CHAPTER OUTLINE

8.1 Introduction
8.2 History of JavaScript
8.3 Hello World Web Page
8.4 Buttons
8.5 Functions
8.6 Variables
8.7 Identifiers
8.8 Assignment Statements and Objects
8.9 Document Object Model
8.10 Forms and How They're Processed: Client-Side Versus Server-Side
8.11 `form` Element
8.12 Controls
8.13 Text Control
8.14 Email Address Generator Web Page
8.15 Accessing a Form's Control Values
8.16 `reset` and `focus` Methods
8.17 Comments and Coding Conventions
8.18 Event-Handler Attributes
8.19 `onchange`, `onmouseover`, `onmouseout`
8.20 Using `noscript` to Accommodate Disabled JavaScript

8.1 Introduction

So far, the HTML/CSS ride has been fairly smooth. It's now time to learn JavaScript, and you may experience some turbulence along the way. But don't worry. If your seat belts are securely fastened and you're prepared to go slowly and think things through, you should be fine. Actually, you should be more than fine because using JavaScript is exhilarating. With JavaScript, you can make your web pages come alive by having them interact with the user. In the previous chapter, you got a small taste of interaction with image sprites, where CSS rules are used to implement rollover effects. But JavaScript is a full-blown programming language, and, as such, anything is possible.

In this chapter, we start with a brief history of the JavaScript language and then quickly move to an example web page where we use JavaScript to display a message when the user clicks a button. In presenting the example, we describe the web page's underlying mechanisms—buttons, functions, and variables.

Next, we describe the basics of the *document object model* (the *DOM*), which will provide a solid foundation for understanding JavaScript constructs that are introduced in this chapter and throughout the rest of the book. The DOM provides hooks that enable various web page objects to do things. For example, web page forms are built with the DOM's underlying framework, and that framework enables forms to process user-entered data when the user clicks a button. In talking about forms and buttons, we describe event handlers, which connect a user's interaction with instructions that tell the browser what to do when the interaction occurs. For example, when a user clicks a button, the browser might display a message. When a user moves the mouse over an image, the browser might swap in a different image.

8.2 History of JavaScript

HTML's first version, designed by Tim Berners-Lee from 1989 to 1991, was fairly static in nature. Except for link jumps with the `a` element, web pages simply displayed content, and the content was fixed. In 1995, the dominant browser manufacturer was Netscape, and one of its employees, Brendan Eich, thought that it would be useful to add dynamic functionality to web pages. So he designed the JavaScript programming language, which adds dynamic functionality to web pages when used in conjunction with HTML. For example, JavaScript provides the ability to update a web page's content when an event occurs, such as when a user clicks a button. It also provides the ability to retrieve a user's input and process that input.

It took Eich only 10 days in May 1995 to implement the JavaScript programming language—a truly remarkable feat. Marc Andreessen, one of Netscape's founders, originally named the new language Mocha and then LiveScript. But for marketing purposes, Andreessen really wanted the name JavaScript. At the time, the software industry was excited about the hot new programming language, Java. Andreessen figured that all the Java bandwagon devotees would gravitate to their new browser programming language if it had the name Java in it. In December 1995, Andreessen got his wish when Netscape obtained a trademark license from Java manufacturer, Sun Microsystems, and LiveScript's name was changed to JavaScript. Unfortunately, many, many people over the years have made the mistake of assuming that JavaScript is the same as Java or very close to it. Don't be fooled by the name—JavaScript is not all that similar to Java. Actually, C++ and other popular programming languages are closer to Java than JavaScript is.

In 1996, Netscape submitted JavaScript to the Ecma International[1] standards organization to promote JavaScript's influence on all browsers (not just Netscape's browser). Ecma International used JavaScript as the basis for creating the ECMAScript standard. As hoped, ECMAScript now serves as the standard for the interactive programming languages embedded in all of today's popular browsers. At the time of this book's printing, the most recent ECMAScript version is version 7, published in 2016. Coming up with the name ECMAScript was a difficult process, with different browser manufacturers having strong opposing views. JavaScript creator Brendan Eich has stated that the result, ECMAScript, is "an unwanted trade name that sounds like a skin disease."[2]

In 1998, Netscape formed the Mozilla free-software community, which eventually implemented Firefox, one of today's premier browsers. Brendan Eich moved to Mozilla, where he and others have continued to update JavaScript over the years, following the ECMAScript standard as set forth by Ecma International.

Other browser manufacturers support their own versions of JavaScript. For their Internet Explorer and Edge browsers, Microsoft uses JScript. For their Chrome browser, Google uses the V8 JavaScript Engine. Fortunately, all the browser manufacturers attempt to follow the ECMAScript

---

[1] Ecma International is a standards organization for information and communication systems. The organization's former name was the European Computer Manufacturers Association (ECMA), but they changed their name to Ecma International (with Ecma no longer being an acronym) to broaden their appeal to those outside of Europe.

[2] Eich, Brendan, "Will there be a suggested file suffix for es4?," Mail.mozilla.org, October 3, 2016, https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html.

standard, so for most tasks, programmers can write one version of their code and it will work for all the different browsers. In this book, we stick with standard ECMAScript code that works the same on all browsers. As with almost everyone in the web-programming community, we refer to our code as JavaScript, even though JavaScript is just one of several ECMAScript implementations (JavaScript is the implementation used in Mozilla's Firefox).

8.3 Hello World Web Page

When learning a new programming language, your first program is supposed to simply print "Hello, world!" That's the traditional Hello World program. Because the program's task is so simple, the code is short and it gives the learner an opportunity to focus on the syntax basics and not get bogged down with too many details. With that said, your first JavaScript "program" is embedded in FIGURE 8.1's Hello web page, which displays "Hello, world!" when the user clicks the button.

For the book's web pages that use JavaScript, as in the Hello web page shown in Figure 8.1, you'll need to enter the web page's URL in a browser and interact with the web page to fully appreciate how the web page works. The book's preface provides the URL where you can find the book's web pages. So go ahead and find the Hello web page's URL, enter it in a browser, click the web page's button, and be amazed as the "To see the ..." text gets replaced by the large "Hello, world!" text.

Now take a look at the Hello web page's source code in FIGURE 8.2. You can see that there's not much JavaScript—it's just the code in the `script` container and the code that follows the `onclick` attribute. The rest of the web page is HTML code. Later, we'll explain the JavaScript

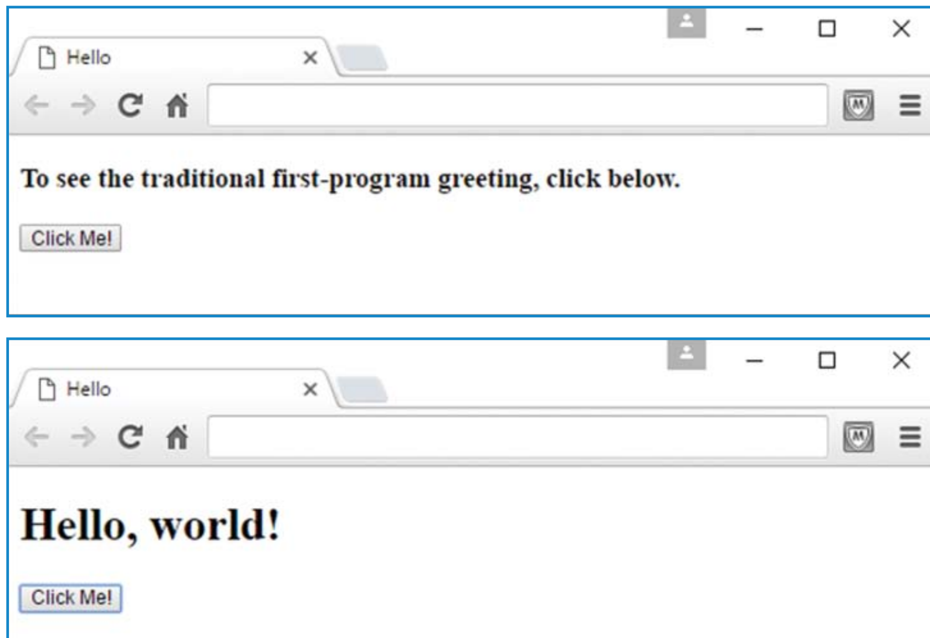
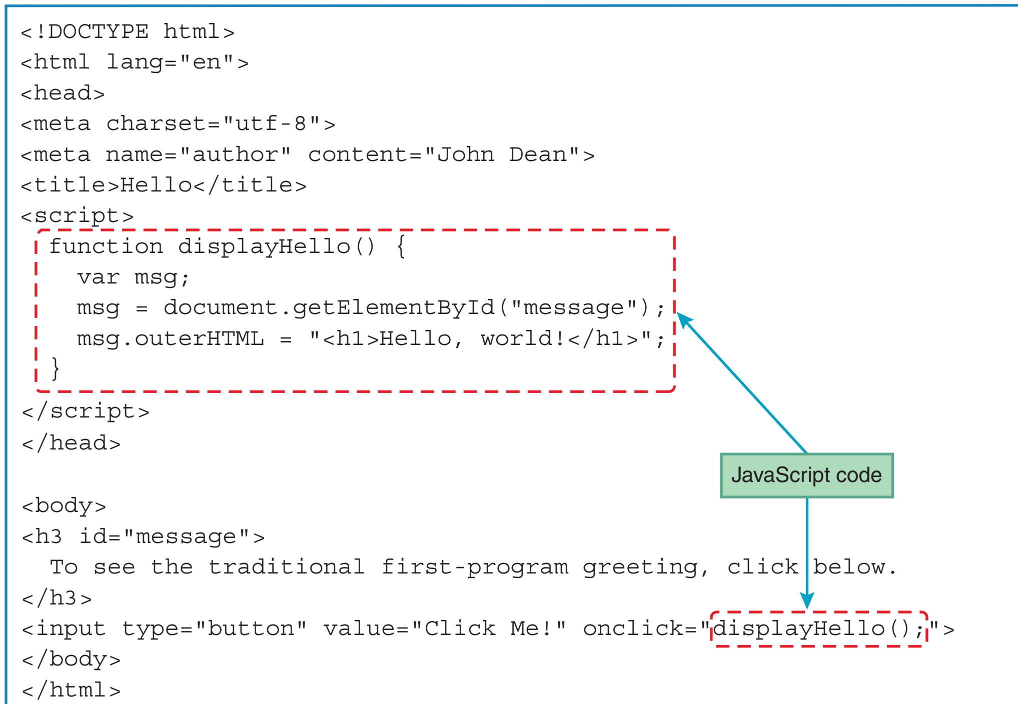


FIGURE 8.1 Initial display and what happens after the user clicks the button on the Hello web page



```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Hello</title>
<script>
  function displayHello() {
    var msg;
    msg = document.getElementById("message");
    msg.outerHTML = "<h1>Hello, world!</h1>";
  }
</script>
</head>

<body>
<h3 id="message">
  To see the traditional first-program greeting, click below.
</h3>
<input type="button" value="Click Me!" onclick="displayHello();">
</body>
</html>
```


FIGURE 8.2 Source code for Hello web page

code that handles the text replacement when the user clicks the button, but let's first examine the HTML code that's in charge of displaying the web page's button.

8.4 Buttons

There are different types of buttons, each with its own syntax. To keep things simple, we'll start with just one type of button, and here's its syntax:

```
<input type="button"
  value="button-label"
  onclick="click-event-handler">
```

Note that the code is a void element that uses the `input` tag. As its name suggests, the `input` tag implements elements that handle user input. You might not think of a button as user input, but it is—the user chooses to do something by clicking a button. Later, we'll introduce other user input elements (e.g., text controls and checkboxes) that also use the `input` tag.

As we've done throughout the book when introducing new constructs, the prior code fragment shows only the most important syntax details, so you don't get overwhelmed with too much to remember. We're showing the `input` element's most common attributes—`type`, `value`, and `onclick`. Note how the `input` element at the bottom of Figure 8.2 follows this syntax pattern and includes those three attributes.

The `input` element is used for different types of user input, and its `type` attribute specifies which type of user input. More formally, the `type` attribute specifies the type of *control* that's being implemented, where a control is a user input entity such as a button, text control, or checkbox. In the Hello web page source code, note that the `type` attribute gets the value `button`, which tells the browser to display a button. If you don't provide a `type` attribute, the browser will display a text control, because that's the default type of control for the `input` element. We'll describe text controls later in this chapter. For now, just know that a *text control* is a box that a user can enter text into, and this is what a (filled-in) text control (with a prompt at its left) looks like:

First Name: Ahmed

Because it uses a box, many web developers refer to text controls as "text boxes." We use the term "text control" because that's the term used more often by the HTML standards organizations.

The `input` element's `value` attribute specifies the button's label. If you don't provide a `value` attribute, the button will have no label. If you want a button with no label, rather than just omitting the `value` attribute, we recommend that you specify `value=""`. That's a form of self-documentation, and it makes your code more understandable.

The `input` element's `onclick` attribute specifies the JavaScript instructions that the JavaScript engine executes when the user clicks the button. What's a JavaScript engine, you ask? A *JavaScript engine* is the part of the browser software that runs a web page's JavaScript. In the Hello web page, the `onclick` attribute's value is JavaScript code that "handles" what's supposed to happen when the user clicks the button. Clicking the button is considered to be an *event*, so the `onclick` attribute's JavaScript code is known as an *event handler*. Besides `onclick`, there are other attributes that call event handlers, like `onfocus` and `onload`, but they aren't used much for buttons, so they won't be introduced until later, when they will be more useful with other types of controls.

In the Hello web page source code, note that the `onclick` attribute's value is simply `displayHello();`. That calls the `displayHello` function, which is defined in the web page's `script` block. We'll discuss function calls and function definitions in the next section.

8.5 Functions

A function in JavaScript is similar to a mathematical function. A mathematical function receives arguments, performs a calculation, and returns an answer. For example, the sin(x) mathematical function receives the x argument, calculates the sine of the given x angle, and returns the calculated sine of x. Likewise, a JavaScript function might receive arguments, will perform a calculation, and might return an answer. Here's the syntax for calling a function:

function-name(*zero-or-more-arguments-separated-by-commas*);

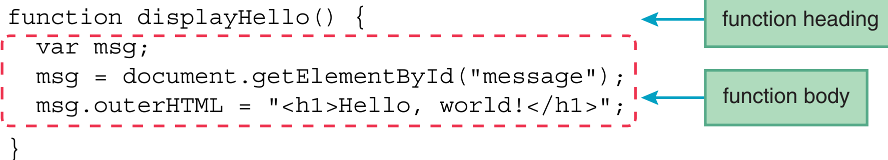
As mentioned earlier, the Hello web page button has an `onclick` attribute with a value of `displayHello();`. That's a JavaScript function call, and its syntax matches the preceding

syntax. Note the parentheses are empty because there's no need to pass any argument values to the `displayHello` function. If there were arguments, they would need to be separated by commas, and proper style suggests that you insert a space after each comma.

Here's the syntax for a function definition:

```
function function-name(zero-or-more-parameters-separated-by-commas) {
  statement-1;
  statement-2;
  ...
  last-statement;
}
```

And here's the `displayHello` function definition from the Hello web page:



You should be able to recognize that the `displayHello` function definition follows the prior syntax. The parentheses in the *function heading* are empty because the function call's parentheses are empty (the function call was `displayHello();`). If there are arguments in the function call, then you'll normally have the same number of parameters in the function heading—one parameter to receive each argument's value. Note how we're using the term *argument* for the values in a function call's parentheses and the term *parameter* for the associated words in a function definition heading's parentheses. Some people use the term argument for both, but to make it easier to distinguish between the function call and the function definition, we'll stick with the separate formal names—argument and parameter.

Normally, function definitions should be placed (1) in a `script` container in the web page's `head` container or (2) in an external JavaScript file. Go back to the Hello web page code in Figure 8.2 and verify that the `displayHello` function definition is in a `script` container. You'll want to use an external JavaScript file if you have lots of JavaScript code. We'll show a web page that uses an external JavaScript file later in the book.

Looking at the previous code fragment, you can see three lines in the function's body. Each line is a JavaScript *statement*, where a statement performs a task. Note the semicolons at the end of all three statements. Semicolons are required at the end of a JavaScript statement only if the JavaScript statement is followed by another JavaScript statement, so it would have been legal to omit the semicolon after the last statement. However, coding conventions dictate that you terminate every statement with a semicolon, even the last one. Why? Suppose there's no semicolon at the end of the last statement and someone later adds a new statement after the last statement. If they forget to insert a semicolon between the two statements, that creates a bug. Another reason to insert a semicolon after the last statement is that if you don't do it, the JavaScript engine does it for you behind the scenes, and that slows things down slightly.

8.6 Variables

Let's continue working our way through the code in the Hello web page's function. Here's the function's first statement:

```
var msg;
```

The `msg` thing is a variable. You should already be familiar with variables in algebra. You can think of a variable as a box that holds a value. In this case, the `msg` variable will hold a string that forms a message.

Before you use a variable in JavaScript code, you should use `var` to declare the variable in a *declaration statement*. For example:

```
var name;
var careerGoals;
```

Words that are part of the JavaScript language are known as *keywords*. The word `var` is a keyword. On the other hand, `name` and `careerGoals` are not keywords because they are specific to a particular web page's code and not to the JavaScript language in general. In the previous section, we showed the `displayHello` function. In that function, besides `var`, can you identify another keyword? The function heading uses the word `function` and `function` is a keyword. With most of JavaScript's keywords, it's illegal for you as a programmer to redefine them to mean something else. So you cannot use "function" as the name of a variable. Those keywords are "reserved" for the JavaScript language, and they are known as *reserved words*. There are a few keywords that can be redefined by a programmer, but an explanation is beyond the scope of this book. Many programmers use the terms "keywords" and "reserved words" interchangeably, but there is a slight difference, as not all keywords are reserved words.

In most programming languages, when you declare a variable, you specify the type of values that the variable will be allowed to hold—numbers, strings, and so on. However, with JavaScript, you do not specify the variable's type as part of the declaration. The variable's type is determined dynamically by the type of the value that's assigned into the variable. For example:

```
name = "Mia Hamm";
careerGoals = 158;
```

What type of value is `"Mia Hamm"`? A string, since a *string* consists of zero or more characters surrounded by a pair of double quotes (`"`) or a pair of single quotes (`'`). What type of value is 158?[3] A number, since a number consists of digits with an optional decimal point. JavaScript is known as a *loosely typed language*, or a *dynamically typed language*, which means that you do not declare a variable's data type explicitly, and you can assign different types of values into a variable at

[3] Mia Hamm held the record of 158 goals in women's international soccer team play until fellow American Abby Wambach broke the record in 2013.

different times during the execution of a program. For example, it would be legal to assign a string to `name` and then later assign a number to `name`. But proper coding conventions dictate that you don't do that, because it can lead to code that's difficult to understand.

8.7 Identifiers

An identifier is the technical term for a program component's name—the name of a function, the name of a variable, and the names of other program components we'll get to later on. In the Hello web page, `displayHello` was the identifier for the function name, and `msg` was the identifier for a variable. In naming your variables and functions, the JavaScript engine requires that you follow certain rules. Identifiers must consist entirely of letters, digits, dollar signs (`$`), and/or underscore (_) characters. The first character must not be a digit. If you do not follow these rules, your JavaScript code won't work.

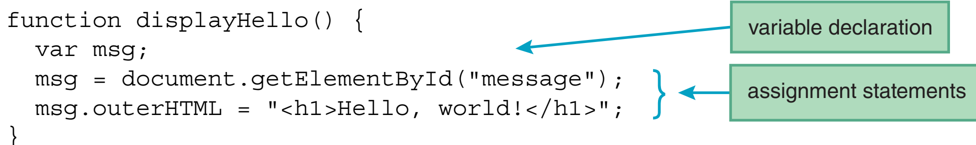
Coding-convention rules are narrower than the preceding rules. Coding conventions suggest that you use letters and digits only, not dollar signs or underscores. They also suggest that all letters should be lowercase except the first letter in the second word, third word, and so on. That's referred to as *camel case*, and here are a few examples: `firstName`, `message`, `daysInMonth`. Notice that the identifiers' words are descriptive. Coding conventions suggest that you use descriptive words for your identifiers. Beginning programmers have a tendency to use names like `x`, `y`, and `num`. Normally, those are bad variable names. However, if you have a situation in which you're supposed to read in a number and the number doesn't represent anything special, then `x` or `num` is OK.

If any of the coding conventions are broken, it won't affect your web page's ability to work properly, but your code will be harder to understand and maintain. If your code is harder to understand and maintain, that means programmers who work on the code in the future will have to spend more time in their efforts, and their time costs money. Normally, programmers spend more time working on old code (making bug fixes and making improvements) rather than writing new code, and all that work on old code costs money. So to help with your present or future company's bottom line (profit and world domination), put in the time up front writing good code.

8.8 Assignment Statements and Objects

We still haven't finished with the Hello web page. Actually, we still haven't explained the magic behind how the web page replaces the initial message with "Hello, world!" when the user clicks the button. To understand how that works, we need to talk about assignment statements and objects.

Once again, here's the Hello web page's `displayHello` function:



In the function's body, the first statement is a variable declaration for the msg variable. After you declare a variable, you'll want to use it, and the first step in using a variable is to put a value inside it. An assignment statement puts/assigns a value into a variable. As you can see in the preceding example, the function body's second and third statements are assignment statements. The assignment operator (=) assigns the value at the right into a variable at the left. So in the first assignment statement, the document.getElementById("message") thing gets assigned into the msg variable. In the second assignment statement, the "<h1>Hello, world!</h1>" thing gets assigned into the msg.outerHTML variable.

Those two assignment statements are pretty confusing. To understand the syntax requires an understanding of objects. An *object* is a software entity that represents something tangible. The fact that it's software means that it can be manipulated with JavaScript code, which provides you, the programmer, with great power!

Behind the scenes, all of the elements in a web page are represented as objects. When a browser loads the Hello web page, the browser software generates objects for the head element, the body element, the h3 element, and so on. There's also an object associated with the entire web page, and that object's name is document. Each object has a set of related properties, plus a set of behaviors. A *property* is an attribute of an object. A behavior is a task that the object can perform. The document object contains properties like the web page's type. Most web pages these days (and all the web pages in this book) have a value of HTML5 for the document object's type property. But it's possible to have other types, like HTML 4.01 or XHTML 1.0 Strict. The type property's value comes from the doctype instruction, which should appear at the top of every web page. Here's the Hello web page's doctype instruction:

```
<!DOCTYPE html>
```

The html value indicates that the document object's type is HTML5.[4] To access an object's property, you specify the object name, a dot, and then the property name. So to access the current web page's document type, use document for the object name, . for dot, and doctype for the property. Here's the JavaScript code:

```
document.doctype
```

Remember that an object is not only a set of properties, but also a set of behaviors. One of the document object's behaviors is its ability to retrieve an element using the element's id value. In JavaScript (and many other programming languages, as well), an object's behaviors are referred to as *methods*. To retrieve an element, the document object uses its getElemementById method. To call an object's method, you specify the object name, a dot, the method name, and

---

[4] It might seem odd that the html value indicates HTML5, but as you might recall, the standards organizations worked very hard to move from older versions of HTML to HTML5. By having html indicate HTML5, the W3C makes HTML5 the default and furthers HTML5's position as king. If you want another version of HTML, like HTML 4.01, you have to provide a doctype instruction with a value different from html—a value too painfully long and ugly to show here.

then parentheses around any arguments you want to pass to the method. For example, here's the getElemementById method call from the Hello web page's displayHello function:

```
document.getElementById("message")
```

See how the method call includes "message" for its argument? In executing the method call, the JavaScript engine searches for an element with id="message". There is such an element in the Hello web page, and here it is:

```
<h3 id="message">
  To see the traditional first-program greeting, click below.
</h3>
```

So the getElementById method call retrieves that h3 element.

The HTML5 standard says that an id attribute's value must be unique for a particular web page. You might recall how we used an id attribute to identify a target for a link within a web page. Using an id attribute is necessary in that situation because we need a link's target to be unique. Likewise, we use an id attribute to retrieve an element (with getElementById) so there won't be any confusion in terms of which element to retrieve.

Let's get back to explaining the displayHello function. Here it is again:

```
function displayHello() {
  var msg;
  msg = document.getElementById("message");
  msg.outerHTML = "<h1>Hello, world!</h1>";
}
```

Previously, we said the getElementById method call retrieves the h3 element. Well, almost. Actually, the getElementById method retrieves the object associated with the h3 element. In the displayHello function, you can see that the getElementById method call is on the right-hand side of an assignment statement, so the method's returned value (the h3 element's object) gets assigned into the variable at the left of the assignment statement. After msg gets the h3 element's object, that object gets updated with this assignment statement:

```
msg.outerHTML = "<h1>Hello, world!</h1>";
```

Note msg.outerHTML. All element objects have an outerHTML property, which stores the element's code, including the element's start and end tags. The msg variable holds the h3 element's object, so msg.outerHTML holds the h3 element's code. Assigning <h1>Hello, world!</h1> to msg.outerHTML causes msg's code to be replaced with <h1>Hello, world!</h1>. Thus, when the button is clicked, the original h3 message gets replaced with an h1 "Hello, world!" message. Go back to Figure 8.1 and confirm that the "Hello, world!" text is larger than the original "To see the …" text. That should make sense, now that you realize that the h3 start and end tags get replaced with h1 start and end tags.

As you might have guessed, in addition to outerHTML, there's also an innerHTML property. It accesses the content that's between the element's start and end tags, and it does not include the element's start and end tags. Later on, we'll use innerHTML in a separate web page example.