TRAINING & REFERENCE

murach's JavaScript 2nd Edition

(Chapter 2)

Thanks for downloading this chapter from **Murach's JavaScript (2nd Edition)**. We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its "how-to" headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our **website**. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on related topics.

Thanks for your interest in our books!

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • www.murach.com

What developers said about the previous edition

"If you are new to web design or an old pro like me, this book is a must-have in my opinion. I love how it starts out with the basics and then moves on to the good stuff. Each chapter is full of examples and sample code showing you how to do the most common techniques that you will face as a web developer/designer. This one will be on my desk for a while!"

Shawn Jackson, Web Developer, Colorado

"I am a graphic artist doing web development, I'm not a programmer. Fortunately for me, and any of you out there like me, Murach wrote this book. It speaks [in] everyday English and doesn't assume you know too much. It has given me the technical knowledge I need to create dynamic websites."

Kurt Pruhs; posted at an online bookseller

"I bought this book to fill in the massive hole left by the required textbook in a college course. The two-page topic style is great. I am looking into other Murach titles!"

Posted at an online bookseller

"An essential characteristic is that the examples, programs, and applications are all thoroughly tested. This book will work WONDERFULLY to help you build better and more robust websites!"

Marvin Schneider, Instructor, New York

"I've read a ton of books on JavaScript, and Murach's wonderful tome is perhaps THE best modern title you'll find on the topic. It's intelligently organized, and written in a concise fashion, with practical examples, delivered with a friendly voice that makes what can often be convoluted or confusing concepts easy to grasp for beginners."

Jason Salas, Developer, Guam

2

Getting started with JavaScript

The goal of this chapter is to get you off to a good start with JavaScript, especially if you're new to programming. If you have programming experience with another language, you should be able to move rapidly through this chapter. Otherwise, take it easy and do the exercises at the end of this chapter.

How to include JavaScript in an HTML document

In chapter 1, you saw how the JavaScript for an application can be coded in a separate file. But there are actually three different ways to include JavaScript in an HTML document. You'll learn all three now.

Two ways to include JavaScript in the head of an HTML document

Figure 2-1 presents two of the three ways to include JavaScript in an HTML document. As you saw in the last chapter, one way is to code the JavaScript in a separate *external file*. Then, you code a script element in the head section of the HTML document to include that file.

In the script element, the src attribute is used to refer to the external file. For this element, you can also code a type attribute with the value "text/javascript" to tell the browser what kind of content the file contains. But with HTML5, that attribute is no longer needed because the assumption is that all files that are referred to in script elements contain JavaScript.

In the example in this figure, the src attribute refers to a file named calculate_mpg.js. The assumption here is that this file is in the same folder as the HTML file. Otherwise, you need to code a relative URL that provides the right path for the file. If, for example, the JavaScript file is in a folder named javascript and that folder is in the same folder as the HTML file, the src attribute would be coded this way:

```
<script src="javascript/calculate_mpg.js"></script>
```

This works the same as it does for any other file reference in an HTML document.

The second way to include JavaScript in an HTML document is to code the JavaScript within the script element in the head section. This can be referred to as *embedded JavaScript*. Note, however, that the application will work the same whether the JavaScript is embedded in the head section or loaded into the head section from an external file.

The benefit of using an external file is that it separates the JavaScript from the HTML. Another benefit is that it makes it easier to re-use the code in other pages or applications.

The benefit of using embedded JavaScript is that you don't have to switch between the HTML and JavaScript files as you develop the application. In the examples in this book, you'll see both uses of JavaScript.

As you'll see in the next figure, script elements can be in the body section of the document, too. In fact, some developers prefer to place their script elements just before the closing body tag. This can make the page appear to load faster because the HTML will render before the JavaScript loads. But if you need JavaScript functionality as the HTML renders, scripts at the bottom won't work. Ultimately, you'll make this decision based on your application's needs.

Two attributes of the script element

Attribute	Description
`src`	Specifies the location (source) of an external JavaScript file.
`type`	With HTML5, this attribute can be omitted. If you code it, use "text/javascript" for JavaScript code.

A script element in the head section that loads an external JavaScript file

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

A script element that embeds JavaScript in the head section

```
<head>
    ...
    <script>
        alert("The Calculate MPG application");
        var miles = prompt("Enter miles driven");
        miles = parseFloat(miles);
        var gallons = prompt("Enter gallons of gas used");
        gallons = parseFloat(gallons);
        var mpg = miles/gallons;
        mpg = parseInt(mpg);
        alert("Miles per gallon = " + mpg);
    </script>
</head>
```

Description

- A script element in the head section of an HTML document is commonly used to identify an *external JavaScript file* that should be included with the page.
- A script element in the head section can also contain the JavaScript statements that are included with the page. This can be referred to as *embedded JavaScript*.
- If you code more than one script element in the head section, the JavaScript is included in the sequence in which the script statements appear.
- When a script element in the head section includes an external JavaScript file, the JavaScript in the file runs as if it were coded in the script element.
- Some programmers prefer to place their script elements at the bottom of the page, just before the closing body tag. This can make a page seem to load faster, but the JavaScript won't run until after the page is loaded.

Figure 2-1 Two ways to include JavaScript in the head of an HTML document

How to include JavaScript in the body of an HTML document

Figure 2-2 shows a third way to include JavaScript in an HTML document. This time, the two script elements in the first example are coded in the body of the HTML document.

If you want to provide for browsers that don't have JavaScript enabled, you can code a noscript element after a script element as shown in the second example. Then, if JavaScript is disabled, the content of the noscript element will be displayed. But if JavaScript is enabled, the code in the script element runs, and the noscript element is ignored. This way, some output will be displayed whether or not JavaScript is enabled.

In the second example, the noscript element is coded right after a script element, so 2015 will replace the output of the script element if JavaScript isn't enabled in the browser. This means that the result will be the same in the year 2015 whether or not JavaScript is enabled. But after 2015, the year will be updated by the JavaScript if it is enabled.

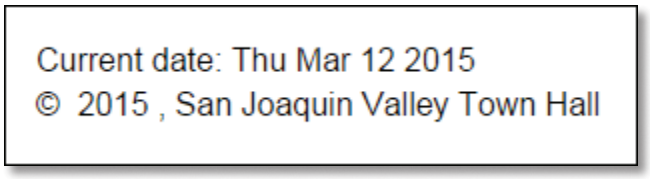
You can also code a noscript element that doesn't follow a script element. For instance, you can code a noscript element at the top of a page that warns the user that the page won't work right if JavaScript is disabled. This is illustrated by the third example. In this case, nothing is displayed if JavaScript is enabled and the message is displayed if JavaScript is disabled.

In most of the applications in this book, the JavaScript is either embedded in the head section of the HTML or in an external file that's identified in the head section of the HTML. However, you shouldn't have any trouble including the JavaScript in the body of an HTML document whenever you want to do that.

JavaScript in the body of an HTML document

```
<p>
    <script>
        var today = new Date();
        document.write("Current date: ");
        document.write(today.toDateString());
    </script>
</p>
<p>&copy; 
    <script>
        var today = new Date();
        document.write( today.getFullYear() );
    </script>
, San Joaquin Valley Town Hall
</p>
```

The result of the JavaScript in a web browser



A noscript element in the body of an HTML document

```
<p>&copy; 
    <script>
        var today = new Date();
        document.write( today.getFullYear() );
    </script>
    <noscript>2015</noscript>
, San Joaquin Valley Town Hall
</p>
```

A noscript element at the start of an HTML document

```
<h2><noscript>To get the most from this web site,
    please enable JavaScript.</noscript></h2>
```

Description

- The JavaScript code in a script element in the body of a document is run when the page is loaded.
- The noscript element can be used to display content when JavaScript is disabled in a user's browser.

Figure 2-2 How to include JavaScript in the body of an HTML document

The JavaScript syntax

The *syntax* of JavaScript refers to the rules that you must follow as you code statements. If you don't adhere to these rules, your web browser won't be able to interpret and execute your statements.

How to code JavaScript statements

Figure 2-3 summarizes the rules for coding *JavaScript statements*. The first rule is that JavaScript is case-sensitive. This means that uppercase and lowercase letters are treated as different letters. For example, *salestax* and *salesTax* are treated as different names.

The second rule is that JavaScript statements must end with a semicolon. If you don't end each statement with a semicolon, JavaScript won't be able to tell where one statement ends and the next one begins.

The third rule is that JavaScript ignores extra whitespace in statements. Since *whitespace* includes spaces, tabs, and new line characters, this lets you break long statements into multiple lines so they're easier to read.

Be careful, though, to follow the guidelines in this figure about where to split a statement. If you don't split a statement at a good spot, JavaScript will sometimes try to help you out by adding a semicolon for you, and that can lead to errors.

A block of JavaScript code

```
var joinList = function() {
    var emailAddress1 = $("email_address1").value;
    var emailAddress2 = $("email_address2").value;

    if (emailAddress1 == "") {
        alert("Email Address is required.");
    } else if (emailAddress2 == "") {
        alert("Second Email Address is required.");
    } else if (emailAddress1 != emailAddress2) {
        alert("Second Email entry must equal first entry.");
    } else if ($("first_name").value == "") {
        alert("First Name is required.");
    } else {
        $("email_form").submit();
    }
};
```

The basic syntax rules

- JavaScript is case-sensitive.
- Each JavaScript statement ends with a semicolon.
- JavaScript ignores extra whitespace within statements.

How to split a statement over two or more lines

- Split a statement after:
  an arithmetic or relational operator such as +, -, *, /, =, ==, >, or <
  an opening brace ({), bracket ([ ), or parenthesis
  a closing brace (})
- Do not split a statement after:
  an identifier, a value, or the *return* keyword
  a closing bracket ( ]) or closing parenthesis

Description

- A JavaScript *statement* has a syntax that's similar to the syntax of Java.
- *Whitespace* refers to the spaces, tab characters, and return characters in the code, and it is ignored by the compiler. As a result, you can use spaces, tab characters, and return characters to format your code so it's easier to read.
- In some cases, JavaScript will try to correct what it thinks is a missing semicolon by adding a semicolon at the end of a split line. To prevent this, follow the guidelines above for splitting a statement.

Figure 2-3 How to code JavaScript statements

How to create identifiers

Variables, functions, objects, properties, methods, and events must all have names so you can refer to them in your JavaScript code. An *identifier* is the name given to one of these components.

Figure 2-4 shows the rules for creating identifiers in JavaScript. Besides the first four rules, you can't use any of the JavaScript *reserved words* (also known as *keywords*) as an identifier. These are words that are reserved for use within the JavaScript language. You should also avoid using any of the JavaScript global properties or methods as identifiers, which you'll learn more about as you progress through this book.

Besides the rules, you should give your identifiers meaningful names. That means that it should be easy to tell what an identifier refers to and easy to remember how to spell the name. To create names like that, you should avoid abbreviations. If, for example, you abbreviate the name for monthly investment as mon_inv, it will be hard to tell what it refers to and hard to remember how you spelled it. But if you spell it out as monthly_investment, both problems are solved.

Similarly, you should avoid abbreviations that are specific to one industry or field of study unless you are sure the abbreviation will be widely understood. For example, mpg is a common abbreviation for miles per gallon, but cpm could stand for a number of different things and should be spelled out.

To create an identifier that has more than one word in it, many JavaScript programmers use a convention called *camel casing*. With this convention, the first letter of each word is uppercase except for the first word. For example, monthlyInvestment and taxRate are identifiers that use camel casing.

The alternative is to use underscore characters to separate the words in an identifier. For example, monthly_investment and tax_rate use this convention. If the standards in your shop specify one of these conventions, by all means use it. Otherwise, you can use whichever convention you prefer...but be consistent.

In this book, underscore notation is used for the ids and class names in the HTML, and camel casing is used for all JavaScript identifiers. That way, it will be easier for you to tell where the names originated.

Rules for creating identifiers

- Identifiers can only contain letters, numbers, the underscore, and the dollar sign.
- Identifiers can't start with a number.
- Identifiers are case-sensitive.
- Identifiers can be any length.
- Identifiers can't be the same as *reserved words*.
- Avoid using global properties and methods as identifiers. If you use one of them, you won't be able to use the global property or method with the same name.

Valid identifiers in JavaScript

```
subtotal        index_1             $
taxRate         calculate_click     $log
```

Camel casing versus underscore notation

```
taxRate                 tax_rate
calculateClick          calculate_click
emailAddress            email_address
firstName               first_name
futureValue             future_value
```

Naming recommendations

- Use meaningful names for identifiers. That way, your identifiers aren't likely to be reserved words or global properties.
- Be consistent: Either use camel casing (taxRate) or underscores (tax_rate) to identify the words within the variables in your scripts.
- If you're using underscore notation, use lowercase for all letters.

Reserved words in JavaScript

```
abstract        else            instanceof      switch
boolean         enum            int             synchronized
break           export          interface       this
byte            extends         long            throw
case            false           native          throws
catch           final           new             transient
char            finally         null            true
class           float           package         try
const           for             private         typeof
continue        function        protected       var
debugger        goto            public          void
default         if              return          volatile
delete          implements      short           while
do              import          static          with
double          in              super
```

Description

- *Identifiers* are the names given to variables, functions, objects, properties, and methods.
- In *camel casing*, all of the words within an identifier except the first word start with capital letters.

Figure 2-4 How to create identifiers

How to use comments

Comments let you add descriptive notes to your code that are ignored by the JavaScript engine. Later on, these comments can help you or someone else understand the code whenever it needs to be modified.

The example in figure 2-5 shows how comments can be used to describe or explain portions of code. At the start, a *block comment* describes what the application does. This kind of comment starts with /* and ends with */. Everything that's coded between the start and the end is ignored by the JavaScript engine when the application is run.

The other kind of comment is a *single-line comment* that starts with //. In the example, the first single-line comment describes what the JavaScript that comes before it on the same line does. In contrast, the second single-line comment takes up a line by itself. It describes what the two statements that come after it do.

In addition to describing JavaScript code, comments can be useful when testing an application. If, for example, you want to disable a portion of the JavaScript code, you can enclose it in a block comment. Then, it will be ignored when the application is run. This can be referred to as *commenting out* a portion of code. Later, after you test the rest of the code, you can enable the commented out code by removing the markers for the start and end of the block comment. This can be referred to *uncommenting*.

Comments are also useful when you want to experiment with changes in code. For instance, you can make a copy of a portion of code, comment out the original code, and then paste the copy just above the original code that is now commented out. Then, you can make changes in the copy. But if the changes don't work, you can restore your old code by deleting the new code and uncommenting the old code.

When should you use comments to describe or explain code? Certainly, when the code is so complicated that you may not remember how it works if you have to maintain it later on. This kind of comment is especially useful if someone else is going to have to maintain the code.

On the other hand, you shouldn't use comments to explain code that any professional programmer should understand. That means that you have to strike some sort of balance between too much and too little. One of the worst problems with comments is changing the way the code works without changing the related comments. Then, the comments mislead the person who is trying to maintain the code, which makes the job even more difficult.

A portion of JavaScript code that includes comments

```
/* this application validates a user's entries for joining
   our email list
*/
var $ = function(id) {                          // the standard $ function
    return document.getElementById(id);
}
// this function gets and validates the user entries
var joinList = function() {
    var emailAddress1 = $("email_address1").value;
    var emailAddress2 = $("email_address2").value;
    var firstName = $("first_name").value;
    var isValid = true;

    // validate the first entry
    if (emailAddress1 == "") {
        $("email_address1_error").firstChild.nodeValue =
            "This field is required.";
        isValid = false;                        // set valid switch to off
    } else {
        $("email_address1_error").firstChild.nodeValue = "";
    }

    // validate the second entry
    ...
    ...
```

The basic syntax rules for JavaScript comments

- Block comments begin with /* and end with */.
- Single-line comments begin with two forward slashes and continue to the end of the line.

Guidelines for using comments

- Use comments to describe portions of code that are hard to understand.
- Use comments to disable portions of code that you don't want to test.
- Don't use comments unnecessarily.

Description

- JavaScript provides two forms of *comments*, *block comments* and *single-line comments*.
- Comments are ignored when the JavaScript is executed.
- During testing, comments can be used to *comment out* (disable) portions of code that you don't want tested. Then, you can remove the comments when you're ready to test those portions.
- You can also use comments to save a portion of code in its original state while you make changes to a copy of that code.

Figure 2-5 How to use comments

How to use objects, methods, and properties

In simple terms, an *object* is a collection of methods and properties. A *method* performs a function or does an action. A *property* is a data item that relates to the object. When you develop JavaScript applications, you will often work with objects, methods, and properties.

To get you started with that, figure 2-6 shows how to use the methods and properties of the window object, which is a common JavaScript object. To *call* (execute) a method of an object, you use the syntax in the summary after the tables. That is, you code the object name, a *dot operator* (period), the method name, and any *parameters* that the method requires within parentheses.

In the syntax summaries in this book, some words are italicized and some aren't. The words that aren't italicized are keywords that always stay the same, like *alert*. You can see this in the first table, where the syntax for the alert method shows that you code the word *alert* just as it is in the summary. In contrast, the italicized words are the ones that you need to supply, like the string parameter you supply to the alert method.

In the first example after the syntax summary, you can see how the alert method of the *window object* is called:

```
window.alert("This is a test of the alert method");
```

In this case, the one parameter that's passed to it is "This is a test of the alert method". So that message is displayed when the alert dialog box is displayed.

In the second example, you can see how the prompt method of the window object is called. This time, though, the object name is omitted. For the window object (but only the window object), that's okay because the window object is the *global object* for JavaScript applications.

As you can see, the prompt method accepts two parameters. The first one is a message, and the second one is an optional default value for a user entry. When the prompt method is executed, it displays a dialog box like the one in this figure. Here, you can see the message and the default value that were passed to the method as parameters. At this point, the user can change the default value or leave it as is, and then click on the OK button to store the entry in the variable named userEntry. Or, the user can click on the Cancel button to cancel the entry, which returns a null value.

The third method in the table doesn't require any parameters. It is the print method. When it is executed, it issues a request to the browser for printing the current web page. Then, the browser starts its print function, usually by displaying a dialog box that lets the user set some print options.

To access a property of an object, you use a similar syntax. However, you code the property name after the dot operator as illustrated by the second syntax summary. Unlike methods, properties don't require parameters in parentheses. This is illustrated by the statement that follows the syntax. This statement uses the alert method of the window object to display the location property of the window object.

As you progress through this book, you'll learn how to use the methods and properties of many objects.

Common methods of the window object

Method	Description
`alert(string)`	Displays a dialog box that contains the string that's passed to it by the parameter along with an OK button.
`prompt(string,default)`	Displays a dialog box that contains the string in the first parameter, the default value in the second parameter, an OK button, and a Cancel button. If the user enters a value and clicks OK, that value is returned as a string. If the user clicks Cancel, null is returned to indicate that the value is unknown.
`print()`	Issues a print request for the current web page.

One property of the window object

Property	Description
`location`	The URL of the current web page.

The syntax for calling a method of an object

```
objectName.methodName(parameters)
```

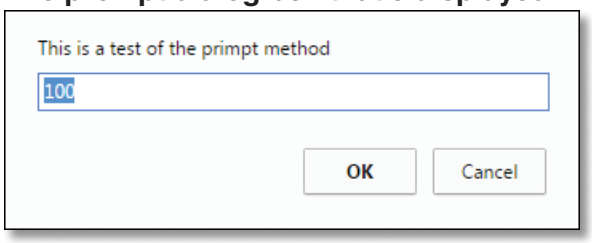
A statement that calls the alert method of the window object

```
window.alert("This is a test of the alert method");
```

A statement that calls the prompt method with the object name omitted

```
var userEntry = prompt("This is a test of the prompt method", 100);
```

The prompt dialog box that's displayed



The syntax for accessing a property of an object

```
objectName.propertyName
```

A statement that displays the location property of the window object

```
alert(window.location);        // Displays the URL of the current page
```

Description

- An *object* has *methods* that perform functions that are related to the object as well as *properties* that represent the data or attributes that are associated with the object.
- When you *call* a method, you may need to pass one or more *parameters* to it by coding them within the parentheses after the method name, separated by commas.
- The *window object* is the *global object* for JavaScript, and JavaScript lets you omit the object name and *dot operator* (period) when referring to the window object.

Figure 2-6 How to use objects, methods, and properties

How to use the write and writeln methods of the document object

Figure 2-7 shows how to use the write and writeln methods of the *document object*. These methods write their data into the body of the document so it's displayed in the browser window.

As the table shows, the only difference between these methods is that the writeln method ends with a new line character. However, the new line character is ignored by the browser unless it's coded in a pre element. As a result, there's usually no difference in the way these methods work.

This is illustrated by the examples in this figure. Note that you can include HTML tags within the parentheses of these methods. For instance, the first write method writes the heading at the top of the page because h1 tags were coded around the text. Similarly, the fourth method in that group writes a
 tag into the document, which is the HTML tag that moves to the next line.

These examples also show that the writeln method doesn't skip to the next line unless it is coded within a pre element in the HTML. For instance, the first two writeln methods in the body of the document display the data with no skipping to a new line. However, when pre tags are added before and after the output, the writeln method does skip to a new line.

Please note that you wouldn't normally use these methods in the head section or an external file to write HTML in the body of the document. Instead, you would code these statements within elements in the body.

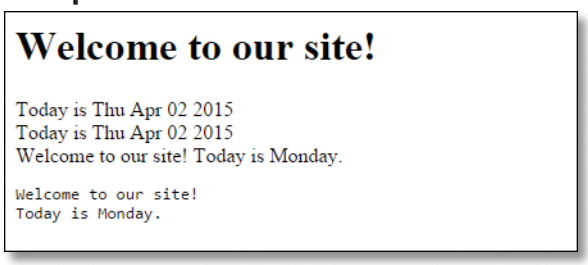
Two methods of the document object

Method	Description
write(*string*)	Writes the string that's passed to it into the document.
writeln(*string*)	Writes the string that's passed to it into the document ending with a new line character (see figure 2-11). However, the new line character isn't recognized by HTML except within the HTML pre element.

Examples of write and writeln methods

```
<head>
    <title>Write Testing</title>
    <script>
        var today = new Date();
        document.write("<h1>Welcome to our site!</h1>");
        document.write("Today is ");
        document.write(today.toDateString());
        document.write("<br>");
        document.writeln("Today is ");
        document.writeln(today.toDateString());
        document.write("<br>");
    </script>
</head>
<body>
    <script>
        document.writeln("Welcome to our site!");
        document.writeln("Today is Monday.");
    </script>
    <script>
        document.writeln("<pre>Welcome to our site!");
        document.writeln("Today is Monday.</pre>");
    </script>
</body>
```

The output in a browser



Description

- The *document object* is the object that lets you work with the Document Object Model (DOM) that represents all of the HTML elements of the page.
- The write and writeln methods are normally used in the body of a document.
- The writeln method doesn't skip to the next line unless it is coded within a pre element. However, you can code
 tags within the output to provide for line spacing.

Figure 2-7 How to use the write and writeln methods of the document object

How to work with JavaScript data

When you develop JavaScript applications, you frequently work with data, especially the data that users enter into the controls of a form. In the topics that follow, you'll learn how to work with the three types of JavaScript data.

The primitive data types

JavaScript provides for three *primitive data types*. The *number data type* is used to represent numerical data. The *string data type* is used to store character data. And the *Boolean data type* is used to store true and false values. This is summarized in figure 2-8.

The number data type can be used to represent either integers or decimal values. *Integers* are whole numbers, and *decimal values* are numbers that can have one or more decimal digits. The value of either data type can be coded with a preceding plus or minus sign. If the sign is omitted, the value is treated as a positive value. A decimal value can also include a decimal point and one or more digits to the right of the decimal point.

As the last example of the number types shows, you can also include an exponent when you code a decimal value. If you aren't familiar with this notation, you probably won't need to use it because you won't be working with very large or very small numbers. On the other hand, if you're familiar with scientific notation, you already know that this exponent indicates how many places the decimal point should be moved to the left or right. Numbers that use this notation are called *floating-point numbers.*

To represent string data, you code the *string* within single or double quotation marks (quotes). Note, however, that you must close the string with the same type of quotation mark that you used to start it. If you code two quotation marks in a row without even a space between them, the result is called an *empty string*, which can be used to represent a string with no data in it.

To represent Boolean data, you code either the word *true* or *false* with no quotation marks. This data type can be used to represent one of two states.

Examples of number values

```
15              // an integer
-21             // a negative integer
21.5            // a decimal value
-124.82         // a negative decimal value
-3.7e-9         // floating-point notation for -0.0000000037
```

Examples of string values

```
"JavaScript"    // a string with double quotes
'String Data'   // a string with single quotes
""              // an empty string
```

The two Boolean values

```
true            // equivalent to true, yes, or on
false           // equivalent to false, no, or off
```

The number data type

- The *number data type* is used to represent an integer or a decimal value that can start with a positive or negative sign.
- An *integer* is a whole number. A *decimal value* can have one or more decimal positions to the right of the decimal point.
- If a result is stored in a number data type that is larger or smaller than the data type can store, it will be stored as the value Infinity or –Infinity.

What you need to know about floating-point numbers

- In JavaScript, decimal values are stored as *floating-point numbers*. In that format, a number consists of a positive or negative sign, one or more significant digits, an optional decimal point, optional decimal digits, and an optional exponent.
- Unless you're developing an application that requires the use of very large or very small numbers, you won't have to use floating-point notation to express numbers. If you need to use this notation, however, it is illustrated by the last example of number values above.

The string data type

- The *string data type* represents character (*string*) data. A string is surrounded by double quotes or single quotes. The string must start and end with the same type of quotation mark.
- An *empty string* is a string that contains no characters. It is entered by typing two quotation marks with nothing between them.

The Boolean data type

- The *Boolean data type* is used to represent a *Boolean value*. A Boolean value can be used to represent data that has two possible states: true or false.

Figure 2-8 The primitive data types

How to code numeric expressions

A *numeric expression* can be as simple as a single value or it can be a series of operations that result in a single value. In figure 2-9, you can see the operators for coding numeric expressions. If you've programmed in another language, these are probably similar to what you've been using. In particular, the first four *arithmetic operators* are common to most programming languages.

Most modern languages also have a *modulus operator* that calculates the remainder when the left value is divided by the right value. In the example for this operator, 13 % 4 means the remainder of 13 / 4. Then, since 13 / 4 is 3 with a remainder of 1, 1 is the result of the expression.

In contrast to the first five operators in this figure, the increment and decrement operators add or subtract one from a variable. To complicate matters, though, these operators can be coded before or after a variable name, and that can affect the result. To avoid confusion, then, it's best to only code these operators after the variable names and only in simple expressions like the one that you'll see in the next figure.

When an expression includes two or more operators, the *order of precedence* determines which operators are applied first. This order is summarized in the table in this figure. For instance, all multiplication and division operations are done from left to right before any addition and subtraction operations are done.

To override this order, though, you can use parentheses. Then, the expressions in the innermost sets of parentheses are done first, followed by the expressions in the next sets of parentheses, and so on. This is typical of all programming languages, as well as basic algebra, and the examples in this figure show how this works.

Common arithmetic operators

Operator	Description	Example	Result
+	Addition	5 + 7	12
-	Subtraction	5 - 12	-7
*	Multiplication	6 * 7	42
/	Division	13 / 4	3.25
%	Modulus	13 % 4	1
++	Increment	counter++	adds 1 to counter
--	Decrement	counter--	subtracts 1 from counter

The order of precedence for arithmetic expressions

Order	Operators	Direction	Description
1	++	Left to right	Increment operator
2	--	Left to right	Decrement operator
3	* / %	Left to right	Multiplication, division, modulus
4	+ -	Left to right	Addition, subtraction

Examples of precedence and the use of parentheses

```
3 + 4 * 5       // Result is 23 since the multiplication is done first
(3 + 4) * 5     // Result is 35 since the addition is done first

13 % 4 + 9      // Result is 10 since the modulus is done first
13 % (4 + 9)    // Result is 0  since the addition is done first

100 + 100 * 2   // Result is 300 since the multiplication is done first
100 + (100 * 2) // Result is still 300 since the multiplication is first
```

Description

- To code a *numeric expression*, you can use the *arithmetic operators* to operate on two or more values.
- The *modulus operator* returns the remainder of a division operation.
- An arithmetic expression is evaluated based on the *order of precedence* of the operators.
- To override the order of precedence, you can use parentheses.
- Because the use of increment and decrement operators can be confusing, it's best to only use these operators in expressions that consist of just a variable name followed by the operator, as shown in the next figure.

Figure 2-9 How to code numeric expressions

How to work with numeric variables

A *variable* stores a value that can change as the program executes. When you code a JavaScript application, you frequently declare variables and assign values to them. Figure 2-10 shows how to do both of these tasks with numeric variables.

To *declare* a numeric variable in JavaScript, code the *var* (for variable) keyword followed by the identifier (or name) that you want to use for the variable. To declare more than one variable in a single statement, code *var* followed by the variable names separated by commas. This is illustrated by the first group of examples in this figure.

To assign a value to a variable, you code an *assignment statement*. This type of statement consists of a variable name, an *assignment operator* like =, and an expression. Here, the expression can be a *numeric literal* like 74.95, a variable name like subtotal, or an arithmetic expression. When the equals sign is the operator, the value of the expression on the right of the equals sign is stored in the variable on the left and replaces any previous value in the variable. The use of this operator is illustrated by the second group of examples.

The second operator in the table in this figure is the += operator, which is a *compound assignment operator*. It modifies the variable on the left of the operator by adding the value of the expression on the right to the value of the variable on the left. When you use this operator, the variable must already exist and have a value assigned to it. The use of this operator is illustrated by the third group of examples.

The fourth group of examples shows three different ways to increment a variable by adding one to it. As you will see throughout this book, this is a common JavaScript requirement. In this group, the first statement assigns a value of 1 to a variable named counter.

Then, the second statement in this group uses an arithmetic expression to add 1 to the value of the counter, which shows that a variable name can be used on both sides of the equals sign. The third statement adds one to the counter by using the += operator.

The last statement in this group uses the increment operator shown in the previous figure to add one to the counter. This illustrates the best way to use increment and decrement operators. Here, the numeric expression consists only of a variable name followed by the increment operator, and it doesn't include an assignment operator.

The last group of examples illustrates a potential problem that you should be aware of. Because decimal values are stored internally as floating-point numbers, the results of arithmetic operations aren't always precise. In this example, the salesTax result, which should be 7.495, is 7.495000000000001. Although this result is extremely close to 7.495, it isn't equal to 7.495, which could lead to a programming problem if you expect a comparison of the two values to be equal. The solution is to round the result, which you'll learn how to do in the next chapter.

The most useful assignment operators

Operator	Description
=	Assigns the result of the expression to the variable.
+=	Adds the result of the expression to the variable.

How to declare numeric variables without assigning values to them

```
var subtotal;                           // declares one variable
var investment, interestRate, years;    // declares three variables
```

How to declare variables and assign values to them

```
var subtotal = 74.00;           // subtotal = 74.00
var salesTax = subtotal * .1;   // salesTax = 7.4
```

How to code compound assignment statements

```
var subtotal = 74.95;           // subtotal = 74.95
subtotal += 20.00;              // subtotal = 94.95
```

Three ways to increment a variable named counter by 1

```
var counter = 1;                // counter = 1
counter = counter + 1;          // counter now = 2
counter += 1;                   // counter now = 3
counter++;                      // counter now = 4
```

A floating-point result that isn't precise

```
var subtotal = 74.95;           // subtotal = 74.95
var salesTax = subtotal * .1;   // salesTax = 7.495000000000001
```

Description

- A *variable* stores a value that can change as the program executes.
- To *declare* a variable, code the keyword *var* and a variable name. To declare more than one variable in a single statement, code *var* and the variable names separated by commas.
- To assign a value to a variable, you use an *assignment statement* that consists of the variable name, an *assignment operator*, and an expression. When appropriate, you can declare a variable and assign a value to it in a single statement.
- Within an expression, a *numeric literal* is a valid integer or decimal number that isn't enclosed in quotation marks.
- If you use a plus sign in an expression and both values are numbers, JavaScript adds them. If both values are strings, JavaScript concatenates them as shown in the next figure. And if one value is a number and one is a string, JavaScript converts the number to a string and concatenates.
- When you do some types of arithmetic operations with decimal values, the results aren't always precise, although they are extremely close. That's because decimal values are stored internally as floating-point numbers. The only problem with this is that an equality comparison may not return true.

Figure 2-10 How to work with numeric variables

How to work with string and Boolean variables

To declare a string or Boolean variable, you use techniques that are similar to those you use for declaring a numeric variable. The main difference is that a *string literal* is a value enclosed in quotation marks, while a numeric literal isn't. Besides that, the + sign is treated as a *concatenation operator* when working with strings. This means that one string is added to the end of another string.

This is illustrated by the first two groups of examples in figure 2-11. In the second group, the first statement assigns string literals to the variables named firstName and lastName. Then, the next statement concatenates lastName, a string literal that consists of a comma and a space, and firstName. The result of this concatenation is

```
Hopper, Grace
```

which is stored in a new variable named fullName.

In the third group of examples, you can see how the += operator can be used to get the same results. When the expressions that you're working with are strings, this operator does a simple concatenation.

In the fourth group of examples, though, you can see what happens if the += operator is used with a string and a numeric value. In that case, the number is converted to a string and then the strings are concatenated.

The fifth group of examples shows how you can use *escape sequences* in a string. Three of the many escape sequences that you can use are summarized in the second table in this figure. These sequences let you put characters in a string that you can't put in just by pressing the appropriate key on the keyboard. For instance, the \n escape sequence is equivalent to pressing the Enter key in the middle of a string. And the \' sequence is equivalent to pressing the key for a single quotation mark.

Escape sequences are needed so the JavaScript engine can interpret code correctly. For instance, since single and double quotations marks are used to identify strings in JavaScript statements, coding them within the strings would cause syntax errors. But when the quotation marks are preceded by escape characters, the JavaScript engine can interpret them correctly.

The last example in this figure shows how to create and assign values to Boolean variables. Here, a variable named isValid is created and a value of false is assigned to it.

The concatenation operator for strings

Operator	Example	Result
+	`"Grace " + "Hopper"`	`"Grace Hopper"`
	`"Months: " + 120`	`"Months: 120"`

Escape sequences that can be used in strings

Operator	Description
`\n`	Starts a new line in a string.
`\"`	Puts a double quotation mark in a string.
`\'`	Puts a single quotation mark in a string.

How to declare string variables without assigning values to them

```
var zipCode;                                    // declares one variable
var lastName, state, zipCode;                   // declares three variables
```

How to declare string variables and assign values to them

```
var firstName = "Grace", lastName = "Hopper"; // assigns two string values
var fullName = lastName + ", " + firstName;   // fullName is "Hopper, Grace"
```

How to code compound assignment statements with string data

```
var firstName = "Grace", lastName = "Hopper";
var fullName = lastName;                // fullName is "Hopper"
fullName += ", ";                       // fullName is "Hopper, "
fullName += firstName;                  // fullName is "Hopper, Grace"
```

How to code compound assignment statements with mixed data

```
var months = 120;
message = "Months: ";
message += months;                      // message is "Months: 120"
```

How escape sequences can be used in a string

```
var message = "A valid variable name\ncannot start with a number.";
var message = "This isn\'t the right way to do this.";
```

How to declare Boolean variables and assign values to them

```
var isValid = false;                    // Boolean value is false
```

Description

- To assign values to string variables, you can use the + and += operators, just as you use them with numeric variables.
- To *concatenate* two or more strings, you can use the + operator.
- Within an expression, a *string literal* is enclosed in quotation marks.
- *Escape sequences* can be used to insert special characters within a string like a return character that starts a new line or a quotation mark.
- If you use a plus sign in an expression and both values are strings, JavaScript concatenates them. But if one value is a number and one is a string, JavaScript converts the number to a string and concatenates the strings.

Figure 2-11 How to work with string and Boolean variables

How to use the parseInt and parseFloat methods

The parseInt and parseFloat methods are used to convert strings to numbers. The parseInt method converts a string to an integer, and the parseFloat method converts a string to a decimal value. If the string can't be converted to a number, the value *NaN* is returned. NaN means "Not a Number".

These methods are needed because the values that are returned by the prompt method and the values that the user enters into text boxes are treated as strings. The is illustrated by the first group of examples in figure 2-12. For this group, assume that the default value in the prompt method isn't changed by the user. As a result, the first statement in this group stores 12345.6789 as a string in a variable named entryA. Then, the third statement in this group converts the string to an integer value of 12345.

Note that the object name isn't coded before the method name in these examples. That's okay because window is the global object of JavaScript. Note too that the parseInt method doesn't round the value. It just removes, or truncates, any decimal portion of the string value.

The last four statements in the first group of examples show what happens when the parseInt or parseFloat method is used to convert a value that isn't a number. In that case, both of these methods return the value NaN.

Note, however, that these methods can convert values that consist of one or more numeric characters followed by one or more nonnumeric characters. In that case, these methods simply drop the nonnumeric characters. For example, if a string contains the value 72.5%, the parseFloat method will convert it to a decimal value of 72.5.

The second group of examples in this figure shows how to get the same results by coding the parse methods as the parameters of the alert methods. For instance, the third statement in this group uses the parseInt method as the parameter of the alert method.

Note in the first set of examples that the values of the entryA, entryB, and entryC variables are all changed by the parse methods. For instance, entryA becomes the number 12345 and entryC becomes NaN. In contrast, the entries aren't changed by the statements in the second set of examples. That's because the parsed values aren't assigned to the variables; they're just displayed by the alert statements.

The parseInt and parseFloat methods of the window object

Method	Description
parseInt(*string*)	Converts the string that's passed to it to an integer data type and returns that value. If it can't convert the string to an integer, it returns NaN.
parseFloat(*string*)	Converts the string that's passed to it to a decimal data type and returns that value. If it can't convert the string to a decimal value, it returns NaN.

Examples that use the parseInt and parseFloat methods

```
var entryA = prompt("Enter any value", 12345.6789);
alert(entryA);                              // displays 12345.6789
entryA = parseInt(entryA);
alert(entryA);                              // displays 12345

var entryB = prompt("Enter any value", 12345.6789);
alert(entryB);                              // displays 12345.6789
entryB = parseFloat(entryB);
alert(entryB);                              // displays 12345.6789

var entryC = prompt("Enter any value", "Hello");
alert(entryC);                              // displays Hello
entryC = parseInt(entryC);
alert(entryC);                              // displays NaN
```

The same examples with the parse methods embedded in the alert method

```
var entryA = prompt("Enter any value", 12345.6789);
alert(entryA);                              // displays 12345.6789
alert(parseInt(entryA));                    // displays 12345

var entryB = prompt("Enter any value", 12345.6789);
alert(entryB);                              // displays 12345.6789
alert(parseFloat(entryB));                  // displays 12345.6789

var entryC = prompt("Enter any value", "Hello");
alert(entryC);                              // displays Hello
alert(parseInt(entryC));                    // displays NaN
```

Description

- The window object provides parseInt and parseFloat methods that let you convert string values to integer or decimal numbers.
- When you use the prompt method or a text box to get numeric data that you're going to use in calculations, you need to use either the parseInt or parseFloat method to convert the string data to numeric data.
- *NaN* is a value that means "Not a Number". It is returned by the parseInt and parseFloat methods when the value that's being parsed isn't a number.
- When working with methods, you can embed one method in the parameter of another. In the second group of examples above, the parse methods are coded as the parameters for the alert methods.

Figure 2-12 How to use the parseInt and parseFloat methods

Two illustrative applications

This chapter ends by presenting two applications that illustrate the skills that you've just learned. These aren't realistic applications because they get the user entries from prompt statements instead of from controls on a form. However, these applications will get you started with JavaScript.

The Miles Per Gallon application

Figure 2-13 presents a simple application that issues two prompt statements that let the user enter the number of miles driven and the number of gallons of gasoline used. Then, the application calculates miles per gallon and issues an alert statement to display the result in a third dialog box.

Because this application uses prompt methods to get the input and an alert method to display the output, the HTML for this application is trivial. It contains one h1 element that is displayed after the JavaScript finishes executing.

In the JavaScript, you can see how the user's entries are stored in variables named miles and gallons and then parsed into decimal values. After that, miles is divided by gallons, and the result is saved in a variable named mpg. Then, the value in that variable is parsed into an integer so any decimal places are removed. Last, an alert statement displays the result that's shown in the third dialog box. When the user clicks on the OK button in that box, the JavaScript ends and the page with its one heading is displayed in the browser.

Can you guess what will happen if the user enters invalid data in one of the prompt dialog boxes? Then, the parse method will return NaN instead of a number, and the calculation won't work. Instead, the last alert statement will display:

```
Miles per gallon = NaN
```

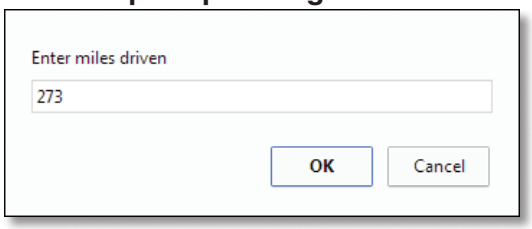
Unlike other languages, though, the JavaScript will run to completion instead of crashing when the calculation can't be done, so the web page will be displayed.

When you look at the second and third dialog boxes in this figure, you can see that they contain checkboxes with messages that say "Prevent this page from creating additional dialogs." After an application has popped up one dialog box, most modern browsers will include this message on any more dialog boxes. This is a security feature designed to prevent endless loops of dialog boxes from locking the browser, and this can't be disabled.

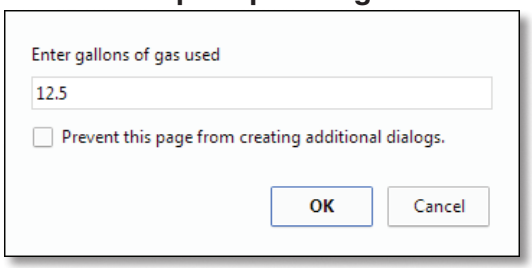
Incidentally, the dialog boxes in this figure are the ones for a Chrome browser. When you use other browsers, the dialog boxes will work the same but have slightly different appearances.

The dialog boxes for the Calculate MPG application

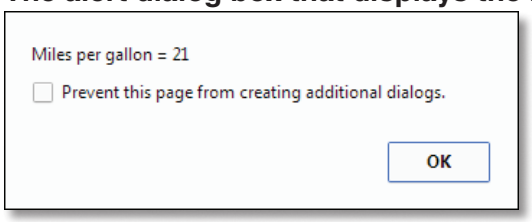
The first prompt dialog box



The second prompt dialog box



The alert dialog box that displays the result



The HTML and JavaScript for the application

```
<html>
<head>
    <title>The Calculate MPG Application</title>
    <script>
        var miles = prompt("Enter miles driven");
        miles = parseFloat(miles);
        var gallons = prompt("Enter gallons of gas used");
        gallons = parseFloat(gallons);
        var mpg = miles/gallons;
        mpg = parseInt(mpg);
        alert("Miles per gallon = " + mpg);
    </script>
</head>
<body>
    <!-- Will show after the JavaScript has run -->
    <h1>Thanks for using the Miles Per Gallon application!</h1>
</body>
</html>
```

Description

- If an application pops up more than one dialog box, most browsers will give you the option to prevent the page from creating any more. This is a built in security measure that can't be disabled.

Figure 2-13 The Miles Per Gallon application

The Test Scores application

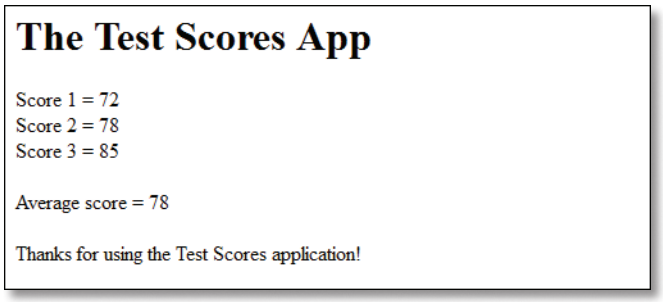
Figure 2-14 presents a simple application that uses prompt methods to let the user enter three test scores. After the third one is entered, this application calculates the average test score. That ends the JavaScript that's embedded in the head element of the HTML document.

Then, the JavaScript that's coded in the body element is executed. It uses one write method to write an h1 element at the top of the page. Then, it uses another write method to write the three test scores and the average score into the body of the document so it is displayed in the browser window. This shows that the variables that were created by the JavaScript in the head element are available to the JavaScript in the body element.

If you take another look at the JavaScript in the head element, you can see that it starts by declaring the variables that are needed to get and process the entries. The entry and average variables are declared but not assigned a value. The entry variable will be used to receive the user entries, and the average variable will receive the calculated average of the scores entered by the user.

In contrast, the total variable is assigned a starting value of zero. Then, each entry is added to this total value, and it is divided by 3 to calculate the average score. In addition, the score1, score2, and score3 variables are declared and assigned values after each score has been entered and parsed.

The results displayed in the browser



The HTML and JavaScript for the application

```
<html>
<head>
    <title>Average Test Scores</title>
    <script>
        var entry;
        var average;
        var total = 0;

        //get 3 scores from user and add them together
        entry = prompt("Enter test score");
        entry = parseInt(entry);
        var score1 = entry;
        total = total + score1;

        entry = prompt("Enter test score");
        entry = parseInt(entry);
        var score2 = entry;
        total = total + score2;

        entry = prompt("Enter test score");
        entry = parseInt(entry);
        var score3 = entry;
        total = total + score3;

        //calculate the average
        average = parseInt(total/3);
    </script>
</head>
<body>
    <script>
        document.write("<h1>The Test Scores App</h1>");
        document.write("Score 1 = " + score1 + "<br>" +
            "Score 2 = " + score2 + "<br>" +
            "Score 3 = " + score3 + "<br><br>" +
            "Average score = " + average + "<br><br>");
    </script>
    Thanks for using the Test Scores application!
</body>
</html>
```

Figure 2-14 The Test Scores application

Perspective

If you have programming experience, you can now see that JavaScript syntax is similar to other languages like Java and C#. As a result, you should have breezed through this chapter. You may also want to skip the exercises.

On the other hand, if you're new to programming and you understand all of the code in both of the applications in this chapter, you're off to a good start. Otherwise, you need to study the applications until you understand every line of code in each application. You should also do the exercises that follow.

Terms

external JavaScript file
embedded JavaScript
JavaScript statement
syntax
whitespace
identifier
reserved word
keyword
camel casing
comment
block comment
in-line comment
comment out
uncomment
object
method
property
call a method
dot operator (dot)
parameter
window object
global object
document object
primitive data type
number data type
integer
decimal value
floating-point number
string data type
string
empty string
Boolean data type
Boolean value
numeric expression
arithmetic operator
modulus operator
order of precedence
variable
declare a variable
assignment statement
assignment operator
compound assignment operator
numeric literal
concatenate
concatenation operator
string literal
escape sequence
NaN

Summary

- The JavaScript for an HTML document page is commonly coded in an *external JavaScript file* that's identified by a script element. However, the JavaScript can also be *embedded* in a script element in the head or body of a document.
- A JavaScript *statement* has a *syntax* that's similar to Java's. Its *identifiers* are case-sensitive and usually coded with either *camel casing* or underscore notation. Its co*mments* can be block or in-line.

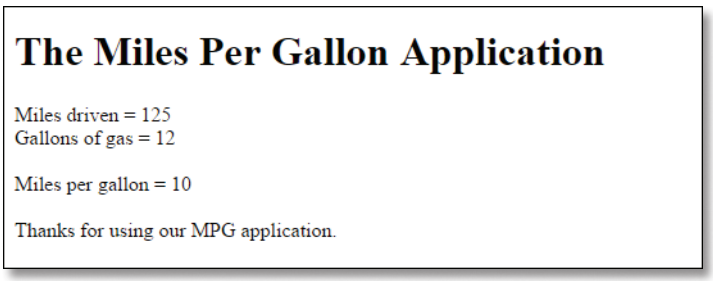
- JavaScript provides many *objects* that provide *methods* and *properties* that you can *call* or refer to in your applications. Since the *window object* is the *global object* for JavaScript, you can omit it when referring to its methods or properties.
- The *document object* provides some commonly used methods like the write and writeln methods.
- JavaScript provides three *primitive data types*. The *number data type* provides for both *integers* and *decimal values*. The *string data type* provides for character (*string*) data. And the *Boolean data type* provides for true and false values.
- To assign a value to a *variable*, you use an *assignment operator*.
- When you assign a value to a number *variable*, you can use *numeric expressions* that include *arithmetic operators*, variable names, and *numeric literals*.
- When you assign a value to a string variable, you can use *string expressions* that include *concatenation operators*, variable names, and *string literals*. Within a string literal, you can use *escape sequences* to provide special characters.

Before you do the exercises for this book…

If you haven't already done so, you should install the Chrome browser and install the downloads for this book as described in appendix A.

Exercise 2-1 Modify the Miles Per Gallon application

In this application, you'll modify the code for the MPG application so the results are displayed in the browser window instead of an alert dialog box. The browser window should look something like this:



1. Open your text editor or IDE. Then, open this HTML file:
   `c:\javascript\exercises\ch02\mpg.html`
2. Run the application with valid entries, and note the result. Then, run it with invalid entries like zeros or spaces, and note the result.
3. Modify this application so the result is displayed in the browser instead of an alert statement. This result should include the user's entries for miles and gallons as shown above.

4. If you have any problems when you test your exercises, please use Chrome's developer tools as shown in figure 1-16 of the last chapter.

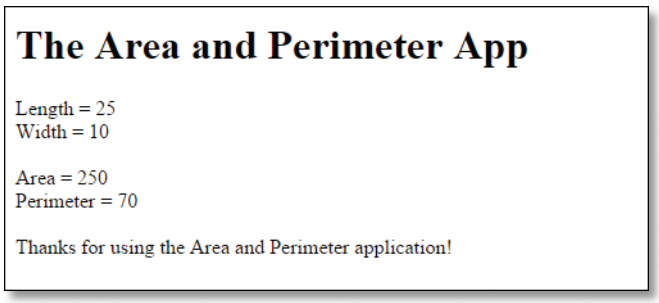
Exercise 2-2 Modify the Test Scores application

In this exercise, you'll modify the Test Scores application so it works the same but uses less code. The output of the application will still be displayed in the browser as it is in figure 2-14.

1. Open your text editor or IDE. Then, open this HTML file:
   ```
   c:\javascript\exercises\ch02\scores.html
   ```
2. Run the application with valid entries, and note the result.
3. Modify this application so the variable declarations for the three scores parse the entries before storing their values. In other words, the two lines of code for each variable
   ```
   entry = parseInt(entry);
   var score1 = entry;
   ```
   should be combined into one.
4. Change the statements that accumulate the score total so they use the += assignment operator shown in figure 2-10 instead of the equals operator.

Exercise 2-3 Create a simple application

Copying and modifying an existing application is often a good way to start a new application. So in this exercise, you'll modify the Miles Per Gallon application so it gets the length and width of a rectangle from the user, calculates the area and the perimeter of the rectangle, and displays the results in the browser like this:



1. Open your text editor or IDE. Then, open this HTML file:
   ```
   c:\javascript\exercises\ch02\rectangle.html
   ```
   As you can see, this file contains the code for the Miles Per Gallon application.
2. Modify the code for this application so it works for the new application. (The area of a rectangle is just length times width. The perimeter is 2 times length plus 2 times width.)

How to build your JavaScript skills


Mike Murach, Publisher

The easiest way is to let **Murach's JavaScript (2nd Edition)** be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. You can use it to:

- Teach yourself how to use JavaScript to create websites that deliver the dynamic user interfaces and fast response times that today's users expect
- Master the JavaScript essentials that you need to use jQuery
- Pick up a new skill whenever you want or need to by focusing on material that's new to you
- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a JavaScript application
- Loan to your colleagues who are always asking you questions about JavaScript programming

To get your copy, you can order online at **www.murach.com** or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you've ever tried, or send it back within 60 days for a full refund…no questions asked.

Thanks for your interest in Murach books!