

2

CHAPTER

WORKING WITH FUNCTIONS, DATA TYPES, AND OPERATORS

When you complete this chapter, you will be able to:

- › Use functions to organize your JavaScript code
- › Use expressions and operators
- › Identify the order of operator precedence in an expression

So far, the code you have written has consisted of simple statements placed within script sections. However, like most programming languages, JavaScript allows you to group programming statements in logical units. In JavaScript, a group of statements that you can execute as a single unit is called a **function**. You'll learn how to create functions in this chapter, and you'll practice using them to organize your code.

In addition to functions, one of the most important aspects of programming is the ability to store values in computer memory and to manipulate those values. In the

last chapter, you learned how to store values in computer memory using variables. The values, or data, contained in variables are classified into categories known as data types. In this chapter, you'll learn about JavaScript data types and the operations that can be performed on values of each type. You'll also explore the order in which different operations are performed by JavaScript processors, as well as how to change this order.

Working with Functions

In Chapter 1, you learned that procedures associated with an object are called methods. In JavaScript programming, you can write your own procedures, called functions. A function is a related group of JavaScript statements that are executed as a single unit. A function, like all JavaScript code, must be contained within a `script` element. In the following section, you'll learn more about incorporating functions in your scripts.

Defining Functions

JavaScript supports two different kinds of functions: named functions and anonymous functions. A **named function** is a set of related statements that is assigned a name. You can use this name to reference, or **call**, this set of statements in other parts of your code. An **anonymous function**, on the other hand, is a set of related statements with no name assigned to it. The statements in an anonymous function work only in a single context—the place in the code where they are located. You cannot reference an anonymous function anywhere else in your code.

Generally, you use a named function when you want to be able to reuse the function statements within your code, and you use an anonymous function for statements that you need to run only once.

Before you can use a function in a JavaScript program, you must first create, or define, it. The lines that make up a function are called the **function definition**.

The syntax for defining a named function is:

```
function name_of_function(parameters) {  
    statements;  
}
```

The syntax for defining an anonymous function is the same as that for a named function except that it does not specify a name:

```
function (parameters) {  
    statements;  
}
```

Parameters are placed within the parentheses that follow a function name. A **parameter** is a variable that is used within a function. Placing a parameter name within the parentheses of a function definition is the equivalent of declaring a new variable. However, you do not need to include the `var` keyword. For example, suppose that you want to write a function named `calculateSquareRoot()` that calculates the square root of a number contained in a parameter named `number`. The start of the function declaration would then be written as

```
calculateSquareRoot(number)
```

In this case, the function declaration is declaring a new parameter (which is a variable) named `number`. Functions can contain multiple parameters separated by commas. If you wanted to create a function that calculated the volume of a box, the function would need values for the length, width, and height of the box. You could write the start of the function declaration as

```
calculateVolume(length, width, height)
```

Note that parameters (such as `length`, `width`, and `height`) receive their values when you call the function from elsewhere in your program. (You will learn how to call functions in the next section.)

Note

Functions do not have to contain parameters. Many functions only perform a task and do not require external data. For example, you might create a function that displays the same message each time a user visits your website; this type of function only needs to be executed and does not require any other information.

Following the parentheses that contain the function parameters is a set of braces (called **function braces**) that contain the function statements. **Function statements** are the statements that do the actual work of the function, such as calculating the square root of the parameter, or displaying a message on the screen. Function statements must be contained within the function braces. The following is an example of a `calculateVolume()` function including function statements:

```
1 function calculateVolume(length, width, height) {  
2     var volume = length * width * height;  
3     document.write(volume);  
4 }
```

Notice how the preceding function is structured. The opening brace is on the same line as the function name, and the closing brace is on its own line following the function statements. Each statement between the braces is indented. This structure is the preferred format among many JavaScript programmers. However, for simple functions it is sometimes easier to

include the function name, braces, and statements on the same line. (Recall that JavaScript ignores line breaks, spaces, and tabs outside of string literals.) The only syntax requirement for spacing in JavaScript is that semicolons separate statements on the same line.

Note

The code in this book is indented using three space characters. The number of spaces used for indenting isn't important, as long as you use the same number consistently throughout your code. Some programmers prefer to use tab characters instead of spaces for indents; this choice is also simply a question of personal preference, and has no effect on the quality of the code.

It's common practice to place functions in an external .js file and include a script section that references the file at the bottom of an HTML document's body section.

In this chapter, you'll add JavaScript code to a web page for Fan Trick Fine Art Photography, a photography business that sells photographic prints and offers special event photography services. The owners want to expand their website to offer information about digital photography, and to provide a rate estimator for their services for prospective customers. Your Chapter folder for Chapter 2 contains the files you will need for the project. Figure 2-1 shows a preview of the completed web form in a desktop browser, and Figure 2-2 shows it in a mobile browser.

The screenshot shows a web browser displaying the Fan Trick Fine Art Photography website. The header features the company logo and navigation links: About, Portfolio, Estimate, and Digital VIX. The main content area is titled 'Estimate' and includes a description of the services. A green callout box points to the 'Total Estimate: \$1100' on the left. Another green callout box points to the 'Form selections change calculated total' on the right. The form includes sections for Photography and Travel, with various options and prices.

Section	Option	Price
Photography	# of photographers (1-4)	2
	# of hours to photograph (minimum 2)	4
	Memory book	\$250
	Reproduction rights for all photos	\$1250
Travel	Event distance from Austin, TX	25
	\$1/mi per photographer	

Total Estimate: \$1100

Form selections change calculated total

Figure 2-1: Completed estimate form in a desktop browser

© 2015 Cengage Learning®

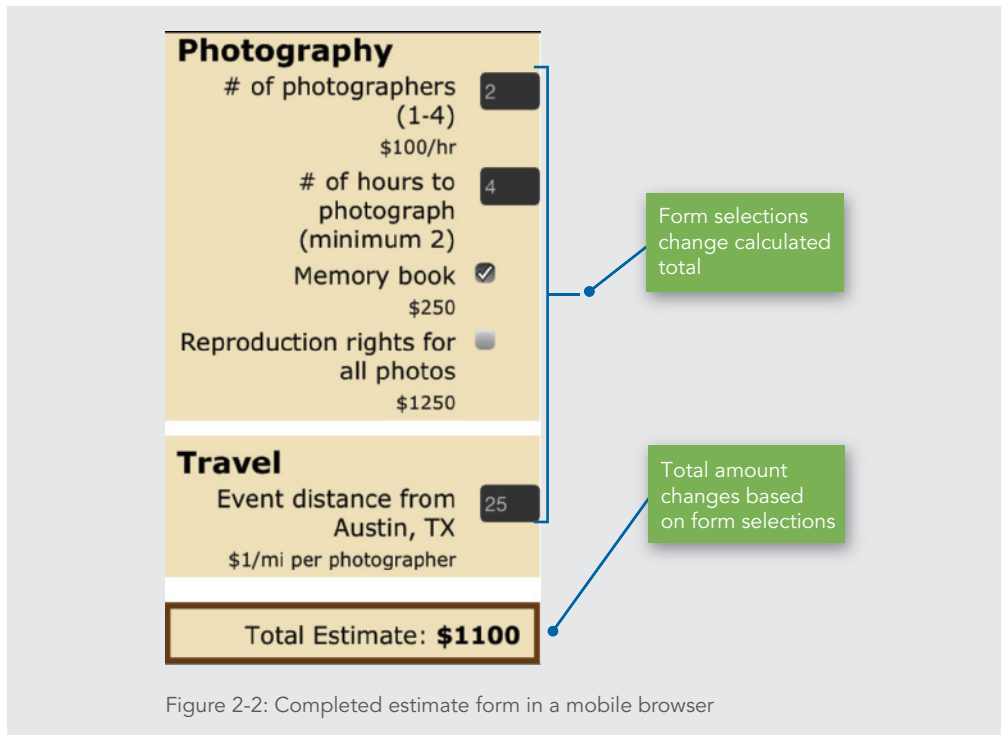


Figure 2-2: Completed estimate form in a mobile browser

To start working on the estimate page:

1. In your text editor, open the **estimate.htm** file, located in your Chapter folder for Chapter 2.
2. In the comment section at the top of the document, type your name and today's date where indicated, and then save your work.
3. Scroll through the document to familiarize yourself with its contents. It includes a form as well as an `aside` element that contains the text "Total Estimate:". Later in the chapter, you'll write code that calculates the cost of the options selected in the form and displays them in the `aside` element.
4. Open **estimate.htm** in a browser, click the **# of photographers** text box, change the value to 2, and then click the **Memory book** check box to insert a check. Nothing on the page changes when you make form selections because no JavaScript code is associated with the form fields. You'll add code later in this chapter.
5. Click your browser's **Refresh** or **Reload** button. Depending on which browser you're using, the form may still display the changes you made.

6. Press and hold the **Shift** key, and then click your browser's **Refresh** or **Reload** button. The form is reset to its default selections. Because your clients don't want to include a Reset button on the form, users could use Shift + Refresh or Shift + Reload to clear the form. However, many users don't know about this shortcut. To make the form as user-friendly as possible, your clients would instead like users to be able to clear the form by simply reloading the page. You'll create a function to accomplish this.
7. In your text editor, open the **ft.js** file located in your Chapter folder for Chapter 2, enter your name and today's date in the comment section, and then save your work. You'll create the code for your app in this external file, and then use a `script` element to reference it in the HTML document.
8. Enter the following code in the **ft.js** file:

```
1  // sets all form field values to defaults
2  function resetForm() {
3      document.getElementById("photognum").value = 1;
4      document.getElementById("photoghrs").value = 2;
5      document.getElementById("membook").checked = false;
6      document.getElementById("reprodrights").checked = false;
7      document.getElementById("distance").value = 0;
8  }
```

This code creates a function named `resetForm()`. Recall the `getElementById()` method of the `Document` object looks up an element by its HTML `id` attribute value. Each line in the `resetForm()` function references one of the input elements in the form and sets it to its default value. You use the `value` property to set the value for a text input element, and you use the `checked` property to specify whether a check box input element should be checked (`true`) or unchecked (`false`).

9. Save your work, return to the **estimate.htm** file in your text editor, and then near the bottom of the document, just before the closing `</body>` tag, enter the following `script` element:

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

This element loads the code from the **ft.js** file.

Note

Browsers load and render HTML elements in the order they appear in a HTML document. Referencing the script at the end of the body section ensures that browsers delay loading the .js file until the web page contents are loaded. This is a good habit to develop because it ensures the page content is displayed as quickly as possible, providing a better user experience on large websites.

10. Save your work, refresh or reload **estimate.htm** in your browser, click the # of **photographers** text box, change the value to **2**, click the **Memory book** check box to select it, and then click your browser's **Refresh** or **Reload** button. Unless your browser previously cleared input when you clicked Refresh or Reload, note that nothing happens. This is because browsers don't execute functions until they are triggered.

Calling Functions

A named function definition does not execute automatically. Creating a named function definition only names the function, specifies its parameters, and organizes the statements it will execute. To execute a named function, you must invoke, or call, it from elsewhere in your program. The code that calls a named function is referred to as a **function call** and consists of the function name followed by parentheses, which contain any variables or values to be assigned to the function parameters.

Passing Arguments to a Function

The variables or values that you place in the parentheses of the function call statement are called **arguments** or **actual parameters**. Sending arguments to the parameters of a called function is known as **passing arguments**. When you pass arguments to a function, the value of each argument is then assigned to the value of the corresponding parameter in the function definition. (Again, remember that parameters are simply variables that are declared within a function definition.)

For instance, the following code contains three functions. The `calculateSum()` and `calculateVolume()` functions each perform different mathematical manipulations on values passed to them when they are called. The last line of each function, however, is identical, passing the results of the calculation and the reference to an element as arguments to a third function, `updateResult()`. Based on the values passed to it, the `updateResult()` function updates the result value in the document.

```
1  function calculateSum(value1, value2, value3) {  
2      var sum = value1 + value2 + value3;  
3      var location = document.getElementById("sum");  
4      updateResult(sum, location);
```

```

5  }
6  function calculateVolume(length, width, height) {
7      var volume = length * width * height;
8      var location = document.getElementById("volume");
9      updateResult(volume, location)
10 }
11 function updateResult(result, element) {
12     element.innerHTML = result;
13 }

```

Note

Later in this chapter, you'll learn how the `+` and `*` operators work and what the `innerHTML` property does. For now, just notice that you can pass arguments to a function when you call it.

Handling Events with Functions

You can also call functions in response to browser events. Browsers support three different methods for doing this: HTML attributes, object properties, and event listeners.

The simplest way to specify a function as an event handler is to specify the function as the value for the associated HTML attribute. You saw in Chapter 1 how to use an event attribute to run a statement. For instance, the following code displays an alert box containing a message when a user clicks the `input` element in which it is located:

```

<input type="submit" onclick="window.alert('Thanks for your
order! We appreciate your business.')" />

```

Instead of specifying a JavaScript statement as the value of the `onclick` attribute, you can instead specify a function name. Then, when the event fires, the function will be executed. For instance, if you had a named function called `showMessage()` that you instead wanted to specify as the event handler for the click event on the `input` element, you could replace the previous code with the following:

```

<input type="submit" onclick="showMessage()" />

```

Whenever a user clicks this `input` element, the browser runs the `showMessage()` function.

One drawback of specifying event handlers with HTML attributes is they require developers to place JavaScript code within HTML code. Just as developers generally avoid using inline CSS styles to keep HTML and CSS code separate, most developers prefer not to mix HTML

and JavaScript code in the same file. Instead, they maintain separate HTML and JavaScript files. Fortunately, there are two other ways to specify functions as event handlers.

One alternative is to specify the function as a property value for the object representing the HTML element. Every element has an *onevent* property for each event it supports. For instance, a Submit button (an `input` element with a `type` value of `submit`) supports the `click` event, so you can specify a value for its `onclick` property. Within a `.js` file, you could specify the `showMessage()` function as the event handler for a submit button as follows:

```
document.getElementById("submitButton").onclick =  
    showMessage;
```

This code references the Submit button using the value of its HTML `id` attribute. Note that the function name is not followed by parentheses.

Although this option enables you to separate HTML and JavaScript code, it has one major drawback: you can assign only one event handler per event. In more complex code, you might want to specify several event handlers to fire in response to a given event. To assign multiple event handlers to a single event, you need to use the third and final technique, which is the `addEventListener()` method. Every web page element has an `addEventListener()` method, which uses the following syntax:

```
element.addEventListener("event", function, false);
```

where *element* is a reference to the element, *event* names the event to handle, and *function* names a function that will handle the event. The third argument, `false`, is generally included for compatibility with some older browsers. Specifying an event handler with the `addEventListener()` method is known as **adding an event listener**. The following statement specifies the `showMessage()` function as an event handler for the `click` event on the element with the `id` `submitButton`:

```
var submit = document.getElementById("submitButton");  
submit.addEventListener("click", showMessage, false);
```

Note that as in the object property above, the function name is not followed by parentheses in an event listener. Also notice that the event is specified by the event name only ("`click`") rather than by prefixing the event name with `on` ("`onclick`").

Note

The `addEventListener()` method is supported by all modern browsers, but it is not supported by Internet Explorer version 8 or earlier. These older browsers support a different method, `attachEvent()`, which serves the same purpose. In Chapter 3, you'll learn to use conditional statements to write code that detects and uses the appropriate method for the current browser. The remaining code in this chapter, however, will not work in IE8.

You can also specify an anonymous function as an event handler using the `addEventListener()` method. To do so, you simply substitute the entire function definition for the function name in the statement. For instance, the following statement replaces the `showMessage` function reference in the previous example with an anonymous function:

```
1 var el = document.getElementById("submitButton");
2 el.addEventListener("click", function() {
3     window.alert("Thanks for your order! We appreciate your
4     business.");
5 }, false);
```

The highlighted section of the above code represents the anonymous function. Notice that it comes after the name of the event, and is followed by the word `false`, a closing parenthesis, and a semicolon, just as in the previous statement that used a function name.

Next, you will add code to call the `resetForm()` function by using the `addEventListener()` method to specify the function as an event handler of the `onload` event handler of the `Window` object.

To specify the `resetForm()` function as an event handler:

1. Return to the `ft.js` file in your text editor.
2. On a new line below the function you added, type the following comment and statement:

```
// resets form when page is reloaded
document.addEventListener("load", resetForm, false);
```

This statement specifies the `resetForm()` function you just created as an event handler for the `load` event of the `Window` object. Because this statement is not part of the function, it is executed as soon as a browser parses it. This ensures whenever a user reloads or refreshes the page, your `resetForm()` function returns all form values to their defaults.

3. Save your work, refresh or reload **estimate.htm** in your browser, change the value in the **# of photographers** text box to 2, click the **Memory book** check box, and then click your browser's **Refresh** or **Reload** button. Reloading the page should trigger the function you created, and the form values should return to their defaults.

Note

If the form values don't reset as expected, check the contents of your function against Step 8 in the previous set of steps, check your `script` element against Step 9 in the previous set of steps, check your function call against Step 2 above, make any necessary changes, and then repeat Step 3. If your form values still don't reset, continue to the next section, where you'll use the browser console to identify errors in your code.

Locating Errors with the Browser Console

Even the most careful developer introduces unintentional errors into code from time to time. Even a small mistake, such as incorrect capitalization or the omission of a closing quote, parenthesis, or brace, can prevent a browser from processing your code, and result in a document that doesn't function as you intended.

When a browser encounters an error that keeps it from understanding code, it generates an error message. However, this message is displayed in a pane known as a **browser console**, or simply **console**, which is hidden by default to avoid alarming users. As a developer, however, it can be useful to display the browser console pane to see any errors that your code may generate.

Next you'll introduce an intentional error in your code, and then open the browser console to view the error message generated by the error. You'll then fix the error, and verify that the error message does not reappear in the console.

To introduce an intentional error in your code and view an error message in the browser console:

1. Return to **ft.js** in your text editor.
2. At the end of the `resetForm()` function, delete the closing `}`. Function statements must always be enclosed in braces, so removing this brace causes an error when a browser processes the function.
3. Save your changes to **ft.js**, return to the **estimate.htm** document in your browser, and then click the **Reload** or **Refresh** button.

4. Click the **Memory book** check box to check it, click the **Reproduction rights** box to check it, and then click your browser's **Reload** or **Refresh** button. Depending on which browser you're using, the check boxes may remain checked, indicating your `resetForm()` function did not execute as expected. Next you'll open the browser console to view any error messages. Table 2-1 lists the keyboard shortcuts and menu commands to open the console in modern versions of the major browsers.

BROWSER	KEYBOARD SHORTCUT	MENU STEPS
Internet Explorer	F12 , then Ctrl + 2	Click the Tools button, click F12 Developer Tools on the menu, and then in the window that opens, click the Console button.
Firefox	Ctrl + Shift + K (Win) option + command + K (Mac)	Click the Firefox button (Win) or Tools (Mac or Win), point to Web Developer , and then click Web Console .
Chrome	Ctrl + Shift + J (Win) option + command + J (Mac)	Click the Customize and control Google Chrome button, point to Tools , and then click JavaScript console .

Table 2-1: Steps to open the browser console in major browsers

5. Open the console in your browser using the appropriate command from Table 2-1, and then refresh or reload **estimate.htm**.

Note

If your browser isn't listed, or if one of the listed methods doesn't work, check your browser's documentation for the correct steps, or try a different browser. Remember that different browsers may use different terms for the JavaScript console, including "web console" or "error console."

The console indicates a syntax error resulting from a missing `}` after the function. Figure 2-3 shows the browser console in Internet Explorer, Figure 2-4 shows the console in Chrome, and Figure 2-5 shows the Firefox console.

Notice that each console specifies a line number where it identifies an error. IE and Firefox both indicate line 23, which is the start of the next statement after the function, as the place they ran into the error (your line number may differ depending on the number of blank lines in your code). Chrome identifies line 1, which is not particularly helpful in this case. Both IE and Firefox also recognize that the problem stems from a missing brace, while Chrome simply indicates "Unexpected end of input."

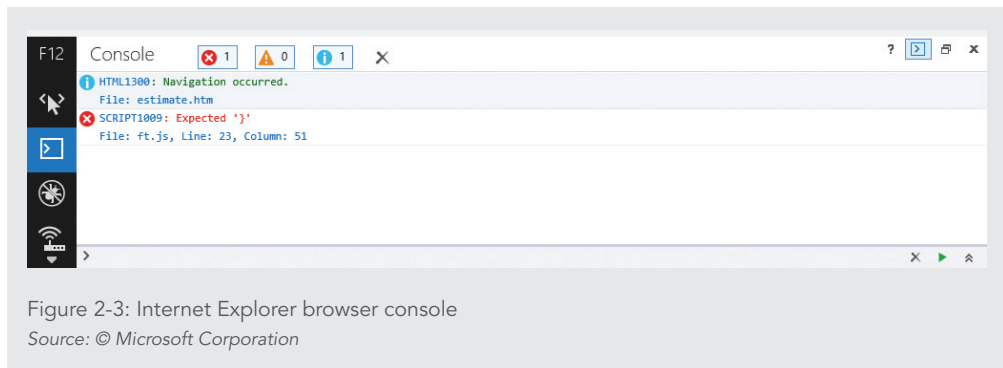


Figure 2-3: Internet Explorer browser console

Source: © Microsoft Corporation

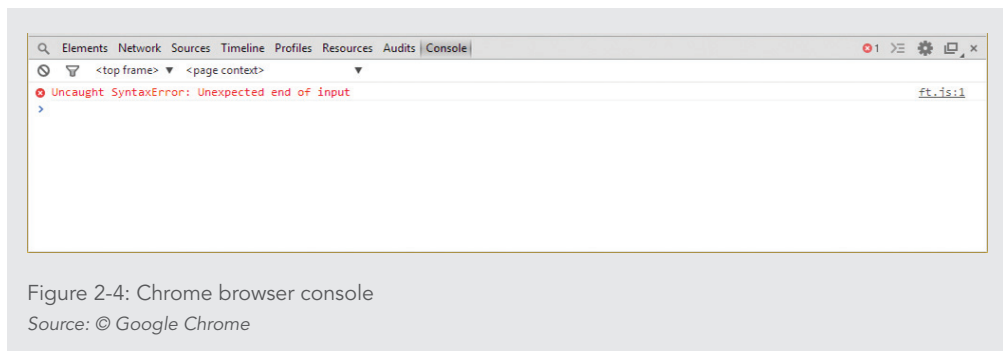


Figure 2-4: Chrome browser console

Source: © Google Chrome

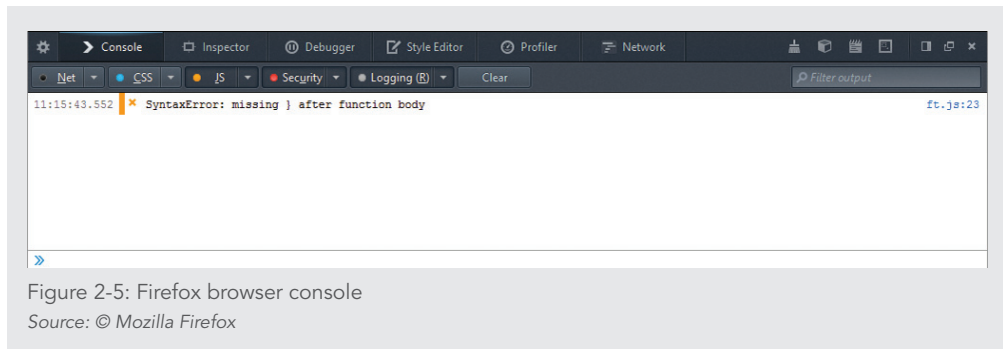


Figure 2-5: Firefox browser console

Source: © Mozilla Firefox

Note

Because different browsers report errors differently, it can sometimes help to check the console in more than one browser to identify a hard-to-find error.

6. Return to **ft.js** in your text editor, and then at the end of the `resetForm()` function, below the statement `document.getElementById("distance").value = 0;`, add a `}` character. This returns your function to its previous state. Your `resetForm()` function should match the one shown in the previous set of steps.
7. Save your changes to **ft.js**, and then in your browser reload or refresh the **estimate.htm** document. The browser console no longer displays an error message.

Note

Don't worry if your console displays other informational messages. If it displays other error messages, though, examine your code for additional errors, and fix them until the console shows no further errors.

Using Return Statements

In many instances, you may want your code to receive the results from a called function and then use those results in other code. For instance, consider a function that calculates the average of a series of numbers that are passed to it as arguments. Such a function would be useless if your code could not print or use the result elsewhere. As another example, suppose that you have created a function that simply prints the name of a student. Now suppose that you want to alter the code so it uses the student name in another section of code. You can return a value from a function to a calling statement by assigning the calling statement to a variable. The following statement calls a function named `averageNumbers()` and assigns the return value to a variable named `returnValue`. The statement also passes three literal values to the function.

```
var returnValue = averageNumbers(1, 2, 3);
```

To actually return a value to the `returnValue` variable, the code must include a return statement within the `averageNumbers()` function. A **return statement** is a statement that returns a value to the statement that called the function. To use a return statement, you use the `return` keyword with the variable or value you want to send to the calling statement. The following script contains the `averageNumbers()` function, which calculates the average of three numbers. The script also includes a return statement that returns the value (contained in the `result` variable) to the calling statement:

```
1  function averageNumbers(a, b, c) {  
2      var sum_of_numbers = a + b + c;  
3      var result = sum_of_numbers / 3;  
4      return result;  
5  }
```

Note

A function does not necessarily have to return a value. For instance, the only purpose of the `resetForm()` function you created earlier in the chapter is to make changes to form values, so it does not need to return a value.

Understanding Variable Scope

When you use a variable in a JavaScript program, particularly a complex JavaScript program, you need to be aware of the **variable scope**—that is, you need to think about where in your code a declared variable can be used. A variable's scope can be either global or local. A **global variable** is one that is declared outside a function and is available to all parts of your code. A **local variable** is declared inside a function and is available only within the function in which it is declared. Local variables cease to exist when a function ends. If you attempt to use a local variable outside the function in which it is declared, browsers log an error message to the console.

Note

The parameters within the parentheses of a function declaration behave like local variables.

You must use the `var` keyword when you declare a local variable. However, when you declare a global variable, the `var` keyword is optional. For example, you can write the statement

```
var myVariable = "This is a variable.";
```

as

```
myVariable = "This is a variable.";
```

If you declare a variable within a function and do not include the `var` keyword, in most cases the variable automatically becomes a global variable. However, it is considered good programming technique to always use the `var` keyword when declaring variables because it indicates where in your code you intend to start using each variable. Also, it is considered poor programming technique to declare a global variable inside of a function by not using the `var` keyword because it makes it harder to identify the global variables in your scripts. Using the `var` keyword forces you to explicitly declare your global variables outside of any functions and local variables within functions.

The following code includes declarations for two global variables, `salesPrice` and `shippingPrice`, which are defined outside of the function. It also contains a declaration for a third variable, `totalPrice`, declared within the function using a `var` statement; this

makes it a local variable. When the function is called, the global variables and the local variable print successfully from within the function. After the call to the function, the global variables again print successfully from the individual statements. However, the statement that tries to print the total price generates an error message because the local variable ceases to exist when the function ends.

```
1  var salesPrice = 100.00;
2  var shippingPrice = 8.95;
3  function applyShipping() {
4      var totalPrice = salesPrice + shippingPrice;
5      document.write(salesPrice); // prints successfully
6      document.write(shippingPrice); // prints successfully
7      document.write(totalPrice); // prints successfully
8  }
9  applyShipping();
10 document.write(salesPrice); // prints successfully
11 document.write(shippingPrice); // prints successfully
12 document.write(totalPrice); // error message
```

To correct this code, you could simply declare the `totalPrice` variable outside of the function without assigning it a value, making it a global variable, and then simply change its value within the function, as follows:

```
1  var salesPrice = 100.00;
2  var shippingPrice = 8.95;
3  var totalPrice;
4  function applyShipping() {
5      totalPrice = salesPrice + shippingPrice;
6  }
...

```

Note

An ellipsis (...) before or after code indicates that the code is a snippet from a larger document.

If code contains a global variable and a local variable with the same name, the local variable takes precedence when its function is called. However, the value assigned to a local variable within a function is not assigned to a global variable of the same name. For example, the following code contains a global variable named `color`, and a local variable named `color`.

The global variable `color` is assigned a value of “green”. Next, the function that contains the local variable `color` is called. This function then assigns the local `color` variable a value of “purple”. However, this has no effect on the value of the global variable `color`. After the function ends, “green” is still the value of the global `color` variable.

```
1  var color = "green";
2  function duplicateVariableNames() {
3      var color = "purple";
4      document.write(color);
5      // value printed is purple
6  }
7  duplicateVariableNames();
8  document.write(color);
9  // value printed is green
```

Figure 2-6 shows the output in a web browser.



purple
green

Figure 2-6: Output of code that contains a global variable and a local variable with the same name

Note that, although the code that displays the output shown in Figure 2-6 is syntactically correct, it is poor programming practice to use the same name for local and global variables because it makes your scripts confusing, and it is difficult to track which version of the variable is currently being used by the code. Instead, you should ensure that all variable names within a related set of scripts are unique.

Next, you will add global variables to the `ft.js` file, for use with `estimate.htm`. The form fields on the `estimate.htm` page allow users to select options for special event photography, and the code you’ll create will total these costs in response to user input. To start creating the program, you’ll create global variables to store the total cost for photographers and the total estimate. You’ll complete the `estimate.htm` page later in this chapter.

To add global variables to the `ft.js` file:

1. Return to the `ft.js` file in your text editor.

2. Below the comment section and above the `resetForm()` function, enter the following statements to define the two global variables:

```
// global variables
var photographerCost = 0;
var totalCost = 0;
```

3. Save your work.

Using Built-in JavaScript Functions

In addition to custom functions that you create yourself, JavaScript allows you to use the built-in functions listed in Table 2-2.

FUNCTION	DESCRIPTION
<code>decodeURI (string)</code>	Decodes text strings encoded with <code>encodeURIComponent ()</code>
<code>decodeURIComponent (string)</code>	Decodes text strings encoded with <code>encodeURIComponent ()</code>
<code>encodeURIComponent (string)</code>	Encodes a text string so it becomes a valid URI
<code>encodeURIComponent (string)</code>	Encodes a text string so it becomes a valid URI component
<code>eval (string)</code>	Evaluates expressions contained within strings
<code>isFinite (number)</code>	Determines whether a number is finite
<code>isNaN (number)</code>	Determines whether a value is the special value NaN (Not a Number)
<code>parseFloat (string)</code>	Converts string literals to floating-point numbers
<code>parseInt (string)</code>	Converts string literals to integers

Table 2-2: Built-in JavaScript functions

In this book, you will examine several of the built-in JavaScript functions as you need them. For now, you just need to understand that you call built-in JavaScript functions in the same way you call custom functions. For example, the following code calls the `isNaN()` function to determine whether the `socialSecurityNumber` variable is *not* a number.

```
var socialSecurityNumber = "123-45-6789";
var checkVar = isNaN(socialSecurityNumber);
document.write(checkVar);
```

Because the Social Security number assigned to the `socialSecurityNumber` variable contains dashes, it is not a true number. Therefore, the `isNaN()` function returns a value of `true` to the `checkVar` variable.

Short Quiz 1

1. What is the difference between a named function and an anonymous function?
2. Why does a named function not execute automatically? How do you execute it?
3. What alternatives exist to specifying an event handler using an HTML attribute?
4. How do you view any error messages that a browser might generate when processing your code?
5. Why is it poor programming practice to declare a global variable inside of a function by not using the `var` keyword?

Working with Data Types

Variables can contain many different kinds of values—for example, the time of day, a dollar amount, or a person’s name. A **data type** is the specific category of information that a variable contains. The concept of data types is often difficult for beginning programmers to grasp because in real life you don’t often distinguish among different types of information. If someone asks you for your name, your age, or the current time, you don’t usually stop to consider that your name is a text string and that your age and the current time are numbers. However, a variable’s specific data type is very important in programming because the data type helps determine how much memory the computer allocates for the data stored in the variable. The data type also governs the kinds of operations that can be performed on a variable.

Data types that can be assigned only a single value are called **primitive types**. JavaScript supports the five primitive data types described in Table 2-3.

DATA TYPE	DESCRIPTION
number	A positive or negative number with or without decimal places, or a number written using exponential notation
Boolean	A logical value of <code>true</code> or <code>false</code>
string	Text such as “Hello World”
undefined	An unassigned, undeclared, or nonexistent value
null	An empty value

Table 2-3: Primitive JavaScript data types

Note


The JavaScript language also supports a more advanced data type, object, which is used for creating a collection of properties. You will learn about the object type in Chapter 7.

Null is a data type as well as a value that can be assigned to a variable. Assigning the value `null` to a variable indicates the variable does not contain a usable value. A variable with a value of `null` has a value assigned to it—`null` is really the value “no value.” You assign the `null` value to a variable when you want to ensure the variable does not contain any data. In contrast, an undefined variable is a variable that has never had a value assigned to it, has not been declared, or does not exist.

The value `undefined` indicates that the variable has never been assigned a value—not even the `null` value. One use for an undefined variable is to determine whether a value is being used by another part of your script. As an example of an undefined variable, the following code declares a variable named `stateTax` without a value. When the second statement uses the `document.write()` method to print the `stateTax` variable, a value of `undefined` is printed because the variable has not yet been assigned a value. The variable is then assigned a value of 40, which is printed to the screen, and then a value of `null`, which is also printed to the screen.

```
1 var stateTax;  
2 document.write(stateTax);  
3 stateTax = 40;  
4 document.write(stateTax);  
5 stateTax = null;  
6 document.write(stateTax);
```

Figure 2-7 shows the output in a web browser.



```
undefined  
40  
null
```

Figure 2-7: Variable assigned values of `undefined` and `null`

Many programming languages require that you declare the type of data that a variable contains. Programming languages that require you to declare the data types of variables are

called **strongly typed** programming languages. A strongly typed language is also known as **statically typed**, because data types do not change after they have been declared. Programming languages that do not require you to declare the data types of variables are called **loosely typed** or **duck typed** programming languages. A loosely typed language is also known as **dynamically typed**, because data types can change after they have been declared. JavaScript is a loosely typed programming language. In JavaScript, you are not required to declare the data type of variables and, in fact, are not allowed to do so. Instead, a JavaScript interpreter automatically determines what type of data is stored in a variable and assigns the variable's data type accordingly. The following code demonstrates how a variable's data type changes automatically each time the variable is assigned a new literal value:

```
1 diffTypes = "Hello World"; // String
2 diffTypes = 8;             // Integer number
3 diffTypes = 5.367;         // Floating-point number
4 diffTypes = true;          // Boolean
5 diffTypes = null;          // Null
```

The next two sections focus on two especially important data types: number and Boolean data types.

Working with Numeric Values

Numeric values are an important part of any programming language and are particularly useful for arithmetic calculations. The number data type in JavaScript supports two types of numeric values: integers and floating-point numbers. An **integer** is a positive or negative number with no decimal places. Integer values in JavaScript can range from -9007199254740990 (-2^{53}) to 9007199254740990 (2^{53}). The numbers -250, -13, 0, 2, 6, 10, 100, and 10000 are examples of integers. The numbers -6.16, -4.4, 3.17, .52, 10.5, and 2.7541 are not integers; they are floating-point numbers because they contain decimal places. A **floating-point number** is a number that contains decimal places or that is written in exponential notation.

Programming Concepts

Exponential Notation

Exponential notation, or **scientific notation**, is a shortened format for writing very large numbers or numbers with many decimal places. Numbers written in exponential notation are represented by a value between 1 and 10 multiplied by 10 raised to some power. The value of 10 is written

Continued on next page...

with an uppercase *E* or lowercase *e*. For example, the number 200,000,000,000 can be written in exponential notation as 2.0e11, which means “two times ten to the eleventh power.” Floating-point values in JavaScript range from approximately $\pm 1.7976931348623157 \times 10^{308}$ to $\pm 5 \times 10^{-324}$. Floating-point values that exceed the largest positive value of $\pm 1.7976931348623157 \times 10^{308}$ result in a special value of `Infinity`. Floating-point values that exceed the smallest negative value of $\pm 5 \times 10^{-324}$ result in a value of `-Infinity`.

The owners of Fan Trick Fine Art Photography want to add a page to their website that explains the basics of digital photography for prospective clients. In addition to basic information about digital photography, they want to include a table comparing the many prefixes used in the metric system, to provide context for the term “megapixel.” Next, you will create a script that uses variables containing integers, floating-point numbers, and exponential numbers to print the prefixes. A metric prefix, or SI prefix, is a name that precedes a metric unit of measure. For example, the metric prefix for centimeter is “centi,” which denotes a value of 1/100th. In other words, a centimeter is the equivalent of 1/100th of a meter. Likewise, “mega” denotes a value of a million, meaning that a megapixel consists of roughly a million pixels.

To create a script that prints metric prefixes:

1. In your text editor, open the **digital.htm** document from the Chapter folder for Chapter 2.
2. Enter your name and today’s date where indicated in the comment section, save the file, and then open **digital.htm** in a browser. The page displays explanatory text, followed by a table listing the prefixes. You’ll add JavaScript statements to write values into the second column.
3. Return to your text editor and then, above the opening `<table>` tag, enter the following script element, which contains variable declarations for the 20 metric prefixes:

```
1  <script>
2      var yotta = 1e24;
3      var zetta = 1e21;
4      var exa = 1e18;
5      var peta = 1e15;
6      var tera = 1e12;
7      var giga = 1e9;
8      var mega = 1e6;
```

```

9      var kilo = 1000;
10     var hecto = 100;
11     var deca = 10;
12     var deci = .1;
13     var centi = .01;
14     var milli = .001;
15     var micro = 1e-6;
16     var nano = 1e-9;
17     var pico = 1e-12;
18     var femto = 1e-15;
19     var atto = 1e-18;
20     var zepto = 1e-21;
21     var yocto = 1e-24;
22  </script>

```

4. Within the `table` element, in the second row, within the empty `td` element, enter the following script section to print the value of the variable that corresponds to the prefix:

```

<script>
    document.write(yotta);
</script>

```

5. In the third row, within the empty `td` element, enter the following script section:

```

<script>
    document.write(zetta);
</script>

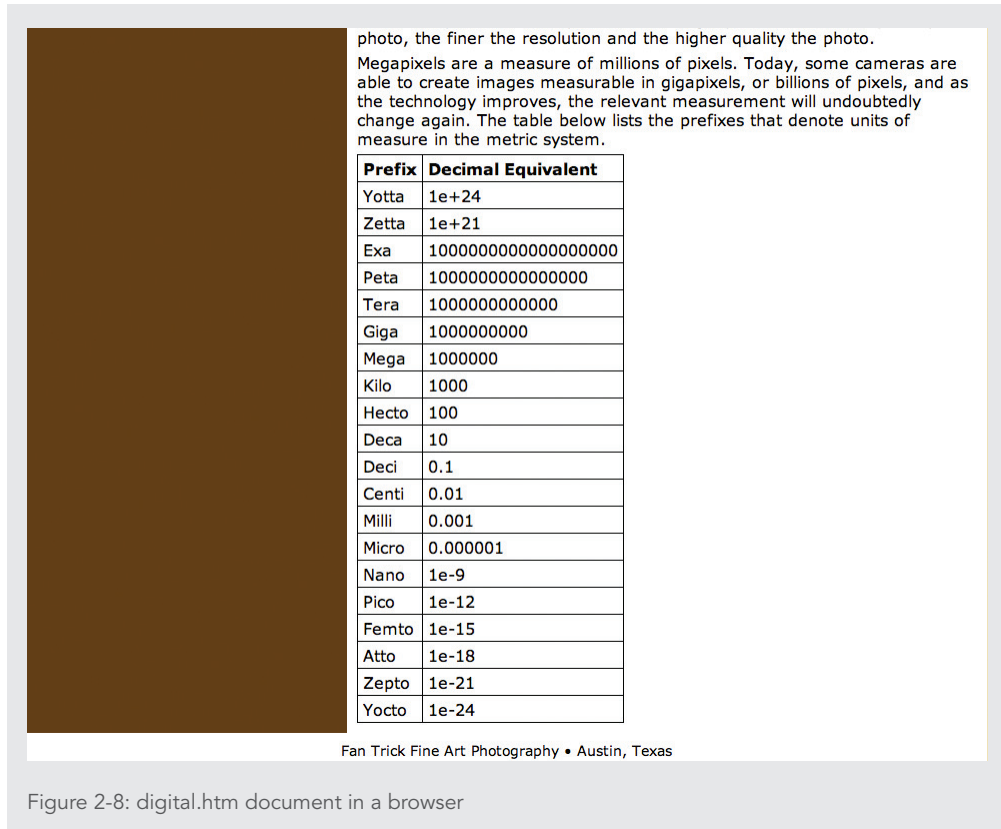
```

6. Repeat Step 5 for the remaining 18 rows of the table, replacing `zetta` with the relevant variable name for each row.

Note

Remember that capitalization counts when referencing variable names, so be sure to enter the variable names in all lowercase to match the variable definitions you created in Step 3.

7. Save your work, and then refresh or reload **digital.htm** in your browser. Figure 2-8 shows how the document looks in a web browser.



Note that browsers automatically decide whether to display a large or small number in exponential notation. Also note that you never had to specify that the variables you declared contained numeric values. Instead, the JavaScript processor in your browser recognized that they contained numeric values and treated them accordingly. This is one of the benefits of JavaScript being a loosely typed language.

8. Close digital.htm in your browser, and then close digital.htm in your text editor.

Best Practices

Calculate with Whole Numbers, Not Decimals

JavaScript treats all numeric values as binary values, rather than as decimals—that is, the numbers are calculated using the two-digit binary system rather than the 10-digit decimal system. While the binary system can accurately represent any value that has a decimal equivalent, when it comes to floating point values, calculations performed on binary representations can result in slightly different results than the same calculations performed on decimal values. Because users enter decimal values in web interfaces and the interfaces display decimal results to users, this discrepancy can cause problems, especially when it comes to calculating exact monetary values such as dollars and cents. JavaScript programmers have developed a straightforward workaround, however: when manipulating a monetary value in a program, first multiply the value by 100, to eliminate the decimal portion of the number. In essence, this means calculating based on a value in cents (for instance, $\$10.51 * 100 = 1051\text{¢}$). Because calculations on integer values are the same in binary and decimal, any calculations you perform will be accurate. When your calculations are finished, simply divide the result by 100 to arrive at the correct, final value in dollars and cents.

Working with Boolean Values

A **Boolean value** is a logical value of `true` or `false`. You can also think of a Boolean value as being yes or no, or on or off. Boolean values are most often used for deciding which code should execute and for comparing data. In JavaScript programming, you can only use the words `true` and `false` to indicate Boolean values. In other programming languages, you can use the integer values of 1 and 0 to indicate Boolean values of `true` and `false`—1 indicates `true` and 0 indicates `false`. The following shows a simple example of two variables that are assigned Boolean values, one `true` and the other `false`.

```
1  var newCustomer = true;
2  var contractorRates = false;
3  document.write("<p>New customer: " + newCustomer + "</p>");
4  document.write("<p>Contractor rates: " + contractorRates +
5      "</p>");
```

Figure 2-9 shows the output in a web browser.

```
New customer: true
Contractor rates: false
```

Figure 2-9: Boolean values

Next, you will add Boolean global variables to the `ft.js` file for the Fan Trick Fine Art Photography site. These variables will determine whether the prospective client wants a memory book or reproduction rights for the photos.

To add two Boolean global variables to the `ft.js` file:

1. Return to the `ft.js` file in your text editor.
2. Below the statement `var totalCost = 0;`, add the following global variables for the memory book and reproduction rights selections:

```
var memoryBook = false;
var reproductionRights = false;
```

3. Within the `resetForm()` function, in the statements that set the values for the checked values of the `membook` and `reprodrights` elements, change the `false` values to the appropriate variable values as follows:

```
document.getElementById("membook").checked = memoryBook;
document.getElementById("reprodrights").checked = reproductionRights;
```

4. Save the `ft.js` file.

Working with Strings

As you learned in Chapter 1, a text string contains zero or more characters surrounded by double or single quotation marks. Examples of strings you may use in a script are company names, usernames, and comments. You can use a text string as a literal value or assign it to a variable.

A literal string can also be assigned a zero-length string value called an **empty string**. For example, the following statement declares a variable named `customerName` and assigns it an empty string:

```
var customerName = "";
```

This simply specifies that the variable is a string variable with no content.

When you want to include a quoted string within a literal string surrounded by double quotation marks, you surround the quoted string with single quotation marks. When you want to include a quoted string within a literal string surrounded by single quotation marks, you

surround the quoted string with double quotation marks. Whichever method you use, a string must begin and end with the same type of quotation marks. For example, you can use either

```
document.write("Boston, MA is called 'Beantown.'")
```

or

```
document.write('Boston, MA is called "Beantown."')
```

Thus, the statement

```
document.write("This is a text string.");
```

is valid, because it starts and ends with double quotation marks, whereas the statement

```
document.write("This is a text string.');
```

is invalid, because it starts with a double quotation mark and ends with a single quotation mark. In the second case, you would receive an error message because a web browser cannot tell where the literal string begins and ends. The following code shows an example of a script that prints strings containing nested quotes.

```
1 document.write("<h1>Speech at the Berlin Wall<←  
2     (excerpt)</h1>");  
3 document.write("<p>Two thousand years ago, the proudest boast<←  
4     was 'civis Romanus sum.'<br />");  
5 document.write('Today, in the world of freedom, the proudest<←  
6     boast is "Ich bin ein Berliner."</p>');  
7 var speaker = "<p>John F. Kennedy</br>";  
8 var date = 'June 26, 1963</p>';  
9 document.write(speaker);  
10 document.write(date);
```

Figure 2-10 shows the output.

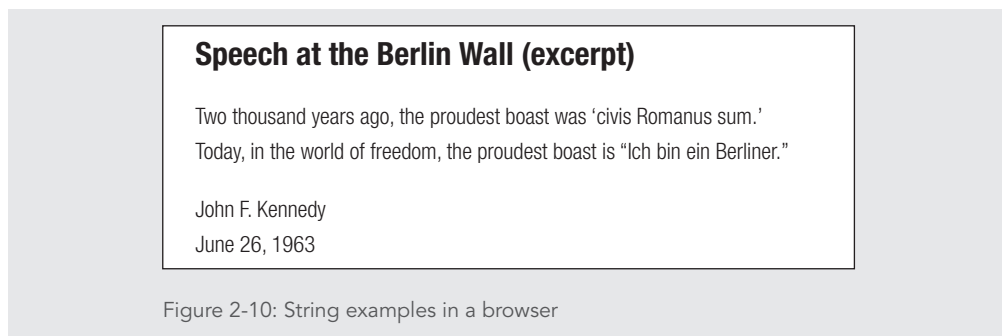


Figure 2-10: String examples in a browser

Note

Unlike other programming languages, JavaScript includes no special data type for a single character, such as the *char* data type in the C, C++, and Java programming languages.

String Operators

JavaScript has two operators that can be used with strings: `+` and `+=`. When used with strings, the plus sign is known as the concatenation operator. The **concatenation operator** (`+`) is used to combine two strings. You have already learned how to use the concatenation operator. For example, the following code combines a string variable and a literal string, and assigns the new value to another variable:

```
var destination = "Honolulu";
var location = "Hawaii";
destination = destination + " is in " + location;
```

The combined value of the location variable and the string literal that is assigned to the destination variable is “Honolulu is in Hawaii”.

You can also use the compound assignment operator (`+=`) to combine two strings. You can think of `+=` as meaning “Set the first operand equal to its current value plus the second operand.” The following code combines the two text strings, but without using the location variable:

```
var destination = "Honolulu";
destination += " is in Hawaii";
```

The second line of this code tells the JavaScript interpreter to combine the current value of the destination variable—“Honolulu”—with the string “is in Hawaii”. The result is the same as in the previous code.

Note that the same symbol—a plus sign—serves as both the concatenation operator and the addition operator. When used with numbers or variables containing numbers, expressions using the concatenation operator return the sum of the two numbers. As you learned earlier in this chapter, if you use the concatenation operator with a string value and a number value, the string value and the number value are combined into a new string value, as in the following example:

```
var textString = "The oldest person ever lived to ";
var oldestAge = 122;
newString = textString + oldestAge;
```

The value of `newString` is “The oldest person ever lived to 122”.

Note that the value of the `textString` variable includes a space before the final quote. When values are concatenated, the last character of the first value is directly followed by the first character of the second value. If the space were not included at the end of the `textString` value, there would be no space between the two values, resulting in a `newString` value of “The oldest person ever lived to122”. (Note the lack of space between “to” and “122”.)

Escape Characters and Sequences

You need to use extra care when using single quotation marks with possessives and contractions in strings, because JavaScript interpreters always look for the first closing single or double quotation mark to match an opening single or double quotation mark. For example, consider the following statement:

```
document.write('<p>My mom's favorite color is blue.</p>');
```

This statement causes an error. A JavaScript interpreter assumes that the literal string ends with the apostrophe following “mom” and looks for the closing parentheses for the `document.write()` statement immediately following “mom”. To get around this problem, you include an escape character before the apostrophe in “mom’s”. An **escape character** tells compilers and interpreters that the character that follows it has a special purpose. In JavaScript, the escape character is the backslash (`\`). Placing a backslash before an apostrophe tells JavaScript interpreters that the apostrophe is to be treated as a regular keyboard character, such as *a*, *b*, *l*, or *2*, and not as part of a single quotation mark pair that encloses a text string. The backslash in the following statement tells the JavaScript interpreter to print the apostrophe following the word “mom” as an apostrophe.

```
document.write('<p>My mom\'s favorite color is blue.</p>');
```

You can also use the escape character in combination with other characters to insert a special character into a string. When you combine the escape character with a specific other character, the combination is called an **escape sequence**. The backslash followed by an apostrophe (`\'`) and the backslash followed by a double quotation mark (`\''`) are both examples of escape sequences. Most escape sequences carry out special functions. For example, the escape sequence `\t` inserts a tab into a string. Table 2-4 describes the escape sequences that can be added to a string in JavaScript.

ESCAPE SEQUENCE	CHARACTER
\\	Backslash
\b	Backspace
\r	Carriage return
\"	Double quotation mark
\f	Form feed
\t	Horizontal tab
\n	Newline
\0	Null character
\'	Single quotation mark (apostrophe)
\v	Vertical tab
\xXX	Latin-1 character specified by the XX characters, which represent two hexadecimal digits
\uXXXX	Unicode character specified by the XXXX characters, which represent four hexadecimal digits

Table 2-4: JavaScript escape sequences

Note

If you place a backslash before any character other than those listed in Table 2-4, the backslash is ignored.

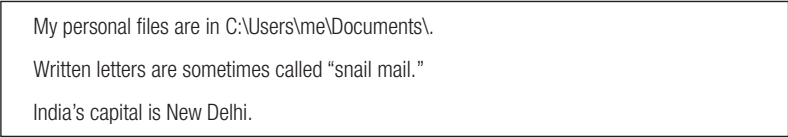
Notice that one of the characters generated by an escape sequence is the backslash. Because the escape character itself is a backslash, you must use the escape sequence \\ to include a backslash as a character in a string. For example, to include the path “C:\Users\me\Documents\Cengage\WebWarrior\JavaScript\” in a string, you must include two backslashes for every single backslash you want to appear in the string, as in the following statement:

```
document.write("<p>My JavaScript files are located in  
C:\\Users\\me\\Documents\\Cengage\\WebWarrior\\  
JavaScript\\</p>");
```

The following code shows an example of a script containing strings with several escape sequences.

```
1  <script>
2  document.write("<p>My personal files are in↵
3      C:\\Users\\me\\Documents\\.</p>"); // Backslash
4  document.write(("<p>Written letters are sometimes called↵
5      \"snail mail.\"</p>")); // Double quotation mark
6  document.write('<p>India\'s capital is New Delhi.</p>');
7      // Single quotation mark
8  </script>
```

Figure 2-11 shows the output.



My personal files are in C:\Users\me\Documents\
Written letters are sometimes called "snail mail."
India's capital is New Delhi.

Figure 2-11: Output of script with strings containing escape sequences

Short Quiz 2

1. What is the difference between an integer and a floating-point number?
2. Which possible values can a Boolean variable have?
3. What is an empty string?
4. Why do you sometimes need to insert an extra space in a string when using the concatenation operator?

Using Operators to Build Expressions

In Chapter 1, you learned the basics of how to create expressions using basic operators, such as the addition operator (+) and multiplication operator (*). In this section, you will learn about additional types of operators you can use with JavaScript. Table 2-5 lists the operator types that you can use with JavaScript.

OPERATOR TYPE	OPERATORS	DESCRIPTION
Arithmetic	addition (+) subtraction (-) multiplication (*) division (/) modulus (%) increment (++) decrement (--) negation (-)	Perform mathematical calculations
Assignment	assignment (=) compound addition assignment (+=) compound subtraction assignment (-=) compound multiplication assignment (*=) compound division assignment (/=) compound modulus assignment (%=)	Assign values to variables
Comparison	equal (==) strict equal (===) not equal (!=) strict not equal (!==) greater than (>) less than (<) greater than or equal (>=) less than or equal (<=)	Compare operands and return a Boolean value
Logical	And (&&) Or () Not (!)	Perform Boolean operations on Boolean operands
String	concatenation (+) compound concatenation assignment (+=)	Perform operations on strings

Continued on next page...

OPERATOR TYPE	OPERATORS	DESCRIPTION
Special	property access (.) array index ([]) function call (()) comma (,) conditional expression (? :) delete (delete) property exists (in) object type (instanceof) new object (new) data type (typeof) void (void)	Various purposes; do not fit within other operator categories

Table 2-5: JavaScript operator types

JavaScript operators are binary or unary. A **binary operator** requires an operand before and after the operator. The equal sign in the statement `myNumber = 100;` is an example of a binary operator. A **unary operator** requires just a single operand either before or after the operator. For example, the increment operator (`++`), an arithmetic operator, is used to increase an operand by a value of one. The statement `myNumber++;` changes the value of the `myNumber` variable to 101.

In the following sections, you will learn more about the different types of JavaScript operators.

Note

Another type of JavaScript operator, bitwise operators, operate on integer values; this is a fairly complex topic. Bitwise operators and other complex operators are beyond the scope of this book.

Arithmetic Operators

Arithmetic operators are used in JavaScript to perform mathematical calculations, such as addition, subtraction, multiplication, and division. You can also use an arithmetic operator to return the modulus of a calculation, which is the remainder left when you divide one number by another number.

Arithmetic Binary Operators

JavaScript binary arithmetic operators and their descriptions are listed in Table 2-6.

NAME	OPERATOR	DESCRIPTION
Addition	+	Adds two operands
Subtraction	-	Subtracts one operand from another operand
Multiplication	*	Multiplies one operand by another operand
Division	/	Divides one operand by another operand
Modulus	%	Divides one operand by another operand and returns the remainder

Table 2-6: Arithmetic binary operators

Note

The operand to the left of an operator is known as the left operand, and the operand to the right of an operator is known as the right operand.

The following code shows examples of expressions that include arithmetic binary operators.

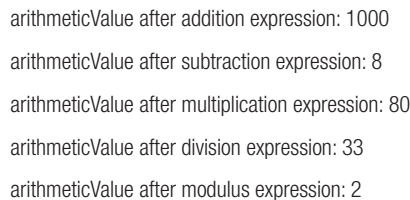
```
1  var x = 0, y = 0, arithmeticValue = 0;
2  // ADDITION
3  x = 400;
4  y = 600;
5  arithmeticValue = x + y; // arithmeticValue changes to 1000
6  document.write("<p>arithmeticValue after addition<br>
7    expression: " + arithmeticValue + "</p>");
8  // SUBTRACTION
9  x = 14;
10 y = 6;
11 arithmeticValue = x - y; // arithmeticValue changes to 8
12 document.write("<p>arithmeticValue after subtraction<br>
13   expression: " + arithmeticValue + "</p>");
14 // MULTIPLICATION
15 x = 20;
16 y = 4;
17 arithmeticValue = x * y; // arithmeticValue changes to 80
```

```

18 document.write("<p>arithmeticValue after multiplication<
19     expression: " + arithmeticValue + "</p>");
20 // DIVISION
21 x = 99;
22 y = 3;
23 arithmeticValue = x / y; // arithmeticValue changes to 33
24 document.write("<p>arithmeticValue after division<
25     expression: " + arithmeticValue + "</p>");
26 // MODULUS
27 x = 5;
28 y = 3;
29 arithmeticValue = x % y; // arithmeticValue changes to 2
30 document.write("<p>arithmeticValue after modulus<
31     expression: " + arithmeticValue + "</p>");

```

Figure 2-12 shows how the expressions appear in a web browser:



```

arithmeticValue after addition expression: 1000
arithmeticValue after subtraction expression: 8
arithmeticValue after multiplication expression: 80
arithmeticValue after division expression: 33
arithmeticValue after modulus expression: 2

```

Figure 2-12: Results of arithmetic expressions

Notice in the preceding code that when JavaScript performs an arithmetic calculation, it performs the operation on the right side of the assignment operator, and then assigns the value to the variable on the left side of the assignment operator. For example, in the statement `arithmeticValue = x + y;` on line 5, the operands `x` and `y` are added, then the result is assigned to the `arithmeticValue` variable on the left side of the assignment operator.

You may be confused by the difference between the division (`/`) operator and the modulus (`%`) operator. The division operator performs a standard mathematical division operation. For example, dividing 15 by 6 results in a value of 2.5. By contrast, the modulus operator returns the remainder that results from the division of two integers. The following code, for instance, uses the division and modulus operators to return the result of dividing 15 by 6. The division of 15 by 6 results in a value of 2.5, because 6 goes into 15 exactly 2.5 times. But if you allow

only for whole numbers, 6 goes into 15 only 2 times, with a remainder of 3 left over. Thus the modulus of 15 divided by 6 is 3, because 3 is the remainder left over following the division.

```
1  var divisionResult = 15 / 6;
2  var modulusResult = 15 % 6;
3  document.write("<p>15 divided by 6 is " +
4      + divisionResult + ".</p>"); // prints '2.5'
5  document.write("<p>The whole number 6 goes into 15 twice,
6      with a remainder of " + modulusResult + ".</p>");
7      // prints '3'
```

Figure 2-13 shows the output.



15 divided by 6 is 2.5.

The whole number 6 goes into 15 twice, with a remainder of 3.

Figure 2-13: Division and modulus expressions

You can include a combination of variables and literal values on the right side of an assignment statement. For example, any of the following addition statements are correct:

```
arithmeticValue = 250 + y;
arithmeticValue = x + 425;
arithmeticValue = 250 + 425;
```

However, you cannot include a literal value as the left operand because the JavaScript interpreter must have a variable to which to assign the returned value. Therefore, the statement `362 = x + y;` causes an error.

When performing arithmetic operations on string values, a JavaScript interpreter attempts to convert the string values to numbers. The variables in the following example are assigned as string values instead of numbers because they are contained within quotation marks. Nevertheless, JavaScript interpreters correctly perform the multiplication operation and return a value of 20.

```
x = "4";
y = "5";
arithmeticValue = x * y; // the value returned is 20
```

However, JavaScript interpreters do not convert strings to numbers when you use the addition operator. When you use the addition operator with strings, the strings are combined instead of being added together. In the following example, the operation returns a string value of “54” because the `x` and `y` variables contain strings instead of numbers:

```
x = "5";
y = "4";
arithmeticValue = x + y; // a string value of 54 is returned
```

Arithmetic Unary Operators

Arithmetic operations can also be performed on a single variable using unary operators. Table 2-7 lists the arithmetic unary operators available in JavaScript.

NAME	OPERATOR	DESCRIPTION
Increment	++	Increases an operand by a value of one
Decrement	--	Decreases an operand by a value of one
Negation	-	Returns the opposite value (negative or positive) of an operand

Table 2-7: Arithmetic unary operators

The increment (++) and decrement (--) unary operators can be used as prefix or postfix operators. A **prefix operator** is placed before a variable name. A **postfix operator** is placed after a variable name. The operands ++count and count++ both increase the count variable by one. However, the two statements return different values. When you use the increment operator as a prefix operator, the value of the operand is returned *after* it is increased by a value of one. When you use the increment operator as a postfix operator, the value of the operand is returned *before* it is increased by a value of one. Similarly, when you use the decrement operator as a prefix operator, the value of the operand is returned *after* it is decreased by a value of one, and when you use the decrement operator as a postfix operator, the value of the operand is returned *before* it is decreased by a value of one. If you intend to assign the incremented or decremented value to another variable, then whether you use the prefix or postfix operator makes a difference.

You use arithmetic unary operators in any situation in which you want to use a more simplified expression for increasing or decreasing a value by 1. For example, the statement `count = count + 1;` is identical to the statement `++count;`. As you can see, if your goal is only to increase a variable by 1, then it is easier to use the unary increment operator.

Skills at Work | Making Your Code Self-Documenting

While including comments in your code can make your code easier for you and other programmers to read and understand, using comments is not the only strategy for documenting code. Another strategy is to make your code self-documenting, meaning that the code is written as simply and directly as possible, so its statements and structures are easier to understand at a glance. For instance, it's a good idea to name variables with descriptive names so it's easier to remember what value is stored in each variable. Rather than naming variables `var1`, `var2`, and so on, you can use names like `firstName` or `lastName`. Creating statements that are easy to read is another instance of self-documenting code. For example, code that you create with the increment operator (`++`) is more concise than code that assigns a variable a value of itself plus 1. However, in this shortened form, the code can be more challenging to read and understand quickly. For this reason, some developers choose not to use the `++` operator at all, relying instead on the `+` and `=` operators to make their code easier for themselves and other developers to read. As you build your programming skills, you'll learn multiple ways to code many different tasks, but the best option is usually the one that results in code that is easy for you and other programmers to read and understand. This results in code that can be more easily maintained and modified by other developers when you move on to another job.


Remember that, with the prefix operator, the value of the operand is returned *after* it is increased or decreased by a value of 1. By contrast, with the postfix operator, the value of the operand is returned *before* it is increased or decreased by a value of 1.

For an example of when you would use the prefix operator or the postfix operator, consider an integer variable named `studentID` that is used for assigning student IDs in a class registration script. One way of creating a new student ID number is to store the last assigned student ID in the `studentID` variable. When it's time to assign a new student ID, the script could retrieve the last value stored in the `studentID` variable and then increase its value by 1. In other words, the last value stored in the `studentID` variable will be the next number used for a student ID number. In this case, you would use the postfix operator to return the value of the expression *before* it is incremented by using a statement similar to `currentID = studentID++`. If you are storing the last assigned student ID in the `studentID` variable, you would want to increment the value by 1 and use the result as the next student ID. In this scenario, you would use the prefix operator, which returns the value of the expression *after* it is incremented using a statement similar to `currentID = ++studentID`.

The following code uses the prefix increment operator to assign three student IDs to a variable named `curStudentID`. The initial student ID is stored in the `studentID` variable and initialized to a starting value of 100.

```
1  var studentID = 100;
2  var curStudentID;
3  curStudentID = ++studentID; // assigns '101'
4  document.write("<p>The first student ID is " +
5      curStudentID + "</p>");
6  curStudentID = ++studentID; // assigns '102'
7  document.write("<p>The second student ID is " +
8      curStudentID + "</p>");
9  curStudentID = ++studentID; // assigns '103'
10 document.write("<p>The third student ID is " +
11     curStudentID + "</p>");
```

Figure 2-14 shows the output.



The first student ID is 101
The second student ID is 102
The third student ID is 103

Figure 2-14: Output of the prefix version of the student ID script


The code below performs the same tasks, but using a postfix increment operator. Notice that the output in Figure 2-15 differs from the output in Figure 2-14. Because the first example of the script uses the prefix increment operator, which increments the `studentID` variable *before* it is assigned to `curStudentID`, the script does not use the starting value of 100. Rather, it first increments the `studentID` variable and uses 101 as the first student ID. By contrast, the second example of the script does use the initial value of 100 because the postfix increment operator increments the `studentID` variable *after* it is assigned to the `curStudentID` variable.

```
1  var studentID = 100;
2  var curStudentID;
3  curStudentID = studentID++; // assigns '100'
```

```

4  document.write("<p>The first student ID is " +
5      curStudentID + "</p>");
6  curStudentID = studentID++; // assigns '101'
7  document.write("<p>The second student ID is " +
8      curStudentID + "</p>");
9  curStudentID = studentID++; // assigns '102'
10 document.write("<p>The third student ID is " +
11     curStudentID + "</p>");

```



The first student ID is 100
The second student ID is 101
The third student ID is 102

Figure 2-15: Output of the postfix version of the student ID script

Assignment Operators

An **assignment operator** is used for assigning a value to a variable. You have already used the most common assignment operator, the equal sign (=), to assign values to variables you declared using the `var` statement. The equal sign assigns an initial value to a new variable or assigns a new value to an existing variable. For example, the following code creates a variable named `favoriteColor`, uses the equal sign to assign it an initial value, then uses the equal sign again to assign it a new value:

```

var favoriteColor = "blue";
favoriteColor = "yellow";

```

JavaScript includes other assignment operators in addition to the equal sign. Each of these additional assignment operators, called **compound assignment operators**, performs mathematical calculations on variables and literal values in an expression, and then assigns a new value to the left operand. Table 2-8 displays a list of the common JavaScript assignment operators.

NAME	OPERATOR	DESCRIPTION
Assignment	=	Assigns the value of the right operand to the left operand
Compound addition assignment	+=	Combines the value of the right operand with the value of the left operand (if the operands are strings), or adds the value of the right operand to the value of the left operand (if the operands are numbers), and assigns the new value to the left operand
Compound subtraction assignment	-=	Subtracts the value of the right operand from the value of the left operand, and assigns the new value to the left operand
Compound multiplication assignment	*=	Multiplies the value of the right operand by the value of the left operand, and assigns the new value to the left operand
Compound division assignment	/=	Divides the value of the left operand by the value of the right operand, and assigns the new value to the left operand
Compound modulus assignment	%=	Divides the value of the left operand by the value of the right operand, and assigns the remainder (the modulus) to the left operand

Table 2-8: Assignment operators

You can use the += compound addition assignment operator to combine two strings as well as to add numbers. In the case of strings, the string on the left side of the operator is combined with the string on the right side of the operator, and the new value is assigned to the left operand. Before combining operands, a JavaScript interpreter attempts to convert a nonnumeric operand, such as a string, to a number. This means that unlike the + operator, which concatenates any string values, using the += operator with two string operands containing numbers results in a numeric value. For instance, the following code defines two variables with string values: `x` with a value of “5” and `y` with a value of “4”. In processing the third line of code, a JavaScript interpreter first converts the strings to the values 5 and 4, respectively, then adds them, and then assigns the result, the value 9, to the `x` variable.

```
x = "5";
y = "4";
x += y; // a numeric value of 9 is returned for x
```

If a nonnumeric operand cannot be converted to a number, you receive a value of `NaN`. The value `NaN` stands for “Not a Number” and is returned when a mathematical operation does not result in a numerical value. The sole exception to this rule is +=, which simply concatenates operands if one or both can’t be converted to numbers.

The following code shows examples of the different assignment operators:

```
1  var x, y;
2  // += operator with string values
3  x = "Hello ";
4  x += "World"; // x changes to "Hello World"
5  // += operator with numeric values
6  x = 100;
7  y = 200;
8  x += y; // x changes to 300
9  // -= operator
10 x = 10;
11 y = 7;
12 x -= y; // x changes to 3
13 // *= operator
14 x = 2;
15 y = 6;
16 x *= y; // x changes to 12
17 // /= operator
18 x = 24;
19 y = 3;
20 x /= y; // x changes to 8
21 // %= operator
22 x = 3;
23 y = 2;
24 x %= y; // x changes to 1
25 // *= operator with a number and a convertible string
26 x = "100";
27 y = 5;
28 x *= y; // x changes to 500
29 // *= operator with a number and a nonconvertible string
30 x = "one hundred";
31 y = 5;
32 x *= y; // x changes to NaN
```

Next, you will add functions to the `ft.js` file to calculate the cost of hiring photography staff for an event. The calculation involves multiplying the number of staff by an hourly rate of \$100 per photographer and by the total number of hours.

To modify the `ft.js` file to calculate the cost of hiring photography staff:

1. Return to the `ft.js` file in your text editor.
2. Above the `resetForm()` function you created earlier, add the following code to define the `calcStaff()` function:

```
// calculates all costs based on staff and adds to total cost
function calcStaff() {
}
```

3. Within the braces for the `calcStaff()` function, enter the following statements:

```
var num = document.getElementById("photognum");
var hrs = document.getElementById("photoghrs");
```

Each of these statements uses a variable to store a reference to an element in the web document. Using these variables will make the code for your calculations shorter and easier to understand at a glance.

4. Within the function, below the `var` statements you just entered, add the following statement:

```
totalCost -= photographerCost;
```

This statement uses the compound subtraction assignment operator to subtract the current value of the `photographerCost` variable from the value of the `totalCost` variable, and then assign the result as the new value of the `totalCost` variable. This results in a `totalCost` value that includes no `photographerCost` value. In the following steps, you'll add code to calculate the new `photographerCost` value and then add it back to the `totalCost` value.

5. Within the function, below the statement you just entered, add the following statement:

```
photographerCost = num.value * 100 * hrs.value;
```

This statement uses the variables you created in Step 3 that reference the input elements with the ids `photognum` and `photoghrs`. It multiplies the value entered for # of photographers (`num.value`) by 100 (the fixed hourly rate per photographer) and by the value entered for # of hours to photograph (`hrs.value`), and then assigns the result to the `photographerCost` variable.

6. Within the function, below the statement you just entered, add the following statement:

```
totalCost += photographerCost;
```

This statement uses the compound addition assignment operator to add the current value of the `photographerCost` variable to the value of the `totalCost` variable, and then assign the result as the new value of the `totalCost` variable. This reverses what the first statement in the function did, adding back the value of `photographerCost` into the calculation of `totalCost`.

7. Within the function, below the statement you just entered, add the following statement:

```
document.getElementById("estimate").innerHTML = "$" +  
    totalCost;
```

This statement uses the `getElementById()` method of the `Document` object to locate the element with the `id` value of `estimate`, and then assigns a value to that element's `innerHTML` property. The value of an element's **innerHTML property** is the content between its opening and closing tags. By assigning a new value to an element's `innerHTML` property, you replace any existing content. The value assigned by this statement is the `$` character followed by the value of the `totalCost` variable. Your `calcStaff()` function should match the following:

```
1  // calculates all costs based on staff and adds to total cost  
2  function calcStaff() {  
3      var num = document.getElementById("photognum");  
4      var hrs = document.getElementById("photoghrs");  
5      totalCost -= photographerCost;  
6      photographerCost = num.value * 100 * hrs.value;  
7      totalCost += photographerCost;  
8      document.getElementById("estimate").innerHTML = "$" +  
9          totalCost;  
10 }
```

8. Scroll down to the `resetForm()` function you created earlier, and then, just above the closing brace `}`, add the following statement:

```
calcStaff();
```

This statement calls the `calcStaff()` function whenever the `resetForm()` function runs. Because `resetForm()` is called each time the page loads, adding this statement

will replace the empty value of the estimate element with the `totalCost` value calculated based on the default form values.

9. Below the closing `}` for the `resetForm()` function, enter the following code to create the `createEventListeners()` function:

```
// creates event listeners
function createEventListeners() {
}
```

To implement the `calcStaff()` function you just created, you have to add code to create event listeners.

10. Within the command block for the `createEventListeners()` function, enter the following two statements to add event listeners to the elements with the `id` values `photognum` and `photoghrs`:

```
1 document.getElementById("photognum").  
2   addEventListener("change", calcStaff, false);  
3 document.getElementById("photoghrs").  
4   addEventListener("change", calcStaff, false);
```

The two event listeners will call the `calcStaff()` function each time a user changes the values for # of photographers or # of hours to photograph.

11. Scroll up to the `resetForm()` function, and then, just before the closing `}`, enter the following statement:

```
createEventListeners();
```

This calls the `createEventListeners()` function when the page loads. Your updated `resetForm()` function should match the following:

```
1 function resetForm() {  
2   document.getElementById("photognum").value = 1;  
3   document.getElementById("photoghrs").value = 2;  
4   document.getElementById("membook").checked = memoryBook;  
5   document.getElementById("reprodrights").checked =  
6   reproductionRights;  
7   document.getElementById("distance").value = 0;  
8   calcStaff();  
9   createEventListeners();  
10 }
```

12. Save your changes to **ft.js**, and then in your browser, refresh or reload **estimate.htm**.
13. Change the value in the # of photographers box to 2, press **Tab**, change the value in the # of hours to photograph box to 3, and then press **Tab**. Pressing Tab fires the `change` event for an `input` element. Notice that the Total Estimate value in the sidebar is calculated automatically when the page reloads, and changes after each time you press Tab—first to \$400 and then to \$600.

Note

All modern browsers support the `textContent` property for web page elements. This property is similar to the `innerHTML` property, except that a `textContent` value excludes any HTML markup, while `innerHTML` includes it. Using `textContent` results in JavaScript code that's more secure. However, Internet Explorer 8 supports only `innerHTML`, not `textContent`. All the sites in this book are designed to work with IE8, so they use `innerHTML`. However, any site you create that doesn't need to support IE8 should use `textContent` instead of `innerHTML`.

Comparison and Conditional Operators

A **comparison operator**, or **relational operator**, is used to compare two operands and determine if one value is greater than another. A Boolean value of `true` or `false` is returned after two operands are compared. For example, the statement `5 < 3` would return a Boolean value of `false`, because 5 is not less than 3. Table 2-9 lists the JavaScript comparison operators.

NAME	OPERATOR	DESCRIPTION
Equal	<code>==</code>	Returns true if the operands are equal
Strict equal	<code>===</code>	Returns true if the operands are equal and of the same type
Not equal	<code>!=</code>	Returns true if the operands are not equal
Strict not equal	<code>!==</code>	Returns true if the operands are not equal or not of the same type
Greater than	<code>></code>	Returns true if the left operand is greater than the right operand
Less than	<code><</code>	Returns true if the left operand is less than the right operand
Greater than or equal	<code>>=</code>	Returns true if the left operand is greater than or equal to the right operand
Less than or equal	<code><=</code>	Returns true if the left operand is less than or equal to the right operand

Table 2-9: Comparison operators

Note

The comparison operators (`==` and `===`) consist of two and three equal signs, respectively, and perform a different function than the one performed by the assignment operator that consists of a single equal sign (`=`). The comparison operators compare values, whereas the assignment operator assigns values. Confusion between these two types of operators is a common source of bugs, and you should check for it if your code compares values and is not delivering expected results.

You can use number or string values as operands with comparison operators. When two numeric values are used as operands, JavaScript interpreters compare them numerically. For example, the statement `arithmeticValue = 5 > 4;` results in `true` because the number 5 is numerically greater than the number 4. When two nonnumeric values are used as operands, the JavaScript interpreter compares them in lexicographical order—that is, the order in which they would appear in a dictionary. The statement `arithmeticValue = "b" > "a";` returns `true` because the letter *b* comes after than the letter *a* in the dictionary. When one operand is a number and the other is a string, JavaScript interpreters attempt to convert the string value to a number. If the string value cannot be converted to a number, a value of `false` is returned. For example, the statement `arithmeticValue = 10 === "ten";` returns a value of `false` because JavaScript interpreters cannot convert the string “ten” to a number.

Note

The comparison operators `==` and `===` perform virtually the same function. However, for reasons that are beyond the scope of this book, using the `==` operator can occasionally produce unexpected results. The same is true of the `!=` and `!==` operators. For this reason, the code in this book uses `===` and `!==` unless otherwise noted.

The comparison operator is often used with another kind of operator, the conditional operator. The **conditional operator** executes one of two expressions, based on the results of a conditional expression. The syntax for the conditional operator is

```
conditional expression ? expression1 : expression2;
```

If the conditional expression evaluates to `true`, then `expression1` executes. If the conditional expression evaluates to `false`, then `expression2` executes.

The following code shows an example of the conditional operator.

```
1  var intVariable = 150;
2  var result;
3  intVariable > 100 ? ↵
4      result = "intVariable is greater than 100" : ↵
5      result = "intVariable is less than or equal to 100";
6  document.write(result);
```

In line 3 of the example, the conditional expression checks to see if the `intVariable` variable is greater than 100. If `intVariable` is greater than 100, then the text “intVariable is greater than 100” is assigned to the `result` variable. If `intVariable` is not greater than 100, then the text “intVariable is less than or equal to 100” is assigned to the `result` variable. Because `intVariable` is equal to 150, the conditional statement returns a value of `true`, the first expression executes, and “intVariable is greater than 100” prints to the screen.

When a term in a conditional operator is particularly long or complex, you can add parentheses around it. This provides visual clarification of where the term starts and ends without changing its meaning.

Next, you will create functions for the estimate form that change the calculated estimate value based on whether a prospective client wants a memory book and photo reproduction rights. You’ll use conditional operators in your functions to determine whether to add or subtract the cost of each item.

To create functions for the estimate form that add the cost of a memory book and reproduction rights:

1. Return to the `ft.js` file in your text editor.
2. Below the closing `}` for the `calcStaff()` function, add the following code to create the new `toggleMembook()` function:

```
// adds/subtracts cost of memory book from total cost
function toggleMembook() {
}
```

3. Within the braces for the `toggleMembook()` function, enter the following statement:

```
(document.getElementById("membook").checked === false) ? ↵
    totalCost -= 250 : totalCost += 250;
```

This statement looks up the `checked` property of the element with the `id` of `membook`. For an input element with a `type` attribute set to `checkbox` or `radio`,

the checked property is true if the element is selected, and false if it is not selected. If the checked value is false, the conditional expression subtracts the charge for a memory book, \$250, from the totalCost value. If the checked value is true, the conditional expression adds \$250 to the totalCost value.

4. Within the toggleMembook() function, below the statement you just entered, add the following statement:

```
document.getElementById("estimate").innerHTML =  
"$" + totalCost;
```

This statement changes the innerHTML value of the element with the id of estimate to a \$ symbol followed by the current value of the totalCost variable.

5. Below the toggleMembook() function, add the following code to create the new toggleRights() function:

```
// adds/subtracts cost of reproduction rights from total cost  
function toggleRights() {  
}
```

6. Within the braces for the toggleRights() function, enter the following statement:

```
(document.getElementById("reprodrights").checked === false) ?  
totalCost -= 1250 : totalCost += 1250;
```

This statement is similar to the conditional statement for the toggleMembook() function. It subtracts \$1250 from the totalCost variable if the input element with the id of reprodrights is unchecked, and adds that amount if the element is checked.

7. Within the toggleRights() function, below the statement you just entered, add the following statement:

```
document.getElementById("estimate").innerHTML =  
"$" + totalCost;
```

The following code shows the completed toggleMembook() and toggleRights() functions:

```
1 // adds/subtracts cost of memory book from total cost  
2 function toggleMembook() {  
3     (document.getElementById("membook").checked === false) ?  
4         totalCost -= 250 : totalCost += 250;
```

```

5     document.getElementById("estimate").innerHTML +=
6         "$" + totalCost;
7 }
8 // adds/subtracts cost of reproduction rights from total cost
9 function toggleRights() {
10     (document.getElementById("reprodrights").checked ===
11         false) ? totalCost -= 1250 : totalCost += 1250;
12     document.getElementById("estimate").innerHTML +=
13         "$" + totalCost;
14 }

```

8. Scroll down to the `createEventListeners()` function, and then before the closing `}`, enter the following two statements to create event listeners for the elements with the `id` values `membook` and `reprodrights`:

```

1 document.getElementById("membook").
2     addEventListener("change", toggleMembook, false);
3 document.getElementById("reprodrights").
4     addEventListener("change", toggleRights, false);

```

9. Save your changes to the **ft.js** file, and then switch to **estimate.htm** in your text editor.
10. Save your work, switch to your browser, refresh or reload **estimate.htm**, click the **Memory book** check box to check it, and then click the **Reproduction rights box** to check it. Notice that the Total Estimate value in the sidebar changes each time you select a box, first to \$450 and then to \$1700.
11. Click the **Memory book** check box to uncheck it, and then click the **Reproduction rights box** to uncheck it. The Total Estimate value is reduced each time you uncheck a box, first to \$1450, and then to \$200.

Understanding Falsy and Truthy Values

JavaScript includes six values that are treated in comparison operations as the Boolean value `false`. These six values, known as **falsy values**, are

```

> ""
> -0
> 0
> NaN

```

```
> null
> undefined
```

All values other than these six falsy values are the equivalent of Boolean `true`, and are known as **truthy values**.

Developers commonly take advantage of falsy and truthy values to make comparison operations more compact. For instance, suppose you were writing a conditional statement that checks if a text field in a form contains a value. You could write the code for the conditional statement as

```
(document.getElementById("fname").value !== "") ? ␣
// code to run if condition is true : ␣
// code to run if condition is false;
```

However, another way to approach this is to simply check whether the value of the text field is falsy (an empty string) or truthy (anything else). You can do this by omitting the comparison operator and simplifying the code as follows:

```
(document.getElementById("fname").value) ? ␣
// code to run if condition is true : ␣
// code to run if condition is false;
```

Note that the revised code replaces the comparison in the first line with only a reference to the web page element. If this element is empty, then its value is `"` (an empty string), which is a falsy value, in which case the conditional expression evaluates to `false`. However, if the element is not empty, then the value is truthy, and the conditional expression evaluates to `true`.

Logical Operators

Logical operators are used to modify Boolean values or specify the relationship between operands in an expression that results in a Boolean value. For example, a script for an automobile insurance company may need to determine whether a customer is male *and* under 21 in order to determine the correct insurance quote. As with comparison operators, a Boolean value of `true` or `false` is returned after two operands are evaluated. Table 2-10 lists the JavaScript logical operators.

The Or (`||`) and the And (`&&`) operators are binary operators (requiring two operands), whereas the Not (`!`) operator is a unary operator (requiring a single operand). Logical operators are often used with comparison operators to evaluate expressions, allowing you to combine the results of several expressions into a single statement. For example, the And (`&&`) operator is used for determining whether two operands return an equivalent value.

NAME	OPERATOR	DESCRIPTION
And	<code>&&</code>	Returns <code>true</code> if both the left operand and right operand return a value of <code>true</code> ; otherwise, it returns a value of <code>false</code>
Or	<code> </code>	Returns <code>true</code> if either the left operand or right operand returns a value of <code>true</code> ; if neither operand returns a value of <code>true</code> , then the expression containing the Or <code> </code> operator returns a value of <code>false</code>
Not	<code>!</code>	Returns <code>true</code> if an expression is false, and returns <code>false</code> if an expression is true

Table 2-10: Logical operators

The operands themselves are often expressions. The following code uses the And (`&&`) operator to compare two separate expressions:

```

1  var gender = "male";
2  var age = 17;
3  var riskFactor = gender === "male" && age <= 21;
4  // returns true

```

In the preceding example, the `gender` variable expression evaluates to `true` because it is equal to “male”, and the `age` variable expression evaluates to `true` because its value is less than or equal to 21. Because both expressions are true, `riskFactor` is assigned a value of `true`. The statement containing the And (`&&`) operator essentially says, “if variable `gender` is equal to “male” *and* variable `age` is less than or equal to 21, then assign a value of `true` to `riskFactor`. Otherwise, assign a value of `false` to `riskFactor`.”

In the following code, `riskFactor` is assigned a value of `false`, because the `age` variable expression does not evaluate to `true`:

```

1  var gender = "male";
2  var age = 25;
3  var riskFactor = gender === "male" && age <= 21;
4  // returns false

```

The logical Or (`||`) operator checks to see if either expression evaluates to `true`. For example, the statement in the following code says, “if the variable `speedingTicket` is greater than 0 *or* if the variable `age` is less than or equal to 21, then assign a value of `true` to `riskFactor`. Otherwise, assign a value of `false` to `riskFactor`.”

```

1  var speedingTicket = 2;
2  var age = 25;

```

```

3  var riskFactor = speedingTicket > 0 || age <= 21;
4  // returns true

```

The `riskFactor` variable in the above example is assigned a value of `true`, because the `speedingTicket` variable expression evaluates to `true`, even though the `age` variable expression evaluates to `false`. This result occurs because the Or (`||`) statement returns `true` if *either* the left or right operand evaluates to `true`.

The following code is an example of the Not (`!`) operator, which returns `true` if an operand evaluates to `false` and returns `false` if an operand evaluates to `true`. Notice that since the Not operator is unary, it requires only a single operand.

```

var trafficViolations = true;
var safeDriverDiscount = !trafficViolations;
// returns false

```

Note

Logical operators are often used within conditional and looping statements such as the *if*, *for*, and *while* statements. You will learn about conditional and looping statements in Chapter 3.

Special Operators

JavaScript also includes the special operators that are listed in Table 2-11. These operators are used for various purposes and do not fit within any other category.

NAME	OPERATOR	DESCRIPTION
Property access	<code>.</code>	Appends an object, method, or property to another object
Array index	<code>[]</code>	Accesses an element of an array
Function call	<code>()</code>	Calls up functions or changes the order in which individual operations in an expression are evaluated
Comma	<code>,</code>	Allows you to include multiple expressions in the same statement
Conditional expression	<code>?:</code>	Executes one of two expressions based on the results of a conditional expression
Delete	<code>delete</code>	Deletes array elements, variables created without the <code>var</code> keyword, and properties of custom objects
Property exists	<code>in</code>	Returns a value of <code>true</code> if a specified property is contained within an object

Continued on next page...

NAME	OPERATOR	DESCRIPTION
Object type	<code>instanceof</code>	Returns <code>true</code> if an object is of a specified object type
New object	<code>new</code>	Creates a new instance of a user-defined object type or a predefined JavaScript object type
Data type	<code>typeof</code>	Determines the data type of a variable
Void	<code>void</code>	Evaluates an expression without returning a result

Table 2-11: Special operators

You will be introduced to the special JavaScript operators as necessary throughout this book. One special operator that you will use in this section is the `typeof` operator. This operator is useful because the data type of variables can change during the course of code execution. This can cause problems if you attempt to perform an arithmetic operation and one of the operands is a string or the `null` value. To avoid such problems, you can use the `typeof` operator to determine the data type of a variable. The syntax for the `typeof` operator is

```
typeof(variablename);
```

You should use the `typeof` operator whenever you need to be sure a variable is the correct data type. The values that can be returned by the `typeof` operator are listed in Table 2-12.

RETURN VALUE	RETURNED FOR
<code>number</code>	Integers and floating-point numbers
<code>string</code>	Text strings
<code>boolean</code>	True or false
<code>object</code>	Objects, arrays, and null variables
<code>function</code>	Functions
<code>undefined</code>	Undefined variables

Table 2-12: Values returned by `typeof` operator

Short Quiz 3

1. What is the difference between a binary operator and a unary operator?
2. How does JavaScript deal with code that performs arithmetic operations on string values?
3. What is a comparison operator? What kind of value does it return?
4. What is a falsy value? What are the six falsy values in JavaScript?

Understanding Operator Precedence

When using operators to create expressions in JavaScript, you need to be aware of the precedence of an operator. **Operator precedence** is the system that determines the order in which operations in an expression are evaluated. Table 2-13 shows the order of precedence for JavaScript operators. Operators in the same grouping in Table 2-13 have the same order of precedence. When performing operations with operators in the same precedence group, the order of precedence is determined by the operator's **associativity**—that is, the order in which operators of equal precedence execute. Associativity is evaluated from left-to-right or right-to-left, depending on the operators involved, as explained shortly.

OPERATORS	DESCRIPTION	ASSOCIATIVITY
.	Objects—highest precedence	Left to right
[]	Array elements—highest precedence	Left to right
()	Functions/evaluation—highest precedence	Left to right
new	New object—highest precedence	Right to left
++	Increment	Right to left
--	Decrement	Right to left
-	Unary negation	Right to left
+	Unary positive	Right to left
!	Not	Right to left
typeof	Data type	Right to left
void	Void	Right to left
delete	Delete object	Right to left

Continued on next page...

OPERATORS	DESCRIPTION	ASSOCIATIVITY
* / %	Multiplication/division/modulus	Left to right
+ -	Addition/concatenation and subtraction	Left to right
< <= > >=	Comparison	Left to right
instanceof	Object type	Left to right
in	Object property	Left to right
== != === !==	Equality	Left to right
&&	Logical And	Left to right
	Logical Or	Left to right
?:	Conditional	Right to left
=	Assignment	Right to left
+= -= *= /= %=	Compound assignment	Right to left
,	Comma—lowest precedence	Left to right

Table 2-13: Operator precedence

Note

The preceding list does not include bitwise operators. As explained earlier, bitwise operators are beyond the scope of this book.

In Table 2-13, operators in a higher grouping have precedence over operators in a lower grouping. For example, the multiplication operator (*) has a higher precedence than the addition operator (+). Therefore, the expression `5 + 2 * 8` evaluates as follows: the numbers 2 and 8 are multiplied first for a total of 16, then the number 5 is added, resulting in a total of 21. If the addition operator had a higher precedence than the multiplication operator, then the statement would evaluate to 56, because 5 would be added to 2 for a total of 7, which would then be multiplied by 8.

As an example of how associativity is evaluated, consider the multiplication and division operators. These operators have an associativity of left to right. Thus the expression `30 / 5 * 2` results in a value of 12. Although the multiplication and division operators have equal precedence, the division operation executes first due to the left-to-right associativity of both operators. Figure 2-16 conceptually illustrates the left to right associativity of the `30 / 5 * 2` expression.

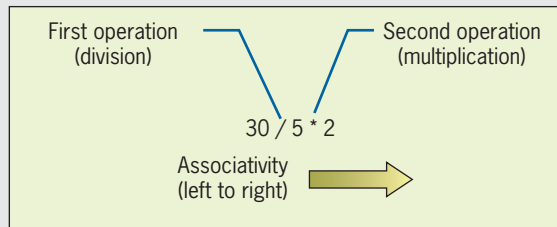


Figure 2-16: Conceptual illustration of left-to-right associativity

If the multiplication operator had higher precedence than the division operator, then the statement `30 / 5 * 2` would result in a value of 3 because the multiplication operation (`5 * 2`) would execute first. By contrast, the assignment operator and compound assignment operators—such as the compound multiplication assignment operator (`*=`)—have an associativity of right to left. Therefore, in the following code, the assignment operations take place from right to left.

```
var x = 3;
var y = 2;
x = y *= ++x;
```

The variable `x` is incremented by one *before* it is assigned to the `y` variable using the compound multiplication assignment operator (`*=`). Then, the value of variable `y` is assigned to variable `x`. The result assigned to both the `x` and `y` variables is 8. Figure 2-17 conceptually illustrates the right to left associativity of the `x = y *= ++x` statement.

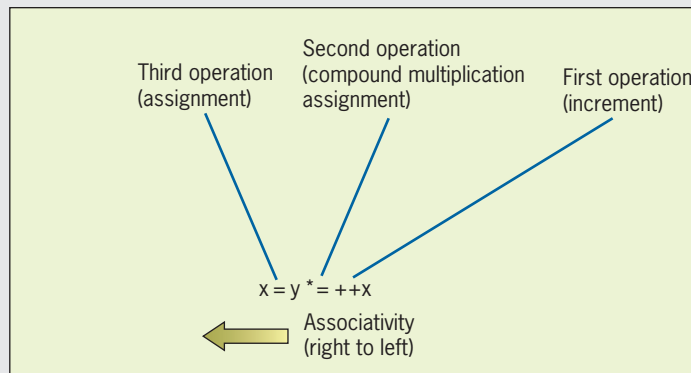


Figure 2-17: Conceptual illustration of right-to-left associativity

As you can see from Table 2-13, parentheses have the highest precedence. Parentheses are used with expressions to change the associativity with which individual operations in an expression are evaluated. For example, the expression $5 + 2 * 8$, which evaluates to 21, can be rewritten to $(5 + 2) * 8$, which evaluates to 56. The parentheses tell JavaScript interpreters to add the numbers 5 and 2 before multiplying by the number 8. Using parentheses forces the statement to evaluate to 56 instead of 21.

To finish the Fan Trick estimate form, you need to modify the `calcStaff()` function to calculate mileage costs. You'll edit the statement that calculates the value of the `photographerCost` variable, adding the product of the value entered in the # of photographers box and the value entered in the Event distance box. (The mileage charge is \$1 per mile per photographer, but you will omit this from your calculation because multiplying by 1 will not change the result.)

To modify the `calcStaff()` function to include mileage charges:

1. Return to the `ft.js` file in your text editor.
2. In the `calcStaff()` function, below the statement that begins `var hrs`, insert the following statement:

```
var distance = document.getElementById("distance");
```

Similar to the `var` statements you created earlier for this function, this statement assigns a reference to the element with the `id` value `distance` to a variable.

3. Locate the statement `photographerCost = num.value * 100 * hrs.value;`, click before the closing `;`, type a **space**, and then type **+ `distance.value * num.value`**

This code multiplies the value entered in the box with the `id` value `distance` by the value entered in the box with the `id` value `photognum`, and adds the product to the earlier calculations in this statement. Because multiplication is higher in the order of precedence than addition, the code evaluates exactly the way you want: first the `num` and `hrs` values are multiplied to determine the staff hourly charges, then the `distance` and `num` values are multiplied to determine the mileage charges, and then the two results are added to determine the total charges for staff. Your completed `calcStaff()` function should match the following:

```
1  function calcStaff() {
2      var num = document.getElementById("photognum");
3      var hrs = document.getElementById("photoghrs");
4      var distance = document.getElementById("distance");
5      totalCost += photographerCost;
```

```

6     photographerCost = num.value * 100 * hrs.value
7     + distance.value * num.value;
8     totalCost += photographerCost;
9     document.getElementById("estimate").innerHTML = "$" +
10     totalCost;
11 }

```

4. Scroll down to the `createEventListeners()` function, and then before the closing `}` add the following statement:

```

document.getElementById("distance").
    addEventListener("change", calcStaff, false);

```

Your completed `createEventListeners()` function should match the following:

```

1  function createEventListeners() {
2      document.getElementById("photognum").
3          addEventListener("change", calcStaff, false);
4      document.getElementById("photoghrs").
5          addEventListener("change", calcStaff, false);
6      document.getElementById("membook").
7          addEventListener("change", toggleMembook, false);
8      document.getElementById("reprodrights").
9          addEventListener("change", toggleRights, false);
10     document.getElementById("distance").
11         addEventListener("change", calcStaff, false);
12 }

```

5. Save the **ft.js** file, and then in your browser refresh or reload the **estimate.htm** file.
6. Change the # of hours value to 2, change the # of photographers value to 3, change the Event distance value to 50, and then press **Tab**. Note that changing the event distance increases the estimate by \$100, which is the product of the # of hours value (2) and the Event distance value (50).

Programming Concepts

Creating Reusable Code

In a small JavaScript program, each function you create generally serves a specific purpose. However, when writing more complex code for larger sites, you should try to build functions that you can apply to multiple situations. For instance, instead of specifying an element name within a function, you could use a variable whose value is specified elsewhere in your program, depending on the context in which the function is called. Because such functions are reusable in multiple contexts within a website, they allow you to perform the same amount of work with less code than would be required to create functions for each specific situation. You'll learn additional strategies for creating reusable code in later chapters of this book.

Short Quiz 4

1. When performing operations with operators in the same precedence group, how is the order of precedence determined?
2. How is the expression $5 + 2 * 8$ evaluated? Explain your answer.

Summary

- › A function is a related group of JavaScript statements that are executed as a single unit.
- › To execute a function, you must invoke, or call, it from elsewhere in your program.
- › The scope of a variable determines where in a program it can be used. A variable's scope can be global (for a variable declared outside a function, which is available to all parts of a program) or local (for a variable declared inside a function, which is available only within the function in which it is declared).
- › A data type (such as number, Boolean, or string) is the specific category of information that a variable contains.
- › JavaScript is a loosely typed programming language, meaning it does not require you to declare the data types of variables.
- › The numeric data type in JavaScript supports both integers (positive or negative numbers with no decimal places) and floating-point numbers (numbers that contains decimal places or that are written in exponential notation).
- › A Boolean value is a logical value of true or false.
- › The JavaScript escape character (\\) tells compilers and interpreters that the character that follows it has a special purpose.
- › Operators are symbols used in expressions to manipulate operands, such as the addition operator (+) and multiplication operator (*).
- › A binary operator (such as +) requires operands before and after the operator, while a unary operator (such as ++) requires a single operand either before or after the operator.
- › Arithmetic operators (such as +, -, *, and /) are used in JavaScript to perform mathematical calculations, such as addition, subtraction, multiplication, and division.
- › An assignment operator (such as = or +=) is used for assigning a value to a variable.
- › A comparison operator (such as === or >) is used to compare two operands and determine if one numeric value is greater than another.
- › The conditional operator (? :) executes one of two expressions, based on the results of a conditional expression.
- › Logical operators (&&, ||, and !) are used for comparing two Boolean operands for equality.
- › Operator precedence is the order in which operations in an expression are evaluated.

Key Terms

actual parameters—See arguments.

adding an event listener—Specifying an event handler with the `addEventListener()` method.

anonymous function—A set of related statements with no name assigned to it.

arguments—The variables or values that you place in the parentheses of a function call statement.

arithmetic operators—Operators used to perform mathematical calculations, such as addition, subtraction, multiplication, and division.

assignment operator—An operator used for assigning a value to a variable.

associativity—The order in which operators of equal precedence execute.

binary operator—An operator that requires an operand before and after it.

Boolean value—A logical value of `true` or `false`.

browser console—A browser pane that displays error messages.

call—To invoke a function from elsewhere in your code.

comparison operator—An operator used to compare two operands and determine if one value is greater than another.

compound assignment operators—Assignment operators other than the equal sign, which perform mathematical calculations on variables and literal values in an expression, and then assign a new value to the left operand.

concatenation operator—The plus sign (+) when used with strings; this operator combines, or concatenates, string operands.

conditional operator—The `?:` operator, which executes one of two expressions based on the results of a conditional expression.

console—See browser console.

data type—The specific category of information that a variable contains, such as numeric, Boolean, or string.

duck typed—See loosely typed.

dynamically typed—See loosely typed.

empty string—A zero-length string value.

escape character—The backslash character (`\`), which tells JavaScript compilers and interpreters that the character that follows it has a special purpose.

escape sequence—The combination of the escape character (`\`) with one of several other characters, which inserts a special character into a string; for example, the `\b` escape sequence inserts a backspace character.

exponential notation—A shortened format for writing very large numbers or numbers with many decimal places, in which numbers are represented by a value between 1 and 10 multiplied by 10 raised to some power.

falsy values—Six values that are treated in comparison operations as the Boolean value `false`.

floating-point number—A number that contains decimal places or that is written in exponential notation.

function—A related group of JavaScript statements that are executed as a single unit.

function braces—The set of curly braces that contain the function statements in a function definition.

function call—The code that calls a function, which consists of the function name followed by parentheses, which contain any arguments to be passed to the function.

function definition—The lines that make up a function.

function statements—The statements that do the actual work of a function, such as calculating the square root of the parameter, or displaying a message on the screen, and which must be contained within the function braces.

global variable—A variable that is declared outside a function and is available to all parts of your program, because it has global scope.

innerHTML property—The property of a web page object whose value is the content between the element's opening and closing tags.

integer—A positive or negative number with no decimal places.

local variable—A variable that is declared inside a function and is available only within the function in which it is declared, because it has local scope.

logical operators—The Or (`||`), And (`&&`), and Not (`!`) operators, which are used to modify Boolean values or specify the relationship between operands in an expression that results in a Boolean value.

loosely typed—Description of a programming language that does not require you to declare the data types of variables.

named function—A set of related statements that is assigned a name.

operator precedence—The system that determines the order in which operations in an expression are evaluated.

parameter—A variable that is used within a function.

passing arguments—Sending arguments to the parameters of a called function.

postfix operator—An operator that is placed after a variable name.

prefix operator—An operator that is placed before a variable name.

primitive types—Data types that can be assigned only a single value.

relational operator—See comparison operator.

return statement—A statement in a function that returns a value to the statement that called the function.

scientific notation—See exponential notation.

statically typed—See strongly typed.

strongly typed—Description of a programming language that requires you to declare the data types of variables.

textContent property—A property similar to the `innerHTML` property, except that its value excludes any HTML markup.

truthy values—All values other than the six falsy values; truthy values are treated in comparison operations as the Boolean value `true`.

unary operator—An operator that requires just a single operand either before or after it.

variable scope—The aspect of a declared variable that determines where in code it can be used, either globally (throughout the code) or locally (only within the function in which it is declared).

Review Questions

1. A(n) _____ allows you to execute a related group of statements as a single unit.
 - a. variable
 - b. statement
 - c. event
 - d. function
2. Parameters in a function definition are placed within _____.
 - a. braces
 - b. parentheses
 - c. double quotes
 - d. single quotes
3. A variable that is declared outside a function is called a(n) _____ variable.
 - a. class
 - b. local
 - c. global
 - d. program
4. Which one of the following creates a local variable?
 - a. Declaring it outside of a function with the `var` keyword
 - b. Declaring it outside of a function without the `var` keyword
 - c. Declaring it inside a function with the `var` keyword
 - d. Declaring it inside a function without the `var` keyword
5. Which of the following is a primitive data type in JavaScript?
 - a. Boolean
 - b. Integer
 - c. Floating-point
 - d. Logical

6. Which of the following describes JavaScript?
 - a. Strongly typed
 - b. Statically typed
 - c. Loosely typed
 - d. Untyped
7. Which of the following is an integer?
 - a. -2.5
 - b. 6.02e23
 - c. -11
 - d. 0.03
8. Which of the following is a Boolean value?
 - a. 3.04
 - b. true
 - c. "Greece"
 - d. 6.02e23
9. Which of the following creates an empty string?
 - a. null
 - b. undefined
 - c. ""
 - d. 0
10. Which of the following is a valid JavaScript statement?
 - a. `document.write('Boston, MA is called 'Beantown.')`
 - b. `document.write("Boston, MA is called "Beantown."")`
 - c. `document.write("Boston, MA is called 'Beantown.'")`
 - d. `document.write("Boston, MA is called 'Beantown.'")`
11. Which of the following is a concatenation operator?
 - a. >
 - b. +
 - c. ||
 - d. ++
12. Which of following is the JavaScript escape character?
 - a. "
 - b. '
 - c. \
 - d. /

13. Which of the following is an arithmetic binary operator?
 - a. +
 - b. ||
 - c. =
 - d. &&
14. Which of the following is an arithmetic unary operator?
 - a. ++
 - b. ||
 - c. =
 - d. &&
15. What is the result of the statement `5 < 4`?
 - a. 1
 - b. yes
 - c. true
 - d. false
16. Write a simple function (or copy one used in this chapter), that includes the following parts, and then label the parts:
 - › Name
 - › Parameters
 - › Function braces
 - › Function statements
17. Explain the difference between prefix and postfix operators, and provide an example of each.
18. What is the result of the following expression?
`5 > 4 ? document.write("green") : document.write("blue");`
Name the operators, and explain the steps you took to arrive at your answer.
19. What is the result of the expression `5 % 4`? Name the operator, and explain the steps you took to arrive at your answer.
20. Explain the difference between global and local variables, and describe how using or not using the `var` keyword can affect the scope of a variable.

Hands-On Projects

Hands-On Project 2-1

In this project, you'll create a script that calculates the Celsius equivalent of a Fahrenheit temperature. Note that your result should work on all modern browsers, but will not work on IE8 or previous versions of IE.

1. In your text editor, open **index.htm** from the HandsOnProject2-1 folder in the Chapter02 folder. Enter your name and today's date where indicated in the comment section in the document head.
2. At the bottom of the document, before the closing `</body>` tag, enter `<script>`, insert a blank line, and then enter `</script>` to create a new script section.
3. Within the script section you created in the previous step, enter the following function.

```
1  function convert() {  
2      var degF = document.getElementById("fValue").value;  
3      var degC = degF - 32 * 5 / 9;  
4      document.getElementById("cValue").innerHTML = degC;  
5  }
```

This function, named `convert()`, starts by looking up the Fahrenheit value entered by users and assigning it to a variable named `degF`. It then performs calculations on `degF` to arrive at the Celsius equivalent, which is assigned to a variable named `degC`. Finally, it assigns the value of `degC` as the `innerHTML` value of the element with the `id` value `cValue`.

4. Below the closing `}` for the `convert()` function, but before the closing `</script>` tag, enter the following statement to add an event listener:

```
document.getElementById("button").  
    addEventListener("click", convert, false);
```

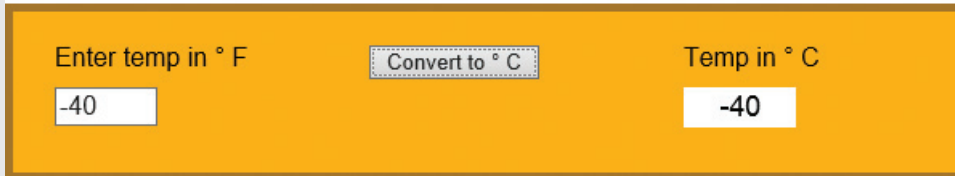
5. Save your work, open **index.htm** in your browser, enter **-40** in the Enter temp in ° F box, and then click the **Convert to ° C** button. -40° Fahrenheit is actually equivalent to -40° Celsius. However, the formula incorrectly calculates that -40° F is equivalent to -57.7° C. This is because the calculations in the equation must take place in a different order than the order of precedence dictates.
6. Return to the **index.htm** file in your text editor, and then add two sets of parentheses to the first statement in the `convert()` function to modify the order in which the calculations are performed, as follows:

```
var degC = (degF - 32) * (5 / 9);
```

7. Save your work, refresh or reload **index.htm** in your browser, enter **-40** in the Enter temp in ° F box, and then click the **Convert to ° C** button. As Figure 2-18 shows, the temperature is now calculated correctly as -40° C.

Hands-on Project 2-1

Fahrenheit (° F) to Celsius (° C) converter

A screenshot of a web form titled "Fahrenheit (° F) to Celsius (° C) converter". The form has a yellow background. It contains two input fields: "Enter temp in ° F" and "Temp in ° C". Both fields have the value "-40" entered. Between the two fields is a button labeled "Convert to ° C".

Enter temp in ° F	Convert to ° C	Temp in ° C
-40		-40

Figure 2-18: Fahrenheit to Celsius converter

Hands-On Project 2-2

In this project, you'll create a script that uses logical operators and the conditional operator to give users feedback based on whether form fields are completed.

1. In your text editor, open **index.htm** from the HandsOnProject2-2 folder in the Chapter02 folder. Enter your name and today's date where indicated in the comment section in the document head.
2. At the bottom of the document, before the closing `</body>` tag, enter `<script>`, insert a blank line, and then enter `</script>` to create a new script section.
3. Within the script section you created in the previous step, enter the following function:

```
1  function submitInfo() {  
2      var name = document.getElementById("nameinput");  
3      var email = document.getElementById("emailinput");  
4      var phone = document.getElementById("phoneinput");  
5      (name.value && email.value && phone.value) ?   
6          alert("Thank you!") : alert("Please fill in all fields");  
7  }
```

This function starts by declaring three variables, which point to three web page elements. The remaining code is a single conditional expression. The statement to be tested for a Boolean `true` or `false` value checks if the element with the `id` value of `nameinput` is `truthy` (in this case, is not `""`), and `(&&)` if the element with the `id` of `emailinput` is `truthy`, and if the element with the `id` of `phoneinput` is `truthy`. If all of these statements are `true`, then the entire statement has the Boolean value `true`,

and an alert box is displayed with the text “Thank you!”. If any of these statements is false, then the entire statement has the Boolean value `false`, and an alert box is displayed with the text “Please fill in all fields”.

4. Below the closing `}` for the `submitInfo()` function, but before the closing `</script>` tag, enter the following statement to add an event listener:

```
document.getElementById("submit").  
    addEventListener("click", submitInfo, false);
```

5. Save your work, open **index.htm** in your browser, and then click the **Submit** button. The browser displays an alert box containing the text “Please fill in all fields” because the fields were all blank when you clicked the button. Click the **OK** button in the alert box, enter text in just the Name box, and then click **Submit**. The browser again displays an alert box containing the text “Please fill in all fields” because two of the fields were still blank when you clicked the button. Click the **OK** button in the alert box, enter text in all three fields, and then click **Submit**. As shown in Figure 2-19, the browser now displays an alert box containing the text “Thank you!” because none of the boxes were empty when you clicked the button. Click **OK**.

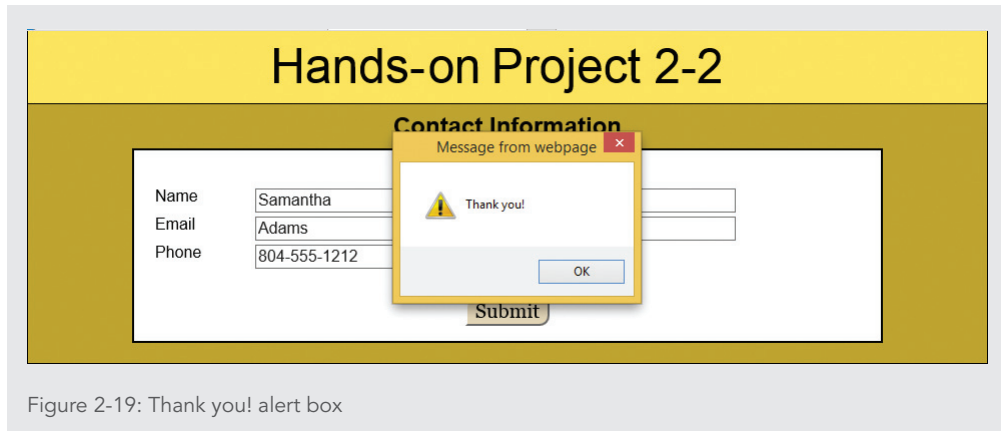


Figure 2-19: Thank you! alert box

Hands-On Project 2-3

In this project, you'll create a script that displays an alert box containing a custom message based on the element a user clicks. Note that your result should work on all modern browsers, but will not work on IE8 or previous versions of IE.

1. In your text editor, open **index.htm** from the **HandsOnProject2-3** folder in the **Chapter02** folder. Enter your name and today's date where indicated in the comment section in the document head.

2. Open **index.htm** in your browser. The web page displays three shapes: a square, a triangle, and a circle.
3. Return to your text editor. At the bottom of the document, before the closing `</body>` tag, enter `<script>`, insert a blank line, and then enter `</script>` to create a new script section.
4. Within the script section you created in the previous step, enter the following event listener:

```
1 document.getElementById("square").  
2     addEventListener("click", function() {  
3         alert("You clicked the square");  
4     }, false);
```

This code adds an event listener to the element with the `id` of `square`. The code uses an anonymous function as the second argument for the `addEventListener()` method. When a user clicks the element, the anonymous function is executed, which generates an alert box containing the text “You clicked the square”.

5. Below the event listener code you added in the previous step, but before the closing `</script>` tag, enter the following code to add event listeners for the remaining two shape elements:

```
1 document.getElementById("triangle").  
2     addEventListener("click", function() {  
3         alert("You clicked the triangle");  
4     }, false);  
5 document.getElementById("circle").  
6     addEventListener("click", function() {  
7         alert("You clicked the circle");  
8     }, false);
```

6. Save your work, refresh or reload **index.htm** in your browser, and then click the **square**. As Figure 2-20 shows, the browser opens an alert box that displays the text “You clicked the square”. Click **OK**, and then repeat for the triangle and the circle. Each alert box should name the shape you clicked.

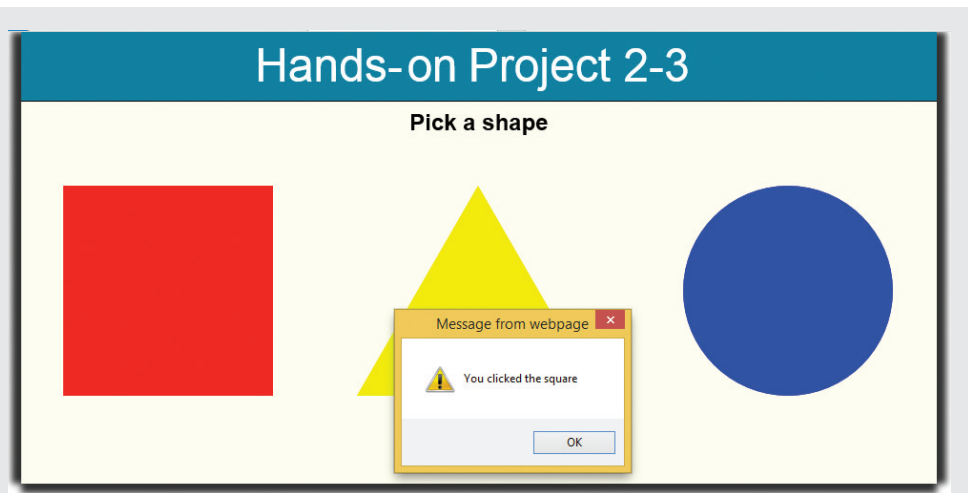


Figure 2-20: Alert box displayed after clicking square element

Hands-On Project 2-4

In this project, you'll create a script that totals purchases and adds tax. Note that your result should work on all modern browsers, but will not work on IE8 or previous versions of IE.

1. In your text editor, open **index.htm** from the HandsOnProject2-4 folder in the Chapter02 folder. Enter your name and today's date where indicated in the comment section in the document head.
2. Open **index.htm** in your browser. The web page displays a form with five check boxes and a Submit button. Each check box allows users to select an item from a lunch menu. You'll create a script that totals the prices of all the elements a users selects, adds sales tax, and displays the order total in an alert box.
3. Return to your text editor. At the bottom of the document, before the closing `</body>` tag, enter `<script>`, insert a blank line, and then enter `</script>` to create a new script section.
4. Within the script section you created in the previous step, enter the following function.

```
1  function calcTotal() {  
2      var itemTotal = 0;  
3      var item1 = document.getElementById("item1");  
4      var item2 = document.getElementById("item2");  
5      var item3 = document.getElementById("item3");
```

```

6      var item4 = document.getElementById("item4");
7      var item5 = document.getElementById("item5");
8      (item1.checked) ? (itemTotal += 8) : (itemTotal += 0);
9      (item2.checked) ? (itemTotal += 9) : (itemTotal += 0);
10     (item3.checked) ? (itemTotal += 8) : (itemTotal += 0);
11     (item4.checked) ? (itemTotal += 13) : (itemTotal += 0);
12     (item5.checked) ? (itemTotal += 6) : (itemTotal += 0);
13     var salesTaxRate = 0.07;
14     var orderTotal = itemTotal + (itemTotal * salesTaxRate);
15     alert("Your order total is $" + orderTotal);
16 }

```

The `calcTotal()` function starts by creating six local variables—`itemTotal`, and one variable storing a reference to each check box element. Each statement in lines 8–12 contains a conditional statement that evaluates whether one of the check boxes on the page is checked. If so, the price corresponding to that check box is added to the `itemTotal` variable; if not, 0 is added to the `itemTotal` variable. After examining the values of all the check boxes, the function declares two additional variables. The `salesTaxRate` variable specifies the percentage of the purchase price that must be added to the total for tax. In line 14, the value of the `orderTotal` variable is calculated by first multiplying `itemTotal` and `salesTaxRate` to determine the sales tax amount and then adding that amount to the `itemTotal` value. The function ends by generating an alert box containing the text “Your order total is \$” followed by the value of the `orderTotal` variable.

5. Below the closing `}` for the `calcTotal()` function, but before the closing `</script>` tag, enter the following code to add an event listener:

```

document.getElementById("submit").  
    addEventListener("click", calcTotal, false);

```

6. Save your work, refresh or reload **index.htm** in your browser, click the **Fried chicken** check box, and then click the **Submit** button. As Figure 2-21 shows, the alert box displays a total of \$8.56. Click OK.

Hands-on Project 2-4

Lunch selections

- ☒ Fried chicken (\$8.00)
- ☐ Fried halibut (\$9.00)
- ☐ Hamburger (\$8.00)
- ☐ Grilled salmon (\$13.00)
- ☐ Side salad (\$6.00)

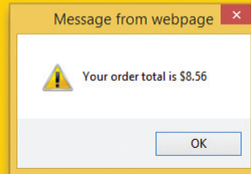


Figure 2-21: Alert box displaying order total

7. Select other items or combinations of items, and verify that the reported total realistically represents the total of the selected items plus 7% tax.

Hands-On Project 2-5

In this project, you'll enhance the document you worked on in Hands-On Project 2-4 to avoid potential differences between floating point and integer calculations. Note that your result should work on all modern browsers, but will not work on IE8 or previous versions of IE.

1. In the file manager for your operating system, copy the completed contents of the HandsOnProject2-4 folder to the HandsOnProject2-5 folder.
2. In your text editor, open the **index.htm** file from the HandsOnProject2-5 folder, change "Hands-on Project 2-4" to **Hands-On Project 2-5** in the comment section, in the `title` element, and in the `h1` element, and then save your changes.
3. In the `calcTotal()` function, after the statement `var SalesTaxRate = 0.07;`, add the following statement:

```
itemTotal *= 100;
```

This multiplies the total of selected items by 100 and assigns the result as the new value of `itemTotal`. This converts the item total from dollars (such as \$7.99) to cents (such as 799¢), which is an integer, thus avoiding possible issues with floating point calculations.

4. In the next statement, which begins `var orderTotal`, before the first occurrence of `itemTotal`, add an opening parenthesis, before the closing `;` add a closing parenthesis, and then before the closing `;` add `/ 100`. Your updated statement should match the following:

```
var orderTotal =  $\left( \text{itemTotal} + (\text{itemTotal} * \text{salesTaxRate}) \right) / 100;$ 
```

The existing content of the right operand is enclosed in parentheses, and the entire calculated value is divided by 100, converting it from cents back to dollars.

5. Save your changes, open `index.htm` in your browser, click the **Fried chicken** check box, and then click the **Submit** button. Verify that the alert box still displays a total of \$8.56. Because the `orderTotal` calculation doesn't encounter any floating-point problems, you won't see different results based on the changes you made. However, if the page's requirements changed in the future to require a calculation that might introduce floating-point issues, your changes would prevent users from experiencing any problems.

Case Projects

Individual Case Project

Plan and add a feature to one of the web pages in your personal site that uses at least one function to perform a mathematical calculation based on user input. Test the page to ensure it works as planned.

Team Case Project

Choose one of the web pages from your team website to enhance with at least two functions. Common uses of functions include performing actions based on user input (validation, personalization of the web page) and performing calculations. Divide your team into subgroups equal to the number of functions your page will include. After each subgroup has created its function, come back together as a full group and incorporate the functions in the web page. Test the functions to verify the page works as planned, doing any troubleshooting and making any edits to the functions as a full team.