IN THIS CHAPTER

» Getting a feel for programming in general, and JavaScript in particular

» Checking out the tools you need to get coding

» Adding comments to your JavaScript code

» Storing your code in a separate JavaScript file

Chapter 1 JavaScript: The Big Picture

In this chapter, you explore some useful JavaScript basics. Don't worry if you've never programmed before. I take you through everything you need to know, step-by-step, nice and easy. As you're about to find out, it really is fun to program.

Adding JavaScript Code to a Web Page

Okay, it's time to roll up your sleeves, crack your knuckles, and start coding. This section describes the standard procedure for constructing and testing a script and takes you through a couple of examples.

The <script> tag

The basic container for a script is, naturally enough, the HTML `<script>` tag and its associated `</script>` end tag:

```
<script>
    JavaScript statements go here
</script>
```

Where do you put the <script> tag?

With certain exceptions, it doesn't matter a great deal where you put your `<script>` tag. Some people place the tag between the page's `</head>` and `<body>` tags. The HTML standard recommends placing the `<script>` tag within the page header (that is, between `<head>` and `</head>`), so that's the style I use in this book:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Where do you put the script tag?
  </title>
        <script>
            JavaScript statements go here
        </script>
    </head>
    <body>
    </body>
</html>
```

Here are the exceptions to the put-your-script-anywhere technique:

- If your script is designed to write data to the page, the `<script>` tag must be positioned within the page body (that is, between the `<body>` and `</body>` tags) in the exact position where you want the text to appear.
- If your script refers to an item on the page (such as a form object), the script must be placed *after* that item.
- With many HTML tags, you can add one or more JavaScript statements as attributes directly within the tag.

REMEMBER

It's perfectly acceptable to insert multiple `<script>` tags within a single page, as long as each one has a corresponding `</script>` end tag, and as long as you don't put one `<script>` block within another one.

Example #1: Displaying a message to the user

You’re now ready to construct and try out your first script. This example shows you the simplest of all JavaScript actions: displaying a basic message to the user. The following code shows the script within an HTML file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Displaying a Message to the User
   </title>
        <script>
            alert("Hello JavaScript World!");
        </script>
    </head>
    <body>
    </body>
</html>
```

As shown in here, place the script within the header of a page, save the file, and then open the HTML file within your browser.

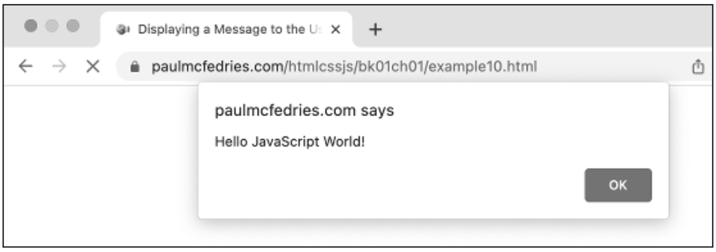
This script consists of just a single line:

```
alert("Hello JavaScript World!");
```

This is called a *statement*, and each statement is designed to perform a single JavaScript task. Your scripts will range from simple programs with just a few statements to huge projects consisting of hundreds of statements.

You may be wondering about the semicolon (`;`) that appears at the end of the statement. Good eye. You use the semicolon to mark the end of each of your JavaScript statements.

In the example, the statement runs the JavaScript `alert()` method, which displays to the user whatever message is enclosed within the parentheses (which could be a welcome message, an announcement of new features on your site, an advertisement for a promotion, and so on). Figure 1-1 shows the message that appears when you open the file.


Displaying a Message to the U
paulmcfedries.com/htmlcssjs/bk01ch01/example10.html
OK

paulmcfedries.com says

Hello JavaScript World!

FIGURE 1-1: This "alert" message appears when you open the HTML file containing the example script.

How did the browser know to run the JavaScript statement? When a browser processes (*parses*, in the vernacular) a page, it basically starts at the beginning of the HTML file and works its way down, one line at a time. If it trips over a `<script>` tag, it knows one or more JavaScript statements are coming, and it automatically executes those statements, in order, as soon as it reads them. The exception is when JavaScript statements are enclosed within a *function*, which I explain in Chapter 5.

WARNING

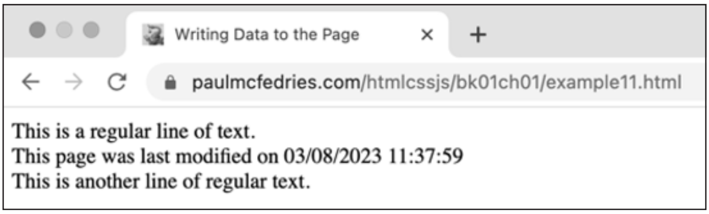
One of the cardinal rules of JavaScript programming is "one statement, one line." That is, each statement must appear on only a single line, and there should be no more than one statement on each line. I said "should" in the second part of the previous sentence because it is possible to put multiple statements on a single line, as long as you separate each statement with a semicolon (`;`). There are rare times when it's necessary to have two or more statements on one line, but you should avoid it for the bulk of your programming because multiple-statement lines are difficult to read and to troubleshoot.

Example #2: Writing text to the page

One of JavaScript's most powerful features is the capability to write text and even HTML tags and CSS rules to the web page on-the-fly. That is, the text (or whatever) gets inserted into the page when a web browser loads the page. What good is that? For one thing, it's ideal for time-sensitive data. For example, you may want to display the date and time that a web page was last modified so that visitors know how old (or new) the page is. Here's some code that shows just such a script:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Writing Data to the Page</title>
    </head>
    <body>
        This is a regular line of text.<br>
        <script>
            document.write("Last modified: " +
  document.lastModified)
        </script>
        <br>This is another line of regular text.
    </body>
</html>
```

Notice how the script appears within the body of the HTML document, which is necessary whenever you want to write data to the page. Figure 1-2 shows the result.



This is a regular line of text.
This page was last modified on 03/08/2023 11:37:59
This is another line of regular text.

FIGURE 1-2: When you open the file, the text displays the date and time the file was last modified.

This script makes use of the `document` *object*, which is a built-in JavaScript construct that refers to whatever HTML file (document) the script resides in (check out Chapter 6 for more about the `document` object). The `document.write()` statement tells the browser to insert whatever is within the parentheses to the web page. The `document.lastModified` portion returns the date and time the file was last changed and saved.

What You Need to Get Started

One of the nicest things about HTML and CSS is that the hurdles you have to leap to get started are not only short but few in number. In fact, you really need only two things, both of which are free: a text editor to enter the text, tags, and properties; and a browser to view the results. (You'll also need a web server to host the finished pages, but the server isn't necessary when you're creating the pages.) Yes, there are high-end text editors and fancy graphics programs, but these fall into the "Bells and Whistles" category; you can create perfectly respectable web pages without them.

The basic requirements for JavaScript programming are exactly the same as for HTML: a text editor and a browser. Again, programs are available to help you write and test your scripts, but you don't need them.

Dealing with Two Exceptional Cases

In this book, I make a couple of JavaScript assumptions related to the people who'll be visiting the pages you post to the web:

- Those people have JavaScript enabled in their web browser.
- Those people are using a relatively up-to-date version of a modern web browser, such as Chrome, Edge, Safari, or Firefox.

These are pretty safe assumptions, but it pays to be a bit paranoid and wonder how you may handle the teensy percentage of people who don't pass one or both tests.

Handling browsers with JavaScript turned off

You don't have to worry about web browsers not being able to handle JavaScript, because all modern browsers have supported JavaScript for a very long time. You may, however, want to worry about people who don't support JavaScript. Although rare, some folks have turned off their browser's JavaScript functionality. Why would someone do such a thing? Many people disable JavaScript because they're concerned about security, they don't want cookies written to their hard drives, and so on.

To handle these iconoclasts, place the `<noscript>` tag within the body of the page:

```
<noscript>
    <p>
        Hey, your browser has JavaScript turned
  off!
    </p>
    <p>
        Okay, cool, perhaps you'll prefer this <a
  href="no-js.html">non-JavaScript version</a> of
  the page.
    </p>
</noscript>
```

If the browser has JavaScript enabled, the browser doesn't display any of the text within the `<noscript>` tag. However, if JavaScript is disabled, the browser displays the text and tags within the `<noscript>` tag to the user.

To test your site with JavaScript turned off, here are the techniques to use in some popular browsers:

- **Chrome (desktop):** Open Settings, click Privacy and Security, click Site Settings, click JavaScript, and then select the Don't Allow Sites to Use JavaScript option.
- **Chrome (Android):** Open Settings, tap Site Settings, tap JavaScript, and then tap the JavaScript switch to off.
- **Edge:** Open Settings, click the Settings menu, click Cookies and Site Permissions, click JavaScript, and then click the Allowed switch to off.
- **Safari (macOS):** Open Settings, click the Advanced tab, select the Show Develop Menu in Menu Bar, and then close Settings. Choose Develop ➪ Disable JavaScript.

- **Safari (iOS or iPadOS):** Open Settings, tap Safari, tap Advanced, and then tap the JavaScript switch to off.
- **Firefox (desktop):** In the Address bar, type **about:config** and press Enter or Return. If Firefox displays a warning page, click Accept the Risk and Continue to display the Advanced Preferences page. In the Search Preference Name box, type **javascript**. In the search results, look for the `javascript.enabled` preference. On the far right of that preference, click the Toggle button to turn the value of the preference from `true` to `false`.

Handling very old browsers

In this book, you learn the version of JavaScript called ECMAScript 2015, also known as ECMAScript 6, or just ES6. Why this version, in particular, and not any of the later versions? Two reasons:

- ES6 has excellent browser support, with more than 98 percent of all current browsers supporting the features released in ES 6. Later versions of JavaScript have less support.
- ES6 has everything you need to add all kinds of useful and fun dynamic features to your pages. Unless you're a professional programmer, the features released in subsequent versions of JavaScript are way beyond what you need.

Okay, so what about that couple of percent of browsers that don't support ES6?

First, know that the number of browsers that choke on ES6 features is getting smaller every day. Sure, it's 2 percent now (about 1.7 percent, actually), but it will be 1 percent in six months, a 0.5 percent in a year, and so on until the numbers just get too small to measure.

Second, the percentage of browsers that don't support ES6 varies by region (it's higher in many countries in Africa, for example) and by environment. Most of the people running browsers that don't fully support ES6 are using Internet Explorer 11, and most of those people are in situations in which they can't upgrade (some corporate environments, for example).

If luck has it that your web pages draw an inordinate share of these older browsers, you may need to eschew the awesomeness of ES6 in favor of the tried-and-true features of ECMAScript 5. To that end, as I introduce each new JavaScript feature, I point out those that arrived with ES6 and let you know if there's a simple fallback or workaround (known as a *polyfill* in the JavaScript trade) if you prefer to use ES5.

Commenting Your Code

A script that consists of just a few lines is usually easy to read and understand. However, your scripts won't stay that simple for long, and these longer and more complex creations will be correspondingly more difficult to read. (This difficulty will be particularly acute if you're looking at the code a few weeks or months after you first coded it.) To help you decipher your code, it's good programming practice to make liberal use of comments throughout the script. A *comment* is text that describes or explains a statement or group of statements. Comments are ignored by the browser, so you can add as many as you deem necessary.

For short, single-line comments, use the double-slash (`//`). Put the `//` at the beginning of the line, and then type your comment after it. Here's an example:

```
// Display the date and time the page was last
  modified
document.write("This page was last modified on " +
  document.lastModified);
```

You can also use `//` comments for two or three lines of text, as long as you start each line with `//`. If you have a comment that stretches beyond that, however, you're better off using multiple-line comments that begin with the `/*` characters and end with the `*/` characters. Here's an example:

```
/*
This script demonstrates JavaScript's ability
to write text to the web page by using the
```

```
document.write() method to display the date and
  time the web page file was last modified.

This script is Copyright Paul McFedries.
*/
```

WARNING

Although it's fine to add quite a few comments when you're just starting out, you don't have to add a comment to everything. If a statement is trivial or its purpose is glaringly obvious, forget the comment and move on.

Moving to External JavaScript Files

Earlier in this chapter, I talk about adding JavaScript code to a web page by inserting the `<script>` and `</script>` tags into the page header (that is, between the `<head>` and `</head>` tags), or sometimes into the page body (that is, between the `<body>` and `</body>` tags). You then write your code between the `<script>` and `</script>` tags.

Putting a script inside the page in this way isn't a problem if the script is relatively short. However, if your script (or scripts) take up dozens or hundreds of lines, your HTML code can look cluttered. Another problem you may run into is needing to use the same code on multiple pages. Sure, you can just copy the code into each page that requires it, but if you make changes down the road, you need to update every page that uses the code.

The solution to both problems is to move the code out of the HTML file and into an external JavaScript file. Moving the code reduces the JavaScript presence in the HTML file to a single line (as you'll learn shortly) and means that you can update the code by editing only the external file.

Here are some things to note about using an external JavaScript file:

- The file must use a plain text format.
- Use the `.js` extension when you name the file.

- Don't use the `<script>` tag within the file. Just enter your statements exactly as you would within an HTML file.
- The rules for when the browser executes statements within an external file are identical to those used for statements within an HTML file. That is, statements outside of functions are executed automatically when the browser comes across your file reference, and statements within a function aren't executed until the function is called. (Not sure what a "function" is? You get the full scoop in Chapter 5.)

To let the browser know that an external JavaScript file exists, add the `src` attribute to the `<script>` tag. For example, if the external file is named `myscripts.js`, your `<script>` tag is set up as follows:

```
<script src="myscripts.js">
```

This example assumes that the `myscripts.js` file is in the same directory as the HTML file. If the file resides in a different directory, adjust the `src` value accordingly. For example, if the `myscripts.js` file is in a subdirectory named `scripts`, you use this:

```
<script src="scripts/myscripts.js">
```

You can even specify a file from another site (presumably your own!) by specifying a full URL as the `src` value:

```
<script src="http://www.host.com/myscripts.js">
```

As an example, the following code shows a one-line external JavaScript file named `footer.js`:

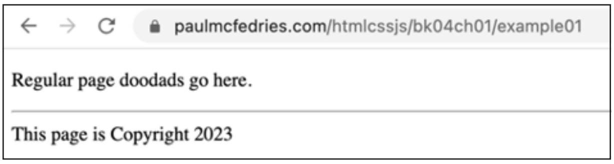
```
document.write("This page is Copyright "
  + new Date().getFullYear());
```

This statement writes the text "Copyright" followed by the current year. (I know: This code looks like some real gobbledygook right now. Don't sweat it, because you'll learn exactly what's going on here when I discuss the JavaScript Date object in Chapter 8.)

The following code shows an HTML file that includes a reference for the external JavaScript file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Using an External JS File</title>
    </head>
    <body>
        <p>
            Regular page doodads go here.
        </p>
        <hr>
        <footer>
            <script src="footer.js">
            </script>
        </footer>
    </body>
</html>
```

When you load the page, the browser runs through the HTML line by line. When it gets to the `<footer>` tag, it notices the external JavaScript file that's referenced by the `<script>` tag. The browser loads that file and then runs the code within the file, which writes the Copyright message to the page, as shown in Figure 1-3.



Regular page doodads go here.

This page is Copyright 2023

FIGURE 1-3: This page uses an external JavaScript file to display a footer message.