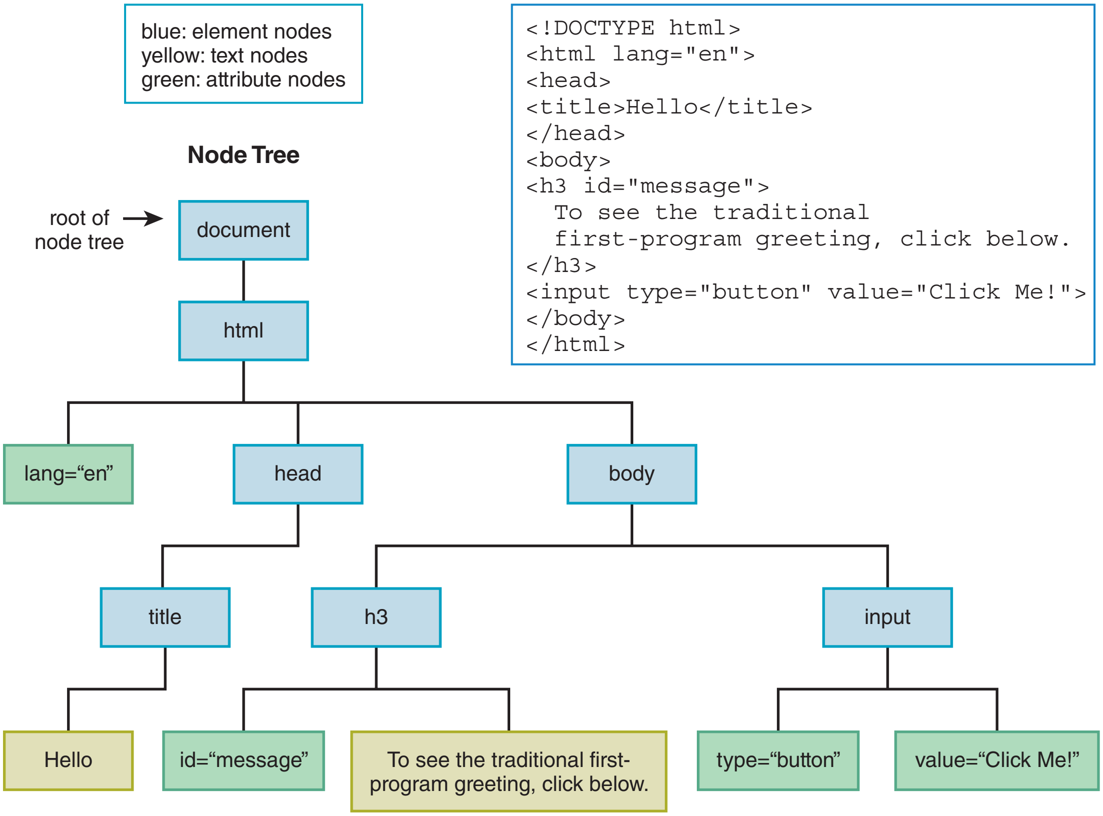
8.9 Document Object Model

Whew! We've finally finished examining the Hello web page code. The examination process required getting down in the weeds and learning about objects. Now let's step back and look at a big-picture issue related to objects. Let's examine how a web page's objects are organized.

The Document Object Model, which is normally referred to as the DOM, models all of the parts of a web page document as nodes in a node tree. A node tree is similar to a directory tree, except instead of showing directories that include other directories (and files), it shows web page elements that include other elements (and text and attributes). Each node represents either (1) an element, (2) a text item that appears between an element's start and end tags, or (3) an attribute within one of the elements. If that doesn't make sense, no worries. See the node tree example in **FIGURE 8.3**, and you should be able to understand things better by examining how the code maps to the nodes in the node tree.

The figure's code is a stripped-down version of the Hello web page code shown earlier, with some of its elements and attributes (e.g., the `meta` and `script` elements) removed. The node tree shows blue nodes for each element in the web page code (e.g., `head` and `title`). It shows yellow nodes for each text item that appears between an element's start and end tags (e.g., "Hello"). And it shows green nodes for each attribute in the web page document's elements (e.g., `h3`'s `id` attribute).



```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Hello</title>
</head>
<body>
<h3 id="message">
  To see the traditional
  first-program greeting, click below.
</h3>
<input type="button" value="Click Me!">
</body>
</html>
```

FIGURE 8.3 **Node tree for simplified Hello web page**

Note that the nodes are arranged in a hierarchical fashion, where nodes at the top contain the nodes below them (e.g., the `head` node contains the `title` node). The node at the top of the node tree is the `document` object, which we discussed earlier. Using computer science terminology, the node at the top of a tree is called the *root node*.

The term *dynamic HTML* refers to updating the web page's content by manipulating the DOM's nodes. Assigning a value to an element object's `outerHTML` property (as in the Hello web page) is one way to implement dynamic HTML. We'll see other techniques later.

The main point of explaining the DOM is for you to get a better grasp of how everything in a web page is represented behind the scenes as an object. As a web programmer, you can use the DOM's hierarchical structure to access and update different parts of the web page. The DOM provides different ways to access the nodes in the node tree. Here are three common techniques:

1. You can retrieve the node tree's root by using `document` (for the `document` object) in your code and then use the root object as a starting point in traversing down the tree.
2. You can retrieve the node that the user just interacted with (e.g. a button that was clicked) and use that node object as a starting point in traversing up or down the tree.
3. You can retrieve a particular element node by calling the `document` object's `getElementById` method with the element's `id` value as an argument.

In the Hello web page, we used the third technique, calling `getElementById`. Later on, we'll provide web page examples that use the first two techniques. We hope you're excited to know what you have to look forward to![5]

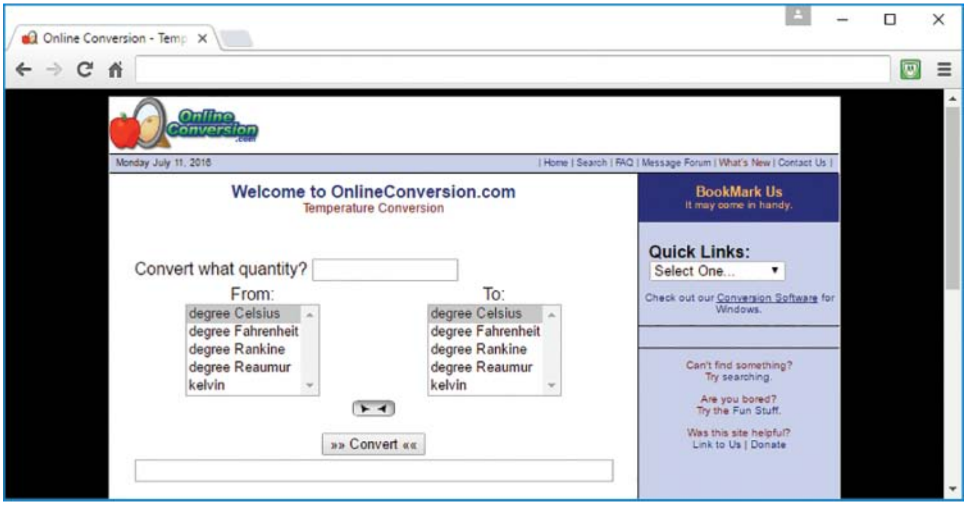
8.10 Forms and How They're Processed: Client-Side Versus Server-Side

Have you ever filled out input boxes on a web page and clicked submit in order to have some task performed, like converting miles to kilometers or buying a canine selfie stick? If so, you've used a form. A *form* is a mechanism for grouping input controls (e.g., buttons, text controls, and checkboxes) within a web page.

If you've spent much time on the Internet, you probably know that forms are very popular. So why did we wait until now to introduce them? Before this chapter, all you knew was HTML, which is very limited in terms of processing capabilities. With HTML, you can implement forms and controls, but HTML won't help you process the user's input. To make forms useful, you need to read the user's input, process it, and display the results. And to do all that, you need JavaScript.

Before we dig into the details of how to implement a form with HTML and how to process the input with JavaScript, let's look at an example web page that uses a form. **FIGURE 8.4** shows a temperature conversion calculator. Note the quantity text control at the top, the result text control at

[5] If you read about those techniques in the later chapters and that doesn't satiate your quest for knowledge, you can learn yet another technique on your own. Using a node object from the DOM node tree, you can call `getElementsByTagName` to retrieve all of the node's descendant elements that are of a particular type (e.g., all the `div` elements).



Courtesy of onlineconversion.com

FIGURE 8.4 Web page that performs temperature conversions

the bottom, the two list boxes at the sides, and the convert button in the center. All those controls are inside a form. Behind the scenes, the convert button has a JavaScript event handler. When the user clicks the button and submits the form, the event handler code reads the form's input values, does the calculation, and displays the result.

There are two basic strategies for processing a form's input data. The calculations may occur on the *client side* (on the browser's computer) or on the *server side* (on the web server's computer). With server-side processing, the form input values are transmitted across the Internet to the server computer. The server then does the calculations and transmits the answers back to the client computer. The answers are in the form of a new web page or an updated version of the original web page. With client-side processing, there's no need to go back and forth across the Internet with user input and generated results. After the web page downloads, the client computer does all the work. Therefore, client-side processing tends to be faster. So normally, you should use client-side processing for relatively simple web pages.

On the other hand, there are several reasons why server-side processing is sometimes preferable:

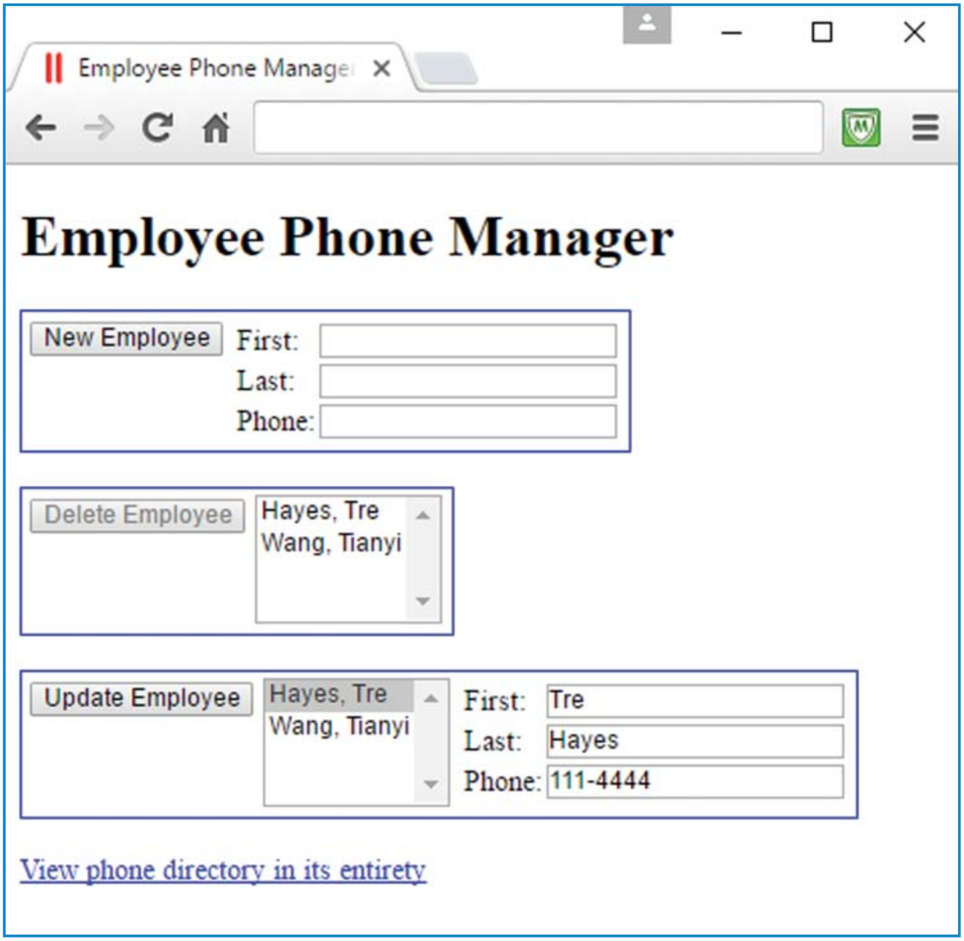
- When the calculations require a lot of programming code. If client-side processing were used, all the calculation code would have to be downloaded to the client, and that would slow things down. Slowdowns can lead to impatient users giving up and going away.
- When the calculations require the use of large amounts of data, which usually means using a database. The rationale is basically the same as for the case where there's lots of programming code. With large amounts of data, you don't want to have to download it across the Internet to the browser because that would slow things down. Therefore, you should keep the data on the server side and do all the processing there.
- When the code is proprietary. *Proprietary code* is code that gives the programmer (or, more often, the programmer's company) a competitive advantage. You should keep proprietary code on the server side, where it's more difficult for a competitor or hacker to access it.

- When the inputs and/or calculation results need to be shared by other users. In order for the data to be shared, it needs to be transmitted to the server so it can be later transmitted to other users.
- When user information needs to be processed securely behind the scenes on the server. For example, credit card numbers and passwords should be processed on the server side.

Quiz time: For Figure 8.4's temperature conversion web page, should processing take place on the client side or the server side? Think before you read on.

The calculations are simple enough that all the programming can be done on the client side, and client-side would lead to a slightly faster experience, so client-side processing is preferred. Be aware that some developers like to use server-side for almost all their web pages. Although that's not recommended, you should be aware that that's sometimes the case. If someone knows a server-side tool really well (e.g., ASP.NET or PHP, which are beyond the scope of this book), they might be inclined to use it for everything. After all, if your only tool is a hammer, everything looks like a nail.

Let's look at a second example web page that uses a form. **FIGURE 8.5** shows a web page that manages the phone numbers for employees at a company. Once again, should processing take



Employee Phone Manager

Hayes, Tre
Wang, Tianyi

Hayes, Tre
Wang, Tianyi

First: Tre
Last: Hayes
Phone: 111-4444

FIGURE 8.5 Web page that manages the phone numbers for a company's employees

place on the client side or the server side? With a large company, there would be a large number of employees, a large amount of data, and a database would be appropriate, so server-side processing is the way to go. With a small company, there wouldn't be a large amount of data, but, regardless, you need to save the data permanently on the server side, so the updated employee phone data can be viewed later by other users. So with a small company, server-side processing is still the way to go.

8.11 `form` Element

Let's now discuss the `form` element, which is in charge of grouping a form's controls. Here's a template for the `form` element's syntax:

```
<form>
  label
  text-box, list-box, check-box, etc.
  label
  text-box, list-box, check-box, etc.
  ...
  submit-button
</form>
```

Note how there's a submit button control at the bottom and other controls above it. That's probably the most common layout because it encourages the user to first provide input for the controls at the top before clicking the button at the bottom. However, you should not try to pigeonhole every one of your web page forms into the template. If it's more appropriate to have your submit button at the top or in the middle, then you should put your submit button at the top or in the middle. One other thing to note in the template is the labels. The labels are text prompts that tell the user what to enter in the subsequent controls.

The following code implements a form with two text controls and a submit button:

```
<form>
  First Name:
  <input type="text" id="first" size="15"><br>
  Last Name:
  <input type="text" id="last" size="15"><br><br>
  <input type="button" value="Generate Email"
    onclick="generateEmail(this.form);">
</form>
```

Notice how this code matches the template provided earlier. The first two controls are text controls that hold first name and last name user entries. We'll cover text control syntax details shortly, but not quite yet. The bottom control is a button. When the button is clicked, its `onclick` event handler calls the `generateEmail` function that combines the entered first and last names to form an email address. We'll explain the event handler's `this.form` argument later, when we present the web page that this form is part of. But first, let's finish talking about forms.

Although it’s legal to use `input` elements—like text controls and buttons—without surrounding them with a `form` element, you’ll usually want to use a form. Here are some reasons for doing so:

- Forms can lead to faster JavaScript processing of the input elements. Understanding why that’s the case will make sense after we explain the JavaScript code in an upcoming web page later in this chapter.
- Forms provide support for being able to check user input to make sure it follows a particular format. That’s called *input validation*, and we’ll spend a considerable amount of time on it in the next chapter.
- Forms provide the ability to add a reset button to assign all the form controls to their original values. To implement a reset button, specify `reset` for the `type` attribute, like this:

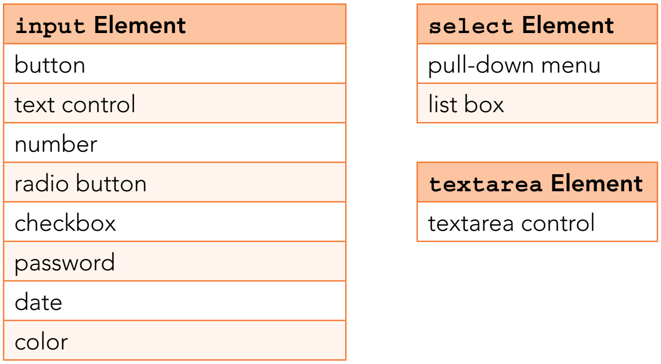
```
<input type="reset" value="Reset">
```

8.12 Controls

There’s lots more syntax to cover when it comes to HTML controls, but before returning to the syntax jungle, a controls overview might be helpful. It’s rather difficult to keep track of which controls use which elements, and this section attempts to make the learning process easier. Read it now and use it as a reference later.

FIGURE 8.6 shows some of the more popular controls and the elements used to implement them. As you can see, most of the controls use the `input` element for their implementation. But just to make things difficult,[6] not all controls use the `input` element. Some important controls use the `select` and `textarea` elements.

In the figure, note the controls in the first table that use the `input` element. You’ve already learned about the button control. You’ve been introduced to the text control, and you’ll learn its syntax details in the next section. You’ll learn about the number control in Chapter 9. For now,



`input` Element
button
text control
number
radio button
checkbox
password
date
color

`select` Element
pull-down menu
list box

`textarea` Element
textarea control

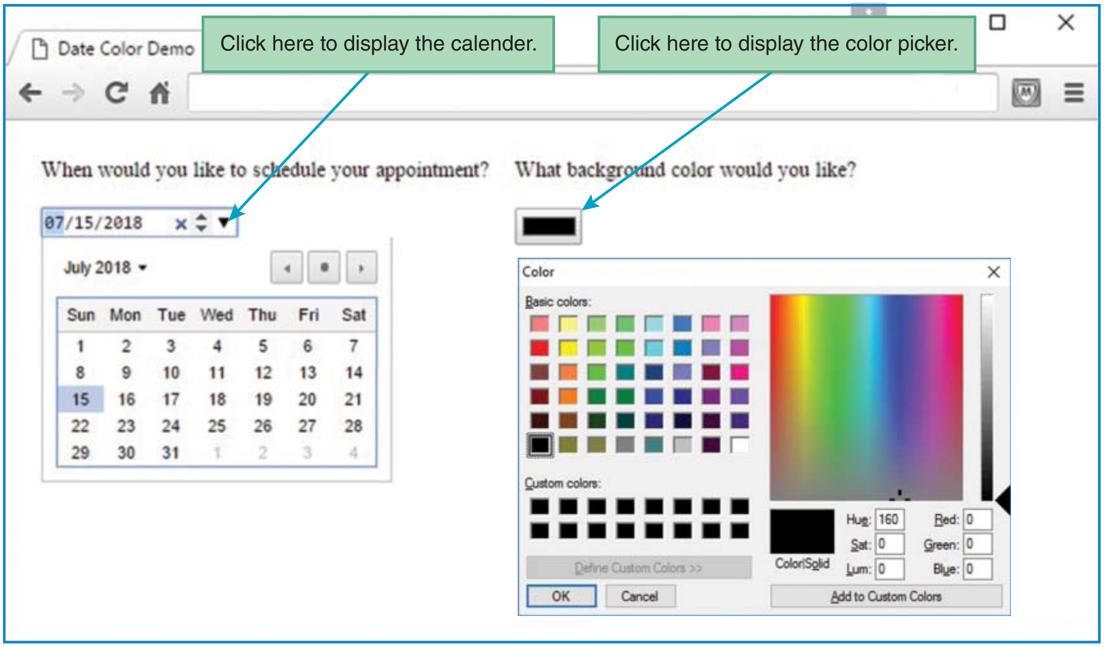
FIGURE 8.6 Some of the more popular controls and the elements used to implement them

[6] Although difficulty is generally not fun, it’s not *always* bad. As difficulty goes up, web programmer wages go up.

just know that the number control provides a mechanism for users to enter a number for input, and it has built-in checking to make sure the input is a properly formatted number. You'll learn about the radio button and checkbox controls in Chapter 10. You've probably seen those controls many times on the Web, so we'll forgo a preliminary explanation at this point.

At this point, we're not providing code examples for the password, date, and color controls, but you should understand what they do. The password control allows the user to enter text into a box where, to help with privacy, the entered characters are obscured. Typically, that means the characters display as bullets. The date control allows the user to enter a month-day-year value for a date. Most browsers implement the date control with a drop-down calendar where the user picks a date from it. Note **FIGURE 8.7**, which shows a calendar displayed after the user clicks the down arrow on the date control's top-right corner. The color control enables the user to select a color from a *color picker* tool. Figure 8.7 shows a color picker displayed after the user clicks the color control's black button.

Be aware that you might run into older browsers that don't support the date and color controls fully. Instead of displaying date and color pickers, they just display boxes that users can enter text into.[7]


Click here to display the calender.

Click here to display the color picker.

When would you like to schedule your appointment?

07/15/2018

What background color would you like?

FIGURE 8.7 Web page that illustrates the date and color controls

[7] Web Hypertext Application Technology Working Group (WHATWG), "The `input` element," https://html.spec.whatwg.org/#the-input-element. We encourage you to peruse the WHATWG's `input` element page for more details about the password, date, and color controls, and to learn about all the other controls that use the `input` element.

In Figure 8.6, note the two controls that use the `select` element—the pull-down menu and list box controls. You'll learn about those controls in Chapter 10. Both controls allow the user to select an item(s) from a list of items. The pull-down menu control normally displays just one item at a time from the list and displays the rest of the list only after the user clicks the control's down arrow. On the other hand, the list box control displays multiple list items simultaneously without requiring the user to click a down arrow.

And finally, in Figure 8.6, note the control that uses the `textarea` element—the textarea control. You'll learn about the textarea control in Chapter 10. For now, just know that it allows the user to enter text into a multiline box. So it's the same as the text control except for the height of the box. We cover the text control in all its glory in the next section.

8.13 Text Control

Earlier, we described the text control as a box that a user can enter text into. Now it's time to dig into text control details. Here's a template for the text control's syntax:

```
<input type="text" id="text-box-identifier"
  placeholder="user-entry-description"
  size="box-width" maxlength="maximum-typed-characters">
```

As you can see, and as you might recall from our description of the button control, the `input` element is a void element, so there's just one tag and no end tag.

The preceding text control template does not include all the attributes for a text control—just the more important ones. We'll describe the attributes shown, plus a few others shortly, but let's first look at an example text control code fragment:

```
<input type="text" id="ssn"
  placeholder="#########" size="9" maxlength="9">
```

Note how the example follows the syntax pattern shown earlier. The text control is for storing a Social Security number, so the `id` attribute's `ssn` value is an abbreviation for Social Security number. What's the purpose of the nine `#`'s for the `placeholder` attribute? Social Security numbers have nine digits, so the nine `#`'s implicitly tell the user to enter nine digits with no hyphens.

Attributes

Here are the text control attributes we'll talk about in this subsection:

Text Control Attributes								
`type`	`id`	`placeholder`	`size`	`maxlength`	`value`	`autofocus`	`disabled`	`readonly`

As mentioned earlier (when describing the `input` element for the button control), the `type` attribute specifies the type of control. For a text control, use `type="text"`. The default value for the `type` attribute is `text`, so if you omit the `type` attribute, you'll get a text control. But for self-documentation purposes, we recommend that you always include `type="text"` for your text controls.

The `id` attribute's value serves as an identifier for the text control, so it can be accessed with JavaScript. Previously, in the Hello web page, we used an `h3` element's `id` value and called `getElementById` to retrieve the object associated with the `h3` element. In an upcoming example, we'll do the same thing using a text control's `id` value.

The `placeholder` attribute provides a word or a short description that helps the user to know what to enter into the text control. When the page loads, the browser puts the `placeholder`'s value in the text control. As soon as the user enters a character into the text control, the entire `placeholder` value disappears.

The `size` attribute specifies the text control's width, where the width value is an integer that approximates the number of average-size characters that can fit in the box. So `size="5"` means approximately 5 characters could display in the box simultaneously. The default size is 20.

The `maxlength` attribute specifies the maximum number of characters that can be entered in the box. By default, an unlimited number of characters is allowed. Entries that exceed the box's width cause input scrolling to occur.

The next four attributes are popular, but not quite as popular as the prior attributes, and that's why they don't appear in the previous text control example. Like the other attributes, they are not required by the HTML5 standard. Use them if you need them.

The `value` attribute specifies an initial value for the text control. The `value` attribute's value is treated as user input. If the user wants a different input, the user must first delete the `value` attribute's value. If the user does nothing and there's JavaScript code that retrieves the user input, it gets the `value` attribute's value by default.

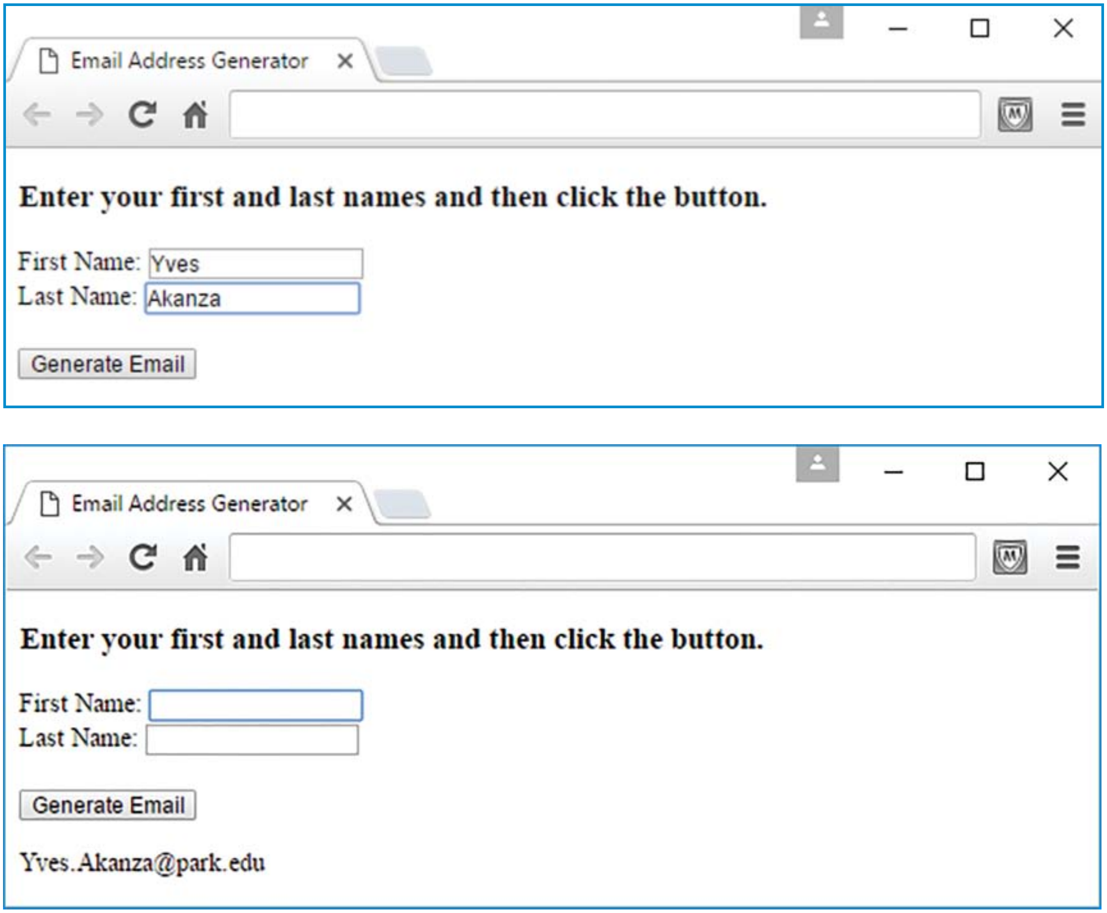
The `autofocus` attribute specifies that after the page has loaded, the browser engine positions the cursor in the text control. To achieve autofocus, specify `autofocus` by itself. As you may recall, when you specify an attribute by itself, that's known as an *empty attribute*.

The `disabled` attribute specifies that the text control cannot receive the focus, and, therefore, the user cannot copy or edit the text control's value. To disable a control, specify `disabled` by itself. The `readonly` attribute specifies that the user can highlight the control's value and copy it, but the user cannot edit it. To make a control read-only, specify `readonly` by itself. For disabled and read-only text controls, the only way to change their values is to use JavaScript assignment statements. You'll see an example of that later in this chapter.

8.14 Email Address Generator Web Page

In this section, we examine a web page that uses text controls for a person's first and last names. In **FIGURE 8.8**, you can see what happens on that web page when the user clicks the Generate Email button. The underlying JavaScript code retrieves the text controls' user-entered values and displays an email address that's built with those values.

In dissecting the Email Address Generator web page's implementation, let's start with the `body` container, which you can see in **FIGURE 8.9A**. Note the `form` container and the `h3`



Enter your first and last names and then click the button.

First Name: Yves
Last Name: Akanza

Enter your first and last names and then click the button.

First Name:
Last Name:

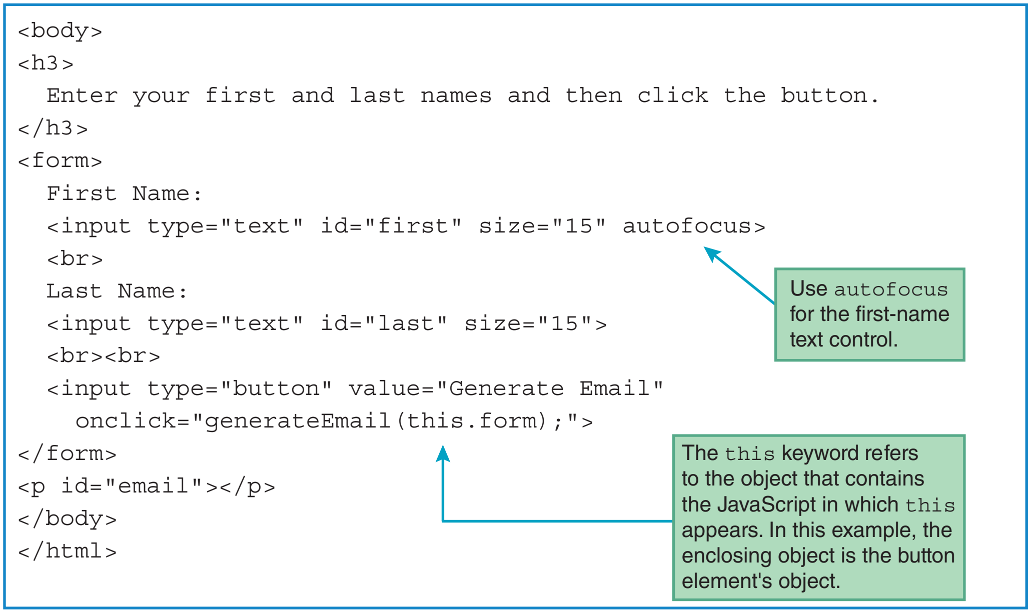
Yves.Akanza@park.edu

FIGURE 8.8 Email Address Generator web page—what happens after the user enters first and last names and what happens after the user clicks the Generate Email button

element above the form. It would be legal to move the `h3` element inside the form, but we recommend not doing so. It's good to keep the form clean, with just control elements and their labels inside it. As you'll see later, having less content within a form can lead to faster retrieval of user input.

Note the two text controls with `size="15"`. So are the user entries limited to 15 characters each? No. The boxes are 15 characters wide, but the user can enter as many characters as desired. Note the first-name text control's `autofocus` attribute. That causes the browser to load the web page with the cursor in that text control.

Note the `p` element below the form. It's a placeholder for the generated email address. When the user clicks the button, the JavaScript engine calls the `generateEmail` function, which assigns the generated email address to the empty area between the `p` element's start and end tags.



```
<body>
<h3>
  Enter your first and last names and then click the button.
</h3>
<form>
  First Name:
  <input type="text" id="first" size="15" autofocus>
  <br>
  Last Name:
  <input type="text" id="last" size="15">
  <br><br>
  <input type="button" value="Generate Email"
    onclick="generateEmail(this.form);">
</form>
<p id="email"></p>
</body>
</html>
```

Use `autofocus` for the first-name text control.

The `this` keyword refers to the object that contains the JavaScript in which `this` appears. In this example, the enclosing object is the button element's object.

FIGURE 8.9A `body` container for Email Address Generator web page

Note the `this.form` argument in the button event handler's `generateEmail` function call. The `this.form` argument requires some in-depth explanation. In the `generateEmail` function (which we'll examine later), we'll need to retrieve the user inputs from the form. To make that possible, when the user clicks the button, we need to pass the form to the `generateEmail` function. So why `this.form` for the generateEmail function call's argument? In general, the `this` keyword refers to the object that contains the JavaScript in which `this` appears. Specifically in this example, the enclosing object is the button element's object. The button element's object has a `form` property that holds the form that surrounds the button. Therefore, we can pass the form object to the `generateEmail` function by calling `generateEmail(this.form)`.

8.15 Accessing a Form's Control Values

In the previous section, you learned how the Email Address Generator's button event handler passes its form object to the `generateEmail` function by using `this.form`. In this section, we'll examine the function itself and learn how to use the form object to access control values within the form.

As you learned earlier, whenever you pass an argument to a function, you should have an associated parameter in the function's heading. Therefore, to receive the form object passed to the `generateEmail` function, there's a `form` parameter in the function's heading, as you can see here:

```
function generateEmail(form)
```

FIGURE 8.9B shows the head container for the Email Address Generator web page. Note the form parameter in the generateEmail function's heading. Be aware that you're not required to use the word "form" for the parameter. We could have used a different parameter name, like "namesForm," but then everywhere you see form within the function body, we'd need to change it to the new parameter name.

Within the generateEmail function body, we use the form parameter to retrieve the text control user inputs. Here's the code for retrieving the user input from the first-name text control:

```
form.elements["first"].value
```

To access the controls that are within a form, we use the form object's elements property. The elements property holds a collection of controls, where a *collection* is a group of items that are of the same type. To access a control within the elements collection, you put quotes around the control's id value and surround the quoted value with []'s. So in the preceding code, you can see first within the []'s, and first is the value for the control's id attribute. Go back to Figure 8.9A and verify that the first-name text control uses id="first". After retrieving the control, there's still one more step (which people forget all the time). To get the user input, you need more than just the control by itself; you need to access the control's value property as shown.

As an alternative to using form.elements["first"], you can use form["first"]. We don't use the form[] syntax in the book's examples because it uses quirky syntax that works with JavaScript, but not with other programming languages. You should get used to standard

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Email Address Generator</title>
<script>
  // This function generates an email address.

  function generateEmail(form) {
    document.getElementById("email").innerHTML =
      form.elements["first"].value + "." +
      form.elements["last"].value + "@park.edu";
    form.reset();
    form.elements["first"].focus();
  } // end generateEmail
</script>
</head>
```

Parameter that holds the form object.

FIGURE 8.9B head **container for Email Address Generator web page**

programming language syntax. The `elements` property is a collection of things, and in JavaScript, to access an element within a collection, you use `[]`'s. On the other hand, `form["first"]` relies on the form object somehow morphing into a collection so `[]`'s are used—very odd indeed! But on the other other hand (assuming you have three hands), if you feel comfortable using the `form[]` syntax, go for it. It uses less code, which leads to slightly faster downloads.

JavaScript Object Properties and HTML Element Attributes

In the `form.elements["first"].value` code fragment shown in the previous section, the `value` property returns the text control's user-entered value. If there's no user entry, then the value of the text control's `value` attribute is returned. If there's no user entry and there's also no `value` attribute, then the value property holds the empty string by default. Having a corresponding JavaScript `value` property for the HTML `value` attribute is indicative of a pattern. There's a parallel world between JavaScript properties and HTML element attributes. In our earlier presentation of the text control element's syntax, we showed these text control element attributes:

```
type, placeholder, size, maxlength, value, autofocus, disabled,
readonly
```

Here are the corresponding JavaScript properties for a text control element object:

```
type, placeholder, size, maxLength, value, autofocus, disabled,
readOnly
```

Note that HTML attributes use all lowercase, whereas JavaScript properties use camel case, which means the two-word properties are spelled `maxLength` and `readOnly`. Get used to that weirdness—use all lowercase for HTML attributes, but camel case for JavaScript properties. JavaScript is case sensitive, so you must use camel case for your code to work. HTML is not case sensitive, but you should use all lowercase in order to exhibit proper style.

Control Elements' `innerHTML` Property

In the Email Address Generator web page's `generateEmail` function, the goal is to update the following `p` element by replacing its empty content with a generated email address:

```
<p id="email"></p>
```

To do that, we retrieve the `p` element's object and then use its `innerHTML` property, like this:

```
document.getElementById("email").innerHTML
```

Remember the `outerHTML` property? It accesses the control element's code, including its start and end tags. The `innerHTML` property accesses the content within the control element's code, not including its start and end tags.

In the generateEmail function, here's the assignment statement that uses innerHTML to update the p element with a generated email address:

```
document.getElementById("email").innerHTML =
  form.elements["first"].value + "." +
  form.elements["last"].value + "@park.edu";
```

string concatenation operator

To connect a string to something else (e.g., another string, a number), you need to use the concatenation operator, +. The resulting connected value forms a string. So in the preceding assignment statement, the three concatenation operations form a single string, and that string gets assigned into the innerHTML part of the retrieved p element.

In the generateEmail function, we use form.elements to retrieve the two text controls. As an alternative, we could have used document.getElementById to retrieve the controls (e.g., document.getElementById["first"]). Why is it better to use form.elements? Because document.getElementById has to search through all the element nodes in the web page's entire node tree, whereas form.elements has to search through only the control nodes in the form part of the web page's node tree. This difference in speed won't be noticeable with web pages that don't use much code (like the Email Address Generator web page), but it's good to use coding practices that *scale* well to web pages with lots of code.

8.16 reset and focus Methods

Go back to Figure 8.9B, and you can see that we still haven't talked about the last two lines in the generateEmail function. Here are those lines:

```
form.reset();
form.elements["first"].focus();
```

The form object's reset method reassigns the form's controls to their original values. Because the Email Address Generator web page has no value attributes for its text controls, the reset method call assigns empty strings to the text controls, thereby blanking them out.

When an element object calls the focus method, the browser puts the focus on the element's control if it's possible to do so. For text control elements, like the first-name text control retrieved in the preceding code, putting the focus on it means the browser positions the cursor in the text control.

8.17 Comments and Coding Conventions

In prior chapters, you learned about various coding conventions for HTML and CSS. Earlier in this chapter, you learned a few coding conventions for JavaScript, such as needing to use descriptive variable names. In this section, you'll get a deeper immersion into JavaScript coding conventions. Remember—it's important to follow coding conventions so your code is understandable, robust, and easy to maintain.

Comments

Let's start with a very important coding convention—use appropriate comments. JavaScript has two types of comments: one type for short comments and one type for longer comments. The syntax for short comments is simply two forward slashes (//) at the left of the descriptive text. Here's an example:

```
// An "admin" user can create and edit accounts.
form.elements["username"].value = "admin";
```

The JavaScript engine ignores JavaScript comments, so why bother to include them? One of the primary purposes of comments is to explain tricky code so programmers can understand the code more easily. Some programmers might find the preceding `focus` method call confusing, and the comment attempts to alleviate some of that confusion.

If you have a comment that spans multiple lines, you can preface each line of the comment with its own //, but that can get cumbersome for long comments. For long comments, you'll normally want to use the other JavaScript comment syntax. Here's the syntax for the other type of comment:

```
/* descriptive-text-goes-here */
```

Typically, this syntax is used for comments that span multiple lines, but it's legal to use it for single-line comments as well. Here's an example comment that spans multiple lines:

```
/* After entering an invalid password 3 times, disable the
   password control so the user cannot try again this session.*/
form.elements["password"].readOnly = true;
```

The /* … */ syntax should look familiar. CSS uses the same syntax for its comments.

In all of these examples, note the blank spaces next to each of the comment characters (after //, after /*, and before */). The spaces are not required by the JavaScript language, but coding conventions suggest that you include them. Why? So the words in your comments stand out and are clear.

Code Skeleton That Illustrates Coding Conventions

There are quite a few coding conventions that we'd like to introduce in rapid-fire succession. To help with the explanations, we'll refer you to the code skeleton shown in **FIGURE 8.10**.

As stated earlier, you should use comments to explain tricky code. In addition, you should include a comment above every function to describe the function's purpose. To make a function's preliminary comment and its subsequent function heading stand out, you should insert a blank line between them. In Figure 8.10, note the two functions and the comments with blank lines above them.

As you read the following coding conventions, for each convention, go to Figure 8.10 and verify that the code skeleton follows that convention:

- If there are two or more functions, separate each adjacent pair of functions with a line of *'s surrounded by blank lines.

- Put all variable declarations at the top of a function's body, and for each variable declaration, provide a comment that describes the variable.
- Provide an "end …" comment for each function's closing brace.
- Position a function's opening brace ({) at the right of the function heading, separated by a space.
- Position a function's closing brace (}) in the same column as the function heading's first character.
- Between a function's opening and closing braces, indent each statement with two spaces.

We'll introduce coding conventions throughout the book's remaining chapters. Appendix B describes all of the JavaScript coding conventions used in this book. Go ahead and skim through it now, and refer back to it later on as questions arise.

Why You Should Use `var` for Variable Declarations

Earlier in the chapter, you were told that before you use a variable, you should use `var` to declare the variable in a declaration statement. Unfortunately, many JavaScript programmers do not use `var`, and you should understand why it's better to use `var`.

Using `var` helps programmers to identify the variables in a function quickly, and that makes the function easier to understand and maintain. If `var` is not used for a variable, then the JavaScript engine creates a *global variable*. A global variable is a variable that's shared between all the functions for a particular web page. Such sharing can be dangerous in that if you coincidentally use same-named variables in different functions, changing the variable's value in one function affects the variable in the other function.

By using `var`, you can use same-named variables in different functions, and the JavaScript engine creates separate local variables. A local variable is a variable that can be used only within the function in which it is declared (with `var`). The *scope* of a variable refers to where the variable

```
// Check whether the entered username is valid.

function validUsername(form) {
  var username; // object for username text control
  ...
} // end validUsername

//**************************************

// Check whether the entered password is valid.

function validPassword(form) {
  var password; // object for username text control
  ...
} // end validPassword
```

FIGURE 8.10 Code skeleton that illustrates coding conventions

can be used, so the scope of a function's local variables is limited to the function's body. If you have same-named local variables in different functions, changing one of the variables won't affect the other variable because each variable is a separate entity. Such separation is normally considered a good thing because that makes it harder for the programmer to accidentally mess things up.

8.18 Event-Handler Attributes

Remember the `onclick` attribute for the button control's input element? That attribute is known as an *event-handler attribute* because its value is an event handler. As you know, an event handler is JavaScript code that tells the JavaScript engine what to do when a particular event takes place. When an event takes place, we say that the event *fires*. For the button control's `onclick` attribute, the event is clicking the button.

Take a look at the table of event-handler attributes and their associated events in **FIGURE 8.11**. We'll provide a brief overview of those event-handler attributes in this section and put them to use in web page examples later on.[8]

The first event-handler attribute shown in Figure 8.11's table is `onclick`, which you should already be familiar with. It's very common to use `onclick` with a button, but the HTML5 standard indicates that you can use it with any element.

The next event-handler attribute is `onfocus`. You can use `onfocus` to do something special when a control gains focus. For example, when the user clicks within a text control, you could implement an `onfocus` event handler to make the text control's text become blue.

The next event-handler attribute is `onchange`. You can use `onchange` to do something special when a control's value changes. For example, when the user clicks a radio button, you could implement an `onchange` event handler that displays an "Are you sure you want to change your selection?" message.

Event-Handler Attributes	Events
`onclick`	User clicks on an element.
`onfocus`	An element gains focus.
`onchange`	The value of a form control has been changed.
`onmouseover`	Mouse moves over an element.
`onmouseout`	Mouse moves off an element.
`onload`	An element finishes loading.

FIGURE 8.11 Some of the more popular event-handler attributes and their associated events

[8] Web Hypertext Application Technology Working Group (WHATWG), "Event handlers on elements, Document objects, and Window objects," https://html.spec.whatwg.org/multipage/webappapis.html#event-handlers-on-elements,-document-objects,-and-window-objects. If you'd like to learn about additional event-handler attributes, peruse the WHATWG's event handler page.

The next event-handler attributes, `onmouseover` and `onmouseout`, are often used to implement rollovers for `img` elements. The mouseover event is triggered when the mouse moves on top of an element. The mouseout event is triggered when the mouse moves off of an element.

The last event-handler attribute shown in Figure 8.11 is `onload`. The load event is triggered when the browser finishes loading an element. It's common to use the `onload` attribute with the `body` element so you can do something special after the entire web page loads.

8.19 onchange, onmouseover, onmouseout

In this section, we provide web page examples that put into practice what you learned earlier about event-handler attributes. Specifically, we'll use the `onchange` event-handler attribute to improve the Email Address Generator web page. Then we'll use `onmouseover` and `onmouseout` to implement a rollover in another web page.

Improving the Email Address Generator Web Page with onchange

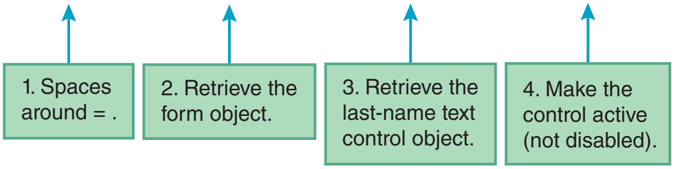
In the Email Address Generator web page, suppose you want to force the user to enter the first name before the last name. To do that, you can disable the last-name text control initially and remove that restriction after the first-name text control has been filled in. To determine whether the first-name text control has been filled in, you can rely on the text control's change event firing. A text control's change event fires after the user clicks or tabs away from the text control after the user has made changes to the text control. By adding an `onchange` event-handler attribute to the text control's `input` element, the text control can "listen" for the first-name text control being changed and then act accordingly.

In implementing the improvements to the Email Address Generator web page, the first step is to disable the last-name text control when the web page first loads. Note the `disabled` attribute:

```
Last Name:
<input type="text" id="last" size="15" disabled>
```

The next step involves adding an `onchange` event handler to the first-name text control's `input` element. Note the `onchange` event handler:

```
First Name:
<input type="text" id="first" size="15" autofocus
  onchange = "this.form.elements['last'].disabled=false;">
```



1. Spaces around = .

2. Retrieve the form object.

3. Retrieve the last-name text control object.

4. Make the control active (not disabled).

Before we explain the onchange event handler's rather complicated details, let's first appreciate its overall nature. In our previous event-handler examples, the event handler has always been a function call, like this:

```
onclick="generateEmail(this.form)";
```

With a function call, the work is done in the function's body. In the onchange event handler shown earlier, the event handler contains code that does the work "inline." Inline JavaScript is appropriate when there is just one statement and there is only one place on the web page where the code is used. An advantage of using inline JavaScript is that it can lead to code that is easier to understand because all the code (the HTML control code and the event handler JavaScript code) is in one place.

Now let's dig into the details of the onchange event handler shown earlier. The following four items refer to four noteworthy details from the onchange event handler. As you read each item, go to the same-numbered callout next to the onchange event handler code fragment and see where the item is located within the code fragment.

1. For normal attribute-value pairs, you should not surround the = with spaces. But for an event-handler attribute, if its value is not short, separate the value from the attribute with spaces around the =. For the onchange event handler, we have inline JavaScript code and the event handler is not short, so spaces around the = are appropriate.
2. If you're inside a form control, to retrieve the form element's object, use this.form. The example code fragment is for an input element, and the input element is indeed inside a form (as you can verify by going back to the web page's source code in Figure 8.9A).
3. To retrieve the last-name text control object, specify elements['last'] with single quotes around 'last' to avoid terminating the prior opening double quote. In the event-handler code fragment, note the double quote that begins the onchange attribute's value. To nest strings inside strings, you can use double quotes for the outer string and single quotes for the inner string (as shown in the example code fragment) or vice versa.
4. To make the retrieved text control active (not disabled), assign false to the text control object's disabled property.

Suppose you've added the disabled attribute to the last-name text control and the onchange event handler to the first-name text control as described earlier. With the new code added, what happens after a user clicks the Generate Email button and wants to enter first and last names for a second email address? Will the user's experience be the same? (Having a consistent experience is a good thing, by the way.) Specifically, will the user again be forced to enter the first name first?

Well… actually no. The onchange event handler activates the last-name text control, and it remains active after that. So, what's the solution? After clicking the button, you need to disable the last-name text control. To do that, you should add this code at the bottom of the `generateEmail` function:

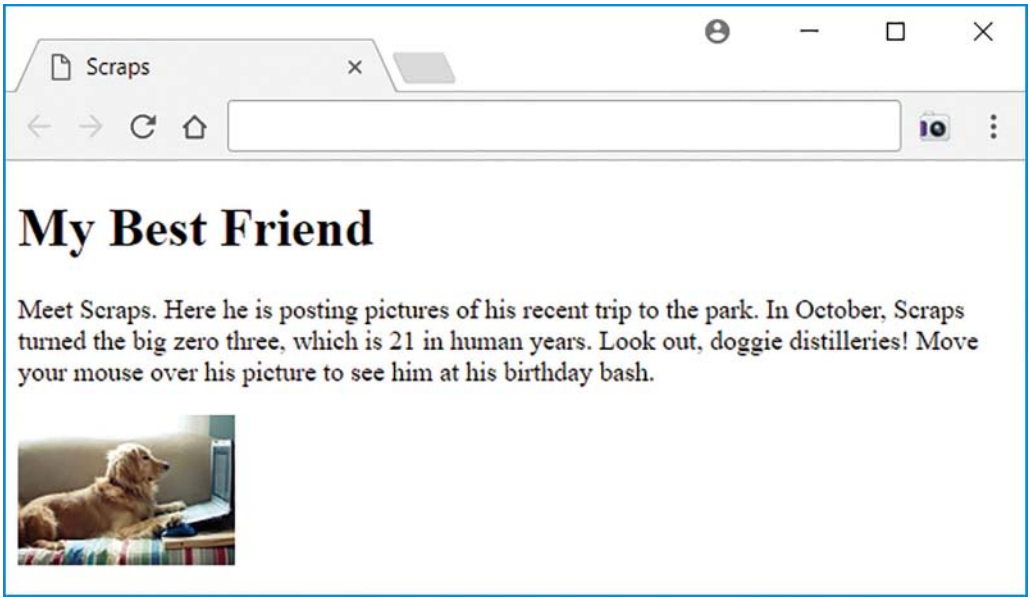
```
    form.elements["last"].disabled = true;
```

Implementing a Rollover with `onmouseover` and `onmouseout`

A rollover is when an image file changes due to the user rolling the mouse over the image. As you learned in the previous chapter, you can implement a rollover with a CSS image sprite. Now, you'll learn how to implement a rollover with `onmouseover` and `onmouseout` event handlers that reassign values to the image object.

Take a look at the Scraps the Dog web page in **FIGURE 8.12**. If the user moves the mouse over the image, the browser swaps out the original picture and displays a picture of Scraps at his third birthday party. If the mouse moves off of the image, the browser swaps out the birthday picture and displays the original picture.

FIGURE 8.13 shows the source code for the Scraps web page. Let's focus on the event-handler code. The `onmouseover` and `onmouseout` event handlers both rely on the `this` keyword. Read the figure's left callout and make sure you understand why `this` refers to the `img` element. With that in mind, `this.src` refers to the `img` element's `src` attribute, which is in charge of specifying the `img` element's image file. So it's the event handlers' assignment of files to the `src` attribute that implements the rollover functionality.

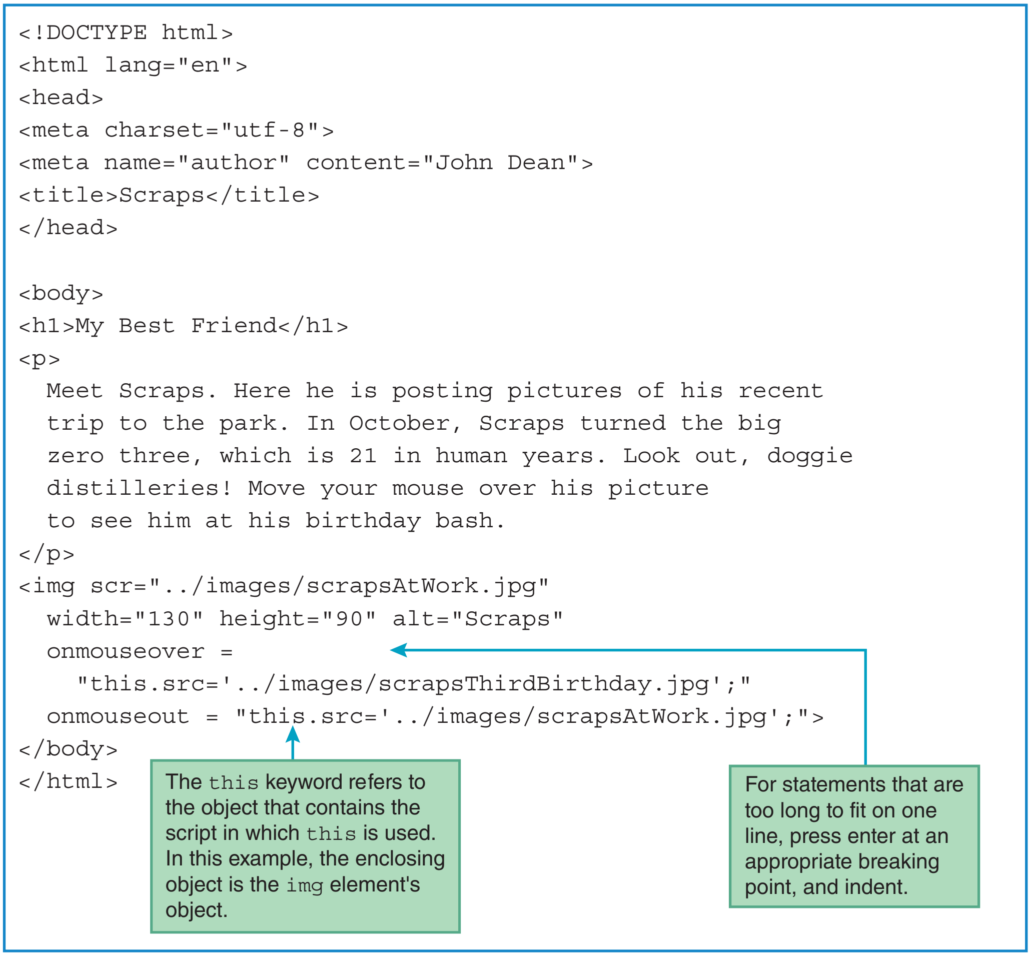


My Best Friend

Meet Scraps. Here he is posting pictures of his recent trip to the park. In October, Scraps turned the big zero three, which is 21 in human years. Look out, doggie distilleries! Move your mouse over his picture to see him at his birthday bash.

© Elizabeth Aldridge/Getty Images

FIGURE 8.12 **Scraps the Dog web page**



```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Scraps</title>
</head>

<body>
<h1>My Best Friend</h1>
<p>
  Meet Scraps. Here he is posting pictures of his recent
  trip to the park. In October, Scraps turned the big
  zero three, which is 21 in human years. Look out, doggie
  distilleries! Move your mouse over his picture
  to see him at his birthday bash.
</p>
<img scr="../images/scrapsAtWork.jpg"
  width="130" height="90" alt="Scraps"
  onmouseover =
    "this.src='../images/scrapsThirdBirthday.jpg';"
  onmouseout = "this.src='../images/scrapsAtWork.jpg';">
</body>
</html>
```

The `this` keyword refers to the object that contains the script in which `this` is used. In this example, the enclosing object is the `img` element's object.

For statements that are too long to fit on one line, press enter at an appropriate breaking point, and indent.

FIGURE 8.13 Source code for Scraps the Dog web page

Read the right callout in Figure 8.13 and note the line break in the source code after `onmouseover =`. The line break is necessary because the event-handler code is long enough to run the risk of bumping against the edge of a printer's right margin. If that happens, then line wrap occurs. Several chapters ago, we introduced the concept of line wrap for HTML code, and the concept is the same with JavaScript code. For statements that might be too long to fit onto one line, press enter at an appropriate breaking point, and on the next line, indent past the starting point of the prior line.

8.20 Using `noscript` to Accommodate Disabled JavaScript

So far, you might have assumed that all users will be able to take advantage of the cool JavaScript that you've learned. That assumption is valid for the vast majority of users, but with 3.7 billion users in the world and counting,[9] you'll probably run into a lack of JavaScript support every now and then.

Older browsers don't support JavaScript, but the bigger roadblock is that some users intentionally disable JavaScript on their browsers. Typically, they do that because they're concerned that executing JavaScript code can be a security risk. However, most security experts agree that JavaScript is relatively safe. After all, it was/is designed to have limited capabilities. For example, JavaScript is unable to access a user's computer in terms of the computer's files and what's in the computer's memory. Also, JavaScript can send requests to web servers only in a constrained (and safe) manner.

Despite JavaScript's built-in security measures, some users will continue to disable JavaScript on their browsers. For your web pages that use JavaScript, it's good practice to display a warning message on browsers that have JavaScript disabled. To display such a message on only those browsers and not on browsers that have JavaScript enabled, use the `noscript` element. Specifically, add a `noscript` container to the top of your `body` container, and insert explanatory text inside the `noscript` container. Here's an example:

```
<noscript>
  <p>
    This web page uses JavaScript. For proper results,
    you must use a web browser with JavaScript enabled.
  </p>
</noscript>
```

[9] InternetLiveStats.com, "*Internet Users*," http://www.internetlivestats.com/internet-users