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:

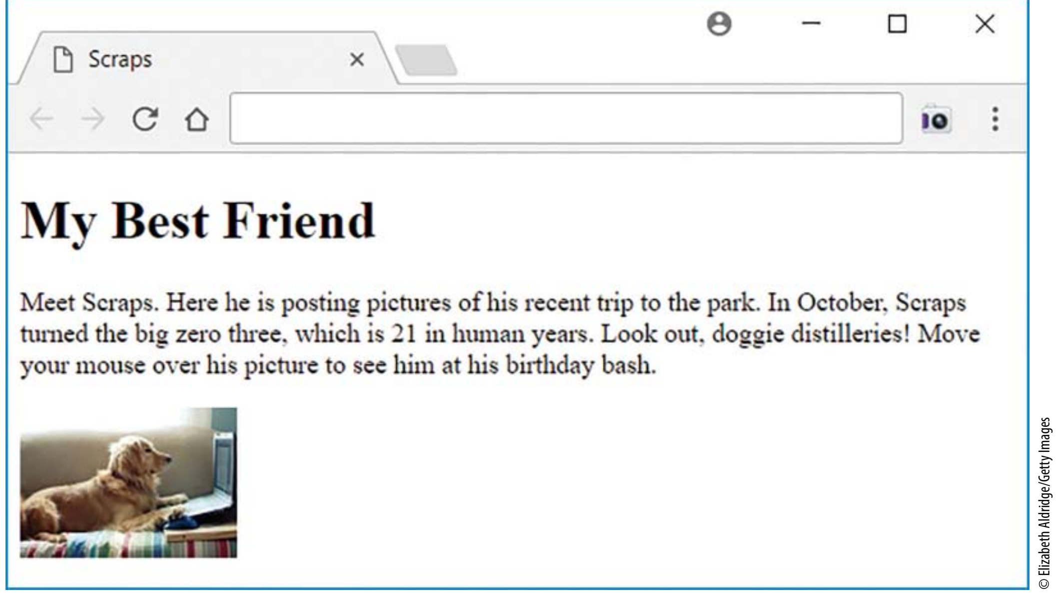
```
form.elements["last"].disabled = true;
```

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.



© 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