Chapter 9
How a Web Page Works

The language used on a Web page is HyperText Markup Language, or HTML. Hypertext is the part of the Web that allows for the linking together of Web content through the use of clickable "hyperlinks." The markup language defines the document form. Remember back in grade school getting an essay back, "marked up" with symbols stating that an item should be capitalized, or a new paragraph should begin here? Well, that is pretty much what HTML does for a Web page. In this chapter I will cover what makes up a well formed Web page, what the different sections are, how to use JavaScript for form validations, and how Web Connection can process a page that contains embedded expressions.

HTML is yet another technology that new Web developers need to understand. When developing Visual FoxPro forms using the form designer, developers had the luxury of a fully object-oriented design surface that was seamlessly integrated with the data engine. It was possible to simply drag and drop fields from a table onto the form and have the resulting text box bound to the data field. When designing an HTML form, there is no integration between the form objects and the data engine. Also, there are no object-oriented form objects in HTML. In Visual FoxPro, developers had the easy life. It was possible to create class libraries of the base form objects, give them custom looks and behaviors, and make full use of inheritance to allow for easy maintenance. In HTML, the closest thing to inheritance is Cascading Style Sheets, or CSS. CSS is what brings uniformity and easy site maintenance to Web development. Web pages should not be designed without them!

In this chapter, I will use the HTML-Kit authoring tool from Chamai Software. It is a free tool, easy to use, and very well supported. See the "Chapter resources" section for the download site. The forms that will be used to build the "TODO" application in Chapter 12, "A Web Connection Application from Start to Finish," will be dealt with in detail here. These forms are simple but utilize most of the principals needed to build complex forms using HTML and Web Connection.

A brief history of HTML

Tim Berners Lee developed HTML in the early 1990s. Mr. Lee was the inventor of the World Wide Web as we know it today. HTML was born out of the need at the CERN High Energy Research Labs in Europe to tie together the documents from researchers from all over the world. The scientists and engineers brought their own machines and software, and trying to get them to communicate was a difficult task. HTML was developed as a standard document format that could be used over the existing network protocols.

HTML was based on SGML, or the Standard Generalized Markup Language, which had been in use for some time already. SGML had already been using "tags," or instructions enclosed in "<>" angle brackets, to denote the form of the document. Mr. Lee made every effort to keep the form of HTML very similar since people were already familiar with the syntax of SGML. As it turned out, this syntax choice also made it very easy for humans to read

and understand the code that made up a Web page. As you'll see, the various tags that make up a Web page give the document structure. In other words, HTML tags are to a Web page what steel beams are to a building. They provide the structure, but they do not necessarily determine the outward appearance. Cascading Style Sheets determine the appearance of a Web page, like a façade determines a building's appearance. Structure and appearance need to be separated, as we shall see.

The editor interface

The HTML editor that will be used in this chapter is HTML-Kit from Chami Software (see the "Chapter resources" section). There are many HTML authoring tools available, such as Adobe's GoLive and Macromedia's Dreamweaver. Dreamweaver is a very sophisticated authoring tool with some great features. It would be beyond the scope of this book, however, to get into using a tool like Dreamweaver. HTML-Kit is a highly glorified "Notepad" editor that provides preview ability and some great add-on utilities (see **Figure 1**). Its menu system allows for the insertion of all of the HTML elements as well as Cascading Style Sheet attributes. So instead of having to type in all the HTML tags, you can select and insert them from the menu system. There is also an integrated FTP client to make it simple to upload the files to the production server. For easy viewing of the HTML code used in this chapter's examples, HTML-Kit is a great choice. It is easy to use and makes it easy for beginners to learn their way around an HTML document.

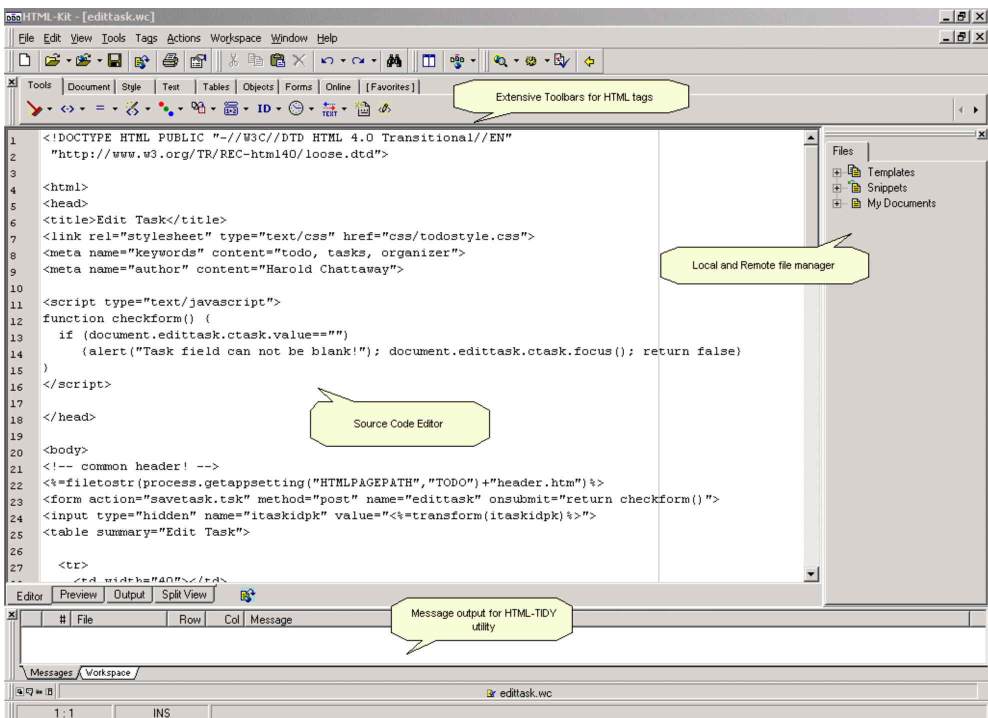

***Figure 1**. The HTML-Kit authoring tool interface.*

However, for more advanced site management and HTML authoring ability, Macromedia's Dreamweaver is an excellent choice. Dreamweaver has an extensible development environment, very much like Visual FoxPro's. It has an integrated JavaScript interpreter that can be used to interact with the Web page under development similar to the way Builders in Visual FoxPro can interact with a Visual FoxPro form in the forms designer. One of its most powerful features is its template technology. Unlike templates in HTML-Kit and virtually every other tool, Dreamweaver's templates provide the ability to create a "baseclass" form. When this base template is updated, all pages that are based on this template are updated too! This makes for greatly simplified site management. Though it is not a base class in the same sense as a real object-oriented base class, it provides basically the same benefit.

The Tools tab in HTML-Kit along the top of the editor is used to gain access to different categories of HTML tags. Under the Forms tab are all the form object tags. You can insert textboxes, dropdowns, and list boxes from here. The Preview tab allows you to see the page rendered in a browser. This is not a WYSIWYG editor, so it is necessary to switch back and forth between the Editor and Preview tabs. Remember that this will not correctly evaluate the embedded ASP tags in the page. Previewing does not cause the page to be evaluated by your Web Connection application!

On the right side of the editor is the File interface. This allows you to specify local directories within the editor for easy access to your files. You can also specify FTP sites to make it easy to drag and drop files from your local machine to a production server.

An important step in validating Web pages is to validate them against the HTML standards. You can easily do this in HTML-Kit by choosing Tools | Check Code Using TIDY from the main menu. This will find any errors in the HTML code that doe not adhere to the standards for the version of HTML used in the Web page. The standard that the TIDY utility will apply to the page is specified by the DOCTYPE definition at the top of the Web page. This is described in detail later.

The example "TODO" application

The example used in this chapter and described in detail in Chapter 12 is a task management system called "TODO." This is a simple but very useful application that will be used to illustrate many of the key features of Web Connection and Web development in general. This chapter will focus on how the Web pages are constructed and how a developer can improve the user experience. The application allows users to enter tasks, assign them to others, and assign status and priority. Reports can be run that list all tasks for ADMIN users or tasks filtered only to the current user. Once the task is completed, "Date Complete" and "Solution" can be filled in. While many sample applications do not have a practical application, this one could be installed either locally or on an intranet and serve a very useful purpose.

Different techniques for generating a page

There are three different techniques a developer can use to generate a Web page using Web Connection:

1. *Templates:* Templates in Web Connection are pages that contain both HTML markup tags and Active Server Pages (<%=%>) style expressions that contain any valid

Visual FoxPro expression. Templates are used in the TODO application. The first page described in this chapter is a template page. The location on disk for templates is referenced by the HTMLPAGEPATH entry in the application's INI file. There could be another custom entry made to point to the location on disk where the templates are stored. For example, the HTMLPAGEPATH setting could be used for static HTML pages that require no processing by Web Connection. Then there could be another custom setting called SCRIPTPAGEPATH that is the path to the directory holding dynamic pages such as scripts and templates. These templates do not have to be stored in the Web site's root directory with the regular HTML pages. Since these pages are first read and parsed by Visual FoxPro and Web Connection, they can be stored in any directory the developer sees fit. The Web site root directory is a valid place to store them, but they do not have to be there. For the examples shown in this book, static and dynamic pages will reside in the same directory. One advantage to using templates is that you can make updates to Web pages by simply FTPing a new page to the site. A new Web Connection EXE does not have to be uploaded. File extensions are usually not .HTM. An extension of either .WC or .WCT can be used to identify it as a Web Connection template page.

2. *Scripts:* Scripts are Web pages that contain HTML tags as well as Visual FoxPro code blocks. TASKLIST.WCS is an example of a script page. An example of a use for scripts is where an HTML table is laid out in a Web page editor. The HTML table will represent rows of a Visual FoxPro cursor. In order to have the HTML table rows repeated for each cursor row, those tags are wrapped in a SCAN/ENDSCAN loop. Any Visual FoxPro language construct can be included on a Web page by enclosing it in <% %> delimiters. This technique is described in greater depth in Chapters 10 and 12. The location of these files follows the same rules as described for templates. Here, like templates, if a change is needed the new page simply has to be FTPed up to the server. For script pages, the extension is usually .WCS.
3. *Generating pages from Visual FoxPro:* In some cases it may be necessary and easier to generate Web pages manually from within a Web Connection application. This works best for simple pages. Using templates or scripts that are laid out using an HTML authoring tool and then modified to contain ASP expressions is generally the easiest way to develop a site. Some developers have laid out pages in FrontPage, for example, and then cut and pasted each of line of code into their Web Connection application using the WRITE() or WRITELN() function to output each line individually. This approach is very cumbersome and breaks the link between the HTML authoring tool and the HTML code. Also, if any changes are needed to the generated page, a new EXE must be uploaded and swapped out with the old EXE.

It is generally best to use the tool most appropriate for the underlying language. Visual FoxPro is not an editor for HTML, and an HTML authoring tool is not the best editor for Visual FoxPro. With this in mind, templates fit this rule the best. Scripts are a great technique to use when necessary as long as a lot of Visual FoxPro code is not embedded in the Web page.

The TODO sample application

In this book, a sample "TODO" application is used to illustrate how to use Web Connection. It is a task management application that can be used to record your daily checklist of activities. It consists of a main task list, which is a simple listing of all tasks you have entered. Each one contains a hyperlink to a page showing the detail of that task. Entries can be added with a description of what the task is, whom it is assigned to, and when it is due. The task list can be filtered to show only your tasks or everyone's. This application is included with this book's source code, available at **www.hentzenwerke.com**, and can be used however you like.

How is this Web page rendered?

The Web page shown in **Figure 2** is displayed when the user clicks on a particular TODO item in the main listing. This is the rendered version of the EDITTASK.WC file that is stored on the Web server. The method EDITTASK is executed in the CLASS_TSK file. It is described in detail in Chapter 12, but briefly this translates to the link http://localhost/todo/edittask.tsk?id =1000. This method then performs a query and pulls out the record in the TODO table with a primary key of 1000. Then the EDITTASK method calls the Web Connection framework method EXPANDTEMPLATE() with the following line:

```
response.expandtemplate(PROCESS.cHTMLPAGEPATH+"edittask.wc",loHeader)
```

Web Connection then opens up the file EDITTASK.WC on disk and begins the parsing process. It is looking for ASP tags like <% %>. ASP is a technology, not a language. Any language can be utilized inside of these delimeters. When using Web Connection to parse a template, the language inside these delimeters will be Visual FoxPro.

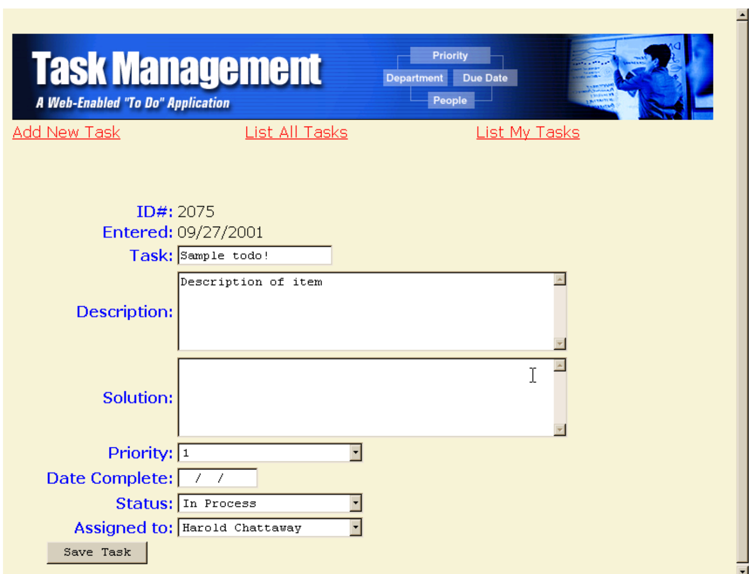

***Figure 2**. The form EDITTASK.TSK.*

Any expression contained within the <% %> delimeters is evaluated and the result is inserted back into that location. The only requirement is that the result of the expression be a character string. The resulting page is what is then sent to the browser to be rendered. All of the ASP-style tags are evaluated on the server! The expression can consist of a simple in-line statement such as <%=date()%> or can be a call to a 1000-line method. There is really no limit to what can be done with these expressions. The ASP tags serve as a window into the entire Visual FoxPro language. The expression has access to any value currently in scope at the time of evaluation. So after the server gets done evaluating all of these expressions on the page, the hard-coded HTML page is then sent back to the browser that requested it. The browser reads in the HTML page and does some parsing of its own. It determines what objects (text boxes, dropdowns, text areas) are going to be used and displays the visual representation of these in the browser. It evaluates any JavaScript that may need to be run as the page is loading. It also determines whether there are any external links to this page like GIF or JPG images. If there are any references to such files, it opens up another channel and sends a request to the server to retrieve it. So after all the parsing, validating, and link resolution is complete, you should be looking at a fully formed Web page. As you can imagine, there is a lot of work that goes on to fully render a page. And still a great deal of the transmitting is done over dial-up modem connections that run no faster than about 56K.

It is a good idea to keep your audience in mind. If your Web page is targeted to users sitting at home, it would be best to keep the pages very "lite"—low on graphics. If your user base is mostly a commercial crowd sitting on fast fiber connections, you may have some more leeway.

What makes up a Web page?

To understand what makes up a Web page, let's take a look at the "TODO" data entry form that is part of the application that will be developed in Chapter 12. The HTML shown in **Figure 3** is the source code for the EDITTASK.WC Web page. The HTML tags that make up a Web page are processed by a browser. It is the browser's job to correctly interpret the HTML tags and display them on a Web page. The ASP-style tags that contain Visual FoxPro expressions are processed on the server. When the embedded expression is evaluated, the resulting string is inserted into the Web page at the same location and sent to the browser. How these two pieces interact will be explained later in this chapter.

Figure 3 will serve as the outline of for the rest of the chapter. This code is available with the source code for this book at **www.hentzenwerke.com**, so it's not necessary for you to re-create this code in order to try it out.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
2   "http://www.w3.org/TR/REC-html40/loose.dtd">
3
4  <html>
5  <head>
6  <title>Edit Task</title>
7  <link rel="stylesheet" type="text/css" href="css/todostyle.css">
8  <meta name="keywords" content="todo, tasks, organizer">
9  <meta name="author" content="Harold Chattaway">
10
11 <script type="text/javascript">
12 function checkform() {
13   if (document.edittask.ctask.value=="")
14      {alert("Task field can not be blank!"); document.edittask.ctask.focus(); return false}
15 }
16 </script>
17
18 </head>
19
20 <body>
21 <!-- common header! -->
22 <%=filetostr(process.getappsetting("HTMLPAGEPATH","TODO")+"header.htm")%>
23 <form action="savetask.tsk" method="post" name="edittask" onsubmit="return checkform()">
24 <input type="hidden" name="itaskidpk" value="<%=transform(itaskidpk)%>">
25 <table summary="Edit Task">
26
27   <tr>
28     <td width="40"></td>
29     <td class="label">ID#:</td>
30     <td><%=transform(itaskidpk)%></td>
31   </tr>
32   <tr>
33     <td width="40"></td>
34     <td class="label">Entered:</td>
35     <td><%=dentered%></td>
36   </tr>
37   <tr>
38     <td width="40"></td>
39     <td class="label">Task:</td>
40     <td><input type="textbox" name="ctask" value="<%=ctask%>"></td>
41   </tr>
42   <tr>
43     <td width="40"></td>
44     <td class="label">Description:</td>
45     <td><textarea rows="5" cols="50" name="mnotes"><%=mnotes%></textarea></td>
46   </tr>
47   <tr>
48     <td width="40"></td>
49     <td class="label">Solution:</td>
50     <td><textarea rows="5" cols="50" name="msolution"><%=msolution%></textarea>
51     </td>
52   </tr>
53   <tr>
54     <td width="40"></td>
55     <td class="label">Priority:</td>
56     <td><select name="npriority">
57     <option value="1" label="1">1</option>
58     <option value="2" label="2">2</option>
59     <option value="3" label="3">3</option>
60     </select></td>
61   </tr>
62   <tr>
63     <td width="40"></td>
64     <td class="label">Date Complete:</td>
65     <td><input type="textbox" name="dcompleted" value="<%=dcompleted%>"
66          size="10" maxlength="10"></td>
67   </tr>
68   <tr>
69     <td width="40"></td>
70     <td class="label">Status:</td>
71     <td><%=cStatusDD%></td>
72   </tr>
73   <tr>
74     <td width="40"></td>
75     <td class="label">Assigned to:</td>
76     <td><%=cAssignedDD%></td>
77   </tr>
78
79   <tr>
80     <td width="40"></td>
81     <td><input type="submit" value="Save Task"></td>
82     <td></td>
83   </tr>
84
85 </table>
86 </form>
87
88 </body>
89 </html>
```


***Figure 3**. Sample HTML page.*

The header section of an HTML page

The section of the Web page that comes before the <BODY> tag is not displayed in the browser. This area of the document is meant to hold meta tags and scripting functions that can supply form validation or Dynamic HTML (DHTML) functionality. Let's look at each one in more detail.

- **DOCTYPE:** Line 1 of the listing is for specifying the DOCTYPE. This Document Type Definition (DTD) is for the use of form validators only, for now. In Microsoft Internet Explorer 6, using this tag will turn on strict compliance checking for the document. There are utilities that check the validity of HTML forms to make sure they adhere to certain standards. Since there are a number of versions of HTML out now, the validator has to be told what version to check the document against. The DOCTYPE tag does just that:
    - **<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">** This declares the document to be HTML 4.01 Strict. Strict is a trimmed-down version of HTML 4.01 that stresses structure of presentation. Out-of-date elements, frames, and link targets are not allowed in Strict. By adhering to the strict definition, Web authors make their pages accessible and easy to adapt to style sheets.
    - **<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">** This declares the document to be HTML 4.01 Transitional. HTML 4 Transitional includes all elements and attributes of HTML 4 Strict but adds presentational attributes, out-of-date or "deprecated" elements, and link targets. HTML 4 Transitional recognizes the relatively poor browser support for style sheets, allowing many HTML presentation features to be used as a transition toward HTML 4 Strict.
    - **<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/html4/frameset.dtd">** This is a standard specially for pages using frames.
    - **<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">** This specifies that the page adheres to the 3.2 version of HTML. Most browsers support this version of HTML fully.

The form validator goes out to the DTD file specified in the DOCTYPE URL and retrieves the specification in use by that page. The validator can then compare the current page to the specification to see how it matches the definition. HTML-Kit has one of these validators built in (HTML-TIDY) and will do the validation and present a report showing the errors in the page.

- **<HTML>:** This tag on line 4 tells the browser that what follows is HTML code. As you shall see later, non-HTML instructions can be placed above this line.

- **<HEAD></HEAD>:** The HEAD section, or header, contains page-level information. This is where META tags, the page title, style sheet references, and JavaScript code reside.
    - **TITLE tag:** This tag defines the text that will appear in the title bar of the browser window. In most cases this is just the name of the site.
    - **LINK tag:** This tag defines the CSS file that will be used to define the appearance of the site. There will be a detailed section on style sheets later on. LINK tags can also be used to reference other types of files, such as external JavaScript function libraries. This will be demonstrated later.
    - **META tags:** These tags can provide additional information about the page to search engines and also special instructions to the browser that can tell the browser how to cache the current page. The most common meta tags are "keywords" and "description." The keywords tag is what the indexing engines from the various search engines use to determine your ranking in those engines. How search engines work and how to best promote your site is covered in a later chapter. The description meta tag is used by most search engines to provide a two- or three- sentence description of your site in the listing report. How to use meta tags will be covered in Chapter 21, "Marketing Your Web Site."
    - **SCRIPT tags:** The script tags tell the browser that what is contained within the script tags is a scripting language and not HTML tags. Typically the scripting language is JavaScript, or JScript in Microsoft Internet Explorer. JavaScript is used in all modern browsers. VBScript as a scripting language is only used in Internet Explorer. It is JavaScript that is used to add form validation and Dynamic HTML (DHTML) functionality to the page. Using JavaScript, the Web developer can access the Document Object Model (DOM). The DOM is what exposes the page elements and allows them to be controlled through code. This will be very familiar to Visual FoxPro programmers who have worked in the form designer. There are very close equivalents on a Web page to the syntax Visual FoxPro developers are accustomed to using.

The <BODY> section of an HTML page

The <BODY> section of a Web page contains the tags that are used to display information in the browser. This is the section where all of the form objects live.

Common menu and banner

One of the first things that is done in the <BODY> section is to pull in a common menu system that will be used throughout the application. In line 14, the Visual FoxPro function FILETOSTR() is used to read the file HEADER.HTM from disk and convert it into a string that is then reinserted back into the page. This HEADER.HTM file does not contain any ASP tags that need to be evaluated, so returning it as a string is sufficient at this point. This is a very convenient way to have a consistent menu and banner for the entire site that is easy to

maintain. Any changes that are made to the menu system or banner are instantly made available to all pages when this technique is used. If the code in HEADER.HTM lived on every page in the site, maintenance would be a nightmare. The smallest update to the menu system would have to be done to all pages. The three menu commands available in this menu are Add Task, List all Tasks, and List my Tasks. Each of these will be covered in Chapter 12.

Give your page some <FORM>

The <FORM> tag as shown on line 15 will form a wrapper for all the form elements such as text boxes, text areas, and list boxes. When a Web page is submitted back to the server, the values of any objects that reside between the <FORM></FORM> tags will be sent to the browser as a series of name/value pairs. Let's see what the various attributes of this tag are:

- **ACTION=:** The ACTION attribute tells the server what should be done when the page is submitted. In Web Connection applications, this would typically be a call to a custom method in the Web Connection application. In this example, the method SAVETASK will be executed inside the CLASS_TSK file. The mechanics of script mapping will be covered in detail in Chapter 12. Basically the action SAVETASK.TSK is mapped to the method SAVETASK in the CLASS_TSK procedure file through the use of a CASE statement in the main program. Very simple, but very powerful and elegant. This ACTION is initiated when the user click on a SUBMIT button like the one defined in line 81. This is a special type of button that is associated with the FORM tag that contains the button. FORM tags cannot be nested, so there is never any confusion as to what FORM tag a SUBMIT button belongs to.
- **METHOD=Post:** The METHOD attribute specifies whether the browser is to send form values to the server, or retrieve information from the server. The POST value tells the browser to send the form values to the server, as opposed to the GET value, used for retrieving information from the server. When submitting a form, the METHOD will always be POST. Hyperlinks send their request to the server in the form of a GET with any additional information being sent to the server on the URL. For example a GET request might take the form of http://localhost/todo/edittask.tsk?id=1000**.**
- **NAME="EDITTASK":** This attribute allows any scripting that may be on the page to refer to the form by a name. For example, instead of referring to a page element as "document.form[0].lastname.value="Smith", it can be referred to as "document.edittask.lastname.value="Smith".

How values are stored on a page

When designing a conventional Visual FoxPro form, it is sometimes necessary to store information in a form property. This piece of data may or may not be displayed in a form object. In a Web page, you have pretty much the same ability. As can be seen on line 24, a form property can be created by using:

```
<INPUT TYPE="hidden" NAME="itaskidpk" VALUE="<%=transform(itaskidpk)%>">
```

This form object stores the value in an invisible object that can be referenced through scripting and is submitted along with all the visible form elements. Here is an example of how easy it is to reference a Visual FoxPro field value. Simply enclose the expression in the ASP tags and make sure the value returned to the page is a string. The TRANSFORM() function is perfect for this since it will convert any data type to a string with the value trimmed both left and right. Any values associated with visible form elements are submitted with the page and do not need a “hidden” version of the property.

Storing data as hidden form variables is an easy way to carry forward information from one hit to the next. As you shall see in Chapter 12, all Web requests are independent. There is no knowledge whatsoever from one request to the next as to what the last request was and who made it. So if there is any information that needs to be carried forward to the next request, a good place to store that data is in hidden form variables.

Placing form elements on the page

When laying out a Web page, you typically use the <TABLE> element to position elements on a page. This tag allows the creation of as many rows and columns as you need. In our example, there are three columns used. The first column is used to create a left-hand margin, the second column is for labels, and the third is for holding the form element such as a text box. Some attributes for the <TABLE> tag are:

- **Width="x":** This controls the total width of the table in either pixels or a percentage basis. If using percentage, it is not generally a good idea for it to be 100%. The page has a much better appearance with both left-hand and right-hand margins. A percentage of 90% works well. In order for the appearance to be consistent, it is best to design to a fixed width for most pages. For example, the minimum horizontal resolution is 800 for most people. So width=“800” will fix the width of the table to be 800 pixels.
- **Border="x":** This defines whether there is a visible border. This is more a personal preference, but many times a border of “0” looks best. This controls the outside border of the table, not the lines that separate the rows and columns.
- **Cellpadding="x":** This sets the distance between the border and the contents of the cell.
- **Cellspacing="x":** This controls the distance between the cells. This in effect makes the lines separating the rows and columns larger or smaller.

There are two sub-elements to the <TABLE> tag:

- **<TR></TR>:** This tag creates a new row within a table. The closing </TR> tag must be used after the last cell in the row.
    - **Align="right|left":** This controls horizontal alignment of cell contents.
    - **Valign="top|middle|bottom":** This controls vertical alignment.
- **<TD></TD>:** This tag creates individual cells within a row. These elements should hold one object only. If two TEXTBOX elements were placed in one cell, for

example, the appearance would be very hard to control. There are some attributes to the <TD> tag that can be used to help control appearance.

- **Align="right|left":** This controls horizontal alignment of cell contents.
- **Valign="top|middle|bottom":** This controls vertical alignment.
- **Height="40":** This attribute controls the height of the cell in either pixels or percentage. This attribute is available for the <TR> element as well. It is common to have to force the height of row to a specific value to create the proper spacing.

- **<TH></TH>:** This tag is used to create a header for each column. This makes the header bold and centered. This is an accessibility guideline requirement.

More and more it is becoming important to make sure Web pages can be used by people with various disabilities. For example, the "title" attribute should now be used on all form objects. This makes it possible for agents to actually read the page to a person who is blind. See the reference in the "Chapter resources" section for accessibility guidelines. For any developers who are doing work for government agencies, it will be a requirement very soon to make Web sites accessible to all!

By setting the table width as well as cell widths and heights, correct placement of form elements can be achieved. Most HTML forms designers allow this to be done visually. The simpler HTML-Kit editor used in this chapter does not have a WYSIWYG editor, but Adobe's GoLive and Macromedia's Dreamweaver do. Dreamweaver has the most sophisticated one in the bunch. While the preceding display attributes can be set directly in the page, you will see later that it is best to externalize these attributes in a style sheet. Style sheets can be a Web designer's best friend!

Common form objects

The form elements that are available on a Web page are pretty similar to what is available in the Visual FoxPro form designer. The one huge difference is that HTML objects do not support inheritance. So the designer does not have the luxury of creating a class library with customized versions of all the controls. However, as you shall see shortly, style sheets bring some of this convenience to a Web page.

This section will cover the key form objects and show how they are used within a Web Connection application. For a more complete explanation of HTML objects, see the "Chapter resources" section. Let's take a look at these objects one by one:

- **<input type = "textbox" name = "ctask" value="<%=ctask%>" size="30 " maxlength="50">** This object is the equivalent of the textbox object in Visual FoxPro.

In file "edittask.wc" on server	As browser sees it
<input type = "textbox" name = "ctask" value="<%=ctask%>" size="30 " maxlength="50">	<input type = "textbox" name = "ctask" value="Create edituser form" size="30 " maxlength="50">

- **input type="textbox":** This creates a standard single-line text box field.
- **Name="ctask":** This name is used when the form is submitted to the server to identify the associated value. The listing of name/value pairs simply lists each field and its current value. This can also be used by JavaScript embedded in the page to read and set the value of that object. If using this to display table values, the object name should be the same name as the field in the table. This makes for much more readable code.
- **Value="<%=ctask%>":** The value attribute sets the value that will be displayed in the text box. The value can be derived from an ASP expression as shown here. This is typically a field name in a table or a memory variable. It can also be a call to a custom method. The value should be enclosed in quotes. Remember, all data types on a Web page are character! If there are no quotes and there are spaces in the value string, the value when read will be truncated after the first space.
- **Size="30":** This sets the visible size of the field on screen. This is typically the size of the field in the corresponding table.
- **Maxlength="50":** This can be used if the size attribute needs to be set lower to conserve screen real estate. Maxlength will limit the maximum number of characters entered into that field. So if the field is 50 characters in length, but there is only room to show 20, size could be set to 20 and maxlength to 50, and any characters over 20 will scroll in the text box. Remember, the size attribute has to be set manually. When a field is dropped onto a Visual FoxPro form, this is set automatically. If this is not set properly and the on-screen size is larger than the width in the table, characters will be truncated when stored in the field!

- **<input type="checkbox" value="<%=cadmin%>" name="cadmin" checked>** The checkbox object can be used to set flags. In this example, there could be an admin field in the user table. If checked, that person would have administration rights. The "checked" attribute determines whether the box is initially checked when the form is displayed. The way this would be handled with script is to create an expression that checks the current value and then returns the string "checked" or nothing. For example, <%=iif(cadmin="Y","checked","")%> could be used to dynamically output the correct definition for this object. *Tip:* Since the only data type that a form value can have is character, it makes more sense in many cases to define the matching field in the table to be character also. In this example, an ADMIN field would typically be of type logical. However, if it is made character and either a "Y" or an "N" is used, it eliminates a lot of unnecessary data conversions. Of course, this technique should not always be used! For example, it is always a good idea to store dates as dates so date functions can be used easily!

In file "edittask.wc" on server	As browser sees it
<input type="checkbox" value="<%=cadmin%>" name="cadmin" checked>	<input type="checkbox" value="Y" name="cadmin" checked>

- **<input type="password" value="" name="password" size="10">** The password field is the same as the "textbox" field except the on-screen characters are replaced with "*" as the user types. This is only a screen effect; the data is still sent as clear text to the server when the form is posted! The only way around this is to have an SSL certificate installed on the server and for the link to be secure.
- **<input type="submit" name="save" value="Save!">** The Save button works along with the <FORM> it resides in. When the Submit button is clicked, all form values that are inside of the <FORM></FORM> tags are sent to the server for processing. These are made available to your Web Connection application as a series of name/value pairs. Web Connection functions such as request.form() are used to parse this list of name/value pairs to return individual posted values. The "value" attribute here is used to set the label text.
- **<input type="reset">** This simply resets all fields within the current <FORM></FORM> tags to the values they had when the page was first displayed.
- **<input type="radio" value="radioValue" name="default">** This tag would be repeated for the number of buttons needed. The name attribute ties them all together. So if there were three radio buttons all with the same name, when the form is submitted, the value of the one that is chosen is what gets sent to the server. The same technique shown earlier for the checkbox object can be used here to pre-select a button when editing data. The following expression can be used on each radio button to determine which one is pre-selected.

  ```
  <%=iif(cpaymethod="I","checked","")%>
  ```

  If the radio buttons were used to select the method of payment (Invoice, Credit card, or Purchase order), each radio button could check to see whether the stored value matched its value. For example, the radio button for selecting Invoice could be:

  ```
  <input type="radio" value="I" name="cpaymethod"
  <%=iif(cpaymethod="I","checked","")%>>
  ```

  Each of the other radio buttons would check to see whether its cpaymethod was equal to either "C" for Credit card or "P" for Purchase order.
- **<input type="file" name="fileGetterName" size="16">** This is a powerful form object that allows users to upload local files to the server. This is useful if you want your users to be able to upload attachments such as Word DOC files or pictures such as JPG files. This object creates a Browse button that allows the user to navigate to a file on the local network. When the file is chosen, its full path is shown in the textbox object. Forms that allow this type of upload must use the enctype="multipart/form-data" attribute in the <FORM> tag! This embeds the contents of the file along with the other form content during the POSTing process. If this encoding is used, instead of using request.form() to retrieve form values, request.GetMultipartFormVar() must be used. To retrieve the embedded file itself, the Web Connection function

request.GetMultiPartFile() is used. The file is returned as a string and can then be saved to disk with strtofile() or stored in a memo field.

- **<textarea name="mnotes" cols="40" rows="4"><%=mnotes%></textarea>** The textarea object is equivalent to the editbox in Visual FoxPro. This is used to enter in freeform text for input into a memo field, for example. This is different syntax from the textbox object shown earlier. TEXTAREA requires a closing </TEXTAREA> tag. Instead of having a VALUE attribute, the value that is to be displayed appears *between* the <TEXTAREA> and </TEXTAREA> tags. So in the example shown here, the value of the memo field "mnotes" is displayed using the ASP expression <%=mnotes%>.
  - **cols="40" rows="4":** This simply sets the size of the text area.
  - **Name="mnotes":** This serves to name the field when submitted to the server and allows for manipulation through scripting.

In file "edittask.wc" on server	As browser sees it
<textarea name="mnotes" cols="40" rows="4"><%=mnotes%></textarea>	<textarea name="mnotes" cols="40" rows="4">Create a form that allows for the maintenance of users.</textarea>

- **<Select></Select>** This object is used to create a dropdown list box. The example from Figure 3 has hard-coded values in the dropdown.

```
<select name="nPriority" size="1" >
    <option value="1">1</option>
    <option selected value="2">2</option>
    <option value="3">3</option>
</select>
```

  - **name="nPriority":** Names the matching value for submission to the server and for scripting.
  - **Size="1":** Determines the height of the dropdown. A value of 1 makes it a dropdown. A value greater than 1 makes it a scrolling list box.
  - **<option value="1">1</option>:** The <option> elements create the rows in the dropdown. The value is what is returned to the server. The string appearing between the <option></option> tags is what is displayed to the user. This string does not necessarily need to match the value attribute for the option, but instead can consist of more readable text. Adding "selected" after "option" in the opening tag will show the dropdown with this option pre-selected.

The preceding example shows a hard-coded dropdown. Quite often, however, in a Web Connection application you will want to have the values pulled from a table dynamically. There is a class definition in the wwDBFPopup program file called wwDBFPopup. This

allows the current open cursor to be used to populate a dropdown. In the TODO application shown in Chapter 12, this function is used to generate a dropdown of users and statuses.

```
SELECT NAME ;
  FROM (THIS.cdatapath+"users") ;
 ORDER BY cusername ;
  INTO CURSOR tnames

   lopopup=CREATEOBJECT("wwDBFPopup")

   lopopup.cKeyValueExpression="cusername"
   lopopup.cDisplayExpression="tnames.cusername"
   lopopup.cFormVarName="userfilter"

   lopopup.caddfirstitem = "No Filter"
   lopopup.cselecteddisplayvalue = THIS.cUserFilter
   lopopup.BuildList()

   cPeople = lopopup.GetOutput()
```

- The property cKeyValueExpression determines the value that will be used for the value="x" attribute.
- The property cDisplayExpression" is the expression that will be inserted between the <option></option> tags. This is what the user will see.
- The property cFormVarname is what the name attribute is set to.
- The property CaddFirstitem is optional. This can be used to show a message such as "Select an item…" as the first element in the list.
- The property cSelectedDisplayValue can be used to pre-select an item in the dropdown. This would be used if a value is already stored in a table and an edit screen is being presented to the user. For example, the current value of the dropdown would need to be shown instead of "Select an item…"
- The BuildList() method then creates the HTML code and the GetOutput() method returns the HTML code as a string. This variable can then be referenced using an ASP tag like <%=cPeople%> in a Web Connection template page. The HTML code generated by wwDBFPopup will be inserted into that location!

Scripting the Web page

JavaScript is a huge topic that is beyond the scope of this chapter. The book *The JavaScript Bible*, by Danny Goodman, is one of the best references on JavaScript. It is listed in the "Chapter resources" section.

This chapter should serve as a good introduction to the topic and will make you aware of some of the things that can be done with JavaScript.

In order to perform form validations on the client and to generate and manipulate HTML dynamically, the developer must also know how to write scripts. The most widely used

language for writing scripts is JavaScript. The Microsoft browsers support two scripting languages, VBScript and JavaScript (or JScript), while all modern browsers understand JavaScript.

Back in 1995, Netscape needed a new language to allow for easy integration of Web pages with its new support for Java. Brendan Eich was charged with developing a new language to fill this need. The original name given to the new language was "LiveScript." This reflected its ability to interact with the Web page. However, a marketing decision led to the name being changed to JavaScript. It was deemed an easier language to learn than Java and did not need a complicated IDE to use. Until the most recent versions of IE and Netscape, writing cross-browser JavaScript was a nightmare. Lots of code bracketing was needed to test for what browser version was being run. Now with IE 5.5 and Netscape 6.01, much of the language works unchanged across these browsers.

The ASP style expressions we have seen so far are executed on the server. JavaScript is executed in the browser. It is the browser that has the built-in interpreter for the JavaScript language.

Why use client-side scripting?

Many of the functions that can be done with JavaScript on the client can be done on the server as well. Many forms when submitted perform validations on the server, and an error page is sent back if there are any problems. However, doing the validations on the client is a lot more interactive. There are no delays in processing, no blinking screens.

Okay, so how do I do simple form validations?

In order to validate fields on a form, there must be a place where the script code can sit until called. The place for form validation functions is in the <HEAD> section of the Web page. Here is a simple example to illustrate how validations are performed:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>Todo Validation Example</title>
<script type="text/javascript">
function checkform() {
  if (document.edittask.ctask.value=="")
     {alert("Task field can not be blank!"); document.edittask.ctask.focus();
return false}
}
</script>
</head>

<body>
<form action="savetask.tsk" name="edittask" method="post" onsubmit="return
checkform()">
<table>
   <tr>
     <td>Task:</td>
     <td><input type="text" value="" name="ctask"></td>
   </tr>

   <tr>
```

```
      <td><input type="submit" value="Save!"></td>
      <td></td>
    </tr>
</table>

</form>
</body>
</html>
```

The JavaScript function checkform() lives in the <head></head> section of the Web page. The browser must be told that what follows is JavaScript, so the code is enclosed in the <script></script> tags. The checkform() function is invoked when the form's Submit button is clicked. The onsubmit event handler specified in the <Form> tag traps this event. This event handler will execute the checkform() function before the form is actually sent to the server. If the function returns a TRUE, the form is sent on its way. If the validation fails, a message box is displayed warning the user that a field is empty. After the alert message box is displayed, the ctask text box get focus by using its focus() method. The notation used to specify an object is very similar to the way it is done in a Visual FoxPro form. Instead of document.edittask.ctask.value, in a Visual FoxPro form it would be thisform.ctask.value. **Figure 4** will help you visualize how the DOM in a Web page corresponds to the DOM in a Visual FoxPro form

Comparing the DOM's between a web page and Visual FoxPro

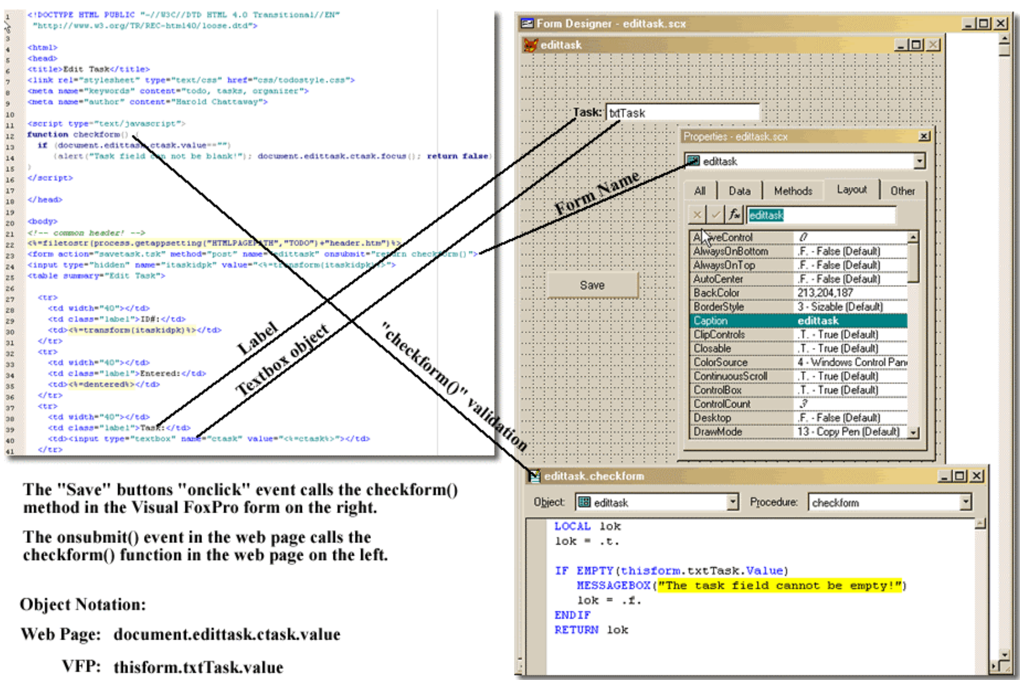

***Figure 4**. Comparison of Web page and Visual FoxPro DOMs.*

Here are some key points to keep in mind when coding in JavaScript:

- JavaScript is case-sensitive! Make sure all variable names match case in all locations. In the preceding example, the name of the text box is ctask, so the reference in the function checkform() has to be ctask also. cTask would not work!
- The expression being evaluated after an IF statement must be enclosed in parentheses.
- If multiple lines of code are being executed like in the preceding IF statement, they must be grouped together by braces {}.
- The code that makes up a function must also be grouped together by braces {}.

JavaScript caution

Even if you include client-side validation using JavaScript, you should also validate all important data on the server. This is because: (1) some browsers either don't support JavaScript or allow users to turn JavaScript support off; and (2) client-side validation can be easily hacked. Thus you should always double-check the validity of data at the server. Your client-side efforts are not wasted, however, as they do improve the user experience and reduce the load on the server.

JavaScript libraries

After working with JavaScript for a while, you will most likely develop libraries of functions that will need to be used across multiple pages. A great source for JavaScript examples is **www.brainjar.com**. One of the examples of DHTML that is shown on the site is a dropdown menu system that was written using JavaScript and style sheets. The list of functions that are needed to run the menu system could be put into a JavaScript library file and referenced on each page that uses the menu. This can be done like this:

```
<script src="mainmenu/menufunctions.js"></script>
```

This also goes in the <HEAD> section of the Web page. This has the effect of pulling in all the code from the referenced JavaScript (js) file so that the Web page can make use of the included functions. This is very much like the SET PROCEDURE TO command in Visual FoxPro. This menu system provides the Web page with a Windows-type menu system. One of the goals of Web-based applications is to not make the user learn a different way of using the application. The Web application should have the same type of interface that users are accustomed to using. The Web is a different delivery mechanism for applications, but it does not have to be whole new user interface experience. **Figure 5** shows this JavaScript menu system.

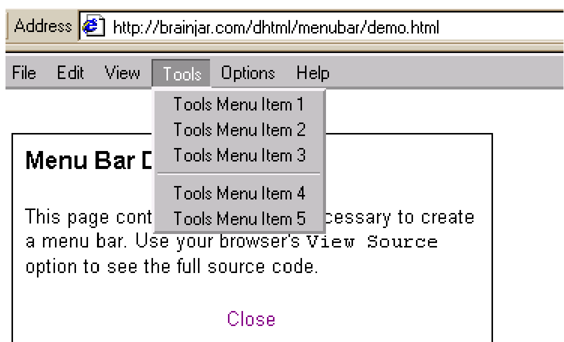

***Figure 5**. Dropdown JavaScript menu system.*

When using a menu system like this in Web page, a common complaint from users is, "When I scroll down the page, I have to scroll back up to the top to choose another menu item—that's too much work!" Well, one of the cool things that JavaScript allows you to do is to dynamically change the form in response to user actions. The menu bar is positioned with absolute coordinates. There is a property "top" that specifies the top position of the menu bar. Using the form's onscroll event, anytime the user scrolls the form, the new top position of the menu system is reset and the menu is always positioned at the top of the visible Web page! This is a very cool and very practical use of JavaScript. An example of this is included in the source code for this book, available at **www.hentzenwerke.com**. **Table 1** summarizes some key events that occur on a Web page. For a complete list of available events, consult *The JavaScript Bible*, mentioned in the "Chapter resources" section. These can be trapped, and script can be written to handle the event further.

***Table 1**. Some key Web page events and Visual FoxPro equivalents.*

Event	Description	Closest Visual FoxPro analogy
Onblur	Occurs when an object loses focus.	Lostfocus()
Onchange	Occurs when the contents of an object change.	Interactivechange()
Onclick	Occurs when the left mouse button is clicked.	Click()
Onfocus()	Occurs when an object receives focus.	Gotfocus()
Onload	Fires when an object is loaded. Frequently used in the <FORM> tag to call a script after a form has fully loaded in the browser.	Form init()
Onscroll	Occurs when the user scrolls the Web page. Used in preceding example to dynamically move the main menu.	Form's Scrolled event
Onsubmit	Occurs just before a form is submitted to the server. Used primarily for form validation scripting. If the function returns TRUE, the submission proceeds; otherwise, the form is not submitted. Use an alert() message box to notify the user of validation problems.	Click event of a form's Save button.

All scripting for these events must be very thoroughly tested across browsers! Browser compatibility is a tricky area and must be tested. See the listings for *The JavaScript Bible* and the Microsoft DHTML reference in the "Chapter resources" section.

Style sheets: Why you need them

Life without style sheets is not pretty, literally and figuratively. Most beginners making their first Web page will use the toolbar in their page designer to apply formatting to text on the page. Using this technique, the presentation tags are embedded in the page and wrap around the text they affect. Here is a common example: Let's say you are designing a maintenance form for users. As discussed earlier, you would probably have a table with two columns. The first column holds the labels, and the second column holds form objects such as text boxes. You decide that all labels should be red on the form. So, naturally, you swipe the mouse to select it, and choose red from the Text icon on the toolbar and make the selected text red. Great! That was easy! Now you go ahead and create an entire site using this same technique. A total of 20 forms are created, all with red labels created in the same way as the first. The following snippet of HTML shows the code that this technique produces:

```
<font color="#FF0000">Lastname:</font>
```

The <font> tag wraps around the text with a color attribute specifying the color to be used. Now, your first critic (your manager) looks at the pages and thinks the red color is a bit too harsh on the eyes and want you to use blue instead. If you're thinking you can simply change a base class somewhere like you can in Visual FoxPro, you are wrong! You would need to open up each Web page and manually change the color attribute! Or a slightly better solution would be to use a global search and replace. But these techniques are pretty archaic by today's standards. What is needed is a way of separating what's on the page from how it looks. This is where style sheets come into the picture! The best solution is to have definitions of how various pieces of the Web page should look in a separate file and have each Web page reference this library of definitions. There is a very simple way of doing this. In the <HEAD> section of each page, insert a reference like this:

```
<link rel="stylesheet" type="text/css" href="css/todostyle.css">
```

This line instructs the browser, upon loading the page, to retrieve the file todostyle.css in the css directory below the site root, and use the styles defined in it when they are referenced in the page. The HREF could also point to a style sheet that resides at another URL. For example, it could point to www.mycompany.com/stylesheets/todostyle.css. So why use style sheets? Well, in order to change the look of your entire site, all that you need to do is update the style definitions in this one file. The next time a page is loaded that references this style sheet, it will inherit the new definitions. No editing of individual pages is required! This is the closest you will get to the convenience of creating base classes in the Visual FoxPro form designer.

How style sheets are structured

Style sheet files are stored in plain text fields with an extension of .CSS for cascading style sheets (more on this later). You can use something as simple as Notepad. What is highly recommended, though, is a tool called "TopStyle" (see the "Chapter resources" section). This provides an IDE designed just for style sheets. It allows the user to pick style properties from a menu, formats the sheet, and provides some very powerful reports.

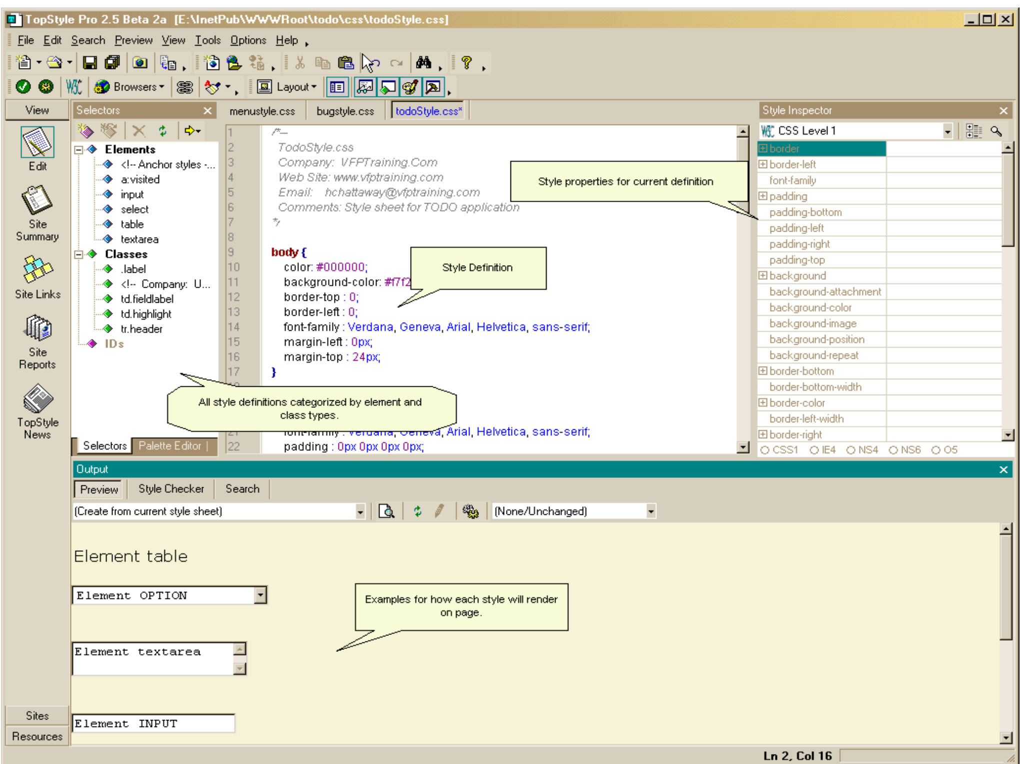

***Figure 6**. The TopStyle style sheet editor.*

Figure 6 shows the IDE for TopStyle. It makes creating correct style sheets a breeze. The Style Inspector in the right panel shows all available properties for each style definition you are creating. The left panel organizes all the definitions into either element or class definitions. And the bottom panel shows a sample Web page with the current style applied to it. Some of the most helpful features of this product are the reports it provides:

- *Orphaned classes:* This report displays style sheets containing classes that are not used in any HTML documents that reference this style sheet. It's a great report for cleaning up the style sheet of dead definitions.
- *Undefined classes:* This report shows all classes that are used in an HTML page that are not defined in the referenced style sheets.

- *Class usage:* This report shows each defined class and where it is used in the entire Web site. It displays a list of files, and you can double-click on a file name to open the file for editing.

It is best to start off in TopStyle by creating a "site." A site is like a project file in Visual FoxPro. You specify the root folder, and TopStyle is then able to create reports linking the pages with all the style sheets that the pages reference.

Element definitions

Element definitions are style definitions that act on the pre-defined HTML form element tags. For example, all pages that reference the todostyle.css style sheet as shown in the Figure 6 would use the style definition for the BODY tag automatically. The designer would not have to reference this style definition explicitly. If you wanted all your TEXTAREAs to have a consistent appearance, you could use a definition like this one:

```
textarea {
  font-size: 10pt;
  font-family: "Courier New"
}
```

The format for a style definition is shown in **Figure 7**.

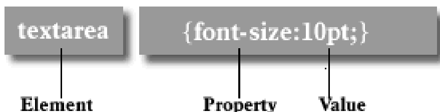

***Figure 7**. Style definition.*

The property and value must be separated by a colon (:), and the property/value pair must be enclosed in brackets {}. A semicolon separates each property/value pair. The list of property/value pairs can all be on one line or on separate lines as shown earlier. TopStyle can automatically format a style sheet file to be on multiple lines. This does make it much easier to read. By having an element definition for all the basic form elements, you can be assured that all of these objects will look the same on all of the pages that reference the common style sheet. A page can have a reference to more than one style sheet, but there should be only one style sheet that defines the standard look and feel for all pages. The style sheet included for the sample "TODO" application in Chapter 12 could be used as a template for other projects as well.

By specifying an element style for all the form elements, all objects throughout the entire site will look exactly the same. It is a good idea, for example, to have all text boxes and labels use the same font, color, and size. For example, there is a definition for the <SELECT> object in the todostyle.css sheet that defines all occurrences to be of the same width.

```
select {
  font-size: 10pt;
  font-family: "Courier New";
```

```
  width: 200px
}
```

This definition states that all <SELECT> objects or dropdowns will be 200 pixels wide, as well set setting the font to 10 point Courier New.

One very interesting use of style sheets is for users to choose their own "themes." With a common set of style definitions, this becomes quite easy. The background color for each page can be controlled by creating a definition for the <BODY> tag. For example, if the following definition is used for the <BODY> tag, it becomes easy to change the background of each page in the entire Web site based on a theme. The background-color property controls the page color for any page that references the todostyle.css sheet.

```
body {
  color: #000000;
  background-color: #f7f2d0;
  border-top : 0;
  border-left : 0;
  font-family : Verdana, Geneva, Arial, Helvetica, sans-serif;
  margin-left : 0px;
  margin-top : 24px;
}
```

On a user profile form, you could provide a dropdown with some pre-defined themes. They could simply be choosing the background color for all of the forms, for example. This could then be saved in the user's profile, and when the pages are generated for that user, an ASP tag with the following expression could be used:

```
<link rel="stylesheet" type="text/css" href="css/<%=process.ctheme%>">
```

Assuming a user needs to log in first, the login routine could look up the chosen theme in the user profile, store it to a session variable, and, when each form is rendered, the current theme name could be built into the prior link dynamically—pretty cool! In this example, what would change between each theme is the value of the background-color property. Remember, the preceding expression is evaluated on the server when the process.ctheme property is in scope. So by the time the browser sees the form, it is hard-coded. The name of the theme that is chosen would correspond to the name of the style sheet. For example, if there was a theme called "Earth Tones," the file on disk could be called earthtones.css. It is this file name that would appear where the <%=process.ctheme%> expression sits in the preceding code line. There would be a separate style sheet for each theme.

Font control

Since Web sites can be run on any platform anywhere in the world, the developer has little control over what fonts will be installed on the client machine. It is important that the site developer use the font-family property correctly so the page content will render properly on the client machine. As in the preceding example, multiple fonts can be listed on the font-family line. If Verdana is not installed on the client machine, the next font, Geneva, will be used. This will continue down the list until a font is found that is installed on the machine. If none of the named fonts are installed the machine, the browser on the client machine will use a

Sans Serif font to render the text. This is the last entry in the comma-delimited list. Serif fonts are proportional and have “serifs,” or the tiny decorations that appear at the end of the main strokes in letters (see **Figure 8**).

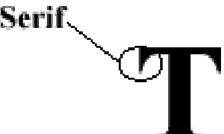

***Figure 8**. Serif on Times New Roman.*

Sans Serif fonts are fonts that do not have these tiny decorations; examples include Helvetica and Geneva. By being able to specify what font families to use, the developer retains some control over how the Web page will be rendered. If the font-family property is not used, the page could come out looking very different from what you intended!

If any object within the BODY needs a different font, the font-family property can be used on specific objects and that will override the definition in the BODY style.

Class definitions

Class definitions can be used for non-element tags. In our first example, we wanted to create a label that has a common site-wide definition. So instead of our HTML looking like this:

```
<td>font color="#FF0000">Lastname:</font></td>
```

it should look like this:

```
<td class="label">Lastname:</td>
```

Then, in the referenced style sheet, we have the following definition:

```
.label {
  font-family : Verdana, Geneva, Arial, Helvetica, sans-serif;
  color : Blue;
  font-weight : bold;
  text-align : left;
}
```

Now, any time you want to create a label with the standard pre-defined look, you simply use the class attribute and specify the label style definition. In almost all cases, the label will be in its own table cell, so the class reference is enclosed inside of the <TD> tag and will then affect anything within that element. Just to be clear, the class names should be pretty generic. By this I mean that the name of this definition should not be “bluelabel,” for example. Naming it just “label” makes it very easy to simply change the style definition to make it any color you like. This allows for *very* easy site maintenance. Updates to the interface become trivial if the right class definitions are defined in an external style sheet. **Figure 9** shows an example of the todostyle.css style sheet and how it affects the look of the edittask Web page from the “TODO” application in Chapter 12.

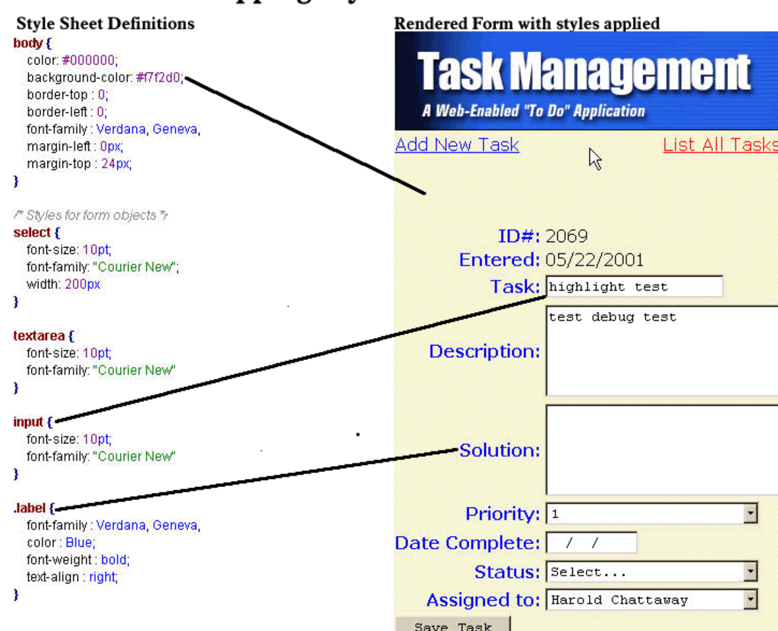

Figure 9*. Mapping style sheet definitions to a rendered form.*

Pseudo-class selectors

There are other types of style definitions that are implied. For example, the <A> tag or anchor tag is used to define an embedded hyperlink. A common need is that when a user moves the mouse over the link, something about it should change to let the user know it is a link. Very often, it is given an underline or the background color changes. Let's see how we can do the latter. In our todostyle.css style sheet, there is a definition that looks like this:

```
/*-- Anchor styles -*/
a:hover {
  color: Red;
  text-decoration: underline;
  background : Yellow;
}
```

There is a pseudo-class selector available for the anchor tag called "hover." This, if defined, is used when a user hovers over a link. Since this is an element definition, it will be used wherever the <A> tag is used automatically. And since the hover class is pre-defined for the anchor tag, this does not have to be explicitly defined either. It will be used automatically when a user moves the mouse over the link. This definition will make the text red and the background yellow. **Table 2** lists the pseudo-class selectors for the anchor tag.

***Table 2**. The pseudo-class selectors for the anchor tag.*

Name	Description
:visited	Any link that has already been clicked on.
:hover	A link that is currently being moused over.
:active	Any link that is about to be clicked.

At times it may be necessary to make special versions of element definitions. For example, let's say there is a frame on the left side of the Web page that will hold the site menu, and the background color is yellow. If the preceding a:hover definition was used on those links, the user would never see the hover effect since the hover background color and the frame background color are the same. So another style sheet definition is needed:

```
/*-- Anchor styles -*/
a.menu {
  color: blue;
}

a.menu:hover {
  color: black;
  text-decoration: underline;
  background : red;
}
```

This definition will make the text black and the background color red when the user mouses over the link. Since this definition has a name to it, it would not be used automatically by the links in the menu frame. In order to reference it, we need to use the class attribute.

```
<a href="edituser.wp" class="menu">Edit Users</a>
```

This technique of specifying the element tag, a period, and then a name can be used on any element tag to create specific versions of styles for each element.

So how do style sheets "cascade"?

The term "cascading style sheets" applies to how various levels of style definitions are applied to the final rendering. The closer a style definition is specified in a Web page, the more weight it has. For example, the definition for the element INPUT could appear in one of three places.

- It could appear in an externally linked style sheet as :

```
input {
  font-size: 10pt;
  font-family: "Courier New"
}
```

- It could appear as an embedded style definition anywhere in the page prior to use as:

```
<style>
input {
  font-size: 10pt;
```

```
  font-family: "Courier New"
}
</style>
```

- Or it can appear as a STYLE definition within the element tag:

```
<input type="textbox" style="font-size: 10pt; font-family: 'Courier New'"
value="">
```

The third variation overrides the second, and the second would override the first. So it is possible to have a site-wide style sheet that is referenced by the LINK tag in the header, but to have the style that is used for rendering cascade down to the definition used either at the page level (as in the second version) or at the element level (as in the third version).

Finally…

So, after all the form elements have been added to the page, it is time to make sure the page has all of the matching closing tags. For example, the body of the Web page started off with the <BODY> tag. There must be a matching closing body tag (</BODY>) to complete this section. Similarly there must be a closing form tag (</FORM>) and a closing HTML tag (</HTML>). This is just like having to match parentheses in Visual FoxPro code with nested functions.

Conclusion

The topic of HTML and style sheets is probably what will cause the new Web developer the most frustration. Even as deep as Web Connection is, it is still all Visual FoxPro and most Fox developers should be able to get through it. HTML, JavaScript, and style sheets, however, are all new and not intuitive at first. If you start off using style sheets from the beginning and are aware of what they and JavaScript can do, you are off to a great start. Going back and retrofitting a Web page with style sheets is a very painful process. Using them correctly from the start, however, will make your work go much faster and make site maintenance much easier as well. If you find yourself working for days on a single page to get it right, it will become incredibly frustrating. Visual FoxPro developers are used to constructing pages rapidly, and using HTML may seem like you are going back to FoxBase days! Study the concepts in this chapter and read the resource material, and it will be much easier.

Chapter resources

- **www.webconnectiontraining.com:** Author-run site with updates to code, videos, and more examples for developing in Web Connection.

- HTML-Kit home page: **www.chami.com/html-kit**

- Dreamweaver home page: **www.macromedia.com/software/dreamweaver/**

- TopStyle Style Sheet Editor: **www.bradsoft.com**

- Online HTML Validator: **http://validator.w3.org/**

- **http://msdn.microsoft.com/library/default.asp?url=/workshop/author/html/reference/elements.asp**: Microsoft's online guide to all HTML objects.

- **http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dom/domoverview.asp**: Excellent MSDN online resource for DHTML overview. Lists all properties, events, and methods with explanations of each.

- **www.w3.org/TR/html4/**: World Wide Web Consortium standards document for HTML 4.01.

- **www.w3.org/TR/REC-CSS2**: Cascading Style Sheet level 2 Standard.

- *The JavaScript Bible*, by Danny Goodman: **www.amazon.com/exec/obidos/ASIN/0764533428/ref=bxgy_sr_text_a/104-4309761-0938307**

- *Cascading Style Sheets: The Definitive Guide*, by Eric Moyer: Published by O'Reilly Press, ISBN 1-56592-622-6.

- HTML Table Specification: **www.ietf.org/rfc/rfc1942.txt**

- Web Accessibilities Policy: **www.w3.org/WAI/Policy/#255**