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:

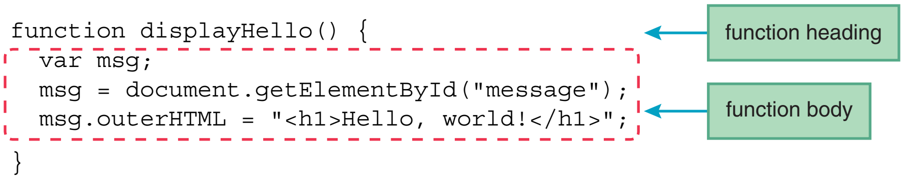
function-name(*zero-or-more-arguments-separated-by-commas*);

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:



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

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.

```
}
```

8.8 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:

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

variable declaration

assignment statements

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 `getElementById` 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.

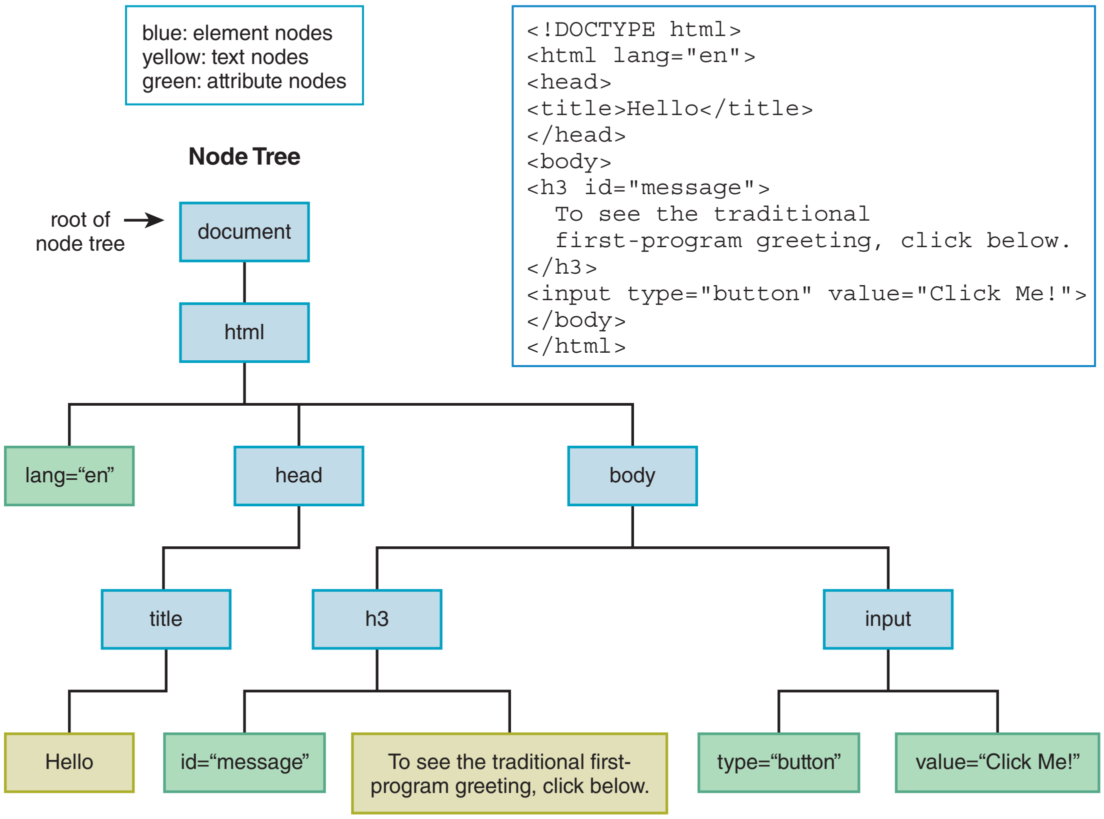
As you might have guessed, in addition to `outerHTML`, there's also an `innerHTML` property. It accesses the content that's between the element's start and end tags, and it does not include the element's start and end tags. Later on, we'll use `innerHTML` in a separate web page example.

8.9 Document Object Model

Whew! We've finally finished examining the Hello web page code. The examination process required getting down in the weeds and learning about objects. Now let's step back and look at a big-picture issue related to objects. Let's examine how a web page's objects are organized.

The Document Object Model, which is normally referred to as the DOM, models all of the parts of a web page document as nodes in a node tree. A node tree is similar to a directory tree, except instead of showing directories that include other directories (and files), it shows web page elements that include other elements (and text and attributes). Each node represents either (1) an element, (2) a text item that appears between an element's start and end tags, or (3) an attribute within one of the elements. If that doesn't make sense, no worries. See the node tree example in FIGURE 8.3, and you should be able to understand things better by examining how the code maps to the nodes in the node tree.

The figure's code is a stripped-down version of the Hello web page code shown earlier, with some of its elements and attributes (e.g., the `meta` and `script` elements) removed. The node tree shows blue nodes for each element in the web page code (e.g., `head` and `title`). It shows yellow nodes for each text item that appears between an element's start and end tags (e.g., "Hello"). And it shows green nodes for each attribute in the web page document's elements (e.g., `h3`'s `id` attribute).



```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Hello</title>
</head>
<body>
<h3 id="message">
  To see the traditional
  first-program greeting, click below.
</h3>
<input type="button" value="Click Me!">
</body>
</html>
```

FIGURE 8.3 **Node tree for simplified Hello web page**

Note that the nodes are arranged in a hierarchical fashion, where nodes at the top contain the nodes below them (e.g., the `head` node contains the `title` node). The node at the top of the node tree is the `document` object, which we discussed earlier. Using computer science terminology, the node at the top of a tree is called the *root node*.

The term *dynamic HTML* refers to updating the web page's content by manipulating the DOM's nodes. Assigning a value to an element object's `outerHTML` property (as in the Hello web page) is one way to implement dynamic HTML. We'll see other techniques later.

The main point of explaining the DOM is for you to get a better grasp of how everything in a web page is represented behind the scenes as an object. As a web programmer, you can use the DOM's hierarchical structure to access and update different parts of the web page. The DOM provides different ways to access the nodes in the node tree. Here are three common techniques:

1. You can retrieve the node tree's root by using `document` (for the `document` object) in your code and then use the root object as a starting point in traversing down the tree.
2. You can retrieve the node that the user just interacted with (e.g. a button that was clicked) and use that node object as a starting point in traversing up or down the tree.
3. You can retrieve a particular element node by calling the `document` object's `getElementById` method with the element's `id` value as an argument.

In the Hello web page, we used the third technique, calling `getElementById`. Later on, we'll provide web page examples that use the first two techniques. We hope you're excited to know what you have to look forward to![5]

[5] If you read about those techniques in the later chapters and that doesn't satiate your quest for knowledge, you can learn yet another technique on your own. Using a node object from the DOM node tree, you can call `getElementsByTagName` to retrieve all of the node's descendant elements that are of a particular type (e.g., all the `div` elements).