IV
JAVASCRIPT FOR BEHAVIOR

CHAPTER 21

INTRODUCTION TO JAVASCRIPT

by Mat Marquis

IN THIS CHAPTER

What JavaScript is and isn't

Variables and arrays

if/else statements and loops

Native and custom functions

Browser objects

Event handlers

In this chapter, I'm going to introduce you to JavaScript. Now, it's possible you've just recoiled a little bit, and I understand. We're into full-blown "programming language" territory now, and that can be a little intimidating. I promise, it's not so bad!

We'll start by going over what JavaScript is—and what it isn't—and discuss some of the ways it is used. The majority of the chapter is made up of an introduction to JavaScript syntax—variables, functions, operators, loops, stuff like that. Will you be coding by the end of the chapter? Probably not. But you will have a good head start toward understanding what's going on in a script when you see one. I'll finish up with a look at some of the ways you can manipulate the browser window and tie scripts to user actions such as clicking or submitting a form.

WHAT IS JAVASCRIPT?

If you've made it this far in the book, you no doubt already know that JavaScript is a programming language that adds interactivity and custom behaviors to our sites. It is a client-side scripting language, which means it runs on the user's machine and not on the server, as other web programming languages such as PHP and Ruby do. That means JavaScript (and the way we use it) is reliant on the browser's capabilities and settings. It may not even be available at all, either because the user has chosen to turn it off or because the device doesn't support it, which good developers keep in mind and plan for. JavaScript is also what is known as a dynamic and loosely typed programming language. Don't sweat this description too much; I'll explain what all that means later.

First, I want to establish that JavaScript is kind of misunderstood.

What It Isn’t

Right off the bat, the name is pretty confusing. Despite its name, JavaScript has nothing to do with Java. It was created by Brendan Eich at Netscape in 1995 and originally named “LiveScript.” But Java was all the rage around that time, so for the sake of marketing, “LiveScript” became “JavaScript.” Or just “JS,” if you want to sound as cool as one possibly can while talking about JavaScript.

JS also has something of a bad reputation. For a while it was synonymous with all sorts of unscrupulous internet shenanigans—unwanted redirects, obnoxious pop-up windows, and a host of nebulous “security vulnerabilities,” just to name a few. There was a time when JavaScript allowed less reputable developers to do all these things (and worse), but modern browsers have largely caught on to the darker side of JavaScript development and locked it down. We shouldn’t fault JavaScript itself for that era, though. As the not-so-old cliché goes, “with great power comes great responsibility.” JavaScript has always allowed developers a tremendous amount of control over how pages are rendered and how our browsers behave, and it’s up to us to use that control in responsible ways.

What It Is

Now we know what JavaScript isn’t: it isn’t related to Java, and it isn’t a mustachioed villain lurking within your browser, wringing its hands and waiting to alert you to “hot singles in your area.” Let’s talk more about what JavaScript *is*.

NOTE

JavaScript was standardized in 1996 by the European Computer Manufacturers Association (ECMA), which is why you sometimes hear it called ECMAScript.

JavaScript is a lightweight but incredibly powerful scripting language. We most frequently encounter it through our browsers, but JavaScript has snuck into everything from native applications to PDFs to ebooks. Even web servers themselves can be powered by JavaScript.

As a dynamic programming language, JavaScript doesn’t need to be run through any form of compiler that interprets our human-readable code into something the browser can understand. The browser effectively reads the code the same way we do and interprets it on the fly.

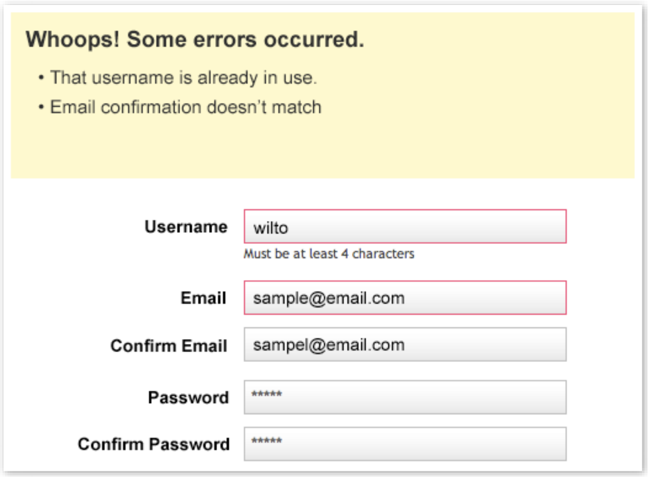
JavaScript is also loosely typed. All this means is that we don’t necessarily have to tell JavaScript what a variable is. If we’re setting a variable to a value of 5, we don’t have to programmatically specify that variable as a number; that is, 5 is a number, and JavaScript recognizes it as such.

Now, you don’t necessarily need to memorize these terms to get started writing JS, mind you—to be honest, I didn’t. This is just to introduce you to a few of the terms you'll hear often while you're learning JavaScript, and they'll start making more and more sense as you go along. This is also to provide you with conversation material for your next cocktail party! "Oh, me? Well, I've been really into loosely typed dynamic scripting languages lately." People will just nod silently at you, which I think means you're doing well conversationally. I don't go to a lot of cocktail parties.

What JavaScript Can Do

Most commonly we'll encounter JavaScript as a way to add interactivity to a page. Whereas the "structural" layer of a page is our HTML markup, and the "presentational" layer of a page is made up of CSS, the third "behavioral" layer is made up of our JavaScript. All of the elements, attributes, and text on a web page can be accessed by scripts using the DOM (Document Object Model), which we'll be looking at in **Chapter 22, Using JavaScript.** We can also write scripts that react to user input, altering either the contents of the page, the CSS styles, or the browser's behavior on the fly.

You've likely seen this in action if you've ever attempted to register for a website, entered a username, and immediately received feedback that the username you've entered is already taken by someone else (FIGURE 21-1). The red border around the text input and the appearance of the "sorry, this username is already in use" message are examples of JavaScript altering the contents of the page. Blocking the form submission is an example of JavaScript altering the browser's default behavior. Ultimately, verifying this information is a job for the server—but JavaScript allows the website to make that request and offer immediate feedback without the need for a page reload.



Whoops! Some errors occurred.

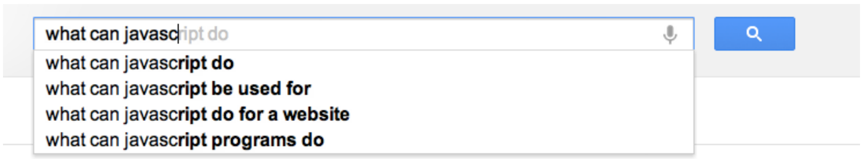
- That username is already in use.
- Email confirmation doesn't match

Username	wilto Must be at least 4 characters
Email	sample@email.com
Confirm Email	sampel@email.com
Password	*****
Confirm Password	*****

FIGURE 21-1. JavaScript inserts a message, alters styles to make errors apparent, and blocks the form from submitting. It can also detect whether the email entries match, but the username would more likely be detected by a program on the server.

In short, JavaScript allows you to create highly responsive interfaces that improve the user experience and provide dynamic functionality, without waiting for the server to load up a new page. For example, we can use JavaScript to do any of the following:

- Suggest the complete term a user might be entering in a search box as he types. You can see this in action on Google.com (FIGURE 21-2).



what can javascript do
what can javascript be used for
what can javascript do for a website
what can javascript programs do

FIGURE 21-2. Google.com uses JavaScript to automatically complete a search term as it is typed in.

- Request content and information from the server and inject it into the current document as needed, without reloading the entire page—this is commonly referred to as "Ajax."
- Show and hide content based on a user clicking a link or heading, to create a "collapsible" content area (FIGURE 21-3).

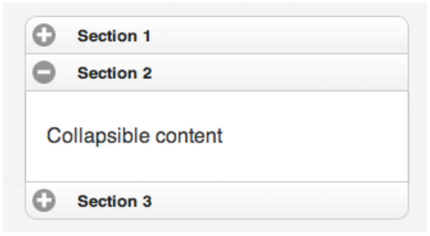
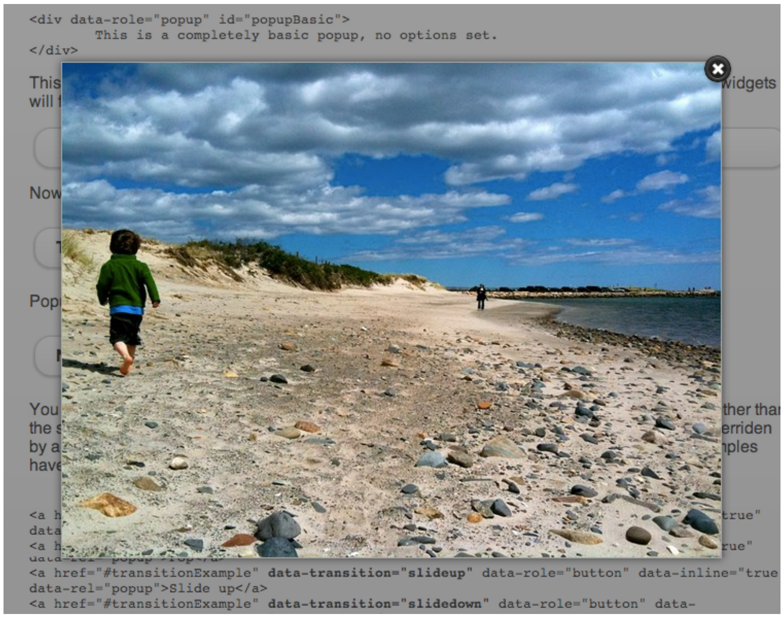


FIGURE 21-3. JavaScript can be used to reveal and hide portions of content.

- Test for browsers' individual features and capabilities. For example, one can test for the presence of "touch events," indicating that the user is interacting with the page through a mobile device's browser, and add more touch-friendly styles and interaction methods.
- Fill in gaps where a browser's built-in functionality falls short, or add some of the features found in newer browsers to older browsers. These kinds of scripts are usually called shims or polyfills.
- Load an image or content in a custom-styled "lightbox"—isolated on the page with CSS—after a user clicks a thumbnail version of the image (FIGURE 21-4).

This list is nowhere near exhaustive!



```
<div data-role="popup" id="popupBasic">
        This is a completely basic popup, no options set.
</div>
```

```
<a href="#transitionExample" data-transition="slideup" data-role="button" data-inline="true
data-rel="popup">Slide up</a>
<a href="#transitionExample" data-transition="slidedown" data-role="button" data-
```

FIGURE 21-4. JavaScript can be used to load images into a lightbox-style gallery.

ADDING JAVASCRIPT TO A PAGE

As with CSS, you can embed a script right in a document or keep it in an external file and link it to the page. Both methods use the **`script`** element.

Embedded Script

To embed a script on a page, just add the code as the content of a **`script`** element:

```
<script>
  … JavaScript code goes here
</script>
```

> **NOTE**
>
> *For documents written in the stricter XHTML syntax, you must identify the content of the script element as CDATA by wrapping the code in the following wrapper:*
>
> ```
> <script type="text/javascript">
>   // <![CDATA[
>   …JavaScript code goes here
>   // ]]>
> </script>
> ```

External Scripts

The other method uses the **`src`** attribute to point to a script file (with a *.js* suffix) by its URL. In this case, the **`script`** element has no content:

```
<script src="my_script.js"></script>
```

The advantage to external scripts is that you can apply the same script to multiple pages (the same benefit external style sheets offer). The downside, of course, is that each external script requires an additional HTTP request of the server, which slows down performance.

Script Placement

The **script** element can go anywhere in the document, but the most common places for scripts are in the **head** of the document and at the very end of the **body**. It is recommended that you don't sprinkle them throughout the document, because they would be difficult to find and maintain.

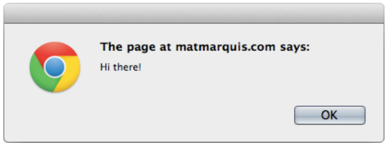
For most scripts, the end of the document, just before the **</body>** tag, is the preferred placement because the browser will be done parsing the document and its DOM structure:

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    …contents of page…
    <script src="script.js"></script>
  </body>
</html>
```

Consequently, that information will be ready and available by the time it gets to the scripts, and they can execute faster. In addition, the script download and execution blocks the rendering of the page, so moving the script to the bottom improves the perceived performance.

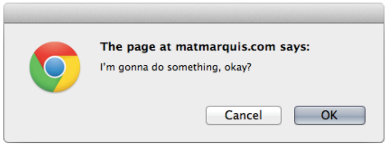
However, in some cases, you might want your script to do something before the body completely loads, so putting it in the **head** will result in better performance. For example, Modernizr (the feature detection tool discussed in **Chapter 19, More CSS Techniques**) recommends its script be placed in the head so the feature detection tests can be run up front.

```
alert("Hi there");
```



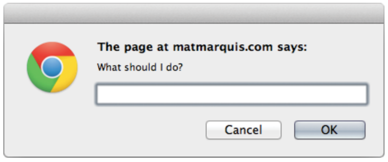
The page at matmarquis.com says:
Hi there!

```
confirm("I'm gonna do something, okay?");
```



The page at matmarquis.com says:
I'm gonna do something, okay?

```
prompt("What should I do?");
```



The page at matmarquis.com says:
What should I do?

FIGURE 21-5. Built-in JavaScript functions: **alert()** (top), **confirm()** (middle), and **prompt()** (bottom).

THE ANATOMY OF A SCRIPT

There's a reason why the book *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly) is 1,100 pages long. There's a *lot* to say about JavaScript! In this section, we have only a few pages to make you familiar with the basic building blocks of JavaScript so you can begin to understand scripts when you encounter them. Many developers have taught themselves to program by finding existing scripts and adapting them for their own needs. After some practice, they are ready to start writing their own from scratch. Recognizing the parts of a script is the first step, so that's where we'll start.

Originally, JavaScript's functionality was mostly limited to crude methods of interaction with the user. We could use a few of JavaScript's built-in functions (FIGURE 21-5) to provide user feedback, such as **alert()** to push a notification to a user, and **confirm()** to ask a user to approve or decline an action. To request the user's input, we were more or less limited to the built-in **prompt()** function. Although these methods still have their time and place today, they're jarring, obtrusive, and—in common opinion, at least—fairly obnoxious ways of interacting with users. As JavaScript has evolved over time, we've been afforded much more graceful ways of adding behavior to our pages, creating a more seamless experience for our users.

In order to take advantage of these interaction methods, we have to first understand the underlying logic that goes into scripting. These are logic patterns common to all manner of programming languages, although the syntax may vary. To draw a parallel between programming languages and spoken languages: although the vocabulary may vary from one language to another, many grammar patterns are shared by the majority of them.

By the end of this section, you're going to know about variables, arrays, comparison operators, **`if`**/**`else`** statements, loops, functions, and more. Ready?

The Basics

There are a few common syntactical rules that wind their way though all of JavaScript.

It is important to know that JavaScript is case-sensitive. A variable named **`myVariable`**, a variable named **`myvariable`**, and a variable named **`MYVariable`** will be treated as three different objects.

JavaScript is case-sensitive.

Also, whitespace such as tabs and spaces is ignored, unless it's part of a string of text and enclosed in quotes. All of the character spaces added to scripts such as the ones in this chapter are for the benefit of humans—they make reading through the code easier. JavaScript doesn't see them.

Statements

A script is made up of a series of statements. A statement is a command that tells the browser what to do. Here is a simple statement that makes the browser display an alert with the phrase "Thank you":

```
alert("Thank you.");
```

The semicolon at the end of the statement tells JavaScript that it's the end of the command, just as a period ends a sentence. According to the JavaScript standard, a line break will also trigger the end of a command, but it is a best practice to end each statement with a semicolon.

Comments

JavaScript allows you to leave comments that will be ignored at the time the script is executed, so you can provide reminders and explanations throughout your code. This is especially helpful if this code is likely to be edited by another developer in the future.

There are two methods of using comments. For single-line comments, use two slash characters (//) at the beginning of the line. You can put single-line comments on the same line as a statement, as long as the comment comes after the statement. It does not need to be closed, as a line break effectively closes it.

```
// This is a single-line comment.
```

Multiple-line comments use the same syntax that you've seen in CSS. Everything within the `/* */` characters is ignored by the browser. You can use this syntax to "comment out" notes and even chunks of the script when troubleshooting.

```
/* This is a multiline comment.
Anything between these sets of characters will be
completely ignored when the script is executed.
This form of comment needs to be closed. */
```

I'll be using the single-line comment notation to add short explanations to example code, and we'll make use of the **alert()** function we saw earlier (FIGURE 21-5) so we can quickly view the results of our work.

Variables

A variable is like an information container.

If you're anything like me, the very term "variables" triggers nightmarish flashbacks to eighth-grade math class. The premise is pretty much the same, though your teacher doesn't have a bad comb-over this time around.

A variable is like an information container. You give it a name and then assign it a value, which can be a number, text string, an element in the DOM, or a function—anything, really. This gives us a convenient way to reference that value later by name. The value itself can be modified and reassigned in whatever way our scripts' logic dictates.

The following declaration creates a variable with the name **foo** and assigns it the value 5:

```
var foo = 5;
```

We start by declaring the variable by using the **var** keyword. The single equals sign (=) indicates that we are assigning it a value. Because that's the end of our statement, we end the line with a semicolon. Variables can also be declared without the **var** keyword, which impacts what part of your script will have access to the information they contain. We'll discuss that further in the section **"Variable Scope and the var Keyword"** later in this chapter.

You can use anything you like as a variable name, but make sure it's a name that will make sense to you later. You wouldn't want to name a variable something like **data**; it should describe the information it contains. In our earlier very specific example, **productName** might be a more useful name than **foo**. There are a few rules for naming a variable:

- It must start with a letter or an underscore.
- It may contain letters, digits, and underscores in any combination.
- It may not contain character spaces. As an alternative, use underscores in place of spaces, or close up the space and use camel case instead (for example, `my_variable` or `myVariable`).
- It may not contain special characters (e.g., `! . , / \ + * =`).

You can change the value of a variable at any time by redeclaring it anywhere in your script. Remember: JavaScript is case-sensitive, and so are those variable names.

Data types

The values we assign to variables fall under a few distinct data types:

Undefined

The simplest of these data types is likely **`undefined`**. If we declare a variable by giving it a name but no value, that variable contains a value of **`undefined`**.

```
var foo;
alert(foo); // This will open a dialog containing "undefined".
```

Odds are you won't find a lot of use for this right away, but it's worth knowing for the sake of troubleshooting some of the errors you're likely to encounter early on in your JavaScript career. If a variable has a value of **`undefined`** when it shouldn't, you may want to double-check that it has been declared correctly or that there isn't a typo in the variable name. (We've all been there.)

Null

Similar to **`undefined`**, assigning a variable of **`null`** (again, case-sensitive) simply says, "Define this variable, but give it no inherent value."

```
var foo = null;
alert(foo); // This will open a dialog containing "null".
```

Numbers

You can assign variables numeric values.

```
var foo = 5;
alert(foo); // This will open a dialog containing "5".
```

The word **`foo`** now means the exact same thing as the number 5 as far as JavaScript is concerned. Because JavaScript is loosely typed, we don't have to tell our script to treat the variable **`foo`** as the *number* 5. The variable behaves the same as the number itself, so you can do things to it that you would do to any other number by using classical mathematical notation: **`+`**, **`-`**, **`*`**, and **`/`** for plus, minus, multiply, and divide, respectively. In this example, we use the plus sign (**`+`**) to add **`foo`** to itself (**`foo + foo`**).

```
var foo = 5;
alert(foo + foo); // This will alert "10".
```

Strings

Another type of data that can be saved to a variable is a string, which is basically a line of text. Enclosing characters in a set of single or double quotes indicates that it's a string, as shown here:

```
var foo = "five";
alert( foo ); // This will alert "five"
```

The variable **`foo`** is now treated exactly the same as the word *five*. This applies to any combination of characters: letters, numbers, spaces, and so on. If the value is wrapped in quotation marks, it will be treated as a string of text. If we were to wrap the number 5 in quotes and assign it to a variable, that variable wouldn't behave as a number; instead, it would behave as a string of text containing the character "5."

Earlier we saw the plus sign (**`+`**) used to add numbers. When the plus sign is used with strings, it sticks the strings together (called concatenation) into one long string, as shown in this example.

```
var foo = "bye"
alert(foo + foo);  // This will alert "byebye"
```

Notice what the alert returns in the following example when we define the value 5 in quotation marks, treating it as a string instead of a number:

```
var foo = "5";
alert( foo + foo ); // This will alert "55"
```

If we concatenate a string and a number, JavaScript will assume that the number should be treated as a string as well, since the math would be impossible.

```
var foo = "five";
var bar = 5;
alert( foo + bar ); // This will alert "five5"
```

Booleans

We can also assign a variable a true or false value. This is called a Boolean value, and it is the lynchpin for all manner of advanced logic. Boolean values use the **`true`** and **`false`** keywords built into JavaScript, so quotation marks are not necessary.

```
var foo = true; // The variable "foo" is now true
```

Just as with numbers, if we were to wrap the preceding value in quotation marks, we'd be saving the word *true* to our variable instead of the inherent value of **`true`** (i.e., "not false").

In a sense, everything in JavaScript has either an inherently true or false value. For example, **`null`**, **`undefined`**, **`0`**, and empty strings (**`" "`**) are all inherently false, while every other value is inherently true. These values, although not identical to the Booleans **`true`** and **`false`**, are commonly referred to as being "truthy" and "falsy." I promise I didn't make that up.

Arrays

An array is a group of multiple values (called members) that can be assigned to a single variable. The values in an array are said to be indexed, meaning you can refer to them by number according to the order in which they appear in the list. The first member is given the index number 0, the second is 1, and so on, which is why one almost invariably hears us nerds start counting things at zero—because that's how JavaScript counts things, and many other programming languages do the same. We can avoid a lot of future coding headaches by keeping this in mind.

So, let's say our script needs all of the variables we defined earlier. We could define them three times and name them something like **foo1**, **foo2**, and so on, or we can store them in an array, indicated by square brackets ([ ]).

```
var foo = [5, "five", "5"];
```

Now anytime you need to access any of those values, you can grab them from the single **foo** array by referencing their index number:

```
alert( foo[0] ); // Alerts "5"
alert( foo[1] ); // Alerts "five"
alert( foo[2] ); // Also alerts "5"
```

Comparison Operators

Now that we know how to save values to variables and arrays, the next logical step is knowing how to compare those values. There is a set of special characters called comparison operators that evaluate and compare values in different ways:

`==`	Is equal to
`!=`	Is not equal to
`===`	Is identical to (equal to and of the same data type)
`!==`	Is not identical to
`>`	Is greater than
`>=`	Is greater than or equal to
`<`	Is less than
`<=`	Is less than or equal to

There's a reason all of these definitions read as parts of a statement. In comparing values, we're making an assertion, and the goal is to obtain a result that is either inherently true or inherently false. When we compare two values, JavaScript evaluates the statement and gives us back a Boolean value depending on whether the statement is true or false.

```
alert( 5 == 5 ); // This will alert "true"
alert( 5 != 6 ); // This will alert "true"
alert( 5 < 1 ); // This will alert "false"
```

Equal versus identical

The tricky part is understanding the difference between “equal to” (`==`) and “identical to” (`===`). We already learned that all of these values fall under a certain data type. For example, a string of “5” and a number 5 are similar, but they’re not quite the same thing.

Well, that’s exactly what `===` is meant to check.

```
alert( "5" == 5 ); // This will alert "true". They're both "5".

alert( "5" === 5 );
/* This will alert "false". They're both "5", but they're not the same
data type. */

alert( "5" !== 5 );
/* This will alert "true", since they're not the same data type. */
```

Even if you have to read through this part a couple of times, understanding the difference between “equal” and “identical to” means you’ve already begun to adopt the special kind of crazy one needs to be a programmer. Welcome! You’re in good company.

WARNING

Be careful not to accidentally use a single equals sign, or you’ll be reassigning the value of the first variable to the value of the second variable!

Mathematical operators

The other type of operator is a mathematical operator, which performs mathematical functions on numeric values (and, of course, variables that contain numeric values). We touched briefly on the straightforward mathematical operators for add (`+`), subtract (`-`), multiply (`*`), and divide (`/`). There are also some useful shortcuts you should be aware of:

`+=`	Adds the value to itself
`++`	Increases the value of a number (or a variable containing a number value) by 1
`--`	Decreases the value of a number (or a variable containing a number value) by 1

if/else statements

`if/else` statements are how we get JavaScript to ask itself a true/false question. They are more or less the foundation for all the advanced logic that can be written in JavaScript, and they’re about as simple as programming gets. In fact, they’re almost written in plain English. The structure of a conditional statement is as follows:

```
if( true ) {
   // Do something.
}
```

It tells the browser “if this condition is met, then execute the commands listed between the curly brackets (`{ }`).” JavaScript doesn’t care about whitespace in our code, remember, so the spaces on either side of the `( true )` are purely for the sake of more readable code.

Here is a simple example using the array we declared earlier:

```
var foo = [5, "five", "5"];

if( foo[1] === "five" ) {
  alert("This is the word five, written in plain English.");
}
```

Since we're making a comparison, JavaScript is going to give us a value of either **`true`** or **`false`**. The highlighted line of code says "true or false: the value of the **`foo`** variable with an index of **`1`** is identical to the word 'five'?"

In this case, the alert would fire because the **`foo`** variable with an index of **`1`** (the second in the list, if you'll remember) is identical to "five". It is indeed true, and the alert fires.

We can also explicitly check if something is false by using the **`!=`** comparison operator, which reads as "not equal to."

```
if( 1 != 2 ) {
  alert("If you're not seeing this alert, we have bigger problems than
JavaScript.");
  // 1 is never equal to 2, so we should always see this alert.
}
```

I'm not much good at math, but near as I can tell, 1 will never be equal to 2. JavaScript says, "That '1 is not equal to 2' line is a true statement, so I'll run this code."

If the statement doesn't evaluate to **`true`**, the code inside the curly brackets will be skipped over completely:

```
if( 1 == 2 ) {
  alert("If you're seeing this alert, we have bigger problems than
JavaScript.");
// 1 is not equal to 2, so this code will never run.
}
```

Idiomatic JavaScript

There is an effort in the JavaScript community to create a style guide for writing JavaScript code. The document "Principles of Writing Consistent, Idiomatic JavaScript" states the following: "All code in any code-base should look like a single person typed it, no matter how many people contributed." To achieve that goal, a group of developers has written an Idiomatic Style Manifesto that describes how whitespace, line breaks, quotation marks, functions, variables, and more should be written to achieve "beautiful code." Learn more about it at *github.com/rwldrn/idiomatic.js/*.

That covers "if," but what about "else"?

Lastly—and I promise we're almost done here—what if we want to do one thing if something is true and something *else* if that thing is false? We could write two **`if`** statements, but that's a little clunky. Instead, we can just say, "else, do something…else."

```
var test = "testing";
if( test == "testing" ) {
      alert( "You haven't changed anything." );
} else {
    alert( "You've changed something!" );
}
```

EXERCISE 21-1. English-to-JavaScript translation

In this quick exercise, you can get a feel for variables, arrays, and **`if`**/**`else`** statements by translating the statements written in English into lines of JavaScript code. You can find the answers in **Appendix A**.

1. Create a variable called **`friends`** and assign it an array with four of your friends' names.
2. Show the user a dialog that displays the third name in your list of **`friends`**.
3. Create a variable called **`name`** and assign it a string value that is your first name.
4. If the value of **`name`** is identical to **`Jennifer`**, show the user a dialog box that says, "That's my name too!"
5. Create a variable called **`myVariable`** and assign it a number value between 1 and 10. If **`myVariable`** is greater than five, show the user a dialog that says "upper." If not, show the user a dialog that says "lower."

Changing the value of the **`test`** variable to something else—anything other than the word *testing*—will trigger the alert "You've changed something!"

EXERCISE 21-1 gives you a chance to write a bit of JavaScript yourself.

Loops

There are cases in which we'll want to go through every item in an array and do something with it, but we won't want to write out the entire list of items and repeat ourselves a dozen or more times. You are about to learn a technique of *devastating power*, readers: loops.

I know. Maybe I overstated how exciting loops can be, but they *are* incredibly useful. With what we've covered already, we're getting good at dealing with single variables, but that can get us only so far. Loops allow us to easily deal with huge sets of data.

Say we have a form that requires none of the fields to be left blank. If we use the DOM to fetch every text input on the page, the DOM provides an array of every text input element. (I'll tell you more about how the DOM does this in the next chapter.) We could check every value stored in that array one item at a time, sure, but that's a lot of code and a maintenance nightmare. If we use a loop to check each value, we won't have to modify our script, regardless of how many fields are added to or removed from the page. Loops allow us to act on every item in an array, regardless of that array's size.

There are several ways to write a loop, but the **`for`** method is one of the most popular. The basic structure of a **`for`** loop is as follows:

```
for( initialize the variable;  test the condition;  alter the value; ) {
   // do something
}
```

Here's an example of a **`for`** loop in action:

```
for( var i = 0; i < 2; i++ ) {
  alert( i ); // This loop will trigger three alerts, reading "0",
"1", and "2" respectively.
}
```

That's a little dense, so let's break it down:

`for()`

First, we're calling the **`for()`** statement, which is built into JavaScript. It says, "For every time this is true, do this." Next we need to supply that statement with some information.

`var i = 0;`

This creates a new variable, **`i`**, with its value set to zero. You can tell it's a variable by the single equals sign. More often than not, you'll see coders using the letter "i" (short for "index") as the variable name, but keep in mind that you could use any variable name in its place. It's a common convention, not a rule.

We set that initial value to 0 because we want to stay in the habit of counting from zero up. That's where JavaScript starts counting, after all.

`i <= 2;`

With **`i <= 2;`**, we're saying, "for as long as **`i`** is less than or equal to 2, keep on looping." Since we're counting from zero, that means the loop will run three times.

`i++`

Finally, **`i++`** is shorthand for "every time this loop runs, add one to the value of **`i`**" (**`++`** is one of the mathematical shortcut operators we saw earlier). Without this step, **`i`** would always equal zero, and the loop would run forever! Fortunately, modern browsers are smart enough not to let this happen. If one of these three pieces is missing, the loop simply won't run at all.

`{` *`script`* **`}`**

Anything inside those curly brackets is executed once for each time the loop runs, which is three times in this case. That **`i`** variable is available for use in the code the loop executes as well, as we'll see next.

Let's go back to the "check each item in an array" example. How would we write a loop to do that for us?

```
var items = ["foo", "bar", "baz"]; // First we create an array.
for( var i = 0; i < items.length; i++ ) {
  alert( items[i] ); // This will alert each item in the array.
}
```

This example differs from our first loop in two key ways:

`items.length`

Instead of using a number to limit the number of times the loop runs, we're using a property built right into JavaScript to determine the "length" of our array, which is the number of items it contains. **`.length`** is just one of the standard properties and methods of the **`Array`** object in JavaScript. In our example, there are three items in the array, so it will loop three times.

`items[i]`

Remember how I mentioned that we can use that **`i`** variable inside the loop? Well, we can use it to reference each index of the array. Good thing we started counting from zero; if we had set the initial value of **`i`** to 1, the first item in the array would have been skipped. The result of our **`for`** loop example is that each item in the array (the text strings **`foo`**, **`bar`**, and **`baz`**) gets returned after each loop and fed to an alert.

Now no matter how large or small that array should become, the loop will execute only as many times as there are items in the array, and will always hold a convenient reference to each item in the array.

There are literally dozens of ways to write a loop in JavaScript, but this is one of the more common patterns you're going to encounter out there in the wild. Developers use loops to perform a number of tasks, such as the following:

- Looping through a list of elements on the page and checking the value of each, applying a style to each, or adding/removing/changing an attribute on each. For example, we could loop through each element in a form and ensure that users have entered a valid value for each before they proceed.
- Creating a new array of items in an original array that have a certain value. We check the value of each item in the original array within the loop, and if the value matches the one we're looking for, we populate a new array with only those items. This turns the loop into a filter of sorts.

Functions

The structure of a function:

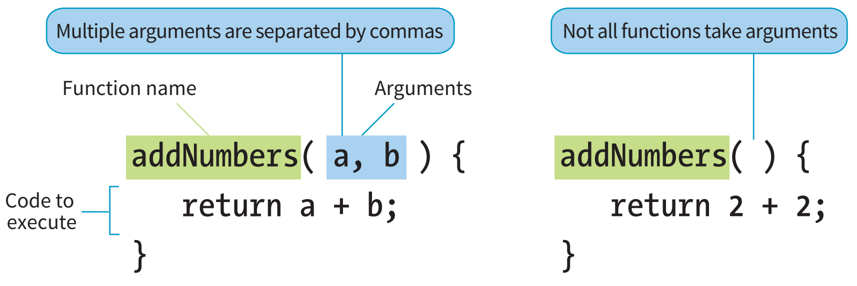
```
function() {
 }
```

I've introduced you to a few functions already in a sneaky way. Here's an example of a function that you might recognize:

```
alert("I've been a function all along!");
```

A function is a bit of code for performing a task that doesn't run until it is referenced or called. `alert()` is a function built into our browser. It's a block of code that runs only when we explicitly tell it to. In a way, we can think of a function as a variable that contains *logic*, in that referencing that variable will run all the code stored inside it. Functions allow code to be reused any time it is referenced so you don't need to write it over and over.

All functions share a common pattern (FIGURE 21-6). The function name is always immediately followed by a set of parentheses (no space), then a pair of curly brackets that contains their associated code. The parentheses sometimes contain additional information used by the function called arguments. Arguments are data that can influence how the function behaves. For example, the `alert()` function we know so well accepts a string of text as an argument, and uses that information to populate the resulting dialog.


Multiple arguments are separated by commas
Not all functions take arguments
Function name
Arguments
Code to execute

```
addNumbers( a, b ) {
   return a + b;
}

addNumbers( ) {
   return 2 + 2;
}
```

FIGURE 21-6. The structure of a function.

There are two types of functions: those that come "out of the box" (native JavaScript functions) and those that you make up yourself (custom functions). Let's look at each.

Native functions

Hundreds of predefined functions are built into JavaScript, including these:

`alert()`, `confirm()`, *and* `prompt()`

These functions trigger browser-level dialog boxes.

`Date()`

Returns the current date and time.

`parseInt("123")`

This function will, among other things, take a string data type containing numbers and turn it into a number data type. The string is passed to the function as an argument.

`setTimeout(`*`functionName`*`, 5000)`

Executes a function after a delay. The function is specified in the first argument, and the delay is specified in milliseconds in the second argument (in the example, 5,000 milliseconds, which equals 5 seconds).

There are scores more beyond this as well. Note that names of functions are case-sensitive, so be sure to write **`setTimeout`** instead of **`SetTimeout`**.

Custom functions

To create a custom function, we type the **`function`** keyword followed by a name for the function, followed by opening and closing parentheses, followed by opening and closing curly brackets:

```
function name() {
   // Our function code goes here.
}
```

Just as with variables and arrays, the function's name can be anything you want, but all the same naming syntax rules apply.

If we were to create a function that just alerts some text (which is a little redundant, I know), it would look like this:

```
function foo() {
   alert("Our function just ran!");
  // This code won't run until we call the function 'foo()'
}
```

We can then call that function and execute the code inside it anywhere in our script by writing the following:

```
foo(); // Alerts "Our function just ran!"
```

We can call this function any number of times throughout our code. It saves a lot of time and redundant coding.

Arguments

Having a function that executes the exact same code throughout your script isn't likely to be all that useful. We can "pass arguments" (provide data) to native and custom functions in order to apply a function's logic to different sets of data at different times. To hold a place for the arguments, create a variable name (or a series of comma-separated names) in the parentheses after the name of the function at the time the function is defined.

An argument is a value or data that a function uses when it runs.

For example, let's say we wanted to create a very simple function that alerts the number of items contained in an array. We've already learned that we can use **`.length`** to get the number of items in an array, so we just need a way to pass the array to be measured into our function. We do that by supplying the array to be measured as an argument. In the code, I've defined a new function named **`alertArraySize()`** and created the variable **`arr`** that holds a place for the argument. That variable will then be available inside the function and will contain whatever argument we pass when we call the function.

```
function alertArraySize(arr) {
        alert(arr.length);
}
```

When we call that function, anything we include between the parentheses after the function name (in this case, **`test`**) will be passed to the argument with the **`arr`** placeholder as the function executes. Here we've defined the variable **`test`** as an array of five items. We've passed that variable to the function, and now that array gets plugged in and the length is returned.

```
var test = [1,2,3,4,5];
alertArraySize(test); // Alerts "5"
```

Returning a value

This part is particularly wild and incredibly useful.

It's pretty common to use a function to calculate something and then give you back a value that you can use elsewhere in your script. We could accomplish this using what we know now, through clever application of variables, but there's a much easier way.

The **`return`** keyword inside a function effectively turns that function into a variable with a dynamic value! This one is a little easier to show than it is to tell, so bear with me while we consider this example:

```
function addNumbers(a,b) {
   return a + b;
}
```

We now have a function that accepts two arguments and adds them together. That wouldn't be much use if the result always lived inside that function, because we wouldn't be able to use the result anywhere else in our script. Here we use the **`return`** keyword to pass the result out of the function. Now any reference you make to that function gives you the result of the function—just like a variable would:

```
alert( addNumbers(2,5) ); // Alerts "7"
```

In a way, the **`addNumbers()`** function is now a variable that contains a dynamic value: the value of our calculation. If we didn't return a value inside our function, the preceding script would alert **`undefined`**, just like a variable that we haven't given a value.

The **`return`** keyword has one catch. As soon as JavaScript sees that it's time to return a value, the function ends. Consider the following:

```
function bar() {
   return 3;
   alert("We'll never see this alert.");
}
```

When you call this function by using **`bar()`**, the alert on the second line never runs. The function ends as soon as it sees it's time to return a value.

Variable Scope and the var Keyword

There are times when you'll want a variable that you've defined within a function to be available anywhere throughout your script. Other times, you may want to restrict it and make it available *only* to the function it lives in. This notion of the availability of the variable is known as its scope. A variable that can be used by any of the scripts on your page is globally scoped, and a variable that's available only within its parent function is locally scoped.

JavaScript variables use functions to manage their scope. If a variable is defined outside a function, it will be globally scoped and available to all scripts. When you define a variable within a function and you want it to be used only by that function, you can flag it as locally scoped by preceding the variable name with the **`var`** keyword:

```
var foo = "value";
```

To expose a variable within a function to the global scope, we omit the **`var`** keyword and simply define the variable:

```
foo = "value";
```

You need to be careful about how you define variables within functions, or you could end up with unexpected results. Take the following JavaScript snippet, for example:

```
function double( num ){
  total = num + num;
  return total;
}
var total = 10;
var number = double( 20 );
alert( total ); // Alerts 40.
```

■ SCOPE CHEAT SHEET

Variable	Location	Scope
`var` identifier value	Outside a function	Global
`var` identifier value	Inside a function	Local
identifier value	Inside a function	Global

You may expect that because you specifically assigned a value of 10 to the variable **`total,`** the **`alert(total)`** function at the end of the script would return 10. But because we didn't scope the **`total`** variable in the function with the **`var`** keyword, it bleeds into the global scope. Therefore, although the variable **`total`** is set to 10, the following statement runs the function and grabs the value for **`total`** defined there. Without the **`var`**, the variable "leaked out."

As you can see, the trouble with global variables is that they'll be shared throughout all the scripting on a page. The more variables that bleed into the global scope, the better the chances you'll run into a "collision" in which a variable named elsewhere (in another script altogether, even) matches one of yours. This can lead to variables being inadvertently redefined with unexpected values, which can lead to errors in your script.

Remember that we can't always control all the code in play on our page. It's very common for pages to include code written by third parties, for example:

- Scripts to render advertisements
- User-tracking and analytics scripts
- Social media "share" buttons

It's best not to take any chances on variable collisions, so when you start writing scripts on your own, locally scope your variables whenever you can (see the sidebar **"Keeping Variables Out of the Global Scope"**).

This concludes our little (OK, not so little) introductory tour of JavaScript syntax. There's a lot more to it, but this should give you a decent foundation for learning more on your own and being able to interpret scripts when you see them. We have just a few more JavaScript-related features to tackle before we look at a few examples.

Keeping Variables Out of the Global Scope

If you want to be sure that all of your variables stay out of the global scope, you can put all of your JavaScript in the following wrapper:

```
<script>
(function() {
  //All your code here!
}());
<script>
```

This little quarantining solution is called an IIFE (Immediately Invoked Functional Expression), and we owe this method and the associated catchy term to Ben Alman (*benalman.com/news/2010/11/immediately-invoked-function-expression/*).

THE BROWSER OBJECT

In addition to being able to control elements on a web page, JavaScript also gives you access to and the ability to manipulate the parts of the browser window itself. For example, you might want to get or replace the URL that is in the browser's address bar, or open or close a browser window.

In JavaScript, the browser is known as the **`window`** object. The **`window`** object has a number of properties and methods that we can use to interact with it. In fact, our old friend **`alert()`** is actually one of the standard browser object methods. TABLE 21-1 lists just a few of the properties and methods that can be used with **`window`** to give you an idea of what's possible. For a complete list, see the Window API reference at MDN Web Docs (*developer.mozilla.org/en-US/docs/Web/API/Window*).

TABLE 21-1. Browser properties and methods.

Property/method	Description
`event`	Represents the state of an event
`history`	Contains the URLs the user has visited within a browser window
`location`	Gives read/write access to the URI in the address bar
`status`	Sets or returns the text in the status bar of the window
`alert()`	Displays an alert box with a specified message and an OK button
`close()`	Closes the current window
`confirm()`	Displays a dialog box with a specified message and an OK and a Cancel button
`focus()`	Sets focus on the current window

EVENTS

JavaScript can access objects in the page and the browser window, but did you know it's also "listening" for certain events to happen? An event is an action that can be detected with JavaScript, such as when the document loads or when the user clicks an element or just moves her mouse over it. HTML 4.0 made it possible for a script to be tied to events on the page, whether initiated by the user, the browser itself, or other scripts. This is known as event binding.

Event handlers "listen" for certain document, browser, or user actions and bind scripts to those actions.

In scripts, an event is identified by an event handler. For example, the **`onload`** event handler triggers a script when the document loads, and the **`onclick`** and **`onmouseover`** handlers trigger a script when the user clicks or mouses over an element, respectively. TABLE 21-2 lists some of the most common event handlers. Keep in mind that these are also case-sensitive.

TABLE 21-2. Common events.

Event handler	Event description
`onblur`	An element loses focus.
`onchange`	The content of a form field changes.
`onclick`	The mouse clicks an object.
`onerror`	An error occurs when the document or an image loads.
`onfocus`	An element gets focus.
`onkeydown`	A key on the keyboard is pressed.
`onkeypress`	A key on the keyboard is pressed or held down.
`onkeyup`	A key on the keyboard is released.
`onload`	A page or an image is finished loading.
`onmousedown`	A mouse button is pressed.
`onmousemove`	The mouse is moved.
`onmouseout`	The mouse is moved off an element.
`onmouseover`	The mouse is moved over an element.
`onmouseup`	A mouse button is released.
`onsubmit`	The submit button is clicked in a form.

There are three common methods for applying event handlers to items within our pages:

- As an HTML attribute
- As a method attached to the element
- Using **`addEventListener()`**

In the upcoming examples of the latter two approaches, we'll use the **`window`** object. Any events we attach to **`window`** apply to the entire document. We'll be using the **`onclick`** event in all of these as well.

As an HTML Attribute

You can specify the function to be run in an attribute in the markup, as shown in the following example:

```
<body onclick="myFunction();"> /* myFunction will now run when the user
clicks anything within 'body' */
```

Although still functional, this is an antiquated way of attaching events to elements within the page. It should be avoided for the same reason we avoid using **`style`** attributes in our markup to apply styles to individual elements. In this case, it blurs the line between the semantic layer and behavioral layers of our pages, and can quickly lead to a maintenance nightmare.

As a Method

This is another somewhat dated approach to attaching events, though it does keep things strictly within our scripts. We can attach functions by using helpers already built into JavaScript:

```
window.onclick = myFunction; /* myFunction will run when the user
clicks anything within the browser window */
```

We can also use an anonymous function rather than a predefined one:

```
window.onclick = function() {
   /* Any code placed here will run when the user clicks anything
within the browser window */
};
```

This approach has the benefit of both simplicity and ease of maintenance, but does have a fairly major drawback: we can bind only one event at a time with this method.

```
window.onclick = myFunction;

window.onclick = myOtherFunction;
```

In the example just shown, the second binding overwrites the first, so when the user clicks inside the browser window, only **`myOtherFunction`** will run. The reference to **`myFunction`** is thrown away.

addEventListener

Although a little more complex at first glance, this approach allows us to keep our logic within our scripts and allows us to perform multiple bindings on a single object. The syntax is a bit more verbose. We start by calling the **`addEventListener()`** method of the target object, and then specify the event in question and the function to be executed as two arguments:

```
window.addEventListener("click", myFunction);
```

Notice that we omit the preceding "on" from the event handler with this syntax.

Like the previous method, **`addEventListener()`** can be used with an anonymous function as well:

```
window.addEventListener("click", function(e) {

});
```

This was just a brief introduction, so I recommend getting more information on **`addEventListener()`** at the "eventTarget.addEventListener" page on the MDN Web Docs (*developer.mozilla.org/en/DOM/element.addEventListener*).

PUTTING IT ALL TOGETHER

Now you have been introduced to many of the important building blocks of JavaScript. You've seen variables, data types, and arrays. You've met **`if`**/**`else`** statements, loops, and functions. You know your browser objects from your event handlers. That's a lot of bits and pieces. Let's walk through a couple of simple script examples to see how they get put together.

Example 1: A Tale of Two Arguments

Here's a simple function that accepts two arguments and returns the greater of the two values:

```
greatestOfTwo( first, second ) {
  if( first > second ) {
    return first;
  } else {
    return second;
  }
}
```

We start by naming our function **`greatestOfTwo`**. We set it up to accept two arguments, which we'll just call "first" and "second" for want of more descriptive words. The function contains an **`if`**/**`else`** statement that returns **`first`** if the first argument is greater than the second, and returns **`second`** if it isn't.

Example 2: The Longest Word

Here's a function that accepts an array of strings as a single argument and returns the longest string in the array. It returns the first occurrence of one of the longest strings (in case they are of the same length).

```
longestWord( strings ) {
  var longest = strings[0];

  for( i = 1; i < strings.length; i++ ) {
    if ( strings[i].length > longest.length ) {
      longest = strings[i];
    }
  }
  return longest;
}
```

First, we name the function and allow it to accept a single argument. Then, we set the **`longest`** variable to an initial value of the first item in the array: **`strings[0]`**. We start our loop at 1 instead of 0 since we already have the first value in the array captured. Each time we iterate through the loop, we compare the length of the current item in the array to the length of the value saved in the **`longest`** variable. If the current item in the array contains more characters than the current value of the **`longest`** variable, we change the value of **`longest`** to that item. If not, we do nothing. After the loop is complete we return the value of **`longest`**, which now contains the longest string in the array.

LEARNING MORE ABOUT JAVASCRIPT

Now that you've seen the basic building blocks and a few simple examples, does it whet your appetite for more? Here are a few resources to take you to the next step:

JavaScript Resources at MDN Web Docs
(*developer.mozilla.org/en-US/docs/Web/JavaScript*)

The folks at MDN Web Docs have assembled excellent tutorials as well as thorough documentation on all the components of JavaScript. It's a great site to visit when you're just starting out, and it is likely to be a go-to reference even after you have years of experience.

***JavaScript for Web Designers* by Mat Marquis (A Book Apart)**

I can say a lot more in a book than in a chapter, so if you're looking for a little more depth in a beginner-level manual, I wrote this book for you.

***Learning JavaScript* by Ethan Brown (O'Reilly)**

For a deeper dive into JavaScript, this book will take you to the next level.

Why not see how you're doing with JavaScript so far with EXERCISE 21-2 and a quick quiz? In the next chapter, you'll see how we use these tools in the context of web design.

EXERCISE 21-2. You try it

In this exercise, you will write a script that updates the page's title in the browser window with a "new messages" count. You may have encountered this sort of script in the wild from time to time. We're going to assume for the sake of the exercise that this is going to become part of a larger web app someday, and we're tasked only with updating the page title with the current "unread messages" count.

I've created a document for you already (*title.html*), which is available in the *materials* folder for this chapter on *learningwebdesign.com*. The resulting script is in **Appendix A**.

1. Start by opening *title.html* in a browser. You'll see a blank page, with the title element already filled out. If you look up at the top of your browser window, you'll notice it reads "Million Dollar WebApp".
2. Now open the document in a text editor. You'll find a **`script`** element containing a comment just before the closing **`</body>`** tag. Feel free to delete the comment.
3. If we're going to be changing the page's title, we should save the original first. Create a variable named **`originalTitle`**. For its value, we'll have the browser get the title of the document using the DOM method **`document.title`**. Now we have a saved reference to the page title at the time the page is loaded. This variable should be global, so we'll declare it outside any functions.

```
var originalTitle = document.title;
```

4. Next, we'll define a function so we can reuse the script whenever it's needed. Let's call the function something easy to remember, so we know at a glance what it does when we encounter it in our code later. **`showUnreadCount()`** works for me, but you can name it whatever you'd like.

```
var originalTitle = document.title;

function showUnreadCount() {
}
```

EXERCISE 21-2. Continued

5. We need to think about what the function needs to make it useful. This function does something with the unread message count, so its argument is a single number referred to as **unread** in this example.

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
}
```

6. Now let's add the code that runs for this function. We want the document title for the page to display the title of the document plus the count of unread messages. Sounds like a job for concatenation (**+**)! Here we set the **document.title** to be (**=**) whatever string was saved for **originalTitle** plus the number in **showUnreadCount**. As we learned earlier, JavaScript combines a string and a number as though they are both strings.

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + unread;
}
```

7. Let's try out our script before we go too much further. Below where you defined the function and the **originalTitle** variable, enter **showUnreadCount(3);**. Now save the page and reload it in your browser (FIGURE 21-7).

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + unread;
}
showUnreadCount(3);
```

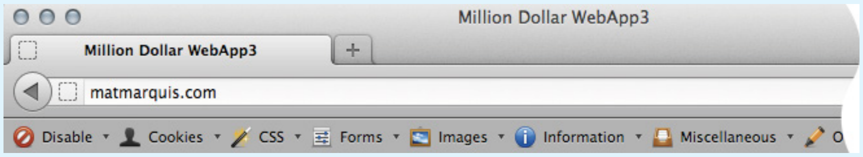


FIGURE 21-7. Our title tag has changed! It's not quite right yet, though.

8. Our script is working, but it's not very easy to read. Fortunately, there's no limit on the number of strings we can combine at once. Here we're adding new strings that wrap the count value and the words "new messages" in parentheses (FIGURE 21-8).

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + "(" + unread + " new messages!)";
}
showUnreadCount(3);
```

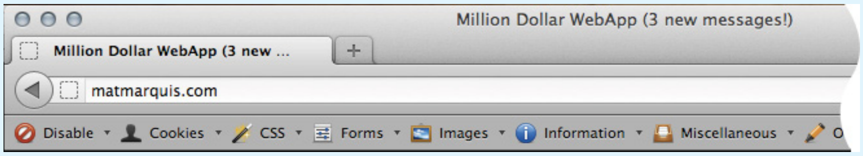

Million Dollar WebApp (3 new messages!)
Million Dollar WebApp (3 new ...
matmarquis.com
Disable
Cookies
CSS
Forms
Images
Information
Miscellaneous

FIGURE 21-8. Much better!

TEST YOURSELF

We covered a lot of new material in this chapter. Here's a chance to test what sunk in. You will find the answers in **Appendix A**.

1. Name one good thing and one bad thing about linking to external *.js* files.

2. Given the following array

   ```
   var myArray = [1, "two", 3, "4"]
   ```

   write what the alert message will say for each of these examples:

   a. `alert( myArray[0] );`

   b. `alert( myArray[0] + myArray[1] );`

   c. `alert( myArray[2] + myArray[3] );`

   d. `alert( myArray[2] - myArray[0] );`

3. What will each of these alert messages say?

   a.
   ```
   var foo = 5;
   foo += 5;
   alert( foo );
   ```

   b.
   ```
   i = 5;
   i++;
   alert( i );
   ```

   c.
   ```
   var foo = 2;
   alert( foo + " " + "remaining");
   ```

   d.
   ```
   var foo = "Mat";
   var bar = "Jennifer";
   if( foo.length > bar.length ) {
     alert( foo  + " is longer." );
   } else {
     alert( bar + " is longer." );
   }
   ```

   e. `alert( 10 === "10" );`

4. Describe what this does:

   ```
   for( var i = 0; i < items.length;  i++ ) {  }
   ```

5. What is the potential problem with globally scoped variables?

6. Match each event handler with its trigger.

a. `onload`	1. The user finishes a form and hits the submit button.
b. `onchange`	2. The page finishes loading.
c. `onfocus`	3. The pointer hovers over a link.
d. `onmouseover`	4. A text-entry field is selected and ready for typing.
e. `onsubmit`	5. A user changes her name in a form field.

CHAPTER

22

USING JAVASCRIPT

AND THE DOCUMENT OBJECT MODEL

by Mat Marquis

Now that you have a sense for the language of JavaScript, let's look at some of the ways we can put it to use in modern web design. First, we'll explore DOM scripting, which allows us to manipulate the elements, attributes, and text on a page. I'll introduce you to some ready-made JavaScript and DOM scripting resources, so you don't have to go it alone. You'll learn about polyfills, which provide older browsers with modern features and normalize functionality. I'll also introduce you to JavaScript libraries that make developers' lives easier with collections of polyfills and shortcuts for common tasks.

IN THIS CHAPTER

Using the DOM to access and change elements, attributes, and contents

Using polyfills to make browser versions work consistently

Using JavaScript libraries

A brief introduction to Ajax

MEET THE DOM

You've seen references to the Document Object Model (DOM for short) several times throughout this book, but now is the time to give it the attention it deserves. The DOM gives us a way to access and manipulate the contents of a document. We commonly use it for HTML, but the DOM can be used with any XML language as well. And although we're focusing on its relationship with JavaScript, it's worth noting that the DOM can be accessed by other languages too, such as PHP, Ruby, C++, and more. Although DOM Level 1 was released by the W3C in 1998, it was nearly five years later that DOM scripting began to gain steam.

The DOM gives us a way to access and manipulate the contents of a document.

The DOM is a programming interface (an API) for HTML and XML pages. It provides a structured map of the document, as well as a set of methods to interface with the elements contained therein. Effectively, it translates our markup into a format that JavaScript (and other languages) can understand. It sounds pretty dry, I know, but the basic gist is that the DOM serves as a map to all the elements on a page and lets us *do* things with them. We can use it to find elements by their names or attributes, and then add, modify, or delete elements and their content.

Without the DOM, JavaScript wouldn't have any sense of a document's contents—and by that, I mean the *entirety* of the document's contents. Everything from the page's **doctype** to each individual letter in the text can be accessed via the DOM and manipulated with JavaScript.

The Node Tree

A simple way to think of the DOM is in terms of the document tree as diagrammed in FIGURE 22-1. You saw documents diagrammed in this way when you were learning about CSS selectors.

```
<!DOCTYPE html>
<html>
<head>
  <title>Document title</title>
  <meta charset="utf-8">
</head>
<body>
  <div>
    <h1>Heading</h1>
    <p>Paragraph text with a <a href="foo.html">link</a> here.</p>
  </div>
  <div>
    <p>More text here.</p>
  </div>
</body>
</html>
```

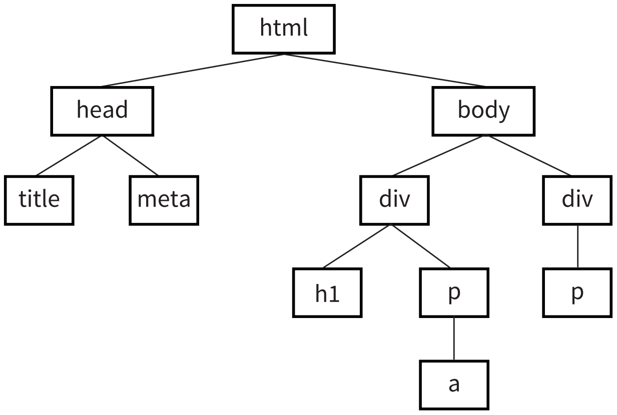


FIGURE 22-1. A simple document.

Each element within the page is referred to as a node. If you think of the DOM as a tree, each node is an individual branch that can contain further branches. But the DOM allows deeper access to the content than CSS because it treats the actual content as a node as well. FIGURE 22-2 shows the structure of the first **p** element. The element, its attributes, and its contents are all nodes in the DOM's node tree.

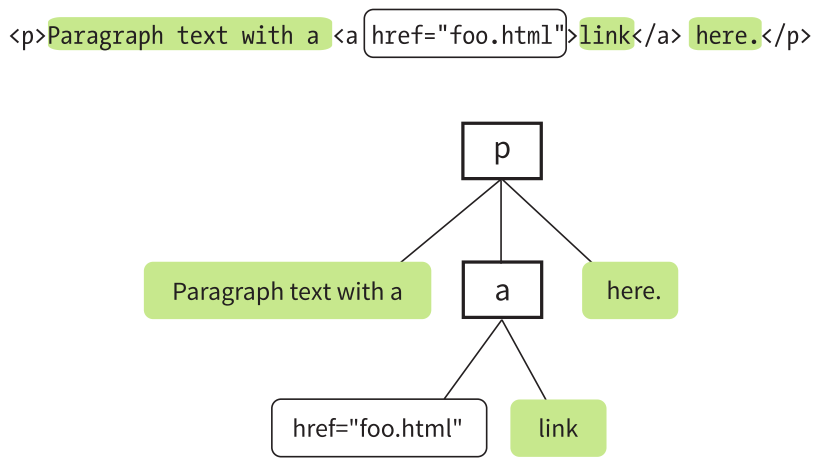

<p>Paragraph text with a <a href="foo.html">link</a> here.</p>
p
Paragraph text with a
a
here.
href="foo.html"
link

FIGURE 22-2. The nodes within the first **p** element in our sample document.

AT A GLANCE

The DOM is a collection of nodes:

- Element nodes
- Attribute nodes
- Text nodes

The DOM also provides a standardized set of methods and functions through which JavaScript can interact with the elements on our page. Most DOM scripting involves reading from and writing to the document.

There are several ways to use the DOM to find what you want in a document. Let's go over some of the specific methods we can use for accessing objects defined by the DOM (we JS folks call this "crawling the DOM" or "traversing the DOM"), as well as some of the methods for manipulating those elements.

Accessing DOM Nodes

The **document** object in the DOM identifies the page itself, and more often than not will serve as the starting point for our DOM crawling. The **document** object comes with a number of standard properties and methods for accessing collections of elements. This is reminiscent of the **length** property we learned about in **Chapter 21, Introduction to JavaScript**. Just as **length** is a standard property of all arrays, the **document** object comes with a number of built-in properties containing information about the document. We then wind our way to the element we're after by chaining those properties and methods together, separated by periods, to form a sort of route through the document.

To give you a general idea of what I mean, the statement in this example says to look on the page (**document**), find the element that has the **id** value "beginner", find the HTML content within that element (**innerHTML**), and save those contents to a variable (**foo**):

```
var foo = document.getElementById("beginner").innerHTML;
```

Because the chains tend to get long, it is also common to see each property or method broken onto its own line to make it easier to read at a glance.

Remember, whitespace in JavaScript is ignored, so this has no effect on how the statement is parsed.

```
var foo = document
  .getElementById("beginner")
    .innerHTML;
```

There are several methods for accessing nodes in the document.

By element name

getElementsByTagName()

We can access individual elements by the tags themselves, using **document.getElementsByTagName()**. This method retrieves any element or elements you specify as an argument.

NOTE

nodeLists are living collections. If you manipulate the document in a nodeList loop—for example, looping through all paragraphs and appending new ones along the way—you can end up in an infinite loop. Good times!

For example, **document.getElementsByTagName("p")** returns every paragraph on the page, wrapped in something called a collection or nodeList, in the order they appear in the document from top to bottom. nodeLists behave much like arrays. To access specific paragraphs in the nodeList, we reference them by their index, just like an array.

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

Based on this variable statement, **paragraphs[0]** is a reference to the first paragraph in the document, **paragraphs[1]** refers to the second, and so on. If we had to access each element in the nodeList separately, one at a time…well, it's a good thing we learned about looping through arrays earlier. Loops work the exact same way with a nodeList.

```
var paragraphs = document.getElementsByTagName("p");
for( var i = 0; i < paragraphs.length; i++ ) {
  // do something
}
```

Now we can access each paragraph on the page individually by referencing **paragraphs[i]** inside the loop, just as with an array, but with elements on the page instead of values.

By id attribute value

getElementById()

This method returns a single element based on that element's ID (the value of its **id** attribute), which we provide to the method as an argument. For example, to access this particular image

```
<img src="photo.jpg" alt="" id="lead-photo">
```

we include the **id** value as an argument for the **getElementById()** method:

```
var photo = document.getElementById("lead-photo");
```

By class attribute value

getElementsByClassName()

Just as it says on the tin, this allows you to access nodes in the document based on the value of a **class** attribute. This statement assigns any element with a **class** value of "column-a" to the variable **firstColumn** so it can be accessed easily from within a script:

```
var firstColumn = document.getElementsByClassName("column-a");
```

Like **getElementsByTagName()**, this returns a nodeList that we can reference by index or loop through one at a time.

WARNING

This is a relatively new method for accessing DOM nodes. Although **getElementsByClassName()** *is available in the current versions of modern browsers, it will not work in IE8 or below.*

By selector

querySelectorAll()

querySelectorAll() allows you to access nodes of the DOM based on a CSS-style selector. The syntax of the arguments in the following examples should look familiar to you. It can be as simple as accessing the child elements of a specific element:

```
var sidebarPara = document.querySelectorAll(".sidebar p");
```

or as complex as selecting an element based on an attribute:

```
var textInput = document.querySelectorAll("input[type='text']");
```

querySelectorAll() returns a nodeList, like **getElementsByTagName()** and **getElementsByClassName()**, even if the selector matches only a single element.

WARNING

querySelectorAll() *isn't supported in IE7 or below.*

Accessing an attribute value

getAttribute()

As I mentioned earlier, elements aren't the only thing you can access with the DOM. To get the value of an attribute attached to an element node, we call **getAttribute()** with a single argument: the attribute name. Let's assume we have an image, *stratocaster.jpg*, marked up like this:

```
<img src="stratocaster.jpg" alt="electric guitar" id="lead-image">
```

In the following example, we access that specific image (**getElementbyId()**) and save a reference to it in a variable ("bigImage"). At that point, we could access any of the element's attributes (**alt**, **src**, or **id**) by specifying it as an argument in the **getAttribute()** method. In the example, we get the value of the **src** attribute and use it as the content in an alert message. (I'm not sure *why* we would ever do that, but it does demonstrate the method.)

```
var bigImage = document.getElementById("lead-image");
alert( bigImage.getAttribute("src") ); // Alerts "stratocaster.jpg".
```

Manipulating Nodes

Once we've accessed a node by using one of the methods discussed previously, the DOM gives us several built-in methods for manipulating those elements, their attributes, and their contents.

`setAttribute()`

To continue with the previous example, we saw how we *get* the attribute value, but what if we wanted to *set* the value of that **`src`** attribute to a new pathname altogether? Use **`setAttribute()`**! This method requires two arguments: the attribute to be changed and the new value for that attribute.

In this example, we use a bit of JavaScript to swap out the image by changing the value of the **`src`** attribute:

```
var bigImage = document.getElementById("lead-image");
bigImage.setAttribute("src", "lespaul.jpg");
```

Just think of all the things you could do with a document by changing the values of attributes. Here we swapped out an image, but we could use this same method to make a number of changes throughout our document:

- Update the **`checked`** attributes of checkboxes and radio buttons based on user interaction elsewhere on the page.
- Find the **`link`** element for our *.css* file and point the **`href`** value to a different style sheet, changing all the page's styles.
- Update a **`title`** attribute with information on an element's state ("this element is currently selected," for example).

`innerHTML`

`innerHTML` gives us a simple method for accessing and changing the text and markup inside an element. It behaves differently from the methods we've covered so far. Let's say we need a quick way of adding a paragraph of text to the first element on our page with a class of **`intro`**:

```
var introDiv = document.getElementsByClassName("intro");
introDiv[0].innerHTML = "<p>This is our intro text</p>";
```

The second statement here adds the content of the string to **`introDiv`** (an element with the **`class`** value "intro") as a *real live element* because **`innerHTML`** tells JavaScript to parse the strings "<p>" and "</p>" as markup.

`style`

The DOM also allows you to add, modify, or remove a CSS style from an element by using the **`style`** property. It works similarly to applying a style with the inline **`style`** attribute. The individual CSS properties are available as properties of the **`style`** property. I bet you can figure out what these statements are doing by using your new CSS and DOM know-how:

```
document.getElementById("intro").style.color = "#fff";

document.getElementById("intro").style.backgroundColor = "#f58220";
    //orange
```

In JavaScript and the DOM, property names that are hyphenated in CSS (such as **`background-color`** and **`border-top-width`**) become camel case (**`backgroundColor`** and **`borderTopWidth`**, respectively) so the "**`-`**" character isn't mistaken for an operator.

In the examples you've just seen, the **`style`** property is used to set the styles for the node. It can also be used to get a style value for use elsewhere in the script. This statement gets the background color of the **`#intro`** element and assigns it to the **`brandColor`** variable:

```
var brandColor = document.getElementById("intro").style.backgroundColor;
```

Adding and Removing Elements

So far, we've seen examples of getting and setting nodes in the existing document. The DOM also allows developers to change the document structure itself by adding and removing nodes on the fly. We'll start out by creating new nodes, which is fairly straightforward, and then we'll see how we add the nodes we've created to the page. The methods shown here are more surgical and precise than adding content with **`innerHTML`**. While we're at it, we'll remove nodes, too.

createElement()

To create a new element, use the aptly named **`createElement()`** method. This function accepts a single argument: the element to be created. Using this method is a little counterintuitive at first because the new element doesn't appear on the page right away. Once we create an element in this way, that new element remains floating in the JavaScript ether until we add it to the document. Think of it as creating a *reference* to a new element that lives purely in memory—something that we can manipulate in JavaScript as we see fit, and then add to the page once we're ready:

```
var newDiv = document.createElement("div");
```

createTextNode()

If we want to enter text into either an element we've created or an existing element on the page, we can call the **`createTextNode()`** method. To use it, provide a string of text as an argument, and the method creates a DOM-friendly version of that text, ready for inclusion on the page. Like **`createElement()`**, this creates a reference to the new text node that we can store in a variable and add to the page when the time comes:

```
var ourText = document.createTextNode("This is our text.");
```

appendChild()

So we've created a new element and a new string of text, but how do we make them part of the document? Enter the **appendChild()** method. This method takes a single argument: the node you want to add to the DOM. You call it on the existing element that will be its *parent* in the document structure. Time for an example.

Here we have a simple **div** on the page with the **id** "our-div":

```
<div id="our-div"></div>
```

Let's say we want to add a paragraph to **#our-div** that contains the text "Hello, world!" We start by creating the **p** element (**document.createElement()**) as well as a text node for the content that will go inside it (**createTextNode()**):

```
var ourDiv = document.getElementById("our-div");
var newParagraph = document.createElement("p");
var copy = document.createTextNode("Hello, world!");
```

Now we have our element and some text, and we can use **appendChild()** to put the pieces together:

```
newParagraph.appendChild( copy );
ourDiv.appendChild( newParagraph );
```

The first statement appends **copy** (that's our "Hello, world!" text node) to the new paragraph we created (**newParagraph**), so now that element has some content. The second line appends the **newParagraph** to the original **div** (**ourDiv**). Now **ourDiv** isn't sitting there all empty in the DOM, and it will display on the page with the content "Hello, world!"

You should be getting the idea of how it works. How about a couple more?

insertBefore()

The **insertBefore()** method, as you might guess, inserts an element before another element. It takes two arguments: the first is the node that gets inserted, and the second is the element it gets inserted in front of. You also need to know the parent to which the element will be added.

So, for example, to insert a new heading before the paragraph in this markup

```
<div id="our-div">
  <p id="our-paragraph">Our paragraph text</p>
</div>
```

we start by assigning variable names to the **div** and the **p** it contains, and then create the **h1** element and its text node and put them together, just as we saw in the last example:

```
var ourDiv = document.getElementById("our-div");
var para = document.getElementById("our-paragraph");

var newHeading = document.createElement("h1");
var headingText = document.createTextNode("A new heading");
newHeading.appendChild( headingText );
// Add our new text node to the new heading
```

Finally, in the last statement shown here, the **insertBefore()** method places the **newHeading h1** element before the **para** element inside **ourDiv**.

```
ourDiv.insertBefore( newHeading, para );
```

replaceChild()

The **replaceChild()** method replaces one node with another and takes two arguments. The first argument is the new child (i.e., the node you want to end up with). The second is the node that gets replaced by the first. As with **insertBefore()**, you also need to identify the parent element in which the swap happens. For the sake of simplicity, let's say we start with the following markup:

```
<div id="our-div">
  <div id="swap-me"></div>
</div>
```

And we want to replace the **div** with the **id** "swap-me" with an image. We start by creating a new **img** element and setting the **src** attribute to the pathname to the image file. In the final statement, we use **replaceChild()** to put **newImg** in place of **swapMe**.

```
var ourDiv = document.getElementById("our-div");
var swapMe = document.getElementById("swap-me");
var newImg = document.createElement("img");
// Create a new image element

newImg.setAttribute( "src", "path/to/image.jpg" );
// Give the new image a "src" attribute
ourDiv.replaceChild( newImg, swapMe );
```

removeChild()

To paraphrase my mother, "We brought these elements into this world, and we can take them out again." You remove a node or an entire branch from the document tree with the **removeChild()** method. The method takes one argument, which is the node you want to remove. Remember that the DOM thinks in terms of *nodes*, not just elements, so the child of an element may be the text (node) it contains, not just other elements.

Like **appendChild()**, the **removeChild()** method is always called on the parent element of the element to be removed (hence, "remove *child*"). That means we'll need a reference to both the parent node and the node we're looking to remove. Let's assume the following markup pattern:

```
<div id="parent">
  <div id="remove-me">
    <p>Pssh, I never liked it here anyway.</p>
  </div>
</div>
```

Our script would look something like this:

```
var parentDiv = document.getElementById("parent");
var removeMe = document.getElementById("remove-me");
```

```
parentDiv.removeChild( removeMe );
// Removes the div with the id "remove-me" from the page.
```

For Further Reading

That should give you a good idea of what DOM scripting is all about. Of course, I've just barely scratched the surface of what can be done with the DOM, but if you'd like to learn more, definitely check out the book *DOM Scripting: Web Design with JavaScript and the Document Object Model, Second Edition*, by Jeremy Keith and Jeffrey Sambells (Friends of Ed).

POLYFILLS

You've gotten familiar with a lot of new technologies in this book so far: new HTML5 elements, new ways of doing things with CSS3, using JavaScript to manipulate the DOM, and more. In a perfect world, all browsers would be in lockstep, keeping up with the cutting-edge technologies and getting the established ones right along the way (see the sidebar **"The Browser Wars"**). In that perfect world, browsers that couldn't keep up (I'm looking at you, IE8) would just vanish completely. Sadly, that is not the world we live in, and browser inadequacies remain the thorn in every developer's side.

I'll be the first to admit that I enjoy a good wheel reinvention. It's a great way to learn, for one thing. For another, it's the reason our cars aren't rolling around on roundish rocks and sections of tree trunk. But when it comes to dealing with every strange browser quirk out there, we don't have to start from scratch. Tons of people smarter than I am have run into these issues before, and have already found clever ways to work around them and fix the parts of JavaScript and the DOM where some browsers may fall short. We can use JavaScript to fix JavaScript.

Polyfill is a term coined by Remy Sharp to describe a JavaScript "shim" that normalizes differing behavior from browser to browser (*remysharp.com/2010/10/08/what-is-a-polyfill*). Or, as Paul Irish put it, a polyfill is

> *A shim that mimics a future API providing fallback functionality to older browsers.*

There's a lot of time travel going on in that quote, but basically what he's saying is that we're making something new work in browsers that don't natively support it—whether that's brand-new technology like detecting a user's physical location or fixing something that one of the browsers just plain got wrong.

There are tons of polyfills out there targeted to specific tasks, such as making old browsers recognize new HTML5 elements or CSS3 selectors, and new ones are popping up all the time as new problems arise. I'm going to fill you in on the most commonly used polyfills in the modern developer's toolbox as

The Browser Wars

JavaScript came about during a dark and lawless time, before the web standards movement, when all the major players in the browser world were—for want of a better term—winging it. It likely won't come as a major surprise to anyone that Netscape and Microsoft implemented radically different versions of the DOM, with the prevailing sentiment being "may the best browser win."

I'll spare you the gory details of the Battle for JavaScript Hill, but the two competing implementations were so different that they were both largely useless, unless you wanted to either maintain two separate code bases or add a "best viewed in Internet Explorer/Netscape" warning label to your sites.

Enter the web standards movement! During this cutthroat time, the W3C was putting together the foundations for the modern-day standardized DOM that we've all come to know and love. Fortunately for us, Netscape and Microsoft got on board with the standards movement. The standardized DOM is supported all the way back to Internet Explorer 5 and Netscape Navigator 6. Unfortunately, Internet Explorer's advancements in this area stagnated for quite some time following IE6. As a result, older versions of IE have a few significant differences from the modern-day DOM. Fortunately with Internet Explorer 9 and later, they're catching right back up.

The trouble is, your project likely still needs to support those users with older versions of IE. It's a pain, but we're up for it. We have an amazing set of tools at our disposal, such as polyfills and JavaScript libraries full of helper functions, that normalize the strange little quirks we're apt to encounter from browser to browser.

of the release of this book. You may find that new ones are necessary by the time you hit the web design trenches. You may also find that some of these techniques aren't needed for the browsers you need to support.

HTML5 shim (or shiv)

You may remember seeing this one back in **Chapter 5, Marking Up Text,** but let's give it a little more attention now that you have some JavaScript under your belt.

NOTE

If you don't need to support IE8 and earlier, you don't need an HTML5 shim.

An HTML5 shim/shiv is used to enable Internet Explorer 8 and earlier to recognize and style newer HTML5 elements such as **`article`**, **`section`**, and **`nav`**.

There are several variations on the HTML5 shim/shiv, but they all work in much the same way: crawl the DOM looking for elements that IE doesn't recognize, and then immediately replace them with the same element so they are visible to IE in the DOM. Now any styles we write against those elements work as expected. Sjoerd Visscher originally discovered this technique, and many, many variations of these scripts exist now. Remy Sharp's version is the one in widest use today.

The shim must be referenced in the **`head`** of the document, in order to "tell" Internet Explorer about these new elements before it finishes rendering the page. The script is referenced inside an IE-specific conditional comment and runs only if the browser is less than (**`lt`**) IE9—in other words, versions 8 and earlier:

```
<!--[if lt IE 9]>
     <script src="html5shim.js"></script>
<![endif]-->
```

The major caveat here is that older versions of Internet Explorer that have JavaScript disabled or unavailable will receive unstyled elements. To learn more about HTML5 shim/shiv, try these resources:

- The Wikipedia entry for HTML Shiv (*en.wikipedia.org/wiki/HTML5_Shiv*)
- Remy Sharp's original post (*remysharp.com/2009/01/07/html5-enabling-script*)

Selectivizr

Selectivizr (created by Keith Clark) allows Internet Explorer 6–8 to understand complex CSS3 selectors such as **`:nth-child`** and **`::first-letter`**. It uses JavaScript to fetch and parse the contents of your style sheet and patch holes where the browser's native CSS parser falls short.

NOTE

If you don't need to support IE8 and earlier, you don't need Selectivizr.

Selectivizr must be used with a JavaScript library (I talk about them in the next section). The link to the script goes in an IE conditional comment after the link to the library *.js* file, like so:

```
<script type="text/javascript" src="[JS library]"></script>
<!--[if (gte IE 6)&(lte IE 8)]>
  <script type="text/javascript" src="selectivizr.js"></script>
  <noscript><link rel="stylesheet" href="[fallback css]" /></noscript>
<![endif]-->
```

Because we're forgoing the native CSS parser here, we may see a slight performance hit in applicable browsers. See the Selectivizr site (*selectivizr.com*) for more information.

Picturefill (A Responsive Image Polyfill)

Picturefill enables support for the **picture** element, **srcset** and **sizes** attributes, and features related to delivering images based on viewport size and resolution (also known as responsive images, as discussed in **Chapter 7, Adding Images**). It was created by Scott Jehl of Filament Group and is maintained by the Picturefill group.

To use Picturefill, download the script and add it to the **head** of the document. The first script creates a **picture** element for browsers that don't recognize it. The second script calls the Picturefill script itself and the **async** attribute tells the browser it can load Picturefill asynchronously—that is, without waiting for the script to finish before loading the rest of the document.

```
<head>
  <script>
    // Picture element HTML5 shiv
    document.createElement( "picture" );
  </script>
  <script src="picturefill.js" async></script>
</head>
```

On the downside, browsers without JavaScript that also do not support the **picture** element will see only alt-text for the image. Download Picturefill and get information about its use at *scottjehl.github.io/picturefill/*.

JAVASCRIPT LIBRARIES

Continuing on the "you don't have to write everything from scratch yourself" theme, it's time to take on JavaScript libraries. A JavaScript library is a collection of prewritten functions and methods that you can use in your scripts to accomplish common tasks or simplify complex ones.

There are many JS libraries out there. Some are large frameworks that include all of the most common polyfills, shortcuts, and widgets you'd ever need to build full-blown Ajax web applications (see the sidebar **"What Is Ajax?"**). Some are targeted at specific tasks, such as handling forms, animation, charts, or math functions. For seasoned JavaScript-writing pros, starting with a library is an awesome time-saver. And for folks like you who are just getting started, a library can handle tasks that might be beyond the reach of your own skills.

What Is Ajax?

Ajax (sometimes written AJAX) stands for Asynchronous JavaScript And XML. The "XML" part isn't that important—you don't have to use XML to use Ajax (more on that in a moment). The "asynchronous" part is what matters.

Traditionally, when a user interacted with a web page in a way that required data to be delivered from the server, everything had to stop and wait for the data, and the whole page needed to reload when it was available. This made for a not especially smooth user experience.

But with Ajax, because the page can get data from the server in the background, you can make updates to the page based on user interaction smoothly and in real time without the page needing to be reloaded. This makes web applications feel more like "real" applications.

You see this on a number of modern websites, although sometimes it's subtle. On Twitter, for example, scrolling to the bottom of a page loads in a set of new tweets. Those aren't hardcoded in the page's markup; they're loaded dynamically as needed. Google's image search uses a similar approach. When you reach the bottom of the current page, you're presented with a button that allows you to load more, but you never navigate away from the current page.

The term "Ajax" was first coined by Jesse James Garrett in an article "Ajax: A New Approach to Web Applications." Ajax is not a single technology, but rather a combination of HTML, CSS, the DOM, and JavaScript, including the **`XMLHttpRequest`** object, which allows data to be transferred asynchronously. Ajax may use XML for data, but it has become more common to use JSON (JavaScript Object Notation), a JavaScript-based and human-readable format, for data exchange.

Writing web applications with Ajax isn't the type of thing you would do right out of the gate, but many of the JavaScript libraries discussed in this chapter have built-in Ajax helpers and methods that let you get started with significantly less effort.

The disadvantage of libraries is that because they generally contain all of their functionality in one big *.js* file, you may end up forcing your users to download a lot of code that never gets used. But the library authors are aware of this and have made many of their libraries modular, and they continue to make efforts to optimize their code. In some cases, it's also possible to customize the script and use just the parts you need.

jQuery and Other Libraries

As of this writing, the overwhelmingly dominant JavaScript library is jQuery (*jquery.com*). Chances are, if you use a library, it will be that one (or at least that one first). Written in 2005 by John Resig, jQuery has found its way into over two-thirds of all websites. Furthermore, if a site uses a library at all, there is a 97% chance that it's jQuery.

It is free, it's open source, and it employs a syntax that makes it easy to use if you are already handy with CSS, JavaScript, and the DOM. You can supplement jQuery with the jQuery UI library, which adds cool interface elements such as calendar widgets, drag-and-drop functionality, expanding accordion lists, and simple animation effects. jQuery Mobile is another jQuery-based library that provides UI elements and polyfills designed to account for the variety of mobile browsers and their notorious quirks.

Of course, jQuery isn't the only library in town. Others include MooTools (*mootools.net*), Dojo (*dojotoolkit.org*), and Prototype (*prototypejs.org*). As for smaller JS libraries that handle specialized functions, because they are being created and made obsolete all the time, I recommend doing a web search for

"JavaScript libraries for _______________" and see what is available. Some library categories include the following:

- Forms
- Animation
- Image carousels
- Games
- Information graphics
- Image and 3-D effects for the **`canvas`** element
- String and math functions
- Database handling
- Touch gestures

How to Use jQuery

It's easy to implement any of the libraries I just listed. All you do is download the JavaScript (*.js*) file, put it on your server, point to it in your **`script`** tag, and you're good to go. It's the *.js* file that does all the heavy lifting, providing prewritten functions and syntax shortcuts. Once you've included it, you can write your own scripts that leverage the features built into the framework. Of course, what you actually do with it is the interesting part (and largely beyond the scope of this chapter, unfortunately).

As a member of the jQuery Mobile team, I have a pretty obvious bias here, so we're going to stick with jQuery in the upcoming examples. Not only is it the most popular library anyway, but they said they'd give me a dollar every time I say "jQuery."

Download the jQuery .js file

To get started with jQuery (*cha-ching*), go to *jQuery.com* and hit the big Download button to get your own copy of *jquery.js*. You have a choice between a production version that has all the extra whitespace removed for a smaller file size, or a development version that is easier to read but nearly eight times larger in file size. The production version should be just fine if you are not going to edit it yourself.

Copy the code, paste it into a new plain-text document, and save it with the same filename that you see in the address bar in the browser window. As of this writing, the latest version of jQuery is 3.2.1, and the filename of the production version is *jquery-3.2.1.min.js* (the *min* stands for "minimized"). Put the file in the directory with the other files for your site. Some developers keep their scripts in a *js* directory for the sake of organization, or they may simply keep them in the root directory for the site. Wherever you decide put it, be sure to note the pathname to the file because you'll need it in the markup.

Add it to your document

Include the jQuery script the same way you'd include any other script in the document, with a **script** element:

```
<script src="pathtoyourjs/jquery-3.2.1.min.js"></script>
```

And that's pretty much it. There is an alternative worth mentioning, however. If you don't want to host the file yourself, you can point to one of the publicly hosted versions and use it that way. One advantage to this method is that it gets cached by the browser, so there's a chance some of your users' browsers already have a copy of it. The jQuery Download page lists a few options, including the following link to the code on Google's server. Simply copy this code exactly as you see it here, paste it into the **head** of the document or before the **</body>** tag, and you've got yourself some jQuery!

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/ →
jquery.min.js"></script>
```

Get "ready"

You don't want to go firing scripts before the document and the DOM are ready for them, do you? Well, jQuery has a statement known as the ready event that checks the document and waits until it's ready to be manipulated. Not all scripts require this (for example, if you were only firing a browser alert), but if you are doing anything with the DOM, it is a good idea to start by setting the stage for your scripts by including this function in your custom **script** or *.js* file:

```
<script src="pathtoyourjs/jquery-3.2.1.min.js"></script>

<script>
$(document).ready(function(){

   // Your code here

});
</script>
```

Scripting with jQuery

Once you're set up, you can begin writing your own scripts using jQuery. The shortcuts jQuery offers break down into two general categories:

- A giant set of built-in feature detection scripts and polyfills
- A shorter, more intuitive syntax for targeting elements (jQuery's selector engine)

You should have a decent sense of what the polyfills do after making your way through that last section, so let's take a look at what the selector engine does for you.

One of the things that jQuery simplifies is moving around through the DOM because you can use the selector syntax that you learned for CSS. Here is an example of getting an element by its **`id`** value *without* a library:

```
var paragraph = document.getElementById( "status" );
```

The statement finds the element with the ID "status" and saves a reference to the element in a variable (**`paragraph`**). That's a lot of characters for a simple task. You can probably imagine how things get a little verbose when you're accessing lots of elements on the page. Now that we have jQuery in play, however, we can use this shorthand:

```
var paragraph = $("#status");
```

That's right—that's the **`id`** selector you know and love from writing CSS. And it doesn't just stop there. *Any* selector you'd use in CSS will work within that special helper function.

You want to find everything with a class of **`header`**? Use **`$(".header");`**.

By the element's name? Sure: **`$("div");`**.

Every subhead in your sidebar? Easy-peasy: **`$("#sidebar .sub");`**.

You can even target elements based on the value of attributes: **`$("[href='http://google.com']");`**.

But it doesn't stop with selectors. We can use a huge number of helper functions built into jQuery and libraries like it to crawl the DOM like so many, uh, Spider-men. Spider-persons. Web-slingers.

jQuery also allows us to chain objects together in a way that can target things even CSS can't (an element's parent element, for example). Let's say we have a paragraph and we want to add a **`class`** to that paragraph's parent element. We don't necessarily know what that parent element will be, so we're unable to target the parent element directly. In jQuery we can use the **`parent()`** object to get to it:

```
$("p.error").parent().addClass("error-dialog");
```

Another major benefit is that this is highly readable at a glance: "find any paragraph(s) with the class 'error' and add the class 'error-dialog' to their parent(s)."

But What If I Don't Know How to Write Scripts?

It takes time to learn JavaScript, and it may be a while before you can write scripts on your own. But not to worry. If you do a web search for what you need (for example, "jQuery image carousel" or "jQuery accordion list"), there's a very good chance you will find lots of scripts that people have created and shared, complete with documentation on how to use them. Because jQuery uses a selector syntax very similar to CSS, it makes it easier to customize jQuery scripts for use with your own markup.

BIG FINISH

In all of two chapters, we've gone from learning the very basics of variables to manipulating the DOM to leveraging a JavaScript library. Even with all we've covered here, we've just barely begun to cover all the things JavaScript can do.

The next time you're looking at a website and it does something cool, view the source in your browser and have a look around for the JavaScript. You can learn a lot from reading and even taking apart someone else's code. And remember, there's nothing you can break with JavaScript that can't be undone with a few strokes of the Delete key.

Better still, JavaScript comes with an entire community of passionate developers who are eager to learn and just as eager to teach. Seek out like-minded developers and share the things you've learned along the way. If you're stuck on a tricky problem, don't hesitate to seek out help and ask questions. It's rare that you'll encounter a problem that nobody else has, and the open source developer community is always excited to share the things they've learned. That's why you've had to put up with me for two chapters, as a matter of fact.

TEST YOURSELF

Just a few questions for those of you playing along at home. If you need some help, peek in **Appendix A** for the answers.

1. Ajax is a combination of what technologies?

2. What does this do?

   ```
   document.getElementById("main")
   ```

3. What does this do?

   ```
   document.getElementById("main").getElementsByTagName("section");
   ```

4. What does this do?

```
document.body.style.backgroundColor = "papayawhip"
```

5. What does this do? (This one is a little tricky because it nests functions, but you should be able to piece it together.)

```
document
  .getElementById("main")
    .appendChild(
      document.createElement("p")
        .appendChild(
          documentCreateTextNode("Hey, I'm walking here!")
        )
     );
```

6. What is the benefit of using a JavaScript library such as jQuery?

   a. Access to a packaged collection of polyfills

   b. Possibly shorter syntax

   c. Simplified Ajax support

   d. All of the above