Excerpt from "Sams Teach Yourself AngularJS, JavaScript, and jQuery All in One" by Brad Dayley and Brendan Dayley (Sams, 2015)

LESSON 5

Jumping into jQuery and JavaScript Syntax

What You'll Learn in This Lesson:

- Ways to add jQuery and JavaScript to your web pages
- Creating and manipulating arrays of objects
- Adding code logic to JavaScript
- Implementing JavaScript functions for cleaner code

Throughout the book, you'll see several examples of using jQuery and JavaScript to perform various dynamic tasks. jQuery doesn't replace JavaScript; it enhances it by providing an abstract layer to perform certain common tasks, such as finding elements or values, changing attributes and properties of elements, and interacting with browser events.

AngularJS uses JavaScript and jQuery syntax to provide its functionality. It is important for you to understand the jQuery and JavaScript syntax before getting into AngularJS. That is why these are covered first and AngularJS is covered in later lessons.

In this lesson, you learn the basic structure and syntax of JavaScript and how to use jQuery to ease some of the development tasks. The purpose of this lesson is to help you become familiar with the JavaScript language syntax, which is also the jQuery language syntax.

Adding jQuery and JavaScript to a Web Page

Browsers come with JavaScript support already built in to them. That means all you need to do is add your own JavaScript code to the web page to implement dynamic web pages. jQuery, on the other hand, is an additional library, and you will need to add the jQuery library to your web page before adding jQuery scripts.

Loading the jQuery Library

Because the jQuery library is a JavaScript script, you use the `<script>` tag to load the jQuery into your web page. jQuery can either be downloaded to your code directory and then hosted on

your web server, or you can use the hosted versions that are available at jQuery.com. The following statement shows an example of each; the only difference is that the first loads it from the jQuery CDN source and the second loads it from the web server:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="includes/js/jquery-latest.min.js"></script>
```

CAUTION

Remember that you need to place the `<script>` element to load the jQuery library before any script elements that are using it. Otherwise, those libraries will not be able to link up to the jQuery code.

The jQuery library downloads can be found at the following location:

```
http://jquery.com/download/
```

The jQuery library hosted links can be found at the following location:

```
http://code.jquery.com/
```

Implementing Your Own jQuery and JavaScript

JQuery code is implemented as part of JavaScript scripts. To add jQuery and JavaScript to your web pages, first add a `<script>` tag that loads the jQuery library, and then add your own `<script>` tags with your custom code.

The JavaScript code can be added inside the `<script>` element, or the `src` attribute of the `<script>` element can point to the location of a separate JavaScript document. Either way, the JavaScript will be loaded in the same manner.

The following is an example of a pair of `<script>` statements that load jQuery and then use it. The `document.write()` function just writes text directly to the browser to be rendered:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script>
  function writeIt(){
    document.write("jQuery Version " + $().jquery + " loaded.");
  }
</script>
```

NOTE

The `<script>` tags do not need to be added to the `<head>` section of the HTML document; they can also be added in the body. It's useful to add simple scripts directly inline with the HTML elements that are consuming them.

Accessing HTML Event Handlers

So after you add your JavaScript to the web page, how do you get it to execute? The answer is that you tie it to the browser events. Each time a page or element is loaded, the user moves or clicks the mouse or types a character, an HTML event is triggered.

Each supported event is an attribute of the object that is receiving the event. If you set the attribute value to a JavaScript function, the browser will execute your function when the event is triggered.

For example, the following will execute the `writeIt()` function when the body of the HTML page is loaded:

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

TRY IT YOURSELF ▼

Implementing JavaScript and jQuery

Those are the basic steps. Now it is time to try it yourself. Use the following steps to add jQuery to your project and use it dynamically in a web page:

1. In Eclipse, create a source folder named lesson05.
2. In the same folder as the lesson05 folder, add an additional directory called js.
3. Now create a source file named jquery_version.html in the lesson05 folder.
4. Add the usual basic elements (html, head, body).
5. Inside the `<head>` element, add the following line to load the library you just downloaded:

```
06      <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
```

6. Now you can add your own `<script>` tag with the following code to print out the jQuery version to the browser windows:

```
07      <script>
08        function writeIt(){
09          document.write("jQuery Version " + $().jquery + " loaded.");
10        }
11      </script>
```

7. To have your script execute when the document is loaded, tie the `writeIt()` function to the `<body> onload` event using the following line:

```
13    <body onload="writeIt()">
```

8. Save the file and view it in your web browser at the following location. The output should be similar to Figure 5.1:

   `http://localhost/lesson06/jquery_version.html`

FIGURE 5.1
The function `writeIt()` is executed when the body loads and writes the jQuery version to the browser.

LISTING 5.1 jquery_version.html Very Basic Example of Loading Using jQuery in a Web Page to Print Out Its Own Version

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>jQuery Version</title>
05     <meta charset="utf-8" />
06     <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
07     <script>
08       function writeIt(){
09         document.write("jQuery Version " + $().jquery + " loaded.");
10       }
11     </script>
12   </head>
13   <body onload="writeIt()">
14   </body>
15 </html>
```

Accessing the DOM

One of the most important aspects of JavaScript, and especially jQuery, is the capability to access and manipulate the DOM. Accessing the DOM is how you make the web page dynamic by changing styles, size, position, and values of elements.

In the following sections, you learn about accessing the DOM through traditional methods via JavaScript and the much improved methods using jQuery selectors. These sections are a brief introduction. You will get plenty of practice as the lessons roll on.

Using Traditional JavaScript to Access the DOM

Traditionally, JavaScript uses the global `document` object to access elements in the web page. The simplest method of accessing an element is to directly refer to it by `id`. For example, if you have a paragraph with the `id="question"`, you can access it via the following JavaScript `getElementById()` function:

```
var q = document.getElementById("question");
...
<p id="question">Which method to you prefer?</p>
```

Another helpful JavaScript function that you can use to access the DOM elements is `getElementsByTagName()`. This returns a JavaScript array of DOM elements that match the tag name. For example, to get a list of all the `<p>` elements, use the following function call:

```
var paragraphs = document.getElementsByTagName("p");
```

Using jQuery Selectors to Access HTML Elements

Accessing HTML elements is one of jQuery's biggest strengths. jQuery uses selectors that are very similar to CSS selectors to access one or more elements in the DOM; hence the name jQuery. jQuery returns back either a single element or an array of jQuerified objects. jQuerified means that additional jQuery functionality has been added to the DOM object, allowing for much easier manipulation.

The syntax for using jQuery selectors is `$(`*`selector`*`).`*`action`*`()`, where *`selector`* is replaced by a valid selector and *`action`* is replaced by a jQuerified action attached to the DOM element(s).

For example, the following command finds all paragraph elements in the HTML document and sets the CSS `font-weight` property to `bold`:

```
$("p").css('font-weight', 'bold');
```

TRY IT YOURSELF ▼

Using jQuery and JavaScript to Access DOM Elements

Now to solidify the concepts, you'll run through a quick example of accessing and modifying DOM elements using both jQuery and JavaScript. Use the following steps to build the HTML document shown in Listing 5.2:

1. Create a source file named dom_elements.html in the lesson05 folder.
2. Add the usual basic elements (html, head, body).
3. Inside the `<head>` element, add the following line to load the library you just downloaded:

```
06      <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
```

4. Add the following `<script>` element that accesses the DOM using both the JavaScript and jQuery methods. Notice that with jQuery, two actions are chained together. The first sets the CSS `font-weight` property and the second changes text contained in element. With JavaScript, you use the `getElementById()` method, and then you set the `innerHTML` property directly in the DOM to change the text displayed in the browser:

```
07      <script>
08        function writeIt(){
09          $("#heading").css('font-weight', 'bold').html("jQuery");
10          var q = document.getElementById("question");
11          q.innerHTML = "I Prefer jQuery!";
12        }
13      </script>
```

5. To have your script execute when the document is loaded, tie the `writeIt()` function to the `<body>` `onload` event using the following line:

```
15    <body onload="writeIt()">
```

6. Add the following `<p>` elements to the `<body>` to provide containers for the JavaScript code to access:

```
16      <p id="heading">jQuery or JavaScript</p>
17      <p id="question">Which method to you prefer?</p>
```

7. Save the file and view it in a web browser. The output should be similar to Figure 5.2.

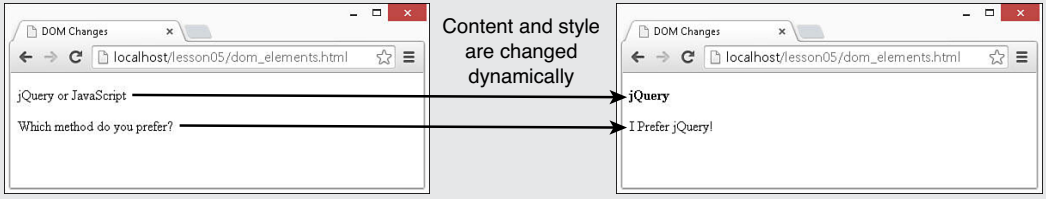

FIGURE 5.2
The function `writeIt()` is executed when the body loads and changes the content and appearance of the text.

LISTING 5.2 Very Basic Example of Using JavaScript and jQuery to Access DOM Elements

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>DOM Changes</title>
05     <meta charset="utf-8" />
06     <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
07     <script>
08       function writeIt(){
09         $("#heading").css('font-weight', 'bold').html("jQuery");
10         var q = document.getElementById("question");
11         q.innerHTML = "I Prefer jQuery!";
12       }
13     </script>
14   </head>
15   <body onload="writeIt()">
16     <p id="heading">jQuery or JavaScript</p>
17     <p id="question">Which method do you prefer?</p>
18   </body>
19 </html>
```

Understanding JavaScript Syntax

Like any other computer language, JavaScript is based on a rigid syntax where specific words mean different things to the browser as it interprets the script. This section is designed to walk you through the basics of creating variables, working with data types, and using looping and functions in JavaScript to manipulate your web pages.

TIP

For the simple JavaScript examples in this lesson, you can test them by starting Node.js using the `node` command from a console prompt to bring up the Node.js interpreter. From the interpreter, you can type in JavaScript code and have it execute as you type each line.

Creating Variables

The first place to begin with in JavaScript is variables. Variables are a means to name data so that you can use that name to temporarily store and access data from your JavaScript files.

Variables can point to simple data types, such as numbers or strings, or they can point to more complex data types, such as objects.

To define a variable in JavaScript, you must use the `var` keyword and then give the variable a name; for example:

```
var myData;
```

You can also assign a value to the variable in the same line. For example, the following line of code creates a variable `myString` and assigns it the value of "Some Text":

```
var myString = "Some Text";
```

This works as well as the following:

```
var myString;
myString = "Some Text";
```

After you have declared the variable, you can use the name to assign the variable a value and access the value of the variable. For example, the following code stores a string into the `myString` variable and then uses it when assigning the value to the `newString` variable:

```
var myString = "Some Text";
var newString = myString + " Some More Text";
```

Your variable names should describe the data that is stored in them so that it is easy to use them later in your program. The only rule for creating variable names is that they must begin with a letter, `$`, or `_`, and they cannot contain spaces. Also remember that variable names are case sensitive, so using `myString` is different from `MyString`.

Understanding JavaScript Data Types

JavaScript uses data types to determine how to handle data that is assigned to a variable. The variable type will determine what operations you can perform on the variable, such as looping or executing. The following list describes the most common types of variables that we will be working with through the book:

- **`String`**—Stores character data as a string. The character data is specified by either single or double quotes. All the data contained in the quotes will be assigned to the string variable. For example:

  ```
  var myString = 'Some Text';
  var anotherString = "Some Other Text";
  ```

- **`Number`**—Stores the data as a numerical value. Numbers are useful in counting, calculations, and comparisons. Some examples are as follows:

```
var myInteger = 1;
var cost = 1.33;
```

- **`Boolean`**—Stores a single bit that is either true or false. Booleans are often used for flags. For example, you might set a variable to false at the beginning of some code and then check it on completion to see whether the code execution hit a certain spot. The following shows an example of defining a true and a false variable:

```
var yes = true;
var no = false;
```

- **`Array`**—An indexed array is a series of separate distinct data items all stored under a single variable name. Items in the array can be accessed by their zero-based index using the `[index]`. The following is an example of creating a simple array and then accessing the first element, which is at index 0:

```
var arr = ["one", "two", "three"]
var first = arr[0];
```

- **Associative Array/Objects**—JavaScript does support the concept of an associative array, meaning accessing the items in the array by a name instead of an index value. However, a better method is to use an object literal. When you use an object literal, you can access items in the object using `object.property` syntax. The following example shows how to create and access an object literal:

```
var obj = {"name":"Brad", "occupation":"Hacker", "age", "Unknown"};
var name = obj.name;
```

- **`Null`**—At times, you do not have a value to store in a variable, either because it hasn't been created or you are no longer using it. At this time, you can set a variable to `null`. That way, you can check the value of the variable in your code and use it only if it is not `null`:

```
var newVar = null;
```

NOTE

JavaScript is a typeless language, meaning you do not need to tell the browser what data type the variable is; the interpreter will automatically figure out the correct data type for the variable.

Using Operators

JavaScript operators provide the capability to alter the value of a variable. You are already familiar with the = operator because you used it several times in the book already. JavaScript provides several operators that can be grouped into two types—arithmetic and assignment.

Arithmetic Operators

Arithmetic operators are used to perform operations between variable and direct values. Table 5.1 shows a list of the arithmetic operations along with the results that get applied.

TABLE 5.1 Table Showing JavaScripts' Arithmetic Operators as Well as Results Based on y=4 to Begin With

Operator	Description	Example	Resulting x	Resulting y
+	Addition	x=y+5	9"45"	4
		x=y+"5"	"Four44"	4
		x="Four"+y+"4"		4
-	Subtraction	x=y-2	2	4
++	Increment	x=y++	4	5
		x=++y	5	5
--	Decrement	x=y--	4	3
		x=--y	3	3
*	Multiplication	x=y*4	16	4
/	Division	x=10/y	2.5	4
%	Modulous (remainder of Division)	x=y%3	1	4

TIP

The + operator can also be used to add strings or strings and numbers together. This allows you to quickly concatenate strings and add numerical data to output strings. Table 5.1 shows that when adding a numerical value and a string value, the numerical value is converted to a string, and then the two strings are concatenated.

Assignment Operators

Assignment operators are used to assign a value to a variable. You are probably used to the = operator, but there are several forms that allow you to manipulate the data as you assign the value. Table 5.2 shows a list of the assignment operations along with the results that get applied.

TABLE 5.2 JavaScripts' Assignment Operators as Well as Results Based on x=10 to Begin With

Operator	Example	Equivalent Arithmetic Operators	Resulting x
=	x=5	x=5	5
+=	x+=5	x=x+5	15
-=	x-=5	x=x-5	5
*=	x*=5	x=x*5	50
/=	x/=5	x=x/5	2
%=	x%=5	x=x%5	0

Applying Comparison and Conditional Operators

Conditionals are a way to apply logic to your applications so that certain code will be executed only under the correct conditions. This is done by applying comparison logic to variable values. The following sections describe the comparisons available in JavaScript and how to apply them in conditional statements.

Comparison Operators

A comparison operator evaluates two pieces of data and returns true if the evaluation is correct or false if the evaluation is not correct. Comparison operators compare the value on the left of the operator against the value on the right.

The simplest way to help you understand comparisons is to provide a list with some examples. Table 5.3 shows a list of the comparison operators along with some examples.

TABLE 5.3 JavaScripts' Comparison Operators as Well as Results Based on x=10 to Begin With

Operator	Example	Example	Result
==	Is equal to (value only)	x==8	false
		x==10	true
===	Both value and type are equal	x===10	true
		x==="10"	false
!=	Is not equal	x!=5	true
!==	Both value and type are not equal	x!=="10"	true
		x!==10	false
>	Is greater than	x>5	true

Operator	Example	Example	Result
>=	Is greater than or equal to	x>=10	true
<	Is less than	x<5	false
<=	Is less than or equal to	x<=10	true

You can chain multiple comparisons together using logical operators. Table 5.4 shows a list of the logical operators and how to use them to chain comparisons together.

TABLE 5.4 JavaScripts' Comparison Operators as Well as Results Based on x=10 and y=5 to Begin With

Operator	Description	Example	Result
&&	and	(x==10 && y==5)(x==10 && y>x)	true false
||	or	(x>=10 || y>x)(x<10 && y>x)	true false
!	not	!(x==y)!(x>y)	true false
	mix	(x>=10 && y<x || x==y)((x<y || x>=10) && y>=5)(!(x==y) && y>=10)	true true false

If

An `if` statement enables you to separate code execution based on the evaluation of a comparison. The syntax is shown in the following lines of code where the conditional operators are in `()` parenthesis and the code to execute if the conditional evaluates to true is in {} brackets:

```
if(x==5){
  do_something();
}
```

In addition to executing code only within the `if` statement block, you can specify an else block that will get executed only if the condition is false. For example:

```
if(x==5){
  do_something();
} else {
  do_something_else();
}
```

You can also chain `if` statements together. To do this, add a conditional statement along with an `else` statement. For example:

```
if(x<5){
  do_something();
} else if(x<10) {
  do_something_else();
} else {
  do_nothing();
}
```

switch

Another type of conditional logic is the `switch` statement. The `switch` statement allows you to evaluate an expression once and then, based on the value, execute one of many sections of code.

The syntax for the switch statement is the following:

```
switch(expression){
  case value:
    <code to execute>
    break;
  case value2:
    <code to execute>
    break;
  default:
    <code to execute if not value or value2>
}
```

This is what is happening. The `switch` statement will evaluate the expression entirely and get a value. The value may be a string, a number, a Boolean, or even an object. The `switch` value is then compared to each value specified by the `case` statement. If the value matches, the code in the `case` statement is executed. If no values match, the `default` code is executed.

NOTE

Typically, each `case` statement will include a break command at the end to signal a break out of the `switch` statement. If no break is found, code execution will continue with the next `case` statement.

▼ TRY IT YOURSELF

Applying If Conditional Logic in JavaScript

To help you solidify using JavaScript conditional logic, use the following steps to build conditional logic into the JavaScript for a dynamic web page. The final version of the HTML document is shown in Listing 5.3:

1. Create a source file named if_logic.html in the lesson05 folder.
2. Create a folder under lesson05 named images.
3. Add your own images for day.png and night.png to the ./images folder in your project or download the ones from the book's website.
4. Add the usual basic elements (html, head, body).
5. Add the following `<script>` element that gets the lesson value using the `Date().getLessons()` JavaScript code. The code uses `if` statements to determine the time of day and does two things: it writes a greeting onto the screen and sets the value of the `timeOfDay` variable:

```
06     <script>
07       function writeIt(){
08         var lesson = new Date().getLessons();
09         var timeOfDay;
10         if(lesson>=7 && lesson<12){
11           document.write("Good Morning!");
12           timeOfDay="morning";
13         } else if(lesson>=12 && lesson<18) {
14           document.write("Good Day!");
15           timeOfDay="day";
16         } else {
17           document.write("Good Night!");
18           timeOfDay="night";
19         }
32       }
33     </script>
```

6. Now add the following `switch` statement that uses the value of `timeOfDay` to determine which image to display in the web page:

```
20         switch(timeOfDay){
21           case "morning":
22           case "day":
23             document.write("<img src='images/day.png' />");
24             break;
25           case "night":
26             document.write("<img src='images/night.png' />");
27             break;
```

TRY IT YOURSELF

```
28            default:
29              document.write("<img src='images/day.png' />");
30          }
```

7. Save the file and view it in a web browser. The output should be similar to Figure 5.3, depending on what time of day it is.

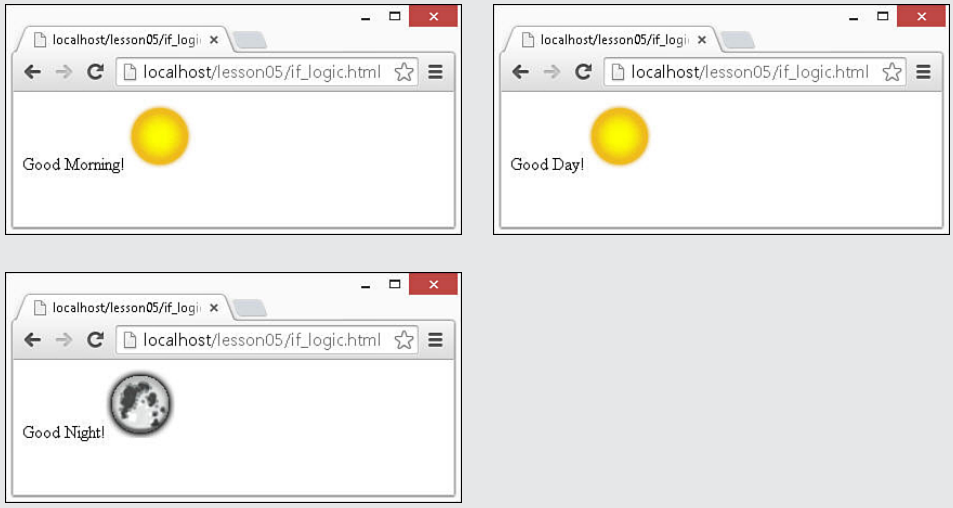

FIGURE 5.3
The function `writeIt()` is executed when the body loads and changes the greeting and image displayed on the web page.

LISTING 5.3 if_logic.html Simple Example of Using Conditional Logic Inside JavaScript

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>If Logic</title>
05     <meta charset="utf-8" />
06     <script>
07       function writeIt(){
08         var hour = new Date().getHours();
09         var timeOfDay;
10         if(hour>=7 && hour<12){
11           document.write("Good Morning!");
12           timeOfDay="morning";
```

```
13          } else if(hour>=12 && hour<18) {
14            document.write("Good Day!");
15            timeOfDay="day";
16          } else {
17            document.write("Good Night!");
18            timeOfDay="night";
19          }
20          switch(timeOfDay){
21            case "morning":
22            case "day":
23              document.write("<img src='../images/day.png' />");
24              break;
25            case "night":
26              document.write("<img src='../images/night.png' />");
27              break;
28            default:
29              document.write("<img src='../images/day.png' />");
30          }
31        }
32      </script>
33    </head>
34    <body onload="writeIt()">
35    </body>
36  </html>
```

Implementing Looping

Looping is a means to execute the same segment of code multiple times. This is extremely useful when you need to perform the same tasks on a set of DOM objects, or if you are dynamically creating a list of items.

JavaScript provides functionality to perform `for` and `while` loops. The following sections describe how to implement loops in your JavaScript.

`while` Loops

The most basic type of looping in JavaScript is the `while` loop. A `while` loop tests an expression and continues to execute the code contained in its {} brackets until the expression evaluates to false.

For example, the following `while` loop executes until the value of `i` is equal to 5:

```
var i = 1;
while (i<5){
```

```
  document.write("Iteration " + i + "<br>");
  i++;
}
```

The resulting output to the browser is as follows:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

`do/while` Loops

Another type of `while` loop is the `do/while` loop. This is useful if you always want to execute the code in the loop at least once and the expression cannot be tested until the code has executed at least once.

For example, the following `do/while` loop executes until the value of day is equal to Wednesday:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
var i=0;
do{
  var day=days[i++];
  document.write("It's " + day + "<br>");
} while (day != "Wednesday");
```

The resulting output to the browser is as follows:

```
It's Monday
It's Tuesday
It's Wednesday
```

`for` Loops

The JavaScript `for` loop allows you to execute code a specific number of times by using a `for` statement that combines three statements into one using the following syntax:

```
for (statement 1; statement 2; statement 3;){
  code to be executed;
}
```

The `for` statement uses those three statements as follows when executing the loop:

- **statement 1**—Executed before the loop begins and not again. This is used to initialize variables that will be used in the loop as conditionals.
- **statement 2**—Expression that is evaluated before each iteration of the loop. If the expression evaluates to true, the loop is executed; otherwise, the `for` loop execution ends.

- **statement 3**—Executed each iteration after the code in the loop has executed. This is typically used to increment a counter that is used in statement 2.

To illustrate a `for` loop, check out the following example. The example not only illustrates a basic `for` loop, it also illustrates the capability to nest one loop inside another:

```
for (var x=1; x<=3; x++){
  for (var y=1; y<=3; y++){
    document.write(x + " X " + y + " = " + (x*y) + "<br>");
  }
}
```

The resulting output to the web browser is as follows:

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
```

`for/in` Loops

Another type of for loop is the `for/in` loop. The `for/in` loop executes on any data type that can be iterated on. For the most part, you will use the `for/in` loop on arrays and objects. The following example illustrates the syntax and behavior of the `for/in` loop on a simple array:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  document.write("It's " + days[idx] + "<br>");
}
```

Notice that the variable `idx` is adjusted each iteration through the loop from the beginning array index to the last. The resulting output is as follows:

```
It's Monday
It's Tuesday
It's Wednesday
It's Thursday
It's Friday
```

Interrupting Loops

When working with loops, at times you need to interrupt the execution of code inside the code itself without waiting for the next iteration. There are two ways to do this using the `break` and `continue` keywords.

The `break` keyword stops execution of the `for` or `while` loop completely. The `continue` keyword, on the other hand, stops execution of the code inside the loop and continues on with the next iteration. Consider the following examples:

Using a `break` if the `day` is `Wednesday`:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  if (days[idx] == "Wednesday")
    break;
  document.write("It's " + days[idx] + "<br>");
}
```

When the value is `Wednesday`, loop execution stops completely:

```
It's Monday
It's Tuesday
```

Using a `continue` if the `day` is `Wednesday`:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  if (days[idx] == "Wednesday")
    continue;
  document.write("It's " + days[idx] + "<br>");
}
```

Notice that the write is not executed for `Wednesday` because of the continue; however, the loop execution did complete:

```
It's Monday
It's Tuesday
It's Thursday
It's Friday
```

Creating Functions

One of the most important parts of JavaScript is making code that is reusable by other code. To do this, you combine your code into functions that perform specific tasks. A function is a series of code statements combined in a single block and given a name. The code in the block can then be executed by referencing that name.

Defining Functions

Functions are defined using the keyword `function` followed by a function name that describes the use of the function, list of zero or more arguments in `()` parentheses, and a block of one or more code statements in `{}` brackets. For example, the following is a function definition that writes "Hello World" to the browser:

```
function myFunction(){
  document.write("Hello World");
}
```

To execute the code in `myFunction()`, all you need to do is add the following line to the main JavaScript or inside another function:

```
myFunction();
```

Passing Variables to Functions

Frequently, you will need to pass specific values to functions that they will use when executing their code. Values are passed in comma-delimited form to the function. The function definition will need a list of variable names in the `()` parentheses that match the number being passed in. For example, the following function accepts two arguments, a `name` and `city`, and uses them to build the output string:

```
function greeting(name, city){
  document.write("Hello " + name);
  document.write(". How is the weather in " + city);
}
```

To call the `greeting()` function, we need to pass in a `name` value and a `city` value. The value can be a direct value or a previously defined variable. To illustrate this, the following code will execute the `greeting()` function with a `name` variable and a direct string for the `city`:

```
var name = "Brad";
greeting(name, "Florence");
```

Returning Values from Functions

Often, functions will need to return a value to the calling code. Adding a `return` keyword followed by a variable or value will return that value from the function. For example, the following code calls a function to format a string, assigns the value returned from the function to a variable, and then writes the value to the browser:

```
function formatGreeting(name, city){
  var retStr = "";
  retStr += "Hello <b>" + name + "</b><br>";
  retStr += "Welcome to " + city + "!";
 return retStr;
}
```

```
var greeting = formatGreeting("Brad", "Rome");
document.write(greeting);
```

You can include more than one `return` statement in the function. When the function encounters a `return` statement, code execution of the function is stopped immediately. If the `return` statement contains a value to return, that value is returned. The following example shows a function that tests the input and returns immediately if it is zero:

```
function myFunc(value){
  if (value == 0)
    return;
  code_to_execute_if_value_nonzero;
}
```

TRY IT YOURSELF ▼

Creating JavaScript Functions

To help solidify functions, use the following steps to integrate some functions into a JavaScript application. The following steps take you through the process of creating a function, calling it to execute code, and then handling the results returned:

1. Create a source file named js_functions.html in the lesson05 folder.
2. Add the usual basic elements (html, head, body).
3. Add a `<script>` tag to the `<head>` element to house the JavaScript.
4. Insert the following object literal definition at the beginning of the script. The object will have planet names for attributes, and each hero name is a reference to an array of villains:

```
07       var superData = {"Super Man":["Lex Luther"],
08                        "Bat Man":["Joker", "Riddler",],
09                        "Spider Man":["Green Goblin",
10                                      "Vulture", "Carnage"],
11                        "Thor":["Loki", "Frost Giants"]};
```

5. Add the following function that will be called by the `onload` event. In this function, you use a nested `for/in` loop to iterate through the `superData` object attributes. The outer loop gets the hero `name` and the inner loop loops through the index of the `villains` array:

```
12       function writeIt(){
13         for (hero in superData){
14           var villains = superData[hero];
15           for (villainIdx in villains){
16             var villain = villains[villainIdx];
17             var listItem = makeListItem(hero, villain);
18             document.write(listItem);
```

```
19            }
20          }
21        }
```

6. Notice that on line 16 of the `writeIt()` function is a call to `makeListItem()`. That function needs to return a value that can be used in line 17 to write to the document. Add the following code to create the function. The function takes two arguments: a `name` and a `value`, then generates an HTML string to create a `<li>` element and returns the string:

```
22        function makeListItem(name, value){
23          var itemStr = "<li>" + name + ": " + value + "</li>";
24          return itemStr;
25        }
```

7. Save the file and open it in a web browser. You should see the results shown in Figure 5.4. You have just created two JavaScript functions: one that takes no arguments and does not return a value and the other that takes two arguments and returns a formatted HTML string containing the argument strings.

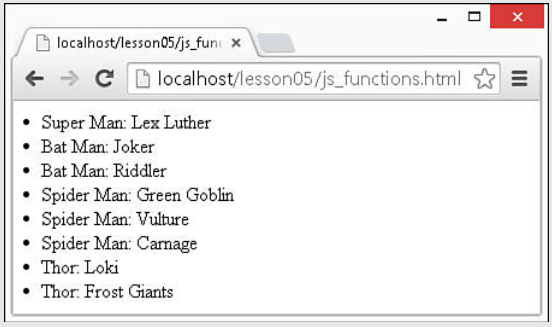

FIGURE 5.4
The function `writeIt()` is executed, which iterates through the `moonData` object and makes calls to the `makeListItem()` function to format the planet and moon names as an HTML `<li>` element.

LISTING 5.4 js_functions.html Simple Example of JavaScript Functions

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>JavaScript Functions</title>
05     <meta charset="utf-8" />
06     <script>
07       var superData = {"Super Man":["Lex Luther"],
08                        "Bat Man":["Joker", "Riddler"],
09                        "Spider Man":["Green Goblin",
```

```
                                    "Vulture", "Carnage"],
                        "Thor":["Loki", "Frost Giants"]};
      function writeIt(){
        for (hero in superData){
          var villains = superData[hero];
          for (villainIdx in villains){
            var villain = villains[villainIdx];
            var listItem = makeListItem(hero, villain);
            document.write(listItem);
          }
        }
      }
      function makeListItem(name, value){
        var itemStr = "<li>" + name + ": " + value + "</li>";
        return itemStr;
      }
    </script>
  </head>
  <body onload="writeIt()">
  </body>
</html>
```

Understanding Variable Scope

After you start adding conditions, functions, and loops to your JavaScript applications, you need to understand variable scoping. Variable scope is simply this: "what is the value of a specific variable name at the current line of code being executed."

JavaScript enables you to define both a global and a local version of the variable. The global version is defined in the main JavaScript, and local versions are defined inside functions. When you define a local version in a function, a new variable is created in memory. Within that function, you will be referencing the local version. Outside that function, you will be referencing the global version.

To understand variable scoping a bit better, consider the following code:

```
<script>
  var myVar = 1;
  function writeIt(){
    var myVar = 2;
    document.write(myVar);
    writeMore();
  }
  function writeMore(){
```

```
09     document.write(myVar);
10   }
11 </script>
```

The global variable `myVar` is defined on line 2. Then on line 4, a local version is defined within the `writeIt()` function. So, line 5 will write to the document. Then in line 6, `writeMore()` is called. Because there is no local version of `myVar` defined in `writeMore()`, the value of the global `myVar` is written in line 9.

Adding Error Handling

An important part of JavaScript coding is adding error handling for instances where there may be problems. By default, if a code exception occurs because of a problem in your JavaScript, the script fails and does not finish loading. This is not usually the desired behavior.

`Try/Catch` Blocks

To prevent your code from totally bombing out, use `try/catch` blocks that can handle problems inside your code. If JavaScript encounters an error when executing code in a `try/catch` block, it will jump down and execute the `catch` portion instead of stopping the entire script. If no error occurs, all of the `try` will be executed and none of the `catch`.

For example, the following `try/catch` block will execute any code that replaces `your_code_here`. If an error occurs executing that code, the error message followed by the string ": happened when loading the script" will be written to the document:

```
try {
    your_code_here
} catch (err) {
  document.write(err.message + ": happened when loading the script");
}
```

Throw Your Own Errors

You can also throw your own errors using a `throw` statement. The following code illustrates how to add throws to a function to `throw` an error, even if a script error does not occur:

```
01 <script>
02   function sqrRoot(x) {
03     try {
04       if(x=="")    throw "Can't Square Root Nothing";
05       if(isNaN(x)) throw "Can't Square Root Strings";
06       if(x<0)      throw "Sorry No Imagination";
07       return "sqrt("+x+") = " + Math.sqrt(x);
08     } catch(err){
09       return err;
10     }
```

```
11   }
12   function writeIt(){
13     document.write(sqrRoot("four") + "<br>");
14     document.write(sqrRoot("") + "<br>");
15     document.write(sqrRoot("4") + "<br>");
16     document.write(sqrRoot("-4") + "<br>");
17   }
18 </script>
```

The function `sqrRoot()` accepts a single argument `x`. It then tests `x` to verify that it is a positive number and returns a string with the square root of `x`. If `x` is not a positive number, the appropriate error is thrown and returned to `writeIt()`.

Using `finally`

Another valuable tool in exception handling is the `finally` keyword. A `finally` keyword can be added to the end of a `try/catch` block. After the `try/catch` blocks are executed, the `finally` block is always executed. It doesn't matter if an error occurs and is caught or if the `try` block is fully executed.

Following is an example of using a `finally` block inside a web page:

```
function testTryCatch(value){
  try {
    if (value < 0){
      throw "too small";
    } else if (value > 10){
      throw "too big";
    }
    your_code_here
  } catch (err) {
    document.write("The number was " + err.message");
  } finally {
    document.write("This is always written.");
  }
}
```

Summary

In this lesson, you learned the basics of adding jQuery and JavaScript to web pages. The basic data types that are used in JavaScript and, consequently, jQuery were described. You learned some of the basic syntax of applying conditional logic to JavaScript applications. You also learned how to compartmentalize your JavaScript applications into functions that can be reused in other locations. Finally, you learned some ways to handle JavaScript errors in your script before the browser receives an exception.

Q&A

Q. When should you use a regular expression in string operations?

A. That depends on your understanding of regular expressions. Those who use regular expressions frequently and understand the syntax well would almost always rather use a regular expression because they are so versatile. If you are not familiar with regular expressions, it takes time to figure out the syntax, and so you will want to use them only when you need to. The bottom line is that if you need to manipulate strings frequently, it is absolutely worth it to learn regular expressions.

Q. Can I load more than one version of jQuery at a time?

A. Sure, but there really isn't a valid reason to do that. The one that gets loaded last will overwrite the functionality of the previous one. Any functions from the first one that were not overwritten may be completely unpredictable because of the mismatch in libraries. The best bet is to develop and test against a specific version and update to a newer version only when there is added functionality that you want to add to your web page.

Workshop

The workshop consists of a set of questions and answers designed to solidify your understanding of the material covered in this lesson. Try to answer the questions before looking at the answers.

Quiz

1. What is the difference between `==` and `===` in JavaScript?
2. What is the difference between the `break` and `continue` keywords?
3. When should you use a `finally` block?
4. What is the resulting value when you add a string "1" to a number 1, ("1"+1)?

Quiz Answers

1. `==` compares only the relative value; `===` compares the value and the type.
2. `break` will stop executing the loop entirely, whereas `continue` will only stop executing the current iteration and then move on to the next.
3. When you have code that needs to be executed even if a problem occurs in the `try` block.
4. The string "11" because the number is converted to a string and then concatenated.

Exercises

1. Open js_functions.html and modify it to create a table instead of a list. You will need to add code to the `writeIt()` function that writes the `<table>` open tag before iterating through the planets and then the closing tag after iterating through the planets. Then modify the `makeListItem()` function to return a string in the form of:

   ```
   <tr><td>planent</td><td>moon</td></tr>
   ```

2. Modify if_logic.html to include some additional times with different messages and images. For example, between 8 and 9, you could add the message "go to work" with a car icon, and between 5 and 6, you could add the message "time to go home" with a home icon. You will need to add some additional cases to the switch statement and set the `timeOfDay` value accordingly.