address). What we'll be developing could be viewed as an early prototype version of the application: we won't have all of the features a commercial blogging system would provide, instructions will be incomplete, the user interface will be clumsy in places, and so on. That said, the application we develop will be operational and should illustrate a variety of topics.

Let's start with the document displayed when the blogger wants to log in and add an entry to the blog (Fig. 2.28). Notice that the left edges of the text boxes and button are aligned. As shown in Figure 2.29, this effect was produced by using a two-column table as the content of the `form` element, with the text labels in the left column of the table and the form controls in the right column. We'll learn another technique for aligning elements in the next chapter. One technique that you should avoid is using spaces to align elements (such as following the `Password:` label with several `&nbsp;` entity references). The problem with this approach is that, as we'll learn in the next chapter, different browsers may use somewhat different fonts to render the document, which could lead to spaces having different widths among browsers. So what appears aligned in one browser may look jagged in another.

If you view the log-in page in a browser and click the `Log in` button, you will receive an error message. This is because we haven't yet written the server-side software that will process the information entered into the log-in form. So, although the value `Login` of the `action` attribute represents a valid relative URL, the server has not yet been told how to respond to requests to the corresponding absolute URL. We'll cover server-side software development beginning in Chapter 6.

If the server-side software was operational and the blogger logged in successfully, the application next displays an add-entry page (Fig. 2.30). As you might expect, the markup for this document is very similar to that shown in Figure 2.29, so I won't show the markup here (it is in the file `addentry.html` in the case study example downloads). There is, however, one small problem with using the earlier table-in-form technique on this page: by default, browsers center each of the elements in a row of a table vertically. And because the `Body:` label is in the same row as a multi-line text area, while I would like this label to appear immediately beneath the `Title:` label, it instead appears much farther down. We'll learn in the next paragraph how to fix this problem using a style property. The only other noteworthy aspect of this document is that it uses markup such as `&lt;b&gt;` to produce the tags displayed in the instructions at the top of the page. Using the markup `<b>` instead would of course have produced bold text rather than displaying the tag. (Yes, I designed the application to recognize the `b` and `i` elements rather than `strong` and `em`; for an end-user application, the `b` and `i` elements are easily remembered and less likely to be mistyped.)

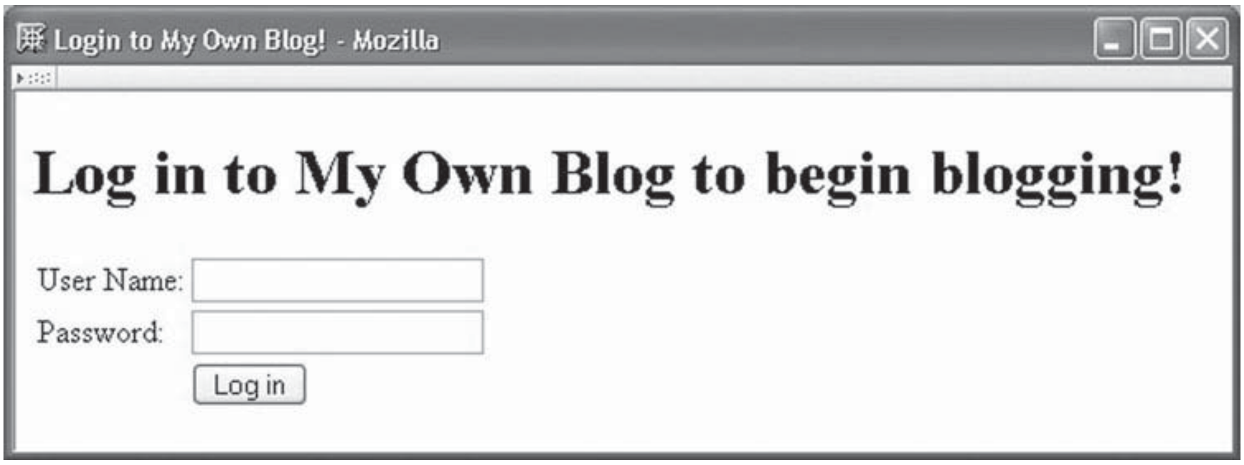

Login to My Own Blog! - Mozilla
Log in to My Own Blog to begin blogging!
User Name:
Password:
Log in

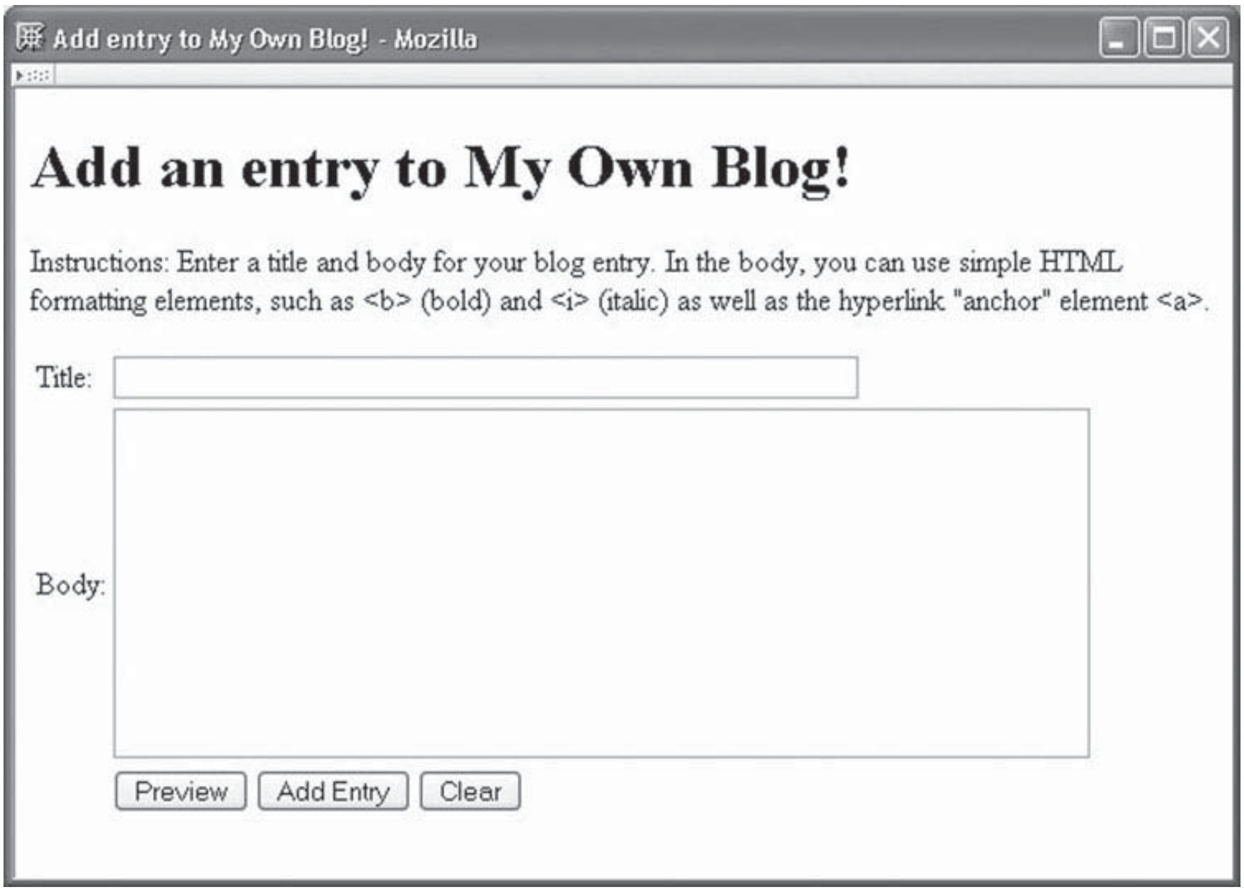
FIGURE 2.28 Log-in screen for blogging application.

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- login.html -->
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Login to My Own Blog!</title>
  </head>
  <body>
    <h1>
      Log in to My Own Blog to begin blogging!
    </h1>
    <form action="Login" method="post">
      <table border="0">
        <tbody>
          <tr>
            <td>User Name:</td>
            <td><input type="text" name="username" /></td>
          </tr>
          <tr>
            <td>Password:</td>
            <td><input type="password" name="pwd" /></td>
          </tr>
          <tr>
            <td>&nbsp;</td>
            <td><input type="submit" name="submit"
                       value="Log in" /></td>
          </tr>
        </tbody>
      </table>
    </form>
  </body>
</html>
```

FIGURE 2.29 HTML document generating the log-in screen for the blogging application.

Finally, let's consider the page that visitors to the blog site will see, the *view-blog* page (Fig. 2.31). Because this is intended to be the page seen by default, I've named it `index.html`. The screen shot shown here is somewhat different than the one in Figure 1.12 in that the latter includes style information, which is covered in the next chapter. The overall structure of the `body` of the HTML markup for this page is shown in Figure 2.32. As shown, the page consists of three basic parts: a banner image at the top, followed by a one-row table split into two elements: a left element containing the blog entries, and a right element containing links (in a fuller implementation, this element might also contain profile information, off-site links, and much more). The `style` attribute is used to ensure that the contents of each of these elements is displayed beginning at the top of the table row, rather than centered. This avoids the problem we saw in the add-entry page with the `Body:` label.

The image is wrapped in a `div` element in order to conform with the XHTML 1.0 DTD. If you look closely at Figure 2.31, you'll see that this image is not centered over the text of the document (it is more noticeable if the window is widened); we'll learn how to center elements in the next chapter.



Add an entry to My Own Blog!

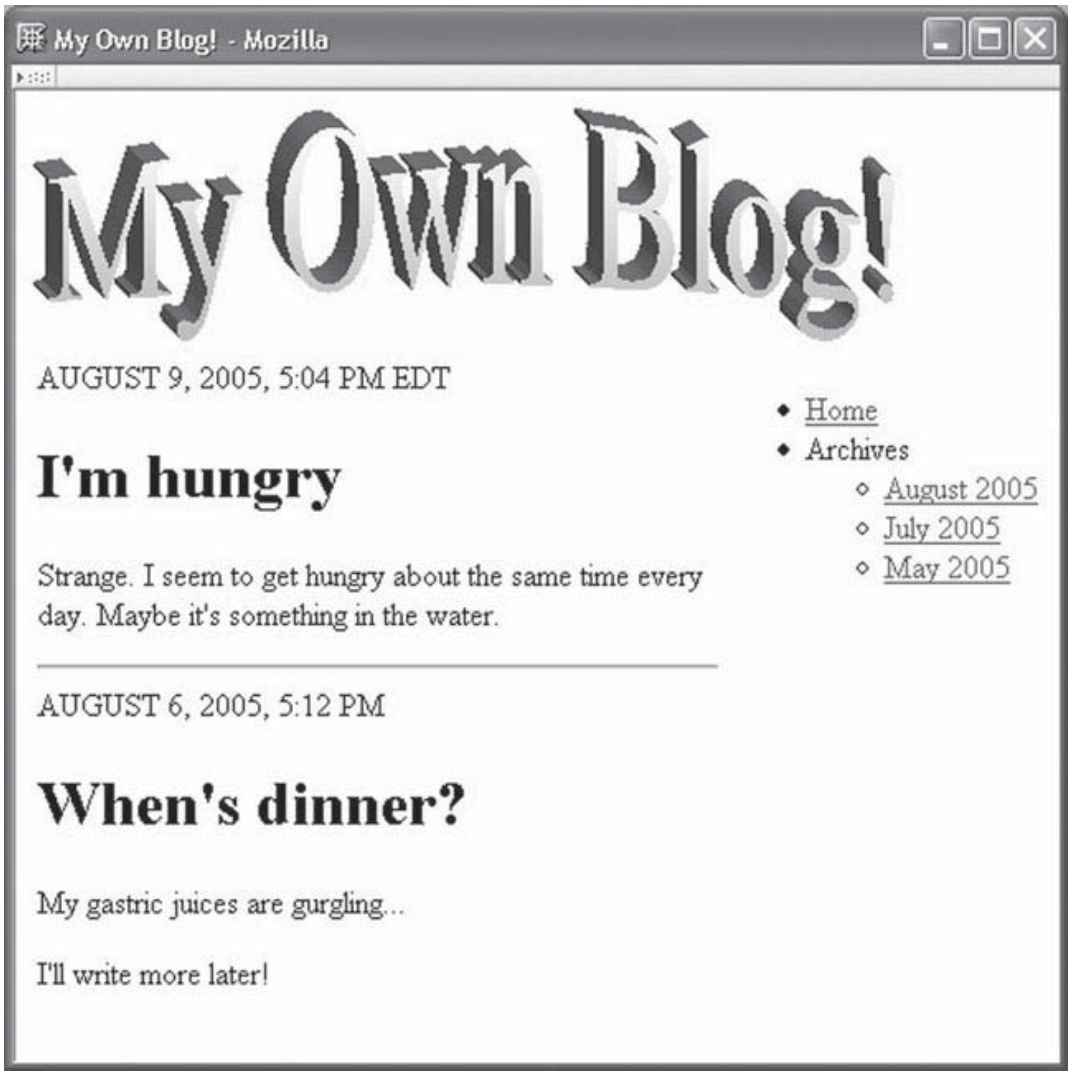
Instructions: Enter a title and body for your blog entry. In the body, you can use simple HTML formatting elements, such as <b> (bold) and <i> (italic) as well as the hyperlink "anchor" element <a>.

FIGURE 2.30 Page for adding blog entries.

Now let's consider the content of the left side of the table. Each blog entry consists of text representing the date and time when the entry was added, an `h1` element containing the title of the entry, and text representing the body of the entry. A horizontal rule (`hr`) is used to separate entries. The first entry is

```
AUGUST 9, 2005, 5:04 PM EDT
<h1>I'm hungry</h1>
<p>
  Strange.  I seem to get hungry about the same time
  every day.  Maybe it's something in the water.
</p>
<hr />
```

The links on the right side of the table are formatted as a nested list, as shown in Figure 2.33. One thing to notice is that the entity reference `&amp;` is used within the values of the last three `href` attributes. I actually want an ampersand (&) within these strings, but since the ampersand is an XML special character I must represent it via a reference. Also notice the use of the `&nbsp;` (nonbreaking space) reference within the text of the hyperlinks. This ensures that each month and year is displayed on a single line. Without this, the browser might split a month and year into separate lines, particularly if the browser window is narrowed.



AUGUST 9, 2005, 5:04 PM EDT

I'm hungry

Strange. I seem to get hungry about the same time every day. Maybe it's something in the water.

AUGUST 6, 2005, 5:12 PM

When's dinner?

My gastric juices are gurgling...

I'll write more later!

- Home
- Archives
  - August 2005
  - July 2005
  - May 2005

FIGURE 2.31 Screen displaying blog entries and navigation links.

```
<div>
  <!-- Banner image -->
  <img src="banner.gif" width="438" height="120"
       alt='"My Own Blog!" Banner' />
</div>
<table border="0">
  <tr>
    <!-- Blog entries -->
    <td style="vertical-align:top">
      ...
    </td>
    <!-- Side information -->
    <td style="vertical-align:top">
      ...
    </td>
  </tr>
</table>
```

FIGURE 2.32 Structure of the body of the HTML document generating the view-blog screen.

```
<ul>
  <li><a href="index.html">Home</a></li>
  <li>Archives
    <ul>
      <li><a href="index.html?month=8&amp;year=2005"
          >August&nbsp;2005</a></li>
      <li><a href="index.html?month=7&amp;year=2005"
          >July&nbsp;2005</a></li>
      <li><a href="index.html?month=5&amp;year=2005"
          >May&nbsp;2005</a></li>
    </ul>
  </li>
</ul>
```

FIGURE 2.33 Side-information markup for the view-blog screen.

The final version of the blogging application will have some additional pages for displaying error information. We'll look at those in later chapters as they're needed.

2.13 References

At this point, you know quite a bit about the basic syntax of HTML documents. You also know about the basic functionality of several elements and attributes, and you know how to use an XHTML DTD to determine how elements can relate to one another and what types of data can be used in attributes. There are still a number of HTML 4.01 elements and attributes that we haven't covered, and in fact most of those won't be covered in this book. But you should now understand HTML well enough to be able to learn about the remaining elements from the HTML language references given here.

The authoritative reference for the XHTML 1.0 language is actually contained in two online documents: [W3C-XHTML-1.0] and [W3C-HTML-4.01]. The latter provides the semantic details of all of the various HTML 4.01/XHTML 1.0 elements (click the Elements link at the top of the page for a complete table of elements with links to details of each). This reference also provides some examples of element usage. However, these are HTML 4.01 examples, and many are not syntactically correct XHTML 1.0, so you should not cut and paste these examples into your own XHTML documents. The former reference carefully explains the syntax of XHTML 1.0 and how this differs from HTML 4.01. A set of annotated DTDs for the XHTML 1.0 language flavors is available at [W3C-XHTML-DTDS]. The specification for relative URLs is contained in [STD-66]. W3C-recommended guidelines for designing web pages for access by users with disabilities are provided at [W3C-WAI].

In order to write real-world HTML applications, you should also be familiar with the idiosyncrasies of various browsers, which may not perfectly implement the XHTML 1.0 recommendation. Covering browser idiosyncrasies in detail is beyond the scope of this book. However, generally speaking, virtually all modern browsers support all of the elements of the HTML 4.01/XHTML 1.0 recommendations (one exception is that Internet Explorer 6 does not support the `abbr` element). So, as far as basic HTML markup is concerned, if you follow the W3C recommendations and the syntax described in this chapter, your markup should be compatible with essentially all browsers. However, beyond basic markup there are some significant differences between browsers; for instance, many differences between Mozilla and IE6 will become evident in Chapter 5. Therefore, you should refer to developer documentation for complete details on each browser. For Internet Explorer, `http://msdn.microsoft.com/workshop/author/dhtml/reference/dhtml_reference_entry.asp` is a good documentation starting point; for Mozilla, `http://www.mozilla.org/docs/web-developer/` has links to documentation.

As discussed earlier, XHTML 1.0 is defined using XML 1.0. This means that the basic syntax of XHTML 1.0 and also the syntax of XML DTDs are defined as part of the XML specification. The authoritative XML 1.0 reference is [W3C-XML-1.0]. This reference makes extensive use of a form of extended BNF notation that is described in the final section of the document ("6 Notation"), which I recommend at least skimming before reading other parts of the document. We'll be covering XML in more detail in Chapter 7, so you shouldn't be concerned about understanding everything in the XML reference at this point.

Exercises

2.1. How many tags are contained in Figure 2.1? How many XHTML elements are contained in the figure? Not counting leading and trailing white space, how many characters of content are contained in the document shown in the figure?

2.2. Draw a complete element tree for the XHTML document shown in Figure 2.12. Assume that the `head` element of the document contains only a `title` element.

2.3. Many web pages are written for private use within a single company. How might developing HTML pages to be used within a company be easier than developing pages to be viewed as part of the World Wide Web?

2.4. Write XHTML markup that will display three paragraphs (each "paragraph" can contain just a few words of text). Write the markup so that there is more vertical space between the second and third paragraphs than there is between the first and second.

2.5. I recommended that two spaces between sentences be produced by using an `&nbsp;` reference followed by a space (Section 2.3.4). Why is this approach to producing two spaces between sentences better than either using a space followed by `&nbsp;` or using two `&nbsp;` references? (Hint: Think about how the sentences will appear as the browser window is resized.) Your answer should not only describe how the sentences might appear, but also explain why.

2.6. Write XHTML markup that assigns the value `"An example is written as <x, b>."` (including the quotes) to the `value` attribute of an `input` control of type `text`. (You may want to test your answer by including your markup in an XHTML document containing a form.)

2.7. Write a short XHTML document that demonstrates the six different heading elements and that also includes normal paragraph text. Compare the fonts used for the various headings with the default font used for paragraph text in Mozilla 1.4.

2.8. What is wrong with the following markup?

```
<pre>
  // Java code example
  if (a < b && c != d) {
    System.out.println("Time for donuts!");
  }
</pre>
```

2.9. If two displays have the same physical size but one has higher resolution than the other, then the same image displayed on both displays with the same width and height in pixels will appear how: smaller on the higher resolution display, larger on the higher resolution display, or the same size on both displays? Explain.

2.10. Using only XHTML as covered in this chapter, explain how you could provide a web page that contains an image that can be distorted by page visitors. Specifically, by manipulating their browsers, users should be able to stretch the image horizontally. The image may appear distorted when it is initially displayed.

2.11. Assume that the base URL for a web page is

```
http://www.example.com/hw1/detail/page7.html
```

Also assume that this page contains the relative URL

```
../images/icon5.gif
```

Give the absolute URL corresponding to this relative URL.

2.12. You are writing an HTML document that will reside in the `App1` subdirectory of the Tomcat `webapps/ROOT` directory. The `App1` directory contains a directory named `legal`. Write a relative URL that could be used within a document contained in `App1` to refer to a document named `copyright.html` in the `legal` directory. Write a second relative URL that could be used within `copyright.html` to refer to an image file named `logo.jpg` in the `App1` directory.

2.13. What is wrong with the following markup?

```
<strong>
  <dl>
    <dt>SGML</dt>
    <dd>Standard Generalized Markup Language</dd>
    <dt>XML</dt>
    <dd>Extensible Markup Language</dd>
  </dl>
</strong>
```

2.14. Rewrite the following example taken from the HTML 4.01 specification so that it complies with the requirements of the XHTML 1.0 standard as discussed in this chapter. The table produced by your version should look exactly the same as the table produced by the original version in a browser compliant with HTML 4.01.

```
<TABLE border="1"
          summary="This table gives some statistics about fruit
                   flies: average height and weight, and percentage
                   with red eyes (for both males and females).">
```

```
<CAPTION><EM>A test table with merged cells</EM></CAPTION>
<TR><TH rowspan="2"><TH colspan="2">Average
    <TH rowspan="2">Red<BR>eyes
<TR><TH>height<TH>weight
<TR><TH>Males<TD>1.9<TD>0.003<TD>40%
<TR><TH>Females<TD>1.7<TD>0.002<TD>43%
</TABLE>
```

2.15. The W3C recommends the use of the `object` HTML element for embedding multimedia—including Java applets—in a web page. However, while both IE6 and Mozilla 1.4 implement the `object` element, the attributes used with this element are generally different in the two browsers. For the most part, IE6 uses the value of the `classid` attribute (which is similar to a URN) to locate the software that will process the multimedia file. In Mozilla 1.4, on the other hand, the `type` attribute is typically used to specify a MIME type for the multimedia file, and browser preferences associate specific player software with the MIME type. Give one potential advantage of each of these approaches over the other.

2.16. The `head` element is declared as follows in the XHTML 1.0 DTD:

```
<!ENTITY % head.misc "(script|style|meta|link|object)*">
<!ELEMENT head (%head.misc;,
     ((title, %head.misc;, (base, %head.misc;)?) |
      (base, %head.misc;, (title, %head.misc;))))>
```

(a) Describe in English what the `head.misc` entity represents.

(b) Describe in English what the content specification of the `head` element represents. For this part of the problem, just translate the content specification directly into English. Also, you can use "`head.misc`" in your description; that is, you do not need to directly mention `script`, `style`, etc., in your description.

(c) Now rewrite your description from the previous part of this problem more succinctly. That is, analyze the content specification and attempt to understand what it means, not just what it says literally. Once you understand the content specification, you should be able to describe it in English reasonably simply.

Research and Exploration

2.17. Review the HTML 2.0 specification (`http://www.w3.org/MarkUp/html-spec/`). What key HTML element discussed in this chapter is missing from this specification? List two `img` attributes covered in this chapter that are missing in this early specification.

2.18. A list of the elements included in XHTML Basic can be found in Sec. 3 of the XHTML Basic recommendation (`http://www.w3.org/TR/xhtml-basic/`). Give two XHTML 1.0 elements discussed in this chapter that are not part of XHTML Basic.

2.19. Create (or download) a copy of the HTML document of Figure 2.1. Then change the document type declaration to indicate that this is an HTML 4.01 document rather than XHTML 1.0, so that it will now be viewed as an SGML document rather than XML. Next, remove as many element tags as possible from the document while producing a document with the following properties:

- The resulting document looks exactly the same as the original when viewed with Mozilla 1.4.

- The document is valid according to the W3C validator [W3C-VAL]. (The validator may warn you that it cannot find a character encoding. The default UTF-8 encoding that it assumes should be fine.)

2.20. Using references to entities supplied by XHTML 1.0, write XHTML markup that will cause a browser to display the following as shown, with italicized text, special characters, and spacing as shown. (This should display properly in a typical installation of Mozilla 1.4, but some of the characters may not display in Internet Explorer browsers.)

(a)

$$\int 2y\ dy$$

(b)

$$A \leq B \Rightarrow B \cap C = \emptyset$$

2.21. Some XHTML 1.0 elements must always have children in an element tree for any valid document. Other elements must always be leaves in an element tree, and still others can be either leaves or nonleaves. Give the content specification from XHTML 1.0 Strict DTD [W3C-XHTML-DTDS] for each of the following XHTML elements, and explain how the specification for each determines which of these three categories applies to the element:

(a) `a`

(b) `tr`

(c) `br`

2.22. Use the XHTML 1.0 Strict DTD [W3C-XHTML-DTDS] to identify at least three XHTML 1.0 Strict elements other than `img` and `form` that have required attributes. Specify the required attribute(s) for each. You can list elements not discussed in the text.

2.23. According to the XHTML 1.0 Strict DTD [W3C-XHTML-DTDS], which of the form controls in Table 2.5 can appear as children of a `form` element?

2.24. In Section 2.4.8, we informally discussed the concepts of inline and block elements. Now we will see exactly what these terms mean as far as the XHTML DTD is concerned.

(a) The XHTML 1.0 Strict DTD [W3C-XHTML-DTDS] defines a parameter entity called `Block` (note the capital `B`; there is also a different entity called `block`). Locate the entity definition of `Block` in the DTD, and rewrite it as an entity definition that contains no references. That is, expand out all entity references contained in the definition of `Block`, in the definitions of those entities, and so on, until no entity references remain. Don't be concerned that you do not recognize some of the elements included in these entity definitions; just write them down whether or not you recognize them (but feel free to look them up in the element list in [W3C-HTML-4.01] if you're curious).

(b) Find at least one element for which a reference to `Block` completely specifies the element's content.

(c) Repeat (a) and (b) for the `Inline` parameter entity (again, for the entity with name beginning with capital `I`).

2.25. The HTML elements recognized by Microsoft's Internet Explorer browser are documented at [MS-DHTML]. Find and briefly describe three elements recognized by IE that are not contained in the HTML 4.01 Transitional standard.

2.26. Read the W3C documentation at [W3C-HTML-4.01] on the `base` element in HTML, and then answer the following question: A collection of web pages has URLs of the form `http://www.example.com/page`*num*`.html`, where *num* is a different number for each page. You want to edit one of the pages, store it in a directory on your machine, and test the modified page. All of the other pages should remain on the web server, not be copied to your machine. The modified page contains several relative URLs of the form `page`*num*`.html`. What one statement would you add to the modified page so that these relative URLs refer to the appropriate pages on the web server?

2.27. The HTML `tabindex` attribute is used to establish a *tab order* among certain elements, primarily form controls such as `input`. That is, the `tabindex` attribute can be used to define the order in which certain elements will be visited as a user presses the Tab key on the keyboard.

(a) To which elements does the `tabindex` apply in XHTML 1.0 Strict? Note: This is not exactly the same set of elements as those listed in [W3C-HTML-4.01]. Instead, you should examine [W3C-XHTML-DTDS].

(b) Provide `tabindex` attribute specifications that will define the tab order for the following markup to be `input` elements named `username` and `password`, in that order; then the submit button; and finally the anchor element:

```
<form action="">
  <fieldset>
    <legend>
      Enter data here
    </legend>
    <label>User name:
      <input type="text" name="username" />
    </label>
    <a href="http://www.example.org/help">I forgot my username/
         password</a>
    <br />
    <label>Password:
      <input type="password" name="password" />
    </label>
    <br />
    <input type="submit" name="login" value="Log me in" />
  </fieldset>
</form>
```

2.28. The HTML `meta` element.

(a) Research the HTML `meta` element, and briefly describe two uses for it (other than the one mentioned in the next part of this question). Give URLs for Web pages that use the element in each way.

(b) Exercise 1.34 in Chapter 1 described the concept of a software *robot*. Read the second part of Section 4.1 of Appendix B of the HTML 4.01 Recommendation [W3C-HTML-4.01], and explain how you would use the `meta` element in an XHTML document in order to request that robots not crawl the document. Note that the examples in [W3C-HTML-4.01] are not valid XHTML. (See `http://www.robotstxt.org/wc/exclusion.html#meta` for more information on this form of robot exclusion.)

2.29. What is the purpose of the HTML `title` attribute (not element)? Describe a scenario in which this attribute might be particularly helpful to a Web site user.

Projects

2.30. HTML reference pages. Implement a subset of the following requirements as specified by your instructor.

- **(a)** Create a set of HTML reference pages for the subset of HTML elements and attributes selected by your instructor. There should first be a page that contains a (short) list of two hyperlinks: one hyperlink to a page with a table of elements, and the other to a page with a table of attributes. The table of elements should contain two columns: the first listing the elements, and the second listing the attributes that are associated with each element. The attributes should be listed one per table row, and each element name should span the rows containing its attributes. Each element and attribute name in the table should be a hyperlink to a detail page describing that element or attribute. Each detail page should use a `dl` style list to define the element or attribute (your description can be short—a sentence or two—but it should be accurate). The table of attributes should be similar, except that the attribute names are in the first column and each attribute name will span a set of rows containing element names associated with the attribute. All pages should have meaningful titles, and tables should include meaningful captions.
- **(b)** Modify the collection of pages in (a) to use frames. Specifically, the user should initially see a page containing three frames: upper left, lower left, and right (much like a Javadoc frames page, such as the one in Fig. 2.6). The upper left frame should contain the two-element list of two hyperlinks to the element and attribute pages. The lower left frame should initially contain the element table, but whenever the user clicks a link in the upper left frame, the lower left frame should display the appropriate table, either element or attribute. When a link is clicked in the lower left frame, the associated page should be displayed in the rightmost frame (initially, this frame should display the detail page for the first element in your element table).
- **(c)** Write all of your pages to the XHTML 1.0 Strict or Frameset standard, as appropriate. Validate your pages at the W3C validator [W3C-VAL], and add a hyperlink with URL `http://validator.w3.org/check/referer` to all Strict pages to make it easy to verify that these pages validate. Note that this link will be able to perform the validation only if the page is viewed by visiting a Web server, and not if the page is available only on your machine or local network.

2.31. Order information form. Implement a subset of the following requirements as specified by your instructor.

- **(a)** Create an HTML document (web page) that gathers information as part of an online product ordering system. This page will request shipping and billing name and address, credit card information, and contact information (e-mail address and phone number). It should provide a menu from which the type of credit card (from a list of approximately four options) can be selected; the default selection should be "Select a Credit Card." There should also be fields for entering the card number and expiration date. Furthermore, there should be a checkbox, initially checked, that is labeled "Please keep me informed about future product offerings." Finally, provide submit and clear buttons. All form controls should have appropriate `name` attributes. Use the GET method for form submission, and specify the empty string for the action.

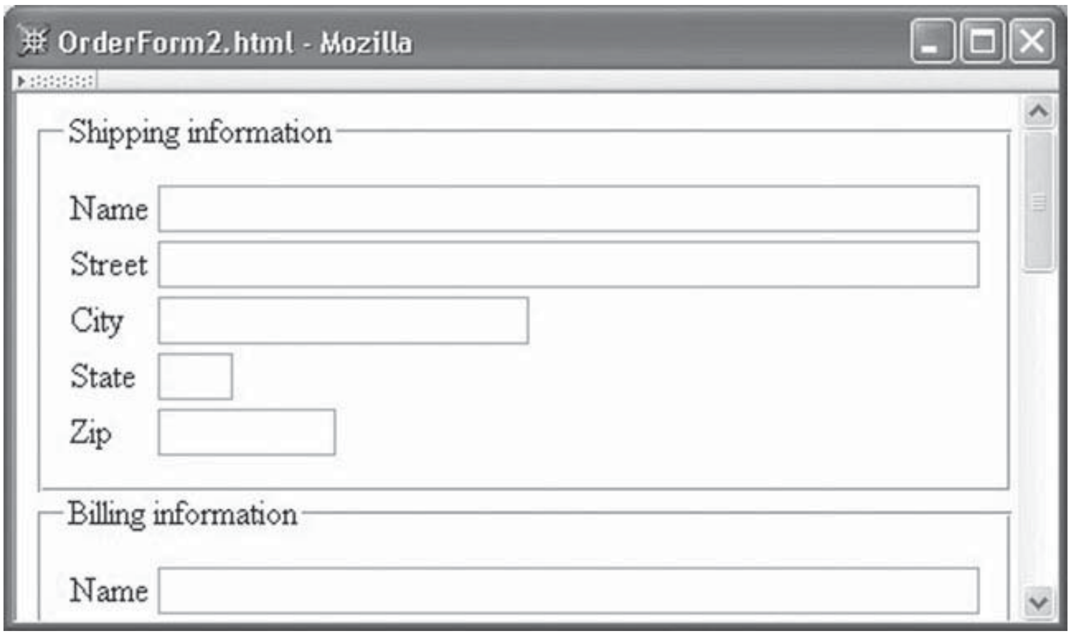


FIGURE 2.34 Example order form layout.

(b) Improve the layout of the form in (a), as illustrated in Figure 2.34. Use fieldset elements to enclose related controls in four groups: shipping, billing, credit card, and contact information (including the checkbox). The submit and clear buttons can be placed outside a fieldset. Within each fieldset, use tables to align the text boxes so that the left edges of the text boxes are all the same distance from the edge of the fieldset, even if the labels of the boxes are of different lengths.

(c) Ensure that your page conforms to the XHTML 1.0 Strict standard. Validate your pages at the W3C validator [W3C-VAL], and add a hyperlink with URL `http://validator.w3.org/check/referer` to all pages to make it easy to verify that your pages validate.

2.32. Entity table generator. Implement a subset of the following requirements as specified by your instructor.

(a) Write a Java program that reads an XHTML entity set and generates an XHTML document that can be used to test a browser's ability to display the entities declared in the entity set. Specifically, the user should enter the name of a file containing an entity set, such as one of the three entity sets imported into the XHTML 1.0 Strict DTD. The program should read the names of all entities declared in the entity set. You may assume for simplicity that each entity declaration begins in the first column of a line and has a single space between the `<!ENTITY` string and the entity name (this assumption is valid for the W3C XHTML 1.0 entity sets). The program should also input the name of a file to which an XHTML document will be written. The generated XHTML document should contain a two-column HTML table with each entity name read from the entity set in the first column and a reference to the entity in the second column.

(b) Rather than reading an entity set from a file, the program should input a URL and obtain the entity set from the given URL. You will probably want to use the `openConnection()` method of the `java.net.URL` class to obtain a `URLConnection` and use the `getInputStream()` method on this `URLConnection object` to obtain an `InputStream` representing the entity set.

(c) Ensure that the generated XHTML document conforms to the XHTML 1.0 Strict standard. Validate your document at the W3C validator [W3C-VAL], and have your program add a hyperlink with URL `http://validator.w3.org/check/referer` to the generated document to make it easy to verify that the document validates.

2.33. Case study extensions. The following questions suggest extensions to the case study of Section 2.12. Implement a subset of the following requirements as specified by your instructor.

(a) Add a profile section to the right element of the view-blog page. The profile should include an image and a list of profile entries, such as name, age, place of birth, place of residence, and occupation. Each entry in the list should consist of a bold label followed by a colon and then the appropriate information.

(b) Design and implement an HTML document that displays a single blog entry followed by comments entered by blog readers. Modify the document of Figure 2.32 so that it contains a hyperlink (with text "Comments") referencing the comments document.

(c) Add a form to the end of the document of part (b). The form should contain appropriate HTML controls allowing a user to enter a title for a comment and separately enter the body of the comment. The controls used to enter this information should be nicely aligned and labeled. Both submit and reset buttons should be provided.

(d) Add a set of five radio buttons to the form of the previous question. The buttons should allow a user to rate a blog entry, giving it from zero to four stars. A checkbox (initially unchecked) should precede the radio button set and be labeled "Rate this blog entry."

(e) Modify the radio buttons of the previous question so that they are "labeled" with appropriate graphics (four empty stars, one filled and three empty stars, etc.) rather than having text labels.

(f) Ensure that your pages conform to the XHTML 1.0 Strict standard. Validate your pages at the W3C validator [W3C-VAL], and add a hyperlink with URL `http://validator.w3.org/check/referer` to the all pages to make it easy to verify that your pages validate.