CHAPTER 3

HTML FORMS AND JAVASCRIPT BASICS

As you can imagine, using HTML forms to collect user data is an integral part of most any Web application. This lesson first explores the HTML elements and attributes responsible for creating HTML form elements. Located conveniently at the beginning of this lesson (sounds like a sales pitch), Section 3.1 will prove valuable in a reference capacity as you construct HTML forms throughout rest of this text. The remainder of this lesson is dedicated to an exploration of the JavaScript programming language. JavaScript can accomplish many things on the front end of Web applications, but we focus on using it to manipulate user data entered into HTML forms. We explore both creating client-side processing utilities and validating user data before submitting it to a CGI program the Web server.

3.1 HTML Forms

We title this section "HTML Forms" but we continue to adhere to XHTML syntax rules. The `form` element is merely a container that holds the *form elements*.

`<form>` *`form elements go here`* `</form>`

From a pure markup perspective, it behaves just like the paragraph block element. For example,

```
In terms of <form>pure markup</form> I am a paragraph block element.
```

only causes paragraph breaks before and after its contents.

```
In terms of

pure markup

I am a paragraph block element.
```

The `form` element is summarized in the following table. Primarily, its attributes are responsible for where and how the actual data carried by the form elements are submitted for processing.

`<form > …</form>      Block Element`

ATTRIBUTE	POSSIBLE VALUES	DEFAULT
`action`	`URL` (relative or absolute)	
description:	Specifies the URL of server-side program that is to receive data from the form's elements.	
`method`	`get, post`	`get`
description:	Specifies one of the two ways that the form's data can be sent to a server-side program. (Recall the discussion about `GET` and `POST` HTML transactions in Section 1.8.)	
`name`	any text string	
description:	A variable name given to the form that is used to refer to the form as a JavaScript object.	

Control Buttons

There are a variety of form elements that go inside a form. Many carry data for the form, while others provide controls that trigger the processing of the data. We first discuss the form control elements, which are just buttons that are clicked to cause something to happen.

Command Button

```
<input type="button" value="Click Me" />
```

A command button (or *generic button*) is used to call a self-defined mechanism that initiates processing of the form's data. Command buttons are used on the Web almost exclusively to call JavaScript functions. The value attribute specifies the text that appears on the button.

Submit Button

```
<input type="submit" value="Submit Form" />
```

A submit button initiates server-side processing of a form's data. They cause the data contained in a form's elements to be submitted to a CGI program located by the URL specified in the `action` attribute of the `form` container. The `value` attribute specifies the text that appears on the button.

Image Submit Button

```
<input type="image" src="images/clickme.gif" />
```

An image submit button is no more than a submit button that lets you choose your own graphic to represent the button, rather than having the browser draw the button for you. A graphic command button can take any of the attributes of the `img` element (`align`, `alt`, `hspace`, etc.) to help position it within the form block. As with other images, is a good idea to specify `height` and `width` attributes, although we have not done so here to simplify the definition.

Reset Button

```
<input type="reset" value="Reset Form" />
```

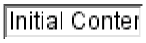
When clicked, a reset button automatically resets a form's elements to their original state. By original state, we mean how the HTML attributes instructed the form elements to be rendered. User changes to the form are wiped out. The value attribute specifies the text that appears on the button.

Text Elements

The remainder of the form elements are responsible for carrying data. These can be grouped into two categories: elements that only carry text strings and those that provide choices for the user. We begin with the four that only carry text.

Text Field

```
<input type="text" value="Initial Contents" size="13" maxlength="20" />
```



This is a single line text entry field. The `value` attribute specifies any initial contents the field is to have. The `size` attribute specifies the width in characters of the field. The default is about 20 characters. The `maxlength` attribute specifies the maximum number of characters that the field will accept. If `maxlength` exceeds `size`, you can see that some characters are not visible in the field.

Password Text Field

<input type="password" value="obscured text" size="15" maxlength="15" />

A password text field is no more than a text field, where the characters are all rendered as asterisks or discs. Astutely named, it is usually used to obtain passwords. That way, someone looking over you shoulder will not see on the screen what you typed. It uses the same attributes as a standard text field.

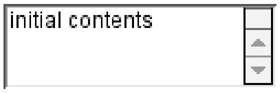
Hidden Field

```
<input type="hidden"  value="hidden data" />
```

This is a text carrying form element where absolutely nothing is rendered in the Web page. The `value` attribute carries hidden data. There are several uses for hidden information, as we shall see.

Text Area

```
<textarea cols="25" rows="3">initial contents</textarea>
```



initial contents

A text area is a multiple line text entry or display area. In contrast to a text field where the initial contents come from the value attribute, a text area gets any initial content from the contents of the `textarea` element. Text is automatically wrapped around in the text area, and scroll bars are provided so that content can exceed the height of the area.

Option-creating Elements

The remainder of the form elements provide choices for the user. The choices can be simple "on/off" (Boolean in nature), or choices that allow the user to choose from a list.

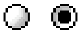
Checkbox

```
<input type="checkbox"  />
<input type="checkbox"  checked="checked" />
```

A checkbox provides an on/off choice. If two or more checkboxes are provided, the user can choose any number of them. That is , they can "check" none, one, two,…, or all of the checkboxes. By default, a checkbox is "unchecked" when it is marked up. The `checked` attribute causes a checkbox to be initially checked when it is rendered. (Before XHTML, `checked` was included as a standalone attribute without a value.)

Radio Button

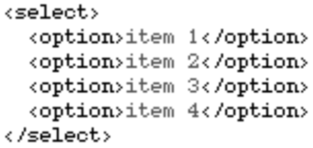
```
<input type="radio" name="group1" />
<input type="radio" name="group1" checked="checked"/>
```



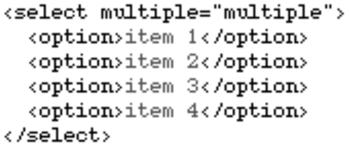
These are similar to checkboxes. However, if a group of them share the same `name`, only one in that group can be selected. This is the *unique selection property*. Using the two above for example, if we were to select the first one, the second one would automatically uncheck. The two radio buttons share the same name, so they form a *unique selection group.* Multiple unique selection groups can be formed by choosing different names, each one common only to all radio buttons in a given group.

They should not be used like checkboxes. If you click on a checked checkbox, it becomes unchecked. However, you can't un-select a radio button by clicking on it. If a radio button is part of a unique selection group, clicking another in the group will uncheck it. The only other way to un-select one is if the form is provided with a reset button (which also clears the rest of the form). In a sense, each radio button does have an on/off functionality, but they really work together to form lists of options, where only one in a given list can be selected. (They get their name from the old car radios where you selected among preset stations by pushing a button. When you did, the button that was previously selected popped out automatically. You show your age if you admit to remembering those radios!)

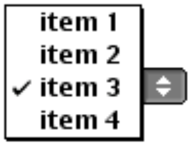
pop-up menu (pull-down menu)

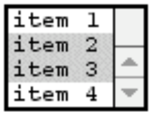


```
<select>
  <option>item 1</option>
  <option>item 2</option>
  <option>item 3</option>
  <option>item 4</option>
</select>
```



```
<select multiple="multiple">
  <option>item 1</option>
  <option>item 2</option>
  <option>item 3</option>
  <option>item 4</option>
</select>
```





A pop-up menu, sometimes called a *pull-down menu*, organizes a list of items from which the user can make a selection(s). The menu on the left, the default variety, allows the user to make only one selection. It is a *single selection menu.* A single selection menu basically functions like a unique selection group of radio buttons. But in a single selection menu, the choices are hidden and only pop-up when the menu is clicked by the user. This is advantageous when the list of choices is long, and would otherwise clutter up the Web page if a group of radio buttons was used.

The menu on the right uses the `multiple="multiple"` feature which allows the user to make more than one choice. (Again, before XHTML the attribute was included as `multiple` without a value.) This type of menu does not pup up on the screen, but forms a visible *multiple selection menu*. The `multiple` attribute gives the list a functionality similar to a group of checkboxes. But with a selection list, there is a `size` attribute that limits the number of options that are displayed on the screen at a given time. Again, a selection list is used when the amount of options would clutter the Web page if checkboxes were used. Physically, multiple selections are made by using the control or command keyboard key in concert with the mouse.

Attribute Summaries

Those are all of the form elements (or form components) we will use for collecting and displaying data. You may have noticed that most of them are `input` HTML elements where the `type` attribute determines which element is created. Only two are created with different tags -- the `textarea` element and the `select` element.

In the above overviews, attributes specific to each `type` of `input` element were given. It's a strange HTML feature that one element, the `input` element, uses different attributes depending upon which `type` has been set for the element. Thus, the input element summary below only lists those attributes used by all types. Summaries for the two form elements that use their own tags follow.

`<input />      Inline Element`

ATTRIBUTE	POSSIBLE VALUES	DEFAULT

type	control buttons	`button, submit, image, reset`	`text`
	data-carrying elements	`text, password, hidden checkbox, radio`	
name		`any text string`	
value		`any text string`	

* Other attributes may be used depending upon the type of input element.

`<textarea > …</textarea>` **`Inline Element`**

ATTRIBUTE	POSSIBLE VALUES	DEFAULT
cols	`integer`	browser specific
rows	`integer`	browser specific
name	`text string`	

* Integer values for columns and rows represents text characters, not pixels.

`<select > …</select>` **`Inline Element`**

ATTRIBUTE	POSSIBLE VALUES	DEFAULT
multiple	`multiple="multiple"`	single selection menu
description:	Allows the user to make multiple selections from the menu. In this case, the menu is displayed as a selection list, rather then a pop-up menu.	
size	integer	1
description:	Specifies the number of items to be displayed. If set to a number greater than 1, the menu is displayed as a selection list, rather then a pop-up menu. Commonly used to limit the number of visible options in a selection list (multiple selection menu).	
name	any text string	

`<option >…</option>` Used only inside the `select` element.

ATTRIBUTES	POSSIBLE VALUES	DEFAULT
selected	`selected="selected"`	no items are selected
description:	Inclusion of this attribute causes an option to be initially selected. If `multiple` is enabled in the `select` element, more than one option can be initially selected.	
value	any text string	

* This is not a form element per se, but an element used to create a list of options within the `select` form element.

NOTE

You may have noticed that all form elements have a `name` attribute. The `name` attribute is used to identify a form element. In contrast, the `value` attributes of form elements carry the actual data. The "`name` for identification, `value` for data" principle is used for two different purposes.

First, form data is organized into *name-value pairs* before it is submitted to a Web server. For example, when the text field

```
<input type="text"  name="address" value="" />
```

is submitted to the server, it's data is summarized with the name-value pair

```
 address="1234 Maple Ave."
```

where the text string carrying the actual address is what the user enters into the field. The name provides identification back on the server as to what piece of data has been submitted.

The second use is for form processing on the client, where names and values are treated differently. The `name` attribute is used to provide reference to a particular `value` attribute. In other words, the form's data is contained in a bunch of attributes, all called `value,` and you need to reference a particular one. The `name` attribute gives you that specific reference. For example, for the above text field, `address.value` is a JavaScript variable containing whatever text string the user enters into the text field.

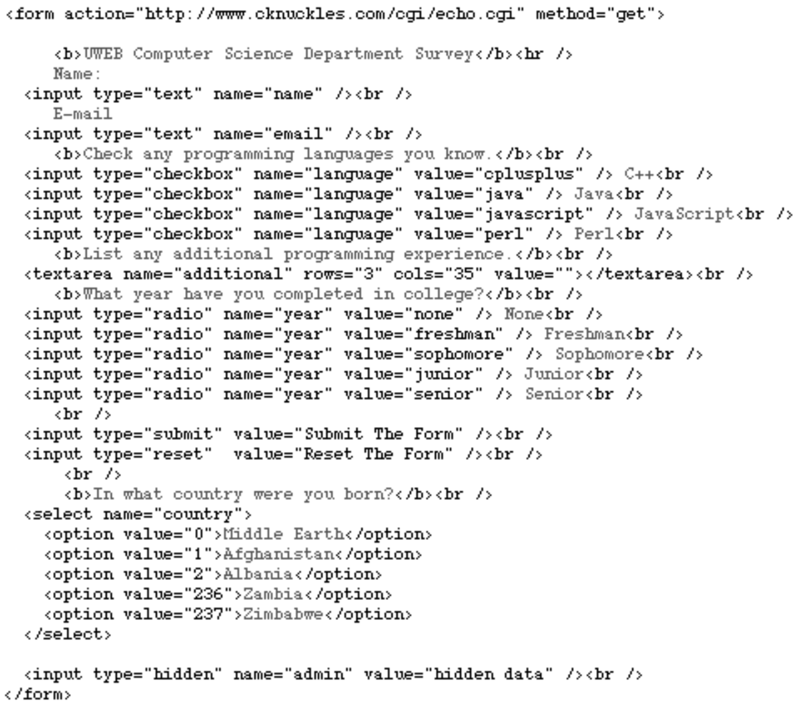
A Form Submission Example

We conclude this section with an example that uses many of the form elements we have introduced. The example serves two purposes. It shows you the form elements put together into a large form, and it shows how name-value pairs are submitted to the server. Figure 3.1 shows an HTML form that simulates an online survey that UWEB's computer science department might use to collect information from transfer students.

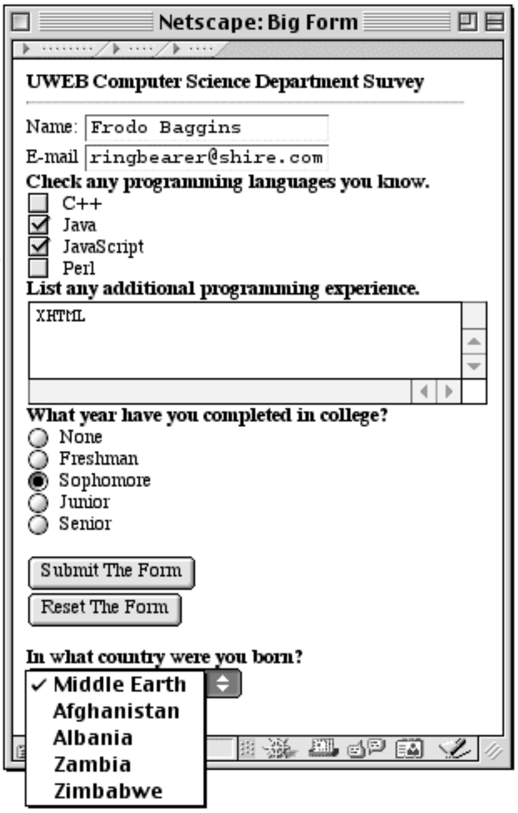
The first thing to note is that the value of the `action` attribute of the `form` element specifies the URL of a server-side CGI program to which the form's data is to be submitted. The program, `echo.cgi`, is a special program we have prepared that does nothing more than send the data submitted by the form back to the browser in a Web page.

The second thing to note, besides the syntax of the form elements themselves, is the names and values given to the form elements. But, neither the text fields nor text area were assigned a `value` in the HTML code. Those values are supplied by the user. The option-giving elements (checkboxes, radio buttons, and menu) have each been given `value` attributes. The user can't supply a `value` since those elements carry no mechanism to collect text from the user. Effectively, those values are hidden data corresponding to the choices given to the user.

You can assume that the hidden field at the bottom of the form definition is used to carry additional information for some administrative purpose. In the source code for Figure 3.1, we have omitted the rest of the HTML document (`head`, `body,` etc.) to save some space. In the full document, the form is contained inside the `body` element.



```
<form action="http://www.cknuckles.com/cgi/echo.cgi" method="get">

     <b>UWEB Computer Science Department Survey</b><br />
     Name:
  <input type="text" name="name" /><br />
     E-mail
  <input type="text" name="email" /><br />
     <b>Check any programming languages you know.</b><br />
  <input type="checkbox" name="language" value="cplusplus" /> C++<br />
  <input type="checkbox" name="language" value="java" /> Java<br />
  <input type="checkbox" name="language" value="javascript" /> JavaScript<br />
  <input type="checkbox" name="language" value="perl" /> Perl<br />
     <b>List any additional programming experience.</b><br />
  <textarea name="additional" rows="3" cols="35" value=""></textarea><br />
     <b>What year have you completed in college?</b><br />
  <input type="radio" name="year" value="none" /> None<br />
  <input type="radio" name="year" value="freshman" /> Freshman<br />
  <input type="radio" name="year" value="sophomore" /> Sophomore<br />
  <input type="radio" name="year" value="junior" /> Junior<br />
  <input type="radio" name="year" value="senior" /> Senior<br />
     <br />
  <input type="submit" value="Submit The Form" /><br />
  <input type="reset"  value="Reset The Form" /><br />
      <br />
      <b>In what country were you born?</b><br />
  <select name="country">
    <option value="0">Middle Earth</option>
    <option value="1">Afghanistan</option>
    <option value="2">Albania</option>
    <option value="236">Zambia</option>
    <option value="237">Zimbabwe</option>
  </select>

  <input type="hidden" name="admin" value="hidden data" /><br />
</form>
```



UWEB Computer Science Department Survey

Name: Frodo Baggins

E-mail ringbearer@shire.com

Check any programming languages you know.

□ C++
☑ Java
☑ JavaScript
□ Perl

List any additional programming experience.

XHTML

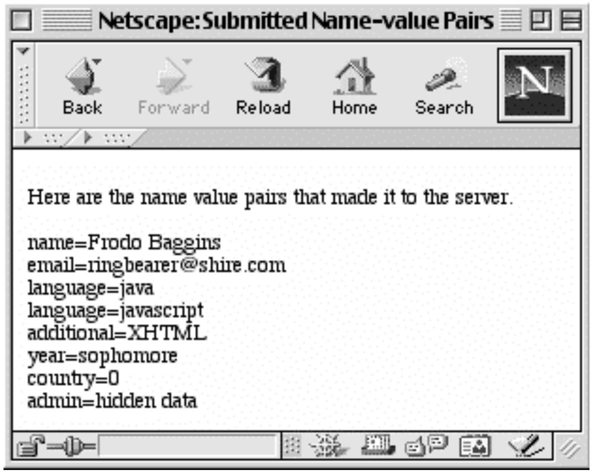
What year have you completed in college?

○ None
○ Freshman
● Sophomore
○ Junior
○ Senior

In what country were you born?

Figure 3.1 A typical HTML form

Let's suppose that the form has been filled out as shown in Figure 3.1 and Frodo hits the submit button. The output returned by the "echo" program is shown in Figure 3.2. Notice that the data is organized into `name=value` pairs, where the names are those given to the form elements in the HTML code. For the text-carrying elements, the values are the user's text entries. For the option-creating elements, the values are the "hidden values" corresponding to the user's choice.



```
Here are the name value pairs that made it to the server.

name=Frodo Baggins
email=ringbearer@shire.com
language=java
language=javascript
additional=XHTML
year=sophomore
country=0
admin=hidden data
```

Figure 3.2 The form's data sent back to the browser by the server-side "echo" program

There are two reasons why you should go to the Web site and submit the form for yourself a couple of times using your own values. The first is so that you can see first hand how the submitted data relates to the names and values hardcoded into the HTML form and to the data you enter. The second reason is so that you see how the data gets to the server. Notice that the form's `method` attribute specifies `get`. That means the data is appended onto the URL that calls the CGI program as a query string. When you try it, you will see a query string similar to the one below.

```
http://www.cknuckles.com/cgi/echo.cgi?name=Frodo+Baggins&email=ringbearer@shire
.com&language=java&language=javascript&additional=XHTML&year=sophomore&count
ry=0&admin=hidden+data
```

As you see, the name=value pairs are concatenated into one long string using ampersands, which are boldfaced here for emphasis, as the delimiting character. Web browser's do that automatically as part of the HTTP protocol. So basically all the "echo" program does on the server is split the individual pairs back out and return them in a Web page.

NOTE

While we won't formally deal with submitted form data in CGI programs until Lesson 6, the "echo" program will be helpful in the mean time. Any time you build an HTML form, simply submit it to the following URL.

`http://www.cknuckles.com/cgi/echo.cgi`

That program accepts both the GET and POST methods of form submission. The "echo" program will prove particularly handy later in this Lesson when we validate that the user has entered proper information into a form before submitting it to the server.

We shall soon see that the names of form elements take on a distinctly different role when HTML forms are processed using JavaScript on the client. In that case, the names are used as variable names in object references. The names reference the form elements in JavaScript statements, but the values still carry the form's data. That being said, let's learn some JavaScript!

3.2 The JavaScript Language

JavaScript was developed by Netscape Communications to provide client-side programming support for their Netscape browser. It reared its head onto the Web in late 1995. The language is syntactically based on Java, but is very different from Java in its functionality. (People often lump the two together. They shouldn't.) With JavaScript gaining in popularity, Microsoft soon introduced its version, called Jscript, in 1996 for use in Explorer 3. But JavaScript was too well entrenched for Microsoft to bully it off of the Web. What is used on the Web today is a common subset of JavaScript and Jscript that is platform and browser independent. It is simply called JavaScript[1].

We won't be able to deal with form data on the back end until we learn some CGI programming with Perl. However, JavaScript provides for a rich processing tool for the data right on the client. There are several useful chores that JavaScript can tackle on the front end of a Web application. Before we get to that, we look at JavaScript as a traditional programming language. If you are familiar with C++ or Java, you will find that the programming concepts you have learned are no different here. Even the syntax is similar. The transition should be seamless.

The HTML `script` element is a container whose only purpose is to contain client-side scripting instructions, JavaScript in our case. In fact, if

```
I am  <script>thinking that you are</script> uncool.
```

is included in an HTML document, it will render as

```
I am uncool.
```

(That's enough incentive to use it only as a script container.) When the HTML processor encounters the `script` element, it ignores it. It's contents are processed by a browser's scripting engine, in our case the JavaScript interpreter. It is best to include JavaScript in an HTML file as follows.

```
<script language="javascript">
<!--
  JavaScript statements
//-->
</script>
```

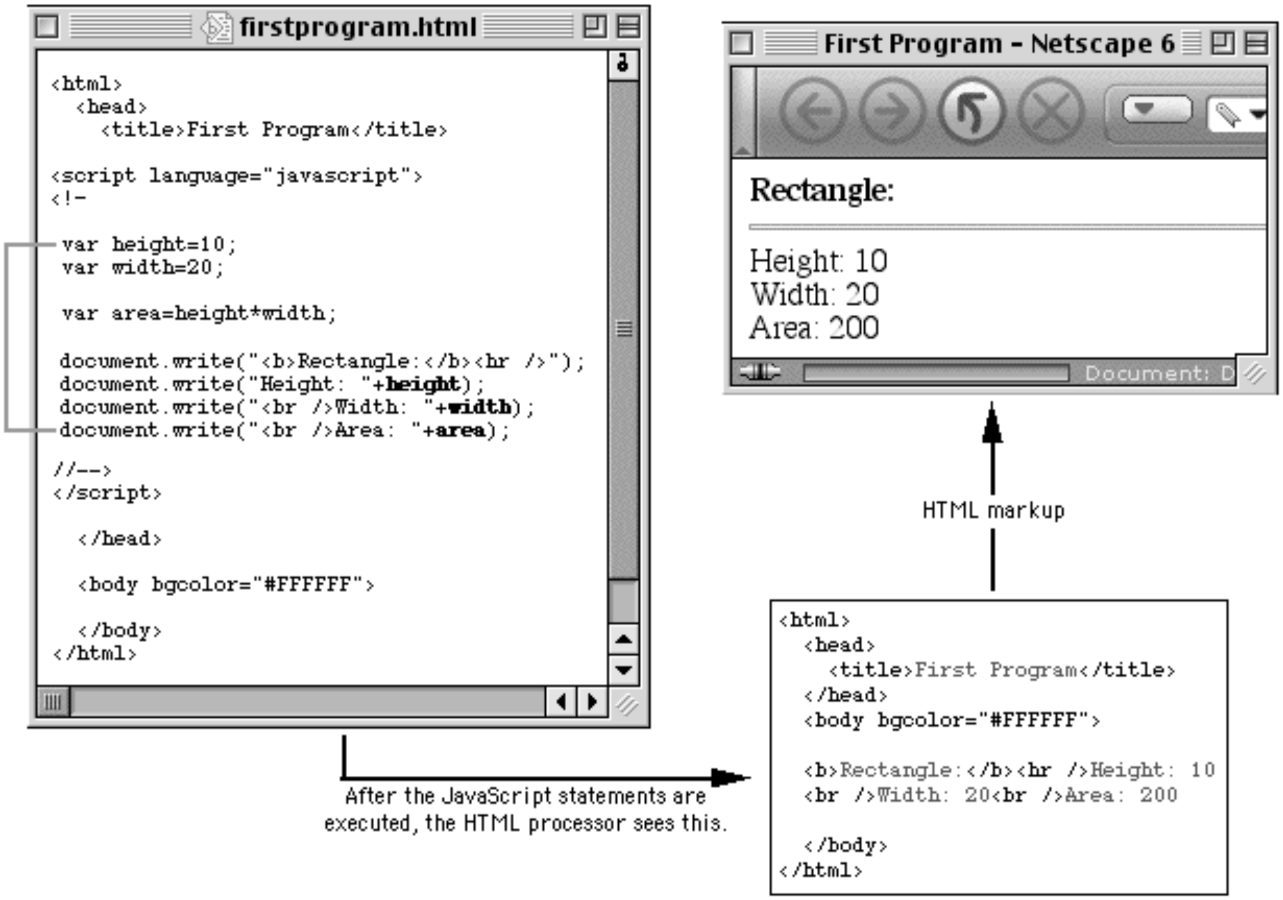
The HTML comment markers hide the JavaScript statements from older browsers that don't know to ignore the `script` element, and the two forward slashes are a JavaScript comment marker to hide the closing HTML comment marker. Since JavaScript has become ubiquitous, most browsers will still execute the statements assuming they are JavaScript even if `language="javascript"` is not included.

NOTE

[1] Similar to what W3C has done for HTML, an international standards organization has issued a core standardization for JavaScript. Technically, the current cross-platform JavaScript version is ECMA-262, or simply ECMAScript.

Your best bet is simply to make a new skeleton document containing the above construct. That way, each time you make a new Web page that uses JavaScript, you can copy the skeleton file. Theoretically, it only is necessary to type the `script` element construct with the comment symbols once.

To show you how simple the language is, we start with an example that any programmer can understand, even if they have never seen JavaScript. Figure 3.3 shows a very basic JavaScript program, together with its rendition. The first thing to note is that you just load the HTML file into a browser, and the JavaScript interpreter handles the code. JavaScript is an **interpreted language**, meaning JavaScript programs (scripts[2]) are never compiled into a platform specific executable. The same script you see in Figure 3.3, can be passed to browsers around the world to be interpreted time and time again, and on any operating system that supports a JavaScript capable browser.



```
<html>
  <head>
    <title>First Program</title>

<script language="javascript">
<!-

 var height=10;
 var width=20;

 var area=height*width;

 document.write("<b>Rectangle:</b><hr />");
 document.write("Height: "+height);
 document.write("<br />Width: "+width);
 document.write("<br />Area: "+area);

//-->
</script>

  </head>

  <body bgcolor="#FFFFFF">

  </body>
</html>
```

First Program - Netscape 6
Document: D

```
Rectangle:

Height: 10
Width: 20
Area: 200
```

HTML markup
After the JavaScript statements are executed, the HTML processor sees this.

```
<html>
  <head>
    <title>First Program</title>
  </head>
  <body bgcolor="#FFFFFF">

  <b>Rectangle:</b><hr />Height: 10
  <br />Width: 20<br />Area: 200

  </body>
</html>
```

Figure 3.3 Our first calculation in a Web page.

When used like a traditional language, many of the same principles apply. Get input from the user, store the input in variables, process the input with control structures (if…else, loops, arrays, etc.), and then give output. In Figure 3.3, the output is simply written to the HTML document using `document.write()` statements. You can see from the HTML code produced by the statements, that the quoted text strings have been written to the document "as is," and the variables were replaced by their contents. We will forgo crude methods for acquiring user input in favor of waiting until we learn enough to process input from HTML forms.

[2] By definition programs give instructions to the operating system, whereas scripts give instructions to other programs (like a browser application). We may abuse the terminology from time to time.

Don’t worry if you don't fully understand the example in Figure 3.3. As we summarize language features below, the details of that script will become clear. If you know some C++ or Java (or even Pascal or some other language) we think you will be surprised to find out that you are already familiar with the majority of the language. The power to use standard programming constructs to give processing capability to a Web page is the new concept here. Alone, HTML can't even perform the simple multiplication calculation accomplished in Figure 3.3.

Statements

A JavaScript statement is terminated with a semi-colon. It is customary to put statements on separate lines for readability. Important: If a JavaScript statement gets so long as to wrap around in the window that you are using to edit the script, let it. Do not hit return on the keyboard in the middle of a statement. In some cases that confuses the interpreter into thinking the statement is terminated.

Variables

In JavaScript variables are **loosely typed**. (That does not mean you should stretch out before typing in variable declarations.) It means any of the primary literal types, numeric, string, or Boolean are assigned to a variable with no formal type declaration. The following three statements declare a variable initializing it with a numeric literal, reassigning it a string literal, and then reassigning it a Boolean literal.

```
var x=3.14;
x="a string";
x=true;
```

No harm, no foul. That's a huge difference from *strongly-typed* languages like C++ and Java where each variable's type must be strictly defined. Also, there is no type for single characters. Characters are simply one character strings.

A variable should be declared using `var`. You can see in the first statement above that a variable can be declared and initialized in one statement. After that, it can be reassigned values of three primary types at will. Assignment of numbers and strings is straight-forward. But note that strings can be assigned using single quotes.

```
x = 'a string';
```

However the convention is to use double quotes.

A Boolean literal is not a string. For example,

```
x = "false";
```

merely stores a five character string into `x`. JavaScript reserves `true` and `false` as special unquoted Boolean literal values. Storing other non-reserved unquoted values into a variable results in an error.

```
x = oops;    //bad declaration
```

JavaScript has the same rules for variable names as most any language. They must start with an alphabetic character, can contain numeric characters (not as the first character, of course), and the only allowed special character is the underscore (_). The names, `num`, `num2`, and `num_2` are legal, but `2num`, `_num`, and `num-2` are not legal. You should also avoid the JavaScript reserved words listed in Appendix A. As in most any language, variable names are case sensitive.

In addition to the standard literal types, number, string, and Boolean, JavaScript has special literal types that it uses when calculations go bad. For example, a statement like

```
var y=2*"oops";
```

causes y to contain the literal value NaN (Not a Number). If a variable has been declared but not initialized

```
var z;
```

it is assigned the literal value undefined. Even division by 0

```
var a = 2/0;
```

results in the special literal value Infinity.

Remember, JavaScript code is completely portable, being embedded in Web pages flying around to client browsers all over the planet. Moreover, one of its main uses is processing user data while it is still on the client. Inherently, user data is unreliable in that you don't know in advance what you are going to get. Thus, it is desirable for JavaScript to generate special literals upon encountering bad calculations rather than crashing the browser, for example. When you see special literals (NaN, undefined, Infinity), you can assume that a calculation has gone bad in your script or that a variable has been left uninitialized. Also note that declared, but uninitialized variables are given the special null literal.

Operations

Operations in JavaScript are standard. For numeric quantities, there are the *arithmetic operations*

```
+  (addition)
-     (subtraction)
*     (multiplication)
/     (division)
%  (modulus)
```

which return numbers. Modulus is the remainder when one integer is divided into another. For example, x=11%3; causes x to contain the number 2.

There are *comparison operations*

```
==    (equals)
!=    (not equals)
>     (greater than)
>=    (greater than or equal to)
<     (less than or equal to)
<=    (less than or equal to)
```

which return Boolean values. For example, x=(4!=4); causes x to contain false, and x=(4==4); causes x to contain true. We have to take this opportunity to make the important note that = **is for variable assignment, and == is for equality** comparison.

Also, comparison of strings is standard, with characters inheriting a sequential ordering from their ASCII numbers. For our purposes, it is sufficient to note that string equality implies that the strings match character for character. So the expressions ("abC"=="abc") and ("bac"=="abc") both evaluate to false.

Finally there are the *logical operations*

```
&&  (and – true only if both operands are true)
||  (or – false only if both operands are false)
!   (not – negates its operand)
```

which compare Boolean expressions and return Boolean values. For example, the expression (2==3)||("abc"==abc") evaluates to true since the second operand is true, but (2==3)&&("abc"==abc") evaluates to false since the first operand is false. It never hurts to emphasize that `and` `or` are radically different. To throw in a negation, `(!(2==3))&&("abc"==abc")` evaluates to `true.`

NOTE

Rather than memorizing rules for associatively that govern the order of evaluation in expressions that mix the operator types, it's best simply to use grouping parentheses liberally to force the expression to evaluate properly.

Shortcut Assignment Operators

While = is the standard operator for variable assignment, JavaScript uses the same shortcut assignments (++, +=, etc.) as C++ and Java. The only one we will use in this text is

```
x++;
```

which has the same effect as

```
x=x+1;
```

which increments a numeric variable `x` by one.

Concatenation (duality of +)

The + operator has a duality in JavaScript (it's overloaded). In a strictly numerical context, it adds numbers. In a string context, it *concatenates* (joins together). For example, the following

```
x=3;
y=4;
z=x+y;
```

causes `z` to contain the number 7 as you would expect. However,

```
x="3";
y="4";
z=x+y;
a=y+x;
```

causes `z` to contain the string "34" and `a` to contain the string "43".

If types are mixmatched in an expression, strings win out. For example,

```
x=3;
y="4";
z=x+y;
```

causes z to contain the string "34". This duality can cause problems when dealing with data from HTML forms. All form data (that is not Boolean), is stored as strings. An HTML form doesn't care if the user enters numeric or alphabetic quantities, it just wants ASCII characters from the keyboard. If we want to add two pieces of form data, will use the `parseFloat()` function to parse the strings into (floating decimal) numbers. For example,

```
x="3";
y="4";
z= parseFloat(x)+parseFloat(y);
```

causes `z` to contain the number 7. If a string is intrinsically a number, the `parseFloat()` function returns the numeric equivalent. In contrast, `parseFloat("abc")` returns the special value `NaN`.

For a practical example, we could have put the concatenation operator to good use in the script of Figure 3.3. We simply concatenate all of the arguments of the `document.write()` statements together into one long string. The following statement produces the exact same result. Again, we have applied boldface to the variables so you can easily see where the strings begin and end.

```
document.write("<b>The following summarizes the rectangle </b><hr/>
Height: "+height+"<br />Width: "+width+"<br />Area: "+area);
```

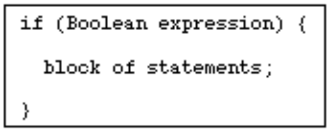
Also, we re-emphasize that in a long statement like above, we let it wrap around according to the margin. Hitting return in the middle of a statement is often problematic.

NOTE

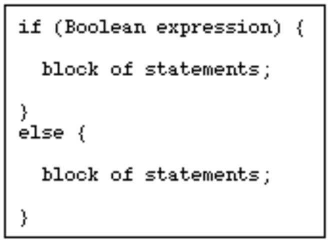
JavaScript chooses the most basic way to represent a numeric quantity. For example, if you assign 4.0 to a variable, you will get the integer 4 when you write the variable's contents. So even though the `parseFloat()` function purports to return floating point decimal numbers, you will get an integer if no decimal points are required to represent the number. For that reason, we generally stick to `parseFloat()`, but there also is a `parseInt()` function.

Conditionals

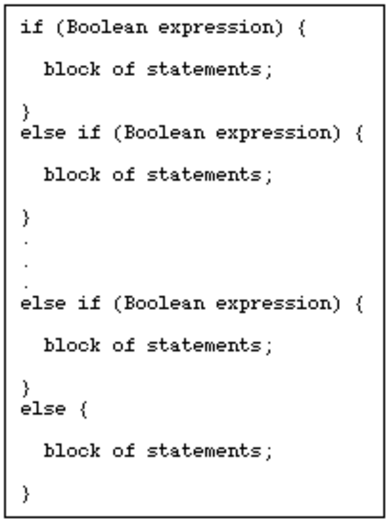
Conditional structures provide for decision making capabilities. In JavaScript their general forms are given below.



```
if (Boolean expression) {

  block of statements;

}
```



```
if (Boolean expression) {

  block of statements;

}
else {

  block of statements;

}
```



```
if (Boolean expression) {

  block of statements;

}
else if (Boolean expression) {

  block of statements;

}
.
.
.
else if (Boolean expression) {

  block of statements;

}
else {

  block of statements;

}
```

Some simple examples will help to further acclimate you to the language. We directly assign the values to the variables but, in practice they can come from user input. The following code simulates user input that is stored in the `bid` variable.

```
var bid="35";
if(bid <= 50) {
   document.write(bid+" does not meet the minimum bid.<br />");
}
else {
   document.write("Your bid of "+bid+" will be considered.<br />");
}
```

The code would results in

```
35 does not meet the minimum bid.<br />
```

being written to the body of the HTML document. The following code simulates validating user input. Assume the user input is stored in the `quantity` variable. The example uses the `isNaN()` function which returns `true` if its argument is intrinsically not a numeric quantity (i.e. if the quantity evaluates to the special variable type NaN). If the quantity is numeric, the "is Not a Number" function returns `false`.

```
var quantity="4x";
var price=19.99;
var discount=0;
var total=0;
if(isNaN(quantity)) {
   document.write("Error : non-numeric quantity");
}
else {
   if((quantity > 10) &&  (quantity <100)) {
      discount=.1;
   }
   if((quantity > 100) {
      discount=.25;
   }

   total=price*quantity*(1-discount);
   document.write("Order Total: "+total);
}
```

The code would result in the error message being written to the body of the HTML document.

NOTE

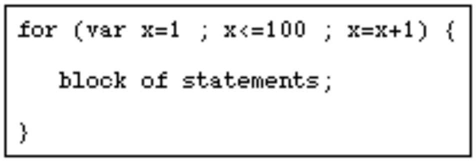
In both of the examples above, we stored the simulated user input as string data. Again, that's how **any** piece of data is stored in an HTML form `value`. You see how flexible JavaScript is. The comparison (`bid <= 50`) reduces to `("35" <= 50)` which is a problem in a strongly-typed language. However, JavaScript converts the comparison into a numeric context on the fly. The duality of + is the only thing to watch out for. The expression ("35"+50) converts to string context, and concatenates the strings into "3550".

When dealing with HTML forms, we will often use conditionals in the manner shown in the following pseudo-code. We use pseudo-code to get the logic of a programming implementation across to the reader, without having to use full syntax. The logic behind this idea is very simple.

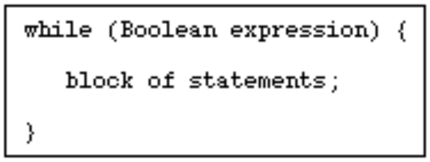
```
if (user enters bad input into form) {
   give an alert – don’t process the form data
}
else {
   process the form data
}
```

Loops

The JavaScript `for` and `while` loops are standard. Below, you see a `for` loop whose counter runs from 1 to 100, and the general form of a `while` loop.



```
for (var x=1 ; x<=100 ; x=x+1) {

   block of statements;

}
```



```
while (Boolean expression) {

   block of statements;

}
```

Since most of the looping we need for form processing in this text involves a pre-determined number of executions, we use the "for" loop in the examples below. This example writes the even numbers from 2 to 2000, each on a separate line in the Web page.

```
for (var x=2 ; x<=2000 ; x=x+2) {
   document.write(x+"<br />");
}
```

The following loop is equivalent to the previous one. It uses the shortcut assignment `x++` (which is the same as `x=x+1`), and demonstrates that the loop counter can be altered inside the body of a loop.

```
for (var x=1 ; x<=1000 ; x++) {
   x++;
   document.write(x+"<br />");
}
```

A more interesting example prints out a table with `r` number of rows and two columns. Each table cell contains the escape sequence for a space (`&nbsp;`). (If you recall, browsers generally don't like to render empty table cells.)

```
document.write("<table border='1'>");
var r=45;
for (var x=1 ; x<=r ; x++) {
   document.write("<tr><td>&nbsp;</td><td>&nbsp;</td></tr>");
}
document.write("</table>");
```

That's a big HTML table for just a few lines of JavaScript code! Here you start to see the power that a programming language can bring to a Web page. To create a table with a variable number of rows and columns, you need to nest two loops. We leave that as an exercise.

NOTE

Be careful if you write HTML attributes to the document using the `document.write()` function. In the above loop that prints out the table, we used single quotes to delimit the value of the border attribute. Using regular quotes in that case would cause a syntax error. The JavaScript interpreter would treat `"<table border="` as the argument of `document.write()` statement.

For the last loop example, we turn to more pseudo-code to emphasize the logic of how we can apply loops to HTML form processing. Suppose a form has a bunch of checkboxes that are for selecting products in an online store. The price of each product is stored in the `value` property of the corresponding checkbox. Remember how checkboxes work -- any number of them can be selected. One way to add up the prices is to use `if` statements.

```
if (first checkbox selected) {
   add its price to total
}
if (second checkbox selected) {
   add its price to total
}
.
.
.
```

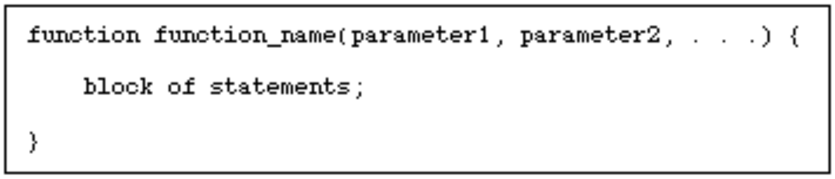
But if there are 50 checkboxes, that would require 50 `if` statements. Of course, the way to deal with the need for repetition is to loop over the checkboxes.

```
var num=the number of checkboxes;
for (var x=1 ; x<=num ; x++) {
   if (checkbox x is selected) {
      add its price to total
   }
}
```

In JavaScript, the scope of the loop counter of a `for` loop is not limited to the execution of the loop. For example, in the loop above that prints out the table, the variable `x` still exists and contains the value 46 after the loop has terminated. The above examples all used `x` as the loop counter. If we were to put all those loops into the same script, that would cause no problem. While a variable declared as a `for` loop counter is effectively a global variable, it is allowed to be re-declared repetitively in the `for` loop context.

Functions

Below is the general form of a JavaScript function. A function can be defined anywhere in a script, but it is customary to define one before you need to use it. When the JavaScript interpreter reads a script it merely makes note of any functions it may encounter. No functions are executed until they are called upon.



```
function function_name(parameter1, parameter2, . . .) {

    block of statements;

}
```

Following the `function` key word, the name of the function must adhere to the same rules that govern variable names. Literals are passed to the parameters by value, rather than by reference. However, JavaScript does pass objects to functions by reference. Thus, when literal values are passed to function parameters, the parameters behave like local variables within the function.

The first example creates and calls a *procedure function* – one which affects the global environment and does not return a value.

```
function customrule(width,char) {
   for (var x=1 ; x<=width ; x++) {
      document.write(char);
   {
   document.write("<br />");
}
```

When the function is called, it must be sent two values, one for each parameter. For example, the call

```
customrule(25,"#");
```

copies the two literals into the function parameters. The function then writes a customized horizontal rule to the document. For the call just above, it would look like

```
#########################
```

Note that the function writes a line break, only after the loop has written the 25 characters.

The second example creates and calls a *return function* – one which returns a value when called and does not affect the global environment.

```
function times10(num) {
   num=num*10;
   return num;
}
```

It's pretty obvious what the function does. The thing to note is that a value function can't be called as a stand alone statement. A call to a value function is replaced with the returned value. For example, the statement

```
times10(2);
```

looks like

```
20;
```

after the value is returned. That is not a viable statement. Rather we assign the returned value to a variable.

```
var num=times10(4);
```

which causes the variable num to contain 40. If decisions need to be made in a return function, multiple return statements can be included. The function simply terminates and returns the value specified by the first return statement it encounters in the flow of execution.

The scope of function parameters and any local variables defined within a function is limited to that function. For example, consider the following block of code.

```
function scopedemo(x) {
   var y=10;

   x++;
   y=x+y;
   z++;
}

var x=2;
var y=3;
var z=4;

scopedemo(z);
```

In the flow of execution, nothing happens until the three variable declarations below the function are encountered. They are initialized with the numbers 2, 3, and 4. Next, the function is called, copying the contents of z into the parameter x. The parameter x, which contains 4, is distinct from the global variable x which contains 2. Next, the local function variable y is created and assigned 10. Again, the local variable is distinct from the global y which contains 3. The next two assignments only involve the x and y local to the function. The global x and y are neither altered nor accessed. Next, since there is no local variable z in the function, the z++ assignment acts on the global level. The legacy of this block of code is that the global variables x, y, and z contain the values 2, 3, and 5 respectively. The two local variables x and y simply no longer exist in the computer's memory. They were erased as soon as the function ceased execution.

Below is a pseudo-code example to show how we will employ functions for HTML form processing. Again, the logic is very simple. We revisit the pseudo-code example from above where we wished to validate a form's contents before processing it. Suppose the form has a button that calls the JavaScript function below.

```
function validateform() {
   if (user enters bad input into form) {
      give an alert
      return false; (don’t process the form data)
   }
   else {
      return true; (ok to process the form data)
   }
}
```

As we shall see, a form's data is carried in global variables. We simply access the global variables from within function. No parameters are required. If no parameters are used in a function, the parentheses ()

must still be provided in both the function definition and the function call. Note that this function both performs a procedure by alerting the user, and it returns a value.

Objects

JavaScript supports self defined objects, but lacks the inheritance mechanism of fully object oriented languages like C++ and Java. Self-defined objects are not often used in JavaScript since you mainly instantiate built-in objects, or simply use the ones a Web browser creates for you. For that reason, JavaScript is sometimes called an *object based* language, rather than object oriented. We will see ample evidence of this beginning in Section 3.3. (Note that future versions of JavaScript are slated to be fully object oriented, with classes and inheritance.)

Reading and Writing to Files

JavaScript is portable code, meaning that the code we write is transported to your computer and executed there. Feeling uneasy? You needn't. The only file on your computer that JavaScript can read from and write to is the cookie file, which is used to (unreliably) store user information on the client between Web transactions.

JavaScript would have been a disaster if it had access to other files on you computer. Oops, there go all your documents. We think you get the idea. JavaScript is not often used to manipulate the cookie file, in favor of dealing with cookies on the server or not using them at all. We will discuss that later in this text.

Arrays

JavaScript fully supports self-defined arrays, which technically are objects. A constructor is called to instantiate an array.

```
var list=new Array();
```

The `list[]` array, then contains an arbitrary number if indices, each vacuously containing the `undefined` literal. For example,

```
document.write(list[100]);
```

would write `undefined` to the HTML document (literally!). With no pre-defined size or index range, indices can be added on the fly.

```
list[0]="hello";
list[1]=3.14;
list[1000]=true;
```

You can see that literal types can be mixmatched in an array.

Every array object has a `length` property that gives the range of the indices that have been assigned. For example, after only the three assignments above, `list.length` is a variable that contains 1001. For that reason, it is best to stick with standard indexing – start at 0, and progress sequentially through the positive integers. That way, you can predictable loop over the array indices.

```
for ( var x=0 ; x < list.length ; x++) {
   alter or access list[x]
}
```

In JavaScript, self defined arrays are useful for storing lists of images for use in rollover displays, for example. In traditional languages, arrays are often used to store lists of data read in from files. In JavaScript that's not a factor.

Most of our use of arrays will involve those that a browser automatically creates. For example, a browser automatically creates an `elements[]` array so you can sequentially access a forms elements. Then, `elements[0]` references the first of a form's elements, `elements[1]` references the second, and so forth.

JavaScript supports multidimensional arrays. In JavaScript, a multidimensional array is technically an array of objects, where the objects are other arrays. Since we have no application for these in this book, that is all we say here.

We offer one more example to re-emphasize that objects are passed to functions by reference. Suppose we have an array

```
var a = new Array();
a[0] = 23;
a[1] = 44;
a[2] = 67;
```

which, for simplicity, only holds three values. Now suppose we define and call a function intended to swap two values in the array.

```
function swap(j,k) {
   var temp = j;
   j = k;
   k = temp;
}

swap(a[0],a[2]);
```

Here the swap function does not effect the global array `a[]` in the least since the two literals are passed to it by value, as are all literals. A solution that actually works is to pass reference to the array to the function, along with the two indices that are to be swapped.

```
function swap(anArray,j,k) {
   var temp = anArray[j];
   anArray[j] = anArray[k];
   anArray[k] = temp;
}

swap(a,0,2);
```

The function call passes reference to the global array into the parameter `anArray`. Of course the indices are passed by value since they are literal values. In other languages, one can designate whether values are passed by value or reference. **In JavaScript when an object name is passed to a function, it is always passed by reference. Literals are always passed by value.**

Another solution is to pass only the indices to the function and alter the array as a global variable inside the function.

```
function swap(j,k) {
   var temp = a[j];
   a[j] = a[k];
   a[k] = temp;
}

swap(0,2);
```

However, it is often considered bad practice to alter global variables inside a function without formally passing reference to the global variable to the function. That is sometimes called a side effect of a function. But when writing JavaScript code to process form data, global variables are often altered inside functions without passing reference. But those global variables are properties of the Browser Object, rather than being self-defined objects.

Comments

Single line comments in JavaScript are of the form

```
// text ignored by JavaScript interpreter
```

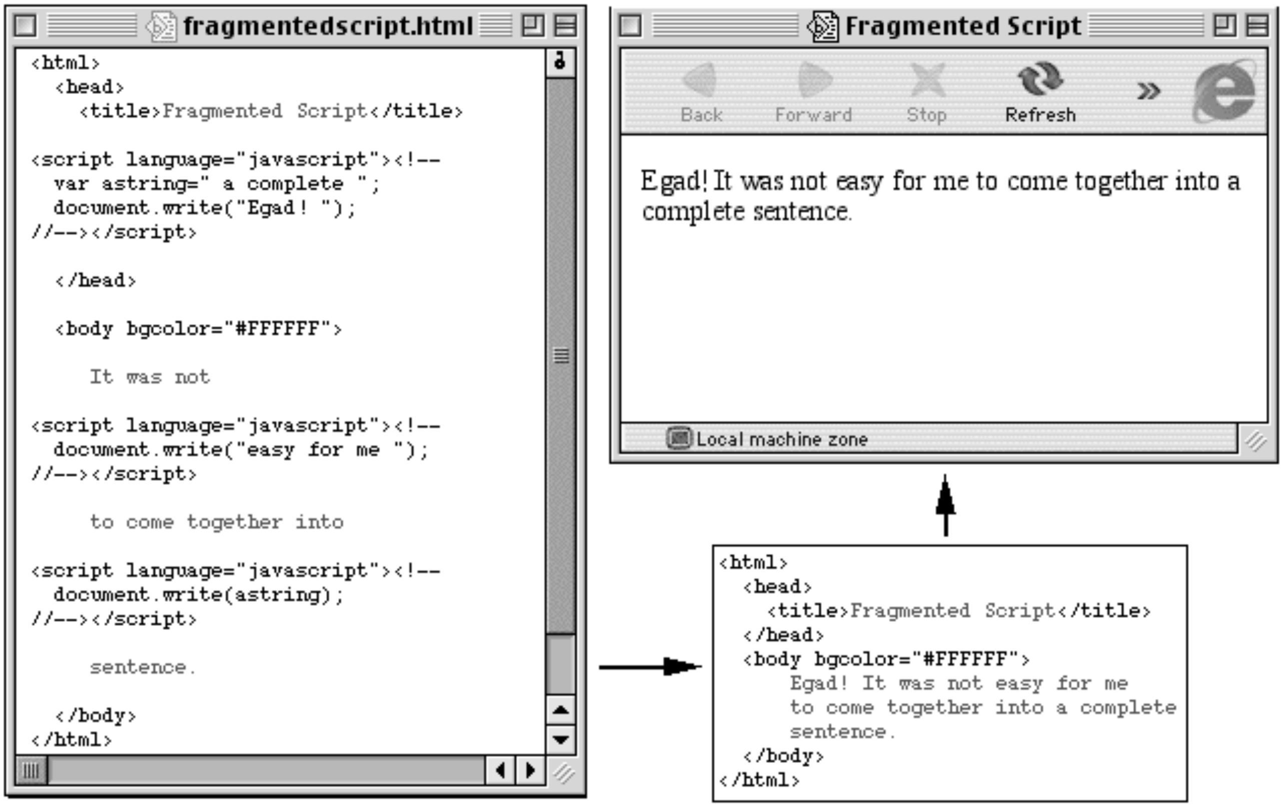
and multiple line comments are of the form

```
/* text ignored by
JavaScript interpreter */
```

We do not use comments in the examples in this lesson since the scripts are fairly short, and we don't want to create extra clutter.

Scripts in General

All of the JavaScript code in a given HTML file collectively forms the script for that file. Script snippets can be mixed in with the HTML at various locations. Of course JavaScript code has to be contained in the HTML `script` element. Figure 3.4 shows the source code and rendition of a Web page that features three different JavaScript code blocks.



```
<html>
  <head>
    <title>Fragmented Script</title>

<script language="javascript"><!--
  var astring=" a complete ";
  document.write("Egad! ");
//--></script>

  </head>

  <body bgcolor="#FFFFFF">

     It was not

<script language="javascript"><!--
  document.write("easy for me ");
//--></script>

     to come together into

<script language="javascript"><!--
  document.write(astring);
//--></script>

     sentence.

  </body>
</html>
```

Egad! It was not easy for me to come together into a complete sentence.

```
<html>
  <head>
    <title>Fragmented Script</title>
  </head>
  <body bgcolor="#FFFFFF">
      Egad! It was not easy for me
      to come together into a complete
      sentence.
  </body>
</html>
```

Figure 3.4 A Script is a global entity for the document.

There are variations among the ways browsers handle scripts when a document is first loaded. But it basically boils down to the JavaScript interpreter goes first, then the HTML parse tree is constructed. The different script sections don't really make any difference to the JavaScript interpreter. It simply starts executing statements from the top down. It does, however, preserve the ordering within the HTML to the extent that a `document.write()` statement writes its argument at the location the statement appears.

You can also see from this that the scope of a global variable is that of the whole document. Indeed, the `astring` variable in Figure 3.4 is declared and initialized in the head section, but its contents are not retrieved until much later in the document. There is no such thing as variable scope within a `script` element. To the JavaScript interpreter it's just one script.

Debugging Scripts

For the first five years of so of its existence, JavaScript lacked the robust syntax debugging support that compiled languages enjoy. Java and C++ development environments usually have elaborate debuggers built in. Moreover, if one of those programs has syntax errors, it simply won't compile. However, if you load a broken script into a browser any number of things can happen. You might get a blank Web page if the interpreter is completely confounded. (In some cases broken JavaScript will even crash a Web browser.) If the browser can make some sense out of the code, you might just get some partial results. Or the script might work except for a small glitch.

The easiest strategy for JavaScript is to load the document into a browser and see what you get (jokingly, WYCIWYG -- What You Code Is What You Get). Often you can figure out what has gone wrong by observing the Web page. Another good method is to put a `document.write("hello")` statement (or a call to the `alert()` function that we will see shortly) in at various locations in the script to see how far the JavaScript interpreter made it through the code. If `"hello"` shows up when the statement is at one location in the code but not another, you can figure (most of the time) that the error is in between.

Fortunately, browsers have error reporting consoles for JavaScript. We highly recommend that you make regular use of one of them. The ones we know of are listed below. Even if a new browser comes out that we like better than the current ones, if it doesn't have a JavaScript error console, we will certainly still keep one of the browsers listed below around just to help debug scripts. (Internet Pillager would be a cool name for a browser.)

For PC's

Explorer 5 and 6: After executing broken JavaScript code, this browser displays a small exclamation point icon in the lower left corner of the window in the status bar. Clicking that icon will report syntax errors.

Netscape 4.7: After executing broken JavaScript code, you simply type javascript: (that's lower case javascript followed by a colon) into the address field of the browser and hit return. You get a JavaScript error console.

Netscape 6: After executing broken JavaScript code, go to the Tasks menu, select Tools, and then JavaScript Console.

For MACs

Netscape 4.7: This has the same error console as Netscape 4.7 for PCs.

Netscape 6: This has the same error console as Netscape 6 for PCs.

NOTE

To our delight, Mozilla released a full featured JavaScript debugger shortly before this book went off to production. Thus, we were just able to slip this note in. The debugger, which has the powerful features you know and love from IDE based debuggers, works in conjunction with any browser built on the Mozilla code base. That includes both Netscape 6 and the new Mozilla browser and on any popular platform. If you end up working with JavaScript a lot (or get a seriously confounding JavaScript error while progressing through this book), it will be worth your while to download it and learn to use it. You will find a link to it on the Web site.

Final Language Notes

We have tried to be succinct. Indeed, the goal here is to learn enough about JavaScript so that we can process HTML form data on the client. If there is some language detail left out that you are wondering about, it probably works the same way in JavaScript as it does in C++ or Java.

Certainly, fancy things that JavaScript can do on the front end (swapping images in and out, moving CSS blocks around, etc.) are beyond the scope of our objectives. Our first book, *Introduction to Interactive Programming on the Internet*, available from John Wiley & Sons, covers JavaScript from a more basic perspective and in more detail, and includes some of its "fancy" implementations.

From the perspective of raw programming, you have seen enough to create really powerful (and portable!) programs. For example, with just the above features, a Mathematician could write an elaborate 10 page JavaScript program that uses numerical analysis to approximate solutions to a differential equation. The language features are all there. JavaScript should not be dismissed as a trivial "scripting" language. Lack of support for reading in data from files might seem a limitation, but JavaScript has direct access to user data collected in HTML forms, and even XML formatted data when it is passed to a browser.

3.3 The Browser Object

Recall Section 2.1 where we represented two blocks of HTML code in terms of parse trees. We wanted you to think of a data structure that an HTML document represents based upon the containment relationships (nesting) of the HTML elements. Those were small trees, but if we consider the Web page as a whole, we get a potentially large tree. The parse tree is what the HTML processor uses to render a document. However, a browser also creates an object based upon the browser window and the HTML document it contains. It is the **Browser Object** shown in Figure 3.5. It is often, and somewhat incorrectly, called the *Document Object Model (DOM*[3]).

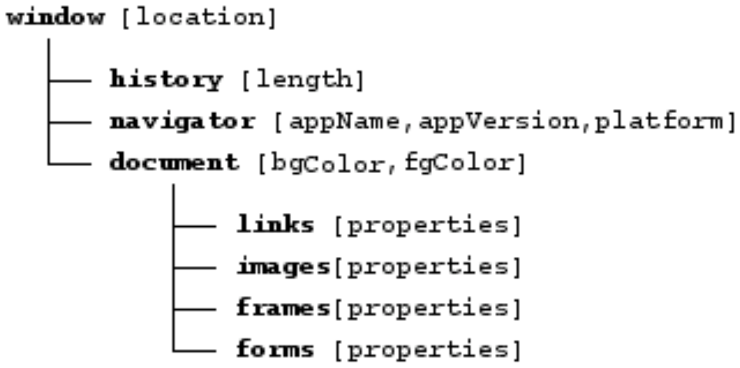


Figure 3.5 A depiction of the Browser Object.

On one level, the browser object is a data structure that holds "state" information for the open window that contains the Web page and more importantly, for the objects inside Web page itself. The primitive variables[4] that hold the state information are listed in brackets after the boldfaced object names. Object properties can be other objects, but eventually there are primitive variables at the ends of the branches that carry the literal values.

On another level, the Browser Object has built-in methods that can be called to change its state. After all, objects by definition have both state and behavior. We discuss the properties of the Browser Object in this section and its methods in the next section.

NOTE

[3] The DOM is very general application programming interface used to expose XML documents to various programming languages in various settings. HTML DOM, the full object exposed to JavaScript in Web browsers, is a "subset" of the DOM. DOM Level 0, which we call the *browser object*, is a "subset" of the HTML DOM. See the JavaScript supplement on the Web site for more details and some history.

[4] Primitives are standard variables that contain literal values : primarily strings, numbers, and Boolean. Objects are not primitives, but bind primitives together in a structured way.

The main purpose of the Browser Object is to provide an *Application Programming Interface (API)* between the Web browser software and the JavaScript language. JavaScript statements can directly access and update properties of the Browser Object, and can call its methods.

The parent is the `window` object. The `window.location` variable contains the URL of the Web page (or other resource) currently displayed in the browser window. You can use JavaScript to change the state of the `window` object by assigning a new value to its `location` property

```
window.location="http://www.cknuckles.com";
```

which instantly loads the page that the URL points to. (Try it yourself, it's just a one-line script!) Changing that property with JavaScript is commonly used to transfer to a new page after a few seconds (like when a site has moved) or to load new pages when form buttons are clicked.

One level down in the hierarchy, the `window.history` object is actually an array object that contains the list of URLs that have been stored in `window.location` since the window has been open. There is a "go" pull-down menu at the top when a browser window is open, which lists titles of the last few pages to which you have surfed. Maybe you have used it to go back a few pages in your surfing "history." The `window.history` array is parallel to that array (menus are just arrays) and contains the URLs of the pages whose titles you see in the "go" menu. The primitive variable `window.history.length` contains the length of the array (number of items in the surfing history).

The `window.navigator` object, whose name comes from Netscape's original Browser Object, has primitive properties that contain information about the browser type, browser version, and platform on which the browser is running. They are used for browser sniffing so customized content can be delivered. The properties of this object are summarized in Appendix A, and an elementary browser sniff is given as an exercise.

The `window.document` object refers to the actual Web page. Its primitive properties and the rest of the objects you see nested below it in Figure 3.5 contain state information for the Web page. Two primitive properties whose values can be set using JavaScript are listed in Figure 3.5. Setting the two properties

```
window.document.bgColor="#000000";
window.document.fgColor="#FFFFFF";
```

in a script causes the background and foreground (text) colors to be instantly changed in the Web page to black and white, respectively. Including those statements in the Web page is equivalent to setting the values using HTML attributes. Either way, the colors are set as the Web page loads[5].

The `window.document.images` object is an array that indexes all of the graphics that have been marked up in the Web page. Rollover graphics are created by using JavaScript to swap images in and out of that array. Ok, we're starting to stray off the path. In fact, we're not going to talk about the `window.document.links` or `window.document.frames` arrays which index those components of Web pages. The point here is that there is a big object, that contains information about the current state of the Web page. JavaScript has access to many of the state variables, and can assign values to them to affect the state of the Web page.

NOTE

Figure 3.5 does not give the complete Browser Object. Different browsers make somewhat different Browser Objects. But even if we were to stick to an object hierarchy common to all newer browsers, it's much too large for our purposes here. It is the `window.document.form` object that is our primary agenda. To fully comprehend that object, it helps to know its place in the scheme of things.

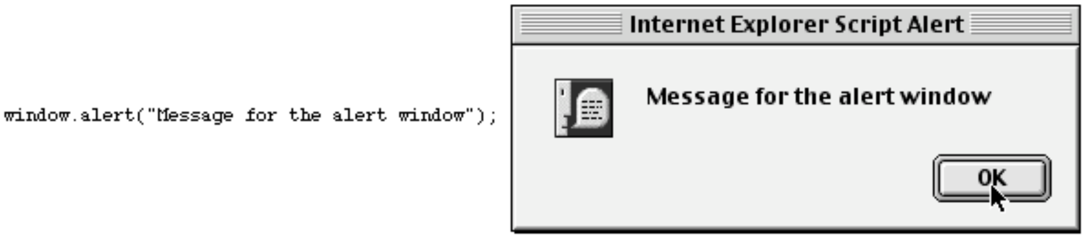
[5] Setting colors in the `body` section of the document will override the corresponding HTML attributes set in the `body` element. However, the HTML color attributes in the HTML body tag may override color settings made by JavaScript in the head section. (There are browser dependent nuances that determine this.)

3.4 Methods for the Browser Object.

We have seen a few instances of how JavaScript statements can be used to influence the state of the Browser Object, and hence the Web page. Now we discuss some of its **methods** – the built in functions that can be called on the Browser Object to change its state. Of course, the methods will be called using JavaScript Statements.

We will spend most of the rest of our time in this lesson manipulating the Browser Object with self-defined JavaScript functions. But some of the built-in methods are quite useful. We discuss a few of those here, explore some others in the supplement on the Web site which discusses some more advanced JavaScript topics, and provide a rather complete reference to the methods of the Browser Object in Appendix A.

At the `window` level, there is the `window.alert()` method. It takes one argument, a string, and causes a small window to pop up containing the string. Figure 3.6 shows a one-line script and the result. (We still haven't figured out what the guy in the alert window is saying.)



```
window.alert("Message for the alert window");
```

Message for the alert window

Figure 3.6 The `alert` method of the window `object`.

Other than the message, all the alert does is to halt the flow of execution of the script. For example, if the line of code in Figure 3.6 appeared in a longer script, the script would stop indefinitely at the alert call. The user would have to hit the OK button before the window would go away, at which time the JavaScript interpreter would only then pass to the next statement in the script.

Another window method of which you have seen evidence time and time again is `window.open()`. A call to this method creates one of those pesky pup-up windows that typically contain an advertisement. The advanced JavaScript supplement provided on our Web site explains how to use the `window.open()` method to redirect the output from CGI programs to a new window.

Another method that you have called time and time again is `window.history.go()`. It takes one parameter, an integer, that moves you to a different index in the `window.history` array. The current page in the window is at index 0. So, for example, the call

```
window.history.go(-1);
```

is equivalent to hitting the "back" button on your browser. Incidentally, `window.history.go(-1)` is equivalent to `window.history.back()`.

We have been using a method of `window.document` for much of this lesson. The method does no more than write strings of text into the HTML document. It is `window.document.write()`.

NOTE

There are two items of importance here. First, why have we been calling the method `document.write()` without using the full reference to the `window` object? Well, you don’t have to. Being the parent object, you lose no specificity if you simply don’t reference it. In fact, we could have accessed all of the object properties and methods in this section, and the last, without referencing the `window` object. We will still use the `window` reference for things like `window.alert()` and `window.history`, even though they are equivalent simply to `alert()` and `history`, in order to emphasize their place in the Browser Object. However, we will forgo reference to it at the `document` level and below.

Second, all of the object, property, and method names we have used are case sensitive. So, for example, document.bgcolor would not be a recognized property in the Browser Object.

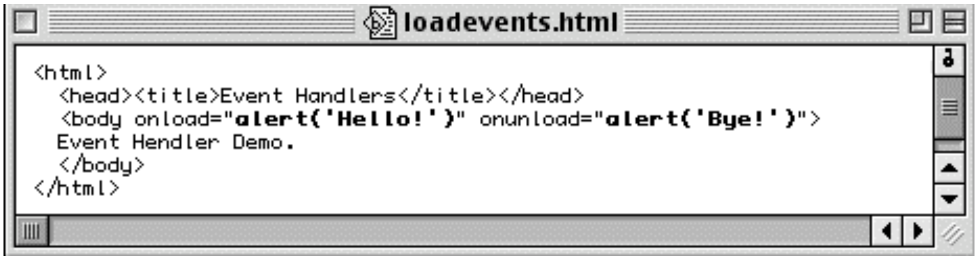
3.5 Events

Interactive programming involves dealing with a ***user event*** – when the user clicks a link or a form button, for example. An **event handler** is a special property of the browser object that can detect user events. The easiest way to handle a user event is to place an HTML attribute corresponding to the event handler inside an HTML tag.

For example, onload and onunload are two event handlers of the window object that are triggered when the Web page in the browser window loads and unloads, respectively. Formally, the event handlers are special properties of the window object.

```
window.onload
window.onunload
```

But they are commonly put into action by placing corresponding event handling attributes in the HTML body element. For example, the document shown in Figure 3.7 reacts both to the user loading the page and leaving the page.



```
<html>
  <head><title>Event Handlers</title></head>
  <body onload="alert('Hello!')" onunload="alert('Bye!')">
  Event Hendler Demo.
  </body>
</html>
```

Figure 3.7 Reacting to onload and onunload events

When this Web page first loads into the browser, the onload event handler is triggered and it executes its value as a JavaScript statement. In this case, the statement is a call to the window.alert() method. Similarly, the onunload event handler calls the window.alert() method, sending it a different message. You should go to the Web site and load the page. The onunload event handler is triggered when you cause the browser to load another page, even if you reload the same page. So hitting the refresh button on the browser triggers onunload and then onload again as the page reloads.

When placing event handlers as HTML attributes, the value of the handler can be any JavaScript statement or sequence of statements separated with semi-colons). For example, we could change the background and text colors color of the Web page when it loads. (Although cause to do so is hard to imagine.)

```
<body onload="document.bgColor='red'; document.fgColor='blue';">
```

Note in the boldfaced JavaScript statements here and in Figure 3.7 that single quotes are used on the inside so that the event handler is well defined as an HTML attribute. Using double quotes on the inside will prematurely terminate the value of the HTML attribute.

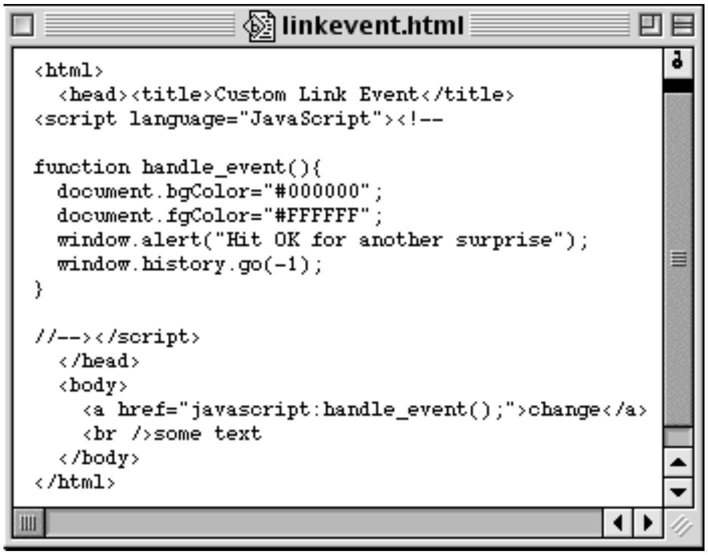
In practice, the JavaScript statement called by an event handler is usually a function call, like in Figure 3.7, rather than an assignment statement. In Figure 3.7, we called a built-in function, but an event handler can call a self-defined function created to handle the event in a specialized way.

One thing a Web programmer needs to understand is the built-in event handler used by the link object to handle link click events. It is triggered using the href attribute. Consider the following two anchor elements.

```
<a href="http://www.cknuckles.com">go</a>

<a href="javascript:window.location='http://www.cknuckles.com';">go</a>
```

If you put these links in a Web page, they both accomplish the exact same thing. What we have done in the second one is override the built-in event handler. Rather than specifying a URL as normal, the `javascript:` directive in the value of the `href` attribute causes the rest of the value (shown in boldface) to be executed as a JavaScript statement. That causes the browser to load our home page, just like the ordinary link.

Pretty tricky, we have programmed our own link. But as we have said, event handlers are usually used to call functions. If multiple JavaScript statements are required to handle the event, it's best simply to call a self-defined function and deal with the event elsewhere. That's precisely what we do in the HTML file in Figure 3.8.



```
<html>
  <head><title>Custom Link Event</title>
<script language="JavaScript"><!--

function handle_event(){
  document.bgColor="#000000";
  document.fgColor="#FFFFFF";
  window.alert("Hit OK for another surprise");
  window.history.go(-1);
}

//--></script>
  </head>
  <body>
    <a href="javascript:handle_event();">change</a>
    <br />some text
  </body>
</html>
```

Figure 3.8 A custom event handler for a link.

When the user causes the click event on the link, control is given to the JavaScript interpreter. The JavaScript statement is a call to the self-defined `handle_event()` function. That function changes the text and background colors of the Web page, calls the `alert()` method, and then transfers the browser one back in the surfing history (hits the back button). Try it! Like always, the source code is on the Web site.

While its cool to make a link do whatever you want it to, the above script is not practical. However, custom event handlers for links are often used to open a new Web page in a pop-up window, for example. The important concept for us here is this. The user causes an event, a JavaScript function is called, and that function does something to the Browser Object to change its state.

Since the primary focus of this Lesson is on manipulating the `form` object, we will primarily use event handlers associated with that object. Typical form events are clicking a command or submit button. We shall shortly learn how to handle those events, but it still boils down to calling functions. With that in mind, there is a subtlety that must be addressed. The subtlety pertains all event handlers, so `window.onload` is sufficiently general to make the point.

Technically, the JavaScript code executed by an event handler should return a Boolean value to the event handler. For example,

```
<body onload="document.bgColor='red'; return true;">
```

purports to inform the event handler that the event was handled successfully. When using a function to handle the event, the function should return a Boolean value.

```
<body onload="somefunction()">

<script language="JavaScript"><!--
function somefunction() {
   handle event
   return true or false depending upon success or failure
}
//--></script>

</body>
```

Returning success/failure status to event handlers is a common error control practice (and not just in JavaScript). However, when the HTML attribute form of the event handler is used, the function is "loosely bound" to the event handler. That means that the returned value does not actually reach the event handler. This is just the way the Browser Object works. For that reason, there is no need to return Boolean values when the HTML attribute form of the event handler is used.

We defined user events as special properties of the browser object. Indeed, the actual event handler is

```
window.onload
```

The rigorous way to bind a JavaScript function to an event handler is to formally assign the function to the event handler.

```
<body>

<script language="JavaScript"><!–
window.onload = somefunction;

function somefunction() {
   handle event
   return true or false depending upon success or failure
}
//--></script>

<body>
```

Here the function is assigned[6] without using parentheses `()`. Formally assigning the function to the event handler assures that the returned Boolean value reaches the event handler. If the event handler has a contingency in place to deal with a potential failure message, it will do so.

However, very few event handlers of the Browser Object actually care about returned Boolean values. In fact, the `onload` event handler will simply ignore a returned value. That is why the HTML attribute form of almost any event handler is used. In general, it is a moot point as to whether an event handler in the Browser Object receives a returned value or not. So you might as well use the HTML attribute form and not worry about returned values.

But there is one case where of importance to us where an event handler needs to receive a returned success/failure message. As we shall soon see, that event handler has to do with the user event of submitting the data from an HTML form to the client. In that case, we will formally assign a function to the event handler as demonstrated just above.

In the next few sections, we will fully explore the form object's event handlers. However, there are a host of other event handlers in the Browser Object used for various purposes. We don't have the latitude to cover them all in this text. But you have unwittingly triggered many of them while surfing. The most common are `onmouseover` and `onmouseout`, which are commonly used to call functions when the mouse passes onto and back off of an image, respectively. That creates the rollover effect you get when

[6] While there is no need to do so in this text, self-defined JavaScript objects can be created. In that case, assigning a function to an object in this fashion establishes the function as a method of the object.

your mouse passes over certain images that are active as links. `Onmouseover` calls a function which swaps in a new image, and `onmouseout` calls a function which swaps back in the original image. The swaps are accomplished by updating the `document.images[]` array in the Browser Object. (These event handlers don't care about returned values, so they are called as HTML attributes.)

NOTE

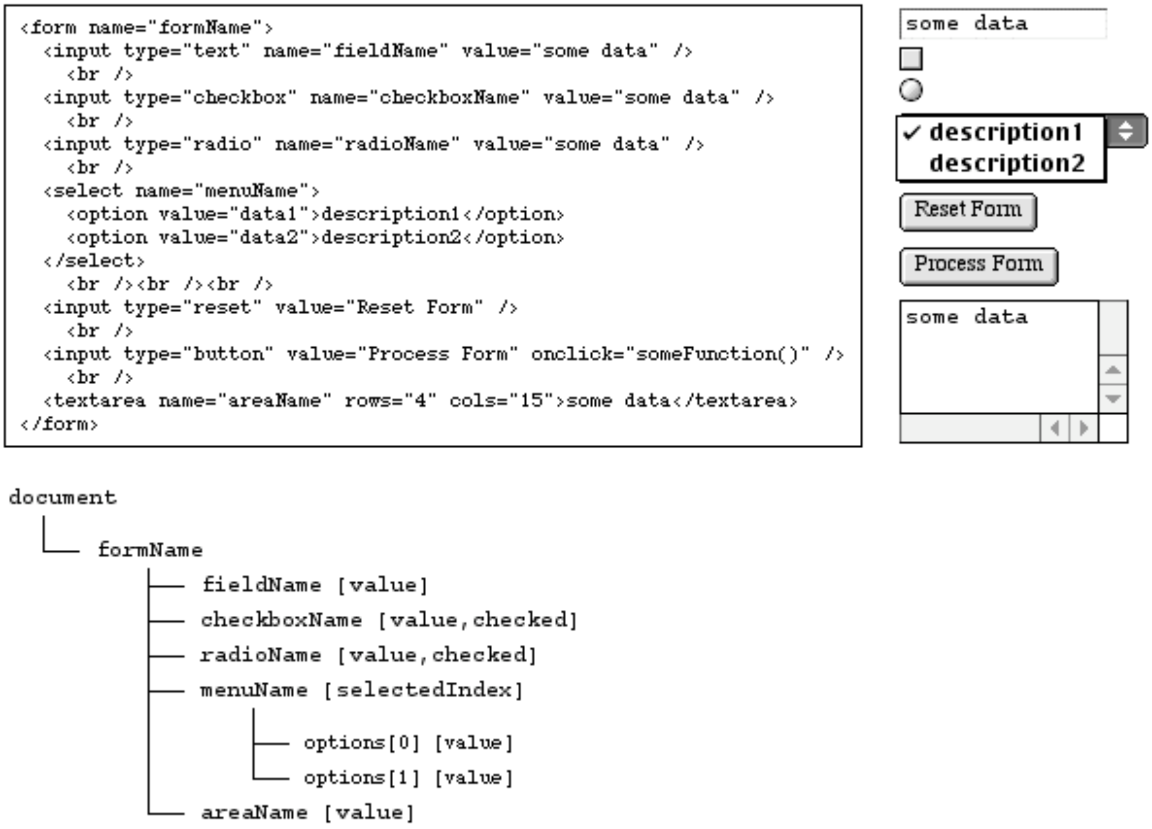
There is a stark contrast between how JavaScript was used in this section compared to the previous ones. Before user events, the JavaScript statements were executed when the page first loads. The first examples used `document.write()` to include the content of a variable (containing the result of a calculation, for example) in the Web page. The next examples in Sections 3.3 and 3.4 used JavaScript to set properties of the Browser Object as the page is first loading.

With user events, the JavaScript statements are in functions, which are bound (loosely or formally) to the event handler, and are not executed as the page loads. Only when the user event occurs do the statements in the functions get executed. That means we are updating the state of the Browser Object after the page is fully loaded. Not all properties of the Browser object can be changed after the page is fully rendered. For example, calling `document.write()` can't update the state of the `document` object once a page is rendered. At that time, the page is already rendered and the text of the page is fixed in place. However, things like the background color can be updated via the Browser Object long after the page is fully rendered since that doesn't require updating the layout of the Web page. In general, the following notes hold true about what can happen as the result of a user event once a Web page is fully rendered by the browser.

- The state of the `window` object can be updated after the page is fully rendered. That includes changing properties like `window.location` and changing properties of the `window.history` object. Also, methods like `window.alert()` can be called at any time.
- Many properties of the state of the `document.form` object can be updated after the page is fully rendered. That includes the data contained in text fields and text areas, whether checkboxes and radio buttons are checked or not, and the currently selected item in a pop-up menu.
- As we have mentioned, objects like images can be swapped in and out after the page is fully loaded to create rollovers. As long as two images are the same size, they can be swapped in and out of the Browser Object without affecting the layout of the Web page.
- Image rollovers are, historically, the most common Dynamic HTML (DHTML) effect. Other fancy DHTML effects like moving blocks around and working with layers can't be accomplished by manipulating the Browser Object. That requires use of CSS and exposure of the style properties to JavaScript through the full HTML DOM. We do discuss the relationship of the Browser Object to the DOM in the DHTML supplment provided on the Web site.

3.6 The Form Object

We saw in Figure 3.5 that an HTML form is an object that is a property of the `document` object. Most properties of the `document.form` object can be both accessed and changed after the Web page containing the form has been loaded. Figure 3.9 shows an HTML form and its structure as an object. Only the data carrying elements are represented in the depiction of the `document.form` object. The only purpose of form buttons is to trigger event handlers



```
<form name="formName">
  <input type="text" name="fieldName" value="some data" />
    <br />
  <input type="checkbox" name="checkboxName" value="some data" />
    <br />
  <input type="radio" name="radioName" value="some data" />
    <br />
  <select name="menuName">
    <option value="data1">description1</option>
    <option value="data2">description2</option>
  </select>
    <br /><br /><br />
  <input type="reset" value="Reset Form" />
    <br />
  <input type="button" value="Process Form" onclick="someFunction()" />
    <br />
  <textarea name="areaName" rows="4" cols="15">some data</textarea>
</form>
```

Figure 3.9 The Browser Object contains an object to represent the HTML form.

The name of the form and the names of the form elements, set by the `name` attributes in the HTML code, give reference to the particular form elements in the object. The primitive, data carrying variables for each element are given in brackets at the ends of the branches. For example,

```
document.formName.fieldName.value
```

is a primitive variable containing the string literal `"some data".` The checkbox and radio button each have two primitive properties. The variables

```
document.formName.checkboxName.checked
```

and

```
document.formName.radioName.checked
```

are Boolean. Since neither of them is checked in Figure 3.9, both of the Boolean variables currently contain the literal `false`. The menu is somewhat more complicated. The menu itself has a `selectedIndex` (Note the capital `I`.) property that contains an integer corresponding to which menu option is currently chosen. Since the first option is selected in Figure 3.9, the primitive

```
document.formName.menuName.selectedIndex
```

currently contains the integer 0. That might seem strange, but as you can see, the menu's options are represented by the `options[]` array. Using standard array indexing,

```
document.formName.menuName.options[0]
```

refers to the first option in the menu. But to get at the data carried by that particular option, you have to reference the primitive variable

```
document.formName.menuName.options[0].value
```

which contains the string "data1". Now, that's a long object reference, but you can easily trace the reference path by looking at the depiction of the object in Figure 3.9. Finally, the string data carried by the text area is contained in

```
document.formName.areaName.value
```

as you would expect.

The control buttons are not given in the object depiction because we do not need to access any of their properties. The buttons are simply to cause something to happen to the form. The reset button has a built in event handler which calls a built-in `reset()` method that returns form object to its original state. The command button simply calls a self-defined function using the `onclick` event handler. You can see the call to `someFunction()` in Figure 3.9.

Above, we discussed what the various properties of the form object contain based on the current state of the form as you see it in Figure 3.9. Those contents can be changed simply by assigning new values to the form's properties in `someFunction()`. For example, in the source code on the Web site, the function is defined as follows

```
function someFunction() {
  document.formName.fieldName.value="What's ";
  document.formName.radioName.checked=true;
  document.formName.menuName.selectedIndex=1;
  document.formName.areaName.value="up!";
}
```

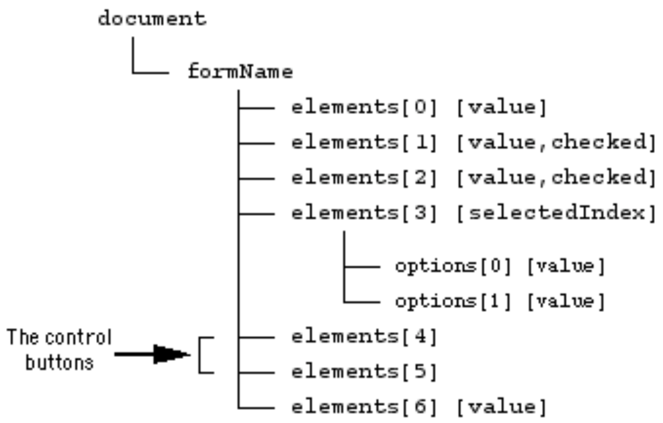
When the user event of clicking the command button calls this function, the state of the form is instantaneously updated. The text display elements are changed to contain new strings, the radio button is "turned on," and the current menu selection is changed to the second item. Try it for yourself on the Web site.

There is a limitation to using the names of the form elements to provide reference to them in the Browser Object. It is difficult to iterate over names. Suppose there are 50 checkboxes whose value properties carry prices for items that can be chosen. If you recall the pseudo-code example for loops in Section 3.2, one solution is to include 50 "if" statements. We can now redo the pseudo-code with real code.

```
var total=0;
if (document.formName.checkkbox1.checked) {
      total=total+parseFloat(document.formName.checkkbox1.value);
}
if (document.formName.checkkbox2.checked) {
      total=total+parseFloat(document.formName.checkkbox2.value);
}
.
.
.
```

Each of the 50 "if" statements sees if a checkbox is checked. If it is, its value is added onto a running total. Remember, all form `values` are strings. So, in this case we have to parse the `values` into numbers or else + will concatenate them.

Clearly we need to iterate over the checkboxes with a loop and do away with the 50 conditionals. The solution is to use an alternate representation of the `form` object that does not use names to reference the form elements. The `elements[]` array automatically gives an indexed referencing scheme for them. To show that, we offer a second version of the object for the form of Figure 3.9.



In this version, reference to a specific form element is given by its position in the `elements[]` array, rather than by its name. The `elements[]` array uses standard indexing, so the first form element is referenced by `elements[0]`. Accessing or changing an element's property is no different than before. For example,

```
document.formName.elements[2].checked
```

references the radio button's Boolean property. Similarly,

```
document.formName.elements[3].options[0].value
```

references the data carried by the pop-up menu's first option.

Indexing a form's elements does add one factor that has to be taken into account. The `elements[]` array indexes all of a form's elements, not just the data carrying ones. Thus, to reference the text area, you have to be careful to skip over the index positions occupied by the form's two control buttons. The button positions are emphasized in the depiction of the indexed version of the form object. Skipping over those, `elements[6]` references the text area. With names, you simply don't name the control buttons and don't reference them.

If we go back to the case of the 50 checkboxes we can now iterate over them. Assuming they are the first 50 elements defined in the form, they occupy positions 0-49 in the `elements[]` array. So, each execution of the loop

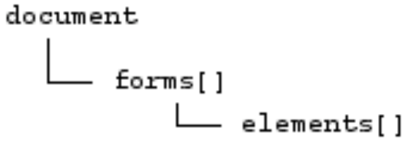
```
var total=0;
for (var x=0 ; x<=49 ; x++) {
  if (document.formName.elements[x].checked) {
      total=total+parseFloat(document.formName.elements[x].value);
  }
}
```

tests whether `elements[x]` has been checked and if so, adds its value onto the running total. As `x` runs from 1 to 49, all of the 50 checkboxes are tested.

If that doesn't convince you of the utility of the `elements[]` array, recall that a unique selection group of radio buttons all share the same name. That's no problem if the form is being submitted to the server because only the selected one is submitted. However, for processing on the client, there is no way to individually reference radio buttons in a unique selection group by name. You have to use the `elements[]` array in that case.

In practice, you will probably want to use a mixture of naming and using the `elements[]` array. When you are writing a function to process a form, it's easy to remember which element to reference if you have given them descriptive names. But sometimes you will need to iterate over form elements or refer to specific radio buttons in a unique selection group. In those cases, names won't help you in an object reference capacity.

In like manner, there is a `forms[]` array that indexes all of the different forms in an HTML document. That means you don't even have to name the form. Using both of these built-in arrays, all of the forms and their elements in a page have the following concise object representation.



For example, assuming the form in Figure 3.9 is the only form in the Web page, we can reference the data carried by the second of the pop-up menu's items with

```
document.forms[0].elements[3].options[1].value
```

and that's a big ugly object reference. In a case like that it is better just to reference for the form and menu with their names. That way you can tell at a glance what you are dealing with.

NOTE

If you are processing form data on the client, it really doesn't matter if you name a form and its elements. If you can keep up with all of the array indices, you are good to go using the built-in arrays. However, you should get in the habit of naming all of your data carrying form elements. When a form is submitted to the server, an element with no name is not submitted, even if it has a value. There is no way for the browser to construct the `name=value` pair.

In contrast, you usually don't want to name submit, reset, or generic buttons in a form. That will trick the browser into sending a `name=value` pair for the button. You usually don't want something like `buttonName="Click Me"` being submitted. However, naming submit buttons can be useful when a form has more than one submit button used to trigger different types of processing on the server. The CGI program can then determine which type of processing to enact based upon the name of the particular submit button that is clicked.

That covers how the data from the current state of a form is exposed to JavaScript via the Browser Object. But, of course, the `form` object also has methods and event handlers. We now give an overview of the most commonly used ones. Full reference to the properties, methods, and event handlers can be found in Appendix A.

The common form-level methods are `reset()` and `submit()`. You can call these methods directly to return the form to its original state when the page loaded or to submit the form the Web server, respectively.

```
document.formName.reset();
document.formName.sumit();
```

However, you would scarcely have cause to do so. The reset and submit form buttons have built-in event handlers that call these two methods.

The methods we feature that apply to specific form elements are `focus()` and `select()`. These are most often used on text fields. By default, none of a form's elements are in focus[7]. This method call

```
document.formName.textFieldName.focus();
```

focuses a text field, which puts the cursor in the field. Such a call might be made from the `onload` event handler to bring the text field for a search engine into focus when the page loads. Otherwise, the user first has to click the mouse on the text field before typing in a query. (There is a `blur()` method that accomplishes the opposite of `focus()`, but is rarely needed.)

[7] Load the form of Figure 3.9 into your browser. Then hit the tab key a few times and you will see focus move sequentially among the form elements.

Another use is to put focus in a particular text field after a form is validated. If there are several text fields, this lets the user know which one into which they entered faulty data. In that case, it is common to also call

```
document.formName.textFieldName.select();
```

which selects all of the text in the field. Selecting the text means it becomes highlighted, like when you highlight some text to copy and paste it.

The event handlers we present for form elements are `onclick` and `onchange`, and `onsubmit`. We saw `onclick` used in the command button in Figure 3.9. That is the only element in which we will use `onclick`. Its purpose is to call a JavaScript function that performs client-side processing of the form's data. This event handler does not require a return value, so we simply supply it as an HTML attribute.

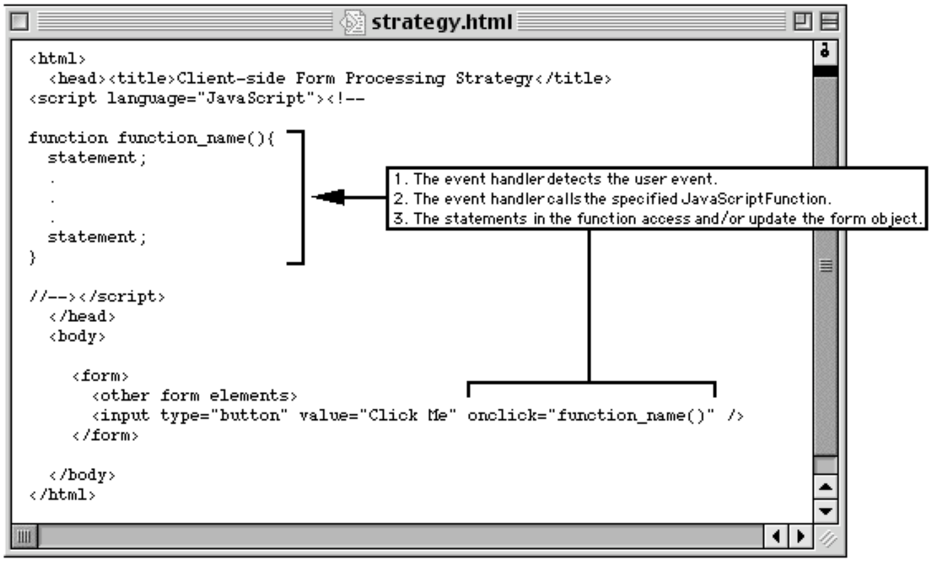
The `onchange` event handler is available for most form elements. Its two main uses are in text fields and pop-up menus. It does not require a return value, so it's attribute form is used. When included in a pop-up menu it is used to call a function each time the user selects a new menu item. Triggering `onchange` from a text field requires adding or deleting a character from the field and then removing focus from the field, by clicking on a different form element with your mouse, for example. The act of typing in a text field technically changes it, but the change is not registered in the Browser Object until focus is removed. That is when it knows you are done typing.

The `onsubmit` event handler is built in to the submit button. Clicking a submit button automatically triggers this handler which then calls the `submit()` method on the form. Of all the event handlers discussed in this book, this is the only one that benefits from receiving a Boolean return value. In Section 3.9, we will see how to override its default behavior of calling the `submit()` method, and assign it a self-defined function that returns a Boolean value that depends upon whether or not the form's data passes client-side validation.

3.7 Processing form elements.

Now that you know the basics of the `form` object, we offer some examples so that you can get a feel for client-side processing of the various types of form elements. The examples of this section are basically client-side utilities that do not feature submission of data to the Web server.

Most any strictly client-side form data processing utility utilizes the strategy depicted in Figure 3.10. A control button is supplied with the `onclick` event handler. The page loads into the browser, and event handler just sits and waits for the click event. When the event happens, a self-defined JavaScript function is called do handle the data processing. For convenience, we placed the function in the head section, although it could go in the `body` section as well.



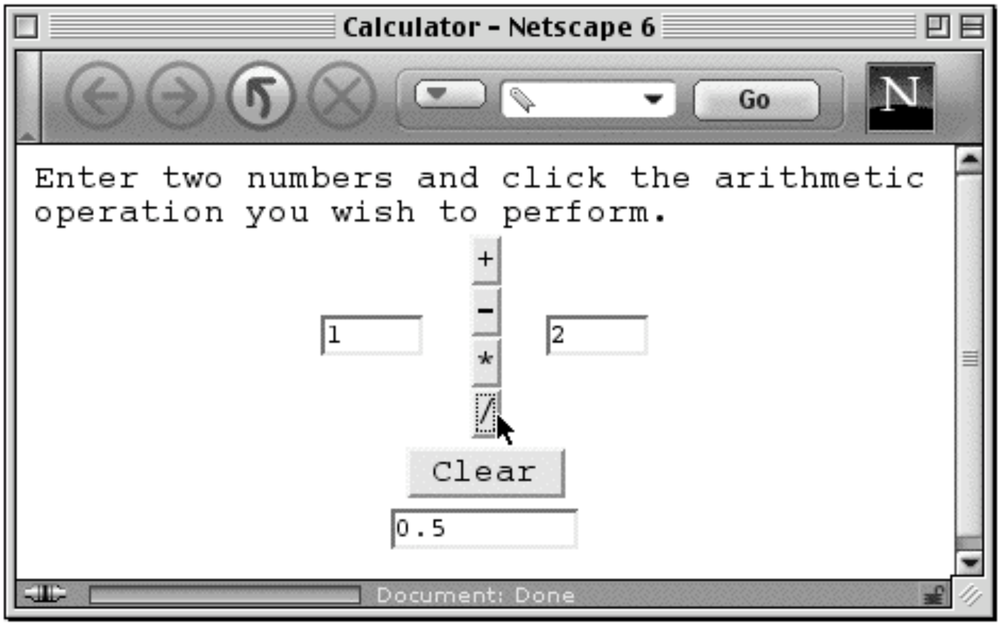
```
<html>
  <head><title>Client-side Form Processing Strategy</title>
<script language="JavaScript"><!--

function function_name(){
  statement;
  .
  .
  .
  statement;
}

//--></script>
  </head>
  <body>

    <form>
      <other form elements>
      <input type="button" value="Click Me" onclick="function_name()" />
    </form>

  </body>
</html>
```

1. The event handler detects the user event.
2. The event handler calls the specified JavaScriptFunction.
3. The statements in the function access and/or update the form object.

Figure 3.10 Using button events to initiate client-side processing of form data.

Processing Text

The first example is a simple calculator. Figure 3.11 shows the HTML form used in the example. The form elements have been formatted with a borderless table, but only the HTML code for the form has been provided. The full source code can be observed by pulling up the source file on the Web site.



Enter two numbers and click the arithmetic operation you wish to perform.

1

2

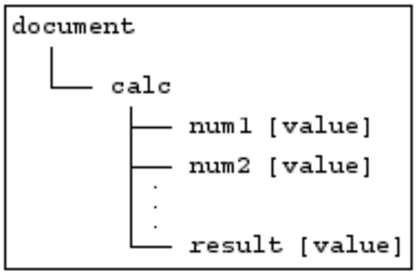
0.5

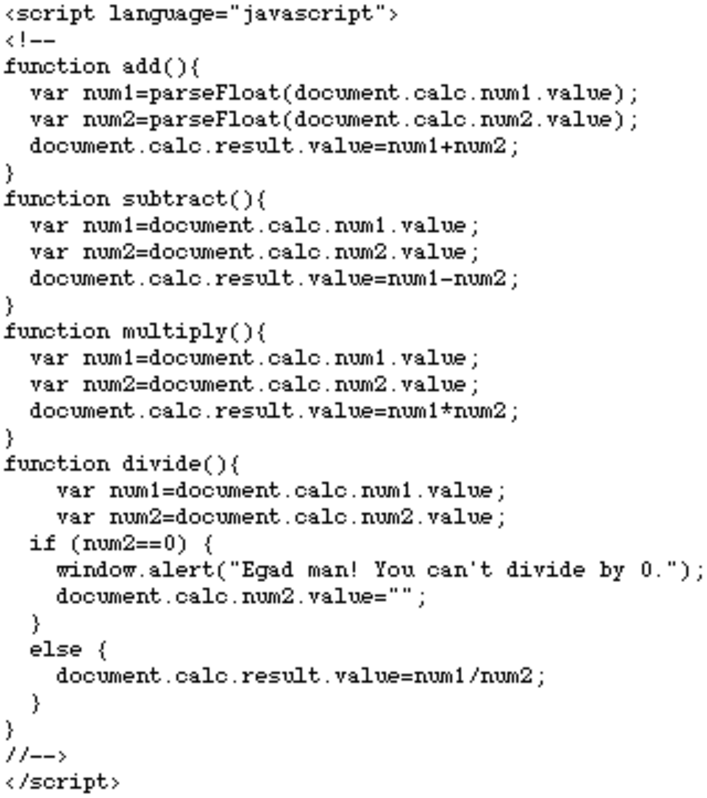
```
<form name="calc">
  <input type="text" size="5" name="num1" />
  <input type="text" size="5" name="num2" />
  <input type="button" value="+" onclick="add()" />
  <input type="button" value="-" onclick="subtract()" />
  <input type="button" value="*" onclick="multiply()" />
  <input type="button" value="/" onclick="divide()" />
  <input type="reset" value="Clear" />
  <input type="text" size="10" name="result" />
</form>
```

Figure 3.11 A simple JavaScript calculator utility.

The two operands for the arithmetic calculation are entered into text fields by the user, and the calculated result appears in the bottom text field. Those elements have been named so that their contents can easily be accessed or updated in the form object. Each of the four buttons that is to initiate an arithmetic operation has an `onclick` event handler. Each event handler calls a different JavaScript function, named appropriately for the operation. An object model for this form and the JavaScript functions that manipulate it are given in Figure 3.12. In the full source document, the JavaScript code is located in the `head` section, although that's not a must.

We begin with the `add()` function. First, the numbers entered by the user are to be added, so they are parsed into numbers. (Otherwise, they would be concatenated as strings.) The numbers returned by the `parseFloat()` function are stored into local variables inside the function. Then, the sum of those two numbers is assigned to the value of the `result` text field. That's all there is to it.





```
<script language="javascript">
<!--
function add(){
  var num1=parseFloat(document.calc.num1.value);
  var num2=parseFloat(document.calc.num2.value);
  document.calc.result.value=num1+num2;
}
function subtract(){
  var num1=document.calc.num1.value;
  var num2=document.calc.num2.value;
  document.calc.result.value=num1-num2;
}
function multiply(){
  var num1=document.calc.num1.value;
  var num2=document.calc.num2.value;
  document.calc.result.value=num1*num2;
}
function divide(){
    var num1=document.calc.num1.value;
    var num2=document.calc.num2.value;
  if (num2==0) {
    window.alert("Egad man! You can't divide by 0.");
    document.calc.num2.value="";
  }
  else {
    document.calc.result.value=num1/num2;
  }
}
//-->
</script>
```

Figure 3.12 An object model and the JavaScript to support the form of Figure 3.11.

The `subtract()` and `multiply()` functions are nearly identical to the one that adds, except that the user input is not parsed into numbers. This is a reminder of the flexibility of JavaScript. With no string context for the – and * operators, the strings are automatically converted into numbers at the time of the calculations.

The `divide()` function is a bit different. The form values are still "dumped" into local variables. Then, the function features a bit of validation. To avoid possible division by 0, the function employs the logic given in the following pseudo-code.

```
if(second operand is 0) {
  alert the user
  clear the second operand
}
else {
  complete the calculation
}
```

The pseudo-code should be compared with the actual code in Figure 3.12.

It is not necessary to store the form data into local variables in the latter three functions, but that is a technique we will continue to employ. That way, you can do calculations using short variable names, rather than the long object references. However, in the `add()` function, there is no alternative. Again, the `value` primitives are among the few JavaScript variables that are not loosely typed. They are always strings. So the following assignment

```
document.calc.num1.value=parseFloat(document.calc.num1.value);
```

accomplishes absolutely nothing. The `parseFloat()` function returns a numeric literal as usual, but assigning it to the form `value` converts it right back into a string. However, assigning the returned value into a local variable preserves the numeric format returned by the parsing function.

One important concept in this example is that different command buttons can call different functions. However, we could have constructed the example so that only one function is called by all four of the buttons. That function would have to take a parameter so that different calls to it could tell it which operation should be performed. Also, other validations of the input could be performed. For example, the calculator might also give an alert if one of the operands is left blank or some joker enters alphabetic characters, rather than numbers. Those modifications are left as an exercise.

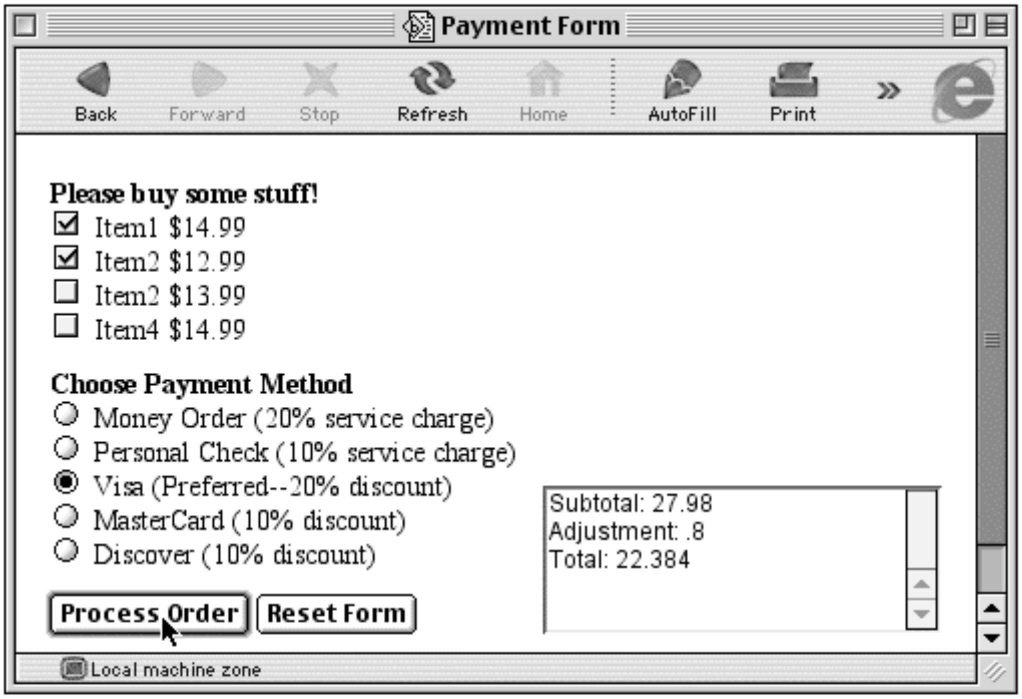
NOTE

> We don't wish to keep referring the reader to the Web site to check out these examples first hand. With interactive examples, a much better feel can be obtained by playing with them. In particular, one needs to test the form validation mechanisms first hand.

Processing Options

The second example of this section focuses on processing radio buttons and checkboxes. Figure 3.13 gives a form that simulates a simple client-side shopping cart. Like the previous example, this form is also formatted with borderless table. But since there is less code for the table this time, the whole contents of the HTML `body` element are provided.

The functionality of the form is straight-forward. The user can select any or all of the checkbox items. Although the prices are marked up for the user to see, they are stored in the checkbox `value` properties. The checkboxes have also been given names although the names are not used in this example. (Recall the note at the end of Section 3.6.) In contrast, the names of the radio buttons are necessary for this example since they form a unique selection group. The radio button `values` store information that is used to adjust the order total based upon payment type. (If any of our students pay with a money order, they lose a letter grade!) Aside from the command button which calls a `total()` function to process the form, the only other point of interest is the text area which is to serve as the display for the shopping cart.



Please buy some stuff!
☑ Item1 $14.99
☑ Item2 $12.99
□ Item2 $13.99
□ Item4 $14.99

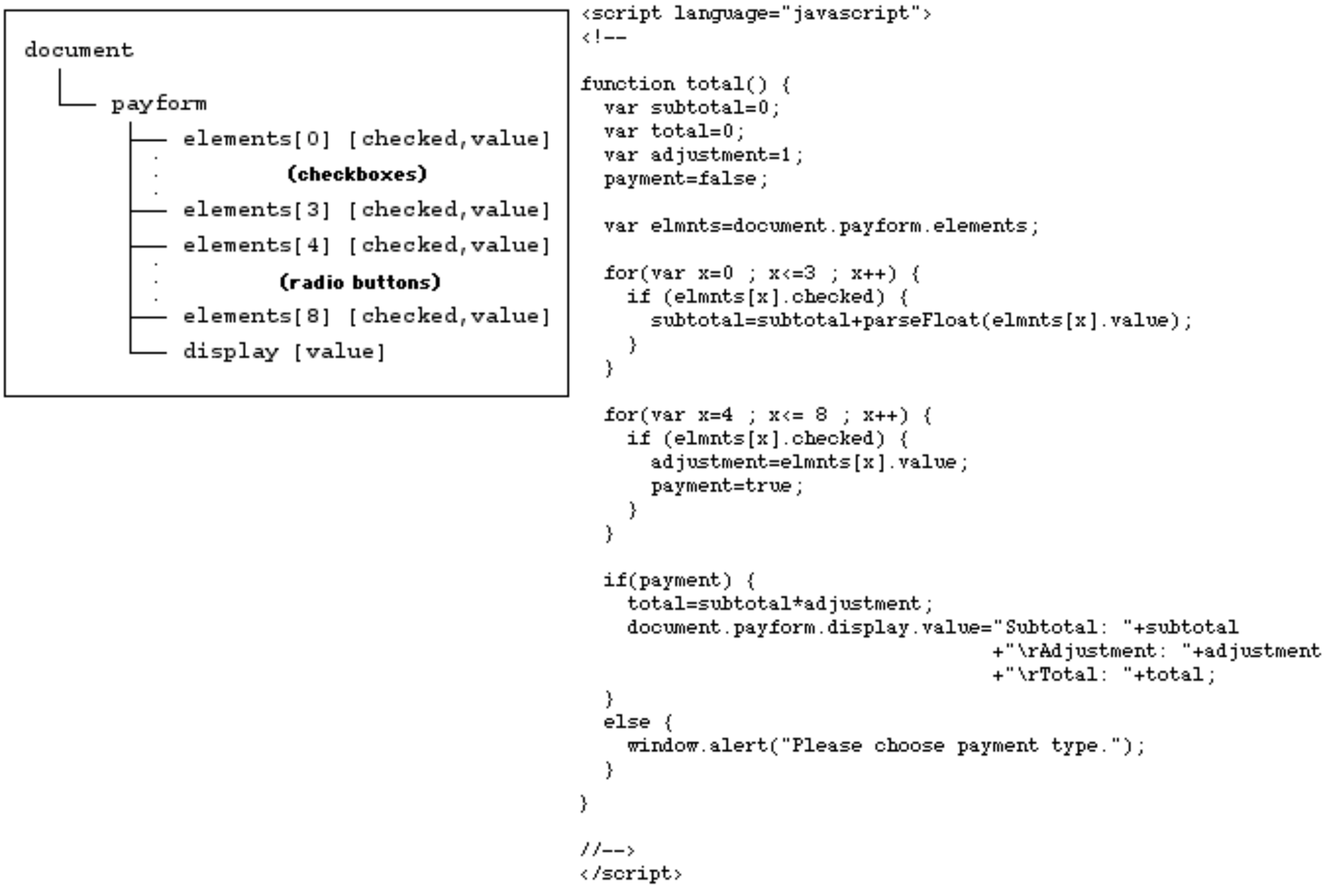
Choose Payment Method
○ Money Order (20% service charge)
○ Personal Check (10% service charge)
● Visa (Preferred--20% discount)
○ MasterCard (10% discount)
○ Discover (10% discount)

```
Subtotal: 27.98
Adjustment: .8
Total: 22.384
```

```
<form name="payform">
  <table border="0" cellpadding="5">
    <tr>
      <td width="250" valign="top">
          <b>Please buy some stuff!</b><br />
        <input type="checkbox" name="item" value="14.99" /> Item1 $14.99<br />
        <input type="checkbox" name="item" value="12.99" /> Item2 $12.99<br />
        <input type="checkbox" name="item" value="13.99" /> Item2 $13.99<br />
        <input type="checkbox" name="item" value="14.99" /> Item4 $14.99<br /><br />
          <b>Choose Payment Method</b><br />
        <input type="radio" name="pay" value="1.2" /> Money Order (20% service charge)<br />
        <input type="radio" name="pay" value="1.1" /> Personal Check (10% service charge)<br />
        <input type="radio" name="pay" value=".8" /> Visa (Preferred--20% discount)<br />
        <input type="radio" name="pay" value=".9" /> MasterCard (10% discount)<br />
        <input type="radio" name="pay" value=".9" /> Discover (10% discount)
          <br /><br />
        <input type="button" value="Process Order" onclick="total()" />
        <input type="reset" value="Reset Form" />
      </td>
      <td width="200" valign="bottom">
        <textarea name="display" rows="5" cols="35"></textarea>
      </td>
    </tr>
  </table>
</form>
```

Figure 3.13 The form for a client-side shopping cart.

We now explain the JavaScript support for this example, which is shown in Figure 3.14 together with an object model for the form. Since we will be iterating over the checkboxes and radio buttons, we have shown the `elements[]` array. Besides, we don't have unique names with which to work anyway. We do refer to the display area using the name we have given it. That way we don't have to worry about its index position in the `elements[]` array.



document
payform
elements[0] [checked,value]
(checkboxes)
elements[3] [checked,value]
elements[4] [checked,value]
(radio buttons)
elements[8] [checked,value]
display [value]

```
<script language="javascript">
<!--

function total() {
  var subtotal=0;
  var total=0;
  var adjustment=1;
  payment=false;

  var elmnts=document.payform.elements;

  for(var x=0 ; x<=3 ; x++) {
    if (elmnts[x].checked) {
      subtotal=subtotal+parseFloat(elmnts[x].value);
    }
  }

  for(var x=4 ; x<= 8 ; x++) {
    if (elmnts[x].checked) {
      adjustment=elmnts[x].value;
      payment=true;
    }
  }

  if(payment) {
    total=subtotal*adjustment;
    document.payform.display.value="Subtotal: "+subtotal
                                +"\rAdjustment: "+adjustment
                                +"\rTotal: "+total;
  }
  else {
    window.alert("Please choose payment type.");
  }
}

//-->
</script>
```

Figure 3.14 An object model and JavaScript support for the form of Figure 3.13.

The first four local variables are to help with the calculations and validation. We will see what those do as we talk through the function. The first main feature is the statement

```
var elmnts=document.payform.elements;
```

This statement creates the local variable `elmnts` and assigns it the form's `elements[]` array.

NOTE

Here you see just how loose JavaScript's typing of variables is. Not only can any literal type be assigned, but here we have assigned an array object to the variable. Objects are assigned by reference in JavaScript, rather than by value. (We saw this in Section 3.2 when we passed an array to a function.) So the local variable `elmnts` is merely a reference (or pointer) to the actual `elements[]` array of the form. Thus `elmnts[0].checked` accesses the Boolean state of the first checkbox in the form, for example.

We made that assignment in order to have a concise reference to the forms data, dispensing with repetitive long object references. But contrast the assignment here with the ones in Figure 3.11. Those assign the actual primitive data values of the form to local variables. Those primitive assignments transfer the literals by value rather than by reference.

The first loop tests all the checkboxes, adding the parsed price of any selected ones onto the local `subtotal` variable. The second loop tests all of the radio buttons, storing the chosen payment adjustment into the local `adjustment` variable. We also see the form's validation feature in that loop. The local `payment` variable was initialized to `false` when it was declared. Only if one of the payment options is found to be checked, is payment set to `true`.

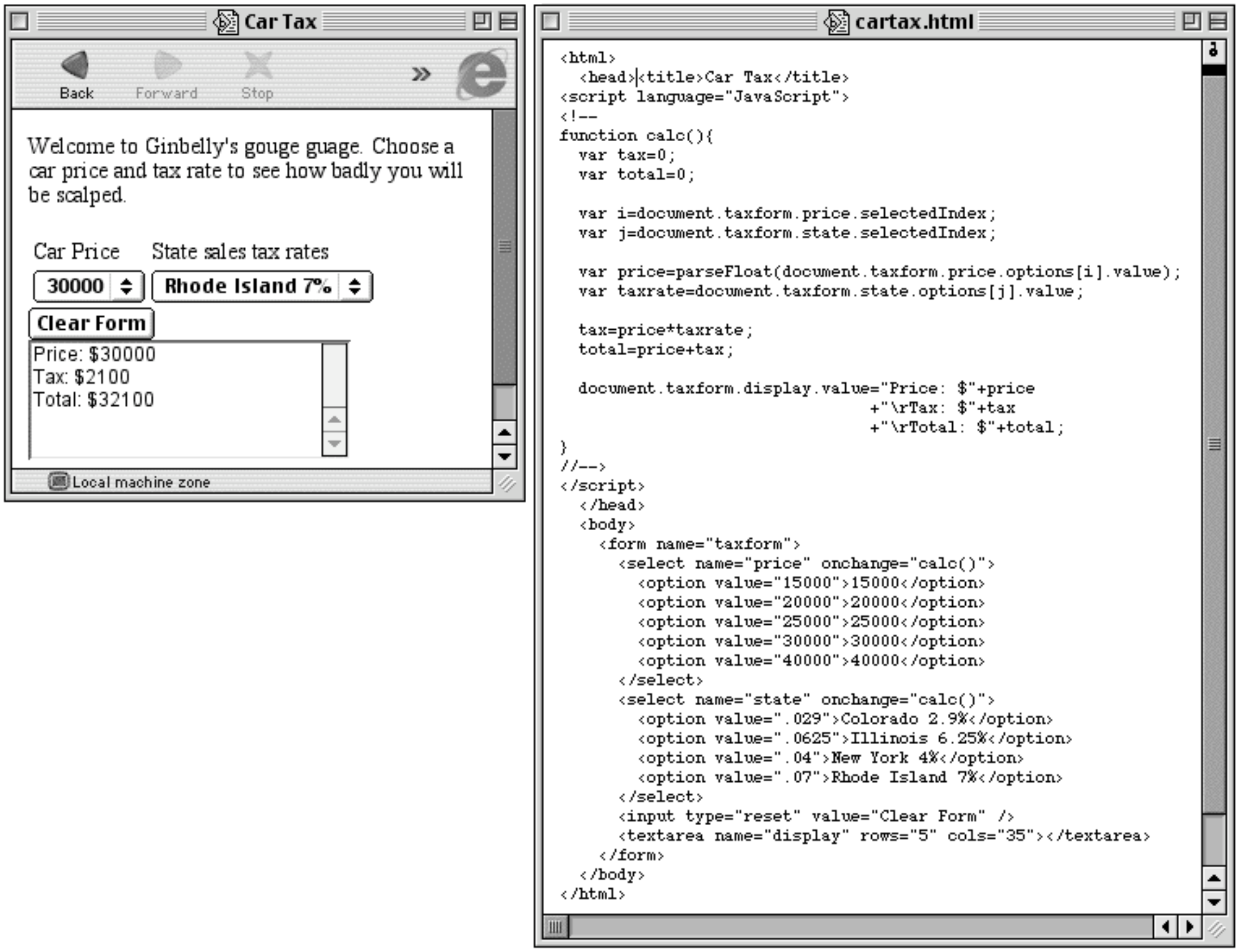
The remainder of the function is straight-forward. We illustrate the logic of the conditional with pseudo-code.

```
if(user has selected a payment option) {
  calculate the total
  update the display
}
else {
   give an alert
}
```

If no payment option is chosen, the display is left blank in favor of the alert. There is one last detail to be explained. Test areas display plain text, not HTML. Thus, if you want a line return in a text area, `<br />` will be of no help. In Figure 3.14, you can see that line breaks were forced inside the long string using the carriage **r**eturn escape character `\r`. Note that the **n**ew line character `\n` may also be used, but we have found that less reliable among all the browser versions than the carriage return. Go figure.

The last example of this section demonstrates how to deal with pop-up menus on the client. Figure 3.15 shows both the source code and rendition of a Web page that features a simple utility with two pop-up menus. Some HTML formatting detail has been removed from the source code to keep it as uncluttered as possible. However, the essential code for the form and the JavaScript code is all there. You can see in the code for the form that we have named all of the form elements. The form is pretty simple, so we do not supply an object diagram.

In contrast to the previous examples, this one does not use a command button to trigger the processing of the form. Rather, each menu is endowed with the `onchange` event handler. As soon as a new choice is made in either menu, the `calc()` function is called. That function computes the sales tax on the dollar amount given as the current choice in the first menu, based on the tax rate that is given as the current choice in the second menu.



Welcome to Ginbelly's gouge guage. Choose a car price and tax rate to see how badly you will be scalped.

Car Price: 30000

State sales tax rates: Rhode Island 7%

Clear Form

```
Price: $30000
Tax: $2100
Total: $32100
```

```
<html>
  <head><title>Car Tax</title>
<script language="JavaScript">
<!--
function calc(){
  var tax=0;
  var total=0;

  var i=document.taxform.price.selectedIndex;
  var j=document.taxform.state.selectedIndex;

  var price=parseFloat(document.taxform.price.options[i].value);
  var taxrate=document.taxform.state.options[j].value;

  tax=price*taxrate;
  total=price+tax;

  document.taxform.display.value="Price: $"+price
                              +"\rTax: $"+tax
                              +"\rTotal: $"+total;
}
//-->
</script>
  </head>
  <body>
    <form name="taxform">
      <select name="price" onchange="calc()">
        <option value="15000">15000</option>
        <option value="20000">20000</option>
        <option value="25000">25000</option>
        <option value="30000">30000</option>
        <option value="40000">40000</option>
      </select>
      <select name="state" onchange="calc()">
        <option value=".029">Colorado 2.9%</option>
        <option value=".0625">Illinois 6.25%</option>
        <option value=".04">New York 4%</option>
        <option value=".07">Rhode Island 7%</option>
      </select>
      <input type="reset" value="Clear Form" />
      <textarea name="display" rows="5" cols="35"></textarea>
    </form>
  </body>
</html>
```

Figure 3.15 Processing pop-up menus on the client.

Aside from two local variables used to store results from the computations, the first noteworthy lines of JavaScript code store indices of the currently chosen menu options into the local variables `i` and `j`. Next, the `values` for the selected menu options are stored into the local variables `price` and `taxrate`. We hate to be anti-climactic, but there is little else to it besides a couple of calculations and updating the display area.

While similar in functionality to a unique selection group of radio buttons, it is much easier to extract the value of the selected menu option in a client-side script. You have to loop over the radio buttons until the selected one is found. However, the `selectedIndex` property of a menu essentially does that for you. The general strategy is given below. While the object references are ugly, the concept is simple. The variable `i` is the selected index, so grab the value of the `i`th `option.`

```
i=document.formName.menuName.selectedIndex;
thevalue=document.formName.menuName.options[i].value;
```

There was no compelling reason to use the `onchange` event handler instead of a command button other than for a change of pace. Certainly, a command button would have worked nicely, perhaps even better. For example, in the `onchange` version, you can't do the calculation for the two menu options that appear when the page first loads without making a change to a menu, and then making a change back to the original option. `onchange` does work with checkboxes and radio buttons as well, and one can sometimes find good uses for it.

NOTE

If you played with the examples of this section, you may have noticed that the results from some of the calculations were not rounded off very well. Occasionally, 16 decimal places result, and other times there are three decimal places for a dollar amount where it would be customary to include only two. There is a stand-alone Math object whose methods can be called to round numbers or to help with calculations in other ways. Its methods can be found in Appendix A. The Math.round() method rounds to the nearest integer. So to round to two decimal places, multiply by 100, round, and then divide by 100. For example, the following code results in the variable num containing 3.15.

```
var num = 3.141592654;
num = Math.round(100*num)/100;
```

3.8 Form Validation before Submission

The calculator example of Figure 3.10 validated the value entered into a text field to prevent division by 0. The shopping cart simulation in Figure 3.12 validated a group of radio buttons to ensure that a payment method was chosen. But the real flavor of client-side validation of form data for Web applications involves testing the data (under various constraints) before submitting it to a program on a Web server. In that case, we no longer use a command button, but rather a submit button.

The Strategy for Validation Before Submission

The general strategy for client-side validation before submission involves the onsubmit event handler of the form object.

```
document.formName.onsubmit
```

This event handler expects a Boolean return value. So we must formally assign it the function we create to handle the event of the user clicking the submit button.

```
document.formName.onsubmit=validate;

function validate() {
   statements to test the form data for validity
   return false if the data fails, true if it passes
}
```

If the self-defined validation function returns true, the event handler automatically calls the forms submit method. If it returns false, the event handler simply does not submit the form. In that case, the function would alert the user that the entered data is faulty. Note that, this event handler can be called as an HTML attribute of the submit button (onsubmit="validate()"), but that renders it's return value ineffective as per the discussion in Section 3.5.

This strategy is contingent upon assigning the function to the event handler in the HTML document **after** the form that is to be submitted is defined. The browser must read the HTML code for the form and create the corresponding form object before its onsubmit event handler is available to be assigned a value. Hence, we will no longer be able to place the validation scripts in the head section of the document. That will be apparent in the examples that follow. Also, the examples of this section and the next are submitted the same echo.cgi program to which we alluded in the note at the end of section 3.1. It simply echoes the submitted name=value pairs back to the Web browser in a Web page. (It won't be long before we learn how to do meaningful calculations with the data on the back end!)

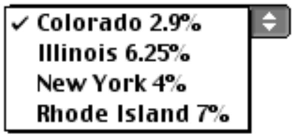
We have introduced formal binding of functions to event handlers by assignment because that works in all cases. But there is one "shortcut" way to ensure that the `onsubmit` event handler, placed as an HTML attribute, receives a value returned from a function. Simply place the `return` command before the function call.

```
<input type="submit" onsubmit="return validate();"
value="Submit Form" />
```

We mention this because you will see this "shortcut" employed it you look at much JavaScript code on the Web.

Menu Validation

We first turn to pop-up menus since validation for those is relatively simple. If no special provisions are provided in a menu, no validation is required. Consider again the sales tax menu used in Figure 3.15.



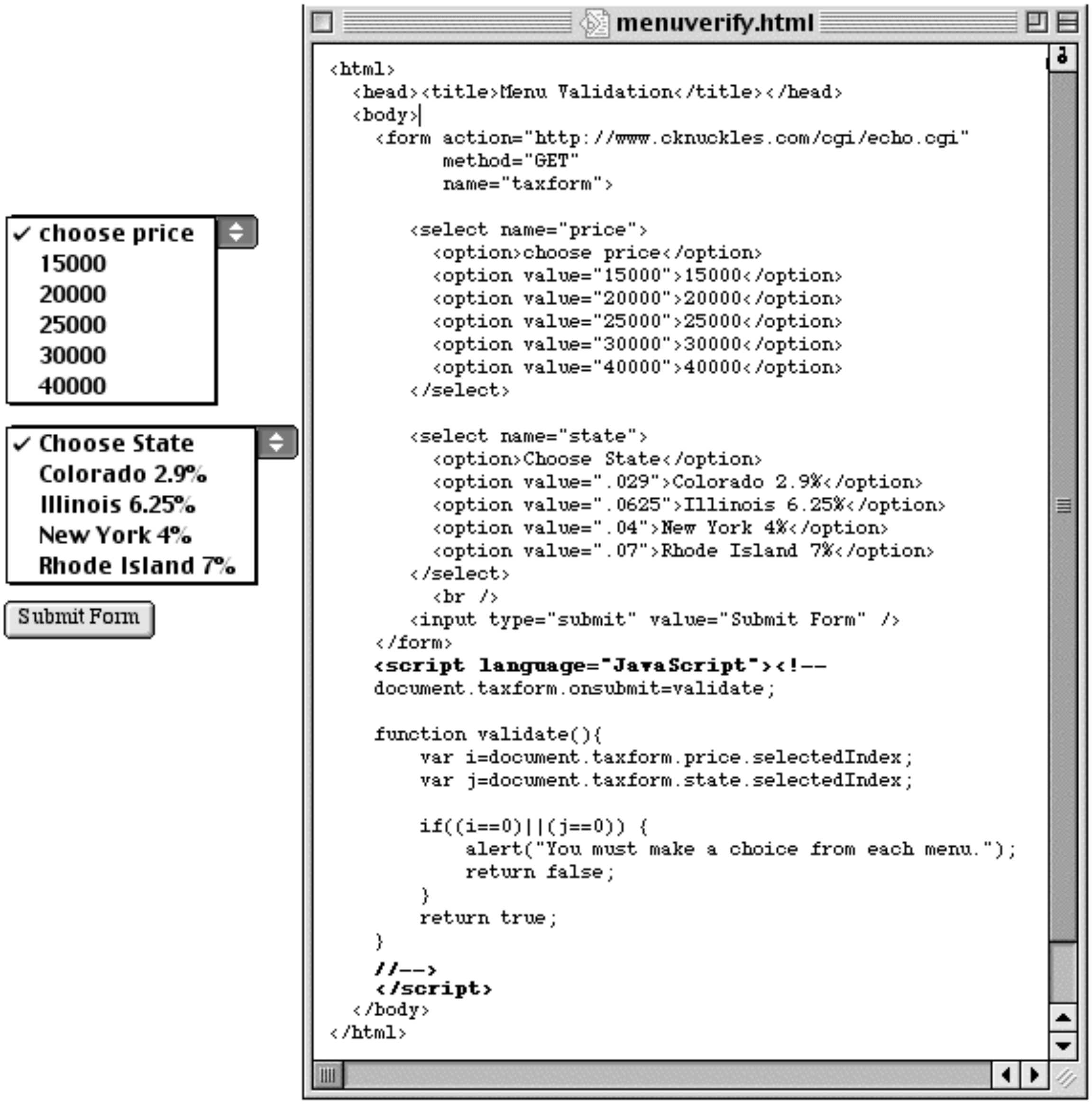
There is nothing to verify. When the page first loads, Colorado is the visible and chosen option, being the first one defined in the menu. If the user makes no change to the menu, Colorado remains chosen. Thus, the first option defined in a menu is the default selection for submission. (Recall that the `selected` attribute can also be placed in a menu option to make it the default selection.)

Suppose that the menu is supplied in a form used for online purchases so that the user can choose the appropriate state tax rate used to assess the sales tax. (Eventually, they will probably tax online purchases!) In such a case, it is not desirable for the menu to have a default selection. For example, if someone from Rhode Island forgets to choose a state, the default Colorado would be used. A better solution is to force the user to make a selection. The strategy for that is to make the menu's first option a "dummy" option. That way, the user gets

when the page first loads. Then, supply JavaScript validation so that the form is not submitted if the user forgets to make a selection. That is, give an alert if the user clicks the submit button, but the `selectedIndex` of the menu still contains 0.

Figure 3.16 uses that strategy in a second version of the car tax form of Figure 3.15. This version does not do any calculations with the data, and has no text area. Rather, it verifies that the user has made a selection from both of the menus. If the form fails the validation, the user is alerted. If the form passes the validation, the form's data is submitted to the "echo" program on the server.

First note that the form has been supplied with the `action` attribute whose value gives the URL of the CGI program to which the form's data is submitted. Also, `method="GET"` specifies that the name value pairs be sent to the server as a query string appended to the URL.



```
<html>
  <head><title>Menu Validation</title></head>
  <body>
    <form action="http://www.cknuckles.com/cgi/echo.cgi"
          method="GET"
          name="taxform">

       <select name="price">
         <option>choose price</option>
         <option value="15000">15000</option>
         <option value="20000">20000</option>
         <option value="25000">25000</option>
         <option value="30000">30000</option>
         <option value="40000">40000</option>
       </select>

       <select name="state">
         <option>Choose State</option>
         <option value=".029">Colorado 2.9%</option>
         <option value=".0625">Illinois 6.25%</option>
         <option value=".04">New York 4%</option>
         <option value=".07">Rhode Island 7%</option>
       </select>
         <br />
       <input type="submit" value="Submit Form" />
    </form>
    <script language="JavaScript"><!--
    document.taxform.onsubmit=validate;

    function validate(){
        var i=document.taxform.price.selectedIndex;
        var j=document.taxform.state.selectedIndex;

        if((i==0)||(j==0)) {
            alert("You must make a choice from each menu.");
            return false;
        }
        return true;
    }
    //-->
    </script>
  </body>
</html>
```

Figure 3.16 Client-side validation for pop-up menus.

The first statement in the script assigns the `validate` function to the `onsubmit` event handler. Again, we stress that it is important that the assignment occur after the form has been defined. (The actual function could be placed in the head section.) The validation function first extracts the indices of the selected menu options. If either index is still 0, a false value is returned to the event handler, causing the submit action to be aborted. Otherwise, a true value is returned to the event handler causing the form to be submitted to the echo program on the server.

NOTE

In a multiple selection menu, most browsers assign `selectedIndex` the least index among multiple selected options or `–1` if none are chosen. So `selectedIndex` easily detects whether at least one option is chosen. But sometimes one needs to validate that exactly two selections are made, for example. The Boolean `selected` property of each menu option

document.formName.menuName.options(i).selected is used in that endeavor. It is straightforward to iterate over the `options()` array and determine the status of each menu option.

A Larger Validation Example

We now apply the validation before submission strategy to a second version of the client-side shopping cart of Figure 3.13. The HTML portion of the new payment form is shown in Figure 3.17. The new version replaces the text area with three text fields to collect shipping information from the user. The following three things are validated on the client before the form is allowed to be submitted.

1. The user must choose (at least) one item.
2. The user must choose a payment type.
3. The user must not leave any of the text fields blank.

The JavaScript function that implements the validation is supplied in Figure 3.18 along with an object diagram for the form. We have not shown the whole document this time, but the script in Figure 3.18 appears in the document after the definition of the HTML form.

The general strategy is the same: formally assign the validation function to the `onsubmit` event handler of the form. That is the first line of the script. Following that is the validation function. Its first line assigns the reference to the form's `elements[]` array to the local variable `elmnts`.

Next we loop through the checkboxes, setting the `buy` variable to true if one of them is found to be checked. If one of them is not found to be checked, the user is alerted and the function returns `false` to the `onsubmit` event handler. Ensuring that one of the radio buttons is checked is handled similarly.

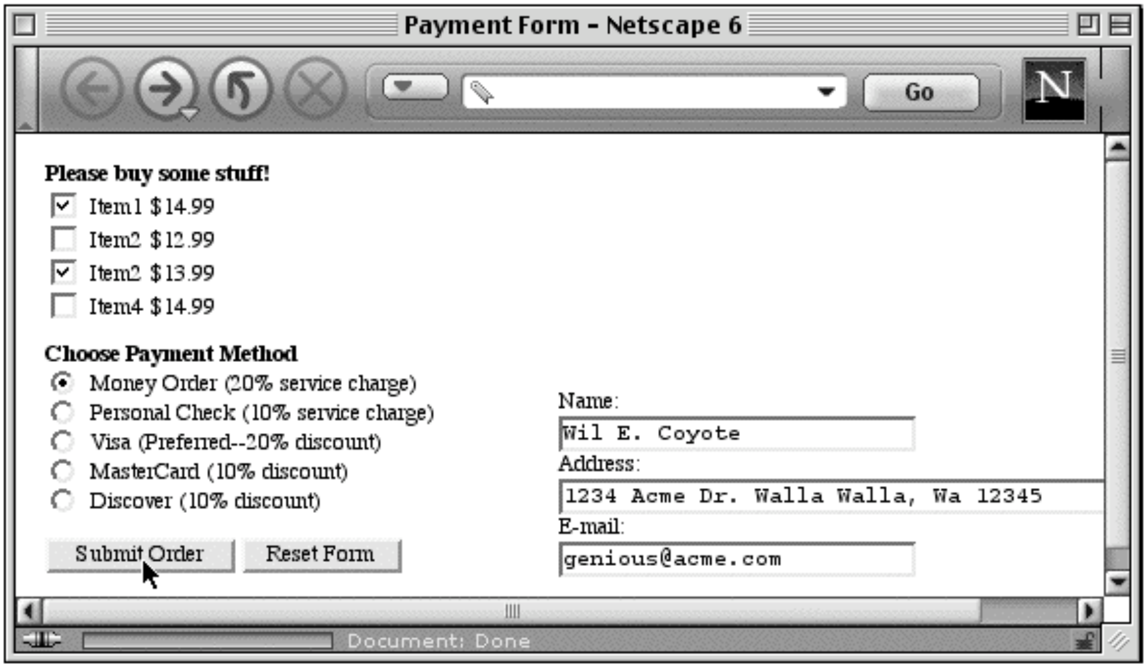
Next we loop through the text fields. If one of them is found to be empty, the user is alerted, that particular text field is brought into focus, it is selected, and the function returns `false`. (Recall the form methods near the end of Section 3.6.) If all of the text fields pass the test, the function finally progresses to the statement that returns `true`, allowing the form to be submitted.

NOTE

There are two things to note here. First is the technique used in the validation function. The same technique was used in Figure 3.15, although there was only one validation performed. Test each form element (or group of elements) separately. If the test fails, return `false`. There is no need to test any further form elements in that case. Only when all tests pass can the function reach the line that returns `true`.

Second, testing for the empty string is a pretty lame validation test. For example, if some joker enters a single white space into one of the text fields, that field will not cause the alert to be triggered. Also, in this case, the `select()` method will have no effect. The only time that method is called is when a text field is empty. But in that case, the browser will not highlight the field since there is no text in it. However, browsers will bring focus to an empty text field. The next section explores better verification techniques for text fields. In the mean time, the focusing and selecting the erroneous field is a good habit to get into.

Just in case you don't have a computer handy to submit the working version on the Web site, Figure 3.19 shows the data returned by the "echo" program when the form is filled out as shown in Figure 3.17 and submitted.



Please buy some stuff!

☑ Item1 $14.99

□ Item2 $12.99

☑ Item2 $13.99

□ Item4 $14.99

Choose Payment Method

● Money Order (20% service charge)

○ Personal Check (10% service charge)

○ Visa (Preferred--20% discount)

○ MasterCard (10% discount)

○ Discover (10% discount)

Submit Order | Reset Form

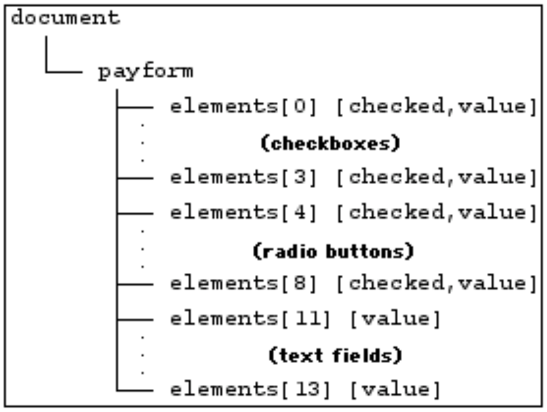
Name: Wil E. Coyote

Address: 1234 Acme Dr. Walla Walla, Wa 12345

E-mail: genious@acme.com

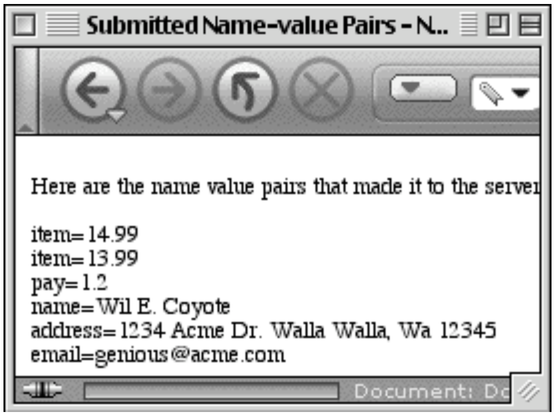
```
<form action="http://www.cknuckles.com/cgi/echo.cgi" method="GET" name="payform">
  <table border="0" cellpadding="5" width="600">
    <tr>
      <td width="250" valign="top">
          <b>Please buy some stuff!</b><br />
        <input type="checkbox" name="item1" value="14.99" /> Item1 $14.99<br />
        <input type="checkbox" name="item2" value="12.99" /> Item2 $12.99<br />
        <input type="checkbox" name="item3" value="13.99" /> Item2 $13.99<br />
        <input type="checkbox" name="item4" value="14.99" /> Item4 $14.99<br /><br />
          <b>Choose Payment Method</b><br />
        <input type="radio" name="pay" value="1.2" /> Money Order (20% service charge)<br />
        <input type="radio" name="pay" value="1.1" /> Personal Check (10% service charge)<br />
        <input type="radio" name="pay" value=".8" /> Visa (Preferred--20% discount)<br />
        <input type="radio" name="pay" value=".9" /> MasterCard (10% discount)<br />
        <input type="radio" name="pay" value=".9" /> Discover (10% discount)
          <br /><br />
        <input type="submit" value="Submit Order" />
        <input type="reset" value="Reset Form" />
      </td>
      <td valign="bottom">
          Name:<br />
        <input type="text" name="name" size="25" /><br />
          Address:<br />
        <input type="text" name="address" size="40" /><br />
          E-mail:<br />
        <input type="text" name="email" size="25" /><br />
      </td>
    </tr>
  </table>
</form>
```

Figure 3.17 A payment form that submits the data to a Web server only if it passes some client-side validation


document
payform
elements[0] [checked,value]
(checkboxes)
elements[3] [checked,value]
elements[4] [checked,value]
(radio buttons)
elements[8] [checked,value]
elements[11] [value]
(text fields)
elements[13] [value]

```
<script language="javascript">
<!--
document.payform.onsubmit=validate;

function validate() {
    var elmnts=document.payform.elements;

    var buy=false;
    for(var x=0 ; x<=3 ; x++) {
        if (elmnts[x].checked) {
             buy=true;
        }
    }
    if(!buy) {
        alert("You must buy something.");
        return false;
    }

    var pay=false;
    for(var x=4 ; x<= 8 ; x++) {
        if (elmnts[x].checked) {
            pay=true;
        }
    }
    if(!pay) {
        alert("You must choose a payment method.");
        return false;
    }

    for(var x=11 ; x<= 13 ; x++) {
        if (elmnts[x].value == "") {
            alert("Required field.");
            elmnts[x].focus();
            elmnts[x].select();
            return false;
        }
    }
    return true;
}

//-->
</script>
```

Figure 3.18 The JavaScript support for the form of Figure 3.16.



Here are the name value pairs that made it to the server

item= 14.99
item= 13.99
pay= 1.2
name= Wil E. Coyote
address= 1234 Acme Dr. Walla Walla, Wa 12345
email=genious@acme.com

Figure 3.19 The data returned by the server-side "echo" program from the form submission pictured in Figure 3.17

3.9 Text Validation with the String Object

We conclude this lesson by using the String object to provide somewhat more rigorous validation of text input. While we have spoken often of string literals, strings also have an object context. A string not explicitly assigned to a variable is a string literal. In the statement

```
document.write("Scooby Doo");
```

the string `"Scooby Doo"` is not an object. However, once assigned to a variable,

```
var str="Scooby Doo";
```

it is an object. Any string stored in a variable may use the `length` property or any of the methods of the String object. We mention only one property because there is only one. Using the above declaration, the assignment

```
var x=str.length;
```

causes the variable `x` to contain the integer 10.

There are several methods that can be called on strings, but we discuss only those used in the following validation example. The rest are listed in Appendix A. It should be no problem to figure out how to use those if the need arises. First, the `charAt()` method takes an integer parameter and returns the character at that position in the string. A string is an array of characters that uses standard array indexing. So the first character in a string has index 0. The assignment

```
x=str.charAt(1);
```

causes `x` to contain the string "c" and

```
x=str.charAt(6);
```

causes `x` to contain the string " " consisting of one white space. There is no character type in JavaScript. Characters are simply one character strings.

The `indexOf()` method takes a one character string as its argument and returns the first index at which the character is found. If the character is not found in the string, `indexOf()` returns `–1`. For example,

```
x=str.indexOf("o");
```

causes `x` to contain the integer 2 and

```
x=str.indexOf("O");
```

causes `x` to contain the integer `–1`. In contrast, the `lastIndexOf()` method takes a one character string, and returns the last index at which the character is found in the string. It also returns `–1` if the character is not found in the string.

Below, we create a self-defined function that counts the number of occurrences of a given character in a string. We will make heavy use of it in the main example of this section. The function takes two parameters, the string to be tested and a one character string corresponding to the character to count. It returns the number of occurrences of the character in the string.

```
function countchar(thestring,thechar) {
  var count=0;
  for(var x=0 ; x<thestring.length ; x++) {
    if(thestring.charAt(x)==thechar) {
      count++;
    }
  }
  return count;
}
```

The `countchar()` function loops through `thestring` one character index at a time (i.e. from character `0` to character `thestring.length-1`). If the character at the current index matches `thechar`, the counter is incremented. After the loop has run its course, the counter value is returned.

Figure 3.20 shows the form for which we will employ the counting character function. The form is rather simple, and we have only provided an object diagram so that you can easily follow the object references that are to follow in the validation code. The form is to be backed up with the following client-side validation features.

1. The name must be at least 4 characters long, must contain at least one blank space, and must neither begin nor end with a space.
2. The e-mail address must be at least 5 characters long, must contain no blank spaces, must contain exactly one @ character, and must contain at least one period.
3. The zip code must be numeric and exactly five characters long.

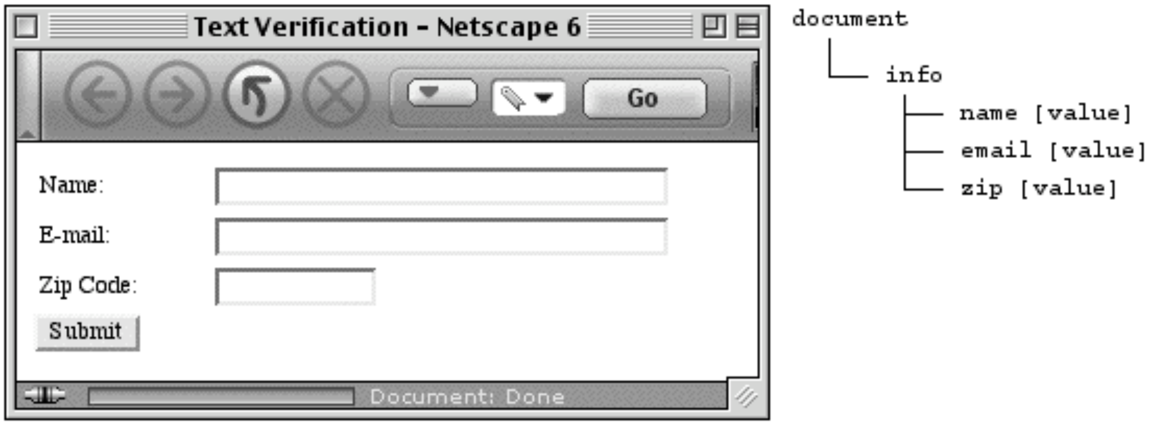


Figure 3.20 An information collection form

These restrictions allow for a minimal full name like `"Ty X",` an e-mail address basically of the form `"x@y.z",` and a standard zip code. Note that the e-mail validation does not rule out something like `"@x.y.z".` Some more stringent validations are left as exercises. Also, note that a name with at least one blank space rules out a numeric value since `isNaN("1 23")` returns `true`, for example. For the same reason, a zip code that must be numeric rules out a five character erroneous zip like `"123 5".`

Figure 3.21 displays the complete JavaScript validation support that implements the list of constraints. Again, because of the formal assignment of the validation function to the form's `onsubmit` event handler, the script follows the definition of the HTML form in the complete document.

We have already discussed the `countchar()` function. Following that is the validation function. The first block of code stores the name field of the form into a local variable and then performs the specified validation tests. If any of the tests fail, the user is alerted, the text field is focused, any text in the fields is selected, and the function returns `false` to the event handler. The next two blocks of code perform the specified tests on the e-mail and zip fields, respectively. The validation function returns true causing the form to be submitted only if each of the three fields passes its test.

```
<script language="JavaScript">
<!--
document.infoform.onsubmit=validate;

function countchar(thestring,thechar) {
 var count=0;
    for(var x=0 ; x<thestring.length ; x++) {
       if(thestring.charAt(x)==thechar) {
       count++;
       }
    }
    return count;
}

function validate() {

 var name=document.infoform.thename.value;
 if(   (name.length<4) ||
         (countchar(name," ")==0) ||
       (name.indexOf(" ")==0) ||
       (name.lastIndexOf(" ")==(name.length-1)) ) {

    alert("Invalid name format.");
    document.infoform.thename.focus();
    document.infoform.thename.select();
    return false;
 }

 var email=document.infoform.email.value;
 if(   (email.length<5) ||
         (countchar(email," ")>0) ||
         (countchar(email,"@")!=1) ||
         (countchar(email,".")==0) ) {

       alert("Invalid email format.");
       document.infoform.email.focus();
       document.infoform.email.select();
    return false;
 }

 var zip=document.infoform.zip.value;
 if( isNaN(zip) || (zip.length!=5) ) {

    alert("Invalid zip format.");
     document.infoform.zip.focus();
     document.infoform.zip.select();
    return false;
 }

 return true;
}
//-->
</script>
```

Figure 3.21 The JavaScript support for the form of Figure 3.20.

3.10 Summary of Key Terms and Concepts

Form Control Buttons – Submit and **Reset buttons** have built-in event handlers which submit a form to the server and return the form to its state when the page first loaded, respectively. **Command** (generic) **buttons** must be supplied with the `onclick` event handler, which calls a self-defined JavaScript function to handle the event in a customized way.

Form Text Elements -- Text fields (single line) and **text areas** (multiple line) are text entry boxes. Their values, which appear in the boxes, can be hardcoded into the HTML `value` attribute. When the user enters text into a box, that text becomes the current `value` of the form element.

Form Option Buttons – A **checkbox** can be both checked and unchecked by the user. A group of checkboxes forms an option list where any number of options can be chosen. A single **radio button**, once checked by the user, can't be manually unchecked by the user. A group of radio buttons, each given the same name, form a *unique selection group* where only one can be selected at a given time. The `value` property of a checkbox or radio button holds "hidden" data which relates to that option.

Form Menus – A **single selection menu** functions like a unique selection group of radio buttons. A **multiple selection menu** functions like a group of checkboxes. The `value` property of an option of either menu type holds "hidden" data which relates to that option.

Names and Values – The names of form elements are used both for object reference in JavaScript and to form `name=value` pairs for submission to the Web server. Whether on the client or server, the values carry the data for the form elements.

The Browser Object – Formally, the *Document Object Model Level 0*, the Browser Object is a Web browser's in memory (RAM) representation of a rendered Web page and the window that contains it. When a Web page first loads, the properties of the Browser Object are initialized according to the HTML markup instructions. After the page is fully loaded, JavaScript can be used to change the state of the Browser Object, thereby changing the rendered state of the Web page. In particular, the data contained in an HTML form is accessible to JavaScript through the Browser Object.

JavaScript -- A programming language, syntactically similar to C++ and Java, that is used primarily to script Web browsers. Typical use is to place JavaScript statements in functions, which are called upon a user event, and which typically manipulate the Browser Object in some way (by changing it's properties or calling its methods). For that reason, it is sometimes called an *object based* language. JavaScript (at the present) lacks formal object classes and inheritance of fully object oriented languages.

Loosely typed – JavaScript's variables are loosely typed, which means that any of the three basic literal types (number, string, Boolean) can be stored into a given variable, interchangeably. Type conversions are hendled "on the fly" by the JavaScript interpreter as necessary when evaluating expressions. JavaScript has special literals (`NaN, infinity, undefined, null`) that are stored into variables when left no alternative due an unresolvable type mismatch. However, the built-in primitive variables bound to the Browser Object are of fixed types. The most noteworthy example is the `value` property of form elements, which is always a string, even if numbers are entered into the form.

Function parameters – JavaScript always passes primitive variables to function parameters by value, and objects by reference.

Events and Event Handlers – An **event** occurs when the user does something to the Web page. An **event handler** is a property of the Browser Object that "listens" for events. Some form elements (submit and reset buttons) have built in event handling capability, while others require self-defined JavaScript functions to be assigned to the event handler. Events are caused by a wide range of actions such as clicking a form control button (`onclick`), changing a form element in some way (`onchange`), submitting a form (`onsubmit`), etc.

Assigning Functions to Event Handlers – A function can be formally assigned to an event handler through the Browser Object. In that case, the event handler receives any return values from the function.

When a function is associated with an event handler by treating the event handler as an HTML attribute (i.e. `onclick="dosomething()"`), the event handler does not receive return values from the function. This suffices in most cases since most event handlers in the browser object don't care about returned values.

Client-side Form Validation – This refers to verifying that a user has entered proper data into an HTML form before it is submitted to the Web server. Client-side validation is much faster than server-side data validation since no network transaction is required. The strategy is to bind a self-defined function to the `onsubmit` event handler of a submit button. If the function returns `true`, the form is automatically submitted. If the function returns `false`, the form submission is automatically aborted. In the latter case, it is customary to alert the user in some informative way.

Client-side Utilities – The combination of JavaScript and HTML forms enable programmers to create programs (scripts) that can perform many types of computations. The result is a portable program that many different people can load into Web browsers over the internet. Common examples are portable interest and mortgage calculators, annuity calculators, forms that give unit conversions (i.e. Fahrenheit/Celsius, Gallons/Liters, etc.), and self-grading (insecure) quizzes.

3.11 Exercises

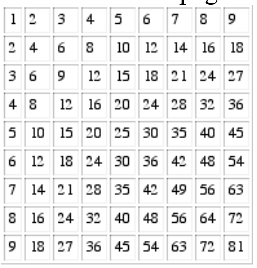
Dynamic Content

1. Create a JavaScript array that contains 10 dot com Web site names (i.e. `amazon`). As the page is first loading, use that array to print out an HTML unordered list of links pointing to the Web sites. Use the names in the array as the underlined text that appears on the links and to construct the URL's to which the links point. (Contrast with Exercise 3.)

2. Work Exercise 1 except generate a list with `n` list items, where `n` is a randomly chosen integer in the range from 5 to 10. The `n` links that appear in the list should also be chosen randomly, with no duplicate links appearing in the list.

3. Work Exercise 1, but the array should contain the full domain names of the Web sites (i.e. `www.amazon.com`). Use the domain names to construct the URLs for the links. However, only the Web site name (i.e. `amazon`) should appear as the underlined text of the link. Use the string object to extract the site names from the domain names.

4. Create a Web page that uses JavaScript to generate a multiplication table as the page is first loading.



1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

(a) Generate a `9x9` multiplication table.
(b) Generate a `nxn` multiplication table where `n` is a randomly generated integer in the range from 5 to 25.

5. Do Exercise 4, but with the following additions:
(a) The rows are given alternating background colors and the numbers in the first row and first column are rendered in boldface.

(b) The numbers in the first row and first column are rendered as boldfaced white text in table cells with a black background. Within the rest of the table, the rows should alternate background colors between white and light grey.

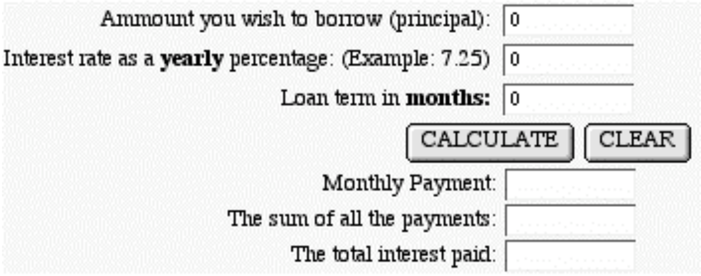
Client-side Utilities

6. Make a conversion calculator that can perform the following conversions:

```
Fahrenheit <->Celcius : F=(9/5*C)+32
Gallons <-> Liters : 1gallon=3.77 liters
Miles <-> Kilometers : 1 mile = 1.61 kilometers
```

The calculator should be able to make the conversions in both directions. Use only two text fields, one for the input and one for the converted value. You will then need one command button for each of the conversions. If a non-numeric value or a negative number is entered by the user, they should be alerted and the calculation aborted. Round all answers to two decimal places.

7. Make calculator similar to the following.



Ammount you wish to borrow (principal): 0
Interest rate as a **yearly** percentage: (Example: 7.25) 0
Loan term in **months:** 0
Monthly Payment:
The sum of all the payments:
The total interest paid:

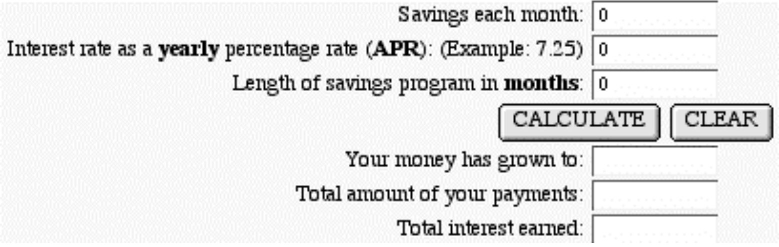
This calculates the monthly payment for a loan, based upon monthly compounding of interest, as well as the sum of payments and total interest. The formula is as follows:

$$R = P \cdot \frac{r}{1-\frac{1}{(1+r)^{n}}}$$

```
P = loan ammount (Principle)
r = monthly interest rate (yearly rate divided by 12)
n = number of months
R = monthly payment (called Rent)
```

If any non-numeric values or a negative numbers are entered by the user, they should be alerted and the calculation aborted. As an added precaution, round the number of months entered by the user to the nearest integer. Round all answers to two decimal places.

8. Make calculator similar to the following.



Savings each month: 0
Interest rate as a **yearly** percentage rate (**APR**): (Example: 7.25) 0
Length of savings program in **months**: 0
Your money has grown to:
Total amount of your payments:
Total interest earned:

This calculates the effect of saving a fixed amount of money each month over a period of time (i.e. an annuity). The savings accrue interest based upon monthly compounding of interest. The formula is as follows:

$$F = R \cdot \frac{(1+r)^{n}-1}{r}$$

```
r = monthly interest rate (yearly rate divided by 12)
n = number of months
R = monthly payment (called Rent)
F = accumulated ammount (Future value)
```

If any non-numeric values or a negative numbers are entered by the user, they should be alerted and the calculation aborted. As an added precaution, round the number of months entered by the user to the nearest integer. Round all answers to two decimal places.

9. Construct a self-grading quiz that consists of:
(i) A multiple choice question where only one answer can be selected. Use radio buttons.
(ii) A multiple choice question where multiple answers can be correct. Use checkboxes.
(iii) A multiple choice question where only one answer can be selected. Use a single selection menu.
(iv) A short answer question where the answer is only one word entered into a text field.
If the user fails to supply an answer for each question, they are alerted and the quiz is not graded. Otherwise, the correct answers appear in a text area along with the number of correct responses. Use a method of the String object (See Appendix A) to ensure that the user does not get the short answer question wrong because of the case (upper vs. lower) of the letters in the word they enter. Note that this quiz is insecure since the answers are stored in JavaScript variables on the client.

10. Make the following lottery number generator. A unique selection group of radio buttons gives options so the user can choose to generate 3, 4, 5, or 6 lottery numbers. A single selection menu allows the user to choose the maximum size (25-60) of the numbers generated. For example, if the user chooses to generate 5 numbers of maximum size 50, then five lottery numbers, each in the range from 1 to 50, are generated. The form should have 6 text fields into which the generated numbers appear when a command button is clicked. (If the user chooses to generate fewer than 6 numbers, not all of the text fields are filled.) Generate the random numbers in such a way that they are all distinct. That is, the same number should not appear twice.

11. Construct a Web page which contains 25 text fields. (The text fields can be generated as the page first loads using a JavaScript loop to write 25 text fields to the Web page.) One command button causes the text fields to be filled with 25 randomly generated integers in the range from 1 to 1000. Another command button causes the numbers to be sorted using a bobble sort. (The bubble sort algorithm can be found in most any introductory C++ or Java text, or inline after a quick search.)

Form Submission and Validation

12. Make a page with a form that includes the radio buttons and checkboxes from Figure 3.13 and the tax menu from Figure 3.16. The form should have two control buttons (besides the reset button). The first button is to "preview order" and the second is to "submit order". The preview button causes the order summary to be displayed in a text area. The submit button causes the form to be submitted to the "echo" program. Both buttons should enact the same validation: at least one item is selected for purchase, a payment method is chosen, and a tax state is chosen. If the validation fails, the user is alerted and the order is either not summarized or submitted, as the case may be. Note that the discount for payment type should be applied after tax is applied to the purchase.

13. Construct the self grading quiz of Exercise 9, but using both a command button and a submit button. When the command button is clicked, the quiz is graded if the validation constraints are met. When the submit button is clicked the quiz is submitted to the "echo" program on the server if the validation constraints are met. Otherwise the user is alerted and the form is not submitted.

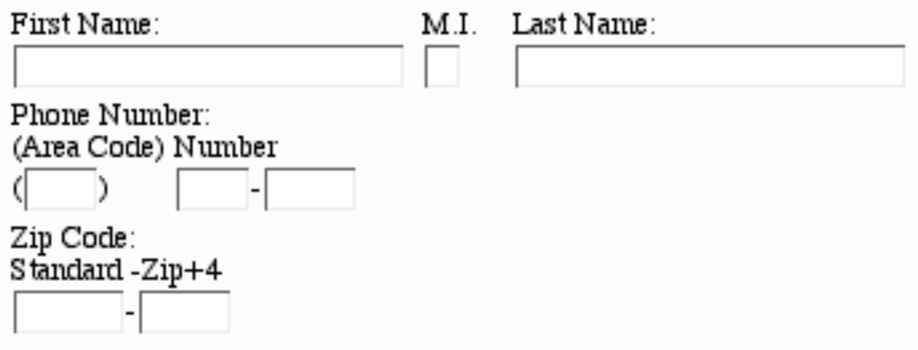
14. Construct a form for an online quiz with the following form elements and validation constraints. Submit the form to the "echo" program on the server if the validation constraints are met. Otherwise, alert the user.
(i) A text field for a first name must be filled out and contain at least 2 characters.
(ii) A unique selection group of radio buttons for a multiple choice question. An answer must be made.
(iii) A group of checkboxes for a multiple choice question that can have more than one right answer (i.e. which of the following apply?). At least one checkbox must be chosen.
(iv) A single selection menu that provides choices for a multiple choice question. The menu must be moved from the first index, which is a "dummy" choice.
(v) A multiple selection menu for a multiple choice question that can have more than one right answer (i.e. which of the following apply?). At least one option must be chosen.

15. Construct a form with two text fields, one for a user name and one for a password. When a submit button is clicked, the form is submitted to the "echo" program if validation is successful. Otherwise, the user is alerted and the form is not submitted. The validation is as follows:

- Both the user name and password must be from 4 to 8 characters in length.
- The user name must be comprised only of alpha-numeric characters (a-z, A-Z, 0-9). The user name must begin with an alphabetic character.
- The password must adhere to the same constraints as the user name with the exception that it may also contain the underscore (_) or hyphen(-), but not as the first character.

Hint: The allowed characters can be stored as one long string. You can then test each character in the user name and password against characters in that string.

16. Construct a form which collects information for a demographics survey as shown below. When a submit button is clicked, the form is submitted to the "echo" program if validation is successful. Otherwise, the user is alerted and the form is not submitted.



The validation is as follows:

Name fields:

- The first and last names must each be more than two characters, but no more than 20 characters. They must not contain any non-alphabetic characters.
- The middle initial field may be left blank. But if it is filled in, it must be exactly one alphabetic character.

Phone number fields:

- The first two fields must be exactly three characters, and the third field must be exactly four characters. All fields must contain integers.

Zip code fields.

- The first field must contain exactly five characters and must be an integer.
- The second field may be left blank, but if it's filled in, it must contain exactly 4 characters and must be an integer.

Hint: Read the hint at the end of Exercise 15.

17. Make a form with one text field for an e-mail address. When a submit button is clicked, the form is submitted to the "echo" program if validation is successful. Otherwise, the user is alerted and the form is not submitted. The following are the constraints on the e-mail address.

- It must have one @ character.
- The @ character must not be the first or last character of the address.
- It must have at least one period (.), and if it has more than one, no two periods can be consecutive (..) It can have no more than three periods. (We wish to allow x@y.a.b.c, but not x@a..b)
- The address can have no period (.) before the @.
- A period (.) must not be the first or last character of the address.

3.12 Project Thread

Here you construct two forms with client-side validation. The forms will also be used for the project thread in subsequent lessons as part of a larger Web application.

A. Work Exercise 14.

B. Work Exercise 15.

C. Add a brief description of and link to each assignment to the table in your homework page. Note that each assignment should be in a separate Web page and separate links should be added in the Homework page. If you are assigned any other exercises from this lesson, add a description(s) and link(s) to the homework page as well.

D. Upload the updated homework page and all other files to the Web server.