

A Tutorial on XHTML and XML

Copyright 2016

Dr. Lixin Tao
<http://csis.pace.edu/lixin>
Pace University

Table of Contents

1	Web Pages Using Web Standards	1
1.1	Overview	1
1.2	Web Architecture	2
1.3	HTML Basics.....	4
1.3.1	Tags, Elements and Attributes.....	4
1.3.2	Basic Structure of an HTML File.....	6
1.3.3	Basic HTML Elements	7
1.4	Cascading Style Sheets (CSS).....	18
1.4.1	Style Rule Format.....	20
1.4.2	Box Model.....	20
1.4.3	Formatting Text	22
1.4.4	Formatting a Subset of Element Instances	24
1.4.5	Customizing Hyperlinks with Pseudo-Classes	25
1.4.6	Formatting Part of Text or Document with Span and Div.....	26
1.4.7	Division-Based Layout.....	30
1.4.8	Dynamically Loading HTML Files in iframes.....	32
1.5	HTML Forms and HTTP Basics	34
1.5.1	HTTP Basics.....	34
1.5.2	HTML Forms	36
1.5.3	Common HTML Form Input Controls	40
1.5.4	HTTP GET vs. HTTP POST	43
1.6	Session Data Management	44
1.6.1	Cookies.....	44
1.6.2	Hidden Fields	45
1.6.3	Query Strings.....	45
1.6.4	Server-Side Session Objects.....	45
1.7	Summary	46
1.8	Self-Review Questions.....	46
1.9	Keys to the Self-Review Questions.....	49
1.10	Exercises	49
1.11	Programming Exercises	49
1.12	References.....	50
2	XML – the ‘X’ in Ajax.....	51
2.1	Overview	51
2.2	XML Documents.....	53
2.2.1	XML Declaration.....	54
2.2.2	Unicode Encoding	54
2.2.3	Tags, Elements and Attributes.....	54
2.2.4	Using Special Characters.....	55

2.2.5	Well-Formed XML Documents	56
2.3	DTD	56
2.3.1	Declaring Elements	57
2.3.2	Declaring Attributes	59
2.3.3	Declaring Entity Names	60
2.4	Associating DTD Declarations to XML Documents	61
2.5	XML Schema	62
2.5.1	XML Namespace.....	63
2.5.2	Declaring Simple Elements and Attributes	64
2.5.3	Declaring Complex Elements	65
2.5.4	Controlling Element Order and Repetition.....	68
2.5.5	Referencing XML Schema Specification in an XML Document	68
2.6	XML Parsing and Validation with SAX and DOM	70
2.7	XML Transformation with XSLT	70
2.7.1	Identifying XML Nodes with XPath	71
2.7.2	Transforming XML Documents to XHTML Documents.....	74
2.8	Summary	77
2.9	Self-Review Questions.....	77
2.10	Keys to the Self-Review Questions	79
2.11	Exercises	79
2.12	Programming Exercises	79
2.13	References.....	79
	Index.....	81

1

Web Pages Using Web Standards

- 1.1 Overview
- 1.2 Web Architecture
- 1.3 HTML Basics
- 1.4 Cascading Style Sheets (CSS)
- 1.5 HTML Forms and HTTP Basics
- 1.6 Session Data Management
- 1.7 Summary
- 1.8 Self-Review Questions
- 1.9 Keys to the Self-Review Questions
- 1.10 Exercises
- 1.11 Programming Exercises
- 1.12 References

Objectives of This Chapter

- Introduce fundamental concepts of Web computing
- Introduce XHTML and Cascading Style Sheets
- Introduce how HTTP protocol supports Web browser and Web server interactions
- Introduce different ways of maintaining session data

1.1 Overview

A Web browser is a graphic user interface for a user to interact with various Web applications. A Web browser communicates with the Web servers that host the Web applications over the Internet. A Web browser can send requests to a Web server for data or service. The Web server will reply and send back the response data in a language called HTML, short for *Hypertext Markup Language*. The Web browser can then present the response data to the user following some rendering directives or defaults.

The focus of this chapter is the introduction to the basics of HTML languages. There are several variations of HTML languages in use now. The popular HTML version 4 is more lenient to syntax errors and has limited support for presenting data in various presentation devices like PCs, PDAs or cellular phones. The

2 Web Pages Using Web Standards

XHTML, short for *Extensible Hypertext Markup Language*, rewrites HTML in XML (*Extensible Markup Language*, to be covered in Chapter 3) for better supporting flexible data presentation on different devices. At this time the browser support for XHTML is still limited. This chapter introduces a subset of XHTML that is supported by any Web browser that supports the traditional HTML version 4, and HTML and XHTML are treated as synonyms.

The chapter starts with the introduction of the basic Web architecture underpinning all Web applications. A typical Web application has four tiers: the presentation tier on the client side (Web browsers), the presentation tier on the Web servers, the business logic tier on the application servers, and the database tier on database servers. Different types of servers may co-exist on a single server machine.

HTML (XHTML) basics will be introduced for mainly defining logical data (contents) structures, and *Cascading Style Sheets* or CSS will be introduced as an important mechanism for defining presentation styles of HTML elements. Since XHTML is a special dialect of XML, the HTML introduction in this chapter also serves as the first-iteration introduction to XML discussed in the following chapter.

Web browsers and Web servers communicate through a simple application protocol named HTTP, short for *Hypertext Transfer Protocol*, on top of the TCP/IP network transportation layer. This chapter will explain the HTTP protocol basics and how HTML forms can be used as the main mechanism for submitting user data to a Web server application.

The basic properties of a Web browser will then be outlined. In particular, you will see what cookies are, and how a Web browser exchanges cookies with Web server applications. This chapter will also briefly explain the security sandbox for applets and JavaScripts.

1.2 Web Architecture

A typical web application involves four tiers as depicted in Figure 1: Web browsers on the client side for rendering data presentation coded in HTML, a Web server program that generates data presentation, an application server program that computes business logic, and a database server program that provides data persistency. The three types of server programs may run on the same or different server machines.

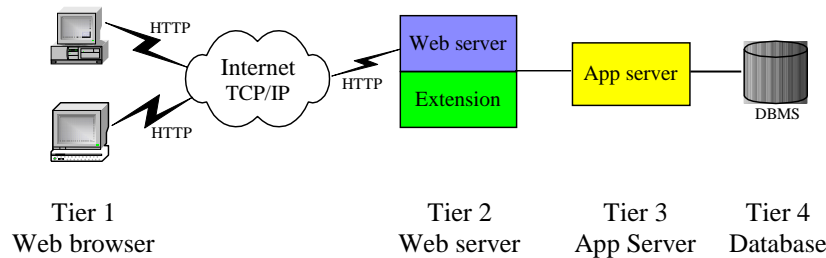


Figure 1 Web architecture

Web browsers can run on most operating systems with limited hardware or software requirement. They are the graphic user interface for the clients to interact with Web applications. The basic functions of a Web browser include:

- Interpret HTML markup and present documents visually;
- Support hyperlinks in HTML documents so the clicking on such a hyperlink can lead to the corresponding HTML file being downloaded from the same or another Web server and presented;
- Use HTML form and HTTP protocol to send requests and data to Web applications and download HTML documents;
- Maintain cookies (name value pairs, explained later in this chapter) deposited on client computers by a Web application and send all cookies back to a Web server if they are deposited by Web applications from the same Web server (cookies will be discussed more later in this chapter);
- Use plug-in applications to support extra functions like playing audio-video files and running Java applets;
- Implement a *Web browser sandbox* security policy: any software component (applets, JavaScripts) running inside a Web browser cannot normally access local clients' resources like data files or keyboards, and can only communicate directly with applications on the Web server from where it is downloaded.

The Web server is mainly for receiving document requests and data submission from Web browsers through the HTTP protocol on top of the Internet's TCP/IP layer. The main function of the Web server is to feed HTML files to the Web browsers. If the client is requesting a static existing file, it will be retrieved on a server hard disk and sent back to the Web browser right away. If the client needs customized HTML pages like the client's bank statement, a software component, like a JSP page or a servlet class (the "Extension" box in Figure 1), needs to retrieve the client's data from the database and compose a response HTML file on-the-fly.

The application server is responsible for computing the business logics of the Web application, like carrying out a bank account balance transfer and computing the shortest route to drive from one city to another. If the business

2 Web Pages Using Web Standards

logic is simple or the Web application is only used by a small group of clients, the application server is usually missing and business logics are computed in the JSP or servlet components of the Web server. But for a popular Web application that generates significant computation load for serving each client, the application server will take advantage of a separate hardware server machine to run business logics more efficiently. This is a good application of the divide-and-conquer problem-solving methodology.

This chapter focuses on the basics of HTML, CSS and HTTP that supports efficient data presentation and browser-server interaction.

1.3 HTML Basics

HTML is a markup language. An HTML document is basically a text document marked up with instructions as to document logical structure and document presentation. There are multiple versions of HTML. While the earlier HTML versions used a more relaxed syntax and focused on more document presentation than document structure, the latest HTML, called XHTML (the *Extensible HyperText Markup Language*), uses the more strict and standard XML (*Extensible Markup Language*, to be covered in the next chapter) syntax to markup text document structures and depends on the separate CSS (*Cascading Style Sheet*) to control the presentation of the document. This separation of document structure and document presentation, even though it is not complete yet, is essential for supporting the same document being rendered by various modern presentation devices including PCs and cell phones that must use very different presentation markups. The HTML concepts and examples in this chapter are based on XHTML v1.0, which is now supported by all the latest Web browsers including Microsoft's *Internet Explorer* v7 and Mozilla's *Firefox* v2.

1.3.1 Tags, Elements and Attributes

An HTML *tag name* is a predefined keyword, like *html*, *body*, *head*, *title*, *p*, *b*, all in lower-case, for describing document structure or presentation.

A tag name is used in the form of a *start tag* or an *end tag*. A start tag is a tag name enclosed in angle brackets < and >, like <html> and <p>. An end tag is the same as the corresponding start tag except it has a forward slash / immediately before the tag name, like </html> and </p>.

An *element* consists of a start tag and a matching end tag based on the same tag name, with optional text or other elements, called *element value*, in between them. The following are some element examples:

```
<p>This is free text</p>
```



```
<p>This element has a nested <b>element</b></p>
```

While the elements can be nested, they cannot be partially nested: the end tag of an element must come after the end tags of all of its nested elements (*first starting last ending*). The following example is not a valid element because it violates the above rule:

```
<p>This is not a valid <b>element<p><b>
```

The *newline* character, the *tab* character and the *space* character are collectively called the *white-space characters*. A sequence of white-space characters acts like a single space for Web browser's data presentation. Therefore, in normal situations, HTML document's formatting is not important (it will not change its presentation in Web browsers) as long as you don't remove all white-space characters between successive words. As a result, the following two html elements are equivalent:

```
<html>
<body>
<p>Sample text</p>
</body>
</html>
```

```
<html><body><p>Sample      text</p></body></html>
```

If an element contains no value, the start tag and the end tag can be combined into a single one as `<tagName />` (the space is for backward compatibility and convention thus not necessary for today's Web browsers; there are some special tags, like `script`, for which such combination cannot be used). Therefore the following two *p* elements are equivalent:

```
<p></p>
```

```
<p />
```

The start tag of an element may contain one or more *attributes*, each in the form "*attributeName*=*attributeValue*". The following is a *p* element with two attributes:

```
<p class="quotation" id="paragraph1">
```

If an attribute value contains quotes, they should be single quote ', as in

```
<p style="font: bold 24px 'Times New Roman', serif">
```

2 Web Pages Using Web Standards

Here we use the “style” attribute to set the font to present the *p* element’s value: boldface, 24 pixels, first choice is font family “Times New Roman”, and the second choice is font family “serif”.

1.3.2 Basic Structure of an HTML File

A basic XHTML v1.0 file that is compatible with HTML v4.0 must start with a “DOCTYPE” declaration for HTML’s root element *html*, followed by a single *html* element. The DOCTYPE declaration specifies a *universal resource identifier* (URI, a unique string for identifying a network resource that may not be an address for accessing the resource), “-//W3C//DTD XHTML 1.0 Transitional//EN”, for the version of HTML used in the current file, as well as a *universal resource location* (URL), “http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd”, for accessing the DTD (*data type definition*, to be introduced in the next chapter) file defining the syntax of the version of HTML used in the current file. Such long strings in this chapter should not be broken by new line characters, even though sometimes we have to break them up in the book samples due to our book page’s limited text width.

Like any XML file, an HTML file can only contain one root element (an element that is not nested inside another element). All the other text and elements must be nested inside this root element. For HTML this root element is *html*. For XHTML v1.0 files, the start tag of an *html* element must have a namespace attribute *xmlns* with value `http://www.w3.org/1999/xhtml`. There are many different specifications of HTML elements, and this attribute specifies a particular specification of HTML that is adopted by XHTML v1.0.

The minimal requirement is that an *html* element must contain exactly one *body* element, which encloses much of document data. An optional *head* element can appear before the *body* element to specify a title of the document to be displayed in the title bar of the Web browser window, and any JavaScript code and CSS style directives, which will be covered later in this chapter.

The following is a sample HTML skeleton that you can use as the starting point of your own HTML files. Be aware that all HTML element and attribute names are in lower-cases, but DOCTYPE must be in upper-cases. All quoted strings in an HTML file, as well as those in XML and program files, must be typed on a single line, even though sometimes we have to break them in our book examples due to the limited page width, as the third line of this HTML skeleton. When we have to print a quoted string value on two lines, we put character ¶ at the end of the first line to indicate that these two lines should be on the same line in HTML files. Our following introduction to HTML features will only use incomplete HTML pieces. To try them out, just copy them in the *body* element of this skeleton and display the resultant file in a Web browser.

```

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>
  Sample Title Shown in Window Title Bar
</title>
</head>
<body>
<p>Sample text</p>
</body>
</html>

```

In this chapter we will introduce many HTML elements in generic terms. For example, we say element *h1* is used to create a large-size heading. Most of the presentation details, like which font is used, in which size, and how the heading is aligned on its line, are not specified. This is because HTML is supposed to specify a document's logical structure, and the document's presentation should be specified by CSS, which will be covered in a later section of this chapter. Each type of Web browser has a default way to present these elements, and we can use CSS style specifications to change the default presentation.

1.3.3 Basic HTML Elements

1.3.3.1 Creating Headings, Paragraphs, Line-Breaks and Formatting Text

HTML supports elements *h1*, *h2*, *h3*, *h4*, *h5* and *h6* to create headings in decreasing font size.

Element *p* is used to create paragraphs. There is extra vertical space between successive paragraphs. White-space characters (*new-line*, *tab* and *space*) are only used to separate successive words, and a sequence of white-space characters is equivalent to a single one. A *new-line* character will not break a line in a Web browser presentation. To break the current line but avoid the extra space introduced by a new paragraph, use a *br* element in form `
`.

Element *b*, like `text`, will present its text in bold.

Element *i*, like `<i>text</i>`, will present its text in italic. Elements *b* and *i* can be nested, as `<i>text</i>`, to present text in bold italic.

Element *tt*, like `<tt>text</tt>`, will present its text in a monospace font.

The text inside a *pre* element will be presented in a monospace font, with all white-space characters preserved. Elements *b* and *i* can be used inside *pre* elements.

2 Web Pages Using Web Standards

The following is an HTML piece using the above elements and its Web browser presentation (copy the HTML piece into the body element of the HTML skeleton file and load the skeleton file in a Web browser).

```
<h2>A Large-Size Heading</h2>
<p>Successive white spaces      are equivalent
to a single one, and the new-line character will
not break the current line in a <b>Web browser</b>.</p>
<p>To break a line without creating a <br />
new paragraph, use element <i>br</i>.</p>
<p>You can introduce a <tt>horizontal line</tt>
with element <i>hr</i>.</p>
<pre>
  To present preformatted text in monospace font,
  use element <i>pre</i>
</pre>
<hr />
```

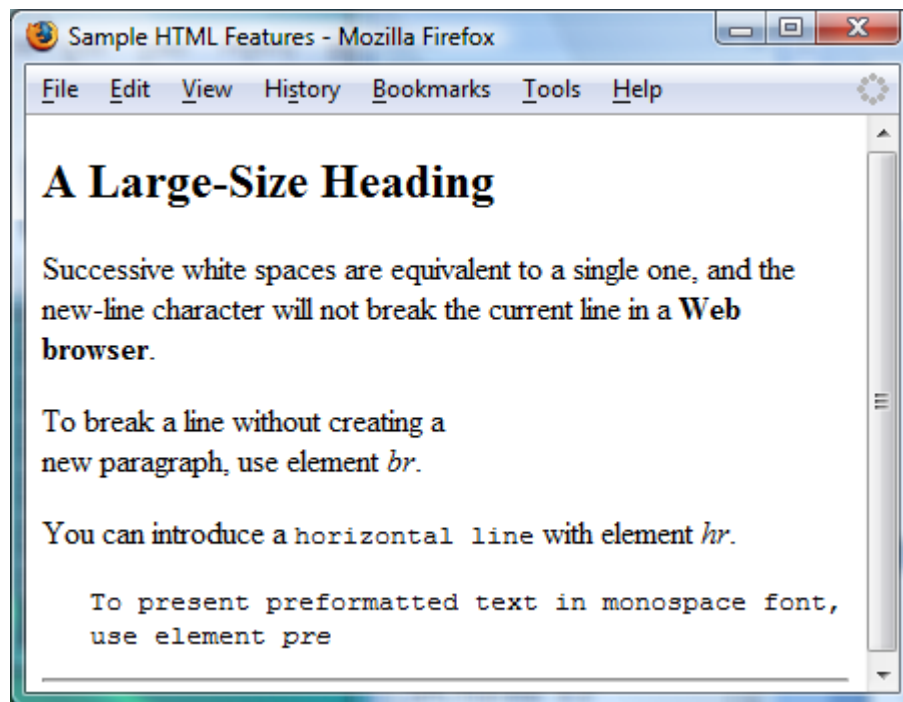
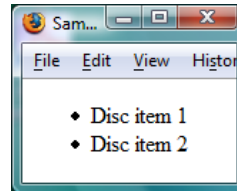


Figure 2 Illustrating elements *h2*, *p*, *br*, *b*, *i*, *tt*, *pre* and *hr*

1.3.3.2 Creating Lists

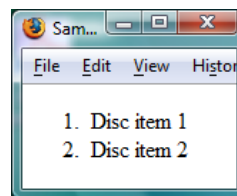
The *ul* (unordered list) elements can be used to create a bullet list, in which each item is a *li* element. The following is an unordered list with two items:

```
<ul>
<li>Disc item 1</li>
<li>Disc item 2</li>
</ul>
```



The *ol* (ordered list) elements can be used to create a numbered list, in which each item is a *li* element. The following is an ordered list with two items:

```
<ol>
<li>Disc item 1</li>
<li>Disc item 2</li>
</ol>
```



The *ul* elements support attribute *style* with values of form “list-style-type: *type*”, where *type* could be *disc* (filled circle, the default), *circle* (unfilled circle), and *square* (filled square).

The *ol* elements support attribute *style* with values of form “list-style-type: *type*”, where *type* could be *decimal* (1, 2, 3, ..., the default), *lower-roman* (i, ii, iii, iv, ...), and *lower-alpha* (a, b, c, ...). The *ol* elements also support attribute *start* for specifying the starting number/letter. For example, the first item of the following ordered list has sequence number 2.

```
<ol start="2"><li>Item 2</li><li>Item 3</li></ol>
```

The *li* elements in an *ol* element can use attribute *value* to specify a sequence number out of order. For example, the second item of the following ordered list has sequence number 3.

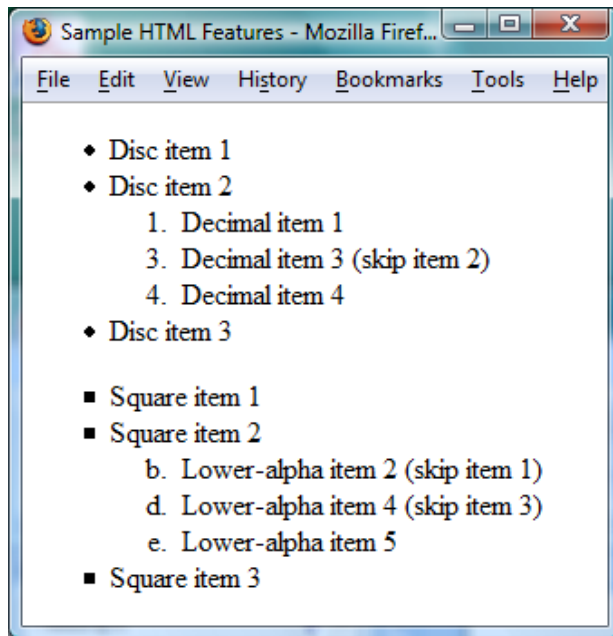
```
<ol><li>Item 1</li><li value="3">Item 3</li></ol>
```

The following is an HTML piece using the above elements and its Web browser presentation (copy the HTML piece into the *body* element of the HTML

2 Web Pages Using Web Standards

skeleton file and load the skeleton file in a Web browser). Make sure that you understand why the Web browser presents this way.

```
<ul>
<li>Disc item 1</li>
<li>Disc item 2
  <ol>
    <li>Decimal item 1</li>
    <li value="3">Decimal item 3 (skip item 2)</li>
    <li>Decimal item 4</li>
  </ol>
<li>Disc item 3</li>
</ul>
<ul style="list-style-type: square">
<li>Square item 1</li>
<li>Square item 2
  <ol style="list-style-type: lower-alpha"
    start="2">
    <li>Lower-alpha item 2 (skip item 1)</li>
    <li value="4">Lower-alpha item 4
      (skip item 3)</li>
    <li>Lower-alpha item 5</li>
  </ol>
<li>Square item 3</li>
</ul>
```



1.3.3.3 Inserting Special Characters

Not all characters have corresponding keys on a computer keyboard. Also, characters `<`, `>` and `&` are meta-characters in HTML and Web browsers will try to interpret them as part of markups so they cannot be part of document text as they are.

As a special case of XML, HTML uses *entities* to specify those special characters. An HTML (XML) entity can be specified with syntax `&code;`, where code could be a predefined entity name or a predefined *Unicode code point* (a positive integer). Only some popular entities have entity names. The following table shows the most useful HTML entity definitions.

Table 1 Popular HTML entities

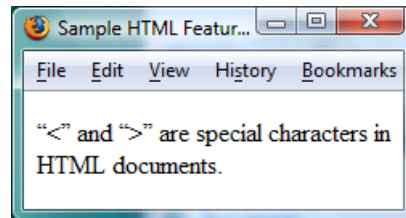
Symbol	Entity with Name	Entity with Code Point
& (ampersand)	<code>&amp;</code>	<code>&#38;</code>
< (less than)	<code>&lt;</code>	<code>&#60;</code>
> (greater than)	<code>&gt;</code>	<code>&#62;</code>
" (straight double quote)	<code>&quot;</code>	<code>&#34;</code>
' (straight single quote)	<code>&apos;</code>	<code>&#39;</code>
(space)		<code>&#32;</code>

2 Web Pages Using Web Standards

(nobreaking space)	 	
(tab)			
© (copyright)	©	©
† (dagger)	†	†
“ (curly double start quote)		“
” (curly double end quote)		”
‘ (curly single start quote)		‘
’ (curly single end quote)		’
. (period)		.

The following is an HTML piece using the above entities and its Web browser presentation.

```
<p>&#147;&lt;&#148; and  
&#147;&gt;&#148; are  
special characters in  
HTML documents&#46;</p>
```



1.3.3.4 Applying Colors

For any HTML element that can contain text as its value, like *body* and *p*, you can apply a foreground color property for rendering its text by assigning value `color: color` to its style attribute, and apply a background color property for the text by assigning value `background-color: color` to its style attribute, where aqua, black, blue, gray, green, lime, navy, red, silver, white and yellow are just a few example predefined color values for color. You can search “HTML color” on the Web to find more HTML color choices, or define your own colors.

If a style attribute specifies more than one properties, the successively properties should be separated by a semi-colon. For example, the following example specifies navy as the *body*’s background color, and blue as its foreground color.

```
<body style="background-color: navy; color: blue">
```


1.3.3.5 Creating Hyperlinks and Anchors

Each Web page on the Internet has a URL (universal resource location) to identify its location. A typical URL has the following format:

```
http://domain-name/application/resource
```

where *domain-name* is a unique name to identify a server computer on the Internet, like *www.amazon.com*; *application* is a server-side folder containing all resources related to an application or service; and *resource* could be the name (or alias or nickname) of an HTML or script/program file residing on a server disk, where the script or program can generate an HTML file on-the-fly based on data submitted by a user. The domain name could be replaced by an IP address, which is four decimal numbers, each between 0 and 255, separated by periods, like 108.168.1.2. Fundamentally all server computers are identified by their unique IP addresses, and the domain names are just nicknames for the IP addresses so they will be easier for people to use. More explanation for URL will be provided in Section 1.5.1 on page 34.

An HTML file can contain hyperlinks to other Web pages so users can click on them to visit different Web pages. A hyperlink has the general structure of `Hyperlink Text`. The Web page linked to by the hyperlink is called the target page of the hyperlink. By default, a Web browser will display a hyperlink's text with an underline, and the hyperlink will have different color based on whether the hyperlink has been visited (clicked) or whether the mouse cursor is hovering on the hyperlink. Later you will learn how to customize the hyperlink views with CSS. For example

```
<a href="http://www.google.com/index.html">Google</a>
```

is a hyperlink to *Google*'s home page. Many Web sites define a "welcome page" so if a user uses an URL for the Web site without the resource name, the welcome page will be returned. Since *Google* has defined "index.html" as its welcome page, the following hyperlink will have the same effect as the previous one:

```
<a href="http://www.google.com">Google</a>
```

The above URLs are also called *absolute paths* for Web pages. An absolute path can be used in any Web page as hyperlink target independent of the page's own URL. If a Web page needs to link to another Web page on the same Web server, say in the same Web server directory, then you can use a shorter *relative path*, which is a path relative to the current page's location. Let us use a scenario to illustrate relative paths. Assume a Web application has three nested directories a/b/c; directory a contains directory b and file a.html; directory b contains

2 Web Pages Using Web Standards

directory `c` and files `b1.html` and `b2.html`; and directory `c` contains file `c.html`. File `b1.html` can use hyperlink `Link A` to link to file `a.html`, where `“../”` represents the parent directory of directory `b`; use hyperlink `Link B` to link to file `b2.html`; and use hyperlink `Link C` to link to file `c.html`. The forward slash `/` used in relative paths are operating system independent.

By default clicking on a hyperlink will lead the target page of the hyperlink to replace the current page in a Web browser. You can also use a `target` attribute in an `a` element to display the target page in a new Web browser instance, as in

```
<a href="http://www.google.com"
  target="_blank">Google</a>
```

You can also use a hyperlink to send emails. You just need to use a URL of form `mailto:email-address`. If a user clicks on the following hyperlink, the user’s default mail application will be started with address `admin@gmail.com` filled in its **To** text field.

```
<a href="mailto:admin@gmail.com">Contact Us</a>
```

You can also specify a subject for the email by using a query string of form `“?subject=Title”` (refer to Section 1.5.1 on page 34 for the definition of query strings). When a user clicks on the following hyperlink, the default mail application will be launched with `admin@gmail.com` in its **To** text field and `Comment` in its **Subject** text field.

```
<a href="mailto:admin@gmail.com?subject=Comment">
Contact Us</a>
```

You can also display a tooltip when a user puts the mouse cursor on top of the hyperlink by using a `title` attribute of an `a` element. When a user puts the mouse cursor on top of the following hyperlink, tooltip `“Comment on the topic”` will be displayed next to the cursor.

```
<a href="mailto:admin@gmail.com?subject=Comment "
  title="Comment on the topic">
```

So far you have been using hyperlinks to link to separate Web pages. But you can also use hyperlinks to link to specific anchors on the same page or other Web pages. When a user clicks on such a hyperlink, the Web browser will jump to display the text close to the anchor. It is a very useful feature for long documents.

An anchor is like a bookmark in an HTML file that can be used as the target of a hyperlink. To define an anchor for word *Conclusion* in an HTML document *test.html*, make *Conclusion* the value of an *a* element as in

```
<a name="conclusion">Conclusion</a>
```

where the value of attribute *name* can be any string. To make a hyperlink to this anchor in the same file, you can use a hyperlink like

```
<a href="#conclusion">View the Conclusion</a>
```

To make a hyperlink to this anchor from another file in the same directory, you can use a hyperlink like

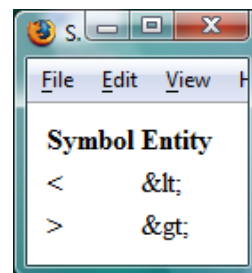
```
<a href="test.html#conclusion">
View the Conclusion</a>
```

1.3.3.6 Creating Tables

Table is a popular format of presenting data. Until the adoption of CSS, tables have also been used to format Web page layout.

A table consists of a few rows, and each row is further divided into a few data fields. In HTML, a *table* element encapsulates all the table rows, a *tr* (table row) element specifies each row, and a *td* (table data) element specifies each data field. A *th* element is similar to a *tr* element except it is used to specify table headers that will be presented in a different style from that for the table data. A table can also have an optional caption created with a *caption* element. The following is a basic table with default properties:

```
<table>
<tr><th>Symbol</th>
    <th>Entity</th>
</tr>
<tr><td>&lt;</td>
    <td>&amp;lt;</td>
</tr>
<tr><td>&gt;</td>
    <td>&amp;gt;</td>
</tr>
</table>
```

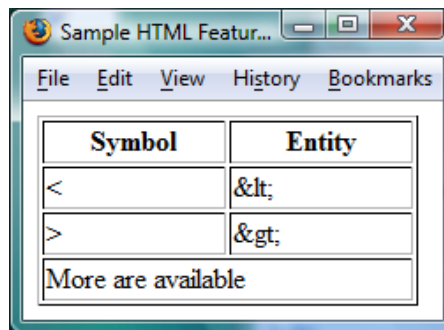


Symbol	Entity
<	<
>	>

2 Web Pages Using Web Standards

By default a table does not have border. You can use the `border` attribute of a `table` element to add a solid border of width 1 pixel (px) by rewriting the start tag as `<table border="1">`. You can use the `width` attribute of a `td` or `th` element to set the width of a column, as in `<th width="100px">`. You can set the width of a table column by setting the width of any single `th` or `td` element in this column. If a data field needs to use two columns and there is a column to its right, you can use a `colspan` attribute of the `td` element to combine the two neighboring data fields, as in `<td colspan="2">`. The following is the above table with the addition of the new features.

```
<table border="1">
<tr><th width="100px">Symbol</th>
    <th width="100px">Entity</th>
</tr>
<tr><td>&lt;</td><td>&amp;lt;</td></tr>
<tr><td>&gt;</td><td>&amp;gt;</td></tr>
<tr><td colspan="2">More are available</td></tr>
</table>
```



At this point you may hope that the text in all the *td* elements be centered. Yes you can do it here, but you need to repeat the text alignment property for each of the five *td* elements, which is a tedious work. Later you will see how you can use CSS to customize table presentation in a more efficient way.

1.3.3.7 Inserting Graphics

Graphics can make a Web page alive and catchy. They are very important for user-friendly Web sites. There are three popular graphic formats for Web page design. *Graphic Interchange Format* (GIF) represents each pixel in 8 bits thus can support only 256 colors. The GIF files are compressed without loss of quality. Many graphics applications can be used to make the background of a GIF file transparent thus easier to mingle with neighboring text, and make a simple

animation by integrating a series images into a single GIF file. GIF file is recommended for images created with graphics applications, like simple icons.

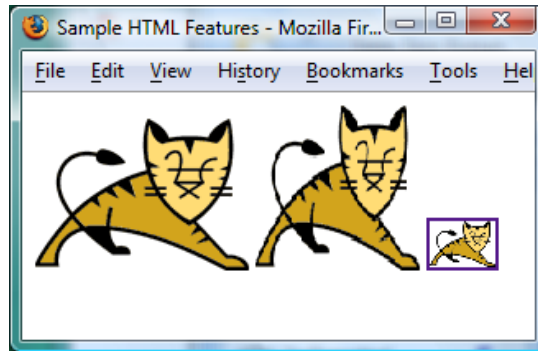
On the other hand, Joint Photographic Experts Group (JPEG, JPG) format represents each pixel with 24 bits thus support up to 1.6 million colors. You can trade-off JPEG file size with image quality: the higher the compression rate, the more loss of precision. JPEG files don't support transparent background or built-in animation, as GIF files do. JPEG files are recommended for images created with cameras.

Portable Network Graphics (PNG) is a new graphics format for combining the advantages of GIF and JPEG as well as overcoming some patent issue with GIF. A PNG image uses 24 or 48 bits to represent a pixel. It supports lossless compression, transparent background and built-in animation. Since more Web browsers are supporting it, PNG is recommended for all new Web graphics.

An *image* element can be used to insert an image in the current Web page location, as in `<image src="tomcat.gif" />`, where attribute `src` is used to specify the image file name. You can also use the `image` element's `width` and `height` attributes to specify the width and height of the image in pixels, and use the `alt` attribute to specify a short text description for the image that will only be presented when the Web browser cannot present the image. Normally you don't specify image width and height at the same time since that may change the original image's aspect ratio, as you see in the second image in the following example. You can also use an image as a hyperlink, as is the case for the third smaller tomcat image in the following example. Here an `a` element's `target` attribute is used to present the target image in a new Web browser window or tab, and its `title` attribute to set a tooltip that will show a message when a mouse cursor is put on the hyperlink image. By default an image embedded in a hyperlink has a border. To remove this border, you can use `image` element's `style` attribute and set its value to "border: none" as in `<image src="tomcat.gif" width="40" style="border: none" />`.

```
<image src="tomcat.gif" />
<image src="tomcat.gif" width="100" height="100"
      alt="Tomcat" />
<a href="tomcat.gif" target="_blank"
  title="Tomcat">
  <image src="tomcat.gif" width="40" />
</a>
```

2 Web Pages Using Web Standards



You can use the style property `float` to flush an image to the left or right, depending on whether you assign value `left` or `right` to `float`. The text will wrap around the image. To move down text vertically until the space occupied by the image becomes “clear”, use style property `clear`. The following example illustrates these features.

```

<h3>Tomcat Web Sever</h3>
<p>Tomcat is an open-source project that supports
servlet container and the basic Web server
functions.</p>
<h3 style="clear: left">Apache Web Server</h3>
<p>Apache is a full-fledge Web server. To enable it
to support servlet/JSP technologies, a Tomcat Web
server is usually integrated to Apache and works
behind it.</p>
```



1.4 Cascading Style Sheets (CSS)

HTML before version 4 uses tags to markup for both logical structure (like the `h1` and `p` elements) and presentation (like the `b` and `i` elements), and lacks the

ability to apply a single directive to format many elements. Starting with HTML version 4, HTML tags are recommended to mainly markup logical structures, and the presentation details will be specified with separate and better structured cascading style sheets.

Cascading style sheets are based on the success of word processor's *style* concept. In word processing, a user can define styles for formatting each type of document elements, and you can format a particular document element instance by simply applying a predefined style.

Each Web browser has a default way to render HTML elements. For example, the HTML standards do not specify the font size of `h1` elements, and the Web browser designers have the freedom to choose a font size to present `h1` elements as long as the font size for `h1` elements is no smaller than that for `h2` elements. Such default behavior of a Web browser can be modified in multiple ways:

A user may use the Web browser's graphic user interface, most likely under the *View* menu, to change some limited aspects of HTML document's presentation. For example, almost all Web browsers allow users to change text size.

An HTML file may import external cascading style sheets using a *link* element inside a *head* element. The following example shows how to import stylesheet entries from a CSS file named "default.css" in the same directory as the HTML file.

```
<head>
<title> ... </title>
<link rel="stylesheet" type="text/css"
      href="default.css" />
<style>
Local CSS definitions go here
</style>
</head>
```

An HTML file may also specify some local style rules within a *style* element, nested inside the *head* element, as shown in the previous example.

Each start tag in the HTML file may also contain a *style* attribute to define style properties for that particular element. You have seen some examples in this category earlier in this chapter.

If an HTML element has some presentation aspects not defined by the HTML standard, the Web browser will search for their potential definitions in reverse order of the previous list, the closest definitions first, and apply the first found definitions. This is the first reason why cascading style sheets are so named. You can always override general style rules with lower-level style definitions.

2 Web Pages Using Web Standards

On another hand, HTML elements are highly nested. A style rule specified for an element will also be applied to elements nested in it unless it is overridden in its child elements. This also suggests the name of cascading style sheets.

The ability for many HTML files to share style definitions in external CSS files is very important. A Web site can change many Web page's presentation by modifying only a single CSS file.

The follow sections will show many CSS definitions. To test them, you can either copy them inside a `style` element or copy them in an external CSS file that is linked to the HTML file with a `link` element, as shown earlier in this section.

1.4.1 Style Rule Format

A style sheet consists of a list of style rules, and most style rules in a CSS sheet are of form

$e_1 e_2 \dots e_k \{ \text{attribute}_1: \text{value}_1; \text{attribute}_2: \text{value}_2; \dots \text{attribute}_n: \text{value}_n \}$

where “ $e_1 e_2 \dots e_k$ ”, called a selector, is a list of space-separated elements, and each of them, except the first one, is nested in the element to its left (later notations based on attributes `id` and `class` will be introduced to represent a subset of elements and they can also be used in the style rule selectors, but the general concept of from general to specific in a selector list is still true). This style rule specifies values to attributes for all e_k elements in the current document that are successively nested in e_{k-1} , ..., e_2 , e_1 . As a simple example,

```
p {border-style: solid; border-width: 2px}
```

specifies that all paragraphs in the current document will have a 2-pixel width solid external border. If you only need to apply this style to a particular paragraph, you can use the `p` element's `style` attribute to specify the same external border:

```
<p style="border-style: solid; border-width: 2px">
```

Note that the attribute value strings must be on the same line in HTML files, even though sometimes they have to be printed on multiple lines in this book. While the following discussions will mainly introduce CSS attributes in the style sheet format, you should be able to follow this example to rewrite them in the form of an element's `style` attribute if necessary.

1.4.2 Box Model

All HTML elements that can contain text value are based on the same box model as shown below. There is a border around the content area, which may be invisible by default. The border has a width and a color. There is a padding area between the border and the content area so that the text will not touch the border. The background color for an element is applied to the area formed by the content and padding. There is also a margin area between the border and the external invisible boundary of the box, which is transparent and mainly used to control the distance between neighboring elements.

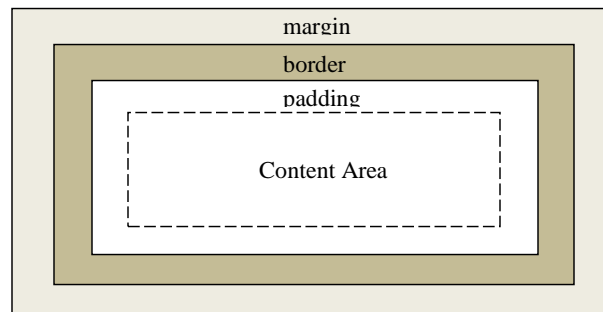


Figure 3 CSS box model

The border style is controlled by attribute `border-style`, which can take on values `solid`, `dotted`, `dashed`, `double`, `groove`, `ridge`, `inset`, `outset` and `none`. The border width is controlled by attribute `border-width`, whose value is the number of pixels. The border color is controlled by attribute `border-color`. The following style rule specifies that the paragraphs will have a solid blue border of width 2 pixels.

```
p {border-style: solid; border-width: 2px;
    border-color: blue}
```

We can use attribute `border` as a shortcut to specify all properties of a border together in the following order: size, color, style. The above example is now rewritten with the `border` attribute.

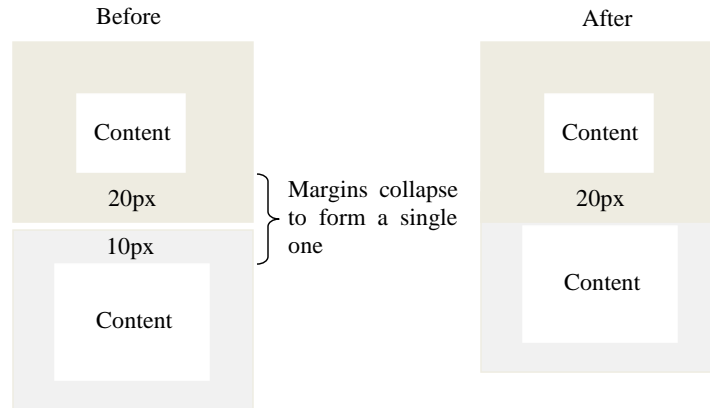
```
p {border: 2px blue solid}
```

Padding and margins are optional and default to zero. You can specify a padding size with attribute `padding`, and a margin size with attribute `margin`. Attributes `color` and `background-color` are used to specify the foreground (text) color and the background color within the border. The following is an example style rule applying these new attributes.

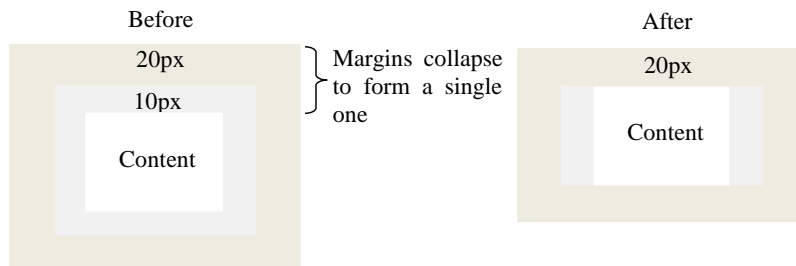
2 Web Pages Using Web Standards

```
p {padding: 20px; margin: 40px;  
background-color: grey;  
border: 2px blue dotted;  
color: white}
```

When two or more vertical margins meet, they will collapse to form a single margin. The height of this margin will equal the height of the larger of the two collapsed margins, as illustrated below.



When one element is contained within another element, assuming there is no padding or border separating margins, their top and/or bottom margins will also collapse together, as illustrated below.



1.4.3 Formatting Text

Specifying a certain font to appear on a page can be tricky because not everyone has the same fonts installed. To work around this problem, you can specify a font family rather than an individual font. A *font family* is a set of fonts listed in order of preference. If the computer displaying your page does not have the first font in

the list, it checks the second, and the third, and so on until it finds a match. The last font on a font family list is normally a font that is guaranteed to be available on any computer. Such generic fonts are specified without using double quotes around them. If a Web browser cannot find any font match, it will use its default font to display the text.

You can use the `font-family` attribute to specify font families. The following are some commonly used font families:

- "Arial Black", "Helvetica Bold"
- "Arial", "Helvetica", sans-serif
- "Verdana", "Geneva", "Arial", "Helvetica", sans-serif
- "Times New Roman", "Times", serif
- "Courier New", "Courier", monospace
- "Georgia", "Times New Roman", "Times", serif
- "Zapf-Chancery", cursive
- "Western", fantasy

If you specify a font family in an element's `style` attribute, the double quotes around the font names should be dropped. The following examples show how to specify a font family in a style rule and in a `style` attribute:

```
p { font-family: "Times New Roman", serif }
```

```
<p style="font-family: Times New Roman, serif">
...</p>
```

Font size can be specified with attribute `font-size`. The commonly used `font-size` values include `small`, `medium` (default), `large`, `12px` (any font size specified in pixel number), `120%` (120% of the base/inherited size).

Attribute `font-style` can be used to specify whether the text should be in normal or in italic style, where `normal` and `italic` are `font-style`'s most popular values.

Attribute `font-weight` can be used to specify the darkness or boldness of the text. Attribute `font-weight`'s popular values include `lighter`, `normal` (default), `bold`, and `bolder`.

The font color is specified by attribute `color`. The background color of text is specified by attribute `background-color`. The popular color values include blue, green, red, yellow, grey, magenta, lime and white. For more color values, make a Web search for "HTML color".

Text alignment can be specified with attribute `text-align`, which can take on values `left`, `right`, `center` and `justify` with the same meaning as they have in word processors.

2 Web Pages Using Web Standards

Attribute `text-indent` can be used to specify the indentation of the first line of a paragraph, as in style rule `p {text-indent: 20px}`.

The line height is the amount of space between each line, which is also referred to as *leading*. You can use attribute `line-height` to specify line height as a percentage of the base one, with popular values 100% (single-spacing), 150% and 200% (double-spacing).

HTML text can be further decorated with lines or blinking effects with attribute `text-decoration`, which supports the following values: `underline` (line under the text), `overline` (line over the text), `line-through` (strikethrough), `blink` (flashing text), and `none` (remove all inherited decoration).

You can also control the extra spacing between successive words with attribute `word-spacing`, and extra spacing between successive letters with attribute `letter-spacing`. By default both `word-spacing` and `letter-spacing` have value 0 pixels. If you specify positive integers, the spacing increases. If you specify negative integers, the spacing decreases. Usually one or two pixels in either direction are plenty. As an example, style rule `p {word-spacing: 1px}` increases the space between successive words by one pixel.

1.4.4 Formatting a Subset of Element Instances

So far you have learned how to apply style rules to all elements of a particular type, say `p`. In practice you need to be able to support exceptions. For example, while specifying that all paragraphs start with an indentation on their first lines, you may also want the first line of the first paragraph to have no text indentation. You may also want paragraphs in different sections of a document to be formatted differently.

If you need to format a unique element instance, say a specific paragraph, of an HTML document differently, you can use attribute `id` to assign a unique string value to this element, and use a special style rule to format this element instance differently. The selector for this style rule is the sharp character `#` followed by the unique `id` string value. For example, the following style rules and HTML body will indent the first line of each paragraph by 20 pixels except for the first paragraph that will have no first-line indentation. Each attribute `id` of an HTML, or XML, document must have unique value in the document, even though several Web browsers are not enforcing this rule.

```
p { text-indent: 20px}
#first {text-indent: 0px}
```

```
<body>
<p id="first">No first-line indentation ...</p>
```

```
<p>With first-line 20 pixel indentation ...</p>
.....
</body>
```

If you need to format a subset of element instances, say all paragraphs in a particular section, of an HTML document differently, you can use attribute `class` to assign a class name to those element instances, and use a special style rule to format these element instances differently. The selector for this style rule is the period character `.` followed by the class name. A document can have many elements carrying the same class value, and these elements may be based on different tag names. The following style rule and HTML body show how you can define a class “important” to present several elements in red color.

```
p {color: black}
.important {color: red}
```

```
<body>
<h2 class="important">Title in Red</h2>
<p class="important">Text in this paragraph will
  be in red</p>
<p>Text in this paragraph will be in black</p>
</body>
```

1.4.5 Customizing Hyperlinks with Pseudo-Classes

In most Web browsers, by default textual hyperlinks appear as underlined blue text, and visited hyperlinks (that is, hyperlinks to pages that you have already visited) appear as underlined purple text. You cannot just change a element’s attributes to customize hyperlink views because hyperlinks may have different views depending on hyperlink’s status. HTML defined several *pseudo-classes*, which are basically modifiers to style sheet selectors to refine the selections, to allow you to customize hyperlink views.

You can use pseudo-classes `a:link` to define color for unvisited hyperlinks, `a:visited` to define color for visited hyperlinks, `a:hover` to define color for hyperlinks when the mouse cursor is close to them, and `a:active` to define color for hyperlinks when the user is clicking on them (*active* is less used since it is hard-to-observe transient colors). The following is an example for customizing hyperlink views. Hyperlinks in HTML files adopting this style sheet will have no underlines. They will be in blue, green, lime, or red depending on whether they are unvisited, visited, having a mouse cursor nearby, or clicked by a mouse cursor.

2 Web Pages Using Web Standards

```
a:link {color: blue; text-decoration: none}
a:visited {color: green}
a:hover {color: lime}
a:active {color: red}
```

1.4.6 Formatting Part of Text or Document with Span and Div

So far you have learned how to format all text in an element differently. Sometime you also need to format a few words in an element differently. You cannot do so at this point because these words may not be all text in an element and your style rules or style attributes can only be applied to all text in an element. The solution is the introduction of a new type of element: *span*. A *span* element itself has no visual effect on text formatting. But like all HTML elements, *span* supports attributes *style*, *id* and *class*, therefore you can use style attributes and style rules to format text in a *span* element differently.

On another hand, you may also want to format all elements in a particular logical section of an HTML document in a special way. You can use a *div* element to enclose all elements in a logical section and assign an *id* or *class* attribute value to identify this division. You can also use the *style* attribute to apply formatting to the entire *div* box. Like *span*, element *div* itself has no visual effects on a Web page's view. It depends on style rules or style attributes to format its contents. While *span* is an inline box, part of a text line, that does not start new lines, *div* usually encloses elements like paragraphs, lists and headers that are separated from elements outside of the *div* element by some vertical space.

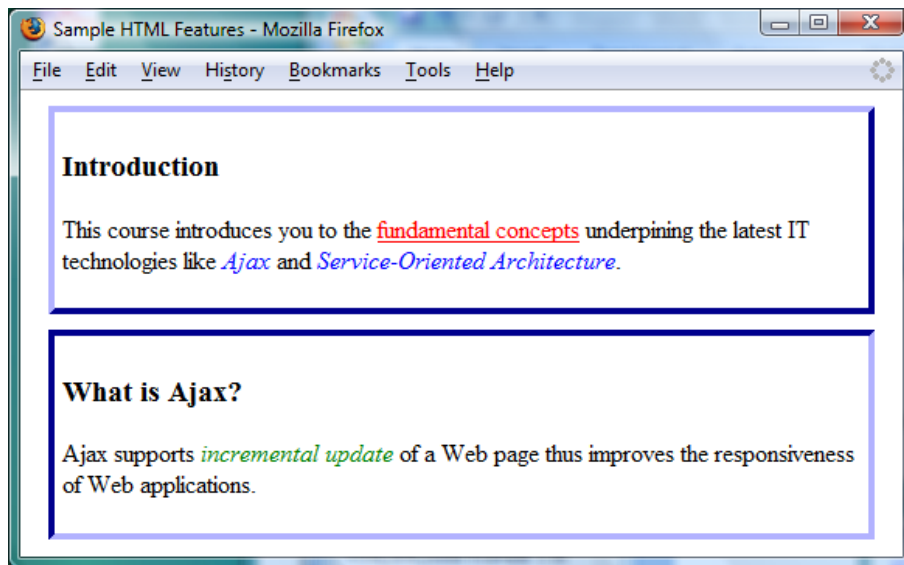
The following example illustrates the HTML features introduced in this subsection. It uses *id* "id1" to display a single term in the document in underlined red. It defines two divisions, and displays each of the divisions in a different boundary box. While all elements of class "keyword" are set to display in italic blue, all elements of class "keyword" inside division "ajax" are set to display in italic green. As explained earlier, a style rule for a specific subset of elements overrides that for a more generic style rule applied to a larger scope containing that subset.

```
<style>
#id1 {color: red; text-decoration: underline}
.keyword {color: blue; font-style: italic}
#ajax .keyword {color: green; font-style: italic}
#intro {border: 4px blue outset;
        margin: 10px; padding: 5px}
#ajax {border: 4px blue inset;
        margin: 10px; padding: 5px}
</style>
```

```

<body>
<div id="intro">
<h3>Introduction</h3>
<p>This course introduces you to the
<span id="id1">fundamental concepts</span>
underpinning the latest IT technologies like
<span class="keyword">Ajax</span> and
<span class="keyword">Service-Oriented
Architecture</span>.
</p>
</div>
<div id="ajax">
<h3>What is Ajax?</h3>
<p>Ajax supports <span class="keyword">incremental
update</span> of a Web page thus improves the
responsiveness of Web applications.</p>
</div>
</body>

```

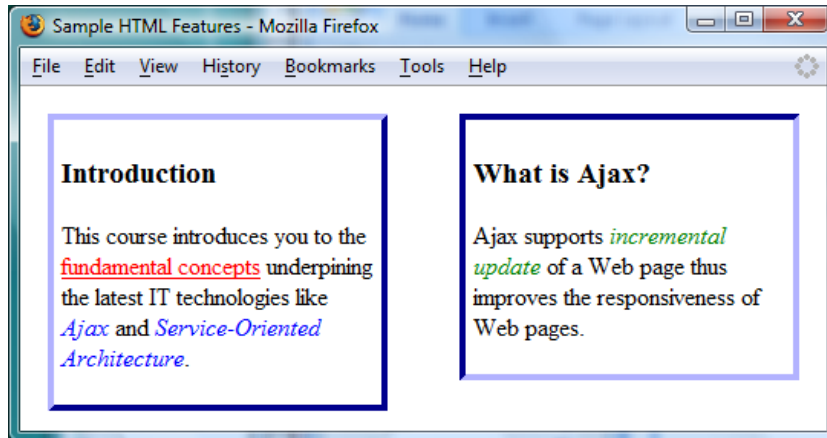


You can use attribute `width` to specify the width of a division, whose value can be in the form of `100px` for 100 pixels or `40%` for 40% of the Web browser window's width. You can use attribute `float` to specify whether the division should float to the left or right, depending on whether you assign to it value `left` or `right`. As an example, if you add the following two style rules to the last

2 Web Pages Using Web Standards

example, the two divisions will be displayed side by side, each taking 40% of the screen width.

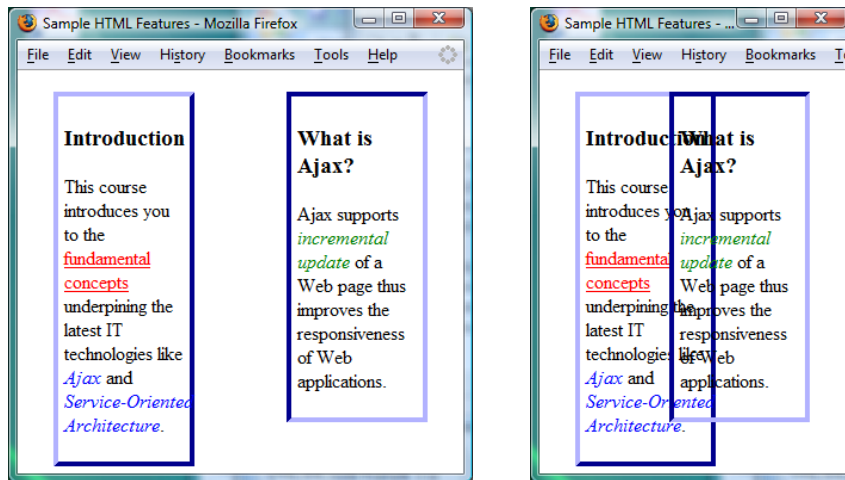
```
#intro {float: left; width: 40%}  
#ajax {float: right; width: 40%}
```



Sometimes you may want to put a division at a specific location of the Web page relative to its parent element, normally the body element. On other occasions you may want to move the division relative to its natural position. Even though both of these two positioning mechanisms may lead to content overlapping, they could be handy for advanced page layout. You can use `div`'s attribute `position` to specify a division's location: if its value is `absolute`, the division's position is relative to its parent element, normally the top of a Web page; if its value is `relative`, the division's position is relative to its natural position. Attribute `position` must be used in conjunction with attributes `left`, `right`, `top`, or `bottom` to specify the specific location. As an example, if you change the style rules for `#intro` and `#ajax` to absolute positioning as the following,

```
#intro {position: absolute; left: 20px;  
        width: 100px}  
#ajax {position: absolute; right: 20px;  
        width: 100px}
```

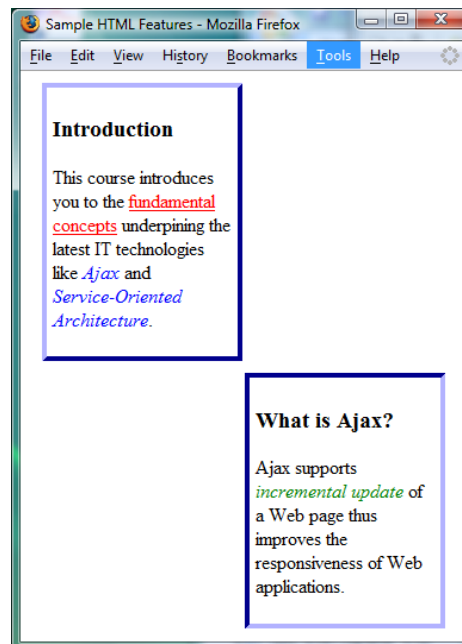
the screen captures below show that the two divisions are 20 pixels from their left and right browser window boundaries respectively, and they may overlap if the browser window is too narrow.



The following example modifies the style rules for #intro and #ajax to use relative positioning as the following:

```
#intro {position: relative; left: 0px;
        width: 150px}
#ajax {position: relative; left: 170px;
        width: 150px}
```

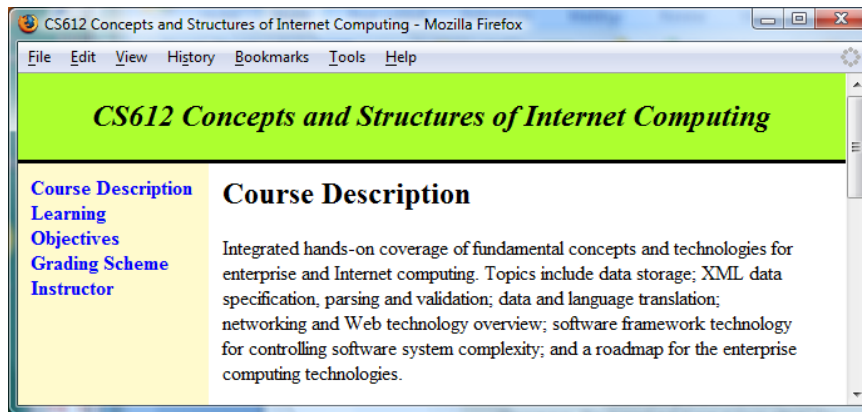
and the screen capture below shows the resulting Web browser display:



2 Web Pages Using Web Standards

1.4.7 Division-Based Layout

As an application for what you have learned about cascading style sheet, we now present an application of CSS in implementing a popular Web page layout that has a banner section in the top pane, a navigation link section in the bottom left pane, and a contents section in the bottom right pane. The following is its screen capture:



The following is the contents of the main HTML file "cssLayout.html". Load this page in a Web browser and you can see the above display. This design uses author's JavaScript file "loadHtml.js" to support dynamic updating of the contents of the bottom right contents pane, one of the essences of the Ajax technology. Element *script* is used to load the external JavaScript file "loadHtml.js". Attribute *onclick* of the *a* element is for specifying some action to be activated when the hyperlink is clicked on. JavaScript function `loadHtml(url, elementID)` is defined in file "loadHtml.js" for downloading the HTML data (normally only text or elements without the *html* or *body* elements, but it can be complete HTML file too) at the specified URL and setting its contents in the HTML element that has the specified *id* value. Bookmark `self` is a dummy one without definition. Its only purpose is to make the *a* elements syntactically correct while not referring to any real Web resources. You can change `self` to any other undefined bookmarks and the example still works.

If you want to see this page in action, you need to download the author's JavaScript file "loadHtml.js" from this book's resource Web site, and create dummy HTML files "cs612objectives.html", "cs612grading.html" and "cs612instructor.html" with any contents. Then you can put all these files in the same directory of a Web server. You can also download a single file "html.war" from the book's resource Web site, download Tomcat Web server from <http://tomcat.apache.org> and install it on your computer, and copy file "html.war" in directory "webapps" of your Tomcat's installation directory. Now

you can test this example by launching your Tomcat Web server and using a Web browser to visit URL <http://localhost:8080/html>.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>CS612 Concepts and Structures of Internet
    Computing</title>
<script language="JavaScript"
    src="loadHtml.js"></script>
<style type="text/css">
body { margin: 0px; }
#header { font: bold 24px "Times New Roman", serif;
    font-style: italic;
    text-align: center;
    padding: 20px;
    border-bottom: 3px solid black;
    background-color: GreenYellow;
    margin-bottom: 0px; }
#contents { float: right;
    padding: 1px 35px 1px 10px;
    width: 70%;
    margin: 0px;
    border: none;
    background-color: White; }
#nav { float: left;
    width: 20%;
    height: 600px;
    margin-top: 0px;
    font-weight: bold;
    padding: 10px;
    border: none;
    background-color: LemonChiffon; }
#nav a { text-decoration: none;
    color: Blue }
#nav a:hover { color: Red }
h2 { margin-top: 10px; }
</style>
</head>
<body>
```

2 Web Pages Using Web Standards

```
<div id="header">CS612 Concepts and Structures of
    Internet Computing</div>
<div id="contents">
<h2>Course Description</h2>
<p>Integrated hands-on coverage of fundamental
concepts and technologies for enterprise and
Internet computing. Topics include data storage;
XML data specification, parsing and validation;
data and language translation; networking and Web
technology overview; software framework technology
for controlling software system complexity; and a
roadmap for the enterprise computing
technologies.</p>
</div>
<div id="nav">
<a href="cssLayout.html">
Course Description</a><br />
<a href="#self" onclick='loadHtml(
"cs612objectives.html", "contents")'>Learning
Objectives</a><br />
<a href="#self" onclick='loadHtml(
"cs612grading.html", "contents")'>Grading Scheme
</a><br />
<a href="#self" onclick='loadHtml(
"cs612instructor.html", "contents")'>Instructor
</a><br />
</div>
</body>
</html>
```

Even though you dynamically loaded data in `div` elements in this example, you can also use the same mechanism to load dynamic data into `span` elements.

1.4.8 Dynamically Loading HTML Files in *iframes*

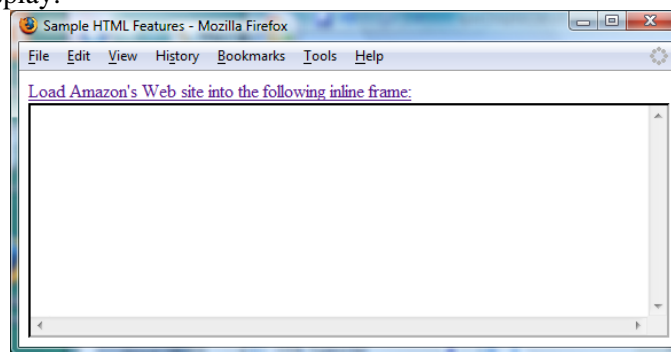
In the last example you saw how to dynamically load HTML text/elements in divisions or spans. Technically, you are only supposed to load text or HTML elements not including *html* or *body* elements. If there are hyperlinks in the loaded HTML data, you cannot follow them to visit other Web pages. The Web browser navigation buttons will not work for the loaded HTML pages either. We can use the newer *iframe* (inline frame) elements of HTML to overcome this limitation. Element *iframe* has attribute *name* that functions like *id* for *div*, attribute *frameborder* with possible values *yes* and *no* for deciding whether a

frame border is needed, attribute `scrolling` with possible values `yes` and `no` for deciding whether the frame should support scrolling, and attributes `width` and `height` for specifying the frame size in pixels. One limitation of `iframe` is its frame size must be hard coded. Some older Web browsers may not support `iframe` well yet. To load a new Web page, say `http://www.amazon.com`, in an `iframe`, we can use JavaScript code `"frames['IFrameName'].location.href='http://www.amazon.com'"` where `IFrameName` represents the name of the `iframe`.

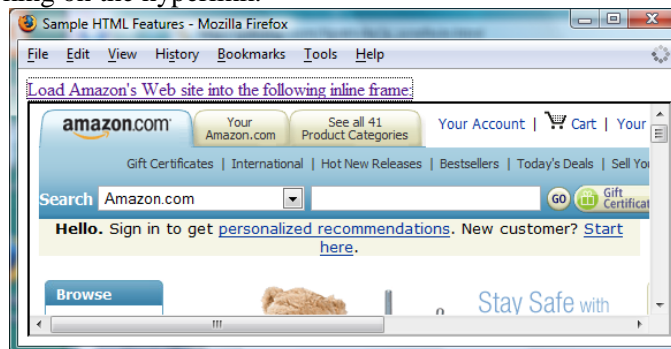
The following is a simple example to show how `iframe` works. The `iframe` start with no contents. When a user clicks on the hyperlink, Amazon's home page is loaded in the `iframe`.

```
<a href="#self" onclick="frames['IFrameName'].location.href='http://www.amazon.com'">Load Amazon's Web site into the following inline frame:</a>
<iframe name="IFrameName" frameborder="yes" width="100%" height="200" scrolling="yes" />
```

Initial display:



After clicking on the hyperlink:



2 Web Pages Using Web Standards

1.5 HTML Forms and HTTP Basics

Web browsers interact with Web servers with a simple application-level protocol called HTTP (HyperText Transfer Protocol), which runs on top of TCP/IP network connections. The main function of HTTP is for Web browsers or programs to download Web pages or any data from Web servers, or submit user data to Web servers. HTML *form* elements can be used to create graphic user interfaces in Web browsers and interact with Web servers through the HTTP protocol.

1.5.1 HTTP Basics

You need to review the concept and general format of a URL (Uniform Resource Locator), first introduced on page 13 when we introduced hyperlinks. A URL is an address for uniquely identifying a Web resource, like a particular Web page, and it has the following general format:

`http://domain-name:port/application/resource?query-string`

where *http* is the protocol for accessing the resource (*https* and *ftp* are popular alternative protocols standing for *secure HTTP* and *File Transfer Protocol*); *domain-name* is for uniquely identifying a server computer on the Internet, like *www.amazon.com*; *port* is an integer between 0 and 65535 for identifying a particular server process; *application* is a server-side folder containing all resources related to a Web site, a Web application or a Web service; *resource* could be the name (alias or nickname) of an HTML or script/program file residing on a server hard disk; and the optional query string passes user data to the Web server.

The domain name is typically in the form of a sequence of three strings separated by periods, like *www.amazon.com*. The right-most is one of the top-level domain names, among which “com” stands for companies, “edu” for education, and “gov” for government. The string to its immediate left is one of its *subdomains*, typically representing a company or an institution. The left-most string is normally an alias for one of the server computers in the company or institution. A server computer may have multiple domain names all referring to the same server computer. For example, *www* in many URLs is optional. But each domain name must refer to no more than one server computer (which may be a façade or interface for a cluster of computers working behind the scene). The domain name could be replaced by an IP address, which is four decimal numbers, each between 0 and 255, separated by periods, like *108.168.1.2*. On Windows you can easily find your computer’s IP address by typing command *ipconfig* in a *Command Prompt* window. Fundamentally each server computer is identified by

one or more IP addresses, and one or more domain names are used as the nicknames for each IP address so they will be easier for people to use. There is a special domain name “localhost” that is normally defined as an alias of local IP address 127.0.0.1. Domain name “localhost” and IP address 127.0.0.1 are for addressing a local computer, very useful for testing Web applications where the Web browser and the Web server are running on the same computer. When a user uses a domain name to specify a URL, the Web browser will use a DNS (domain name server) server on the Internet to translate the domain name into an IP address.

A server computer may run many server applications, like Web servers and database servers, and you may run more than one Apache Web server on the same computer too. A running program is called a *process*. A computer may have many server processes running at the same time, and some of them may be running the same application. When a client sends information or request to this computer, there need a way for the client to specify that the information or request is directed to which server process. The port numbers are used to identify different server processes. Each server process will claim an unused port number and only listen to messages directed to that port number. No two server processes can use the same port number. If you start a server program that uses a port but the port is already in use by another process, the server program will fail to start. Port numbers from 0 to 1024 are reserved for popular server applications and user applications are suggested not to use them. For example, by default the HTTP protocol of Web servers uses port 80, the HTTPS (secure HTTP) protocol uses port 443, the FTP (File Transfer Protocol) protocol uses ports 20 and 21 (for data transfer and FTP command separately), the SSH (Secure Shell) protocol uses port 22, the telnet protocol uses port 23, the DNS (Domain Name Server) protocol uses port 53, and the email IMAP protocol uses port 220. Many server applications allow you to change the port numbers.

One way for a Web browser or client program to submit user data to a Web server is to use query string, which was originally used for sending database query criteria. A query string starts with the question character ? and consists of a sequence of “name=value” (both name and value are strings) assignments separated by character &. Since a valid URL cannot contain some special characters, like space and those with special meanings in HTML, *URL encoding* is used to encode these special characters in the query strings. For example, space is encoded as + or %20, tab as %09, linefeed as %0a, carriage return as %0d, & as %26, ; as %3b, ? as %3f, / as %2f, : as %3a, # as %23, = as %3d, < as %3c, > as %3e, + as %2b, % as %25, " as %22, ' as %27, ~ as %7e, | as %7c, \$ as %24, * as %2a, (as %28,) as %29, and , as %2c. For example, a query string containing names “lang” and “os” with values “Java & C++” and “unix” respectively will be encoded in a URL as “?lang=Java+%26+C%2b%2b&os=unix”.

2 Web Pages Using Web Standards

HTTP is a stateless protocol. Every time a user uses a Web browser or program to interact with a Web server through HTTP, HTTP has no memory of the user's recent interactions with the Web server. For a Web server to remember the recent interactions with a user, it needs to adopt some mechanisms like *cookies* and server-side *session* objects (cookies will be explained later in this chapter, and session objects will be introduced when servlet/JSP technologies are covered) to explicitly record interaction history.

HTTP GET and HTTP POST are the two main HTTP methods for a Web browser or client-side program to interact with a Web server. When you click on a hyperlink in a Web browser, the Web browser will generate an HTTP GET request to the Web server specified by the hyperlink. For a Web browser to interact with a Web server with the HTTP POST method, you need to use an HTML form.

1.5.2 HTML Forms

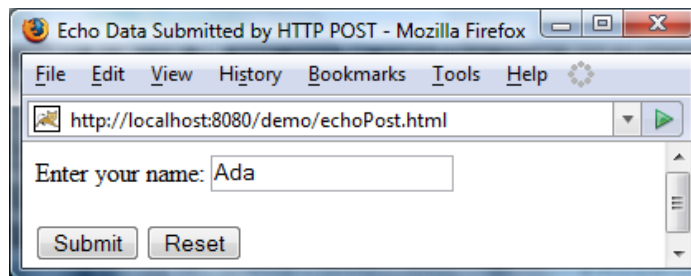
HTML *form* elements are used to create simple graphic user interfaces in a Web browser for the user to interact with a Web server with HTTP GET or HTTP POST methods. The `form` element has two major attributes: `method` for specifying HTTP submission method (with common value `get` or `post`), and `action` for specifying the URL of a Web resource that will accept this HTTP request. The following is an excerpt of example file `echoPost.html` deployed in this book's demo Web application. In this example, HTTP POST is used to submit user data to Web resource `echo` (a Java servlet) inside Web application demo deployed in your local Tomcat Web server.

```
.....
<html xmlns="http://www.w3.org/1999/xhtml">
.....
<body>
  <form method="post"
        action="http://localhost:8080/demo/echo">
    Enter your name: <input type="text"
                          name="user"/> <br/><br/>
    <input type="submit" value="Submit"/>
    <input type="reset" value="Reset"/>
  </form>
</body>
</html>
```


A `form` element can contain free text and most other HTML elements. For each element type that is introduced here for collecting data from a user, it supports a parameter called `name`. This `name` parameter is for specifying a unique variable name representing the data that the user specifies through this input element. The server scripts or programs can use this variable name to access the data that the user has specified through this input element.

In this example, an input element `<input type="text" name="user"/>` is used to create a text field with name “user”. Element `input` can be used to specify several types of *input controls* (devices), and its `type` attribute specifies its particular input control type. Another input element of type “submit”, `<input type="submit" value="Submit"/>`, is used to create a submit button. The `value` attribute here is used to specify the string on top of the button. When a user clicks on a submit button, all data that the user has entered in the form will be submitted to the target Web server resource, as specified by the `action` attribute value of the `form` element, with either HTTP POST or HTTP GET method, as specified by the `method` attribute value of the `form` element. A third input element, `<input type="reset" value="Reset"/>`, is used to specify a reset button with type value “reset”. Its `value` attribute is used to specify the string on top of this reset button. When a user clicks on a reset button, all the data that the user has entered the form will be erased and reset to the form’s initial state so the user can enter the data again from scratch.

Make sure that you have followed our earlier instruction to deploy book resource file “demo.war” in your Tomcat installation, and your local Tomcat Web server is running at its default port 8080. If you load file “<http://localhost:8080/demo/echoPost.html>” into a Web browser, you will see a graphic user interface similar to the following one. Here the user has typed string “Ada” in the text field.



If the user clicks on the submit button now, the Web browser will generate an HTTP POST request to the Web resource “<http://localhost:8080/demo/echo>” specified by the `action` attribute of the `form` element. Basically, a TCP/IP communication channel will be created to connect the Web browser to the

2 Web Pages Using Web Standards

Tomcat Web server running on port 8080; and the following HTTP request text (simplified) will be sent through the TCP/IP channel to the Tomcat Web server.

```
POST    /demo/echo    HTTP/1.1
Accept:  text/html
Accept:  audio/x
User-agent:  Mozilla/5.0
Referer:  http://localhost:8080/demo/echoPost.html
Content-length:  8

user=Ada
```

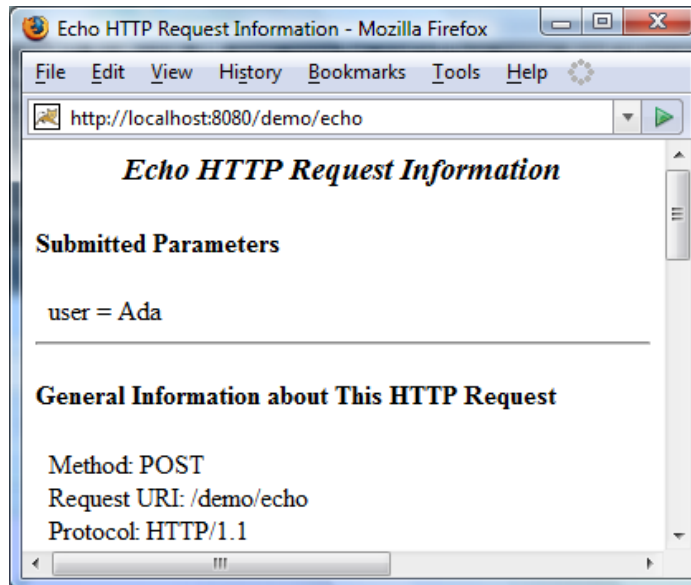
The first line of a HTTP request is used to specify the submission type, GET or POST; the specific Web resource on the Web server for receiving and processing the submitted data; and the latest HTTP version that the Web browser supports. As of 2008, version 1.1 is the latest HTTP specification. The following lines, up to before the blank line, are HTTP *header lines* for declaring Web browser capabilities and extra information for this submission, each of form “name: value”. The first two Accept headers declare that the Web browser can process HTML files and any standard audio file formats from the Web server. The User-agent header declares the software architecture of the Web browser. The Referer header specifies the URL of a Web page from which this HTTP request is generated (this is how online companies like Amazon and Yahoo collect money for advertisements on their Web pages from their sponsors). Any text after the blank line below the header lines is called the *entity body* of the HTTP request, which contains user data submitted through HTTP POST. The Content-length header specifies the exact number of bytes that the entity body contains. If the data is submitted through HTTP GET, the entity body will be empty and the data go to the query string of the submitting URL, as you will see later.

In response to this HTTP POST request, the Tomcat Web server will forward the submitted data to resource `echo` of Web application `demo`, and the resource `echo` will generate dynamically an HTML page for most data it can get from the submission and let Tomcat send the HTML page back to the Web browser as the entity body of the following HTTP response.

```
HTTP/1.1    200    OK
Server:  NCSA/1.3
Mime_version:  1.0
Content_type:  text/html
Content_length:  2000

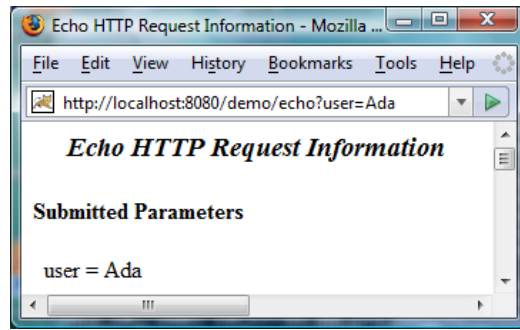
<HTML>
.....
</HTML>
```

The first line of an HTTP response specifies the latest HTTP version that the Web browser supports. The first line also provides a Web server processing status code, the popular values of which include 200 for OK, 400 if the server doesn't understand the request, 404 if the server cannot find the requested page, and 500 for server internal error. The third entry on the first line is a brief message explaining the status code. The first two header lines declare the Web server capabilities and meta-data for the returned data. In this example, the Web server is based on a software architecture named "NCSA/1.3", and it supports *Multipurpose Internet Mail Extension* (MIME) specification v1.0 for Web browsers to submit text or binary data with multi-parts. The last two header lines declare that the entity body contains HTML data with exactly 2000 bytes. The Web browser will parse this HTTP response and present the response data in a window similar to the following one:



Example file `http://localhost:8080/demo/echoGet.html` is the same as `http://localhost:8080/demo/echoPost.html` except the value of `form` attribute `method` has been changed from "post" to "get". If you type "Ada" in its text field and click on the submit button, the submitted data will be in the form of URL query string, as shown below, and the HTTP GET request's entity body will be empty.

2 Web Pages Using Web Standards



1.5.3 Common HTML Form Input Controls

1.5.3.1 Text Field

To generate a **text field** for collecting user input, use an `input` element of form

```
<input type="text" name="variableName"
       value="initial text"
       size="number of characters"/>
```

As an example, the following element

```
<input type="text" name="user"
       value="" size="10"/>
```

will generate a text field that is long enough for showing ten characters (user can type more than ten characters, but only ten of them can be visible at a time) and with its initial value being empty; and the Web server can access the string in this text field through variable “user”.

1.5.3.2 Text Area

If you need to enter more than one line of text, you can use a **text area** element `textarea` to create a large multi-line text entry field supporting horizontal and vertical scroll bars. A `textarea` element is of form

```
<textarea name="variableName" rows="20" cols="80">
Initial text
</textarea>
```

where 20 and 80 are example row size and column size in characters. The user can edit the initial text by default. If the `textarea` is intended to be read only, you can add a “`readonly="readonly"`” attribute specification in the start tag of the `textarea` element.

1.5.3.3 Password Control

If you need to prompt a user to type a password, you can use a **password control** of the same form as the text field control except you replace “type=“text”” with “type=“password””. The only difference between a text field control and a password control is, the password control will use the * character to echo each password character so people cannot read the password over your shoulders. No other security measures are implied by the password control.

1.5.3.4 Select Control

You can use **select** and **option** elements to create a **select control** (list box or dropdown menu) for a user to choose values from multiple pre-specified values. A select control can be specified with the following format:

```
<select name="variableName" size="1"
      multiple="multiple">
  <option value="value1" selected="selected">
    option 1 text
  </option>
  <option value="value2">option 2 text</option>
  .....
</select>
```

This example will generate a select control showing the text of one option a time. If you need to show more options a time, increase the value of the size attribute accordingly. Initially the option with attribute “selected=“selected”” will be selected and its text will be displayed. If none of the options have this selected attribute, the first option will be selected. You can select more than one option by holding down the left *Ctrl* key when you use the mouse to select. If you drop the multiple attribute of the select element, then you can only make one selection a time. When you click on a submission button of the form containing this select element, all the selected values will be submitted to the target Web resource of the form. If an option has the value attribute, its value is the option’s value to be submitted. Otherwise the displayed option element value is submitted.

1.5.3.5 Radio Buttons

Radio Buttons are used when you want the user to select one of a limited number of choices. To generate a set of radio buttons so only one of them can be selected, use the following input element format for each of the radio buttons and use the same name for all of these radio buttons.

```
<input type="radio" name="variableName"
      value="variable value"/>
```

2 Web Pages Using Web Standards

As an example, the following two elements

```
<input type="radio" name="sex" value="male"/> Male  
<br />  
<input type="radio" name="sex" value="female"/>  
Female
```

will generate a set of two radio buttons for a user to specify his/her sex.

1.5.3.6 Checkbox

Checkboxes are used when you want the user to select one or more options of a limited number of choices. Its syntax is the same as that for radio buttons except the `type` attribute will have value “checkbox”. All checkboxes in the same group will have the same variable name, and more than one of them can be selected at the same time. The Web server can use the variable name to retrieve a list of checked values. The following is a checkbox example.

```
Which language do you know? <br />  
English: <input type="checkbox" name="lang"  
          value="English" />  
  
<br />  
French:  <input type="checkbox" name="lang"  
          value="French" />
```

1.5.3.7 Submission Button

To generate a **submission button** for submitting all data that is already in a form, use an `input` element of form

```
<input type="submit" name="variableName"  
       value="button face string"/>
```

As an example, the following element

```
<input type="submit" name="b" value="Finish"/>
```

will generate a submission button labeled “Finish”. If the form contains multiple submission buttons with the same name attribute value but different button face strings, the Web server can check the value of variable “b” to see which submission button has been clicked.

1.5.3.8 Reset Button

To generate a **reset button** for resetting data already in a form so the user can enter data again from scratch, use an `input` element of form

```
<input type="reset" value="button face string"/>
```

As an example, the following element

```
<input type="reset" value="Clear"/>
```

will generate a reset button labeled “Clear”.

1.5.3.9 Hidden Field

Sometimes a Web application can use a **hidden field** to remember some data related to the current user. A hidden field has syntax

```
<input type="hidden" name="variableName"
      value="hiddenValue" />
```

If a Web application includes such a hidden field in a form and sends it to a user’s Web browser, the Web browser will not present the hidden field elements even though the user can see them by reading the source of the Web page. When the user clicks a submission button of the form, all input data in the form, including those in the hidden fields, will be sent back to the Web server so the Web server can remember some information about the user.

In addition to HTML forms, hyperlinks can also generate HTTP GET requests. When a user clicks on a hyperlink in a Web browser, an HTTP GET request is sent to the Web resource the URL of which is specified as the `href` value of the hyperlink. A hyperlink can only send data to the Web resource through query strings hard-coded in it.

1.5.4 HTTP GET vs. HTTP POST

HTTP GET was initially designed for downloading static Web pages from Web servers, and it mainly used short query strings to specify the Web page search criteria. HTTP POST was initially designed for submitting data to Web servers, so it used the request entity body to send data to the Web servers as a data stream, and its response normally depended on the submitted data and the submission status. While both HTTP GET and HTTP POST can send user requests to Web servers and retrieve HTML pages from Web servers for a Web browser to present, they have the following subtle but important differences:

- HTTP GET sends data as query strings so people can read the submitted data over submitter’s shoulders.
- Web servers have limited buffer size, typically 512 bytes, for accommodating query string data. If a user submits more data than that limit, either the data would be truncated, or the Web server would crash, or the submitted data could potentially overwrite some computer code on the server and the server was led to run some hideous code hidden as part of the query string data. The last case is the so-called *buffer overflow*, a common way for hackers to take over the control of a server and spread virus or worms.

2 Web Pages Using Web Standards

- By default Web browsers keep (cache) a copy of the Web page returned by an HTTP GET request so the future requests to the same URL can be avoided and the cached copy could be easily reused. While this can definitely improve the performance if the requested Web page doesn't change, it could be disastrous if the Web page changes or depends on the data submitted by the user.

1.6 Session Data Management

Most Web applications need a user to interact with it multiple times to complete a business transaction. For example, when you shop at Amazon, you choose one book a time by clicking on some HTML form's submission buttons in a Web browser, and Amazon will process your submitted data and send you another HTML form for further shopping. A sequence of related HTTP requests between a Web browser and a Web application for accomplishing a single business transaction is called a *session*. All data specified by the user is called the *session data*. Session data are private so they must be protected from other users. A session normally starts when you first visit a Web site in a particular day, and terminates when you pay off your purchase or shut down your Web browser. Since the HTTP protocol has no memory, Web applications have to use some special mechanisms to securely maintain the user session data.

1.6.1 Cookies

A *cookie* is a pair of name and value, as in (name, value). A Web application can generate multiple cookies, set their life spans in terms of how many milliseconds each of them should be alive, and send them back to a Web browser as part of an HTTP response. If cookies are allowed, a Web browser will save all cookies on its hosting computer, along with their originating URLs and life spans. When an HTTP request is sent from a Web browser of the same type on the same computer to a Web site, all live cookies originated from that Web site will be sent to the Web site as part of the HTTP request. Therefore session data can be stored in cookies. This is the simplest approach to maintain session data. Since the Web server doesn't need to commit any resources for the session data, this is the most scalable approach to support session data of large number of users. But it is not secure or efficient for cookies to go between a Web browser and a Web site for every HTTP request, and hackers could eavesdrop for the session data along the Internet path.

1.6.2 *Hidden Fields*

Some Web users have great concern of the cookie's security implications and they disable cookie support on their Web browsers. A Web application can check the header fields to detect whether cookies are supported by the requesting Web browser. If the cookies are disabled, the Web application will normally use form hidden fields to store session data. Upon receiving submitted data through an HTTP request, the Web application will generate a new HTML form for the user to continue the business transaction, and it will populate all useful session data in the new HTML form as hidden fields. When the user submits the form again, all the data that the user just entered the form, as well as all data saved in the form as hidden fields, will be sent back to the Web application again. Therefore this hidden fields approach for maintaining session data shares most of the advantages and disadvantages of the cookie approach.

1.6.3 *Query Strings*

Sometimes query strings can also be used to maintain small amount of session data. This is particular true for maintaining the short session IDs that will be introduced below. But since most business transactions are implemented with HTML forms, this approach is less useful.

1.6.4 *Server-Side Session Objects*

For improving the security of session data and avoiding the wasted network bandwidth for session data to move back and forth between a Web browser and a Web server, you can also save much of the session data on the Web server as server-side *session objects*. A session object has a unique session ID for identifying a specific user. A session object is normally implemented as a hash table (lookup table) consisting of (name, value) pairs. A single cookie, hidden field of a form, or query string of a hyperlink can be used to maintain the session ID. Since session ID is a fixed size small piece of data, it will not cause much network overhead for going between a Web browser and a Web server for each HTTP request. For securing the session data, you need to make sure that the session ID is unique and properly protected on the client site. Since this approach stores all session data on the Web server, it takes the most server resources and is relatively harder to serve large number of clients concurrently.

2 Web Pages Using Web Standards

1.7 Summary

Web technologies are based on a tiered Web architecture with each tier having its well-defined roles. HTML is the Web language for describing the logical structure of Web documents, and cascading style sheets are for customizing the presentation of the Web documents. HTTP is the application-level protocol to support dynamic interactions between Web browsers and Web servers. In general HTTP POST is a more secure way for a client to interact with Web applications. While there are several ways to maintain client session data, each of them has its pros and cons.

1.8 Self-Review Questions

1. HTML is a language for specifying data presentation in Web browsers.
 - a. True
 - b. False
2. CSS is a language for specifying data presentation in Web browsers.
 - a. True
 - b. False
3. XHTML and HTML are totally different languages.
 - a. True
 - b. False
4. Can users introduce new tags in an XHTML document?
 - a. True
 - b. False
5. Attributes are mainly for specifying large chunks of business data.
 - a. True
 - b. False
6. Which HTML elements are normally used to define the general layout of a Web page?
 - a. table

- b. frame
 - c. form
 - d. div
7. Can you customize hyperlink views without using CSS?
- a. True
 - b. False
8. Multiple elements of an HTML document can have the same value for their `id` attribute.
- a. True
 - b. False
9. Attributes `id` and `class` are for defining special formatting of subsets of elements.
- a. True
 - b. False
10. URL is for specifying the location of a network resource.
- a. True
 - b. False
11. HTTP is a network protocol similar to TCP/IP.
- a. True
 - b. False
12. The port number in a URL is for identifying a server-side process for receiving the HTTP GET/POST request.
- a. True
 - b. False
13. HTTP GET is more secure in submitting large amount of data to a Web server.
- a. True
 - b. False
14. Using HTTP GET to submit data from a text field or text area could lead to

2 Web Pages Using Web Standards

- a. crash of the Web server
 - b. Web server buffer overflow
 - c. viruses or worms being implanted on the Web server and starting to run
 - d. Web browser presenting outdated data
15. HTTP GET should be used to request the price of a stock.
- a. True
 - b. False
16. If you don't want people to read over your shoulders what you have submitted to a Web server, you should use which method to submit the data?
- a. HTTP GET
 - b. HTTP POST
17. When you click on a hyperlink, an HTTP GET request will be sent to the Web server specified by the hyperlink.
- a. True
 - b. False
18. Can you use the space character explicitly in a query string?
- a. True
 - b. False
19. Which mechanisms can help maintain client session data?
- a. Cookies
 - b. Form hidden fields
 - c. URL query strings
 - d. Server-side session data
20. Cookies are client session data stored on Web servers.
- a. True
 - b. False
21. CSS is a language for specifying data presentation in Web browsers.

- a. True
 - b. False
22. If client session data are stored on Web servers, then there is no need to use cookies, form hidden fields or URL query strings to store any data for a client.
- a. True
 - b. False

1.9 Keys to the Self-Review Questions

(To be provided later)

1.10 Exercises

1. What are the main advantages of using CSS to format data presentation relative the old approach of formatting data presentation inside HTML?
2. How do the `id` and `class` attributes of HTML elements support the special data presentations for a subset of elements?
3. What are the major components of a URL and what are their functions?
4. What are the major differences between HTTP GET and HTTP POST for submitting form data to a Web server?
5. What are the pros and cons of using cookies to support session data of a Web application?
6. Consider supporting session data with cookies or with server-side session objects. For maintaining session data for large amount of concurrent users, which approach provides better response time? Which approach is more secure? Which approach takes the least server-side resources?

1.11 Programming Exercises

1. Use XHTML and CSS to create a Web site for the course that adopts this book. The main Web page has three sections: the top banner section for Web site title and some graphics, the narrow bottom-left section for navigation links, and the large bottom-right section for the contents of the

2 Web Pages Using Web Standards

link that the user has chosen on the navigation link section. CSS division-based layout should be used for your solution.

2. Create a Web application on Tomcat that will collect student information from each of its user and echo the user data back for the user to review. You can use this chapter's *demo* Web application as the foundation of your project. Make sure that your graphics user interface uses all common HTML form input controls introduced in Section 1.5.3.

1.12 References

1. Laura Lemay and Rafe Colburn. *Sams Teach Yourself Web Publication with HTML and CSS in One Hour a Day*, Sams, 2006. ISBN 0-672-32886-0.
2. Faithe Wempen. *HTML and XHTML Step by Step*, Microsoft Press, 2006. ISBN 0-7356-2263-9.
3. *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*. <http://www.w3.org/TR/xhtml1/>
4. *XHTML Tutorial*. <http://www.w3schools.com/xhtml/>
5. *HTML Tutorial*. <http://www.w3schools.com/html/>

2

XML – the ‘X’ in Ajax

- 2.1 Overview
- 2.2 XML Documents
- 2.3 DTD
- 2.4 XML Schema
- 2.5 XML Parsing with SAX
- 2.6 XML Parsing with DOM
- 2.7 XML Transformation with XSLT
- 2.8 Summary
- 2.9 Self-Review Questions
- 2.10 Keys to the Self-Review Questions
- 2.11 Exercises
- 2.12 Programming Exercises
- 2.13 References

Objectives of This Chapter

- Introduce the motivation and importance of XML technologies
- Explain the major technologies for defining XML dialects
- Illustrate how to parse, validate and process XML documents
- Introduce XSL and XSLT for transforming XML documents

2.1 Overview

For two information systems to integrate smoothly, they must either adopt the same data structures, or have the ability to interpret their partners’ data accurately. This is the *data integration* challenge, the focus of this chapter.

Different business data types have different logical structures. For example, the hospital patients’ medical records have totally different structure from the bank transaction records. For efficient processing, each business data type must be represented in some well-defined formats. Such data representations are not unique and the different businesses may represent the same type of business data in different data formats. For example, different hospitals may represent their patients’ medical record in different data formats. Such inconsistencies could

lead to difficulties in integrating the information systems of the cooperating businesses.

On another hand, for computers to process and store a type of business data, each information system needs to implement the data format adopted by the business in a particular programming language. Different programming languages may have different data specification mechanisms and data types for specifying the same type of business data. If two cooperating information systems are implemented with different programming languages, they could have difficulties in processing their partners’ data. The properties of computer hardware, operating systems and networking protocols could also add complications to such data integration.

XML, or *Extensible Markup Language*, is a technology introduced mainly for business data specification and integration. XML is a simplified descendant of *SGML*, or *Standard Generalized Markup Language*. Like XHTML/HTML, it uses tags and attributes to mark up data. But XML is generic and doesn’t have a specific application domain. It does not limit what tag or attribute names can be used. For different types of business data you may need to define different concrete markup languages to define their data structures. Each of these concrete markup languages needs to follow the general syntax structure of XML but uses a set of tag and attribute names predefined with XML syntax specification mechanisms like *DTD (Data Type Definition)* or *XML Schema*, which will be introduced in the following sections. In this sense people usually say that XML is a meta-language for defining markup languages in specific application domains, and the latter are called *XML dialects* and can only use predefined tags and attributes. Each XML dialect document is a special case of an XML document, and is called an *instance document* of the XML dialect specification. The popular XML dialects include XHTML for specifying Web page structures, *SOAP* (originally standing for *Simple Object Access Protocol*, and more recently for *Service Oriented Architecture Protocol*) for specifying message structure representing remote method invocations, and *BPEL (Business Process Execution Language)* for specifying business processes.

For a particular type of business data, different information systems may have different specification mechanisms for its logical structure. An XML dialect could be defined and adopted by the cooperating systems and become an intermediate language for data exchange among these systems. For a system to exchange data with its partners, it only needs to have the ability to transform data between its proprietary format and the accepted XML dialect format. Since XML processing functions have been integrated into most operating systems and are freely available, such XML-based data integration is quite cost-effective.

This chapter first introduces the syntax of basic XML documents. DTD (Document Type Definition) and XML Schema mechanisms are then used to define XML dialects for specifying logical data structures. SAX (Simple API for XML) and DOM (Document Object Model) will be used to parse and validate XML documents. The XSL (Extensible Stylesheet Language) and XSLT (XSL

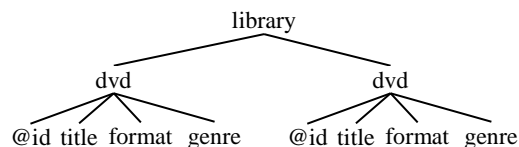
Transformation) techniques will then be introduced to transform XML documents to other data formats.

2.2 XML Documents

An XML document contains an optional *XML declaration* followed by a single top-level element, which may contain nested elements and text, as shown by the following example (the contents are in file “dvd.xml”):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This XML document describes a DVD library -->
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</ genre >
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

This XML documents starts with an optional XML declaration. The second line is an example of *XML comment*, which always starts with `<--` and ends with `-->`. Such comments can occur anywhere and continue over multiple lines in an XML document, and they are ignored by XML processors. The main contents of this example XML document is an element named `library`, which includes three nested elements named `dvd`. Each of the `dvd` elements in turn contains three elements named `title`, `format` and `genre`. Each of the `dvd` elements also contains an attribute `id` which specifies a unique ID number. The nesting structure of such an XML document can be described by the following tree that grows downwards. Here `library` is called the root or top-level element. Prefix `@` is used to indicate that the following name is for an attribute.



2.2.1 XML Declaration

If it is used, the optional XML declaration must be the first line of an XML document. It declares the XML version and character encoding of the following XML document. Different versions of the XML specification have different capabilities and features (backward compatible), and by 2008 the latest XML version is 1.1 and the most popular version is 1.0. If an XML document doesn’t have an XML declaration, the XML processors will assume to be based on some default XML version and character encoding. Since such defaults are not standardized, it is much safer to declare them so the XML processors will process the XML documents with predictable behavior.

2.2.2 Unicode Encoding

XML data are based on *Unicode* (<http://unicode.org>), an industry standard character coding system designed to support the worldwide interchange, processing, and display of the written texts of the diverse languages and technical disciplines of the modern world. The Unicode standard assigns unique integers, called *code points*, to characters of most languages, as well as defines methods for storing the integers as byte sequences in a computer. There are three approaches, named UTF-8, UTF-16, and UTF-32 where UTF stands for Unicode Transformation Format. UTF-8 stores each Unicode character as a sequence of one to four 8-bit values (one byte for the 128 US-ASCII characters, two or three bytes for most of the remainder characters, four bytes for some rarely used characters), and it is the most space-efficient data encoding method if the data is mainly based on the US-ASCII characters, as is the case for English.

2.2.3 Tags, Elements and Attributes

Each *XML element* consists of a *start tag* and an *end tag* with nested elements or text in between. The matching start tag and end tag must be based on the same tag name, which is also called the element name. The nested elements or text between the matching start and end tags are called the *value of the element*. The start tag is of form `<tagName>`, and the end tag is of form `</tagName>`. For example, `<format>Movie</format>` is an element, its start tag is `<format>`, its end tag is `</format>`, the element is based on tag name `format`, so it is also called a `format` element. This `format` element has text `Movie` as its value. Any string consisting of a letter followed by an optional sequence of letters or digits and having no variations of “xml” as its prefix is a valid XML tag name. Tag names are case-sensitive. An element that is not nested in another element is called a root or top-level one. By specification an XML document can have exactly one root element.

If an element has no value, like `<tagName></tagName>`, it can be abbreviated into a more concise form `<tagName />` (where the space before `/` is optional).

Elements can be nested. In the above example, `title`, `format` and `genre` elements are nested inside `dvd` elements, which are in turn nested in a `library` element. Elements cannot partially overlap each other. For example, “`<a>datadata`” contains two partially overlapping `a` and `b` elements thus is not allowed in a valid XML document. For avoiding partial elements overlapping, the element started the first must be ended the last.

The start tag of an element may contain one or more attribute specifications, in the form of a sequence of `attributeName="attributeValue"` separated by white spaces, as in `<dvd id="1">`, where the `dvd` element has attribute `id` with its value being `1`. Attribute values must be enclosed in either matching single straight quotes (`'`) or matching double straight quotes (`"`). Any string consisting of a letter followed by an optional sequence of letters or digits can be a valid attribute name. While most information of an XML document is in the form of element values, attributes are usually used for specifying short categorizing values for the elements.

2.2.4 Using Special Characters

The following five characters are used for identifying XML document structures thus cannot be used in XML data directly: `&`, `<`, `>`, `"`, and `'`. If you need to use them as value of XML elements or attributes, you need to use `&` for `&`, `<` for `<`, `>` for `>`, `"` for `"`, and `'` for `'`. These alternative representations of characters are examples of *entity references*.

As an example, the following XML string is invalid:

```
<Organization>IBM & Microsoft</Organization>
```

Whereas the following is valid XML:

```
<Organization>IBM &amp; Microsoft</Organization>
```

If your keyboard will not allow you to type the characters you want, or if you want to use characters outside the limits of the encoding scheme that you have chosen, you can use a symbolic notation called *entity referencing*. If the character that you need to use has hexadecimal Unicode code point *nnn*, you can use syntax `&#xnnn;` to represent it in XML documents. If the character that you need to use has decimal Unicode code point *nnn*, you can use syntax `&#nnn;` to represent it in XML documents. If you use a special character multiple times in a document, you could define an *entity name* for it in DTD, which will be covered in Section 2.3.3 of this chapter, for easier referencing. An *entity* assigns a string name to an entity reference. For example, if your keyboard has no Euro symbol (€), you can type `€` to represent it in XML documents, where 8364 is the

decimal Unicode code point for the Euro symbol. If you need to use the Euro symbol multiple times in a document, you can define an entity name, say *euro*, through a DTD declaration `<!ENTITY euro "€">` (more explanation will be available in the section on DTD). Then you can use the more meaningful entity reference `€` in your XML document. In general, if there is an entity name *ccc* for a character, you can represent the character with syntax `&ccc;` in XML documents. Entity names “amp”, “lt”, “gt”, “quot” and “apos” are predefined for &, <, >, " and ' and you can use them in your XML documents without declaring them with DTD. Table 1 on page 11 listed the popular HTML entities. The entity references with Unicode code points in the third column can be used in XML documents too, but only the first five entity names are predefined in XML specifications.

2.2.5 Well-Formed XML Documents

A well-formed XML document must conform to the following rules, among others:

- Non-empty elements are delimited by a pair of matching start tag and end tag.
- Empty elements may be in their self-ending tag form, such as `<tagName />`.
- All attribute values are enclosed in matching single (') or double (") quotes.
- Elements may be nested but must not partially overlap. Each non-root element must be completely contained in another element.
- The document complies with its declared or default character encoding.

Both the SAX and DOM XML parsers, which will be introduced in the following sections, will check whether the input XML document is well-formed. If it is not, the parsing process will be terminated with error messages.

2.3 DTD

DTD (Document Type Definition) is the first mechanism for defining XML dialects. As a matter of fact, it is part of the XML specification v1.0 so all XML processors must support it. But DTD itself does not follow the general XML syntax. The following is an example DTD declaration for the earlier DVD XML document example (the contents are in example file “dvd.dtd”):

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT library (dvd+)>
```

```

<!ELEMENT dvd (title, format, type)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT format (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ATTLIST dvd id CDATA #REQUIRED>

```

Since DTD is part of XML specification, it is advised to start its declarations with the XML declaration for XML version and character encoding.

2.3.1 Declaring Elements

To declare a new XML element or tag name (element type) in an XML dialect, use the following syntax:

```
<!ELEMENT elementName (elementContent)>
```

2.3.1.1 Empty elements

Empty elements are declared with the keyword `EMPTY` inside the parentheses:

```
<!ELEMENT elementName (EMPTY)>
```

As an example, `<!ELEMENT br (EMPTY)>` declares that `br` is an empty element.

2.3.1.2 Elements with data

Elements with data are declared with the data type inside parentheses in one of the following forms:

```

<!ELEMENT elementName (#CDATA)>
<!ELEMENT elementName (#PCDATA)>
<!ELEMENT elementName (ANY)>

```

`#CDATA` means character data in which the parser will not try to identify nested elements or entity references.

`#PCDATA` means character data in which the parser will not try to identify nested elements but will try to identify entity references and replace them with their corresponding entity values.

The keyword `ANY` declares an element with any content as its value.

As an example, `<!ELEMENT index (#PCDATA)>` declares a new `index` element type that the XML parsers will further identify and process XML elements nested in values of this type of elements.

2.3.1.3 Elements with children (sequences)

An element with one or more nested children elements as its value are defined with the names of the children elements inside the parentheses:

```
<!ELEMENT elementName (childElementNames)>
```

where `childElementNames` is a sequence of child element names separated by commas. These children must appear in the same sequence in XML documents adopting this DTD declaration. In a full declaration, the child elements must also be declared, and the children can also have children.

As an example, `<!ELEMENT index (term, pages)>` declares a type of elements named `index` whose value contains a `term` element and a `pages` element in the same order.

As another example, `<!ELEMENT footnote (message)>` declares a type of elements named `footnote` that can only contain exactly one `message` element as its value.

For declaring zero or more occurrences of the same element as the value of a new element type, use syntax

```
<!ELEMENT elementName (childName*)>
```

Here symbol `*` indicates that the previous element should occur zero or more times, a notation originally adopted by *regular expressions*.

For example, `<!ELEMENT footnote (message*)>` declares that elements of type `footnote` should contain zero or more occurrences of `message` elements.

If you change the symbol `*` to symbol `+` in the above syntax, then elements of the new element type should have one or more occurrences of the child element.

For declaring zero or one occurrence of an element as the value of a new element type, use syntax

```
<!ELEMENT elementName (childName?)>
```

Here the `?` symbol declares that the previous element can occur zero or one time, also a notation originated from regular expressions.

For example, `<!ELEMENT footnote (message?)>` declares that a `footnote` element should contain either no elements or one `message` element as its value.

If an element can contain alternative elements, you can use the pipe symbol `|` to separate the alternatives. For example, DTD declaration

```
<!ELEMENT section (section1 | section2)?>
```

specifies that a `section` element contains either a `section1` element or a `section2` element, but not both.

2.3.1.4 Declaring mixed content

As an example, look at declaration

```
<!ELEMENT email (to+,from,header,message*,#PCDATA)>
```

The example above declares that the element `email` must contain in the same order at least one `to` child element, exactly one `from` child element, exactly one

header element, zero or more message elements, and some other parsed character data as well.

2.3.2 Declaring Attributes

In DTD, XML element attributes are declared with an `ATTLIST` declaration. An attribute declaration has the following syntax:

```
<!ATTLIST elementName attributeName attributeType
    defaultValue>
```

As you can see from the syntax above, the `ATTLIST` declaration specifies the element which can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value.

The attribute type can have values including the following ones:

Type	Explanation
CDATA	The value is character data
(eval1 eval2 ...)	The value must be one of the enumerated
ID	The value is a name unique in the document
IDREF	The ID value of another element
IDREFS	Space separated list of ID values in the document
ENTITY	The value is an entity

The attribute default value can have the following values:

Value	Explanation
Default-value	The attribute is optional and has this default value
#REQUIRED	The attribute value must be included in the element
#IMPLIED	The attribute is optional with no default value
#FIXED value	The attribute value is fixed to the one specified

DTD example:

```
<!ELEMENT circle EMPTY>
<!ATTLIST circle radius CDATA "1">
```

XML example:

```
<circle radius="10"></circle> <circle />
```

In the above example the element `circle` is defined to be an empty element with the attribute `radius` of type `CDATA`. The `radius` attribute has a default value of 1. The first `circle` element has radius 10, and the second `circle` element has the default radius 1.

If you want to make an attribute optional but you don’t want to provide a default value for it, you can use the special value `#IMPLIED`. In the above example, if you change the attribute declaration to

```
<!ATTLIST circle radius CDATA #IMPLIED>
```

then the second `circle` element above will have no `radius` value.

On the other hand, if you change the above attribute declaration to

```
<!ATTLIST circle radius CDATA #REQUIRED>
```

then the second `circle` element above is not valid and will be rejected by XML validating parsers since it misses a required value for its `radius` attribute.

If you change the above attribute declaration to

```
<!ATTLIST circle radius CDATA #FIXED "10">
```

then all `circle` elements must specify 10 as its `radius` value, and the second `circle` element above is not valid and will be rejected by XML validating parsers since it misses the required value 10 for its `radius` attribute.

The following line declares a `type` attribute for `circle` elements

```
<!ATTLIST circle type (solid|outline) "solid">
```

which can take on either `solid` or `outline` as its value. If a `circle` element doesn’t have a `type` attribute value specified, it would have the default `type` value `solid`.

2.3.3 Declaring Entity Names

An entity name can be declared as a nickname or shortcut for a character or a string. It is mainly used to represent special characters that must be specified with Unicode, or long strings that repeat multiple times in XML documents.

To declare an entity name, use the following syntax:

```
<!ENTITY entityName "entityValue">
```

where *entityName* can be any string consisting of a letter followed by an optional sequence of letters or digits. The following two declarations define `euro` as an entity name for the Euro symbol (€) and `cs` as an entity name for string “Computer Science”.

```
<!ENTITY euro "&#8364;">
<!ENTITY cs "Computer Science">
```

XML documents can use syntax `&entityName;` in its text to represent the character or string associated with *entityName*. For example, if an XML document includes the above two DTD declarations, then `€` and `&cs;` in its text will be read by XML parsers as the same as € and `Computer Science`. Table 1 on page 11 listed the popular HTML entities. The entity numbers in the third column can be used in XML documents too, but only the first five entity names are predefined in XML specifications.

2.4 Associating DTD Declarations to XML Documents

To specify that an XML document is an instance of an XML dialect specified by a set of DTD declarations, you can either include the set of DTD declarations inside the XML document, which is less useful but convenient for teaching purpose; or save the DTD declarations in a separate DTD file and link the XML document to it, which is common practice.

If the DTD declarations are to be included in your XML document, they should be wrapped in a DOCTYPE definition with the following syntax

```
<!DOCTYPE rootElementTag [DTD-Declarations]>
```

and the DOCTYPE definition should be between the XML declaration and the root element of an XML document.

For example, file “dvd_embedded_dtd.xml” has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE library [
<!ELEMENT library (dvd+)>
<!ELEMENT dvd (title, format, genre)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT format (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ATTLIST dvd id CDATA #REQUIRED>
]>
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

To link a DTD declaration file to an XML document, use the following DOCTYPE definition between the XML declaration and the root element:

```
<!DOCTYPE rootElementTag SYSTEM DTD-URL>
```

where DTD-URL can be either a file system path for the DTD file on the local file system, or a URL for the DTD file deployed on the Internet.

The following contents of file *dvd_dtd.xml* shows how it links to the DTD file *dvd.dtd* next to it in the local file system.

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

```

<!DOCTYPE library SYSTEM "dvd.dtd">
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>

```

2.5 XML Schema

While DTD is part of XML specification and supported by any XML processors, it is weak in its expressiveness for defining complex data structures. *XML Schema* (<http://www.w3.org/XML/Schema>) is an alternative industry standard for defining XML dialects. XML Schema itself is an XML dialect, thus it can take advantage of many existing XML techniques and processors. It also has a much more detailed way to define what the data can and cannot contain, and promotes declaration reuse so common declarations can be factored out and referenced by multiple element or attribute declarations.

The following is an example XML Schema declaration for the earlier XML DVD dialect; and file *dvd.xsd* contains this declaration.

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="dvd" minOccurs="0"
          maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title"
                type="xs:string"/>
              <xs:element name="format"
                type="xs:string"/>
              <xs:element name="genre"

```

```

        type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id"
        type="xs:integer" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

2.5.1 XML Namespace

For convenient usage, XML element and attribute names are supposed to be short and meaningful. Therefore XML dialects declared by different people or companies have the tendency of adopting the same names. If an XML document uses elements from more than one of these dialects, then naming conflict may happen. XML namespace is introduced to avoid XML name conflicts.

A set of XML elements, attributes and data types can be associated with a *namespace*, which could be any *unique string*. For easier to be unique, namespaces are normally related to the declaring company or institution's URL. For example "http://www.w3.org/2001/XMLSchema" is the namespace for the 2001 version of XML Schema, and "http://www.w3.org/1999/xhtml" is the namespace for *XHTML 1.0 Transitional*, as specified in all the XHTML examples in the previous chapter. String "http://csis.pace.edu" could be another example namespace for XML Schemas declared by *Pace University's School of Computer Science and Information Systems*. While namespaces normally look like URLs, they don't need be. There are usually no Web resources corresponding to namespaces.

To specify XML elements, attributes or data types of a namespace in an XML document, they are supposed to be qualified by their namespace. Since namespaces are normally long to be unique, *namespace prefixes*, which are normally one to four characters long, could be declared to represent the full namespaces. Each XML document can choose its own namespace prefixes. In the above example, "xs" is an XML prefix representing namespace "http://www.w3.org/2001/XMLSchema". The association between a namespace prefix and a namespace is specified in the opening tag of the root element in the form of attribute specification, except the namespace prefix has its own prefix "xmlns:". For example, to specify that "xs" is the namespace prefix for namespace "http://www.w3.org/2001/XMLSchema", the following two lines in the example XML document are used:

```

<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

```

In an XML document, if an element is qualified by its namespace prefix, as “`xs:element`” for element `element` declared in namespace “`http://www.w3.org/2001/XMLSchema`”, its attributes and nested elements by default belong to the same namespace.

If an XML document uses several namespaces, but the majority of the elements and attributes use the same namespace, you can use attribute `xmlns` to declare the default namespace in the start tag of the root element so that those elements, attributes or data types not qualified by namespace prefixes will be assumed to belong to this default namespace. As an example, if an XML document has the following attribute declaration in the start tag of its root element,

```
xmlns="http://csis.pace.edu"
```

then all unqualified elements and attributes in this document are supposed to belong to namespace “`http://csis.pace.edu`”.

If an XML Schema document declares an XML dialect belonging to a particular namespace, its root element should contain a `targetNamespace` attribute to specify the target namespace for elements, attributes and data types declared in this dialect. As an example, if an XML Schema document’s root element includes attribute `targetNamespace` like

```
<xs:schema targetNamespace="http://csis.pace.edu"
.....>
```

then all elements, attributes and data types declared in this document belong to namespace “`http://csis.pace.edu`”. Example file *dvd-ns.xsd* contains the same contents as file *dvd.xsd* but it declares all elements and attributes under namespace “`http://csis.pace.edu`”. Without such a `targetNamespace` attribute in the XML Schema root element, the XML dialect does not belong to any namespace, as in the previous example.

All XML Schema declarations for elements and data types immediately nested in the root `schema` element are called *global declarations*. Normally, the declarations of attributes are nested inside the declarations of their elements, and the declarations of the nested elements are nested inside the declaration of their hosting element. The proper usage of global declarations can promote declaration reuse, as you will see soon.

In the following examples, namespace prefix “`xs`” is assumed for the XML Schema namespace.

2.5.2 Declaring Simple Elements and Attributes

A *simple element* is an XML element that can contain only text based on simple data types defined in XML Schema specification (including `string`, `decimal`, `integer`, `positiveInteger`, `boolean`, `date`, `time`, `anyType`), those derived from such simple data types, or user custom types. It cannot contain any other elements or attributes. The following are some examples.

To declare element `color` that can take on any string value, use

```
<xs:element name="color" type="xs:string"/>
```

As a result, element `<color>blue</color>` will have value “blue”, and element `<color />` will have no value.

To declare element `color` that can take on any string value with “red” to be its default value, use

```
<xs:element name="color" type="xs:string"
  default="red"/>
```

As a result, element `<color>blue</color>` will have value “blue”, and element `<color />` will have the default value “red”.

To declare element `color` that can take on only the fixed string value “red”, use

```
<xs:element name="color" type="xs:string"
  fixed="red"/>
```

As a result, element `<color>red</color>` will be correct, element `<color>blue</color>` will be invalid, and element `<color />` will have the fixed (default) value “red”.

While simple elements cannot have attributes, the syntax for declaring attributes in XML Schema is very similar to that for simple elements. You just need to change “`xs:element`” to “`xs:attribute`” in the above examples. For example,

```
<xs:attribute name="lang" type="xs:string"
  default="EN"/>
```

declares that `lang` is an attribute of type `xs:string`, and its default value is “EN”. Such attribute declarations are always embedded in the declarations of complex elements to which they belong.

Attributes are optional by default. You can use attribute element’s `use` attribute to specify that the declared attribute is required for its hosting element. For example, if the above attribute `lang` doesn’t have a default value but it must be specified for its hosting element, you can use the following declaration:

```
<xs:attribute name="lang" type="xs:string"
  use="required"/>
```

2.5.3 Declaring Complex Elements

A *complex element* is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

- empty elements
- elements that contain only other elements

- elements that contain only customized simple types or attributes
- elements that contain both other elements and text

To declare that `product` is an empty element type with optional integer-typed attribute `pid`, you can use:

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="pid" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

Example `product` elements include `<product />` and `<product pid="1">`.

The following example declares that an `employee` element’s value is a sequence of two nested elements: a `firstName` element followed by a `lastName` element, both of type string.

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstName"
        type="xs:string"/>
      <xs:element name="lastName"
        type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The following is an example `employee` element:

```
<employee>
  <firstName>Tom</firstName>
  <lastName>Sawyer</lastName>
</employee>
```

Such nested element declarations have two problems. First, the width of paper or computer display will make deep element nesting hard to declare and read. Second, what if you also need to declare a `manager` element that also contains a sequence of `firstName` and `lastName` elements? The name declaration for the `employee` and `manager` elements would be duplicated.

Fortunately you can use global declarations and XML Schema element `type` attribute to resolve the above two problems. The following is the above `employee` element declaration in global declaration format as well as the declaration of a new `manager` element type.

```
<xs:element name="employee" type="fullName"/>
<xs:element name="manager" type="fullName"/>
<xs:complexType name="fullName">
  <xs:sequence>
```

```

    <xs:element name="firstName" type="xs:string"/>
    <xs:element name="lastName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

```

The following example declares a `complexType` element, `shoeSize`. The content is defined as an integer value, and the `shoeSize` element also contains an attribute named `country`:

```

<xs:element name="shoeSize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country"
          type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

```

An example `shoeSize` element is `<shoeSize country="france">35</shoeSize>`.

A mixed complex type element can contain attributes, elements, and text. You use attribute `mixed="true"` of the `complexType` element to specify that the value is a mixture of elements and text. The following declaration is for a letter element that can have a mixture of elements and text as its value:

```

<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderID"
        type="xs:positiveInteger"/>
      <xs:element name="shipDate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

The following is an example letter element:

```

<letter>
  Dear Mr.<name>John Smith</name>,
  Your order <orderID>1032</orderID>
  will be shipped on <shipDate>2008-09-23</shipDate>.
</letter>

```

2.5.4 Controlling Element Order and Repetition

For elements that contain other elements, the application of the sequence element (sequence, all and choice are called order indicators of XML Schema) enforces an order of the nested elements, as is the case for the previous employee element in which the nested lastName element must follow the firstName element. If you need to stipulate that each of the nested elements can occur exactly once but in any order, you can replace the sequence element with the all element, as in the following modified employee example:

```
<xs:element name="employee">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstName"
        type="xs:string"/>
      <xs:element name="lastName"
        type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

If you need to specify that an employee element need contain either a firstName element or a lastName element, but not both, you can replace the all element with the choice element in the above example.

Occurrence indicators are used to define how many occurrences that an element can be used. XML Schema has two occurrence indicators, maxOccurs and minOccurs, both are attributes of XML Schema element element.

Attribute maxOccurs specifies up to how many times that its hosting element can occur at that location. It takes on a non-negative integer or “unbounded” as the upper limit. Its default value is unbounded (unlimited).

Attribute minOccurs specifies at least how many times that its hosting element should occur at that location. It takes on a non-negative integer as the lower limit. Its default value is 1.

As an example, the following declaration specifies that the dvd element can occur zero or unlimited number of times.

```
<xs:element name="dvd" minOccurs="0"
  maxOccurs="unbounded">
```

2.5.5 Referencing XML Schema Specification in an XML Document

Not like DTD declarations, XML Schema declarations are always put in files separated from their instance document files. When you create an XML document, you may want to declare that the document is an instance of an XML

dialect specified by an XML Schema file. The method of such association depends on whether the XML Schema declaration uses target namespaces.

2.5.5.1 Specifying an XML Schema without Target Namespace

Assume that an XML dialect is specified with an XML Schema file *schemaFile.xsd* without using a target namespace, and the Schema file has URL *schemaFileURL*, which is either a local file system path like “*schemaFile.xsd*” or a Web URL like “*http://csis.pace.edu/schemaFile.xsd*”. The instance documents of this dialect can be associated with its XML Schema declaration with the following structure, where *rootTag* is the name of a root element, *xsi* is defined as the namespace prefix for *XML Schema Instance*, and the latter includes a *noNamespaceSchemaLocation* attribute for specifying the location of the XML Schema file that does not use a target namespace.

```
<rootTag
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
  xsi:noNamespaceSchemaLocation="schemaFileURL"
>
```

2.5.5.2 Specifying an XML Schema with Namespace

Assume that an XML dialect is specified with an XML Schema file *schemaFile.xsd* using target namespace *namespaceString* (say “*http://csis.pace.edu*”, and the Schema file has URL *schemaFileURL*, which is either a local file system path like “*schemaFile.xsd*” or a Web URL like “*http://csis.pace.edu/schemaFile.xsd*”. The instance documents of this dialect can be associated with its XML Schema declaration with the following structure, where *rootTag* is the name of a root element, *xsi* is defined as the namespace prefix for *XML Schema Instance*, and the latter includes a *schemaLocation* attribute for specifying the location of the XML Schema file that uses a target namespace.

```
<rootTag
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
  xsi:schemaLocation=
    "namespaceString schemaFileURL"
>
```

2.6 XML Parsing and Validation with SAX and DOM

Most XML applications need to read in an XML document, analyze its data structure, and activate events when some language features are found. SAX (Simple API for XML) and DOM (Document Object Model) support two popular types of XML parsers for parsing and processing XML documents. SAX works as a pipeline. It reads in the input XML document sequentially, and fires events when it detects the start or end of language features like elements and attributes. The application adopting a SAX parser needs to write an event handler class that has a processing method for each of the event types, and the methods are invoked by the SAX parser when corresponding types of events are fired. Since the XML document doesn't need be stored completely in computer memory, SAX is very efficient for some types of applications that don't need to search information backwards in an XML document.

On another hand, a DOM parser builds a complete tree data structure in the computer memory so it can be more convenient for detailed document analysis and language transformation. Even though DOM parsers use more computer memory, it is the main type of XML parser that supports the Ajax technology.

Both SAX and DOM can work in validation mode. As part of the parsing process, they can check whether the input XML document is well-formed. Furthermore, if the parser is fed both the XML dialect specification in DTD or XML Schema as well as an XML document, the parser can check whether the XML document is an instance of the XML dialect.

Since SAX is not used in Ajax, it will not be discussed further in this book. DOM parsing will be discussed in the following chapter.

2.7 XML Transformation with XSLT

As intermediate language representation of business data, it is critically important for XML instance documents being able to be transformed into other XML dialects, or into XHTML documents for customized Web presentation.

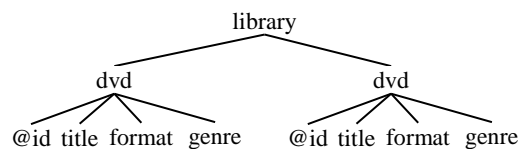
The World Wide Web Consortium (W3C) specified XSL (Extensible Stylesheet Language) as the standard language for writing stylesheets to transform XML documents among different dialects or into other languages. XSL stylesheets themselves are pure XML documents so they can be processed by the standard XML tools. XSL includes three components: XSLT (XSL Transformation) as an XML dialect for specifying XML transformation rules or stylesheets, XPath as a standard notation system for specifying subsets of elements in an XML document, and XSL-FO for formatting XML documents. This section briefly introduces XPath and XSLT. Most recent Web browsers support XPath and XSLT, and so is Sun's recent JDK versions.

Most examples in this section are based on file “dvd.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This XML document describes a DVD library -->
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</ genre >
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

2.7.1 Identifying XML Nodes with XPath

Before you can specify transformation rules for an XML dialect, you need to be able to specify subsets of XML elements that will be transformed based on some rules. You can visualize all components in an XML document, including the elements, attributes and text, as graph nodes, and describe an XML document as a tree growing downward in which a node is connected to another node under it if the latter is immediately nested in the former or is a parameter or text value of the former. This is basically a DOM tree for representing an XML document in computer memory. The attribute names have symbol @ as their prefix in such a tree. The sibling nodes are ordered as they appear in the XML document. As an example, the contents of file “dvd.xml” can be described by the following tree.



XPath uses *path expressions* to select nodes in an XML document. The node is selected by following a path similar to file system paths. The most useful path expressions include:

Expression	Description
<i>nodeName</i>	Selects all child nodes of the named node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the

	selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes
text()	Selects the text value of the current element
*	Selects any element nodes
@*	Selects any attribute node
node()	Selects any node of any kind (elements, attributes, ...)

Relative to the previous XML document `dvd.xml`, *path expression* `library` selects all the child nodes of the `library` element; `/library` selects the root element `library`; `library/dvd` selects all `dvd` elements that are children of `library`; `//dvd` selects all `dvd` elements no matter where they are in the document (no matter how many levels they are nested in other elements); `library//title` selects all `title` elements that are descendants of the `library` element no matter where they are under the `library` element; `//@id` selects all attributes that are named “id”; and `/library/dvd/title/text()` selects the text values of all the `title` elements under the `dvd` elements.

Predicates in square brackets can be used to further narrow down the subset of chosen nodes. For example, `/library/dvd[1]` selects the first `dvd` child element of `library` (IE5 and later uses `[0]` for the first child); `/library/dvd[last()]` selects the last `dvd` child element of `library`; `/library/dvd[last()-1]` selects the last but one `dvd` child element of `library`; `/library/dvd[position()<3]` selects the first two `dvd` child elements of `library`; `//dvd[@id]` selects all `dvd` elements that have an `id` attribute; `//dvd[@id='2']` selects the `dvd` element that has an `id` attribute with value 2; `/library/dvd[genre='Classic']` selects all `dvd` child elements of `library` that have “Classic” as their `genre` value; and `/library/dvd[genre='Classic']/title` selects all `title` elements of `dvd` elements of `library` that have “Classic” as their `genre` value. Path expression predicates can use many popular binary operators in the same meaning as they are used in programming languages, including `+`, `-`, `*`, `div` (division), `=` (equal), `!=` (not equal), `<`, `<=`, `>`, `>=`, `or` (logical disjunction), and `and` (logical conjunction), and `mod` (modulus).

You can use XPath wildcard expressions `*`, `@*` and `node()` to select unknown XML elements. For example, for the previous XML document `dvd.xml`, `/library/*` selects all the child nodes of the `library` element; `/*` selects all elements in the document; and `//dvd[@*]` selects all `dvd` elements that have any attribute.

Several path expressions can also be combined by the disjunctive operator `|` for logical or. For example, `//title | //genre` selects all `title` and `genre` elements in the previous document.

XPath also defines a set of *XPath axes* for specifying node subsets relative to the current node in a particular direction in the XML document's tree representation. The following table lists the popular XPath axis names and their meanings.

Axis Name	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the end tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects everything in the document that is before the start tag of the current node
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

There are two ways to specify a path expression for the location of a set of nodes: absolute or relative. An *absolute location path* starts with a slash / and has the general form of

/step/step/...

and a *relative location path* does not start with a slash / and has the general form of

step/step/...

In both cases, the path expression is evaluated from left to right, and each step is evaluated in the current node set. Each step has the following general form (items in square brackets are optional):

[axisName::]nodeTest[predicate]

where the optional axis name specifies the tree-relationship between the selected nodes and the current node; the node test identifies a node within an axis; and zero or more predicates are for further refining the selected node set. As examples relative to the XML document “dvd.xml”, `child::dvd` selects all dvd nodes that are children of the current node; `attribute::id` selects the id attribute of the current node; `child::*` selects all children of the current node; `attribute::*` selects all attributes of the current node; `child::text()` selects all text child nodes of the current node; `child::node()` selects all child

nodes of the current node; `descendant::dvd` selects all `dvd` descendants of the current node; `ancestor::dvd` selects all `dvd` ancestors of the current node; and `child::*/child::title` selects all `title` grandchildren of the current node.

2.7.2 Transforming XML Documents to XHTML Documents

XSLT is the major component of XSL, and it allows you to use the XML syntax to transform the instance documents of a particular XML dialect into those of another XML dialect, or into other document types like PDF. One of the popular functions of XSLT is to transform XML documents into HTML ones for Web-based presentation, which is the context for the examples in this section.

XSLT is based on DOM tree representation in computer memory. A common way to describe the transformation process is to say that XSLT transforms an XML source tree into an XML result tree. In the transformation process, XSLT uses XPath expressions to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

XSLT is an XML dialect which is declared under namespace “<http://www.w3.org/1999/XSL/Transform>”. Its root element is `stylesheet` or `transform`, and its current version is 1.0. The following is the contents of file “`dvdToHTML.xsl`” that can transform XML document “`dvd.xml`” into an HTML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" version="4.0"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>DVD Library Listing</title>
        <link rel="stylesheet" type="text/css"
          href="style.css"/>
      </head>
      <body>
        <table border="1">
          <tr>
            <th>Title</th>
            <th>Format</th>
            <th>Genre</th>
          </tr>
          <xsl:for-each select="/library/dvd">
            <xsl:sort select="genre"/>
            <tr>
              <td>
```

```

        <xsl:value-of select="title"/>
      </td>
      <td>
        <xsl:value-of select="format"/>
      </td>
      <td>
        <xsl:value-of select="genre"/>
      </td>
    </tr>
  </xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>

```

The root element `stylesheet` declares a namespace prefix “`xsl`” for XSL namespace “`http://www.w3.org/1999/XSL/Transform`”. This root element could also be `transform`. The 4th line’s `xsl:output` element specifies that the output file of this transformation should follow the specification of HTML v4.0. Each `xsl:template` element specifies a transformation rule: if the document contains nodes satisfying the XPath expression specified by the `xsl:template`’s `match` attribute, then they should be transformed based on the value of this `xsl:template` element. Since this particular `match` attribute has value “/” selecting the root element of the input XML document, the rule applies to the entire XML document. The `template` element’s body (element value) dumps out an HTML template linked to an external CSS stylesheet named “`style.css`”. After generating the HTML table headers, the XSLT template uses an `xsl:for-each` element to loop through the `dvd` elements selected by the `xsl:for-each` element’s `select` attribute. In the loop body, the selected `dvd` elements are first sorted based on their `genre` value. Then the `xsl:value-of` elements are used to retrieve the values of the elements selected by their `select` attributes.

To use a Web browser to transform the earlier file `dvd.xml` with this XSLT file `dvdToHTML.xsl` into HTML, you can add the following line after the XML declaration:

```
<?xml-stylesheet type="text/xsl" href="dvdToHTML.xsl"?>
```

The resultant XML file is `dvd_XSLT.xml` and its entire contents is shown below.

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

```
<?xml-stylesheet type="text/xsl"
  href="dvdToHTML.xsl"?>
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

The following CSS file *style.css* is used for formatting the generated HTML file:

```
body, td
{
  font-weight: normal;
  font-size: 12px;
  color: purple;
  font-family: Verdana, Arial, sans-serif;
}
th {
  font-weight: bold;
  font-size: 12px;
  color: green;
  font-family: Verdana, Arial, sans-serif;
  text-align: left;
}
```

The following screen capture shows the Web browser presentation of the HTML file generated by this XSLT transformation.



The screenshot shows a web browser window with the title 'DVD Library Listing - M...'. The browser's menu bar includes File, Edit, View, History, Bookmarks, and Tools. The main content area displays a table with three columns: Title, Format, and Genre. The table contains two rows of data: 'Gone with the Wind' (Movie, Classic) and 'Star Trek' (TV Series, Science fiction). The text in the table is purple, and the headers are green.

Title	Format	Genre
Gone with the Wind	Movie	Classic
Star Trek	TV Series	Science fiction

Element `xsl:value-of` can also be used to retrieve the value of attributes. For example, to retrieve the value of attribute `id` of the first `dvd` element, you can use

```
<xsl:value-of select="/library/dvd[1]/@id"/>
```

2.8 Summary

XML technologies are at the core of supporting platform and language independent system and data integration across the networks. They support the portable approaches of defining customized languages for describing business data structures, parsing and validating business data, and transforming business data among various forms.

2.9 Self-Review Questions

1. XML is mainly used for specifying business data in platform and programming language independent way.
 - a. True
 - b. False
2. An XML document can contain multiple root elements.
 - a. True
 - b. False
3. An XML dialect is a special XML document type that uses a predefined set of tag and attribute names and follows a predefined set of syntax rules; and it is for specifying the data structure of a particular type of documents.
 - a. True
 - b. False
4. DTD and XML Schema are the main mechanisms are declaring XML dialects.
 - a. True
 - b. False
5. XML Schema is more expressiveness in declaring XML dialects.
 - a. True

78 XML – the ‘X’ in Ajax

- b. False
- 6. An XML instance document can be claimed valid without referring to its dialect specification in DTD or XML Schema.
 - a. True
 - b. False
- 7. The value of an attribute must be inside a pair of double quotes or a pair of single quotes.
 - a. True
 - b. False
- 8. Namespace is for avoiding naming conflicts for element or attribute names so several XML dialects can be used in a single XML instance document.
 - a. True
 - b. False
- 9. If a namespace string is in the form of a URL, then there must be a corresponding Web resource deployed at the URL location.
 - a. True
 - b. False
- 10. SAX can always parse large XML documents more efficiently.
 - a. True
 - b. False
- 11. Each `template` element in an XSL document functions like a transformation rule.
 - a. True
 - b. False
- 12. XML processing tools are part of the latest Web browsers and Sun Java JDKs.
 - a. True
 - b. False

2.10 Keys to the Self-Review Questions

(To be provided later)

2.11 Exercises

1. What are the main functions of XML in today's IT technologies?
2. What kinds of XML documents are well-formed?
3. What kinds of XML documents are valid?
4. Why namespaces are important in XML technologies?
5. What are the similarities and differences of SAX and DOM parsers?
6. List some XML language features that can be specified with XML Schema but not with DTD?
7. What are the major differences between CSS and XSL stylesheets?
8. What is the function of XPath in XSLT?

2.12 Programming Exercises

1. Declare an XML dialect for specifying a subset of student course registration information with DTD.
2. Declare an XML dialect for specifying a subset of student course registration information with XML Schema.
3. Write an XSLT document to transform the instance documents of the above XML dialect into HTML through a Web browser.

2.13 References

1. Michael Morrison. *Sams Teach Yourself XML in 24 Hours, 2nd Edition*, Sams, 2002. ISBN 0-672-32213-7
2. Paul Whitehead, Earnest Friedman-Hill and Emily Vander Veer. *Java and XML*, Wiley Publishing, Inc., 2002. ISBN 0-7645-3683-4
3. W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. <http://www.w3.org/TR/xml/>
4. W3C. *XML Schema*. <http://www.w3.org/XML/Schema/>
5. W3C. *The Extensible Stylesheet Language Family (XSL)*. <http://www.w3.org/Style/XSL/>
6. SAX. <http://www.saxproject.org/>

7. W3C. *Document Object Model (DOM)*. <http://www.w3.org/DOM/>
8. *XML Tutorial*. <http://www.w3schools.com/xml/>
9. Sun Microsystems. *Simple API for XML*.
<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXPSAX.html>
10. *XML DOM Tutorial*. <http://www.w3schools.com/dom/>
11. *XSLT Tutorial*. <http://www.w3schools.com/xsl/>
12. *XPath Tutorial*. <http://www.w3schools.com/xpath/>

Index

#CDATA, 57
#PCDATA, 57
a:active, 25
a:hover, 25
a:link, 25
a:visited, 25
absolute location path, 73
absolute paths, 13
action, 36
all, 67
alt, 17
anchors, 14
application server, 2
ATTLIST, 59
attribute, 55
attributes, 5
b, 7
background-color, 12
blink, 24
body, 6
bold, 23
bolder, 23
border, 16, 21
border-color, 21
border-style, 21
border-width, 21
box model, 21
BPEL, 52
buffer overflow, 43
Business Process Execution Language, 52
caption, 15
Cascading style sheets, 19
Cascading Style Sheets, 2
character encoding, 54
checkbox, 42
Checkbox, 42
choice, 67
circle, 9
class, 25
clear, 18
code points, 54
collapsed margins, 22
color, 12, 23
colspan, 16
complex element, 65
complexType, 67
Content-length, 38
cookie, 44
cookies, 3
CSS, 2, 19
dashed, 21
data integration, 51
decimal, 9
disc, 9
div, 26
Division-Based Layout, 30
DNS, 35
DOCTYPE, 6
Document Object Model, 70
Document Type Definition, 56
DOM, 70
domain name, 34
domain name server, 35
domain-name, 13
dotted, 21
double, 21
DTD, 56
element, 4
ELEMENT, 57
element value, 4
elements, 53
Empty elements, 57
end tag, 4, 54
entities, 11
entity, 55
ENTITY, 60
entity body, 38
entity name, 11, 55, 60
entity number, 11
entity references, 55
entity referencing, 55
Extensible Hypertext Markup Language, 2
eXtensible Markup Language, 52
Extensible Stylesheet Language, 70
Firefox, 4
float, 18
font family, 22
font-size, 23
font-style, 23
font-weight, 23
form, 34, 36
frameborder, 32
ftp, 34
GIF, 16
global declarations, 64
graphic, 16

82 Index

Graphic Interchange Format, 16
groove, 21
h1, 7
h2, 7
h3, 7
h4, 7
h5, 7
h6, 7
head, 6, 19
header lines, 38
height, 17
hidden, 43
Hidden Field, 43
hidden fields, 45
href, 14
html, 6
HTML, 1, 4
HTML entities, 11
HTML *form*, 36
HTML skeleton, 6
HTML version 4, 1
HTTP, 2, 34
HTTP GET, 36, 43
HTTP POST, 36, 43
https, 34
hyperlinks, 13
Hypertext Markup Language, 1
Hypertext Transfer Protocol, 2
HyperText Transfer Protocol, 34
i, 7
id, 24
iframe, 32
image, 17
input, 40
input controls, 37
Input Controls, 40
inset, 21
Internet Explorer v7, 4
IP address, 13, 34
Joint Photographic Experts Group, 17
JPEG, 17
JPG, 17
JSP, 3
large, 23
leading, 24
left, 18
letter-spacing, 24
li, 9
lighter, 23
line-height, 24
line-through, 24
link, 19
list-style-type, 9
localhost, 35
lower-alpha, 9
lower-roman, 9
mailto, 14
margin area, 21
maxOccurs, 68
medium, 23
meta-characters, 11
method, 36
MIME, 39
minOccurs, 68
mixed complex type element, 67
multiple, 41
Multipurpose Internet Mail Extension, 39
name, 37
namespace, 6, 63
namespace prefixes, 63
nested elements, 5
noNamespaceSchemaLocation, 69
none, 21, 24
normal, 23
occurrence indicators, 68
ol, 9
onclick, 30
option, 41
order indicators, 67
ordered list, 9
outset, 21
overline, 24
p, 7
padding area, 21
Password Control, 41
path expressions, 71
pixel, 16
PNG, 17
port number, 35
Portable Network Graphics, 17
position, 28
pre, 7
Predicates, 72
process, 35
pseudo-classes, 25
query string, 35, 43
query strings, 45
radio, 41
Radio Buttons, 41
Referer, 38
regular expressions, 58
relative, 28
relative location path, 73
relative path, 13
reset, 42
Reset Button, 42

resource, 34
 ridge, 21
 right, 18
 root element, 6
 SAX, 70
 schemaLocation, 69
 scrolling, 33
 select, 41, 75
 Select Control, 41
 selected, 41
 sequence, 67
 server applications, 35
 server computer, 35
Service Oriented Architecture Protocol, 52
 servlet, 3
session, 44
session data, 44
 session ID, 45
session objects, 45
SGML, 52
 Simple API for XML, 70
Simple Object Access Protocol, 52
 size, 41
 small, 23
SOAP, 52
 software component, 3
 solid, 21
span, 26
 square, 9
 src, 17
 start, 9
start tag, 4, 54
 style, 9, 12
 Style Rule Format, 20
 style rules, 20
 stylesheet, 74
 stylesheets, 70
subdomains, 34
 Submission Button, 42
 submit, 42
 table, 15
tag name, 4, 54, 57
 target, 14, 17
 targetNamespace, 64
 TCP/IP, 3
td, 15
 Text Area, 40
 Text Field, 40
 text-align, 23
 textarea, 40
 text-indent, 24
th, 15
 title, 14, 17
 tooltip, 14, 17
 top-level domain names, 34
 top-level element, 53
tr, 15
 transform, 74
tt, 7
ul, 9
 unbounded, 68
 underline, 24
Unicode, 54
 Unicode Encoding, 54
 Unicode Transformation Format, 54
 Uniform Resource Locator, 34
universal resource identifier, 6
 universal resource location, 13
 unordered list, 9
 URI, 6
 URL, 13, 34
URL encoding, 35
 UTF-16, 54
 UTF-32, 54
 UTF-8, 54
 value, 41
value of the element, 54
 W3C, 70
 Web applications, 3
 Web Architecture, 2
 Web browser, 1
Web browser sandbox, 3
 Web server, 1
 well-formed XML document, 56
white-space characters, 5, 7
 width, 17
 word-spacing, 24
 World Wide Web Consortium, 70
 XHTML, 2, 4, 6
XML, 52
XML comment, 53
XML declaration, 53, 54
 XML dialect, 52, 57
 XML document, 53
 XML namespace, 63
 XML parser, 70
XML Schema, 62
 xmlns, 63
 XPath, 70, 71
XPath axes, 73
 xs:attribute, 65
 xs:element, 63
 xs:schema, 64
 XSL, 70
 XSL Transformation, 70
 xsl:for-each, 75

84 Index

xsl:output, 75
xsl:template, 75
xsl:value-of, 75

XSL-FO, 70
XSLT, 70, 74